



OPEN
DAYLIGHT

OpenDaylight Developer Guide

master (March 4, 2015)



OPEN
DAYLIGHT

OpenDaylight Developer Guide

Opendaylight Community

master (2015-03-04)

Copyright © 2014 Linux Foundation All rights reserved.

This guide describes how to develop using OpenDaylight.

This program and the accompanying materials are made available under the terms of the Eclipse Public License v1.0 which accompanies this distribution, and is available at <http://www.eclipse.org/legal/epl-v10.html>

Table of Contents

| | |
|--|----|
| I. Overview | 1 |
| 1. Getting started with Git and Gerrit | 3 |
| Overview of Git and Gerrit | 3 |
| Setting up a Gerrit account | 3 |
| Generating SSH keys for your system | 6 |
| Registering your SSH key with Gerrit | 6 |
| 2. Pulling and Pushing the Code from the CLI | 9 |
| Pulling code via Git CLI | 9 |
| Setting up Gerrit Change-id Commit Message Hook | 9 |
| Building the code | 10 |
| Runing OpenDaylight from local build | 10 |
| Commit the code using Git CLI | 11 |
| Pulling the Code changes via Git CLI | 12 |
| Pushing the Code via Git CLI | 12 |
| II. Project-Specific Development Guides | 15 |
| 3. Authentication Service | 20 |
| Authenthentication data model | 20 |
| How the ODL Authentication Service works | 21 |
| Configuring Authentication service | 22 |
| How federated authentication is set up | 23 |
| Mapping users to roles and domains | 23 |
| Actors in ODL Authentication Service | 26 |
| Sub-components of ODL Authentication Service | 26 |
| 4. BGP LS PCEP | 28 |
| BGPCEP Overview | 28 |
| Implementing an Extension to PCEP | 29 |
| Update Configuration | 29 |
| Implementing an Extension to BGP | 30 |
| Updating Configuration | 30 |
| Vendor Information TLV | 33 |
| Vendor Information Object | 36 |
| 5. Controller | 40 |
| OpenDaylight Controller: MD-SAL Developers' Guide | 41 |
| API types | 41 |
| Basic YANG concepts and their rendition in APIs | 41 |
| MD-SAL: Plugin types | 45 |
| Protocol library | 46 |
| MD-SAL: Southbound plugin development guide | 46 |
| Definition of YANG models | 47 |
| RPCs | 47 |
| Augmentations | 49 |
| Best practices | 49 |
| Implementation | 49 |
| Notifications | 50 |
| Best practices | 50 |
| OpenDaylight Controller: MD-SAL FAQs | 51 |
| OpenDaylight Controller Configuration: Java Code Generator | 58 |
| Service interfaces generating | 58 |

| | |
|--|-----|
| Runtime beans generating | 59 |
| OpenDaylight Controller MD-SAL: Restconf | 59 |
| Mount point | 60 |
| Something practical | 64 |
| OpenDaylight Controller: Configuration | 67 |
| APIs and SPIs | 69 |
| OpenDaylight Controller configuration: Initial | 71 |
| Initial configuration for controller | 71 |
| OpenDaylight Controller configuration: config.ini | 72 |
| OpenDaylight Controller: Configuration Persister | 72 |
| Current configuration for controller distribution | 73 |
| Adding custom initial configuration | 79 |
| Persister Notification Handler | 83 |
| MD-SAL architecture: Clustering Notifications | 86 |
| MD-SAL Architecture: DOM | 87 |
| MD-SAL: Infinispan Data Store | 88 |
| State of the POC | 91 |
| Infinispan-related learnings | 92 |
| Datastore-related learnings | 92 |
| No clarity on the closing of Read-Only transactions | 92 |
| OpenDaylight Controller configuration: FAQs | 95 |
| OpenDaylight Controller configuration: Component map | 95 |
| OpenDaylight Controller: Netconf component map | 97 |
| OpenDaylight Controller Configuration: Examples sample project | 97 |
| OpenDaylight Controller: Configuration examples user guide | 109 |
| OpenDaylight Controller Configuration: Logback Examples | 118 |
| OpenDaylight Controller: Configuration Logback.xml | 125 |
| Configuration example of thread pools using yangcli-pro | 126 |
| Configuration example of thread pools using telnet | 126 |
| Connecting to plaintext TCP socket | 126 |
| Configuring threadfactory | 127 |
| Configuring fixed threadpool | 130 |
| OpenDaylight Controller MD-SAL: Model reference | 131 |
| 6. Defense4all | 132 |
| Defense4All Design | 132 |
| Defense4All in an ODL Environment | 133 |
| Framework View | 134 |
| Application View | 136 |
| ODL Reps View | 138 |
| Basic Control Flow | 141 |
| Configurations and Setup Flow | 141 |
| Attack Detection Flow | 142 |
| Attack Mitigation Flow | 142 |
| Continuity | 143 |
| 7. DLUX | 146 |
| Setup and Run | 146 |
| DLUX Modules | 148 |
| Yang Utils | 151 |
| 8. Group-Based Policy | 153 |
| Group-Based Policy Architecture Overview | 154 |
| Policy Model | 155 |

| | |
|---|-----|
| State Repositories | 169 |
| Renderers | 170 |
| 9. L2Switch | 189 |
| Checking out the L2Switch project | 189 |
| Testing your changes to the L2Switch project | 189 |
| Architecture of the L2Switch project | 190 |
| Developer's Guide for Packet Dispatcher | 191 |
| Developer's Guide for Loop Remover | 191 |
| Developer's Guide for Arp Handler | 193 |
| Developer's Guide for Address Tracker | 195 |
| Developer's Guide for Host Tracker | 197 |
| Developer's Guide for L2Switch Main | 197 |
| 10. LISP Flow Mapping | 200 |
| OpenDaylight Locator/ID Separation Protocol (LISP) Flow Mapping | |
| Overview | 200 |
| LISP Flow Mapping Service | 201 |
| LISP Service Architecture | 201 |
| LISP APIs | 203 |
| LISP Configuration Options | 203 |
| Developer Tutorial | 203 |
| LISP Support | 210 |
| Installing LISP Flow Mapping | 210 |
| 11. ODL-SDNi | 216 |
| 12. OpenFlow Protocol Library | 217 |
| 13. OpenFlow Plugin | 218 |
| OpenFlow Plugin: Sequence diagrams | 219 |
| OpenFlow Plugin:Config subsystem | 223 |
| Message Spy in OF Plugin | 229 |
| OpenFlow Plugin:Mininet | 232 |
| Installation | 232 |
| Usage | 235 |
| Coding tips for OpenFlow Plugin | 235 |
| OpenFlow Plugin: Wiring up notifications | 237 |
| OpenFlow Plugin:Python test scripts | 239 |
| General | 240 |
| ODL Test (odl_crud_tests.py) | 241 |
| Parameters | 242 |
| Stress Test (stress_test.py) | 243 |
| Operational Data Test (oper_data_test.py) | 243 |
| Switch restart (sw_restart_test.py) | 243 |
| OpenFlow Plugin: Robot framework tests | 244 |
| TLS support for OF Plugin | 245 |
| Configuring the ODL OpenFlow plugin | 247 |
| Configuring openvswitch SSL | 247 |
| Configuring a hardware switch with TLS | 248 |
| Open Flow Plugin: Support for extensibility | 249 |
| Overload protection in the OF Plugin | 251 |
| 14. OVSDB Integration | 254 |
| OpenDaylight OVSDB integration | 254 |
| Building and running OVSDB | 257 |
| OVSDB integration design | 260 |

| | |
|--|-----|
| OpenDaylight OVSDb southbound plugin architecture and design | 260 |
| OVSDb southbound plugin | 261 |
| Connection service | 261 |
| Network Configuration Service | 263 |
| OpenDaylight OVSDb Developer Getting Started Video Series | 267 |
| OVSDb integration: New features | 267 |
| 15. Packet Cable MultiMedia (PCMM) | 275 |
| Checking out the Packetcable PCMM project | 275 |
| System Overview | 275 |
| Dependency Map | 276 |
| Packetcable Components | 276 |
| Download and Install | 277 |
| Preparing to Work with the Packetcable PCMM Service | 277 |
| Explore and exercise the PacketCable REST API | 281 |
| RESTCONF API Explorer | 281 |
| Postman | 282 |
| Custom Testsuite | 282 |
| Using Wireshark to Trace PCMM | 282 |
| Debugging and Verifying DQoS Gate (Flows) on the CMTS | 283 |
| Find the Cable Modem | 283 |
| Arris | 285 |
| RESTCONF API for Packetcable PCMM | 285 |
| 16. Plugin for OpenContrail | 289 |
| 17. Service Function Chaining | 290 |
| 18. SNBI Developers' Guide | 291 |
| Defining characteristics of SNBI bootstrapping | 291 |
| SNBI components | 291 |
| How SNBI works | 292 |
| 19. SNMP4SDN | 296 |
| 20. TCP-MD5 | 297 |
| 21. Table Type Patterns | 298 |
| Introduction | 298 |
| Using The REST APIs | 299 |
| Limitations | 309 |
| 22. VTN | 312 |
| Virtual Tenant Network (VTN) | 312 |
| Hacking VTN Coordinator | 326 |
| Hacking VTN Manager | 329 |
| Hacking VTN Manager(Helium) | 330 |
| Openstack Support Developer Guide | 332 |
| 23. YANG Tools | 336 |
| Prerequisites for YANG Tools Project | 336 |
| Pulling code using ssh | 336 |
| Pulling code using https | 336 |
| Building the code | 337 |
| Mapping YANG to Java | 337 |
| Additional Packages | 338 |
| Data Interface | 340 |
| Service Interface | 340 |

List of Figures

| | |
|--|-----|
| 1.1. Signing in to OpenDaylight account | 4 |
| 1.2. Gerrit Account signup/management link | 4 |
| 1.3. Sign-up link for Gerrit account | 4 |
| 1.4. Sign-up with User Name/Password Image | 5 |
| 1.5. Filling out the details | 5 |
| 1.6. Signin in to OpenDaylight repository | 7 |
| 1.7. Settings page for your Gerrit account | 7 |
| 1.8. Adding your SSH key | 7 |
| 2.1. OpenDaylight Main Page | 10 |
| 2.2. Gerrit Code Review Sample | 13 |
| 2.3. Gerrit Code Merge Sample | 14 |
| 5.1. AD-SAL and MD-SAL | 51 |
| 5.2. Plugin development process | 53 |
| 5.3. Flow deleted at controller | 54 |
| 5.4. External app adds flow | 55 |
| 5.5. SAL consumer and producer view | 57 |
| 5.6. Get | 63 |
| 5.7. Put | 64 |
| 5.8. Configuration states | 68 |
| 5.9. Transaction states | 68 |
| 5.10. Persister | 73 |
| 8.1. Group-Based Policy Architecture | 154 |
| 8.2. Policy Model: Contract Selection | 156 |
| 8.3. Policy Model: Clauses and Subject Selection | 157 |
| 8.4. Policy Model: Subject Contents | 158 |
| 8.5. Policy Model: Forwarding | 159 |
| 8.6. GBP OVS Network Topology Example | 172 |
| 8.7. GBP OVS Routing Example | 175 |
| 8.8. GBP OVS Example of Communication With Outside Endpoints | 176 |
| 8.9. GBP OVS Packet Processing Pipeline | 177 |
| 8.10. GBP OVS ARP Optimization | 182 |
| 10.1. Architecture Overview | 201 |
| 10.2. LISP Mapping Service Internal Architecture | 202 |
| 10.3. Gerrit Code Review Sample | 214 |
| 10.4. Gerrit Code Merge Sample | 215 |
| 13.1. Message Lifecycle | 219 |
| 13.2. Handshake Scenario | 220 |
| 13.3. Connection Sequence (Handshake) Flow Diagram | 221 |
| 13.4. Message Order Preservation | 222 |
| 13.5. Add Flow Sequence | 222 |
| 13.6. Generic Notification Sequence | 223 |
| 13.7. Configure Compiler Errors and Warnings | 236 |
| 13.8. Configure Javadoc | 237 |
| 13.9. OF Plugin support for extensibility | 249 |
| 13.10. Overload protection | 252 |
| 14.1. Avoid conflicting project names | 256 |
| 14.2. Connection to OVSDb server | 262 |
| 14.3. Successful connection handling | 263 |

| | |
|--|-----|
| 14.4. End-to-end handling of a Create Bridge request | 265 |
| 14.5. End-to-end handling of a monitor response | 266 |
| 14.6. Sample workflow | 271 |
| 15.1. System Overview | 276 |
| 15.2. Dependency Map | 276 |
| 15.3. Sign in to Dlux UI | 279 |
| 15.4. View and Manage Flows in Dlux | 280 |
| 15.5. View and Manage Nodes in Dlux | 280 |
| 15.6. Add CMTS using RESTCONF Explorer | 281 |
| 15.7. Postman Collection for Packetcable PCMM | 282 |
| 18.1. Communication between the controller and FE | 293 |
| 21.1. Filling in URL, content, Content-Type and basic auth | 306 |
| 21.2. Refreshing basic auth headers | 307 |
| 21.3. PUTting a TTP | 308 |
| 21.4. Retrieving the TTP as json via a GET | 308 |
| 21.5. Retrieving the TTP as xml via a GET | 309 |
| 22.1. VTN Architecture | 313 |
| 22.2. VTN Coordinator Architecture | 314 |
| 22.3. VTN Manager Architecture | 315 |
| 22.4. VTN Coordinator Architecture | 316 |
| 22.5. VTN Transaction Co ordinator (TC) Architecture | 318 |
| 22.6. VTN ODC Driver Architecture | 320 |
| 22.7. VTN UPLL Architecture | 322 |
| 22.8. VTN UPPL Architecture | 324 |
| 22.9. OpenStack Architecture | 332 |

List of Tables

| | |
|--|-----|
| 10.1. Nodes in the tutorial | 203 |
| 13.1. OpenFlow plugin: Component map | 218 |
| 15.1. Table of Bundle and Components | 276 |

Part I. Overview

Table of Contents

| | |
|---|----|
| 1. Getting started with Git and Gerrit | 3 |
| Overview of Git and Gerrit | 3 |
| Setting up a Gerrit account | 3 |
| Generating SSH keys for your system | 6 |
| Registering your SSH key with Gerrit | 6 |
| 2. Pulling and Pushing the Code from the CLI | 9 |
| Pulling code via Git CLI | 9 |
| Setting up Gerrit Change-id Commit Message Hook | 9 |
| Building the code | 10 |
| Runing OpenDaylight from local build | 10 |
| Commit the code using Git CLI | 11 |
| Pulling the Code changes via Git CLI | 12 |
| Pushing the Code via Git CLI | 12 |

1. Getting started with Git and Gerrit

Table of Contents

| | |
|--|---|
| Overview of Git and Gerrit | 3 |
| Setting up a Gerrit account | 3 |
| Generating SSH keys for your system | 6 |
| Registering your SSH key with Gerrit | 6 |

Overview of Git and Gerrit

Git is an opensource distributed version control system (dvcs) written in the C language and originally developed by Linus Torvalds and others to manage the Linux kernel. In Git, there is no central copy of the repository. After you have cloned the repository, you have a functioning copy of the source code with all the branches and tagged releases, in your local repository.

Gerrit is an opensource web-based collaborative code review tool that integrates with Git. It was developed at Google by Shawn Pearce. Gerrit provides a framework for reviewing code commits before they are accepted into the code base. Changes can be uploaded to Gerrit by any user. However, the changes are not made a part of the project until a code review is completed. Gerrit is also a good collaboration tool for storing the conversations that occur around the code commits.

The OpenDaylight source code is hosted in a repository in Git. Developers must use Gerrit to commit code to the OpenDaylight repository.



Note

For more information on Git, see <http://git-scm.com/>. For more information on Gerrit, see <https://code.google.com/p/gerrit/>.

Setting up a Gerrit account

1. Using a Google Chrome or Mozilla Firefox browser, go to <https://git.opendaylight.org/gerrit>

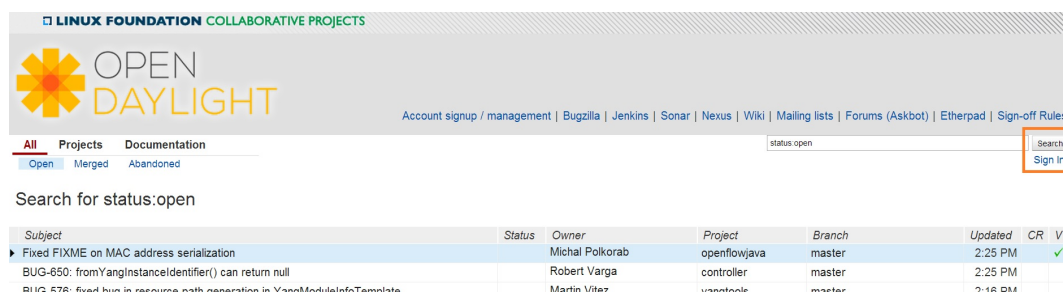
The main page shows existing Gerrit requests. These are patches that have been pushed to the repository and not yet verified, reviewed, and merged.



Note

If you already have an OpenDaylight account, you can click **Sign In** in the top right corner of the page and follow the instructions to enter the OpenDaylight page.

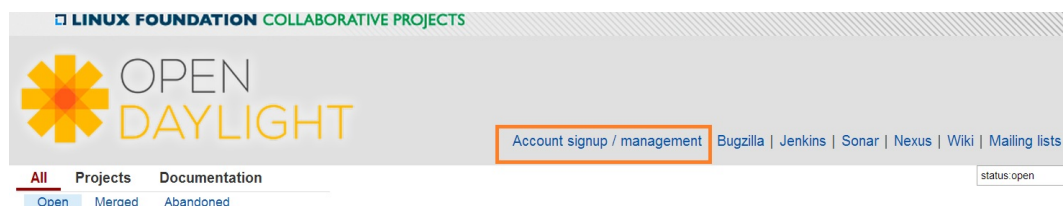
Figure 1.1. Signing in to OpenDaylight account



1. If you do not have an existing OpenDaylight account, click **Account signup/management** on the top bar of the main Gerrit page.

The **WSO2 Identity Server** page is displayed.

Figure 1.2. Gerrit Account signup/management link



1. In the **WSO2 Identity Server** page, click **Sign-up** in the left pane.

There is also an option to authenticate your sign in with OpenID. This option is not described in this document.

Figure 1.3. Sign-up link for Gerrit account



1. Click on the **Sign-up with User Name/Password** image on the right pane to continue to the actual sign-up page.

Figure 1.4. Sign-up with User Name/Password Image

Sign-up with User Name / Password



1. Fill out the details in the account creation form and then click **Submit**.

Figure 1.5. Filling out the details

Home > Identity > Sign-up > User name/Password

Sign-up with User Name/Password

| User Registration | |
|-------------------|--|
| User name* | <input type="text" value="Moi"/> |
| Password* | <input type="password" value="....."/> |
| Re-type Password* | <input type="password"/> |
| Full Name * | <input type="text"/> |
| First Name * | <input type="text"/> |
| Last Name * | <input type="text"/> |
| Organization | <input type="text"/> |
| Address | <input type="text"/> |
| Country | <input type="text"/> |
| Email * | <input type="text"/> |

You now have an OpenDaylight account that can be used with Gerrit to pull the OpenDaylight code.

Generating SSH keys for your system

You must have SSH keys for your system to register with your Gerrit account. The method for generating SSH keys is different for different types of operating systems.

The key you register with Gerrit must be identical to the one you will use later to pull or edit the code. For example, if you have a development VM which has a different UID login and keygen than that of your laptop, the SSH key you generate for the VM is different from the laptop. If you register the SSH key generated on your VM with Gerrit and do not reuse it on your laptop when using Git on the laptop, the pull fails.



Note

For more information on SSH keys for Ubuntu, see <https://help.ubuntu.com/community/SSH/OpenSSH/Keys>. For generating SSH keys for Windows, see <https://help.github.com/articles/generating-ssh-keys>.

For a system running Ubuntu operating system, follow the steps below:

1. Run the following command:

```
mkdir ~/.ssh
chmod 700 ~/.ssh
ssh-keygen -t rsa
```

1. You are prompted for a location to save the keys, and a passphrase for the keys.

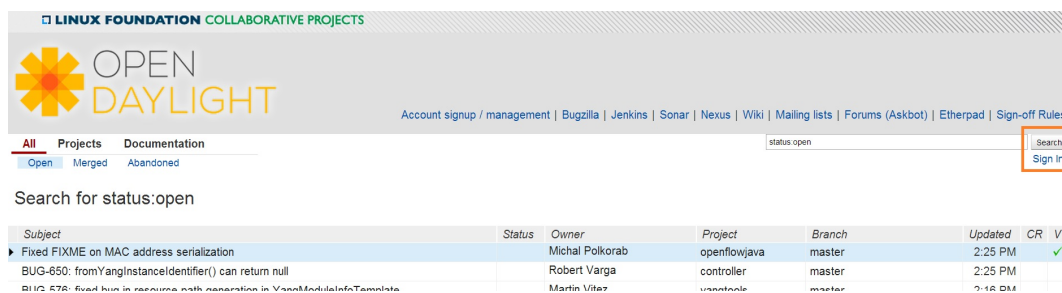
This passphrase protects your private key while it is stored on the hard drive. You must use the passphrase to use the keys every time you need to login to a key-based system.

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/b/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/b/.ssh/id_rsa.
Your public key has been saved in /home/b/.ssh/id_rsa.pub.
```

Your public key is now available as `.ssh/id_rsa.pub` in your home folder.

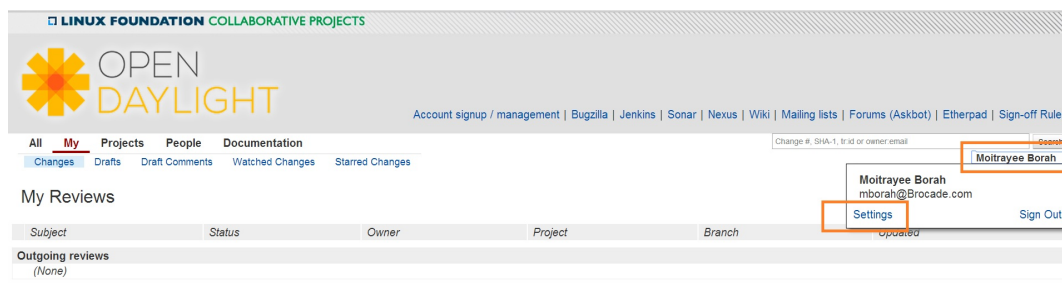
Registering your SSH key with Gerrit

1. Using a Google Chrome or Mozilla Firefox browser, go to <https://git.opendaylight.org/gerrit>.
1. Click **Sign In** to access the OpenDaylight repository.

Figure 1.6. Signin in to OpenDaylight repository

1. Click your name in the top right corner of the window and then click **Settings**.

The **Settings** page is displayed.

Figure 1.7. Settings page for your Gerrit account

1. Click **SSH Public Keys** under **Settings**.

2. Click **Add Key**.

3. In the **Add SSH Public Key** text box, paste the contents of your `id_rsa.pub` file and then click **Add**.

Figure 1.8. Adding your SSH key

To verify your SSH key is working correctly, try using an SSH client to connect to Gerrit's SSHD port.

```
$ ssh -p 29418 <sshusername>@git.opendaylight.org
Enter passphrase for key '/home/cisco/.ssh/id_rsa':
****      Welcome to Gerrit Code Review      ****
Hi <user>, you have successfully connected over SSH.
Unfortunately, interactive shells are disabled.
To clone a hosted Git repository, use: git clone ssh://
<user>@git.opendaylight.org:29418/REPOSITORY_NAME.git
Connection to git.opendaylight.org closed.
```

You can now proceed to either Pulling, Hacking, and Pushing the Code from the CLI or Pulling, Hacking, and Pushing the Code from Eclipse depending on your implementation.

2. Pulling and Pushing the Code from the CLI

Table of Contents

| | |
|---|----|
| Pulling code via Git CLI | 9 |
| Setting up Gerrit Change-id Commit Message Hook | 9 |
| Building the code | 10 |
| Runing OpenDaylight from local build | 10 |
| Commit the code using Git CLI | 11 |
| Pulling the Code changes via Git CLI | 12 |
| Pushing the Code via Git CLI | 12 |

OpenDayligh is a collection of projects, each with their own code repository. This section provides a general guide for to pulling, hacking, and pushing the code for each project. For project specific detail, refer to the project's section in this guide.

Code reviews are enabled through Gerrit. For setting up Gerrit see the section on Getting started with Git and Gerrit.



Note

You will need to perform the Gerrit Setup before you can access git via ssh as described below.

Pulling code via Git CLI

Pull the code by cloning the project's repository.

```
git clone ssh://<username>@git.opendaylight.org:29418/<project_repo_name>.git
```

where <username> is your OpenDaylight username, and <project_repo_name> is the name of the repository for project you are trying to pull. Here is the current list of project repository names:

aaa, affinity, bgpcep, controller, defense4all, dlux, docs, groupbasedpolicy, integration, l2switch, lispflowmapping, odlparent, opendove, openflowjava, openflowplugin, opflex, ovsdb, packetcable, reservation, sdninterfaceapp, sfc, snbi, snmp4sdn, toolkit, ttp, vtn, yangtools.

For an anonymous git clone, you can use:

```
git clone https://git.opendaylight.org/gerrit/p/<project_repo_name>.git
```

Setting up Gerrit Change-id Commit Message Hook

- This command inserts a unique Change-Id tag in the footer of a commit message. This step is optional but highly recommended for tracking changes.

```
cd <project_repo_name>
```

```
scp -p -P 29418 <username>@git.opendaylight.org:hooks/commit-msg .git/hooks/  
chmod 755 .git/hooks/commit-msg
```

- Install and setup Git-review. Git-review is a great tool to simplify the hassle of using several git commands to submit a patch for review. Refer to [How to install and push codes with git-review](#) for instructions. After initializing git-review, both commit-msg hook and a remote repo named gerrit will be created and a patch can be submitted to Gerrit with a single "git review" command.
- Now you can start making your code changes.

Building the code

While you are in the <project_repo_name> directory, run

```
mvn clean install
```

To run without unitests you can skip building those tests running the following:

```
mvn clean install -DskipTests  
/* instead of "mvn clean install" */
```

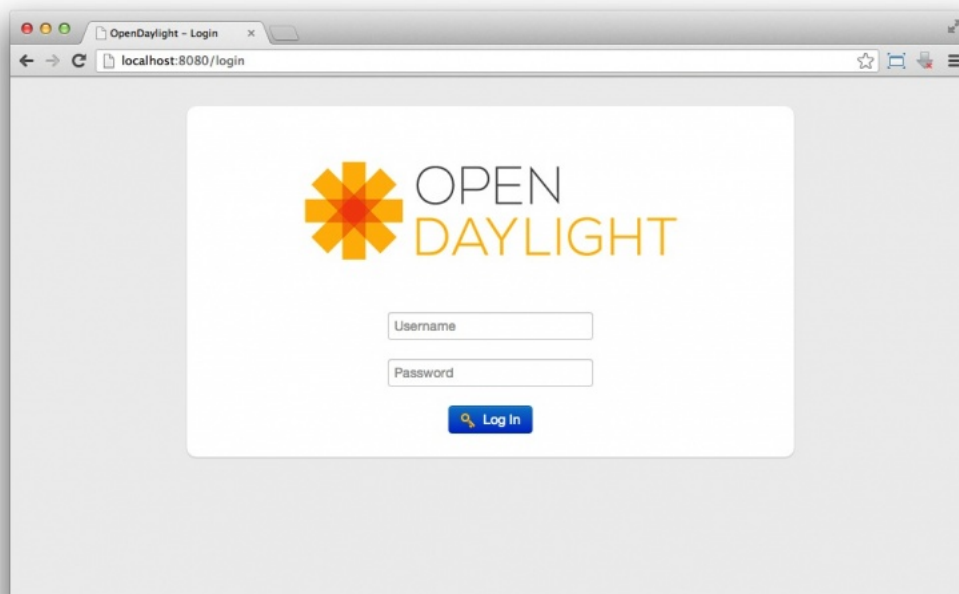
Runing OpenDaylight from local build

Change to the karaf distribution sub-directory, and run

```
./target/assembly/bin/karaf
```

At this point the OpenDaylight controller is running. You can now open a web browser and point your browser at <http://localhost:8080/>

Figure 2.1. OpenDaylight Main Page



Commit the code using Git CLI



Note

To be accepted, all code must come with a [developer certificate of origin](#) as expressed by having a Signed-off-by. This means that you are asserting that you have made the change and you understand that the work was done as part of an open-source license.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

- (a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or
- (b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or
- (c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.
- (d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

Mechanically you do it this way:

```
git commit --signoff
```

You will be prompted for a commit message. If you are fixing a bugzilla bug you can add the associated bug number to your commit message and it will get linked from Gerrit:

For Example:

```
Fix for bug 2.
```

```
Signed-off-by: Ed Warnicke <eaw@cisco.com>
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch develop
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README
#
```

Pulling the Code changes via Git CLI

Pull the latest changes from the remote repository

```
git remote update
git rebase origin/<project_main_branch_name>
```

where <project_main_branch_name> is the the branch you want to commit to. For most projects this is master branch. For some projects such as lispflowmapping, a different branch name (develop in the case of lispflowmapping) should be used.

Pushing the Code via Git CLI

Use git review to push your changes back to the remote repository using:

```
git review
```

You can set a topic for your patch by:

```
git review -t <topic>
```

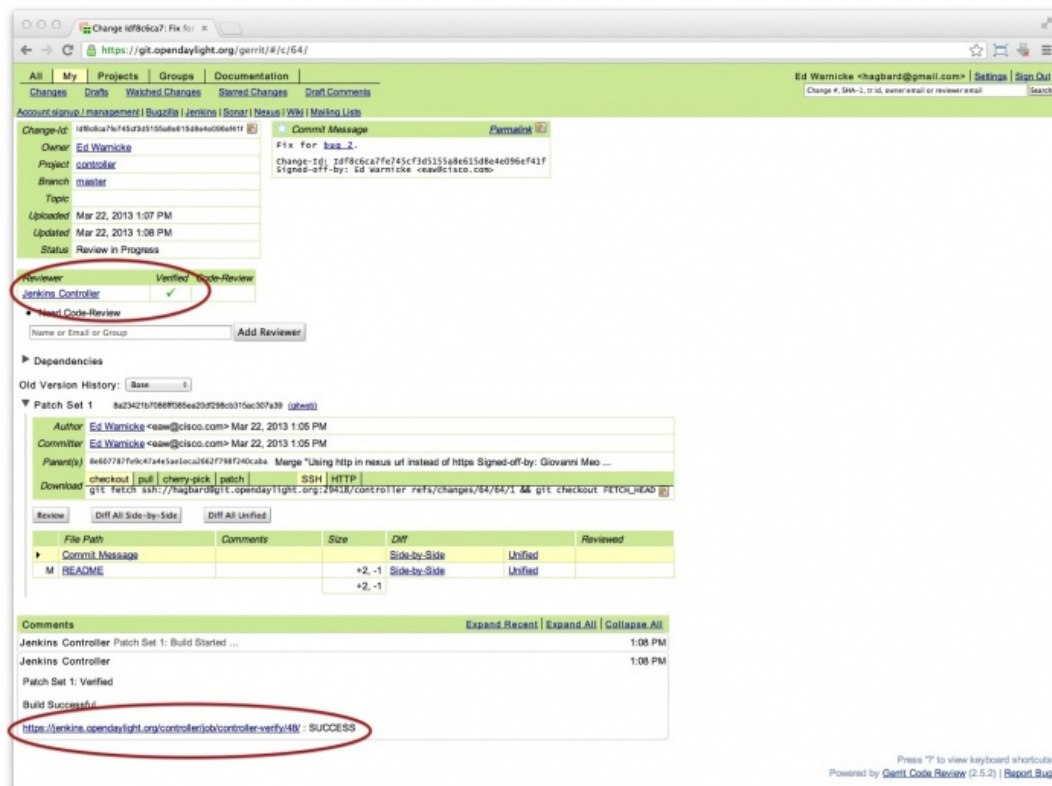
You will get a message pointing you to your gerrit request like:

```
=====
remote: Resolving deltas: 100% (2/2) +
remote: Processing changes: new: 1, refs: 1, done    +
remote: +
remote: New Changes: +
remote:   http://git.opendaylight.org/gerrit/64 +
remote: +
=====
```

The Jenkins Controller User will verify your code and post the result on the your gerrit request.

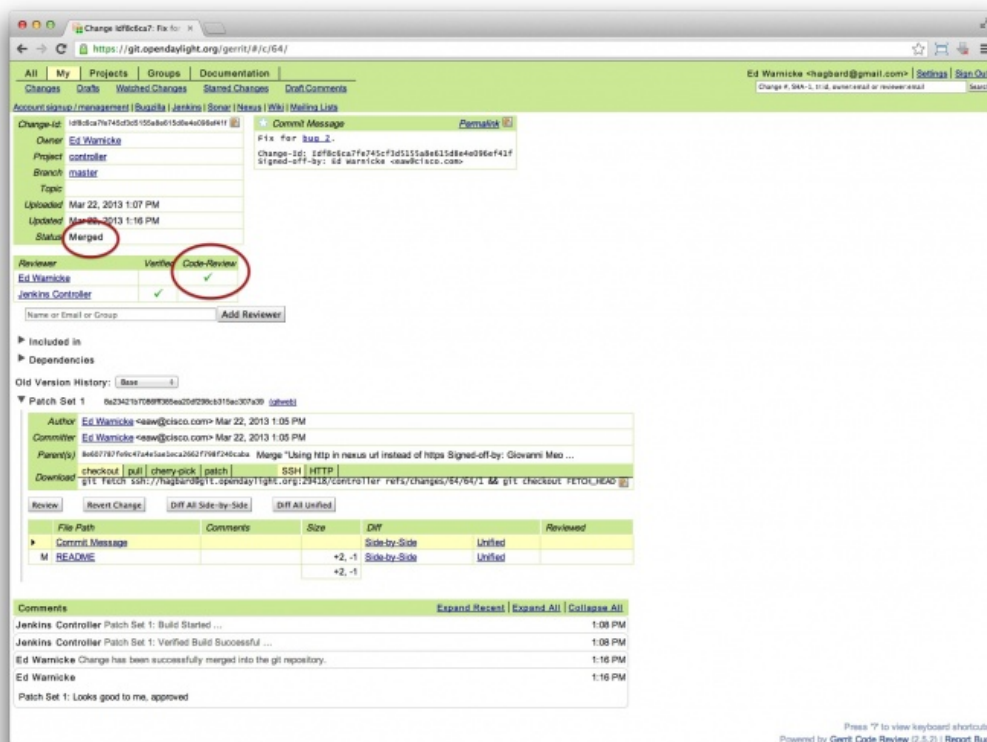
Viewing your Changes in Gerrit

Follow the link you got above to see your commit in Gerrit:

Figure 2.2. Gerrit Code Review Sample

Note that the Jenkins Controller User has verified your code and at the bottom is a link to the Jenkins build.

Once your code has been reviewed and submitted by a committer it will be merged into the authoritative repo, which would look like this:

Figure 2.3. Gerrit Code Merge Sample

Troubleshooting

1. What to do if your Firewall blocks port 29418

There have been reports that many corporate firewalls block port 29418. If that's the case, please follow the [Setting up HTTP in Gerrit](#) instructions and use git URL:

```
git clone https://<your_username>@git.opendaylight.org/gerrit/p/
<project_repo_name>.git
```

You will be prompted for the password you generated in [Setting up HTTP in Gerrit](#).

All other instructions on this page remain unchanged.

To download pre-built images with ODP bootstraps see the following Github project:

[Pre-Built OpenDaylight VM Images](#)

Part II. Project-Specific Development Guides

Table of Contents

| | |
|--|----|
| 3. Authentication Service | 20 |
| Authenthentication data model | 20 |
| How the ODL Authentication Service works | 21 |
| Configuring Authentication service | 22 |
| How federated authentication is set up | 23 |
| Mapping users to roles and domains | 23 |
| Actors in ODL Authentication Service | 26 |
| Sub-components of ODL Authentication Service | 26 |
| 4. BGP LS PCEP | 28 |
| BGPCEP Overview | 28 |
| Implementing an Extension to PCEP | 29 |
| Update Configuration | 29 |
| Implementing an Extension to BGP | 30 |
| Updating Configuration | 30 |
| Vendor Information TLV | 33 |
| Vendor Information Object | 36 |
| 5. Controller | 40 |
| OpenDaylight Controller: MD-SAL Developers' Guide | 41 |
| API types | 41 |
| Basic YANG concepts and their rendition in APIs | 41 |
| MD-SAL: Plugin types | 45 |
| Protocol library | 46 |
| MD-SAL: Southbound plugin development guide | 46 |
| Definition of YANG models | 47 |
| RPCs | 47 |
| Augmentations | 49 |
| Best practices | 49 |
| Implementation | 49 |
| Notifications | 50 |
| Best practices | 50 |
| OpenDaylight Controller: MD-SAL FAQs | 51 |
| OpenDaylight Controller Configuration: Java Code Generator | 58 |
| Service interfaces generating | 58 |
| Runtime beans generating | 59 |
| OpenDaylight Controller MD-SAL: Restconf | 59 |
| Mount point | 60 |
| Something practical | 64 |
| OpenDaylight Controller: Configuration | 67 |
| APIs and SPIs | 69 |
| OpenDaylight Controller configuration: Initial | 71 |
| Initial configuration for controller | 71 |
| OpenDaylight Controller configuration: config.ini | 72 |
| OpenDaylight Controller: Configuration Persister | 72 |
| Current configuration for controller distribution | 73 |
| Adding custom initial configuration | 79 |
| Persister Notification Handler | 83 |
| MD-SAL architecture: Clustering Notifications | 86 |
| MD-SAL Architecture: DOM | 87 |

| | |
|--|-----|
| MD-SAL: Infinispan Data Store | 88 |
| State of the POC | 91 |
| Infinispan-related learnings | 92 |
| Datastore-related learnings | 92 |
| No clarity on the closing of Read-Only transactions | 92 |
| OpenDaylight Controller configuration: FAQs | 95 |
| OpenDaylight Controller configuration: Component map | 95 |
| OpenDaylight Controller: Netconf component map | 97 |
| OpenDaylight Controller Configuration: Examples sample project | 97 |
| OpenDaylight Controller: Configuration examples user guide | 109 |
| OpenDaylight Controller Configuration: Logback Examples | 118 |
| OpenDaylight Controller: Configuration Logback.xml | 125 |
| Configuration example of thread pools using yangcli-pro | 126 |
| Configuration example of thread pools using telnet | 126 |
| Connecting to plaintext TCP socket | 126 |
| Configuring threadfactory | 127 |
| Configuring fixed threadpool | 130 |
| OpenDaylight Controller MD-SAL: Model reference | 131 |
| 6. Defense4all | 132 |
| Defense4All Design | 132 |
| Defense4All in an ODL Environment | 133 |
| Framework View | 134 |
| Application View | 136 |
| ODL Reps View | 138 |
| Basic Control Flow | 141 |
| Configurations and Setup Flow | 141 |
| Attack Detection Flow | 142 |
| Attack Mitigation Flow | 142 |
| Continuity | 143 |
| 7. DLUX | 146 |
| Setup and Run | 146 |
| DLUX Modules | 148 |
| Yang Utils | 151 |
| 8. Group-Based Policy | 153 |
| Group-Based Policy Architecture Overview | 154 |
| Policy Model | 155 |
| State Repositories | 169 |
| Renderers | 170 |
| 9. L2Switch | 189 |
| Checking out the L2Switch project | 189 |
| Testing your changes to the L2Switch project | 189 |
| Architecture of the L2Switch project | 190 |
| Developer's Guide for Packet Dispatcher | 191 |
| Developer's Guide for Loop Remover | 191 |
| Developer's Guide for Arp Handler | 193 |
| Developer's Guide for Address Tracker | 195 |
| Developer's Guide for Host Tracker | 197 |
| Developer's Guide for L2Switch Main | 197 |
| 10. Lisp Flow Mapping | 200 |
| OpenDaylight Locator/ID Separation Protocol (LISP) Flow Mapping Overview | 200 |
| LISP Flow Mapping Service | 201 |

| | |
|--|-----|
| LISP Service Architecture | 201 |
| LISP APIs | 203 |
| LISP Configuration Options | 203 |
| Developer Tutorial | 203 |
| LISP Support | 210 |
| Installing LISP Flow Mapping | 210 |
| 11. ODL-SDNi | 216 |
| 12. OpenFlow Protocol Library | 217 |
| 13. OpenFlow Plugin | 218 |
| OpenFlow Plugin: Sequence diagrams | 219 |
| OpenFlow Plugin:Config subsystem | 223 |
| Message Spy in OF Plugin | 229 |
| OpenFlow Plugin:Mininet | 232 |
| Installation | 232 |
| Usage | 235 |
| Coding tips for OpenFlow Plugin | 235 |
| OpenFlow Plugin: Wiring up notifications | 237 |
| OpenFlow Plugin:Python test scripts | 239 |
| General | 240 |
| ODL Test (odl_crud_tests.py) | 241 |
| Parameters | 242 |
| Stress Test (stress_test.py) | 243 |
| Operational Data Test (oper_data_test.py) | 243 |
| Switch restart (sw_restart_test.py) | 243 |
| OpenFlow Plugin: Robot framework tests | 244 |
| TLS support for OF Plugin | 245 |
| Configuring the ODL OpenFlow plugin | 247 |
| Configuring openvswitch SSL | 247 |
| Configuring a hardware switch with TLS | 248 |
| Open Flow Plugin: Support for extensibility | 249 |
| Overload protection in the OF Plugin | 251 |
| 14. OVSDb Integration | 254 |
| OpenDaylight OVSDb integration | 254 |
| Building and running OVSDb | 257 |
| OVSDb integration design | 260 |
| OpenDaylight OVSDb southbound plugin architecture and design | 260 |
| OVSDb southbound plugin | 261 |
| Connection service | 261 |
| Network Configuration Service | 263 |
| OpenDaylight OVSDb Developer Getting Started Video Series | 267 |
| OVSDb integration: New features | 267 |
| 15. Packet Cable MultiMedia (PCMM) | 275 |
| Checking out the Packetcable PCMM project | 275 |
| System Overview | 275 |
| Dependency Map | 276 |
| Packetcable Components | 276 |
| Download and Install | 277 |
| Preparing to Work with the Packetcable PCMM Service | 277 |
| Explore and exercise the PacketCable REST API | 281 |
| RESTCONF API Explorer | 281 |
| Postman | 282 |

| | |
|---|-----|
| Custom Testsuite | 282 |
| Using Wireshark to Trace PCMM | 282 |
| Debugging and Verifying DQoS Gate (Flows) on the CMTS | 283 |
| Find the Cable Modem | 283 |
| Arris | 285 |
| RESTCONF API for Packetcable PCMM | 285 |
| 16. Plugin for OpenContrail | 289 |
| 17. Service Function Chaining | 290 |
| 18. SNBI Developers' Guide | 291 |
| Defining characteristics of SNBI bootstrapping | 291 |
| SNBI components | 291 |
| How SNBI works | 292 |
| 19. SNMP4SDN | 296 |
| 20. TCP-MD5 | 297 |
| 21. Table Type Patterns | 298 |
| Introduction | 298 |
| Using The REST APIs | 299 |
| Limitations | 309 |
| 22. VTN | 312 |
| Virtual Tenant Network (VTN) | 312 |
| Hacking VTN Coordinator | 326 |
| Hacking VTN Manager | 329 |
| Hacking VTN Manager(Helium) | 330 |
| Openstack Support Developer Guide | 332 |
| 23. YANG Tools | 336 |
| Prerequisites for YANG Tools Project | 336 |
| Pulling code using ssh | 336 |
| Pulling code using https | 336 |
| Building the code | 337 |
| Mapping YANG to Java | 337 |
| Additional Packages | 338 |
| Data Interface | 340 |
| Service Interface | 340 |

3. Authentication Service

Table of Contents

| | |
|--|----|
| Authenthentication data model | 20 |
| How the ODL Authentication Service works | 21 |
| Configuring Authentication service | 22 |
| How federated authentication is set up | 23 |
| Mapping users to roles and domains | 23 |
| Actors in ODL Authentication Service | 26 |
| Sub-components of ODL Authentication Service | 26 |

Authentication uses the credentials presented by a user to identify the user.

Authenthentication data model

A user requests authentication within a domain in which the user has defined roles. The user chooses either of the following ways to request authentication:

- Provides credentials
- Creates a token scoped to a domain. In OpenDaylight, a domain is a grouping of resources (direct or indirect, physical, logical, or virtual) for the purpose of access control.

Terms and definitions in the model

| | |
|------------|---|
| Token | A claim of access to a group of resources on the controller |
| Domain | A group or isolation of resources, direct or indirect, physical, logical, or virtual, for the purpose of access control |
| User | A person who either owns and has, or has, access to a resource or group of resources on the controller |
| Role | Opaque representation of a set of permissions, which is merely a unique string as admin or guest |
| Credential | Proof of identity such as username and password, OTP, biometrics, or others |
| Client | A service or application that requires access to the controller |

Authentication methods

There are three ways a user may authenticate in OpenDaylight:

- Token-based Authentication

- Direct authentication: A user presents username/password and a domain the user wishes to access to the controller and obtains a timed (default is 1 hour) scoped access token. The user then uses this token to access Restconf (for example).
- Federated authentication: A user presents credentials to a third-party Identity Provider (for example, SSSD) trusted by the controller. Upon successful authentication, the controller returns a refresh (unscoped) token with a list of domains that the user has access to. The user then presents this refresh token scoped to a domain that the user has access to obtain a scoped access token. The user then uses this access token to access Restconf (for example).
- Basic Authentication

For backward compatibility with the ODL Hydrogen release, the controller also supports the normal basic authentication with username/password.

Example with token authentication using curl (username/password = admin/admin, domain = sdn):

```
# Create a token
curl -ik -d 'grant_type=password&username=admin&password=admin&scope=sdn'
http://localhost:8181/oauth2/token

# Use the token (e.g., ed3e5e05-b5e7-3865-9f63-eb8ed5c87fb9) obtained from
above (default token validity is 1 hour):
curl -ik -H 'Authorization:Bearer ed3e5e05-b5e7-3865-9f63-eb8ed5c87fb9'
http://localhost:8181/restconf/config/toaster:toaster
```

Example with basic auth using curl:

```
curl -ik -u 'admin:admin' http://localhost:8181/restconf/config/
toaster:toaster
```

How the ODL Authentication Service works

In direct authentication, a service relationship exists between the user and the ODL controller. The user and the controller establish trust that allows them to use, and validate credentials. The user establishes user identity through credentials.

In direct authentication, a user request progresses through the following steps:

1. The user requests the controller administrator for a user account.

Associated with the user account are user credentials, initially created by the administrator. Opendaylight supports only username/password credentials. By default, an administrator account is included with ODL out-of-the-box the username and password for which are admin/admin. In addition to creating the user account, the controller administrator also assigns roles to that account on one or more domains. By default, there are two user roles: admin and user. By default, there is only one domain: sdn.

2. The user presents the credentials to the service within a domain in a request for a token.

3. The request is then passed on to the controller token endpoint.
4. The controller token endpoint sends it to the credential authentication entity which returns a claim for the client.
5. The controller token entity transforms the claim (for user, domain, and roles) into a token which it then provides to the user.

In federated authentication, with the absence of a direct trust relationship between the user and the service, a third-party Identity Provider (IdP) is used for authentication. Federated authentication relies on third-party identity providers (IdP) to authenticate the user. An example of an external IdP is Linux SSSD (System Security Services Daemon) or Openstack Keystone.

The user is authenticated by the trusted IdP and a claim is returned to the ODL authentication services upon successful authentication. The claim is mapped into ODL users or roles and transformed into a token that is passed onto the user. The request is passed on to the claim authentication broker that transforms it to a claim. The controller turns the claim into a token that is passed on to the user.

In a federated authentication set-up, the Opendaylight controller extends SSSD claim support. SSSD also provides mapping capabilities. SSSD maps users in an external LDAP server to users defined on the Opendaylight controller.

Configuring Authentication service

Changes to AAA configurations can be made from the following:

- Webconsole
- CLI (config command in the Karaf shell)
- Editing the etc/org.opendaylight.aaa.*.cfg files directly

Every Authentication Service karaf feature has its configuration file.



Note

Configurations for AAA are all dynamic and require no restart.

To configure features from the Web console:

1. Install the Web console:

```
feature:install webconsole
```

1. On the console (<http://localhost:8181/system/console>) (default Karaf username/ password: karaf/karaf), go to **OSGi > Configuration > ODL AAA Authentication Configuration**.
 - a. **Authorized Clients:** List of software clients that are authorized to access ODL NB APIs.
 - b. **Enable Authentication:** Enable or disable authentication. (The default is enable.)

Configuring tokens

1. On the console, click **ODL AAA Token Configuration**.

The fields you can configure are as follows:

- a. **Memory Configuration:** Configure the maximum number of tokens to be retained in memory.
- b. **Disk Configuration:** The maximum number of tokens to be retained on the disk.



Note

When Memory is exhausted, tokens are moved to the disk.

- a. **Token Expiration:** The number of seconds that a token remains live irrespective of use.
- b. **Unused Token Expiration:** The number of seconds that a token is live without being accessed. (The default period for both Expiration fields is 1 hour or 3600 seconds.)

Configuring AAA federation

1. On the console, click **ODL AAA Federation Configuration**.
2. Use the **Custom HTTP Headers** or **Custom HTTP Attributes** fields to specify the HTTP headers or attributes for federated authentication. Normally, such specification is not required.



Note

As the changes you make to the configurations are automatically committed when they are saved, no restart of the Authentication service is required.

How federated authentication is set up

Use the following steps to set up federated authentication:

1. Set up an Apache front-end and Apache mods for the ODL controller.
2. Set up mapping rules (from LDAP users to ODL users).
3. Use the ClaimAuthFilter in federation to allow claim transformation.

Mapping users to roles and domains

The ODL authentication service transforms assertions from an external federated IdP into Authentication Service data:

1. The Apache web server which fronts ODL AAA sends data to SssdAuthFilter.
2. SssdAuthFilter constructs a JSON document from the data.

3. ODL Authentication Service uses a general purpose transformation mapper to transform the JSON document.

Operational model

The mapping model works as follows:

1. Assertions from an IdP are stored in an associative array.
2. A sequence of rules is applied, and the first rule which returns success is considered a match.
3. Upon success, an associative array of mapped values is returned.
 - The mapped values are taken from the local variables set during the rule execution.
 - The definition of the rules and mapped results are expressed in JSON notation.

Operational Model: Sample code

```
mapped = null
foreach rule in rules {
    result = null
    initialize rule.variables with pre-defined values

    foreach block in rule.statement_blocks {
        for statement in block.statements {
            if statement.verb is exit {
                result = exit.status
                break
            }
            elif statement.verb is continue {
                break
            }
        }
        if result {
            break
        }
        if result == null {
            result = success
        }
    }
    if result == success {
        mapped = rule.mapping(rule.variables)
    }
}
return mapped
```

Mapping Users

A JSON Object acts as a mapping template to produce the final associative array of name/value pairs. The value in a name/value pair can be a constant or a variable. An example of a mapping template and rule variables in JSON: Template:

```
{
  "organization": "BigCorp.com",
  "user": "$subject",
  "roles": "$roles"
```

```
}
```

Local variables:

```
{
  "subject": "Sally",
  "roles": ["user", "admin"]
}
```

The final mapped result will be:

```
{
  "organization": "BigCorp.com",
  "user": "Sally",
  "roles": ["user", "admin"]
}
```

Example: Splitting a fully qualified username into user and realm components

Some IdPs return a fully qualified username (for example, principal or subject). The fully qualified username is the concatenation of the user name, separator, and realm name. The following example shows the mapped result that returns the user and realm as independent values for the fully qualified username is [bob@example.com](#).

The mapping in JSON:

```
{
  "user": "$username",
  "realm": "$domain"
}
```

The assertion in JSON:

```
{
  "Principal": "bob@example.com"
}
```

The rule applied:

```
[
  [
    ["in", "Principal", "assertion"],
    ["exit", "rule_fails", "if_not_success"],
    ["regex", "$assertion[Principal]", "(?P<username>\\w+)(?P<domain>.+)",
    ],
    ["set", "$username", "$regex_map[username]"],
    ["set", "$domain", "$regex_map[domain]"],
    ["exit", "rule_succeeds", "always"]
  ]
]
```

The mapped result in JSON:

```
{
  "user": "bob",
  "realm": "example.com"
}
```

Also, users may be granted roles based on their membership in certain groups.

The Authentication Service allows white lists for users with specific roles. The white lists ensure that users are unconditionally accepted and authorized with specific roles. Users who must be unconditionally denied access can be placed in a black list.

Actors in ODL Authentication Service

ODL Controller administrator The ODL Controller administrator has the following responsibilities:

- Authors Authentication policies using the REST API
- Provides credentials, usernames and passwords to users who request them

ODL resource owners Resource owners authenticate (either by means of federation or directly providing their own credentials to the controller) to obtain an access token. This access token can then be used to access resources on the controller. An ODL resource owner enjoys the following privileges:

- Creates, refreshes, or deletes access tokens
- Gets access tokens from the Secure Token Service
- Passes secure tokens to resource users

ODL resource users Resource users do not need to authenticate: they can access resources if they are given an access tokens by the resource owner. The default timeout for access tokens is 1 hour (This duration is configurable.). An ODL resource user does the following:

- Gets access tokens either from a resource owner or the controller administrator
- Uses tokens at access applications from the north-bound APIs

Sub-components of ODL Authentication Service

| | |
|--|--|
| AuthX authoring service | Provides AuthN and AuthZ Authoring service |
| Light-weight Identity Manager (IdmLight) | Stores local user authentication and authorization data, and roles Provides an Admin REST API for CRUD users/roles/domains |
| Pluggable authenticators | Provides domain-specific authentication mechanisms |
| Authenticator | Authenticates users against the authentication policy and establishes claims |
| Authentication Cache | Caches all authentication states and tokens |
| Authentication Filter | Verifies tokens and extracts claims |
| Authentication Manager | Contains the session token and authentication claim store |

ODL Authorization Service

In progress is the addition of an authorization feature to the authentication service. Authorization will follow successful authentication. Modelled on the Role Based Access Control (RBAC) approach for authentication, the Authorization service will assign roles that define permissions and decide access levels. Authorization will do the following:

- Verify the operations the user or service is authorized to do
- Enforce policies to grant or deny access to resources

4. BGP LS PCEP

Table of Contents

| | |
|---|----|
| BGPCEP Overview | 28 |
| Implementing an Extension to PCEP | 29 |
| Update Configuration | 29 |
| Implementing an Extension to BGP | 30 |
| Updating Configuration | 30 |
| Vendor Information TLV | 33 |
| Vendor Information Object | 36 |

BGPCEP Overview

An extension to a protocol means adding parsers and serializers for new elements, such as messages, objects, TLVs or subobjects. This is necessary when you are extending the protocol with another RFC or draft. Both BGP and PCEP parsers are pluggable and you can specify which extensions to load alongside to the base parser in the configuration file.

Writing an extension to PCE protocol Current standards support Current pcep base-parser implementation supports following RFCs:

[RFC5440](#) - Path Computation Element (PCE) Communication Protocol (PCEP) [RFC5541](#) - Encoding of Objective Functions in the Path Computation Element Communication Protocol (PCEP) [RFC5455](#) - Diffserv-Aware Class-Type Object for the Path Computation Element Communication Protocol [RFC5521](#) - Extensions to the Path Computation Element Communication Protocol (PCEP) for Route Exclusions [RFC5557](#) - Path Computation Element Communication Protocol (PCEP) Requirements and Protocol Extensions in Support of Global Concurrent Optimization

There are already two extensions for: [draft-ietf-pce-stateful-pce](#) - in versions 02 and 07 [draft-ietf-pce-pce-initiated-lsp](#) - versions crabbe-initiated-00 and ietf-initiated-00

```
dependency>
  <groupId>${project.groupId}</groupId>
  <artifactId>pcep-ietf-stateful02</artifactId>
</dependency>
```

```
<dependency>
  <groupId>${project.groupId}</groupId>
  <artifactId>pcep-ietf-stateful07</artifactId>
</dependency>
```



Note

It is important to load the extensions with compatible versions because that they extend each other. In this case crabbe-initiated-00 is compatible with stateful-02 and ietf-initiated-00 is compatible with stateful-07.

Implementing an Extension to PCEP

To implement an extension of PCEP:

1. Create a separate artefact (eclipse project) for your extension. Ensure the dependency on pcep-api and pcep-spi.
2. Write YANG model for new elements or augment existing ones.
3. Perform `mvn install` to generate files from the model.
4. Write parsers and serializers. All parsers need to implement `*Parser` and `*Serializer` interfaces from pcep-spi, (For example: If you are writing a new TLV, your parser must implement `TlvParser` and `TlvSerializer`), add `Activator`, that extends `AbstractPCEPExtensionProviderActivator`, where you register your parsers and serializers.

Update Configuration

Update [32-pcep.xml]. Register your parser as a module in pcep-impl:

```
<module>
  <type
    xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:pcep:impl">
      prefix:pcep-parser-new-parser
    </type>
  <name>pcep-parser-new-parser</name>
</module>
```

- Add it as an extension to pcep-parser-base:

```
<extension>
  <type
    xmlns:pcepspi="urn:opendaylight:params:xml:ns:yang:controller:pcep:spi">
      pcepspi:extension
    </type>
  <name>pcep-parser-new-parser</name>
</extension>
```

- Add the instance to services:

```
<instance>
  <name>pcep-parser-new-parser</name>
  <provider>/config/modules/module[name='pcep-parser-new-parser']/
instance[name='pcep-parser-new-parser']</provider>
</instance>
```

- Update `odl-pcep-impl-cfg.yang` so that it generates `Module` and `ModuleFactory` classes for your new parser.

```
identity pcep-parser-new-parser {
  base config:module-type;
  config:provided-service spi:extension;
  config:java-name-prefix NewParserPCEPParser;
}
```

```
augment "/config/modules/config:module/config:configuration" {
```

```
case pcep-parser-new-parser {  
  when "/config:modules/config:module/config:type = 'pcep-parser-new-parser'";  
}  
}
```

Run `mvn install` on `pcep-impl-config` to generate Module and ModuleFactory files. * Update Module to start your `NewParserPCEPParserModule.java` when it's created

```
@Override  
public java.lang.AutoCloseable createInstance() {  
  return new InitiatedActivator();  
}
```

Writing an Extension to BGP

Current standards support

Current bgp base-parser implementation supports following RFCs:

[RFC4271](#) - A Border Gateway Protocol 4 (BGP-4) [RFC4724](#) - Graceful Restart Mechanism for BGP [RFC4760](#) - Multiprotocol Extensions for BGP-4 [RFC1997](#) - BGP Communities Attribute [RFC4360](#) - BGP Extended Communities Attribute [RFC6793](#) - BGP Support for Four-Octet Autonomous System (AS) Number Space (NEW speaker only)

There is already one extension for: [draft-ietf-idr-ls-distribution](#) - in version 04

Implementing an Extension to BGP

To implement an extension to BGP:

1. Create a separate artefact (eclipse project) for your extension. Ensure it depends on `pcep-api` and `pcep-spi`.
2. Write yang model for new elements or augment existing ones.
3. Perform `mvn install` to generate files from the model.
4. Write parsers and serializers. All parsers need to implement **Parser** and **Serializer** interfaces from `bgp-spi`. For example: If you are writing a new capability, your parser should implement `CapabilityParser` and `CapabilitySerializer`). Add `Activator`, that extends `AbstractBGPExtensionProviderActivator`, where you register your parsers and serializers. If your extension adds another AFI/SAFI you must to add another `Activator` that extends `AbstractRIBExtensionProviderActivator` and registrate new address family and subsequent address family.

Updating Configuration

Update `31-bgp.xml`. Register your parser as a module in `bgp-impl`:

```
<module>  
  <type xmlns:prefix="urn:.opendaylight:params:xml:ns:yang:controller:bgp:new-  
parser">
```

```
    prefix:bgp-new-parser
  </type>
  <name>bgp-new-parser</name>
</module>
```

- Add it as an extension to bgp-parser-base:

```
<extension>
  <type
    xmlns:bgpspi="urn:opendaylight:params:xml:ns:yang:controller:bgp:parser:spi">
    bgpspi:extension
  </type>
  <name>bgp-new-parser</name>
</extension>
```

- Add the instance to services:

```
<instance>
  <name>bgp-new-parser</name>
  <provider>/modules/module[type='bgp-new-parser'][name='bgp-new-parser']</
provider>
</instance>
```

Also, if you are introducing new AFI/SAFI, do not forget to registrate your extension also to RIB.

- Create your own configuration file so that it generates Module and ModuleFactory classes for your new parser.

```
identity bgp-new-parser {
  base config:module-type;
  config:provided-service bgpspi:extension;
  config:provided-service ribspi:extension; // for new AFI/SAFI
  config:java-name-prefix NewParser;
}
```

```
augment "/config:modules/config:module/config:configuration" {
  case bgp-new-parser {
    when "/config:modules/config:module/config:type = 'bgp-new-
parser'";
  }
}
```

Run `mvn install` on your extension artefact to generate Module and ModuleFactory files.

- Update Module to start your NewParserModule.java when it's created.

```
@Override
public java.lang.AutoCloseable createInstance() {
  return new NewParserActivator();
}
```

Programmatic Interface(s)

Howto pull code from gerrit: [OpenDaylight Controller:Pulling, Hacking, and Pushing the Code from the CLI](#) Gerrit repository: [gerrit](#) Bugzilla: [Bugzilla](#) Mailing lists

- bgpcep-bugs@opendaylight.org

- bgpcep-dev@.opendaylight.org

YANG Models - [BGP LS PCEP:Models](#)

API Documentation – [Javadoc API](#)

For debugging purposes, set lower log levels for bgpcep project in logback.xml.

```
<logger name="org.opendaylight.protocol" level="TRACE" />
<logger name="org.opendaylight.bgpcep" level="TRACE" />
```

Vendor Specific Constraints in PCEP

[draft-ietf-pce-rfc7150bis-00](#) - Conveying Vendor-Specific Constraints in the Path Computation Element communication Protocol.

Draft defines new PCEP object - Vendor Information object, that can be used to carry arbitrary, proprietary information such as vendor-specific constraints. Draft also defines new PCEP TLV - Vendor Information TLV that can be used to carry arbitrary information within any PCEP object that supports TLVs.

The ODL PCEP supports *draft-ietf-pce-rfc7150bis-00* and provides abstraction for developers to create vendor-specific TLVs/objects extensions. The yang model of *vendor-information-tlv/object* is defined in *pcep-types.yang* and used in pcep objects/messages as defined in the draft.

This tutorial shows how to develop PCEP extension of vendor-information object and TLV for fictional company named My Vendor, whose enterprise number is 0. A result will be OSGi bundle and initial configuration xml file, that supports MY-VENDOR-TLV and MY-VENDOR-OBJECT in ODL.

- First, create simple maven module named *pcep-my-vendor*. For simplification assume the module parent is *pcep* maven project. For bundle packaging add *plugin maven-bundle-plugin* into *pom.xml* and also *yang-maven-plugin* for compile-time java code generating.

```
<artifactId>pcep-my-vendor</artifactId>
<description>PCEP MY VENDOR EXTENSION</description>
<packaging>bundle</packaging>
<name>${project.artifactId}</name>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <extensions>true</extensions>
      <configuration>
        <instructions>
          <Bundle-Name>${project.groupId}.${project.artifactId}</Bundle-
Name>
        </instructions>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.opendaylight.yangtools</groupId>
      <artifactId>yang-maven-plugin</artifactId>
    </plugin>
  </plugins>
```

```
</build>
```

- Add required dependencies into *pom.xml*.

```
<dependencies>
  <dependency>
    <groupId>org.opendaylight.controller</groupId>
    <artifactId>config-api</artifactId>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>pcep-api</artifactId>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>pcep-spi</artifactId>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>pcep-impl</artifactId>
  </dependency>
</dependencies>
```

Vendor Information TLV

The Vendor Information TLV is used for vendor-specific information that applies to a specific PCEP object by including the TLV in the object. For the purpose of this tutorial, define MY-VENDOR-TLV, which can be loaded with just simple unsigned 32-bit integer (4 bytes) as its value and the TLV is carried in Open object.

Yang model

- Initial step is to extend *pcep-types* and *pcep-message* yang models, augmentation target is *enterprise-specific-information* (choice) located in Open messages's Open object. Create yang file (*pcep-my-vendor.yang*), in project's *src/main/yang* folder, with definition of the vendor information and required augmentations.
- Now build project with maven, after that generated Java API's appears in *target/generated-sources/sal*.

```
grouping my-vendor-information {
  leaf payload {
    type uint32;
  }
}
augment "/msg:open/msg:open-message/msg:open/msg:tlvs/msg:vendor-information-tlv/msg:enterprise-specific-information" {
  case my-vendor {
    when "enterprise-number = 0";
    uses my-vendor-information;
  }
}
```

- Vendor Information TLV parser/serializer
- Next step is an implementation of the enterprise-specific-information (TLV's value) parser/serializer. It is simple serialization/deserialization of unsigned integer

(long type in Java representation), other functionality is already presented in *org.opendaylight.protocol.pcep.impl.tlv.AbstractVendorInformationTlvParser* abstract class. Create class extending *AbstractVendorInformationTlvParser* and implement missing methods.

```
public class MyVendorInformationTlvParser extends
AbstractVendorInformationTlvParser {
    private static final EnterpriseNumber EN = new EnterpriseNumber(0L);
    @Override
    public EnterpriseNumber getEnterpriseNumber() {
        return EN;
    }
    @Override
    public EnterpriseSpecificInformation
    parseEnterpriseSpecificInformation(final ByteBuf buffer)
        throws PCEPDeserializerException {
        return new
MyVendorBuilder().setPayload(buffer.readUnsignedInt()).build();
    }
    @Override
    public void serializeEnterpriseSpecificInformation(final
EnterpriseSpecificInformation esi, final ByteBuf buffer) {
        final MyVendor myVendorInfo = (MyVendor) esi;
        buffer.writeInt(myVendorInfo.getPayload().intValue());
    }
}
```

Vendor Information TLV Activator

- Now, parser/serializer needs to be registered to *VendorInformationTlvRegistry*. Create class extending *AbstractPCEPExtensionProviderActivator* and implement *startImpl* method - register parser identified by enterprise number and register serializer identified by the class extending *EnterpriseSpecificInformation*.

```
public class Activator extends AbstractPCEPExtensionProviderActivator {
    @Override
    protected List<AutoCloseable> startImpl(PCEPExtensionProviderContext
context) {
        final List<AutoCloseable> regs = new ArrayList<>();
        final MyVendorInformationTlvParser parser = new
MyVendorInformationTlvParser();

        regs.add(context.registerVendorInformationTlvParser(parser.getEnterpriseNumber(),
parser));

        regs.add(context.registerVendorInformationTlvSerializer(MyVendor.class,
parser));
        return regs;
    }
}
```

Configuration Module

- Create configuration yang module with name i.e. *pcep-my-vendor-cfg.yang*. Define My Vendor parser extension service provider config module.
- Build project with maven to generate configuration module and module factory. They are located in *src/main/java*.

- Implement *MyVendorPCEPParserModule#createInstance()* - return instance of Activator created above.

```
identity pcep-parser-my-vendor {
    base config:module-type;
    config:provided-service spi:extension;
    config:java-name-prefix MyVendorPCEPParser;
}
augment "/config:modules/config:module/config:configuration" {
    case pcep-parser-my-vendor {
        when "/config:modules/config:module/config:type = 'pcep-parser-my-vendor'";
    }
}
```

```
@Override
public java.lang.AutoCloseable createInstance() {
    return new Activator();
}
```

Initial Configuration

Create initial configuration xml file, where module *pcep-parser-my-vendor* is instantiated and injected into the *global-pcep-extensions*.

```
<snapshot>
  <required-capabilities>
    <capability>urn:opendaylight:params:xml:ns:yang:controller:pcep:spi?
module=odl-pcep-spi-cfg&revision=2013-11-15</capability>

    <capability>urn:opendaylight:params:xml:ns:yang:controller:pcep:my:vendor:cfg?
module=pcep-my-vendor-cfg&revision=2014-09-20</capability>
  </required-capabilities>
  <configuration>
    <data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <modules
xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
        <module>
          <type
xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:pcep:spi">prefix:pcep-
extensions-impl</type>
          <name>global-pcep-extensions</name>
          <extension>
            <type
xmlns:pcepspi="urn:opendaylight:params:xml:ns:yang:controller:pcep:spi">
              pcepspi:extension
            </type>
            <name>pcep-parser-my-vendor</name>
          </extension>
        </module>
      </modules>
      <services
xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
        <type
xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:pcep:my:vendor:cfg">
          prefix:pcep-parser-my-vendor
        </type>
        <name>pcep-parser-my-vendor</name>
      </services>
    </data>
  </configuration>
</snapshot>
```

```

        <service>
        <type
xmlns:pcepspi="urn:opendaylight:params:xml:ns:yang:controller:pcep:spi">
        pcepspi:extension
        </type>
        <instance>
        <name>pcep-parser-my-vendor</name>
        <provider>/config/modules/module[name='pcep-parser-my-
vendor']/instance[name='pcep-parser-my-vendor']/provider>
        </instance>
        </service>
        </services>
        </data>
        </configuration>
</snapshot>

```

Vendor Information Object

For the tutorial purposes, define MY-VENDOR-OBJECT, which can be loaded with Ipv4 address (4 bytes) as it's value and the object is carried in PCRep message's response.

Yang Model

- Initial step is to extend *pcep-types* and *pcep-message* yang models, augmentation target is *enterprise-specific-information* (choice) located in PCRep messages. Create yang file (*pcep-my-vendor.yang*), in project *src/main/yang* folder, with definition of the vendor information and required augmentations.
- Now build project with maven, after that generated Java API's appears in *target/generated-sources/sal*.

```

grouping my-vendor-information {
    leaf payload {
        type inet:ipv4-address;
    }
}

augment "/msg:pcrep/msg:pcrep-message/msg:replies/msg:vendor-information-
object/msg:enterprise-specific-information" {
    case my-vendor {
        when "enterprise-number = 0";
        uses my-vendor-information;
    }
}

```

Vendor Information Object Parser/Serializer

- Implementation of the *enterprise-sepecific-information* (Object value) parser/serializer. It is simple serialization/deserialization of IPv4 address, other functionality is already presented in *org.opendaylight.protocol.pcep.impl.object.AbstractVendorInformationObjectParser* abstract class. Create class extending *AbstractVendorInformationObjectParser* and implement missing methods.

```

public class MyVendorInformationObjectParser extends
AbstractVendorInformationObjectParser {
    private static final EnterpriseNumber EN = new EnterpriseNumber(0L);
    @Override

```

```

    public EnterpriseNumber getEnterpriseNumber() {
        return EN;
    }
    @Override
    public EnterpriseSpecificInformation
    parseEnterpriseSpecificInformation(final ByteBuf buffer)
        throws PCEPDeserializerException {
        return new
    MyVendorBuilder().setPayload(Ipv4Util.addressForByteBuffer(buffer)).build();
    }
    @Override
    public void serializeEnterpriseSpecificInformation(final
    EnterpriseSpecificInformation esi, final ByteBuf buffer) {
        final MyVendor myVendor = (MyVendor) esi;
        buffer.writeBytes(Ipv4Util.bytesForAddress(myVendor.getPayload()));
    }
}

```

Vendor Information Object Activator

Parser/serializer must be registered to VendorInformationObjectRegistry. Create class extending AbstractPCEPExtensionProviderActivator and implement startImpl method - register parser identified by enterprise number and register serializer identified by the class extending EnterpriseSpecificInformation.

```

public class Activator extends AbstractPCEPExtensionProviderActivator {
    @Override
    protected List<AutoCloseable> startImpl(PCEPExtensionProviderContext
    context) {
        final List<AutoCloseable> regs = new ArrayList<>();
        final MyVendorInformationObjectParser parser = new
    MyVendorInformationObjectParser();

        regs.add(context.registerVendorInformationObjectParser(parser.getEnterpriseNumber(),
        parser));

        regs.add(context.registerVendorInformationObjectSerializer(MyVendor.class,
        parser));
        return regs;
    }
}

```

Configuration Module

- Create configuration yang module with name (*pcep-my-vendor-cfg.yang*).
- Define My Vendor parser extension service provider configuration module.
- Build project with maven to generate configuration module and module factory located in *src/main/java*.
- Implement *MyVendorPCEPParserModule#createInstance()* - return instance of Activator created.

```

identity pcep-parser-my-vendor {
    base config:module-type;
    config:provided-service spi:extension;
    config:java-name-prefix MyVendorPCEPParser;
}

```

```
augment "/config:modules/config:module/config:configuration" { case pcep-parser-my-
vendor { when "/config:modules/config:module/config:type = pcep-parser-my-vendor"; } }
```

```
@Override
public java.lang.AutoCloseable createInstance() {
    return new Activator();
}
```

Initial Configuration

Create initial configuration xml file, where module *pcep-parser-my-vendor* is instantiated and injected into the *global-pcep-extensions*.

```
<snapshot>
  <required-capabilities>
    <capability>urn:opendaylight:params:xml:ns:yang:controller:pcep:spi?
module=odl-pcep-spi-cfg&revision=2013-11-15</capability>

    <capability>urn:opendaylight:params:xml:ns:yang:controller:pcep:my:vendor:cfg?
module=pcep-my-vendor-cfg&revision=2014-09-20</capability>
  </required-capabilities>
  <configuration>
    <data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <modules
xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
        <module>
          <type
xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:pcep:spi">
            prefix:pcep-extensions-impl
          </type>
          <name>global-pcep-extensions</name>
          <extension>
            <type
xmlns:pcepspi="urn:opendaylight:params:xml:ns:yang:controller:pcep:spi">
              pcepspi:extension
            </type>
            <name>pcep-parser-my-vendor</name>
          </extension>
        </module>
        <module>
          <type
xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller:pcep:my:vendor:cfg">
            prefix:pcep-parser-my-vendor
          </type>
          <name>pcep-parser-my-vendor</name>
        </module>
      </modules>
      <services
xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
        <service>
          <type
xmlns:pcepspi="urn:opendaylight:params:xml:ns:yang:controller:pcep:spi">
            pcepspi:extension
          </type>
          <instance>
            <name>pcep-parser-my-vendor</name>
            <provider>/config/modules/module[name='pcep-parser-my-
vendor']/instance[name='pcep-parser-my-vendor']/provider</provider>
          </instance>
        </service>
```

```
        </services>  
    </data>  
    </configuration>  
</snapshot>
```


5. Controller

Table of Contents

| | |
|--|-----|
| OpenDaylight Controller: MD-SAL Developers' Guide | 41 |
| API types | 41 |
| Basic YANG concepts and their rendition in APIs | 41 |
| MD-SAL: Plugin types | 45 |
| Protocol library | 46 |
| MD-SAL: Southbound plugin development guide | 46 |
| Definition of YANG models | 47 |
| RPCs | 47 |
| Augmentations | 49 |
| Best practices | 49 |
| Implementation | 49 |
| Notifications | 50 |
| Best practices | 50 |
| OpenDaylight Controller: MD-SAL FAQs | 51 |
| OpenDaylight Controller Configuration: Java Code Generator | 58 |
| Service interfaces generating | 58 |
| Runtime beans generating | 59 |
| OpenDaylight Controller MD-SAL: Restconf | 59 |
| Mount point | 60 |
| Something practical | 64 |
| OpenDaylight Controller: Configuration | 67 |
| APIs and SPIs | 69 |
| OpenDaylight Controller configuration: Initial | 71 |
| Initial configuration for controller | 71 |
| OpenDaylight Controller configuration: config.ini | 72 |
| OpenDaylight Controller: Configuration Persister | 72 |
| Current configuration for controller distribution | 73 |
| Adding custom initial configuration | 79 |
| Persister Notification Handler | 83 |
| MD-SAL architecture: Clustering Notifications | 86 |
| MD-SAL Architecture: DOM | 87 |
| MD-SAL: Infinispan Data Store | 88 |
| State of the POC | 91 |
| Infinispan-related learnings | 92 |
| Datastore-related learnings | 92 |
| No clarity on the closing of Read-Only transactions | 92 |
| OpenDaylight Controller configuration: FAQs | 95 |
| OpenDaylight Controller configuration: Component map | 95 |
| OpenDaylight Controller: Netconf component map | 97 |
| OpenDaylight Controller Configuration: Examples sample project | 97 |
| OpenDaylight Controller: Configuration examples user guide | 109 |
| OpenDaylight Controller Configuration: Logback Examples | 118 |
| OpenDaylight Controller: Configuration Logback.xml | 125 |
| Configuration example of thread pools using yangcli-pro | 126 |
| Configuration example of thread pools using telnet | 126 |

| | |
|---|-----|
| Connecting to plaintext TCP socket | 126 |
| Configuring threadfactory | 127 |
| Configuring fixed threadpool | 130 |
| OpenDaylight Controller MD-SAL: Model reference | 131 |

OpenDaylight Controller: MD-SAL Developers' Guide

Model-Driven SAL (MD-SAL) is a set of infrastructure services aimed at providing common and generic support to application and plugin developers.

MD-SAL currently provides infrastructure services for the following:

- Data Services
- RPC or Service routing
- Notification subscription and publish services

This model-driven infrastructure allows developers to develop applications and plugins against an API type of their choice (Java generated APIs, DOM APIs, REST APIs). The infrastructure automatically provides the other API types. The modelling language of choice for MD-SAL is YANG, which is an IETF standard, for modelling network element configuration. The YANGTools project and its development tools provide support for YANG.

API types

MD-SAL provides three API types:

- Java generated APIs for consumers and producers
- DOM APIs: Mostly used by infrastructure components and useful for XML-driven plugin and application types
- REST APIs: [Restconf](#) that is available to consumer type applications and provides access to RPC and data stores

Basic YANG concepts and their rendition in APIs

The following are the basic concepts in YANG modeling:

- Remote Procedure (RPCs): In MD-SAL, RPCs are used for any call or invocation that crosses the plugin or module boundaries. RPCs are triggered by consumers, and usually have return values.
- Notifications: Asynchronous events, published by components for listeners.
- Configuration and Operational Data tree: The well-defined (by model) tree structure that represents the operational state of components and systems.
 - Instance Identifier: The path that uniquely identifies the sub-tree in the configuration or operational space. Most of the addressing of data is done by Instance Identifier.

RPC

In YANG, Remote Procedure Calls (RPCs) are used to model any procedure call implemented by a Provider (Server), which exposes functionality to Consumers (Clients).

In MD-SAL terminology, the term *RPC* is used to define the input and output for a procedure (function) that is to be provided by a Provider, and adapted by the MD-SAL.

In the context of the MD-SAL, there are three types of RPCs (RPC services):

- Global: One service instance (implementation) per controller container or mount point
- Routed: Multiple service instances (implementations) per controller container or mount point

Global service

- There is only one instance of a Global Service per controller instance. (Note that a controller instance can consist of a cluster of controller nodes.)

Routing

- Binding-Aware MD-SAL (sal-binding)
 - **Rpc Type:** Identified by a generated RpcService class and a name of a method invoked on that interface
- Binding-Independent MD-SAL (sal-dom)
 - **Rpc Type:** Identified by a QName

Routed service

- There can be multiple instances (implementations) of a service per controller instance
- Can be used for southbound plugins or for horizontal scaling (load-balancing) of northbound plugins (services)

Routing

Routing is done based on the contents of a message, for example, *Node Reference*. The field in a message that is used for routing is specified in a YANG model by using the routing-reference statement from the yang-ext model.

- Binding Aware MD-SAL (sal-binding)
- RPC Type: Identified by an RpcService subclass and the name of the method invoked on that interface
- Instance Identifier: In a data tree, identifies the element instance that will be used as the route target. The used class is:

```
org.opendaylight.yang.binding.InstanceIdentifier
```

The Instance Identifier is learned from the message payload and from the model.

- Binding Independent MD-SAL (sal-dom)
- RPC Type: Identified by a QName
- Instance Identifier: In a data tree, identifies the element instance that will be used as the route target. The used class is:

```
org.opendaylight.yang.data.api.InstanceIdentifier
```

RPCs in various API types:

- Java Generated APIs: For each model there is *Service interface. See [YANG Tools: Yang to Java mapping-RPC](#) to understand how YANG statements maps to Service interface.
 - Providers expose their implementation of *Service by registering their implementation to RpcProviderRegistry.
 - Consumers get the *Service implementation from RpcConsumerRegistry. If the implementer uses a different API type, MD-SAL automatically translates data in the background.
- DOM APIs: RPCs are identified by QName.
 - Providers expose their implementation of RPC identified by QName registering their RpcImplementation to RpcProvisionRegistry.
 - Consumers get the *Service implementation from RpcConsumerRegistry. If the implementer uses different API type, MD-SAL automatically translates data in the background.
- REST APIs: RPCs are identified by the model name and their name.
- Consumers invoke RPCs by invoking POST operation to /restconf/operations/model-name:rpc-name.

Notification

In YANG, Notifications represent asynchronous events, published by providers for listeners.

RPCs in various API types:

- Java Generated APIs: For each model, there is *Listener interface and transfer object for each notification. See [YANG Tools: Yang to Java mapping-Notification](#) to understand how YANG statements map to the Notifications interface.
 - Providers publish notifications by invoking the publish method on NotificationPublishService.
 - To receive notifications, consumers register their implementation of *Listener to NotificationBrokerService. If the notification publisher uses a different API type, MD-SAL automatically translates data in the background.
- DOM APIs: Notifications are represented only by XML Payload.
 - Providers publish notifications by invoking the publish method on NotificationPublishService.

- To receive notifications, consumers register their implementation of `*Listener` to `NotificationBrokerService`. If the notification publisher uses a different API type, MD-SAL automatically translates data in the background.
- REST APIs: Notifications are currently not supported.

Instance Identifier

The Instance Identifier is the unique identifier of an element (location) in the yang data tree: basically, it is the **path** to the node that uniquely identifies all the parent nodes of the node. The unique identification of list elements requires the specification of key values as well.

MD-SAL currently provides three different APIs to access data in the common data store:

- Binding APIs (Java generated DTOs)
- DOM APIs
- [OpenDaylight Controller:MD-SAL Restconf APIs](#)

Example

Consider the following simple YANG model for inventory:

```
module inventory {
  namespace "urn:.opendaylight:inventory";
  prefix inv;
  revision "2013-06-07";
  container nodes {
    list node {
      key "id";
      leaf "id" {
        type "string";
      }
    }
  }
}
```

An example having one instance of node with the name foo

Let us assume that we want to create an instance identifier for the node foo in the following bindings or formats:

- **YANG / XML / XPath version**

```
/inv:nodes/inv:node[id="foo"]
```

- **Binding-Aware version (generated APIs)**

```
import org.opendaylight.yang.gen.urn.opendaylight.inventory.rev130607.Nodes;
import org.opendaylight.yang.gen.urn.opendaylight.inventory.rev130607.nodes.
Node;
import org.opendaylight.yang.gen.urn.opendaylight.inventory.rev130607.nodes.
NodeKey;

import org.opendaylight.yangtools.yang.binding.InstanceIdentifier;
```

```
InstanceIdentifier<Node> identifier = InstanceIdentifier.builder(Nodes.class).  
child(Node.class,new NodeKey("foo")).toInstance();
```



Note

The last call, *toInstance()* does not return an instance of the node, but the Java version of Instance identifier which uniquely identifies the node **foo**.

- **HTTP Restconf APIs**

```
http://localhost:8080/restconf/config/inventory:nodes/node/foo
```



Note

We assume that HTTP APIs are exposed on localhost, port 8080.

- **Binding Independent version (yang-data-api)**

```
import org.opendaylight.yang.common.QName;  
import org.opendaylight.yang.data.api.InstanceIdentifier;  
  
QName nodes = QName.create("urn:opendaylight:inventory", "2013-06-07", "nodes");  
QName node = QName.create(nodes, "nodes");  
QName idName = QName.create(nodes, "id");  
InstanceIdentifier = InstanceIdentifier.builder()  
    .node(nodes)  
    .nodeWithKey(node, idName, "foo")  
    .toInstance();
```



Note

The last call, *toInstance()* does not return an instance of node, but the Java version of Instance identifier which uniquely identifies the node **foo**.

MD-SAL: Plugin types

MD-SAL has four component-types that differ in complexity, expose different models, and use different subsets of the MD-SAL functionality.

- **Southbound Protocol Plugin:** Responsible for handling multiple sessions to the southbound network devices and providing common abstracted interface to access various type of functionality provided by these network devices
- **Manager-type application:** Responsible for managing the state and the configuration of a particular functionality which is exposed by southbound protocol plugins
- **Protocol Library:** Responsible for handling serialization or de-serialization between the wire protocol format and the Java form of the protocol
- **Connector Plugin:** Responsible for connecting consumers (and providers) to Model-driven SAL (and other components) by means of different wire protocol or set of APIs

Southbound protocol plugin

The responsibilities of the Southbound Protocol plugin include the following :

- Handling multiple sessions to southbound network devices
- Providing a common abstracted interface to access various type of functionality provided by the network devices

The Southbound Protocol Plugin should be stateless. The only preserved state (which is still transient) is the list of connected devices or sessions. Models mostly use RPCs and Notifications to describe plugin functionality. Example plugins: Openflow Southbound Plugin, Netconf Southbound Plugin, BGP Southbound Plugin, and PCEP Southbound Plugin.

Manager-type application

The responsibilities of the Manager-type applications include the following:

- Providing configuration-like functionality to set or modify the behaviour of network elements or southbound plugins
- Coordinating flows and provide higher logic on top of stateless southbound plugins

Manager-type Applications preserve state. Models mostly use Configuration Data and Runtime Data to describe component functionality.

Protocol library

The OpenFlow Protocol Library is a component in OpenDaylight, that mediates communication between the OpenDaylight controller and the hardware devices supporting the OpenFlow protocol. The primary goal of the library is to provide user (or upper layers of OpenDaylight) communication channel, that can be used for managing network hardware devices.

MD-SAL: Southbound plugin development guide

The southbound controller plugin is a functional component.

The plugin:

- Provides an abstraction of network devices functionality
- Normalizes their APIs to common contracts
- Handles session and connections to them

The plugin development process generally moves through the following phases:

1. Definition of YANG models (API contracts): For Model-Driven SAL, the API contracts are defined by YANG models and the Java interfaces generated for these models. A developers opts for one of the following:
 - Selects from existing models
 - Creates new models
 - Augments (extends) existing models

2. Code Generation: The Java Interfaces, implementation of Transfer Objects, and mapping to Binding-Independent form is generated for the plugin. This phase requires the proper configuration of the Maven build and YANG Maven Tools.
3. Implementation of plugin: The actual implementation of the plugin functionality and plugin components.



Note

The order of steps is not definitive, and it is up to the developer to find the most suitable workflow. For additional information, see [the section called “Best practices” \[49\]](#).

Definition of YANG models

In this phase, the developer selects from existing models (provided by controller or other plugins), writes new models, or augments existing ones. A partial list of available models could be found at: [YANG Tools:Available Models](#).

The mapping of YANG to Java is documented at: [Yang Tools:YANG to Java Mapping](#). This mapping provides an overview of how YANG is mapped to Java.

Multiple approaches to model the functionality of the southbound plugin are available:

- Using RPCs and Notifications
- Using Configuration Data Description
- Using Runtime Data Description
- Combining approaches

RPCs

RPCs can model the functionality invoked by consumers (applications) that use the southbound plugin. Although RPCs can model any functionality, they are usually used to model functionality that cannot be abstracted as configuration data, for example, PacketOut, or initiating a new session to a device (controller-to-device session).

RPCs are modeled with an RPC statement in the following form: `rpc foo { }` This statement is mapped to method.

RPC input To define RPC input, use an input statement inside RPC. The structure of the input is defined with the same statements as the structure of notifications, configuration, and so on.

```
rpc foo {  
  input {  
    ...  
  }  
}
```

RPC output To define the RPC output (structure of result), use the RPC output statement.


```
rpc foo {  
  output {  
    ...  
  }  
}
```

Notifications Use notifications to model events originating in a network device or southbound plugin which is exposed to consumers to listen.

A notification statement defines a notification:

```
notification foo {  
  ...  
}
```

Configuration data

Configuration data is good for the following purposes:

- Model or provide CRUD access to the state of protocol plugin and/or network devices
- Model any functionality which could be exposed as a configuration to the consumers or applications

Configuration data in YANG is defined by using the config substatement with a true argument. For example:

```
container foo {  
  config true;  
  ...  
}
```

Runtime (read-only) data Runtime (read-only) data is good to model or provide read access to the state of the protocol plugin and network devices, or network devices. This type of data is good to model statistics or any state data, which cannot be modified by the consumers (applications), but needs exposure (for example, learned topology, or list of connected switches).

Runtime data in YANG is defined by using config substatement with a false argument:

```
container foo {  
  config false;  
}
```

Structural elements The structure of RPCs, notifications, configuration data, and runtime data is modelled using structural elements (data schema nodes). Structural elements define the actual structure of XML, DataDOM documents, and Java APIs for accessing or storing these elements. The most commonly used structural elements are:

- Container
- List
- Leaf
- Leaf-list
- Choice

Augmentations

Augmentations are used to extend existing models by providing additional structural elements and semantics. Augmentation cannot change the mandatory status of nodes in the original model, or introduce any new mandatory statements.

Best practices

- YANG models must be located under the `src/main/yang` folder in your project.
- Design your models so that they are reusable and extendible by third-parties.
- Always try to reuse existing models and types provided by these models. See [YANG Tools:Available Models](#) or others if there is no model which provides you with data structures and types you need.

Code generation To configure your project for code generation, your build system needs to use Maven. For the configuration of java API generation, see [Yang Tools:Maven Plugin Guide](#).

Artefacts generated at compile time The following artefacts are generated at compile time:

- Service interfaces
- Transfer object interfaces
- Builders for transfer objects and immutable versions of transfer objects

Implementation

This step uses generated artefacts to implement the intended functionality of the southbound plugin.

Provider implementation To expose functionality through binding-awareness, the MD-SAL plugin needs to be compiled against these APIs, and must at least implement the `BindingAwareProvider` interface. The provider uses APIs which are available in the `SAL-binding-api` Maven artifact. To use this dependency, insert the following dependency into your `pom.xml`:

```
<dependency>
  <groupId>org.opendaylight.controller</groupId>
  <artifactId>sal-binding-api</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

BindingAwareProvider implementation A `BindingAwareProvider` interface requires the implementation of four methods, and registering an instance with `BindingAwareBroker`. Use `AbstractBindingAwareProvider` to simplify the implementation.

- `void onSessionInitialized(ConsumerContext ctx)`: This callback is called when `Binding-Aware Provider` is initialized and `ConsumerContext` is injected into it. `ConsumerContext` serves to access all functionality which the plugin is to consume from other controller components.

- `void onSessionInitialized(ProviderContext ctx)`: This callback is called when Binding-Aware Provider is initialized and `ProviderContext` is injected into it. `ProviderContext` serves to access all functionality which the plugin could use to provide its functionality to controller components.
- `Collection<? extends RpcService> getImplementations()`: Shorthand registration of an already instantiated implementations of global RPC services. Automated registration is currently not supported.
- `public Collection<? extends ProviderFunctionality> getFunctionality()`: Shorthand registration of an already instantiated implementations of `ProviderFunctionality`. Automated registration is currently not supported. NOTE: You also need to set your implementation of `AbstractBindingAwareProvider` set as Bundle Activator for MD-SAL to properly load it.

Notifications

To publish events, request an instance of `NotificationProviderService` from `ProviderContext`. Use the following:

```
ExampleNotification notification = (new ExampleNotificationBuilder()).
build();
NotificationProviderService notificationProvider = providerContext.
getSALService(NotificationProviderService.class);
notificationProvider.notify(notification);
```

RPC implementations To implement the functionality exposed as RPCs, implement the generated `RpcService` interface. Register the implementation within `ProviderContext` included in the provider.

If the generated `RpcInterface` is `FooService`, and the implementation is `FooServiceImpl`:

```
@Override
public void onSessionInitiated(ProviderContext context) {
    context.addRpcImplementation(FooService.class, new FooServiceImpl());
}
```

Best practices

RPC Service interface contract requires you to return [Future object](#) (to make it obvious that call may be asynchronous), but it is not specified how this Future is implemented. Consider using existing implementations provided by JDK or Google Guava. Implement your own Future only if necessary.

Consider using [SettableFuture](#) if you intend not to use [FutureTask](#) or submit [Callables](#) to [ExecutorService](#).



Important

Do not implement transfer object interfaces unless necessary. Choose already generated builders and immutable versions. If you want to implement transfer objects, ensure that instances exposed outside the plugin are immutable.

OpenDaylight Controller: MD-SAL FAQs

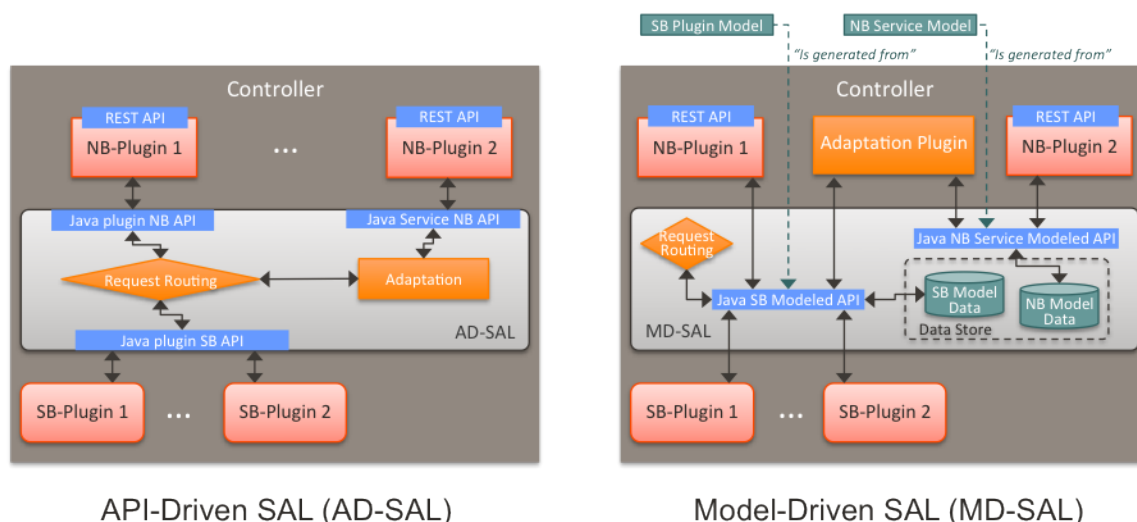
Q-1: What is the overall MD-SAL architecture?

- What is the overall architecture, components, and functionality?
- Who supplies which components, and how are the components plumbed?

A-1: The overall Model-Driven SAL (MD-SAL) architecture did not really change from the API-Driven SAL (AD-SAL). As with the AD-SAL, plugins can be data providers, or data consumers, or both (although the AD-SAL did not explicitly name them as such). Just like the AD-SAL, the MD-SAL connects data consumers to appropriate data providers and (optionally) facilitates data adaptation between them.

Now, in the AD-SAL, the SAL APIs request routing between consumers and providers, and data adaptations are all statically defined at compile or build time. In the MD-SAL, the SAL APIs and request routing between consumers and providers are defined from models, and data adaptations are provided by *internal* adaptation plugins. The API code is generated from models when a plugin is compiled. When the plugin OSGI bundle is loaded into the controller, the API code is loaded into the controller along with the rest of the plugin containing the model.

Figure 5.1. AD-SAL and MD-SAL



The AD-SAL provides request routing (selects an SB plugin based on service type) and optionally provides service adaptation, if an NB (Service, abstract) API is different from its corresponding SB (protocol) API. For example, in the above figure, the AD-SAL routes requests from NB-Plugin 1 to SB Plugins 1 and 2. Note that the plugin SB and NB APIs in this example are essentially the same (although both of them need to be defined). Request routing is based on plugin type: the SAL knows which node instance is served by which plugin. When an NB Plugin requests an operation on a given node, the request is routed to the appropriate plugin which then routes the request to the appropriate node. The AD-SAL can also provide service abstractions and adaptations. For example, in the above figure, NB Plugin 2 is using an abstract API to access the services provided by SB Plugins 1 and 2. The translation between the SB Plugin API and the abstract NB API is done in the Abstraction module in the AD-SAL.

The MD-SAL provides request routing and the infrastructure to support service adaptation. However, it does not provide service adaptation itself: service adaptation is provided by plugins. From the point of view of MD-SAL, the Adaptation Plugin is a regular plugin. It provides data to the SAL, and consumes data from the SAL through APIs generated from models. An Adaptation Plugin basically performs model-to-model translations between two APIs. Request Routing in the MD-SAL is done on both protocol type and node instances, since node instance data is exported from the plugin into the SAL (the model data contains routing information).

The simplest MD-SAL APIs generated from models (RPCs and Notifications, both supported in the yang modeling language) are functionally equivalent to AD-SAL function call APIs. Additionally, the MD-SAL can store data for models defined by plugins. Provider and consumer plugins can exchange data through the MD-SAL storage. Data in the MD-SAL is accessed through getter and setter APIs generated from models. Note that this is in contrast to the AD-SAL, which is stateless.

Note that in the above figure, both NB AD-SAL Plugins provide REST APIs to controller client applications.

The functionality provided by the MD-SAL is basically to facilitate the plumbing between providers and consumers. A provider or a consumer can register itself with the MD-SAL. A consumer can find a provider that it is interested in. A provider can generate notifications; a consumer can receive notifications and issue RPCs to get data from providers. A provider can insert data into SAL storage; a consumer can read data from SAL storage.

Note that the structure of SAL APIs is different in the MD-SAL from that in the AD-SAL. The AD-SAL typically has both NB and SB APIs even for functions or services that are mapped 1:1 between SB Plugins and NB Plugins. For example, in the current AD-SAL implementation of the OpenFlow Plugin and applications, the NB SAL APIs used by OF applications are mapped 1:1 onto SB OF Plugin APIs. The MD-SAL allows both the NB plugins and SB plugins to use the same API generated from a model. One plugin becomes an API (service) provider; the other becomes an API (service) Consumer. This eliminates the need to define two different APIs and to provide three different implementations even for cases where APIs are mapped to each other 1:1. The MD SAL provides instance-based request routing between multiple provider plugins.

Q-2: What functionality does the MD-SAL assume? For example, does the SAL assume that the network model is a part of the SAL?

A-2: The MD-SAL does not assume any model. All models are provided by plugins. The MD-SAL only provides the infrastructure and the plumbing for the plugins.

Q-3: What is the "day in the life" of an MD-SAL plugin?

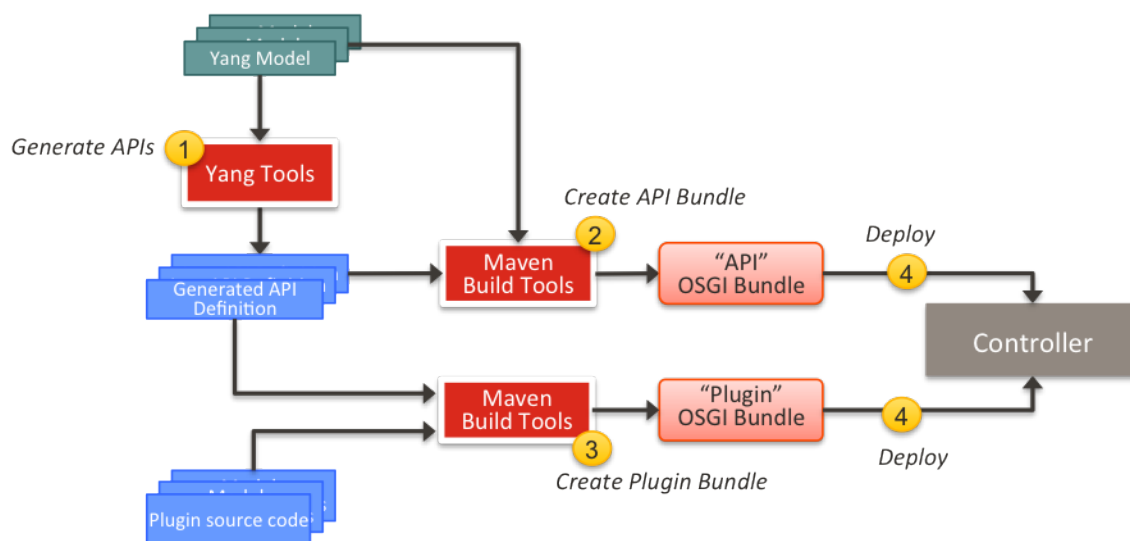
A-3: All plugins (protocol, application, adaptation, and others) have the same lifecycle. The life of a plugin has two distinct phases: design and operation. During the design phase, the plugin designer performs the following actions:

- The designer decides which data will be consumed by the plugin, and imports the SAL APIs generated from the API provider's models. Note that the topology model is just one possible data type that may be consumed by a plugin. The list of currently available data models and their APIs can be found in YANG_Tools:Available_Models.

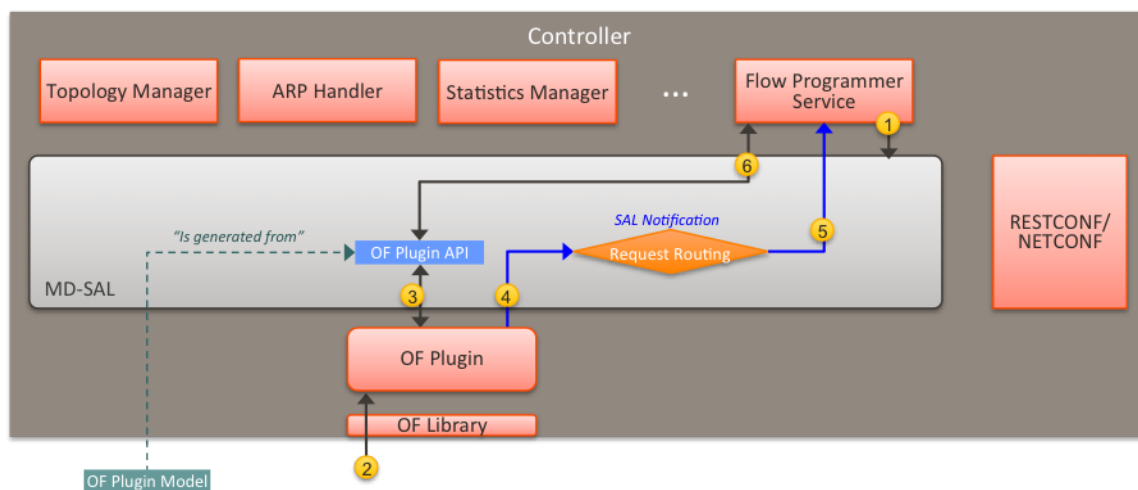
- The designer decides which data and how it will be provided by the plugin, and designs the data model for the provided data. The data model (expressed in yang) is then run through the **YANG Tools**, which generate the SAL APIs for the model.
- The implementations for the generated consumer and provider APIs, along with other plugin features and functionality, are developed. The resulting code is packaged in a "plugin" OSGI bundle. Note that a developer may package the code of a subsystem in multiple plugins or applications that may communicate with each other through the SAL.
- The generated APIs and a set of helper classes are also built and packaged in an "API" OSGI bundle.

The plugin development process is shown in the following figure.

Figure 5.2. Plugin development process



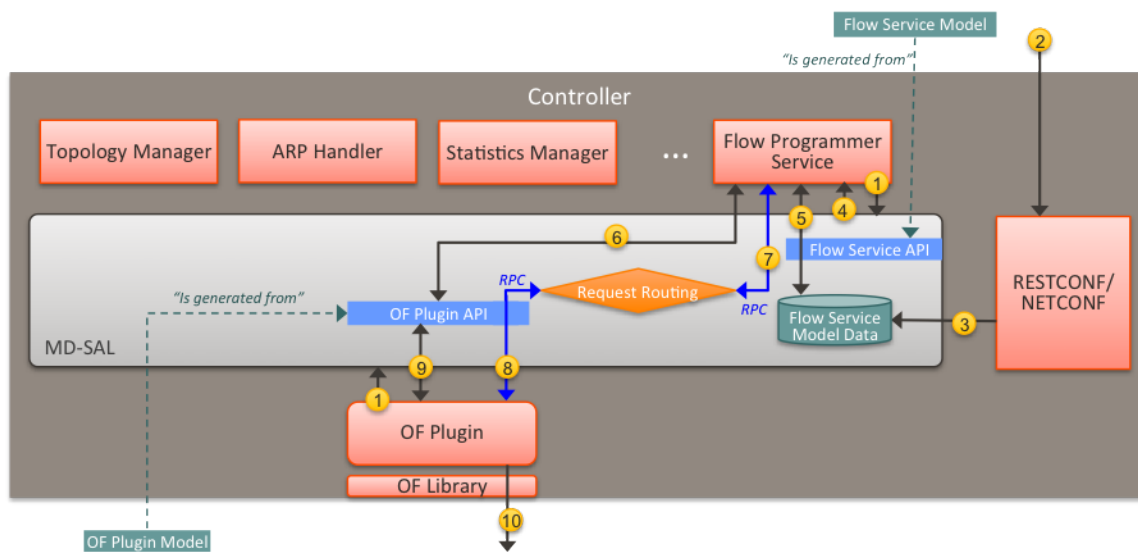
When the OSGI bundle of a plugin is loaded into the controller and activated, the operation phase begins. The plugin operation is probably best explained with a few examples describing the operation of the OF Protocol plugin and OF applications, such as the Flow Programmer Service, the ARP Handler, or the Topology Manager. The following figure shows a scenario where a "Flow Deleted" notification from a switch arrives at the controller.

Figure 5.3. Flow deleted at controller

The scenario is as follows:

1. The Flow Programmer Service registers with the MD SAL for the 'Flow Deleted' notification. This is done when the Controller and its plugins or applications are started.
2. A 'Flow Deleted' OF packet arrives at the controller. The OF Library receives the packet on the TCP/TLS connection to the sending switch, and passes it to the OF Plugin.
3. The OF Plugin parses the packet, and uses the parsed data to create a 'Flow Deleted' SAL notification. The notification is actually an immutable 'Flow Deleted' Data Transfer Object (DTO) that is created or populated by means of methods from the model-generated OF Plugin API.
4. The OF Plugin sends the 'Flow Deleted' SAL notification (containing the notification DTO) into the SAL. The SAL routes the notification to registered consumers, in this case, the Flow Programmer Service.
5. The Flow Programmer Service receives the notification containing the notification DTO.
6. The Flow Programmer Service uses methods from the API of the model-generated OF Plugin to get data from the immutable notification DTO received in Step 5. The processing is the same as in the AD-SAL.

Note that other packet-in scenarios, where a switch punts a packet to the controller, such as an ARP or an LLDP packet, are similar. Interested applications register for the respective notifications. The OF plugin generates the notification from received OF packets, and sends them to the SAL. The SAL routes the notifications to the registered recipients. The following figure shows a scenario where an external application adds a flow by means of the NB REST API of the controller.

Figure 5.4. External app adds flow

The scenario is as follows:

1. Registrations are performed when the Controller and its plugins or applications are started.
 - a. The Flow Programmer Service registers with the MD SAL for Flow configuration data notifications.
 - b. The OF Plugin registers (among others) the 'AddFlow' RPC implementation with the SAL. Note that the RPC is defined in the OF Plugin model, and the API is generated during build time.
2. A client application requests a flow add through the REST API of the Controller. (Note that in the AD-SAL, there is a dedicated NB REST API on top of the Flow Programming Service. The MD-SAL provides a common infrastructure where data and functions defined in models can be accessed by means of a common REST API. For more information, see <http://datatracker.ietf.org/doc/draft-bierman-netconf-restconf/>). The client application provides all parameters for the flow in the REST call.
3. Data from the 'Add Flow' request is deserialized, and a new flow is created in the Flow Service configuration data tree. (Note that in this example the configuration and operational data trees are separated; this may be different for other services). Note also that the REST call returns success to the caller as soon as the flow data is written to the configuration data tree.
4. Since the Flow Programmer Service is registered to receive notifications for data changes in the Flow Service data tree, the MD-SAL generates a 'data changed' notification to the Flow Programmer Service.
5. The Flow Programmer Service reads the newly added flow, and performs a flow add operation (which is basically the same as in the AD-SAL).
6. At some point during the flow addition operation, the Flow Programmer Service needs to tell the OF Plugin to add the flow in the appropriate switch. The Flow Programmer

Service uses the OF Plugin generated API to create the RPC input parameter DTO for the "AddFlow" RPC of the OF Plugin.

7. The Flow Programmer Service gets the service instance (actually, a proxy), and invokes the "AddFlow" RPC on the service. The MD-SAL will route the request to the appropriate OF Plugin (which implements the requested RPC).
8. The 'AddFlow' RPC request is routed to the OF Plugin, and the implementation method of the "AddFlow" RPC is invoked.
9. The 'AddFlow' RPC implementation uses the OF Plugin API to read values from the DTO of the RPC input parameter. (Note that the implementation will use the getter methods of the DTO generated from the yang model of the RPC to read the values from the received DTO.)
10. The 'AddFlow' RPC is further processed (pretty much the same as in the AD-SAL) and at some point, the corresponding flowmod is sent to the corresponding switch.

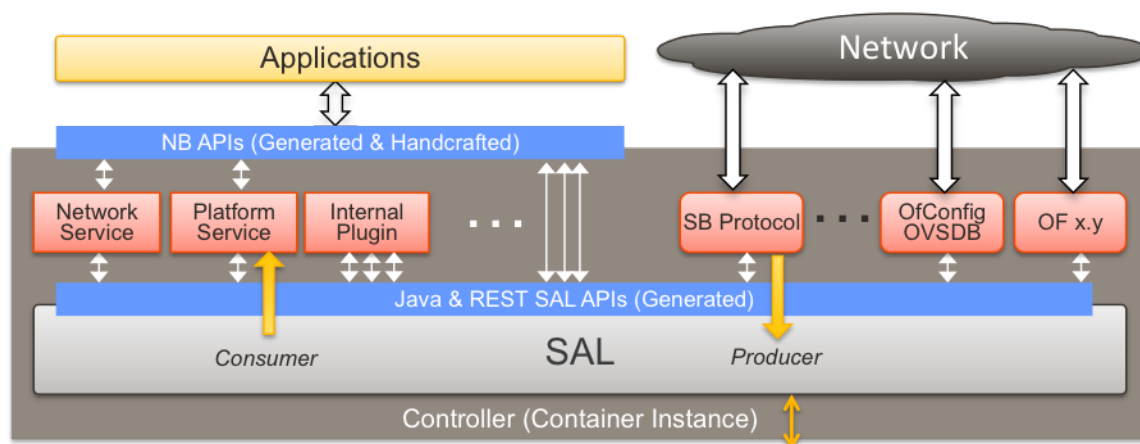
Q-4: Is there a document that describes how code is generated from the models for the MD-SAL?

A-4: [Yangtools](#) documents the Yang to Java generation, including examples of how the yang constructs are mapped into Java classes. You can write unit tests against the generated code. You will have to write implementations of the generated RPC interfaces. The generated code is just Java, and it debugs just like Java.

If you want to play with generating Java from Yang there is a maven archetype to help you get going: [Maven Archetypes: ODL Model Project](#). Or, you can try creating a project in Eclipse as explained at: [YANG to Java conversion: How to create Maven project in Eclipse](#).

Q-5: The code generation tools mention producers and consumers'. How are these related to southbound and 'northbound SAL plugins?

A-5: The difference between southbound and northbound plugins is that the southbound plugins talk protocols to network nodes, and northbound plugins talk application APIs to the controller applications. As far as the SAL is concerned, there is really no north or south. The SAL is basically a data exchange and adaptation mechanism between plugins. The plugin SAL roles (consumer or producer) are defined with respect to the data being moved around or stored by the SAL. A producer implements an API, and provides the data of the API: a consumer uses the API, and consumes the data of the API. While *northbound* and *southbound* provide a network engineer's view of the SAL, *consumer* and *producer* provide a software engineer's view of the SAL, and is shown in the following figure:

Figure 5.5. SAL consumer and producer view

Q-6: Where can I find models that have already been defined in OpenDaylight?

A-6: The list of models that have been defined for the SAL and in various plugins can be found in [MD-SAL Model Reference](#).

Q-7: How do I migrate my existing plugins and services to MD-SAL?

A-7: The migration guide can be found in the [MD-SAL Application Migration Guide](#).

Q-8: Where can I find SAL example code?

A-8: The toaster sample provides a simple yet complete example of a model, a service provider (toaster), and a service consumer. It provides the model of a programmable toaster, a sample consumer application that uses MD-SAL APIs; a sample southbound plugin (a service provider) that implements toaster; and a unit test suite.

The toaster example is in *controller.git* under *.opendaylight/md-sal/samples*.

Q-9: Where is the REST API code for the example?

A-9: The REST APIs are derived from models. You do not have to write any code for it. The controller will implement the [RESTCONF protocol](#) which defines access to yang-formatted data through REST. Basically, all you need to do is define your service in a model, and expose that model to the SAL. REST access to your modeled data will then be provided by the SAL infrastructure. However, if you want to, you can create your own REST API (for example, to be compliant with an existing API).

Q-10: How can one use RESTCONF to access the MD-SAL datastore?

A-10: For information on accessing the MD-SAL datastore, see [MD-SAL Restconf](#).

OpenDaylight Controller Configuration: Java Code Generator

YANG to Java code generator

The Java code for the configuration system is generated by the yang-maven-plugin and the yang-jmx-generator-plugin. The input Yang module files are converted to java files by the definition of the module and the specified templates. the generated java code can represent interfaces, classes, or abstract classes used for configuration.

Service interfaces generating

Service interfaces (SI) are generated from YANG "service-types". Each SI must be defined as "identity" with a "base" statement set to "config:service-type", or another SI. This is because a service must have a globally unique name. Each SI must be annotated with @ServiceInterfaceAnnotation, and must extend AbstractServiceInterface.

Sample YANG module representing service interface

```
module config-test {
  yang-version 1;
  namespace "urn:opendaylight:params:xml:ns:yang:controller:test";
  prefix "test";

  import config { prefix config; revision-date 2013-04-05; }

  description
    "Testing API";

  revision "2013-06-13" {
    description
      "Initial revision";
  }

  identity testing {
    description
      "Test api";

    base "config:service-type";
    config:java-class "java.lang.AutoCloseable";
  }
}
```

The "description" node of identity is generated as javadoc in the service interface. The "config:java-class" is generated as **ServiceInterfaceAnnotation**. It specifies java classes or interfaces in the "osgiRegistrationTypes" parameter. The module implementing this service interface must instantiate a java object that can be cast to any of the java types defined in "osgiRegistrationTypes".

Generated java source file: AutoCloseableServiceInterface

```
package %prefix%.test;
```

```
/**
 * Test api
 */
@org.opendaylight.controller.config.api.annotations.Description(value = "Test
api")
@org.opendaylight.controller.config.api.annotations.
ServiceInterfaceAnnotation(value = "testing", osgiRegistrationType = java.
lang.AutoCloseable.class)
public interface AutoCloseableServiceInterface extends org.opendaylight.
controller.config.api.annotations.AbstractServiceInterface
{
}
```

Module stubs generating

Modules are constructed during configuration transaction. A module implements the ModuleMXBean interface. The ModuleMXBean interface represents getters and setters for attributes that will be exposed to the configuration registry by means of JMX. Attributes can either be simple types, or composite types.

Each ModuleMXBean must be defined in yang as "identity" with the base statement set to "config:module-type". Not only are ModuleMXBeans generated, but also ModuleFactory and Module stubs. Both are first generated as abstract classes with almost full functionality. Then their implementations, which are allowed to be modified by users, are generated, but only once.

Runtime beans generating

Runtime JMX beans are purposed to be the auditors: they capture data about running module instances. A module can have zero or more runtime beans. Runtime beans are hierarchically ordered, and each must be uniquely identified. A runtime bean is defined as a configuration augment of a module, from which interface RuntimeMXBean, RuntimeRegistrator, and RuntimeRegistretion are generated. Augment definition contains arguments representing the data of a module that must be watched.

RPCs

Method calls in yang must be specified as top level elements. The context, where an RPC operation exists, must be defined in the RPC definition itself, and in the runtime bean that provides method implementation.

OpenDaylight Controller MD-SAL: Restconf

Restconf operations overview

Restconf allows access to datastores in the controller. There are two datastores:

- Config: Contains data inserted via controller
- Operational: Contains other data



Note

Each request must start with the URI `/restconf`. Restconf listens on port 8080 for HTTP requests.

Restconf supports **OPTIONS**, **GET**, **PUT**, **POST**, and **DELETE** operations. Request and response data can either be in the XML or JSON format. XML structures according to yang are defined at: [XML-YANG](#). JSON structures are defined at: [JSON-YANG](#). Data in the request must have a correctly set **Content-Type** field in the http header with the allowed value of the media type. The media type of the requested data has to be set in the **Accept** field. Get the media types for each resource by calling the OPTIONS operation. Most of the paths of the Restconf endpoints use [Instance Identifier](#). `<identifier>` is used in the explanation of the operations.

`<identifier>`

- It must start with `<moduleName>:<nodeName>` where `<moduleName>` is a name of the module and `<nodeName>` is the name of a node in the module. It is sufficient to just use `<nodeName>` after `<moduleName>:<nodeName>`. Each `<nodeName>` has to be separated by `/`.
- `<nodeName>` can represent a data node which is a list or container yang built-in type. If the data node is a list, there must be defined keys of the list behind the data node name for example, `<nodeName>/<valueOfKey1>/<valueOfKey2>`.
- The format `<moduleName>:<nodeName>` has to be used in this case as well: Module A has node A1. Module B augments node A1 by adding node X. Module C augments node A1 by adding node X. For clarity, it has to be known which node is X (for example: C:X). For more details about encoding, see: [Restconf 02 - Encoding YANG Instance Identifiers in the Request URI](#).

Mount point

A Node can be behind a mount point. In this case, the URI has to be in format `<identifier>/yang-ext:mount/<identifier>`. The first `<identifier>` is the path to a mount point and the second `<identifier>` is the path to a node behind the mount point. A URI can end in a mount point itself by using `<identifier>/yang-ext:mount`. More information on how to actually use mountpoints is available at: [OpenDaylight Controller:Config:Examples:Netconf](#).

HTTP methods

OPTIONS /restconf

- Returns the XML description of the resources with the required request and response media types in Web Application Description Language (WADL)

GET /restconf/config/<identifier>

- Returns a data node from the Config datastore.
- `<identifier>` points to a data node which must be retrieved.

GET /restconf/operational/<identifier>

- Returns the value of the data node from the Operational datastore.
- <identifier> points to a data node which must be retrieved.

PUT /restconf/config/<identifier>

- Updates or creates data in the Config datastore and returns the state about success.
- <identifier> points to a data node which must be stored.

Example:

```
PUT http://<controllerIP>:8080/restconf/config/module1:foo/bar
Content-Type: applicaton/xml
<bar>
  ...
</bar>
```

Example with mount point:

```
PUT http://<controllerIP>:8080/restconf/config/module1:foo1/foo2/yang-
ext:mount/module2:foo/bar
Content-Type: applicaton/xml
<bar>
  ...
</bar>
```

POST /restconf/config

- Creates the data if it does not exist

For example:

```
POST URL: http://localhost:8080/restconf/config/
content-type: application/yang.data+json
JSON payload:

{
  "toaster:toaster" :
  {
    "toaster:toasterManufacturer" : "General Electric",
    "toaster:toasterModelNumber" : "123",
    "toaster:toasterStatus" : "up"
  }
}
```

POST /restconf/config/<identifier>

- Creates the data if it does not exist in the Config datastore, and returns the state about success.
- <identifier> points to a data node where data must be stored.
- The root element of data must have the namespace (data are in XML) or module name (data are in JSON.)

Example:

```
POST http://<controllerIP>:8080/restconf/config/module1:foo
Content-Type: applicaton/xml/
<bar xmlns="module1namespace">
  ...
</bar>
```

Example with mount point:

```
http://<controllerIP>:8080/restconf/config/module1:foo1/foo2/yang-ext:mount/
module2:foo
Content-Type: applicaton/xml
<bar xmlns="module2namespace">
  ...
</bar>
```

POST /restconf/operations/<moduleName>:<rpcName>

- Invokes RPC.
- <moduleName>:<rpcName> - <moduleName> is the name of the module and <rpcName> is the name of the RPC in this module.
- The Root element of the data sent to RPC must have the name "input".
- The result can be the status code or the retrieved data having the root element "output".

Example:

```
POST http://<controllerIP>:8080/restconf/operations/module1:fooRpc
Content-Type: applicaton/xml
Accept: applicaton/xml
<input>
  ...
</input>
```

The answer from the server could be:

```
<output>
  ...
</output>
```

An example using a JSON payload:

```
POST http://localhost:8080/restconf/operations/toaster:make-toast
Content-Type: application/yang.data+json
{
  "input" :
  {
    "toaster:toasterDoneness" : "10",
    "toaster:toasterToastType": "wheat-bread"
  }
}
```

**Note**

Even though this is a default for the toasterToastType value in the yang, you still need to define it.

DELETE /restconf/config/<identifier>

- Removes the data node in the Config datastore and returns the state about success.
- <identifier> points to a data node which must be removed.

More information is available in the [Restconf RFC](#).

How Restconf works

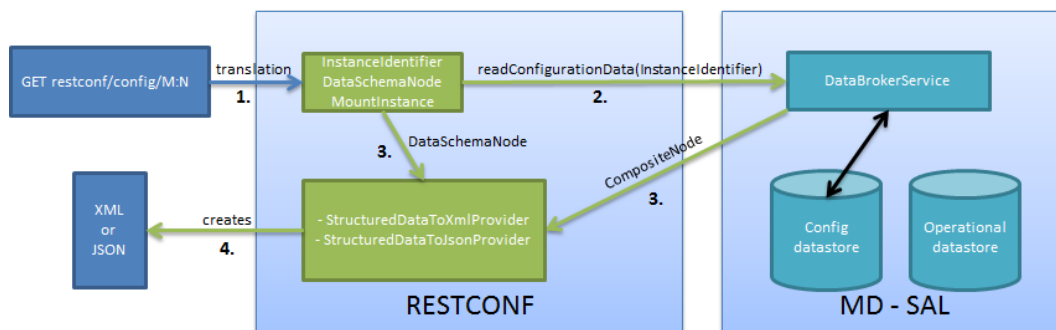
Restconf uses these base classes:

| | |
|--------------------|---|
| InstanceIdentifier | Represents the path in the data tree |
| ConsumerSession | Used for invoking RPCs |
| DataBrokerService | Offers manipulation with transactions and reading data from the datastores |
| SchemaContext | Holds information about yang modules |
| MountService | Returns MountInstance based on the InstanceIdentifier pointing to a mount point |
| MountInstance | Contains the SchemaContext behind the mount point |
| DataSchemaNode | Provides information about the schema node |
| SimpleNode | Possesses the same name as the schema node, and contains the value representing the data node value |
| CompositeNode | Can contain CompositeNode-s and SimpleNode-s |

GET in action

Figure 1 shows the GET operation with URI restconf/config/M:N where M is the module name, and N is the node name.

Figure 5.6. Get



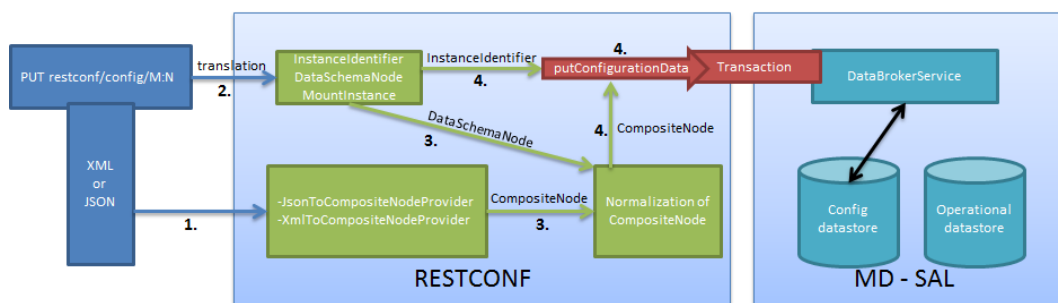
1. The requested URI is translated into the InstanceIdentifier which points to the data node. During this translation, the DataSchemaNode that conforms to the data node is obtained. If the data node is behind the mount point, the MountInstance is obtained as well.

2. Restconf asks for the value of the data node from DataBrokerService based on InstanceIdentifier.
3. DataBrokerService returns CompositeNode as data.
4. StructuredDataToXmlProvider or StructuredDataToJsonProvider is called based on the **Accept** field from the http request. These two providers can transform CompositeNode regarding DataSchemaNode to an XML or JSON document.
5. XML or JSON is returned as the answer on the request from the client.

PUT in action

Figure 2 shows the PUT operation with the URI `restconf/config/M:N` where M is the module name, and N is the node name. Data is sent in the request either in the XML or JSON format.

Figure 5.7. Put



1. Input data is sent to JsonToCompositeNodeProvider or XmlToCompositeNodeProvider. The correct provider is selected based on the Content-Type field from the http request. These two providers can transform input data to CompositeNode. However, this CompositeNode does not contain enough information for transactions.
2. The requested URI is translated into InstanceIdentifier which points to the data node. DataSchemaNode conforming to the data node is obtained during this translation. If the data node is behind the mount point, the MountInstance is obtained as well.
3. CompositeNode can be normalized by adding additional information from DataSchemaNode.
4. Restconf begins the transaction, and puts CompositeNode with InstanceIdentifier into it. The response on the request from the client is the status code which depends on the result from the transaction.

Something practical

1. Create a new flow on the switch openflow:1 in table 2.

HTTP request

```

Operation: POST
URI: http://192.168.11.1:8080/restconf/config/opendaylight-inventory:nodes/
node/openflow:1/table/2

```

Content-Type: application/xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flow
  xmlns="urn:opendaylight:flow:inventory">
  <strict>false</strict>
  <instructions>
    <instruction>
      <order>1</order>
      <apply-actions>
        <action>
          <order>1</order>
          <flood-all-action/>
        </action>
      </apply-actions>
    </instruction>
  </instructions>
  <table_id>2</table_id>
  <id>111</id>
  <cookie_mask>10</cookie_mask>
  <out_port>10</out_port>
  <installHw>false</installHw>
  <out_group>2</out_group>
  <match>
    <ethernet-match>
      <ethernet-type>
        <type>2048</type>
      </ethernet-type>
    </ethernet-match>
    <ipv4-destination>10.0.0.1/24</ipv4-destination>
  </match>
  <hard-timeout>0</hard-timeout>
  <cookie>10</cookie>
  <idle-timeout>0</idle-timeout>
  <flow-name>FooXf22</flow-name>
  <priority>2</priority>
  <barrier>false</barrier>
</flow>
```

HTTP response

Status: 204 No Content

1. Change *strict* to *true* in the previous flow.

HTTP request

Operation: PUT

URI: http://192.168.11.1:8080/restconf/config/opendaylight-inventory:nodes/
node/openflow:1/table/2/flow/111

Content-Type: application/xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flow
  xmlns="urn:opendaylight:flow:inventory">
  <strict>true</strict>
  <instructions>
    <instruction>
      <order>1</order>
      <apply-actions>
        <action>
```

```
        <order>1</order>
        <flood-all-action/>
      </action>
    </apply-actions>
  </instruction>
</instructions>
<table_id>2</table_id>
<id>111</id>
<cookie_mask>10</cookie_mask>
<out_port>10</out_port>
<installHw>false</installHw>
<out_group>2</out_group>
<match>
  <ethernet-match>
    <ethernet-type>
      <type>2048</type>
    </ethernet-type>
  </ethernet-match>
  <ipv4-destination>10.0.0.1/24</ipv4-destination>
</match>
<hard-timeout>0</hard-timeout>
<cookie>10</cookie>
<idle-timeout>0</idle-timeout>
<flow-name>FooXf22</flow-name>
<priority>2</priority>
<barrier>false</barrier>
</flow>
```

HTTP response

Status: 200 OK

1. Show flow: check that *strict* is *true*.

HTTP request

Operation: GET
URI: http://192.168.11.1:8080/restconf/config/opendaylight-inventory:nodes/
node/openflow:1/table/2/flow/111
Accept: application/xml

HTTP response

Status: 200 OK

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<flow
  xmlns="urn:opendaylight:flow:inventory">
  <strict>true</strict>
  <instructions>
    <instruction>
      <order>1</order>
      <apply-actions>
        <action>
          <order>1</order>
          <flood-all-action/>
        </action>
      </apply-actions>
    </instruction>
  </instructions>
  <table_id>2</table_id>
```

```
<id>111</id>
<cookie_mask>10</cookie_mask>
<out_port>10</out_port>
<installHw>false</installHw>
<out_group>2</out_group>
<match>
  <ethernet-match>
    <ethernet-type>
      <type>2048</type>
    </ethernet-type>
  </ethernet-match>
  <ipv4-destination>10.0.0.1/24</ipv4-destination>
</match>
<hard-timeout>0</hard-timeout>
<cookie>10</cookie>
<idle-timeout>0</idle-timeout>
<flow-name>FooXf22</flow-name>
<priority>2</priority>
<barrier>false</barrier>
</flow>
```

1. Delete the flow created.

HTTP request

```
Operation: DELETE
URI: http://192.168.11.1:8080/restconf/config/.opendaylight-inventory:nodes/
node/openflow:1/table/2/flow/111
```

HTTP response

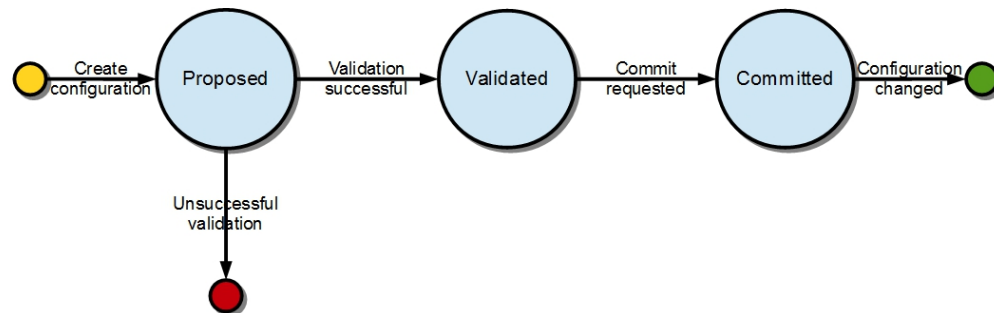
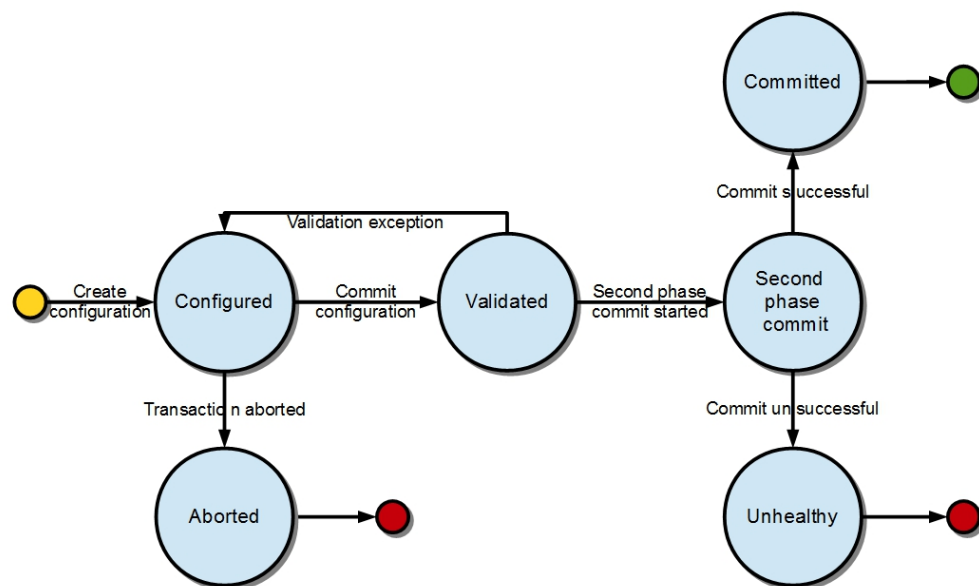
```
Status: 200 OK
```

OpenDaylight Controller: Configuration

The Controller configuration operation has three stages:

- First, a Proposed configuration is created. Its target is to replace the old configuration.
- Second, the Proposed configuration is validated, and then committed. If it passes validation successfully, the Proposed configuration state will be changed to Validated.
- Finally, a Validated configuration can be Committed, and the affected modules can be reconfigured.

In fact, each configuration operation is wrapped in a transaction. Once a transaction is created, it can be configured, that is to say, a user can abort the transaction during this stage. After the transaction configuration is done, it is committed to the validation stage. In this stage, the validation procedures are invoked. If one or more validations fail, the transaction can be reconfigured. Upon success, the second phase commit is invoked. If this commit is successful, the transaction enters the last stage, committed. After that, the desired modules are reconfigured. If the second phase commit fails, it means that the transaction is unhealthy - basically, a new configuration instance creation failed, and the application can be in an inconsistent state.

Figure 5.8. Configuration states**Figure 5.9. Transaction states**

Validation

To secure the consistency and safety of the new configuration and to avoid conflicts, the configuration validation process is necessary. Usually, validation checks the input parameters of a new configuration, and mostly verifies module-specific relationships. The validation procedure results in a decision on whether the proposed configuration is healthy.

Dependency resolver

Since there can be dependencies between modules, a change in a module configuration can affect the state of other modules. Therefore, we need to verify whether dependencies on other modules can be resolved. The Dependency Resolver acts in a manner similar to dependency injectors. Basically, a dependency tree is built.

APIs and SPIs

This section describes configuration system APIs and SPIs.

SPIs

Module `org.opendaylight.controller.config.spi`. Module is the common interface for all modules: every module must implement it. The module is designated to hold configuration attributes, validate them, and create instances of service based on the attributes. This instance must implement the `AutoCloseable` interface, owing to resources clean up. If the module was created from an already running instance, it contains an old instance of the module. A module can implement multiple services. If the module depends on other modules, setters need to be annotated with `@RequireInterface`.

Module creation

1. The module needs to be configured, set with all required attributes.
2. The module is then moved to the commit stage for validation. If the validation fails, the module attributes can be reconfigured. Otherwise, a new instance is either created, or an old instance is reconfigured. A module instance is identified by `ModuleIdentifier`, consisting of the factory name and instance name.

ModuleFactory `org.opendaylight.controller.config.spi`. The `ModuleFactory` interface must be implemented by each module factory. A module factory can create a new module instance in two ways:

- From an existing module instance
- An entirely new instance `ModuleFactory` can also return default modules, useful for populating registry with already existing configurations. A module factory implementation must have a globally unique name.

APIs

| | |
|-----------------------------|--|
| <code>ConfigRegistry</code> | Represents functionality provided by a configuration transaction (create, destroy module, validate, or abort transaction). |
|-----------------------------|--|

| | |
|--------------------------------------|---|
| ConfigTransactionController | Represents functionality for manipulating with configuration transactions (begin, commit config). |
| RuntimeBeanRegistratorAwareConfiBean | The module implementing this interface will receive RuntimeBeanRegistrator before getInstance is invoked. |

Runtime APIs

| | |
|-------------------------------------|---|
| RuntimeBean | Common interface for all runtime beans |
| RootRuntimeBeanRegistrator | Represents functionality for root runtime bean registration, which subsequently allows hierarchical registrations |
| HierarchicalRuntimeBeanRegistration | Represents functionality for runtime bean registration and unregistration from hierarchy |

JMX APIs

JMX API is purposed as a transition between the Client API and the JMX platform.

| | |
|----------------------------------|--|
| ConfigTransactionControllerMBean | Extends ConfigTransactionController, executed by Jolokia clients on configuration transaction. |
| ConfigRegistryMBean | Represents entry point of configuration management for MBeans. |
| Object names | Object Name is the pattern used in JMX to locate JMX beans. It consists of domain and key properties (at least one key-value pair). Domain is defined as "org.opendaylight.controller". The only mandatory property is "type". |

Use case scenarios

A few samples of successful and unsuccessful transaction scenarios follow:

Successful commit scenario

1. The user creates a transaction calling createTransaction() method on ConfigRegistry.
2. ConfigRegistry creates a transaction controller, and registers the transaction as a new bean.
3. Runtime configurations are copied to the transaction. The user can create modules and set their attributes.
4. The configuration transaction is to be committed.
5. The validation process is performed.
6. After successful validation, the second phase commit begins.
7. Modules proposed to be destroyed are destroyed, and their service instances are closed.
8. Runtime beans are set to registrator.
9. The transaction controller invokes the method getInstance on each module.
10. The transaction is committed, and resources are either closed or released.

Validation failure scenario The transaction is the same as the previous case until the validation process.

1. If validation fails, (that is to day, illegal input attributes values or dependency resolver failure), the `validationException` is thrown and exposed to the user.
2. The user can decide to reconfigure the transaction and commit again, or abort the current transaction.
3. On aborted transactions, `TransactionController` and `JMXRegistrator` are properly closed.
4. Unregistration event is sent to `ConfigRegistry`.

Default module instances

The configuration subsystem provides a way for modules to create default instances. A default instance is an instance of a module, that is created at the module bundle start-up (module becomes visible for configuration subsystem, for example, its bundle is activated in the OSGi environment). By default, no default instances are produced.

The default instance does not differ from instances created later in the module life-cycle. The only difference is that the configuration for the default instance cannot be provided by the configuration subsystem. The module has to acquire the configuration for these instances on its own. It can be acquired from, for example, environment variables. After the creation of a default instance, it acts as a regular instance and fully participates in the configuration subsystem (It can be reconfigured or deleted in following transactions.).

OpenDaylight Controller configuration: Initial

The Initial configuration of the controller involves two methods.

Initial configuration for controller

The two ways of configuring the controller:

- Using the [config.ini](#) property file to pass configuration properties to the OSGi bundles besides the config subsystem.
- Using the [configuration persister](#) to push the initial configuration for modules managed by the config subsystem.

Using the config.ini property file

The `config.ini` property file can be used to provide a set of properties for any OSGi bundle deployed to the controller. It is usually used to configure bundles that are not managed by the config subsystem. For details, see [the section called "OpenDaylight Controller configuration: config.ini" \[72\]](#).

Using configuration persister

Configuration persister is a default service in the controller, and is started automatically using the OSGi Activator. Its purpose is to load the initial configuration for the config subsystem and store a snapshot for every new configuration state pushed to the config-

subsystem from external clients. For details, see [the section called “OpenDaylight Controller: Configuration Persister” \[72\]](#).

OpenDaylight Controller configuration: config.ini

Various parts of the system that are not under the configuration subsystem use the file config.ini. Changes to this file apply after a server restart. The tabulation of several important configuration keys and values follows:

| Setting | Description |
|---|---|
| yangstore.blacklist=.*controller.model.* | This regular expression (can be OR-ed using pipe character) tells netconf to exclude the yang files found in the matching bundle symbolic name from the hello message. This is helpful when dealing with a netconf client that has parsing problems. |
| netconf.config.persister.* settings | See the section called “OpenDaylight Controller configuration: Initial” [71] . |
| netconf.tcp.address=0.0.0.0 netconf.tcp.port=8383 netconf.ssh.address=0.0.0.0 netconf.ssh.port=1830 netconf.ssh.pk.path = ./configuration/RSA.pk netconf.tcp.client.address=127.0.0.1 netconf.tcp.client.port=8383 | These settings specify the netconf server bindings. IP address 0.0.0.0 is used when all available network interfaces must be used by the netconf server. When starting the ssh server, the user must provide a private key. The actual authentication is done in the user admin module. By default, users admin:admin and netconf:netconf can be used to connect by means of ssh. Since the ssh bridge acts as a proxy, one needs to specify the netconf plaintext TCP address and port. These settings must normally be identical to those specified by netconf.tcp.*. |

OpenDaylight Controller: Configuration Persister

One way of configuring the controller is to use the configuration persister to push the initial configuration for modules managed by the config subsystem.

Using configuration persister

The configuration persister is a default service in the controller, and is started automatically using the OSGi Activator. Its purpose:

- Load the initial configuration for the config subsystem.
- Store a snapshot for every new configuration state pushed to the config-subsystem from external clients.

It retrieves the base configuration from the config.ini property file, and tries to load the configuration for the config subsystem. The configuration for the config subsystem is pushed as a set of edit-config netconf rpcs followed by a commit rpc since the config persister acts as a netconf client.

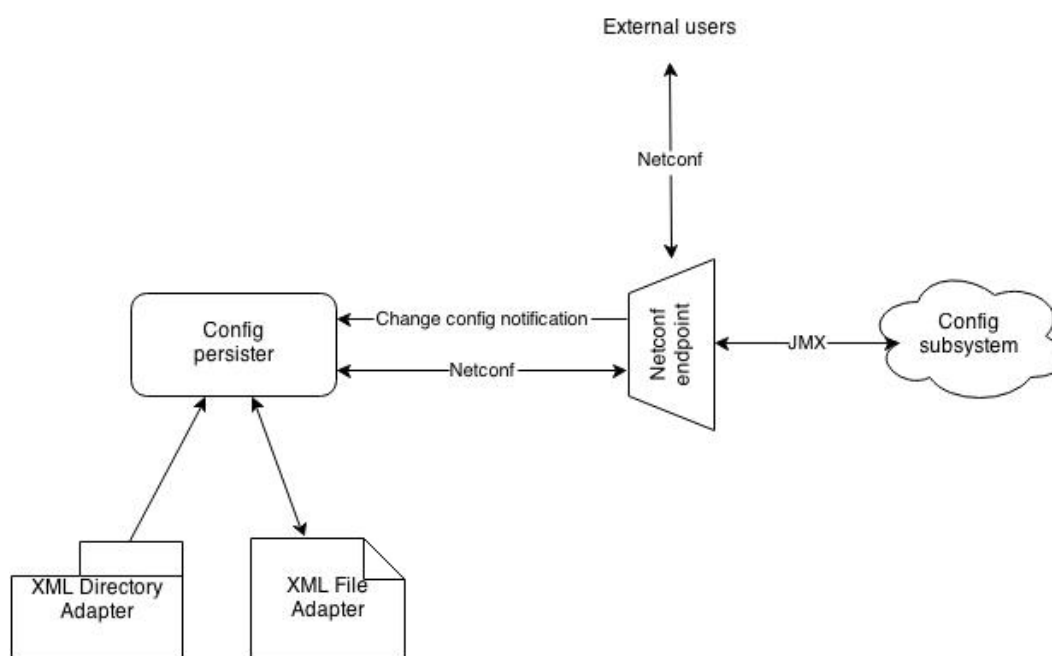
Configuration persister lifecycle:

1. Start the config persister service at *config-persister-impl* bundle startup.

2. Retrieve the base configuration of the adapters from the config.ini property file.
3. Initialize the backing storage adapters.
4. Initialize the netconf client, and connect to the netconf endpoint of the config subsystem.
5. Load the initial configuration snapshots from the latest storage adapter.
6. Send the edit-config rpc with the initial configuration snapshots.
7. Send the commit rpc.
8. Listen for any of the following changes to the configuration and persist a snapshot.

Configuration Persister interactions:

Figure 5.10. Persister



Current configuration for controller distribution

The *config.ini* property file contains the following configuration for the configuration persister:

```
netconf.config.persister.active=1,2  
  
netconf.config.persister.1.storageAdapterClass=org.opendaylight.controller.  
config.persist.storage.directory.autodetect.AutodetectDirectoryStorageAdapter  
  
netconf.config.persister.1.properties.directoryStorage=configuration/initial/  
  
netconf.config.persister.1.readonly=true
```

```
netconf.config.persist.2.storageAdapterClass=org.opendaylight.controller.
config.persist.storage.file.xml.XmlFileStorageAdapter

netconf.config.persist.2.properties.fileStorage=configuration/current/
controller.currentconfig.xml

netconf.config.persist.2.properties.numberOfBackups=1
```

With this configuration, the configuration persister initializes two adapters:

- AutodetectDirectoryStorageAdapter to load the initial configuration files from the *configuration/initial/* folder. These files will be pushed as the initial configuration for the config subsystem. Since this adapter is Read only, it will not store any configuration snapshot during the controller lifecycle.
- XmlFileStorageAdapter to store snapshots of the current configuration after any change in the file *configuration/current/controller.currentconfig.xml* (Only 1 snapshot backup is kept; every new change overwrites the previous one). The initial configuration in the controller distribution consists of 2 files in the [xml format](#).
- *configuration/initial/00-netty.xml*:

```
<snapshot>
  <required-capabilities>
    <capability>urn:opendaylight:params:xml:ns:yang:controller:netty?
module=netty&revision=2013-11-19</capability>

    <capability>urn:opendaylight:params:xml:ns:yang:controller:netty:eventexecutor?
module=netty-event-executor&revision=2013-11-12</capability>

    <capability>urn:opendaylight:params:xml:ns:yang:controller:netty:threadgroup?
module=threadgroup&revision=2013-11-07</capability>

    <capability>urn:opendaylight:params:xml:ns:yang:controller:netty:timer?
module=netty-timer&revision=2013-11-19</capability>
  </required-capabilities>
  <configuration>

    <data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <modules xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:config">
        <module>
          <type xmlns:netty=
"urn:opendaylight:params:xml:ns:yang:controller:netty:threadgroup">netty:netty-
threadgroup-fixed</type>
          <name>global-boss-group</name>
        </module>
        <module>
          <type xmlns:netty=
"urn:opendaylight:params:xml:ns:yang:controller:netty:threadgroup">netty:netty-
threadgroup-fixed</type>
          <name>global-worker-group</name>
        </module>
        <module>
          <type xmlns:netty=
"urn:opendaylight:params:xml:ns:yang:controller:netty:timer">netty:netty-
hashed-wheel-timer</type>
          <name>global-timer</name>
        </module>
      </modules>
    </data>
  </configuration>
</snapshot>
```

```

        <module>
          <type xmlns:netty=
"urn:opendaylight:params:xml:ns:yang:controller:netty:eventexecutor">netty:netty-
global-event-executor</type>
          <name>global-event-executor</name>
        </module>
      </modules>

      <services xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:config">
        <service>
          <type xmlns:netty=
"urn:opendaylight:params:xml:ns:yang:controller:netty">netty:netty-
threadgroup</type>
          <instance>
            <name>global-boss-group</name>
            <provider>/modules/module[type='netty-threadgroup-
fixed'] [name='global-boss-group']</provider>
          </instance>
          <instance>
            <name>global-worker-group</name>
            <provider>/modules/module[type='netty-threadgroup-
fixed'] [name='global-worker-group']</provider>
          </instance>
        </service>
        <service>
          <type xmlns:netty=
"urn:opendaylight:params:xml:ns:yang:controller:netty">netty:netty-event-
executor</type>
          <instance>
            <name>global-event-executor</name>
            <provider>/modules/module[type='netty-global-event-
executor'] [name='global-event-executor']</provider>
          </instance>
        </service>
        <service>
          <type xmlns:netty=
"urn:opendaylight:params:xml:ns:yang:controller:netty">netty:netty-timer</
type>
          <instance>
            <name>global-timer</name>
            <provider>/modules/module[type='netty-hashed-wheel-
timer'] [name='global-timer']</provider>
          </instance>
        </service>
      </services>
    </data>

  </configuration>
</snapshot>

```

This configuration snapshot instantiates netty utilities, which will be utilized by the controller components that use netty internally.

configuration/initial/01-md-sal.xml:

```

<snapshot>

  <configuration>

```

```

    <data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
      <modules xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:config">
        <module>
          <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom:impl">prefix:schema-
service-singleton</type>
            <name>yang-schema-service</name>
          </module>
          <module>
            <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom:impl">prefix:hash-
map-data-store</type>
              <name>hash-map-data-store</name>
            </module>
            <module>
              <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom:impl">prefix:dom-
broker-impl</type>
                <name>dom-broker</name>
                <data-store xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom:impl">
                  <type xmlns:dom=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom">dom:dom-data-
store</type>
                    <!-- to switch to the clustered data store, comment
out the hash-map-data-store <name> and uncomment the cluster-data-store one
-->
                      <name>hash-map-data-store</name>
                      <!-- <name>cluster-data-store</name> -->
                    </data-store>
                  </module>
                  <module>
                    <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl">prefix:binding-
broker-impl</type>
                      <name>binding-broker-impl</name>
                      <notification-service xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl">
                        <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-
notification-service</type>
                          <name>binding-notification-broker</name>
                        </notification-service>
                        <data-broker xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl">
                          <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-
data-broker</type>
                            <name>binding-data-broker</name>
                          </data-broker>
                        </module>
                        <module>
                          <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl">prefix:runtime-
generated-mapping</type>
                            <name>runtime-mapping-singleton</name>
                          </module>
                        </module>

```

```

        <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl">prefix:binding-
notification-broker</type>
        <name>binding-notification-broker</name>
    </module>
    <module>
        <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl">prefix:binding-
data-broker</type>
        <name>binding-data-broker</name>
        <dom-broker xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl">
            <type xmlns:dom=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom">dom:dom-broker-
osgi-registry</type>
                <name>dom-broker</name>
            </dom-broker>
            <mapping-service xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl">
                <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl">binding:binding-
dom-mapping-service</type>
                    <name>runtime-mapping-singleton</name>
                </mapping-service>
            </module>

        </modules>

        <services xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:config">
            <service>
                <type xmlns:dom=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom">dom:schema-
service</type>
                <instance>
                    <name>yang-schema-service</name>
                    <provider>/modules/module[type='schema-service-singleton'][name='yang-
schema-service']</provider>
                </instance>
            </service>
            <service>
                <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-
notification-service</type>
                <instance>
                    <name>binding-notification-broker</name>
                    <provider>/modules/module[type='binding-notification-broker'][name=
'binding-notification-broker']</provider>
                </instance>
            </service>
            <service>
                <type xmlns:dom=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom">dom:dom-data-
store</type>
                <instance>
                    <name>hash-map-data-store</name>
                    <provider>/modules/module[type='hash-map-data-store'][name='hash-map-
data-store']</provider>
                </instance>
            </service>

```

```

    <service>
      <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-
broker-osgi-registry</type>
      <instance>
        <name>binding-osgi-broker</name>
        <provider>/modules/module[type='binding-broker-impl'][name='binding-
broker-impl']</provider>
      </instance>
    </service>
    <service>
      <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-
rpc-registry</type>
      <instance>
        <name>binding-rpc-broker</name>
        <provider>/modules/module[type='binding-broker-impl'][name='binding-
broker-impl']</provider>
      </instance>
    </service>
    <service>
      <type xmlns:binding-impl=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl">binding-
impl:binding-dom-mapping-service</type>
      <instance>
        <name>runtime-mapping-singleton</name>
        <provider>/modules/module[type='runtime-generated-mapping'][name=
'runtime-mapping-singleton']</provider>
      </instance>
    </service>
    <service>
      <type xmlns:dom=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom">dom:dom-broker-
osgi-registry</type>
      <instance>
        <name>dom-broker</name>
        <provider>/modules/module[type='dom-broker-impl'][name='dom-broker']</
provider>
      </instance>
    </service>
    <service>
      <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-
data-broker</type>
      <instance>
        <name>binding-data-broker</name>
        <provider>/modules/module[type='binding-data-broker'][name='binding-data-
broker']</provider>
      </instance>
    </service>

    </services>
  </data>

</configuration>

<required-capabilities>

  <capability>urn:opendaylight:params:xml:ns:yang:controller:netty:eventexecutor?
module=netty-event-executor&amp;revision=2013-11-12</capability>

```

```
<capability>urn:opendaylight:params:xml:ns:yang:controller:threadpool?
module=threadpool&revision=2013-04-09</capability>

<capability>urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding?
module=opendaylight-md-sal-binding&revision=2013-10-28</capability>
  <capability>urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom?
module=opendaylight-md-sal-dom&revision=2013-10-28</capability>

  <capability>urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl?
module=opendaylight-sal-binding-broker-impl&revision=2013-10-28</
capability>

  <capability>urn:opendaylight:params:xml:ns:yang:controller:md:sal:dom:impl?
module=opendaylight-sal-dom-broker-impl&revision=2013-10-28</capability>

  <capability>urn:opendaylight:params:xml:ns:yang:controller:md:sal:common?
module=opendaylight-md-sal-common&revision=2013-10-28</capability>
</required-capabilities>

</snapshot>
```

This configuration snapshot instantiates md-sal modules.

After the controller is started, all these modules will be instantiated and configured. They can be further referenced from the new modules as dependencies, reconfigured, or even deleted. These modules are considered to be the base configuration for the controller and the purpose of them being configured automatically is to simplify the process of controller configuration for users, since the base modules will already be ready for use.

Adding custom initial configuration

There are multiple ways to add the custom initial configuration to the controller distribution:

1. Manually create the config file, and put it in the initial configuration folder.
2. Reconfigure the running controller using the yuma yangcli tool. The XmlFileStorageAdapter adapter will store the current snapshot, and on the next startup of the controller (assuming it was not rebuilt since), it will load the configuration containing the changes.

Custom initial configuration in bgpcep distribution

The BGPCEP project will serve as an example for adding the custom initial configuration. The bgpcep project contains the custom initial configuration that is based on the initial configuration from the controller. It adds new modules, which depend on MD-SAL and netty modules created with the initial config files of the controller. There are multiple config files in the bgpcep project. Only the 30-programming.xml file located under the programming-parent/controller-config project will be described in this section. This file contains the configuration for an instance of the instruction-scheduler module:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- vi: set et smarttab sw=4 tabstop=4: -->
<!--
    Copyright (c) 2013 Cisco Systems, Inc. and others. All rights reserved.
```



```

This program and the accompanying materials are made available under the
terms of the Eclipse Public License v1.0 which accompanies this distribution,
and is available at http://www.eclipse.org/legal/epl-v10.html.
-->
<snapshot>
  <required-capabilities>
    <capability>urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding?
module=opendaylight-md-sal-binding&revision=2013-10-28</capability>
    <capability>urn:opendaylight:params:xml:ns:yang:controller:netty?module=
netty&revision=2013-11-19</capability>
    <capability>urn:opendaylight:params:xml:ns:yang:controller:programming:impl?
module=config-programming-impl&revision=2013-11-15</capability>
    <capability>urn:opendaylight:params:xml:ns:yang:controller:programming:spi?
module=config-programming-spi&revision=2013-11-15</capability>
  </required-capabilities>
</configuration>

<data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <modules xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
    <module>
      <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:programming:impl">prefix:instruction-
scheduler-impl</type>
      <name>global-instruction-scheduler</name>
      <data-provider>
        <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-
data-broker</type>
        <name>binding-data-broker</name>
      </data-provider>
      <notification-service>
        <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-
notification-service</type>
        <name>binding-notification-broker</name>
      </notification-service>
      <rpc-registry>
        <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-
rpc-registry</type>
        <name>binding-rpc-broker</name>
      </rpc-registry>
      <timer>
        <type xmlns:netty=
"urn:opendaylight:params:xml:ns:yang:controller:netty">netty:netty-timer</
type>
        <name>global-timer</name>
      </timer>
    </module>
  </modules>

  <services xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
    <service>
      <type xmlns:pgmspi=
"urn:opendaylight:params:xml:ns:yang:controller:programming:spi">pgmspi:instruction-
scheduler</type>
      <instance>
        <name>global-instruction-scheduler</name>
        <provider>/modules/module[type='instruction-scheduler-impl'][name=
'global-instruction-scheduler']</provider>

```

```

    </instance>
  </service>
</services>
</data>

</configuration>
</snapshot>

```

Instruction-scheduler is instantiated as a module of type *instruction-scheduler-impl* with the name **global-instruction-scheduler**:

```

<module>
  <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:programming:impl">prefix:instruction-
scheduler-impl</type>
  <name>global-instruction-scheduler</name>
  ...

```

There is also an alias created for this module instance, and the alias is **global-instruction-scheduler** of type *instruction-scheduler*:

```

...
<service>
  <type xmlns:pgmspi=
"urn:opendaylight:params:xml:ns:yang:controller:programming:spi">pgmspi:instruction-
scheduler</type>
  <instance>
    <name>global-instruction-scheduler</name>
    <provider>/modules/module[type='instruction-scheduler-impl'][name='global-
instruction-scheduler']</provider>
  </instance>
</service>
...

```

The type of the alias is *instruction-scheduler*. This type refers to a certain service that is implemented by the *instruction-scheduler-impl* module. With this service type, the *global-instruction-scheduler* instance can be injected into any other module that requires *instruction-scheduler* as a dependency. One module can provide (implement) multiple services, and each of these services can be assigned an alias. This alias can be later used to reference the implementation it is pointing to. If no alias is assigned by the user, a default alias will be assigned for each provided service. The default alias is constructed from the name of the module instance with a prefix **ref_** and a possible suffix containing a number to resolve name clashes. It is recommended that users provide aliases for each service of every module instance, and use these aliases for dependency injection. References to the alias *global-instruction-scheduler* can be found in subsequent config files in the bgpcep project for example, *32-pcep.xml* located under the *pcep-parent/pcep-controller-config* project.

The configuration contains four dependencies on the MD-SAL and the netty modules:

```

...
<data-provider>
  <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-
data-broker</type>
  <name>binding-data-broker</name>
</data-provider>

```

```
<notification-service>
  <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-
notification-service</type>
  <name>binding-notification-broker</name>
</notification-service>
<rpc-registry>
  <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">binding:binding-
rpc-registry</type>
  <name>binding-rpc-broker</name>
</rpc-registry>
<timer>
  <type xmlns:netty=
"urn:opendaylight:params:xml:ns:yang:controller:netty">netty:netty-timer</
type>
  <name>global-timer</name>
</timer>
...
```

MD-SAL dependencies:

- Data-provider dependency
- Notification-service dependency
- Rpc-registry dependency

All MD-SAL dependencies can be found in the [MD-SAL initial configuration file](#). For example, rpc-registry dependency points to an alias binding-rpc-broker of the type binding-rpc-registry. This alias further points to an instance of the binding-broker-impl named binding-broker-impl. The Yang module that defines the binding-broker-impl module : [opendaylight-binding-broker-impl.yang](#).

Netty dependencies:

- Timer dependency

This configuration expects these dependencies to be already up and ready. It is the responsibility of the configuration subsystem to find the dependency and inject it. If the configuration of a module points to a non-existing dependency, the configuration subsystem will produce an exception during the validation phase. Every user-created configuration needs to point to existing dependencies. In the case of multiple initial configuration files that depend on one another, the resolution order can be ensured by the names of the files. Files are sorted by their names in ascending order. This means that every configuration file will have the visibility of modules from all previous configuration files by means of aliases.



Note

The configuration provided by initial config files can also be pushed to the controller at runtime using netconf client. The whole configuration located under the data tag needs to be inserted into the config tag in the edit-config rpc. For more information, see [Examples](#).

Configuration Persister

As a part of the configuration subsystem, the purpose of the persister is to save and load a permanent copy of a configuration. The Persister interface represents basic operations over a storage - persist configuration and load last config, configuration snapshot is represented as string and set of it's capabilities. StorageAdapter represents an adapter interface to the ConfigProvider - subset of BundleContext, allowing access to the OSGi framework system properties.

Persister implementation

Configuration persister implementation is part of the Controller Netconf. PersisterAggregator class is the implementation of the Persister interface. The functionality is delegated to the storage adapters. Storage adapters are low level persisters that do the heavy lifting for this class. Instances of storage adapters can be injected directly by means of the constructor, or instantiated from a full name of its class provided in a properties file. There can be many persisters configured, and varying numbers of them can be used.

Example of persisters configuration:

```
netconf.config.persister.active=2,3
# read startup configuration
netconf.config.persister.1.storageAdapterClass=org.opendaylight.controller.
config.persist.storage.directory.xml.XmlDirectoryStorageAdapter
netconf.config.persister.1.properties.fileStorage=configuration/initial/

netconf.config.persister.2.storageAdapterClass=org.opendaylight.controller.
config.persist.storage.file.FileStorageAdapter
netconf.config.persister.2.readonly=true
netconf.config.persister.2.properties.fileStorage=configuration/current/
controller.config.1.txt

netconf.config.persister.3.storageAdapterClass=org.opendaylight.controller.
config.persist.storage.file.FileStorageAdapter
netconf.config.persister.3.properties.fileStorage=configuration/current/
controller.config.2.txt
netconf.config.persister.3.properties.numberOfBackups=3
```

During server startup, ConfigPersisterNotificationHandler requests the last snapshot from the underlying storages. Each storage can respond by giving a snapshot or an absent response. The PersisterAggregator#loadLastConfigs() will search for the first non-absent response from storages ordered backwards as user specified (first 3, then 2). When a commit notification is received, 2 will be omitted because the read-only flag is set to true, so only 3 will have a chance to persist the new configuration. If read-only was false, or not specified, both storage adapters would be called in the order specified by *netconf.config.persister* property.

Persister Notification Handler

ConfigPersisterNotificationHandler class is responsible for listening for netconf notifications containing the latest committed configuration. The listener can handle incoming notifications, or delegates the configuration saving or loading to the persister.

Storage Adapter implementations

XML File Storage

The XmlFileStorageAdapter implementation stores configuration in an xml file.

XML Directory Storage

XmlDirectoryStorageAdapter retrieves the initial configuration from a directory. If multiple xml files are present, files are ordered based on the file names and pushed in this order (for example, 00.xml, then 01.xml..) Each file defines its required capabilities, so it will be pushed when those capabilities are advertized by ODL. Writing to this persister is not supported.

No-Operation Storage

NoOpStorageAdapter serves as a dummy implementation of the storage adapter.

Obsolete storage adapters

- File Storage
- FileStorageAdapter implements StorageAdapter, and provides file based configuration persisting. File path and name is stored as a property and a number of stored backups, expressing the count of the last configurations to be persisted too. The implementation can handle persisting input configuration, and load the last configuration.
- Directory Storage
- DirectoryStorageAdapter retrieves initial configurations from a directory. If multiple files are present, snapshot and required capabilities will be merged together. See configuration later on this page for details. Writing to this persister is not supported.
- Autodetect Directory Storage
- AutodetectDirectoryStorageAdapter retrieves initial configuration from a directory (exactly as Xml Directory Storage) but supports xml as well as plaintext format for configuration files. Xml and plaintext files can be combined in one directory. Purpose of this persister is to keep backwards compatibility for plaintext configuration files.



Important

This functionality will be removed in later releases since Plaintext File/Directory adapters are deprecated, and will be fully replaced by xml storage adapters.

Persisted snapshot format

Configuration snapshots are persisted in xml files for both file and directory adapters. They share the same format:

```
<snapshot>
```

```

    <required-capabilities>
      <capability>urn:opendaylight:params:xml:ns:yang:controller:netty?
module=netty&revision=2013-11-19</capability>
      ...
    </required-capabilities>
    <configuration>

      <data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
        <modules xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:config">
          ...
        </modules>

        <services xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:config">
          ...
        </services>

      </data>

    </configuration>
  </snapshot>

```

The whole snapshot is encapsulated in the snapshot tag that contains two children elements:

- The required-capabilities tag contains the list of yang capabilities that are required to push configurations located under the configuration tag. The config persister will not push the configuration before the netconf endpoint for the config subsystem reports all needed capabilities. Every yang model that is referenced within this xml file (as namespace for xml tag) must be referenced as a capability in this list.
- The configuration tag contains the configurations to be pushed to the config subsystem. It is wrapped in a data tag with the base netconf namespace. The whole data tag, with all its child elements, will be inserted into an edit-config rpc as config tag. For more information about the structure of configuration data, see base yang model for the config subsystem and all the configuration yang files for the controller modules as well as those provided examples. Examples contain multiple explained edit-config rpcs that change the configuration.



Note

XML File adapter adds additional tags to the xml format since it supports multiple snapshots per file.

The xml format for xml file adapter:

```

<persisted-snapshots>
  <snapshots>
    <snapshot>
      <required-capabilities>

        <capability>urn:opendaylight:params:xml:ns:yang:controller:shutdown:impl?
module=shutdown-impl&revision=2013-12-18</capability>
      </required-capabilities>
      <configuration>
        <data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">

```

```

        <modules xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:config">
            ....
        </modules>
        <services xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:config">
            ...
        </services>
    </data>
</configuration>
</snapshot>
<snapshot>
    <required-capabilities>

    <capability>urn:opendaylight:params:xml:ns:yang:controller:shutdown:impl?
module=shutdown-impl&revision=2013-12-18</capability>
    </required-capabilities>
    <configuration>
        <data xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
            <modules xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:config">
                ....
            </modules>
            <services xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:config">
                ...
            </services>
        </data>
    </configuration>
</snapshot>
</snapshots>
</persisted-snapshots>

```

MD-SAL architecture: Clustering Notifications

MD-SAL supports two kinds of messaging exchange pattern:

- Request/Reply
- Publish/Subscribe The RPC module implements the Request/Reply pattern. The notification module implements the Publish/Subscribe functionality. The implementation details are provided at: [OpenDaylight Controller:MD-SAL:Explained:Messaging Patterns](#). The focus now is on Publish/Subscribe implementation. An earlier implementation assumed a single VM deployment of the controller. The message exchange happens only within a VM in memory. The current requirement is to enable these notifications across nodes in the cluster.

Publish/Subscribe notifications are of two kinds:

- Data Change events
- Yang notifications In both cases, the notifications are broadcast to all "listeners".
Requirements Some of the requirements:
 - Ability to publish notifications to any subscriber in the cluster
 - Subscriber ability to specify delivery policy

- 1 of N: Delivery of the notification to any one of N instances of application running in the cluster
- N of N: Broadcasts
- Local only: Notifying events generated on the same node as the application instance
- Load Balancing: Round robin, least loaded etc
- Content Based or any other application specified custom logic
- Publisher capability to attach properties to the message
- Message priority
- Delivery guarantee
- Ability to plug-in external systems such as AMQP based systems

Proposed change

Based on the requirements, a change in the aPI was proposed:

```
Yang notification
publish(Notification notification, MessageProperties props);
registerNotificationListener(org.opendaylight.yangtools.yang.binding.
NotificationListener.NotificationListener listener, Selector selector);
registerNotificationListener(Class notificationType, org.opendaylight.
controller.sal.binding.api.NotificationListener listener, Selector selector);
Data change notification
registerDataChangeListener(LogicalDatastoreType store, P path, L listener,
DataChangeScope triggeringScope, "Selector selector");
public interface MessageProperties{
    public Priority priority();
    ...[add more properties]
}
public enum Priority { HIGH, NORMAL, LOW};
public interface Selector {
    public List<InstanceLocator> select(Notification event, List<InstanceLocator>
instances);
}
```

MD-SAL Architecture: DOM

There are several issues that impede the reliability and performance of mD-SAL:

- Data structures (defined in yang-data-api) are like XML structures. Therefore, it is hard to implement an optimized datastore atop them. Instead, YANG-defined data structures must be used in the data store. YANG-defined data structures are already being used in the MD-SAL: in the Java DTOs generated by YangTools, and in other components.
- The current MD-SAL data contracts do not provide enough capabilities to more accurately specify an the intent of an application and to perform optimizations to clients (for example, *do not unnecessarily deserialize data, or compute only necessary change sets*). The current datastore implementation prevents atomic updates on subtrees.

MD-SAL DOM Data Broker

The current DOM Data Broker design does not include an assumption of a intelligent in-memory cache with tree-like structures that would:

- Be able to track dependencies
- Calculate change sets
- Maintain the relationships between commit handlers, notification listeners and the actual data. This may lead to an inefficient implementation of the two-phase commit, where all state tracking during the is done by the Data Broker itself as follows:
 1. Calculate the affected subtrees.
 2. Filter the commit handlers by the affected subtrees.
 3. Filter data change listeners by the affected subtrees.
 4. Capture the initial state for data change listeners (one read per data change listener set).
 5. Start Request Commit of all the affected commit handlers.
 6. Finish Commit on all the affected commit handlers.
 7. Capture the final state for data change listeners (one read per data change listener set).
 8. Publish the Data Change events to the affected data change listeners. The states that the current DOM Data Broke keeps and maintains are mapping of subtree paths to: *
- Registered commit handlers
- Registered data change listeners
- Registered data readers DOM Data Broker has the following state keeping responsibilities: *
- Read request routing for data readers
- Two phase commit coordination
- Publish Data Change Events
- Capture Before and After state

MD-SAL: Infinispan Data Store

Components of Infinispan Data Store

Infinispan Data Store comprises the following major components:

- Encoding or Decoding a Normalized Node into and from the Infinispan TreeCache

- Managing transactions
- Managing DataChange notifications

Encoding or Decoding a Normalized Node into and from the Infinispan TreeCache

A NormalizedNode represents a tree whose structure closely models the yang model of a bunch of modules. The NormalizedNode tree typically has values either placed in:

- A LeafNode (corresponding to a leaf in yang)
- A LeafSetEntryNode (corresponding to a leaflist in yang) The encoding logic walks the NormalizedNode tree looking for LeafNodes and LeafSetEntryNodes. When the logic finds a LeafNode or a LeafSetEntryNode, it records the finding in a map with the following:
 - Instance Identifier of the parent as the key
 - The value of the leaf or leafset entry store in a map where:
 - The NodeIdentifier of the leaf/leafsetentry is the key.
 - The value of the leaf/leafsetentry is the value. The decoding process involves the following steps:
 1. Uses the interface of TreeCache to get to a certain node in the tree
 2. Walks through the tree, and reconstructs the NormalizedNode based on the key and value in the Infinispan TreeCache
 3. Validates the NormalizedNode against the schema

Managing Transactions

To ensure read-write isolation level, and for other reasons, an infinispan (JTA) transaction for each datastore transaction is created. Since a single thread may be used for multiple JTA transactions, the implementation has to ensure the suspension and resumption of the JTA transactions appropriately. However, this does not seem to have an impact on performance.

Managing DataChange notifications

The current interface for data change notifications supports the registering of listeners for the following notifications:

- Data changes at Node (consider node of a tree) level
- Events for any changes that happen at **one** level (meaning immediate children)
- Any change at the subtree level The event sent to the listener requires that the following snapshots of the tree be maintained:

- Before data change
- After data change



Note

This process is very expensive. It means maintaining a Normalized Node representing a snapshot of the tree. It involves converting the tree in Infinispan to NormalizeNode object tree required by the consumer at the start of each transaction.

To maintain the data changes:

1. At the begin of transaction, get a NormalizedNode Object tree of the current tree in ISPN TreeCache (This is mandated by the current DataChangeEvent interface.)
2. For each CUD operations that happens within the transaction, maintain a transaction log.
3. When the pre-commit of the 3PhaseCommit Transaction Interface is called, prepare data changes. This involves:
 - a. Comparing the transaction log items with the Snapshot Tree one taken at the beginning of the transactions
 - b. Preparing the DataChangeEvent lists based on what level the listeners have registered
4. Upon a commit, send the events to the listeners in a separate executor, that is asynchronously.

Suggested changes

- Remove the requirement for sending the 'before transaction tree' or the 'after transaction tree' within each event.
- Send the changed paths of tree to the consumer, and let the consumer do the reading.

Building the POC

To build or run the POC, you need the latest version of the following:

- Yangtools
- Controller
- OpenFlow plugin

To get yangtools

1. Get the latest yangtools sources, and then create a branch of it using the following command: `: git checkout 306ffd9eea5a52556b4877debd2a79ca0573ff0c -b infinispan-data-store`
2. Build using the following command: `: mvn clean install -DskipTests`

To get the Controller

1. Get the latest controller, and then create a branch using the following command: : `git checkout 259b65622b8c29c49235c2210609b9f7a68826eb -b infinispn-data-store`
2. Apply the following gerrit. : <https://git.opendaylight.org/gerrit/#/c/5900/>
3. Build using the following command: : `mvn clean install -DskipTests`
4. If the build should fails, use the following commang: : `cd.opendaylight/md-sal/sal-ispn-datastore`
5. Build using the following command: `mvn clean install`
6. Return to the controller directory, and build using: : `mvn clean install -DskipTests` or resume build

To get the OpenFlowplugin

1. Get the latest openflowplugin code and then create a branch using the following command: : `git checkout 6affeeef4de51ce4b7de86fd9ccf51add3922f7 -b infinispn-data-store`
2. Build using the following command: : `mvn clean install -DskipTests`
3. Copy the sal-ispn-datastore jar to the plugins folder.

Running the POC

Prerequisite Ensure that the 01-md-sal.xml file has been changed to use the new MD-SAL datastore.

- Run the controller with the infinispn datastore. The section, [the section called "Comparison of In-Memory and Infinispn Datastore" \[93\]](#) provides information about cbench testing.



Note

If you want to see performance numbers similar to those documented, disable datachange notifications. The only way to do that in the POC is to change the code in `ReadWriteTransactionImpl`. Look for the `FIXME` comments.

State of the POC

- Encoding and Decoding a Normalized Node into an Infinispn TreeCache works
- Integrated with the controller
- Eventing works
- With Data Change events disabled, the Infinispn based datastore performs the same, or better than, the custom In-Memory Datastore. Although initially slow, with time it seems to perform more consistently than the In-Memory Datastore.,

- Not fully tested

Infinispan-related learnings

Below par functioning of TreeCache#removeNode API The Infinispan removeNode API failed to remove nodes in the tree, as was promised, correctly. This means, for example, that when a mininet topology changes, some nodes may not be removed from inventory and topology. This behaviour has not been properly evaluated, and no remedy is currently available.

Datastore-related learnings

Multiple transactions can be created per thread This is a problem because if the backing datastore (infinispan) uses JTA transactions, only one transaction can be active per thread. Although this does not necessarily mean the usage of one thread per transaction, it calls for the suspension of one transaction and the resumption of another. TIP:: * Allow only one active transaction per thread. * Add an explicit suspend or resume method to a transaction.

No clarity on the closing of Read-Only transactions

For every DataStore transaction, a JTA transaction needs to be created. This is to ensure isolation (repeatable reads). When the transaction is done, it must be committed, rolled back, or closed in some fashion. Read-only transactions may not close. This leads to JTA transactions being open until they are timed out.

- TIP
- A DataStore may need to do time-outs as well.
 - Call `close` explicitly for read-only transactions.

Write and Delete methods in a read-write transaction do not return a Future

The Write and Delete methods on the DOMWriteTransaction return a void instead of a Future, creating the impression that these methods are synchronous. This is not necessarily true in all cases: for example, in the infinispan datastore, the write was actually done in a separate thread to support multiple transactions on a single thread. TIP: Return a ListenableFuture for both Write and Delete methods.

Expense of creating a DataChange event

Creating a DataChange event is very expensive because it needs to pass the Original Sub tree and the Modified Sub tree. A NormalizedNode object needs to be created to create a DataChange event. The NormalizedNode object may be a snapshot of the complete modules data to facilitate the sending of the original subtree to DataChange listeners. The prohibitive expense prevents this implementation in every transaction. This is a problem not only in the infinispan datastore but also in a distributed system. A distributed system shards data to collocate it on a different node on the cluster with applications and datachange

listeners. For example, while a system may have shards collocated with the inventory application; the topology application may be a datachange listener for datachange events. In this case, the original subtree and the modified sub tree would need to be serialized in some form, and sent to the topology listener. TIP: Remove the `getOriginalSubtree` and `getModifiedSubtree` methods from the datachange listener; understand the use case for providing them; and find a cheaper alternative.

Complications of reconstructing a Normalized Node from different data-structures

The reconstruction of a Normalized Node from a different data-structure, like a map or a key-value store, is complicated or may appear complicated. A `NormalizedNode` is the binding-independent equivalent of data that gets stored in the datastore. For the in-memory datastore, it is the native storage format. It is a complicated structure that basically mirrors the model as defined in yang. Understanding it and properly decoding it could be a challenge for the implementation of an alternate datastore. TIP: Create utility classes to construct a normalized node from a simple tree structure. The `Old CompositeNode` or the `Infinispan Node` for example is a much simpler structure to follow.

Comparison of In-Memory and Infinispan Datastore

Cbench was used to compare the performance of the two datastores. To prepare the controller for testing:



Important

Use the openflow plugin distribution.

1. Remove the simple forwarding, arp handler, and md-sal statistics manager bundles.
2. Set the log level to ERROR.
3. Run the controller with the following command: `./run.sh -Xmx4G -Xms2G -XX:NewRatio=5 -XX:+UseG1GC -XX:MaxPermSize=256m`
4. From the osgi command prompt, use **dropAllPackets on**.

Running cbench

For both the in-memory and infinispan datastore versions, cbench was run 11 times. The first run is ignored in both cases.

- Use the cbench command: `cbench -c <controller ip> -p 6633 -m 1000 -l 10 -s 16 -M 1000`
This was a latency test and the arguments roughly translate to this: `-m 1000` : use 1000 milliseconds per test `-l 10` : use 10 loops per test `-s 16` : fake 16 switches `-M 1000` : use 1000 hosts per switch

The results for In-Memory Datastore

To test the in-memory datastore, a pre-built openflow plugin distribution from Jenkins was downloaded on and on which was enabled the new in-memory datastore. **In-Memory Datastore Results**

| Run | Min | Max | Avg | StdDev |
|-----|-----|------|-----|--------|
| 1 | 365 | 1049 | 715 | 04 |
| 2 | 799 | 1044 | 953 | 71 |
| 3 | 762 | 949 | 855 | 59 |
| 4 | 616 | 707 | 666 | 27 |
| 5 | 557 | 639 | 595 | 24 |
| 6 | 510 | 583 | 537 | 25 |
| 7 | 455 | 535 | 489 | 22 |
| 8 | 351 | 458 | 420 | 38 |
| 9 | 396 | 440 | 417 | 14 |
| 10 | 376 | 413 | 392 | 13 |

Infinispan Datastore

The Infinispan Datastore was built of a master a month old. Since the In-Memory datastore was hardcoded at that time the in-memory datastore was swapped for the the infinispan datastore by modifying the sal-broker-impl sources.

Listed are some steps that were either completed to isolate the changes that were being made, or to tweak performance:

- Infinispan 5.3 was used because to isolate changes to utilize tree cache to the infinispan datastore bundles. Attempting to use version 6.0 caused a problem in loading some classes from infinispan. Ideally, to use infinispan as a backing store, tweak clustering services to obtain a treecache.
- Added an exists method onto the In-Memory ReadTransaction API. This was because it was found that in one place in the BA Broker was code which checked for the existence of nodes in the tree by doing a read. Reads are a little expensive on the Infinispan datastore because of the need to convert to a NormalizedNode. An exists method was added to the interface to just check for node-existence.
- When a transaction was used to read data it was not being closed causing the Infinispan JTA transactions to persist. Again, a change in the broker was made to close a transaction after it was concluded so that it dis not persist and trigger a clean by the reaper.

Infinispan Datastore Results

| Run | Min | Max | Avg | StdDev |
|-----|-----|-----|-----|--------|
| 1 | 43 | 250 | 186 | 61 |
| 2 | 266 | 308 | 285 | 13 |
| 3 | 300 | 350 | 325 | 12 |
| 4 | 378 | 446 | 412 | 24 |
| 5 | 609 | 683 | 644 | 26 |
| 6 | 492 | 757 | 663 | 76 |
| 7 | 794 | 838 | 816 | 11 |
| 8 | 645 | 845 | 750 | 60 |
| 9 | 553 | 829 | 708 | 100 |
| 10 | 615 | 910 | 710 | 86 |

OpenDaylight Controller configuration: FAQs

Generic questions about the configuration subsystem

There is already JMX. Why do we need another system?

Java Management Extensions (JMX) provides programmatic access to management data, defining a clear structure on the level of a single object (MBeans). It provides the mechanism to query and set information exposed from these MBeans, too. It is adequate for replacing properties, but it does not treat the JVM container for what it is: a collection of applications working in concert. When the configuration problem is taken to the level of an entire system, there are multiple issues which JMX does not solve:

- The need to validate that a proposed system is semantically valid before an attempt to change is made
- The ability to synchronize modification multiple properties at the same time, such that both occur at the same time
- The ability to express dependencies between applications
- Machine-readable descriptions of layouts of configuration data

Why use YANG?

The problem of configuring a device has been tackled in [IETF](#) for many years now, initially using [SNMP](#) (with [MIB](#) as the data definition language). While the protocol has been successful for monitoring devices, it has never gained traction as the unified way of configuring devices. The reasons for this have been [analyzed](#) and [NETCONF](#) was standardized as the successor protocol. NETCONF provides the abstractions to deal with configuration validation, and relies on [YANG](#) as its data modeling language. The configuration subsystem is designed to completely align with NETCONF such that it can be used as the native transport with minimal translation.

OpenDaylight Controller configuration: Component map

| Component | Description |
|-----------------------|--|
| config-subsystem-core | Config subsystem core. Manages the configuration of the controller. Responsibilities: <ul style="list-style-type: none">• Scanning of bundles for ModuleFactories.• Transactional management of lifecycle and dependency injection for config modules.• Exposure of modules and their configuration into JMX. |
| netty-config | Config modules for netty related resources, for example, netty-threadgroup, netty-timer and others. |

| Component | Description |
|---|--|
| | Contains config module definition in the form of yang schemas + generated code binding for the config subsystem. |
| controller-shutdown | <p>Controller shutdown mechanism.</p> <p>Brings down the whole OSGi container of the controller.</p> <p>Authorization required in the form of a "secret string".</p> <p>Also contains config module definition in the form of yang schemas + generated code binding for the config subsystem. This makes it possible to invoke shutdown by means of the config-subsystem.</p> |
| threadpool-config | <p>Config modules for threading related resources, for example, threadfactories, fixed-threadpool, and others.</p> <p>Contains config module definition in the form of yang schemas + generated code binding for the config subsystem.</p> |
| logback-config | <p>Config modules for logging (logback) related resources, for example, loggers, appenders, and others.</p> <p>Contains config module definition in the form of yang schemas + generated code binding for the config subsystem.</p> |
| netconf-config-dispatcher-config | <p>Config modules for netconf-dispatcher(from netconf subsystem).</p> <p>Contains config module definition in the form of yang schemas + generated code binding for the config subsystem.</p> |
| yang-jmx-config-generator | Maven plugin that generates the config subsystem code binding from provided yang schemas. This binding is required when the bundles want to participate in the config subsystem. |
| yang-jmx-config-generator-testing-modules | Testing resources for the maven plugin. |
| config-persister | <p>Contains the api definition for an extensible configuration persister(database for controller configuration).</p> <p>The persister (re)stores the configuration for the controller. Persister implementation can be found in the netconf subsystem.</p> <p>The adapter bundles contain concrete implementations of storage extension. They store the config as xml files on the filesystem.</p> |
| config-module-archetype | <p>Maven archetype for "config subsystem aware" bundles.</p> <p>This archetype contains blueprints for yang-schemas, java classes, and other files(for example, pom.xml) required for a bundle to participate in the config subsystem.</p> |

| Component | Description |
|-----------|---|
| | This archetype generates a bundle skeleton that can be developed into a full blown "config subsystem aware" bundle. |

OpenDaylight Controller: Netconf component map

| Component | Description |
|---------------------------|---|
| netconf-server | Implementation of the generic (extensible) netconf server over tcp/ssh. Handles the general communication over the network, and forwards the rpcs to its extensions that implement the specific netconf rpc handles (For example: get-config). |
| netconf-to-config-mapping | API definition for the netconf server extension with the base implementation that transforms the netconf rpcs to java calls for the config-subsystem (config-subsystem netconf extension). |
| netconf-client | Netconf client basic implementation. Simple netconf client that supports netconf communication with remote netconf devices using xml format. |
| netconf-monitoring | Netconf-monitoring yang schemas with the implementation of a netconf server extension that handles the netconf-monitoring related handlers (For example: adding netconf-state to get rpc) |
| config-persister-impl | Extensible implementation of the config persister that persists the configuration in the form of xml,(easy to inject to edit-config rpc) and loads the initial configuration from the persisted files. The configuration is stored after every successful commit rpc. |
| netconf-cli | Prototype of a netconf cli. |

OpenDaylight Controller Configuration: Examples sample project

Sample maven project

In this example, we will create a maven project that provides two modules, each implementing one service. We will design a simple configuration, as well as runtime data for each module using yang. A sample maven project called config-demo was created. This project contains two Java interfaces: Foo and Bar. Each interface has one default implementation per interface, FooImpl and BarImpl. Bar is the producer in our example and produces integers when the method getNextEvent() is called. Foo is the consumer, and its implementation depends on a Bar instance. Both implementations require some configuration that is injected by means of constructors.

- Bar.java:

```
package org.opendaylight.controller.config.demo;
```

```
public interface Bar {  
    int getNextEvent();  
}
```

- **BarImpl.java:**

```
package org.opendaylight.controller.config.demo;  
  
public class BarImpl implements Bar {  
    private final int l1, l2;  
    private final boolean b;  
  
    public BarImpl(int l1, int l2, boolean b) {  
        this.l1 = l1;  
        this.currentL = l1;  
        this.l2 = l2;  
        this.b = b;  
    }  
  
    private int currentL;  
  
    @Override  
    public int getNextEvent() {  
        if(currentL==l2)  
            return -1;  
        return currentL++;  
    }  
}
```

- **Foo.java:**

```
package org.opendaylight.controller.config.demo;  
  
public interface Foo {  
    int getEventCount();  
}
```

- **FooImpl.java:**

```
package org.opendaylight.controller.config.demo;  
  
public class FooImpl implements Foo {  
    private final String strAttribute;  
    private final Bar barDependency;  
    private final int intAttribute;  
  
    public FooImpl(String strAttribute, int intAttribute, Bar barDependency) {  
        this.strAttribute = strAttribute;  
        this.barDependency = barDependency;  
        this.intAttribute = intAttribute;  
    }  
  
    @Override  
    public int getEventCount() {  
        int count = 0;  
    }  
}
```

```

        while(barDependency.getNextEvent() != intAttribute) {
            count++;
        }
        return count;
    }
}

```

- pom.xml (config-demo project is defined as a sub-module of the controller project, and at this point contains only the configuration for maven-bundle-plugin):

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <artifactId>commons.opendaylight</artifactId>
    <groupId>org.opendaylight.controller</groupId>
    <version>1.4.1-SNAPSHOT</version>
    <relativePath>../commons/opendaylight/pom.xml</relativePath>
  </parent>
  <groupId>org.opendaylight.controller</groupId>
  <version>0.1.1-SNAPSHOT</version>
  <artifactId>config-demo</artifactId>
  <packaging>bundle</packaging>
  <name>${project.artifactId}</name>
  <prerequisites>
    <maven>3.0.4</maven>
  </prerequisites>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>2.4.0</version>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-Name>${project.groupId}.${project.
artifactId}</Bundle-Name>
            <Export-Package>
              org.opendaylight.controller.config.demo,
            </Export-Package>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>

```

Describing the module configuration using yang

In order to fully leverage the utilities of the configuration subsystem, we need to describe the services, modules, their configurations, and the runtime state using the yang modeling language. We will define two services and two modules, which will be used to configure

the instances of FooImpl and BarImpl. This definition will be split into two yang files: config-demo.yang (service definition) and config-demo-impl.yang (module definition).

- config-demo.yang

```
module config-demo {
  yang-version 1;
  namespace "urn:opendaylight:params:xml:ns:yang:controller:config:demo";
  prefix "demo";

  import config { prefix config; revision-date 2013-04-05; }

  description
    "Service definition for config-demo";

  revision "2013-10-14" {
    description
      "Initial revision";
  }

  // Service definition for service foo that encapsulates instances of org.
  // opendaylight.controller.config.demo.Foo
  identity foo {
    description
      "Foo service definition";

    base "config:service-type";
    config:java-class "org.opendaylight.controller.config.demo.Foo";
  }

  identity bar {
    description
      "Bar service definition";

    base "config:service-type";
    config:java-class "org.opendaylight.controller.config.demo.Bar";
  }
}
```

The config yang module needs to be imported in order to define the services. There are two services defined, and these services correspond to the Java interfaces Foo and Bar (specified by the config:java-class extension).

- config-demo-impl.yang

```
module config-demo-impl {

  yang-version 1;
  namespace
    "urn:opendaylight:params:xml:ns:yang:controller:config:demo:java";
  prefix "demo-java";

  // Dependency on service definition for config-demo
  /* Service definitions could be also located in this yang file or even
   * in a separate maven project that is marked as maven dependency
   */
  import config-demo { prefix demo; revision-date 2013-10-14; }

  // Dependencies on config subsystem definition
  import config { prefix config; revision-date 2013-04-05; }
```

```
import rpc-context { prefix rpcx; revision-date 2013-06-17; }

description
  "Service implementation for config-demo";

revision "2013-10-14" {
  description
    "Initial revision";
}
                                                                    //-----
module foo-impl ----- //
  // Module implementing foo service
  //
  identity foo-impl {
    //
    base config:module-type;
    //
    config:provided-service demo:foo;
    //
    config:java-name-prefix FooImpl;
    //
  }
  //
  //
  // Configuration for foo-impl module
  //
  augment "/config:modules/config:module/config:configuration" {
    //
    case foo-impl {
      //
      when "/config:modules/config:module/config:type = 'foo-impl'";
      //
      //
      leaf str-attribute {
        //
        type string;
        //
      }
      //
      //
      leaf int-attribute {
        //
        type int32;
        //
      }
      //
      //
      //
      // Dependency on bar service instance
      //
      container bar-dependency {
        //
        uses config:service-ref {
          //

```

```
        refine type {
            //
            mandatory true;
            //
            config:required-identity demo:bar;
            //
        }
        //
    }
    //
}
//

//
// Runtime state definition for foo-impl module
//
augment "/config:modules/config:module/config:state" {
    //
    case foo-impl {
        //
        when "/config:modules/config:module/config:type = 'foo-impl'";
        //

        //
    }
    //
}
//

// ----- //
// Module implementing bar service
identity bar-impl {
    base config:module-type;
    config:provided-service demo:bar;
    config:java-name-prefix BarImpl;
}

augment "/config:modules/config:module/config:configuration" {
    case bar-impl {
        when "/config:modules/config:module/config:type = 'bar-impl'";

        container dto-attribute {
            leaf int-attribute {
                type int32;
            }

            leaf int-attribute2 {
                type int32;
            }

            leaf bool-attribute {
                type boolean;
            }
        }
    }
}
```

```

    }
  }
}

augment "/config:modules/config:module/config:state" {
  case bar-impl {
    when "/config:modules/config:module/config:type = 'bar-impl'";

  }
}
}

```

The config yang module as well as the config-demo yang module need to be imported. There are two modules defined: foo-impl and bar-impl. Their configuration (defined in the augment "/config:modules/config:module/config:configuration" block) corresponds to the configuration of the FooImpl and BarImpl Java classes. In the constructor of FooImpl.java, we see that the configuration of foo-impl module defines three similar attributes. These arguments are used to instantiate the FooImpl class. These yang files are placed under the src/main/yang folder.

Updating the maven configuration in pom.xml

The yang-maven-plugin must be added to the pom.xml. This plugin will process the yang files, and generate the configuration code for the defined modules. Plugin configuration:

```

<plugin>
  <groupId>org.opendaylight.yangtools</groupId>
  <artifactId>yang-maven-plugin</artifactId>
  <version>${yangtools.version}</version>
  <executions>
    <execution>
      <goals>
        <goal>generate-sources</goal>
      </goals>
      <configuration>
        <codeGenerators>
          <generator>
            <codeGeneratorClass>
              org.opendaylight.controller.config.
yangjmxgenerator.plugin.JMXGenerator
            </codeGeneratorClass>
            <outputBaseDir>${project.build.directory}/generated-
sources/config</outputBaseDir>
            <additionalConfiguration>
              <namespaceToPackage1>

urn:opendaylight:params:xml:ns:yang:controller==org.opendaylight.controller.
config.yang

              </namespaceToPackage1>
            </additionalConfiguration>
          </generator>
        </codeGenerators>
        <inspectDependencies>true</inspectDependencies>
      </configuration>
    </execution>
  </executions>
</plugin>

```



```
</executions>
<dependencies>
  <dependency>
    <groupId>org.opendaylight.controller</groupId>
    <artifactId>yang-jmx-generator-plugin</artifactId>
    <version>${config.version}</version>
  </dependency>
</dependencies>
</plugin>
```

The configuration important for the plugin: the output folder for the generated files, and the mapping between the yang namespaces and the java packages (Inspect dependencies must be set to true.). The default location for the yang files is under the src/main/yang folder. This plugin is backed by the artifact yang-jmx-generator-plugin and its class org.opendaylight.controller.config.yangjmxgenerator.plugin.JMXGenerator is responsible for code generation. This artifact is part of the configuration subsystem.

In addition to the yang-maven-plugin, it is necessary to add the build-helper-maven-plugin in order to add the generated sources to the build process:

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <version>1.8</version>
  <executions>
    <execution>
      <id>add-source</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>add-source</goal>
      </goals>
      <configuration>
        <sources>
          <source>${project.build.directory}/generated-sources/
config</source>;
        </sources>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Earlier, the configuration yang module in the yang files was imported. In order to acquire this yang module, we need to add a dependency to the pom file:

```
<dependency>
  <groupId>org.opendaylight.controller</groupId>
  <artifactId>config-api</artifactId>
  <version>${config.version}</version>
</dependency>
```

In addition, a couple of utility dependencies must be added:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
</dependency>
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
```

```
</dependency>
```

Run **mvn clean install**.

Generated java files

A set of new source files divided into two groups is seen. The first group is located under the `${project.build.directory}/generated-sources/config` directory, which was specified in the `yang-maven-plugin` configuration. The second group is located under the `src/main/java` directory. Both groups then define the package `org.opendaylight.controller.config.yang.config.demo.impl`. The first group contains code that must not be edited in any way, since this code can be regenerated by the plugin if necessary. The code that needs to be edited belongs to the second group and is located under `src/main/java`.

Generated config source files examples

- `BarImplModuleMXBean.java`

```
public interface BarImplModuleMXBean
{
    public org.opendaylight.controller.config.yang.config.demo.java.
    DtoAttribute getDtoAttribute();

    public void setDtoAttribute(org.opendaylight.controller.config.yang.
    config.demo.java.DtoAttribute dtoAttribute);
}
```

The `BarImplModuleMXBean` interface represents the getter and the setter for `dtoAttribute` that will be exported to the configuration registry by means of JMX. The attribute was defined in the yang model: in this case, it is the composite type which was converted to `OpenType`.

- Attribute definition from `config-demo-impl.yang`

```
// Module implementing bar service
identity bar-impl {
    base config:module-type;
    config:provided-service demo:foo;
    config:java-name-prefix BarImpl;
}

augment "/config:modules/config:module/config:configuration" {
    case bar-impl {
        when "/config:modules/config:module/config:type = 'bar-impl'";

        container dto-attribute {
            leaf int-attribute {
                type int32;
            }

            leaf int-attribute2 {
                type int32;
            }

            leaf bool-attribute {
```

```

        type boolean;
    }
}
}

```

From the container `dto-attribute`, the `DtoAttribute` java file was generated. The Class contains the plain constructor, and the getters and setters for the attributes defined as container leaves. Not only is `ModuleMXBean` generated from this module definition, but also `BarImplModuleFactory` and `BarImplModule` stubs (in fact `AbstractBarImplModuleFactory` and `AbstractBarImplModule` are generated too.).

- **AbstractBarImplModule.java** This abstract class is almost fully generated: only the method `validate()` has an empty body and the method `createInstance()` is abstract. The user must implement both methods by user. `AbstractBarImplModule` implements its `ModuleMXBean`, `Module`, `RuntimeBeanRegistratorAwareModule`, and the dependent service interface as defined in yang. Moreover, the class contains two types of constructors: one for the module created from the old module instance, and the second for module creation from scratch.
- **AbstractBarImplModuleFactory.java** Unlike `AbstractModule`, `AbstractFactory` is fully generated, but it is still an abstract class. The factory is responsible for module instances creation, and provides two type of `instantiateModule` methods for both module constructor types. It implements the `ModuleFactory` interface.

Next, create the runtime bean for `FoolImplModule`. Runtime beans are designated to capture data about the running module.

- Add runtime bean definition to `config-demo-impl.yang`

Modifying generated sources

Generated source files:

- `src/main/java/**/BarImplModule`
- `src/main/java/**/BarImplModuleFactory`
- `src/main/java/**/FoolImplModule`
- `src/main/java/**/FoolImplModuleFactory`

BarImplModule We will start by modifying `BarImplModule`. Two constructors and two generated methods are seen:

```

@Override
public void validate(){
    super.validate();
    // Add custom validation for module attributes here.
}

@Override
public java.lang.AutoCloseable createInstance() {
    //TODO:implement
    throw new java.lang.UnsupportedOperationException("Unimplemented stub
method");
}

```

```
}
```

In **validate**, specify the validation for configuration attributes, for example:

```
@Override
public void validate(){
    super.validate();
    Preconditions.checkNotNull(getDtoAttribute());
    Preconditions.checkNotNull(getDtoAttribute().getBoolAttribute());
    Preconditions.checkNotNull(getDtoAttribute().getIntAttribute());
    Preconditions.checkNotNull(getDtoAttribute().getIntAttribute2());
    Preconditions.checkState(getDtoAttribute().getIntAttribute() >
getDtoAttribute().getIntAttribute2());
}
```

In **createInstance** you need to create a new instance of the bar service # Bar interface, for example:

```
@Override
public java.lang.AutoCloseable createInstance() {
    return new BarImpl(getDtoAttribute().getIntAttribute(),
getDtoAttribute().getIntAttribute2(), getDtoAttribute()
        .getBoolAttribute());
}
```

Notes:

- **createInstance** returns **AutoCloseable** so the returned type needs to implement it. (You can make **BarImpl** implement **AutoCloseable**, or create a Wrapper class around the **BarImpl** instance that implements **AutoCloseable**, or even extend the **BarImpl** class and make it implement it.)
- You can access all the configuration attributes by means of the getter methods.
- In **config-demo-impl.yang**, we defined the **bar-impl** configuration as a container **dto-attribute**. The code generator creates a transfer object **DtoAttribute** that you can access by means of the **getDtoAttribute()** method, and retrieve configuration data from it. You can even add a new constructor to **BarImpl** that takes this transfer object, and reduces the number of arguments.

FooImplModule We will not add any custom validation in this module. The **createInstance** method will look as follows:

```
@Override
public java.lang.AutoCloseable createInstance() {
    return new FooImpl(getStrAttribute(), getIntAttribute(),
getBarDependencyDependency());
}
```

Adding support for default instances

In order to provide a default instance of module **bar-impl**, we need to further modify the generated code by the overriding method **getDefaultModules** in **src/main/java/**/BarImplModuleFactory** class. The body of this class is empty thus far, and it inherits the default behaviour from its parent abstract factory. Use the following code to replace the empty body:

```
public static final ModuleIdentifier defaultInstanceId = new
    ModuleIdentifier(NAME, "defaultInstance1");

    @Override
    public Set<BarImplModule> getDefaultModules(DependencyResolverFactory
dependencyResolverFactory, BundleContext bundleContext) {
        DependencyResolver depResolver1 = dependencyResolverFactory.
createDependencyResolver(defaultInstanceId);
        BarImplModule defaultModule1 = new BarImplModule(defaultInstanceId,
depResolver1);
        defaultModule1.
setDtoAttribute(getDefaultConfiguration(bundleContext));

        return Sets.newHashSet(defaultModule1);
    }

    private DtoAttribute getDefaultConfiguration(BundleContext bundleContext)
    {
        DtoAttribute defaultConfiguration = new DtoAttribute();

        String property = bundleContext.getProperty("default.bool");
        defaultConfiguration.setBoolAttribute(property == null ? false :
Boolean.parseBoolean(property));

        property = bundleContext.getProperty("default.int1");
        defaultConfiguration.setIntAttribute(property == null ? 55 : Integer.
parseInt(property));

        property = bundleContext.getProperty("default.int2");
        defaultConfiguration.setIntAttribute2(property == null ? 0 : Integer.
parseInt(property));

        return defaultConfiguration;
    }
```

The `getDefaultModules` method now produces an instance of the `bar-impl` module with the name `defaultInstance1`. (It is possible to produce multiple default instances since the return type is a Set of module instances.) Note the `getDefaultConfiguration` method. It provides the default configuration for default instances by trying to retrieve system properties from `bundleContext` (or provides hardcoded values in case the system property is not present).

For the controller distribution, system properties can be fed by means of `config.ini` file.

The method `getDefaultModules` is called automatically after a bundle containing this factory is started in the OSGi environment. Its default implementation returns an empty Set.

The default instances approach is similar to the Activator class approach in OSGi with the advantage of default instances being managed by the configuration subsystem. This approach can either replace the Activator class approach, or be used along with it.

Verifying the default instances in distribution

If we add the `config-demo` bundle to the `opendaylight` distribution, we can verify the presence of the default instance. The file `pom.xml` under the `opendaylight/distribution/opendaylight` folder needs to be modified by adding the `config-demo` dependency:

```
<dependency>
```

```
<groupId>${project.groupId}</groupId>
<artifactId>config-demo</artifactId>
<version>0.1.1-SNAPSHOT</version>
</dependency>
```

Now we need to rebuild the conf-demo module using mvn clean install. Then, we can build the distribution using the same mvn command under the *.opendaylight/distribution/opendaylight* folder. If we go to the *.opendaylight/distribution/opendaylight/target/distribution.opendaylight-osgipackage/opendaylight* folder, and execute run.sh, the.opendaylight distribution should start.

We can check the presence of the default instances by means of JMX using a tool such as *jvisualvm*.

OpenDaylight Controller: Configuration examples user guide

Configuring thread pools with yangcli-pro

Requirements: yangcli-pro version 13.04-9.2 or later

Connecting to plaintext TCP socket and ssh

Currently SSH is exposed by the controller. The network interface and port are configured in configuration/config.ini . The current configuration of netconf is as follows:

```
# Netconf startup configuration
#netconf.tcp.address=127.0.0.1
#netconf.tcp.port=8383

netconf.ssh.address=0.0.0.0
netconf.ssh.port=1830
```

To connect the yangcli-pro client, use the following syntax:

```
yangcli-pro --user=admin --password=admin --transport=ssh --ncport=1830 --
server=localhost
```

If the plaintext TCP port is not commented out, one can use the following:

```
yangcli-pro --user=a --password=a --transport=tcp --ncport=8383 --server=
localhost
```

Authentication in this case is ignored.

For better debugging, include following arguments:

```
--log=/tmp/yuma.log --log-level=debug4 --log-console
```



Note

When the log file is set, the output will not appear on stdout.

Configuring threadfactory

The threadfactory is a service interface that can be plugged into threadpools, defined in config-threadpool-api (see the [yang file](#). The implementation to be used is called threadfactory-naming. This implementation will set a name for each thread created using a configurable prefix and auto incremented index. See the [Yang file](#).

1. Launch yangcli-pro and connect to the server.
2. Enter **get-config source=running** to see the current configuration. Example output:

```
rpc-reply {
  data {
    modules {
      module binding-broker-singleton {
        type binding-impl:binding-broker-impl-singleton
        name binding-broker-singleton
      }
    }
    services {
      service md-sal-binding:binding-broker-osgi-registry {
        type md-sal-binding:binding-broker-osgi-registry
        instance ref_binding-broker-singleton {
          name ref_binding-broker-singleton
          provider /modules/module[type='binding-broker-impl-singleton'][name=
'binding-broker-singleton']
        }
      }
    }
  }
}
```

1. Enter the merge `/modules/module`.
2. At the prompt, enter the string value for the leaf `<name>`. This is the name of the config module. Enter `threadfactory-bgp`.
3. Set the identityref for the leaf `<type>`. Press Tab to see a list of available module names. Enter `threadfactory-naming`.
4. At the prompt, choose the case statement. Example output:

```
1: case netty-threadgroup-fixed:
  leaf thread-count
2: case netty-hashed-wheel-timer:
  leaf tick-duration
  leaf ticks-per-wheel
  container thread-factory
3: case async-eventbus:
  container threadpool
4: case threadfactory-naming:
  leaf name-prefix
5: case threadpool-fixed:
  leaf max-thread-count
  container threadFactory
6: case threadpool-flexible:
  leaf max-thread-count
  leaf minThreadCount
  leaf keepAliveMillis
```

```
        container threadFactory
7: case threadpool-scheduled:
    leaf max-thread-count
    container threadFactory
8: case logback:
    list file-appenders
    list rolling-appenders
    list console-appenders
    list loggers
```

In this case, we chose 4.

1. Next fill in the string value for the leaf <name-prefix>. Enter `bgp`. : (You should get an OK response from the server.)
2. Optionally issue `get-config source=candidate` to verify the change.
3. Issue `commit`.
4. Issue `get-config source=running`. Example output:

```
rpc-reply {
  data {
    modules {
      module binding-broker-singleton {
        type binding-impl:binding-broker-impl-singleton
        name binding-broker-singleton
      }
      module threadfactory-bgp {
        type th-java:threadfactory-naming
        name threadfactory-bgp
        name-prefix bgp
      }
    }
    services {
      service th:threadfactory {
        type th:threadfactory
        instance ref_threadfactory-bgp {
          name ref_threadfactory-bgp
          provider /modules/module[type='threadfactory-naming'][name=
'threadfactory-bgp']
        }
      }
      service md-sal-binding:binding-broker-osgi-registry {
        type md-sal-binding:binding-broker-osgi-registry
        instance ref_binding-broker-singleton {
          name ref_binding-broker-singleton
          provider /modules/module[type='binding-broker-impl-singleton'][name=
'binding-broker-singleton']
        }
      }
    }
  }
}
```

Configuring fixed threadpool

Service interface `threadpool` is defined in the `config-threadpool-api`. The implementation used is called `threadpool-fixed` that is defined in `config-threadpool-impl`. This

implementation creates a threadpool of fixed size. There are two mandatory attributes: size and dependency on a threadfactory.

1. Issue `get-config source=running`. As you can see in the last step of configuring threadfactory, `/services/service`, the node associated with it has instance name `ref_threadfactory-bgp`.
2. Issue `merge /modules/module`.
3. Enter the name `bgp-threadpool`.
4. Enter the type `threadpool`.
5. Select the appropriate case statement.
6. Enter the value for leaf `<max-thread-count>`: 100.
7. Enter the threadfactory for attribute `threadfactory/type`. This is with reference to `/services/service/type`, in other words, the service interface.
8. Enter `ref_threadfactory-bgp`. Server response must be an OK message.
9. Issue `commit`.
10. Issue `get-config source=running`. Example output:

```
rpc-reply {
  data {
    modules {
      module binding-broker-singleton {
        type binding-impl:binding-broker-impl-singleton
        name binding-broker-singleton
      }
      module bgp-threadpool {
        type th-java:threadpool-fixed
        name bgp-threadpool
        threadFactory {
          type th:threadfactory
          name ref_threadfactory-bgp
        }
        max-thread-count 100
      }
      module threadfactory-bgp {
        type th-java:threadfactory-naming
        name threadfactory-bgp
        name-prefix bgp
      }
    }
    services {
      service th:threadpool {
        type th:threadpool
        instance ref_bgp-threadpool {
          name ref_bgp-threadpool
          provider /modules/module[type='threadpool-fixed'][name='bgp-
threadpool']
        }
      }
    }
  }
}
```

```

    service th:threadfactory {
      type th:threadfactory
      instance ref_threadfactory-bgp {
        name ref_threadfactory-bgp
        provider /modules/module[type='threadfactory-naming'][name=
'threadfactory-bgp']
      }
    }
    service md-sal-binding:binding-broker-osgi-registry {
      type md-sal-binding:binding-broker-osgi-registry
      instance ref_binding-broker-singleton {
        name ref_binding-broker-singleton
        provider /modules/module[type='binding-broker-impl-singleton'][name=
'binding-broker-singleton']
      }
    }
  }
}

```

To see the actual netconf messages, use the logging arguments described at the top of this page. To validate that a threadpool has been created, a tool like VisualVM can be used.

Logback configuration - Yuma

This approach to configure logback will utilize a 3rd party cli netconf client from Yuma. We will modify existing console appender in logback and then call reset rpc on logback to clear its status list.

For initial configuration of the controller and startup parameters for yuma, see the threadpool example: [Threadpool configuration using Yuma](#).

Start the controller and yuma cli client as in the previous example.

There is no need to initialize the configuration module wrapping logback manually, since it creates a default instance. Therefore you should see the output containing logback configuration after the execution of get-config source=running command in yuma:

```

rpc-reply {
  data {
    modules {
      module singleton {
        type logging:logback
        name singleton
        console-appenders {
          threshold-filter ERROR
          name STDOUT
          encoder-pattern '%date{"yyyy-MM-dd HH:mm:ss.SSS z"} [%thread]
%-5level %logger{36} - %msg%n'
        }
        file-appenders {
          append true
          file-name logs/audit.log
          name audit-file
          encoder-pattern '%date{"yyyy-MM-dd HH:mm:ss.SSS z"} %msg %n'
        }
      }
    }
  }
}

```

```

        level WARN
        logger-name org.opendaylight.controller.logging.bridge
    }
    loggers {
        level INFO
        logger-name audit
        appenders audit-file
    }
    loggers {
        level ERROR
        logger-name ROOT
        appenders STDOUT
        appenders.opendaylight.log
    }
    loggers {
        level INFO
        logger-name org.opendaylight
    }
    loggers {
        level WARN
        logger-name io.netty
    }
    rolling-appenders {
        append true
        max-file-size 10MB
        file-name logs/opendaylight.log
        name.opendaylight.log
        file-name-pattern logs/opendaylight.%d.log.zip
        encoder-pattern '%date{"yyyy-MM-dd HH:mm:ss.SSS z"} [%thread]
%-5level %logger{35} - %msg%n'
        clean-history-on-start false
        max-history 1
        rolling-policy-type TimeBasedRollingPolicy
    }
}
module binding-broker-singleton {
    type binding-impl:binding-broker-impl-singleton
    name binding-broker-singleton
}
}
services {
    service md-sal-binding:binding-broker-osgi-registry {
        type md-sal-binding:binding-broker-osgi-registry
        instance ref_binding-broker-singleton {
            name ref_binding-broker-singleton
            provider /modules/module[type='binding-broker-impl-singleton'][name=
'binding-broker-singleton']
        }
    }
}
}
}
}

```

Modifying existing console appender in logback

1. Start edit-config with merge option:

```
merge /modules/module
```

1. For Name of the module, enter **singleton**.

2. For Type, enter **logback**.

3. Pick the corresponding case statement with the name logback. We do not want to modify file-appenders, rolling-appenders and loggers lists, so the answer to questions from yuma is N (for no):

```
Filling optional case /modules/module/configuration/logback:
Add optional list 'file-appenders'?
Enter Y for yes, N for no, or C to cancel: [default: Y]
```

1. As we want to modify console-appenders, the answer to the question from Yuma is Y:

```
Filling optional case /modules/module/configuration/logback:
Add optional list 'console-appenders'?
Enter Y for yes, N for no, or C to cancel: [default: Y]
```

1. This will start a new configuration process for console appender and we will fill following values:

- <encoder-pattern> %date{"yyyy-MM-dd HH:mm:ss.SSS z"} %msg %n
- <threshold-filter> INFO
- <name> STDOUT

2. Answer N to the next question.

```
Add another list?
Enter Y for yes, N for no, or C to cancel: [default: N]
```

Notice that we changed the level for threshold-filter for STDOUT console appender from ERROR to INFO. Now issue a commit command to commit the changed configuration, and the response from get-config source=running command should look like this:

```
rpc-reply {
  data {
    modules {
      module singleton {
        type logging:logback
        name singleton
        console-appenders {
          threshold-filter INFO
          name STDOUT
          encoder-pattern '%date{"yyyy-MM-dd HH:mm:ss.SSS z"} [%thread]
%-5level %logger{36} - %msg%n'
        }
        file-appenders {
          append true
          file-name logs/audit.log
          name audit-file
          encoder-pattern '%date{"yyyy-MM-dd HH:mm:ss.SSS z"} %msg %n'
        }
        loggers {
          level WARN
          logger-name org.opendaylight.controller.logging.bridge
        }
        loggers {
```

```

        level INFO
        logger-name audit
        appenders audit-file
    }
    loggers {
        level ERROR
        logger-name ROOT
        appenders STDOUT
        appenders opendaylight.log
    }
    loggers {
        level INFO
        logger-name org.opendaylight
    }
    loggers {
        level WARN
        logger-name io.netty
    }
    rolling-appenders {
        append true
        max-file-size 10MB
        file-name logs/opendaylight.log
        name opendaylight.log
        file-name-pattern logs/opendaylight.%d.log.zip
        encoder-pattern '%date{"yyyy-MM-dd HH:mm:ss.SSS z"} [%thread]
%-5level %logger{35} - %msg%n'
        clean-history-on-start false
        max-history 1
        rolling-policy-type TimeBasedRollingPolicy
    }
}
module binding-broker-singleton {
    type binding-impl:binding-broker-impl-singleton
    name binding-broker-singleton
}
}
services {
    service md-sal-binding:binding-broker-osgi-registry {
        type md-sal-binding:binding-broker-osgi-registry
        instance ref_binding-broker-singleton {
            name ref_binding-broker-singleton
            provider /modules/module[type='binding-broker-impl-singleton'][name=
'binding-broker-singleton']
        }
    }
}
}
}
}

```

Invoking RPCs

Invoking Reset RPC on logback The configuration module for logback exposes some information about its current state(list of logback status messages). This information can be accessed using get netconf operation or get command from yuma. Example response after issuing get command in yuma:

```

rpc-reply {
  data {
    modules {

```

```

module singleton {
  type logging:logback
  name singleton
  status {
    message 'Found resource [configuration/logback.xml] at
[file:/.../controller/.opendaylight/distribution/.opendaylight/target/
distribution.opendaylight-
osgipackage/.opendaylight/configuration/logback.xml]'
    level INFO
    date 2479534352
  }
  status {
    message 'debug attribute not set'
    level INFO
    date 2479534441
  }
  status {
    message 'Will scan for changes in
[[/.../controller/.opendaylight/distribution/.opendaylight/target/distribution.
opendaylight-
osgipackage/.opendaylight/configuration/logback.xml]]
every 60 seconds.'
    level INFO
    date 2479534448
  }
  status {
    message 'Adding ReconfigureOnChangeFilter as a turbo filter'
    level INFO
    date 2479534448
  }
  ...
}

```

Logback also exposes an rpc called reset that wipes out the list of logback status messages and to invoke an rpc with name reset on module named singleton of type logback, following command needs to be issued in yuma:

```
reset context-instance="/modules/module[type='logback' and name='singleton']"
```

After an ok response, issuing get command should produce response with empty logback status message list:

```

rpc-reply {
  data {
    modules {
      module singleton {
        type logging:logback
        name singleton
      }
    }
  }
}

```

This response confirms successful execution of the reset rpc on logback.

Invoking shutdown RPC This command entered in yuma will shut down the server. If all bundles do not stop correctly within 10 seconds, it will force the process to exit.

```
shutdown context-instance="/modules/module[type='shutdown' and name=
'shutdown']",input-secret="",max-wait-time="10000",reason="reason"
```

OpenDaylight Controller Configuration: Logback Examples

Logback Configuration Example

The Logback logger configuration is part of the config subsystem. This module allows changes to the Logback configuration at runtime. It is used here as an example to demonstrate the YANG to Java code generator and to show how the configuration transaction works.

Java code generation

The logging configuration YANG module definition can be found in the config-logging.yang file. The code is generated by the yang-maven-plugin and yang-jmx-generator-plugin. The output java files are located as defined in the plugin configuration, where additional configuration parameters can be set. The logback module is defined as identity, with the base "config:module-type"; it does not provide or depend on any service interface.

```
identity logback {
  description
    "Actual state of logback configuration.";
  base config:module-type;
  config:java-name-prefix Logback;
}
```

The next logback module attributes are defined in the "/config:modules/config:module/config:configuration" augment as the snippet below shows.

```
augment "/config:modules/config:module/config:configuration" {
  case logback {
    when "/config:modules/config:module/config:type = 'logback'";

    list console-appenders {

      leaf encoder-pattern {
        type string;
        mandatory true;
      }

      leaf threshold-filter {
        type string;
        default 'ALL';
      }

      leaf name {
        type string;
        mandatory true;
      }
      config:java-name-prefix ConsoleAppenderTO;
    }
    ...
  }
}
```

Now LogbackModule and LogbackModuleFactory can be generated. In fact, three more java files related to this module will be generated. By the augment definition, TypeObjects

too are generated (that is to say, ConsoleAppenderTO). They are regular java classes with getters and setters for arguments defined as leaves.

- **LogbackModuleMXBean** is the interface containing getters and setters for attributes defined in the configuration augment.
- **AbstractLogbackModule** is the abstract java class, which implements Module, RuntimeBeanRegistrarAwareModule, and LogbackModuleMXBean. It contains almost all functionality, except validate and createInstance methods.
- **AbstractLogbackModuleFactory** is the abstract java class responsible for creating module instances. It implements the ModuleFactory interface.
- **LogbackModule** class extends AbstractLogbackModule. It is located in a different place (source/main/java) and can be modified by the user, so that the abstract method is implemented and the validate method is overridden.
- **LogbackModuleFactory** class extends AbstractLogbackModuleFactory and overrides its instantiateModule methods. Next, the runtime bean is defined in the "/config:modules/config:module/config:state" augment.

```
augment "/config:modules/config:module/config:state" {
  case logback {
    when "/config:modules/config:module/config:type = 'logback'";

    rpcx:rpc-context-instance "logback-rpc";

    list status {
      config:java-name-prefix StatusTO;

      leaf level {
        type string;
      }

      leaf message {
        type string;
      }

      leaf date {
        type uint32;
      }
    }
  }
}
```

- The **Generator** plugin creates another set of java files.
- **LogbackRuntimeMXBean** is the interface extending RuntimeBean. It contains the getter method for the argument defined in the augment.
- **LogbackRuntimeRegistrar** class serves as the registrar for runtime beans.
- **LogbackRuntimeRegistration** class serves as the registration ticket. An instance is returned after registration.

The Logback config also defines logback-rpc with the reset method. It is also defined in the state augment, owing to the context.


```
identity logback-rpc;  
rpc reset {  
  input {  
    uses rpcx:rpc-context-ref {  
      refine context-instance {  
        rpcx:rpc-context-instance logback-rpc;  
      }  
    }  
  }  
}
```

The Reset method is defined in the LogbackRuntimeMXBean interface.

Logback configuration: Jolokia

To create configuration on the running OSGi server: Jolokia (<http://www.jolokia.org/>) is used as a JMX-HTTP bridge, which listens at <http://localhost:8080/controller/nb/v2/jolokia> and curl to request over HTTP.

1. Start the controller. Find more here: https://wiki.opendaylight.org/view/OpenDaylight_Controller:Pulling,_Hacking,_and_Pushing_the_Code_from_the_CLI

2. Request Jolokia:

```
curl http://localhost:8080/controller/nb/v2/jolokia --user admin:admin
```

The response must resemble the following:

```
{  
  "timestamp": 1382425537,  
  "status": 200,  
  "request": {  
    "type": "version"  
  },  
  "value": {  
    "protocol": "7.0",  
    "agent": "1.1.1",  
    "info": {  
      "product": "equinox",  
      "vendor": "Eclipse",  
      "version": "3.8.1.v20120830-144521"  
    }  
  }  
}
```

Jolokia is working. To configure Logback, first, create a configuration transaction. ConfigRegistryModule offers the operation beginConfig(), and to invoke it:

```
curl -X POST -H "Content-Type: application/json" -d '{"type":"exec",  
"mbean":"org.opendaylight.controller:type=ConfigRegistry", "arguments":[],  
"operation":"beginConfig"}' http://localhost:8080/controller/nb/v2/jolokia --  
user admin:admin
```

The configuration transaction was created. The response received:

```
{  
  "timestamp": 1383034210,  
  "status": 200,
```

```

"request": {
  "operation": "beginConfig",
  "mbean": "org.opendaylight.controller:type=ConfigRegistry",
  "type": "exec"
},
"value": {
  "objectName": "org.opendaylight.controller:TransactionName=
ConfigTransaction-1-2,type=ConfigTransaction"
}
}

```

At this stage, the transaction can be aborted, but we want to create the module bean to be configured. In the created ConfigTransaction call createModule method, the module identifier is logback, and the name must be singleton as only one instance of the Logback configuration is needed.

```

curl -X POST -H "Content-Type: application/json" -d '{"type":"exec",
"mbean":"org.opendaylight.controller:TransactionName=ConfigTransaction-1-2,
type=ConfigTransaction","arguments":["logback","singleton"],
"operation":"createModule"}' http://localhost:8080/controller/nb/v2/jolokia --
user admin:admin

```

The LogbackModule bean was created. The response returned:

```

{
  "timestamp": 1383034580,
  "status": 200,
  "request": {
    "operation": "createModule",
    "mbean": "org.opendaylight.controller:TransactionName=
ConfigTransaction-1-2,type=ConfigTransaction",
    "arguments": [
      "logback",
      "singleton"
    ],
    "type": "exec"
  },
  "value": {
    "objectName": "org.opendaylight.controller:TransactionName=
ConfigTransaction-1-2,instanceName=singleton,moduleFactoryName=logback,type=
Module"
  }
}

```

- The configuration bean attributes are set to values obtained from the loggers configuration, with which the server was started. To see attributes, request:

```

curl -X POST -H "Content-Type: application/json" -d '{"type":"read",
"mbean":"org.opendaylight.controller:instanceName=singleton,TransactionName=
ConfigTransaction-1-2,type=Module,moduleFactoryName=logback"}' http://
localhost:8080/controller/nb/v2/jolokia --user admin:admin

```

In the response body, the value contains all attributes (CompositeData) and its nested attribute values. * Now, the proposed configuration can be committed.

```

curl -X POST -H "Content-Type: application/json" -d '{"type":"exec",
"mbean":"org.opendaylight.controller:type=ConfigRegistry","arguments":
["org.opendaylight.controller:instanceName=singleton,TransactionName=
ConfigTransaction-1-2,type=Module,moduleFactoryName=logback"],

```

```
"operation": "commitConfig"}' http://localhost:8080/controller/nb/v2/jolokia --user admin:admin
```

The configuration was successfully validated and committed, and the module instance created.

```
{
  "timestamp": 1383034793,
  "status": 200,
  "request": {
    "operation": "commitConfig",
    "mbean": "org.opendaylight.controller:type=ConfigRegistry",
    "arguments": [
      "org.opendaylight.controller:instanceName=singleton,
TransactionName=ConfigTransaction-1-2,type=Module,moduleFactoryName=logback"
    ],
    "type": "exec"
  },
  "value": {
    "newInstances": [
      {
        "objectName": "org.opendaylight.controller:instanceName=singleton,moduleFactoryName=logback,type=Module"
      }
    ],
    "reusedInstances": [],
    "recreatedInstances": []
  }
}
```

- The runtime bean was registered, and can provide the status information of the configuration and rpc operation reset. To see the status, try requesting:

```
curl -X POST -H "Content-Type: application/json" -d '{"type": "read",
"mbean": "org.opendaylight.controller:instanceName=singleton,type=RuntimeBean,
moduleFactoryName=logback"}' http://localhost:8080/controller/nb/v2/jolokia --user admin:admin
```

The entire logback status is in the response body.

- To invoke the rpc method reset:

```
curl -X POST -H "Content-Type: application/json" -d '{"type": "exec",
"mbean": "org.opendaylight.controller:instanceName=singleton,type=RuntimeBean,
moduleFactoryName=logback",
"operation": "reset", "arguments": []}' http://localhost:8080/controller/nb/v2/jolokia --user admin:admin
```

The answer:

```
{
  "timestamp": 1383035001,
  "status": 200,
  "request": {
    "operation": "reset",
    "mbean": "org.opendaylight.controller:instanceName=singleton,
moduleFactoryName=logback,type=RuntimeBean",
    "type": "exec"
  },
  "value": null
}
```

```
}
```

Now, the runtime bean status attribute will be empty:

```
{
  "timestamp": 1383035126,
  "status": 200,
  "request": {
    "mbean": "org.opendaylight.controller:instanceName=singleton,
moduleFactoryName=logback,type=RuntimeBean",
    "type": "read"
  },
  "value": {
    "StatusTO": []
  }
}
```

Logback configuration: Netconf

In this case, NETCONF RPCs are used to configure logback. The Netconf server listens at port 8383. To communicate over TCP, telnet is used. More about NETCONF is available at: <http://tools.ietf.org/html/rfc6241>. Netconf implementation is a part of the Controller - netconf-subsystem. The RPCs of Netconf are XML, and the operations are mapped to JMX operations. * A server re-start is required. The procedure is the same as above. * Open a terminal and connect to the server:

```
telnet localhost 8383
```

A Hello message received from the server contains the server capabilities and session-id. To establish connection to the client, send a hello message:

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
  </capabilities>
</hello>
]]>]]>
```

- With the connection created, the client and server can communicate. To see the running modules and services, send an RPC to the server:

```
<rpc id="a" a="64" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <get-config>
    <source>
      <running/>
    </source>
  </get-config>
</rpc>
]]>]]>
```

- To configure logback, create a configuration transaction, and create a configuration module. It can be done in one step (in client point of view):

```
<rpc message-id="a" a="64" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <candidate/>
    </target>
  </edit-config>
</rpc>
```

```

        </target>
        <default-operation>merge</default-operation>
        <config>
            <modules xmlns=
"urn:.opendaylight:params:xml:ns:yang:controller:config">
                <module>
                    <name>singleton</name>
                    <type xmlns:logging=
"urn:.opendaylight:params:xml:ns:yang:controller:logback:config">
                        logging:logback
                    </type>
                </module>
            </modules>
        </config>
    </edit-config>
</rpc>
]]>]]>

```

If the configuration worked, the client receives a positive response:

```

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
<ok/>
</rpc-reply>
]]>]]>

```

- The Logback configuration bean attributes contain values loaded from the running Logback configuration. Send a request to the server with an RPC:

```

<rpc id="a" a="64" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
    <get-config>
        <source>
            <candidate/>
        </source>
    </get-config>
</rpc>
]]>]]>

```

- The reply includes the entire configuration that started the server. Assume that we want to change the RollingFileAppender named.opendaylight.log attributes - maxFileSize, filename, and maxHistory. (attribute of TimeBasedRollingPolicy). The proposed configuration:

```

<rpc message-id="a" a="64" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <edit-config>
        <target>
            <candidate/>
        </target>
        <default-operation>merge</default-operation>
        <config>
            <modules xmlns=
"urn:.opendaylight:params:xml:ns:yang:controller:config">
                <module>
                    <name>singleton</name>
                    <type xmlns:logging=
"urn:.opendaylight:params:xml:ns:yang:controller:logback:config">
                        logging:logback
                    </type>
                    <rolling-appenders xmlns=
"urn:.opendaylight:params:xml:ns:yang:controller:logback:config">

```

```

<append>true</append>
<max-file-size>5MB</max-file-size>
<file-name>logs/opendaylight-new.log</file-name>
<name>opendaylight.log</name>
<file-name-pattern>logs/opendaylight.%d.log.zip</file-name-pattern>
<encoder-pattern>%date{"yyyy-MM-dd HH:mm:ss.SSS z"} [%thread] %-5level
%logger{35} - %msg%n</encoder-pattern>
<clean-history-on-start>false</clean-history-on-start>
<max-history>7</max-history>
<rolling-policy-type>TimeBasedRollingPolicy</rolling-policy-type>
  </rolling-appenders>
  </module>
</modules>
</config>
</edit-config>
</rpc>
]]>]]>

```

This configuration is merged with the proposed module configuration. If it passes the validation process successfully, an "ok" reply is received.

- The configuration bean is ready to be committed:

```

<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <commit></commit>
</rpc>
]]>]]>

```

If successful, the ok message is received obtained, and the logback configuration is set. To verify, look into the logs directory to find a new log file named opendaylight-new.log

- Correctly close the session with the session-id:

```

<rpc message-id="2" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <close-session xmlns="urn:ietf:params:xml:ns:netconf:base:1.0"/>
</rpc>
]]>]]>

```

Logback configuration - Yuma

For a yangcli-pro example, see the [user guide](#).

Openaylight Controller: Configuration Logback.xml

Logging in ODL container is done by means of [Logback](#). Comprehensive documentation is available at <http://logback.qos.ch/documentation.html>.

By default, logging messages are appended to stdout of the java process and to file logs/opendaylight.log. When debugging a problem it might be useful to increase logging level:

```
<logger name="org.opendaylight.controller" level="DEBUG"/>
```

Logger tags can be appended under root node <configuration/>. Name of logger is used to select all loggers to which specified rules should apply. Loggers are usually named after

class in which they reside. The example above matches all loggers in controller - they all are starting with `org.opendaylight.controller`. There are 5 logging levels: TRACE, DEBUG, INFO, WARN, ERROR. Additionally one can specify which appenders should be used for given loggers. This might be helpful to redirect certain log messages to another file or send them to syslog or over SMTP. == OpenDaylight Controller Configuration: Examples of Threadpool

Configuration example of thread pools using yangcli-pro

For a yangcli-pro example, see the [Examples User Guide](#).

Configuration example of thread pools using telnet

It is also possible to manipulate the configuration without the yuma cli. With just a telnet or ssh connection, it is possible to send the plain text containing netconf rpcs encoded in the xml format and achieve the same results as with yuma cli.

This example reproduces the configuration of a threadpool and a threadfactory from the previous example using just a telnet connection. We can also use ssh connection, with the netconf rpcs sending procedure remaining the same. For detailed information about initial configuration for the controller as well as the configuration process, see the example using yuma cli.

Connecting to plaintext TCP socket

1. Open a telnet connection:

```
telnet 127.0.0.1 8383
```

1. Open an ssh connection:

```
ssh netconf@127.0.0.1 -p 1830 -s netconf
```

The password for user netconf is : netconf, and the separator for the messages is:

```
]]>]]>
```

Every message needs end with these 6 characters.

The server sends a hello message:

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
<capability>urn:ietf:params:netconf:base:1.0</capability>
<capability>urn:ietf:params:netconf:capability:exi:1.0</capability>
<capability>urn:opendaylight:l2:types?module=opendaylight-l2-types&
&revision=2013-08-27</capability>
<capability>urn:opendaylight:params:xml:ns:yang:controller:netty:threadgroup?
module=threadgroup&revision=2013-11-07</capability>
<capability>urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding?
module=opendaylight-md-sal-binding&revision=2013-10-28</capability>
```

```

<capability>urn:opendaylight:params:xml:ns:yang:controller:threadpool?module=
threadpool&revision=2013-04-09</capability>
<capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
<capability>urn:opendaylight:params:xml:ns:yang:controller:config?module=
config&revision=2013-04-05</capability>
<capability>urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring?module=ietf-
netconf-monitoring&revision=2010-10-04</capability>
<capability>urn:opendaylight:params:xml:ns:yang:controller:netty:eventexecutor?
module=netty-event-executor&revision=2013-11-12</capability>
<capability>urn:ietf:params:xml:ns:yang:rpc-context?module=rpc-context&
&revision=2013-06-17</capability>
<capability>urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl?
module=opendaylight-sal-binding-broker-impl&revision=2013-10-28</
capability>
<capability>urn:opendaylight:params:xml:ns:yang:controller:netty:timer?module=
netty-timer&revision=2013-11-19</capability>
<capability>urn:ietf:params:xml:ns:yang:ietf-inet-types?module=ietf-inet-
types&revision=2010-09-24</capability>
<capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0</
capability>
<capability>urn:opendaylight:params:xml:ns:yang:controller:threadpool:impl?
module=threadpool-impl&revision=2013-04-05</capability>
<capability>urn:ietf:params:xml:ns:yang:ietf-yang-types?module=ietf-yang-
types&revision=2010-09-24</capability>
<capability>urn:opendaylight:params:xml:ns:yang:controller:logback:config?
module=config-logging&revision=2013-07-16</capability>
<capability>urn:opendaylight:params:xml:ns:yang:iana?module=iana&revision=
2013-08-16</capability>
<capability>urn:opendaylight:yang:extension:yang-ext?module=yang-ext&
&revision=2013-07-09</capability>
<capability>urn:opendaylight:params:xml:ns:yang:controller:netty?module=netty&
&revision=2013-11-19</capability>
<capability>http://netconfcentral.org/ns/toaster?module=toaster&revision=
2009-11-20</capability>
<capability>urn:opendaylight:params:xml:ns:yang:ieee754?module=ieee754&
&revision=2013-08-19</capability>
<capability>urn:opendaylight:params:xml:ns:yang:nps-concepts?module=nps-
concepts&revision=2013-09-30</capability>
</capabilities>

<session-id>4</session-id>
</hello>
]]>]]>

```

1. As the client, you must respond with a hello message:

```

<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
  </capabilities>
</hello>
]]>]]>

```

Although there is no response to the hello message, the session is already established.

Configuring threadfactory

1. The following is the Xml equivalent to **get-config source=running**:


```
<rpc xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <get-config>
    <source>
      <running/>
    </source>
  </get-config>
</rpc>
]]>]]>
```

The response containing the current configuration:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <data>
    <modules xmlns="urn:.opendaylight:params:xml:ns:yang:controller:config">
      <module>
        <type xmlns:prefix=
"urn:.opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl">prefix:binding-
broker-impl-singleton</type>
        <name>binding-broker-singleton</name>
      </module>
    </modules>
    <services xmlns="urn:.opendaylight:params:xml:ns:yang:controller:config">
      <service>
        <type xmlns:prefix=
"urn:.opendaylight:params:xml:ns:yang:controller:md:sal:binding">prefix:binding-
broker-osgi-registry</type>
        <instance>
          <name>ref_binding-broker-singleton</name>
          <provider>/modules/module[type='binding-broker-impl-singleton'][name=
'binding-broker-singleton']</provider>
        </instance>
      </service>
    </services>
  </data>
</rpc-reply>]]>]]>
```

1. To create an instance of threadfactory-naming with the name threadfactory-bgp, and the attribute name-prefix set to bgp, send the message:

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <candidate/>
    </target>
    <default-operation>merge</default-operation>
    <config>
      <modules xmlns="urn:.opendaylight:params:xml:ns:yang:controller:config">
        <module xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" nc:operation=
"merge">
          <name>threadfactory-bgp</name>
          <type xmlns:th-java=
"urn:.opendaylight:params:xml:ns:yang:controller:threadpool:impl">th-
java:threadfactory-naming</type>
          <name-prefix xmlns=
"urn:.opendaylight:params:xml:ns:yang:controller:threadpool:impl">bgp</name-
prefix>
        </module>
      </modules>
    </config>
  </edit-config>
```

```
</rpc>]]>]]>
```

1. To commit the threadfactory instance, send a commit message:

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <commit/>
</rpc>]]>]]>
```

The Netconf endpoint should respond with ok to edit-config, as well as the commit message:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <ok/>
</rpc-reply>]]>]]>
```

1. The response to the get-config message (the same as the first message sent in this example) should contain the committed instance of threadfactory-naming:

```
<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" message-id="101">
  <data>
    <modules xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
      <module>
        <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding:impl">prefix:binding-
broker-impl-singleton</type>
        <name>binding-broker-singleton</name>
      </module>

      <module>
        <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:threadpool:impl">prefix:threadfactory-
naming</type>
        <name>threadfactory-bgp</name>
        <name-prefix xmlns=
"urn:opendaylight:params:xml:ns:yang:controller:threadpool:impl">bgp</name-
prefix>
      </module>
    </modules>

    <services xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
      <service>
        <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:threadpool">prefix:threadfactory</
type>
        <instance>
          <name>ref_threadfactory-bgp</name>
          <provider>/modules/module[type='threadfactory-naming'] [name=
'threadfactory-bgp']</provider>
        </instance>
      </service>
      <service>
        <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">prefix:binding-
broker-osgi-registry</type>
        <instance>
          <name>ref_binding-broker-singleton</name>
          <provider>/modules/module[type='binding-broker-impl-singleton'] [name=
'binding-broker-singleton']</provider>
        </instance>
```

```

    </service>
  </services>
</data>
</rpc-reply>]]>]]>

```

Configuring fixed threadpool

- To create an instance of **threadpool-fixed**, with the same configuration and the same dependency as before, send the following message:

```

<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
    <target>
      <candidate/>
    </target>
    <default-operation>merge</default-operation>
    <config>
      <modules xmlns="urn:.opendaylight:params:xml:ns:yang:controller:config">
        <module xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0" nc:operation=
"merge">
          <name>bgp-threadpool</name>
          <type xmlns:th-java=
"urn:.opendaylight:params:xml:ns:yang:controller:threadpool:impl">th-
java:threadpool-fixed</type>
          <max-thread-count xmlns=
"urn:.opendaylight:params:xml:ns:yang:controller:threadpool:impl">100</max-
thread-count>
          <threadFactory xmlns=
"urn:.opendaylight:params:xml:ns:yang:controller:threadpool:impl">
            <type xmlns:th=
"urn:.opendaylight:params:xml:ns:yang:controller:threadpool">th:threadfactory</
type>
              <name>ref_th-bgp</name>
            </threadFactory>
          </module>
        </modules>

        <services xmlns="urn:.opendaylight:params:xml:ns:yang:controller:config">
          <service>
            <type xmlns:prefix=
"urn:.opendaylight:params:xml:ns:yang:controller:threadpool">prefix:threadfactory</
type>
              <instance>
                <name>ref_th-bgp</name>
                <provider>/modules/module[type='threadfactory-naming'][name=
'threadfactory-bgp']</provider>
              </instance>
            </service>
          </services>
        </config>
      </edit-config>
    </rpc>]]>]]>

```

Notice the *services* tag. If an instance is to be referenced as a dependency by another module, it needs to be placed under this tag as a service instance with a unique reference name. Tag *provider* points to a unique instance that is already present in the config subsystem, or is created within the current edit-config operation. The tag *name* contains the reference name that can be referenced by other modules to create a dependency. In

this case, a new instance of threadpool uses this reference in its configuration under the *threadFactory* tag).

You should get an ok response again, and the configuration subsystem will inject the dependency into the threadpool. Now you can commit the configuration (ok response once more) and the process is finished. The config subsystem is now in the same state as it was at the end of the previous example.

OpenDaylight Controller MD-SAL: Model reference

A full list of models, with links to the yang, descriptions, JavaDoc and REST APIs, see the OpenDaylight wiki page here: https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Model_Reference

6. Defense4all

Table of Contents

| | |
|---|-----|
| Defense4All Design | 132 |
| Defense4All in an ODL Environment | 133 |
| Framework View | 134 |
| Application View | 136 |
| ODL Reps View | 138 |
| Basic Control Flow | 141 |
| Configurations and Setup Flow | 141 |
| Attack Detection Flow | 142 |
| Attack Mitigation Flow | 142 |
| Continuity | 143 |

Defense4All Design

Defense4All is a security SDN application for detecting and driving mitigation of DoS and DDoS attacks in different SDN topologies. It realizes anti-DoS in OOP mode for the ProgrammableFlow SDN environment. Administrators can configure Defense4All to protect certain networks and servers, known as protected networks or Protected Objects (POs). Defense4All exploits SDN capabilities to count specified traffic, and installs traffic counting flows for each protocol of each configured PO in every network location (VTN Vexternals) through which traffic of the subject PO flows. Defense4All then monitors traffic of all configured POs, summarizing readings, rates, and averages from all relevant network locations. If it detects a deviation from normal learned traffic behavior in a protocol (such as TCP, UDP, ICMP, or the rest of the traffic) of a particular PO, Defense4All declares an attack against that protocol in the subject PO. The Defense4All learning period has a minimum of one week from the installation of the counting flows in which Defense4All does not detect attacks.

To mitigate a detected attack, Defense4All performs the following procedure:

1. Validates that the DefensePro device is alive and selects a live connection to it (if DefensePro is not alive or does not have a live connection from a PFS, then no traffic diversion is performed. For more information, refer to the Continuity section.
2. Configures DefensePro with a security policy and normal rates of the attacked traffic. The latter speeds up DefensePro's efficient mitigation of the attack.
3. Starts monitoring and logging syslogs arriving from DefensePro for the subject traffic. As long as it continues receiving syslog attack notifications from DefensePro regarding this attack, Defense4All continues attack mitigation through traffic diversion even if the Vexternal FlowFilter counters do not indicate any more attacks.
4. Maps the selected physical DefensePro connection to the relevant VTN by creating a pair of Vexternals and mapping them to the selected pair of physical PFS ports connected to DefensePro. Automatically learns and preserves VLAN tagging if it exists. If Defense4All has already created and mapped a pair of Vexternals with the same VLAN in the VTN,


then the same pair is also reused for diversion of the new traffic (rather than creating new Vexternals for the same VTN and VLAN).

5. Installs higher priority flow filter entries in every north Vexternal through which the attacked traffic flows in order to redirect traffic to the "north DP-In Vexternal". It also selects one of the live north interfaces of the Vbr connected to all those Vexternals (there can be exactly one Vbr with the same VLAN). Defense4All re-injects traffic from the "DP-Out Vexternal" to the selected interface of the Vbr. When Defense4All decides that the attack is over (no indication from either PFC FlowFilter counters or from DefensePro) it reverts the previous actions: it stops monitoring for DefensePro syslogs about the subject traffic, it removes the traffic diversion FlowFilters, removes the "DP-In and DP-Out Vexternals" (if this is the last attack in this VTN and VLAN), and removes the security configuration from DefensePro. Defense4All then returns to peacetime monitoring.

In this version, Defense4All runs as a single instance (non-clustered), but integrates the following main fault tolerance features:

- Runs as a Linux service that is automatically restarted should it fail.
- State entirely persisted in stable storage, and upon restart Defense4All obtains the latest state.
- Carries a health tracker with restart and reset capabilities to overcome certain logical and aging bugs.

Defense4All monitors the status of DefensePro, switch connections to DefensePro, relevant Vbrs in various VTNs, northbound interfaces of those Vbrs, and north Vexternals and adjusts, cancels, and (re)initiates attack traffic diversion accordingly.

The following figure illustrates the possible state of any given PO. Radware's DefensePro (DP) is an example of an incorporated AMS. [Workflow of Defense4All Attack Mitigation]

Defense4All in an ODL Environment

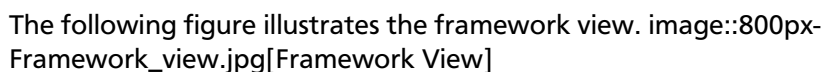
Defense4All comprises of an SDN application framework and the Defense4All application itself, packaged as a single entity. Application integration into the framework is pluggable, so any other SDN application can benefit from the common framework services.

The main advantages of this architecture are:

- Faster application development and changes - The Framework contains common code for multiple applications, complex elements (such as clustering and repository services) are implemented once for the benefit of any application.
- Faster, flexible deployment in different environments, form-factors, satisfying different NFRs – The Framework masks from SDN application factors such as required data survivability, scale and elasticity, availability, security.
- Enhanced robustness - Complex framework code is implemented and tested once, cleaner separation of concerns leads to more stable code, and the framework can increase robustness proactively with no additional code in the application logic (such as periodic application recycle).

- Unified management of common aspects – Common look and feel.

Framework View

The following figure illustrates the framework view.  image::800px-Framework_view.jpg[Framework View]

The framework contains the following elements:

FrameworkMain – The Framework root point contains references to all Framework modules and global repositories, as well as the roots of deployed SDN applications (in the current version, the framework can accommodate only one application). This is also the point to start, stop, or reset the framework (along with its hosted application) Web server, Jetty Web server running the Jersey RESTful Web services framework, with Jackson parser for JSON encoded parameters. The REST Web server runs a servlet for the framework and another servlet for each deployed application (currently only one). All REST and CLI APIs are supported through this REST Web server.

FrameworkRestService – A set of classes constituting the framework servlet that responds to framework REST requests (get latest Flight Recorder records, perform factory reset, and so on). The FrameworkRestService invokes control and configuration methods against the FrameworkMgmtPoint, and for reporting it retrieves information directly from the relevant repositories. For flight recordings, it invokes methods against the FlightRecorder.

FrameworkMgmtPoint – The point to drive control and configuration commands (start, stop, reset, set address of the hosting machine, and so on). FrameworkMgmtPoint in turn invokes methods against other relevant modules in the correct order. It forwards lifecycle requests (start, stop, reset) directly to FrameworkMain to drive them in the correct order.

Defense4All Application – The AppRoot object that should be implemented/extended by any SDN application (in this case, Defense4All). SDN applications do not have “main,” and their lifecycle (start, stop, reset) is managed by the framework operating against the application root object, which then drives all lifecycle operations in the application. This module also contains references back to the framework, allowing the application to use framework services (such as create a Repo and log a flight record) and common utilities.

Common classes and Utilities – A library of convenient classes and utilities from which any framework or SDN application module can benefit. Examples include wrapped threading services (for asynchronous, periodic, or background execution), short hash of a string, and confirmation by user.

Repository services – One of the key elements in the framework philosophy is decoupling the compute state from the compute logic. All durable states should be stored in a set of repositories that can be then replicated, cached, distributed under the covers, with no awareness of the compute logic (framework or application). Repository services comprise the RepoFactory and Repo or its annotations-friendly equivalent – the EntityManager. The RepoFactory is responsible for establishing connectivity with the underlying repository plugged-in service, instantiate new requested repositories, and return references to existing ones. The chosen underlying repository service is Hector Client over Cassandra NoSQL DB. Repo presents an abstraction of a single DB table. It enables reading the whole table, only table keys (tables are indexed by only the single primary key), records or single cells, as well as writing records or single cells with controlled eagerness. A sub-record (with only a portion of cells) may be written. In this case, the displayed cells override existing ones in the

repository. Other cells in the repository remain unchanged. In contrast to a relational DB, in which all columns must be specified up-front (in a schema design), Repo leverages the underlying Cassandra support to contain rows (records) in the same table with different sets of columns, some of which may not being even defined up-front. Furthermore, cells with new columns can be added or removed on the fly. RepoFactory and Repo (as well as its Entity Manager annotation-friendly equivalent) constitute a convenient library targeted to framework and SDN applications goals on top of the Hector client library communicating with Cassandra Repository cluster. Scaling the Cassandra cluster, distributing data shards across Cassandra cluster members, and configuring read/write eagerness and consistency are for the most part encapsulated in this layer.

Logging and Flight Recorder services – The logging service uses Log4J library to log error, warning, trace, or informational messages. These logs are mainly for Defense4All developers. Administrators can obtain additional details about failures from the error log. FlightRecorder records all flight records recorded by any Defense4All module, including information received from external network elements such as ODC and AMSs. It then allows a user or administrator to obtain that information through the REST API or the CLI. Flight records can be filtered by categories (zero or more can be specified) and by time ranges. FlightRecorder stores all flight records in its own Repo (with another repo holding time ranges for efficient time ranges retrieval from the records repo). Because all flight records are stored in Cassandra, the number of flight records Defense4All can keep is limited only by the size of the underlying persistent storage capacity of all Cassandra servers, and so even on a single Cassandra instance, months of historical information can be kept.

HealthTracker – The point to hold the aggregated runtime health of Defense4All and to act in response to severe deteriorations. Any module, upon sensing unexpected and/or faulty behavior in it or in any other module can record a “health issue” in the HealthTracker, providing health issue significance. This is instead of directly triggering a Defense4All termination. This means that numerous health issues in a short period with high aggregated significance are likely to indicate a significant wide-spread Defense4All problem, but sporadic and/or intermittent operational “hiccups” can be neglected, even if Defense4All remains less than 100% operational (the administrator can always reset it to fully recover). As a result, every non-permanent health issue has a gradually diminished effect over time. If Defense4All health deteriorates below a predefined threshold, HealthTracker triggers responsive actions depending on the nature of the health issues. A restart can heal transient problems, and so the HealthTracker triggers Defense4All termination (running as a Linux service, Defense4All is automatically restarted). To recover from more permanent problems, HealthTracker may additionally trigger a Defense4All reset. If this does not help, the next time the HealthTracker attempts a more severe reset. As a last resort, the administrator can be advised to perform a factory reset.

ClusterMgr – Currently not implemented. This module is responsible for managing a Defense4All cluster (separate from Cassandra or ODC clusters, modeled as separate tier clusters). A clustered Defense4All carries improved high availability and scalability. Any module in the Defense4All framework or application can register with ClusterMgr for a clustered operation, specifying whether its functionality should be carried out by a single or by multiple/all active instances (running on different Defense4All cluster members). When cluster membership changes, ClusterMgr notifies each instance in each module about its role in the clustered operation of that module. If there is a single active instance, that instance is notified of its role in the cluster, while all other instances are notified that they are in standby mode. If there are multiple active instances, each active instance is notified

about the number of active instances and its logical enumeration in that range. All states are stored in a globally accessible and shared repository, so any instance of a module is stateless, and can perform any role after every membership change. For example, following membership change N, an instance can be enumerated as 2 out of 7, as a result performing the relevant portion of the work. At membership change N+1, the same instance can be enumerated 5 out of 6, and perform the work portion allocated for 5 and not for 2. Peer messaging services are skipped which the ClusterMgr can provide for a more coordinated cross-instance operation.

The Defense4All application is highly pluggable. It can accommodate different attack detection mechanisms, different attack mitigation drivers, and drivers (called reps [representative]) to different versions of the ODC and different AMSs. The Defense4All application comprises “core” modules and “pluggable” modules implementing well-defined Defense4All application APIs.

Application View

The following figure illustrates the application view. image::800px-D4a_application_view.jpg[Application View]

The following is a description of the Defense4All application modules:

DFAppRoot – The root module of the Defense4All application. The Defense4All application does not have “main,” and its lifecycle (start, stop, reset) is managed by the Framework operating against this module, which in turn drives all lifecycle operations in the Defense4All application. DFAppRoot also contains references to all Defense4All application modules (core and pluggable), global repositories, and references back to the framework, allowing the Defense4All application modules to use framework services (such as create a Repo and log a flight record) and common utilities.

DFRestService – A set of classes constituting the Defense4All application servlet that responds to Defense4All application REST requests. The DFRestService invokes control and configuration methods against the DFMgmtPoint, and for reporting it retrieves information directly from the relevant repositories. For flight recordings, it invokes methods against the FlightRecorder.

DFMgmtPoint – The point to drive control and configuration commands (such as addams and addpn). DFMgmtPoint in turn invokes methods against other relevant modules in the right order.

ODL Reps – A pluggable module-set for different versions of the ODC. Comprises two functions in two sub-modules: stats collection for, and traffic diversion of, relevant traffic. These two sub-modules adhere to StatsCollectionRep DvsnRep APIs. ODL Reps is detailed in Figure 6 and the description that follows it.

SDNStatsCollector – Responsible for setting “counters” for every PN at specified network locations (physical or logical). A counter is a set of OpenFlow flow entries in ODC-enabled network switches and routers. The SDNStatsCollector periodically collects statistics from those counters and feeds them to the SDNBasedDetectionMgr (see the description below). The module uses the SDNStatsCollectionRep to both set the counters and read latest statistics from those counters. A stat report consists of read time, counter specification, PN label, and a list of trafficData information, where each trafficData element contains the latest bytes and packet values for flow entries configured for <protocol,port,direction> in

the counter location. The protocol can be {tcp,udp,icmp,other ip}, the port is any Layer 4 port, and the direction can be {inbound, outbound}.

SDNBasedDetectionMgr – A container for pluggable SDN-based detectors. It feeds stat reports received from the SDNStatsCollector to plugged-in SDN based detectors. It also feeds all SDN based detectors notifications from the AttackDecisionPoint (see the description below) about ended attacks (so as to allow reset of detection mechanisms).

RateBasedDetector sub-module – This detector learns for each PN its normal traffic behavior over time, and notifies AttackDecisionPoint (see the description below) when it detects traffic anomalies. For each protocol {TCP, UDP, ICMP, other IPs} of each PN, the RateBasedDetector maintains latest rates and exponential moving averages (baselines) of bytes and packets, as well as last reading time. The detector maintains those values both for each counter as well as the aggregation of all counters for each PN. The organization at two levels of calculations (counter and PN aggregate) allows for better scalability (such as working with clustered ODCs, where each instance is responsible for obtaining statistics from a portion of network switches, and bypassing the ODC single instance image API). Such organizations also enable a more precise stats collection (avoiding the difficulty of collecting all stats during a very small time interval). Stats are processed at the counter level, and periodically aggregated at the PN level. Continuous detections of traffic anomalies cause the RateBasedDetector to notify AttackDecisionPoint about attack detection. Then, absence of anomalies for some period of time causes the detector to stop notifying the AttackDecisionPoint about attack detection. The detector specifies a detection duration within which the detection is valid. After that time, the detection expires but can be “prolonged” with another notification about the same attack.

AttackDecisionPoint – This module is responsible for maintaining attack lifecycles. It can receive attack detections from multiple detectors. Defense4All supports the RateBasedDetector, external detectors (scheduled for future versions), and AMS-based detector reference implementation (over Radware’s DefensePro). In the current version, AttackDecisionPoint fully honors each detection (max detector confidence and max detection confidence). It declares a new attack for every detection of a newly attacked traffic (PN, protocol, and port), and adds more detections for existing (already declared) attacks. The module periodically checks the statuses of all attacks. As long as there is at least one unexpired detection (each detection has an expiration time), the attack is kept declared. If all detections are expired for a given attack the AttackDecisionPoint declares the attack has ended. The module notifies the MitigationMgr (see description below) to start mitigating any new declared attack. It notifies the MitigationMgr to stop mitigating ended attacks, and also notifies the detectionMgr to reset stats calculations for traffic on which an attack has just ended.

MitigationMgr - A container for pluggable mitigation drivers. The MitigationMgr maintains the lifecycle of all mitigations as a result of mitigation notifications from AttackDecisionPoint. It holds a pre-ordered list of the MitigationDriver sub-modules, and attempts to satisfy each mitigation in that order. If MitigationDriver_i indicates to MitigationMgr that it does not mitigate a mitigation (because of per PN preferences, unavailability of AMS resources, network problems, and so on) MitigationMgr will attempt mitigation by MitigationDriver_{i+1}. If none of the plugged-in MitigationDrivers handle mitigation, it remains at the status ‘not-mitigated.’

MitigationDriverLocal – This mitigation driver is responsible for driving attack mitigations using AMSs in their sphere of management. When requested to mitigate an attack, this mitigator performs the following sequence of steps:

1. Consults with the plugged in DvsnRep (see description below) about topologically feasible options of diversion for each of the managed AMSs from each of the relevant network locations. In this version, the diversion is always performed from the location where the stats counters are installed.
2. The MitigationDriverLocal selects an AMS out of all feasible options (in the first release, the selection is trivial—it is the first in list).
3. Optionally configures all the AMSs (each diversion source may have a different AMS associated with it) prior to instructing to divert traffic to each. This is done through the plugged in AMSRep.
4. MitigationDriverLocal instructs the DvsnRep to divert traffic from each source NetNode (in this version, NetNode is modeled over an SDN switch) to the AMS associated with that NetNode. Diversion can be either for inbound traffic only or both for inbound and outbound traffic.
5. Mitigation driver notifies the AMSBasedDetector to optionally start monitoring the attack status in all the AMSs, and feed attack detections to the AttackDecisionPoint.
6. In future versions, the MitigationDriverLocal is scheduled to monitor health of all AMSs and relevant portions of network topologies, re-selecting AMSs should some fail, or should network topologies changes require that. When mitigation should be ended, the MitigationDriverLocal notifies AMSBasedDetector to stop monitoring the attack status for the ended attack, notifies DvsnRep to stop traffic diversions to all AMSs for this mitigation, and finally notifies the AMSRep to optionally clean all mitigation-related configuration sets in each relevant AMS.

AMSBasedDetector – This optional module (which can be packaged as part of the AMSRep) is responsible for monitoring/querying attack mitigation by AMSs. Registering as a detector, this module can then notify AttackDecisionPoint about attack continuations and endings. It monitors only specified AMSs and only for specified (attacked) traffic.

AMSRep - A pluggable module for different AMSs. The module adheres to AMSRep APIs. It can support configuration of all introduced AMSs (permanently or before/after attack mitigations). It can also receive/query security information (attack statuses), as well as operational information (health, load). AMSRep module is entirely optional – AMSs can be configured and monitored externally. In many cases, attacks can continue be monitored solely via SDN counters. Defense4All contains a reference implementation AMSRep that communicates with Radware's DefensePro AMSs.

ODL Reps View

The following figure illustrates the Defense4All application ODL Reps module-set structure. image::D4a_odl_reps_view.jpg[ODL Reps View]

Different versions of OFC may be represented by different versions of the ODL Reps module-set. ODLReps comprises two functions: stats collection for, and traffic diversion of, relevant traffic. Both or either of the functions may be utilized in a given deployment. As such, they have a common point to communicate with the ODC and hold all general information for the ODC.

ODL Reps supports two types of SDN switches: `sdn-hybrid`, which supports both SDN and legacy routing, and `sdn-native`, which supports SDN only routing. Counting traffic on the `sdn-hybrid` switch is done by programming a flow entry with the desired traffic selection criteria and the action “send to normal”, that is, to continue with legacy routing. Counting traffic on `sdn-native` switch requires an explicit routing action (which output port to send the traffic to). Defense4All avoids learning all routing tables by requiring an `sdn-native` switch which is more or less a bump-in the wire with respect to traffic routing (that is, traffic entering port 1 normally exits port 2 and traffic entering port 3 normally exits port 4 and vice versa). Such a switch allows for easy programming of flow entries just to count traffic or to divert traffic to/from the attached AMS. When Defense4All programs a traffic counting flow entry with selection criteria that includes port 1, its action is output to port 2, and similarly with 3 to 4. In future versions, this restriction is scheduled to be lifted.

The following is a description of the sub-modules:

StatsCollectionRep - The module adheres to StatsCollectionRep APIs. Its main tasks are:

- Offer counter placement NetNodes in the network. The NetNodes offered are all NetNodes defined for a PN. This essentially maps which of SDN switches the traffic of the given PN flows.
- Add a peacetime counter in selected NetNodes to collect statistics for a given PN. StatsCollectionRep creates a single counter for a PN in each NetNode. (Overall, a NetNode can have multiple counters for different PNs; and a PN can have multiple counters in NetNodes as specified for the given PN). StatsCollectionRep translates the installation of a counter in a NetNode to programming four flow entries (for TCP, UDP, ICMP, and the rest of the IPs) for each “north traffic port” in that NetNode port from which traffic from a client to a protected PN enters the SDN switch. For example, StatsCollectionRep adds for a given PN 12 flow entries in an SDN switch with three ports through that PN’s inbound traffic enters the OFS. And, if another NetNode (SDN switch) was specified to have that PN’s inbound traffic entering it through two ports, then StatsCollectionRep programs for this PN eight flow entries in that second NetNode.
- Remove a peacetime counter.
- Read latest counter values for a specified counter. StatsCollectionRep returns a vector of latest bytes and packets counted for each protocol-port in each direction (currently only “north to south” is supported), along with the time it received the reading from the ODC.

DvsnRep - The module adheres to DvsnRep APIs. Its main tasks are:

- Return diversion properties from a given NetNode to a given AMS. In this version, an empty property is returned if such a diversion is topologically feasible (AMS is directly attached to the SDN switch over which the specified NetNode is modeled. Otherwise no properties are returned. This leaves room for remote diversions in future versions, and topological costs to each distant AMS, such as latency, bandwidth reservation, and cost).
- Divert (attacked) traffic from a specified NetNode through an AMS. As such, the new flow entries take precedence over the peacetime ones. DvsnRep programs flow entries to divert inbound attacked traffic (or all traffic, if so specified for the PN) from every “north traffic port” into the AMS “north” port. If “symmetric diversion” (for both inbound

and returning, outbound traffic) has been specified for that PN, DvsnRep programs another set of flow entries to divert attacked (or all) traffic from every “south traffic port” into the AMS “south” port. In an sdn-hybrid switch deployment, DvsnRep adds a flow entry for inbound traffic that returns from the AMS south port, with the action sent to normal, and similarly it adds a flow entry for outbound returning traffic from the AMS north port, with action of also sent to normal. In an SDN-native switch, the action is to send to the correct output port, however if this scenario the process is more complex for determining the correct port. North port MAC learning is used to determine from the source/destination MAC in the packet the correct output port. This scheme of flow entries works well for TCP, UDP and ICMP attacks. For “other IP” attacks, the flow entries programming is more complex, and is suppressed here for clarity. The set of flow entries programmed to divert (but still count) traffic comprises the “attack traffic floor”. There may be many attack traffic floors, all of which take precedence over the peacetime stats collection floor (by programming higher priority flow entries). Additional attacks (except “other IP” attacks, which is a special case, and is suppressed here) are created with higher priority traffic floors over previously set attack traffic floors. Attacks may fully or partially “eclipse” earlier attacks (for example, TCP port 80 over TCP, or vice versa), or be disjointed (such as TCP and UDP). Stats collection is taken from all traffic floors, both peacetime and attacks. An SDN-based detector aggregates all statistics into overall rates, thus determining if the attack is still in progress. (Note that eclipsed peacetime counted traffic may show zero rates, and that counting is complemented by the higher priority floor counters.)

- End diversion. DvsnRep removes the relevant attack traffic floor (removing all of its flow entries from the NetNode). Note that this affects neither traffic floors “above” the removed floor nor the traffic floors “below.” In addition, the SDN-based detector receives the same aggregated rates from counters of remaining floors, so its operation also is not affected.

ODLCommon – This module contains all common elements needed to program flow entries in the ODC. This allows for coherent programming of configured ODCs (in this version, at most one) by StatsCollectionRep and DvsnRep. For instance ODLCommon instantiates connectivity with the ODCs, maintains a list of programmed flow entries and cookies assigned to each. It also maintains references to DFAppRoot and FrameworkMain. When an sdn-native NetNode is added ODLCommon programs 2 flow entries per each protected link (pair of input-to-output ports) to transfer traffic between the two ports (traffic entering north port is routed to south port and vice versa). ODLCommon adds two more flow entries for each port connecting to an AMS to block returning ARP traffic (so as to avoid ARP floods if the AMSs are not configured to block them). This “common traffic floor” flow entries are set with the lowest priority. Their counters are accounted for neither stats collections nor traffic diversion. When a NetNode is removed, ODLCommon removes this common traffic floor flow entries.

FlowEntryMgr – This module provides an API to perform actions on flow entries in an SDN switch managed by an ODC, and retrieves information about all nodes managed by an ODC. Flow entries actions include adding a specified flow entry in a specified NetNode (SDN switch/router), removing a flow entry, toggling a flow entry, getting details of a flow entry, and reading statistics gathered by the flow entry. FlowEntryMgr uses the connector modules to communicate with the ODC.

Connector – This module provides the basic API calls to communicate with the ODC, wrapping REST communications. After initializing connection details with a specified ODC,

the connector allows getting or deleting data from the ODC, as well as posting or putting data to the ODC.

ODL REST Pojos – This set of Java classes are part of the ODC REST API, specifying the Java classes of the parameters and the results of interaction with the ODC.

Basic Control Flow

Control flows are logically ordered according to module runtime dependencies, so if module A depends on module B then module B should be initialized before module A, and terminate after it. Defense4All application modules depend on most Framework modules, except WebServer.

Startup — Defense4All initializes all its modules and re-applies previously configured infrastructure and security set-ups, obtaining them from persistent repositories. At the end of the Startup process, Defense4All resumes its prior operation. **Termination** - restart — Defense4All persists any relevant data into stable storage repositories, and terminates itself. If the termination is for restart, the automatic restart mechanism restarts Defense4All. Otherwise (such as upgrading) Defense4All does not automatically restart. **Reset** — In this flow, all modules are reset to factory level. This means that all dynamically obtained data as well as user configurations are deleted

Configurations and Setup Flow

OFC (OpenFlowController = ODC) – When DFMgmtPoint receives from DFRestService a request to add an OFC, it first records the added OFC in the OFC's Repo, and then notifies ODLStatsCollectionRep and ODLDvsnRep, which in turn notify the ODL to initiate a connection to the added OFC (ODC). ODL instantiates a REST client for communication with the ODC.

NetNode - Multiple NetNodes can be added. Each NetNode models a switch or similar network device, along with its traffic ports, protected links, and connections to AMSs. When DFMgmtPoint receives from DFRestService a request to add a NetNode, it first records the added NetNode in NetNodes Repo, and then notifies ODLStatsCollectionRep and ODLDvsnRep, followed by MitigationMgr. ODLStatsCollectionRep and ODLDvsnRep then notify the ODL, and the ODL installs low priority flow entries to pass traffic between the protected links' port pairs. MitigationMgr notifies MitigationDriverLocal, which updates its NetNode-AMS connectivity groups for consistent assignment of AMSs to diversion from given NetNodes.

AMS – Multiple AMSs can be added. When DFMgmtPoint receives from DFRestService a request to add an AMS, it first records the added AMS in the AMS's Repo, and then notifies AMSRep. AMSRep can optionally pre-configure protection capabilities in the added AMS, and start monitoring its health.

PN - Multiple PNs can be added. When DFMgmtPoint receives from DFRestService a request to add a PN, it first records the added PN in the PN's Repo, notifies MitigationMgr, and then finally notifies the DetectionMgr. MitigationMgr notifies MitigationDriverLocal, which then notifies AMSRep. AMSRep can preconfigure the AMS for this PN, as well its EventMgr to accept events related to this PN's traffic. DetectionMgr notifies RateBasedDetector, which then notifies StatsCollector. StatsCollector queries ODLStatsCollectionRep about

possible placement of stats collection counters for this PN. `ODLStatsCollectionRep` returns all `NetNodes` configured for this PN (and if none are configured, it returns all `NetNodes` currently known to `Defense4All`). `StatsCollector` “chooses” the counter locations option (the only available option in this version). For each of the `NetNodes`, it then asks `ODLStatsCollectionRep` to create a counter for the subject PN. The counter is essentially a set of flow entries set for the protocols of interest (TCP, UDP, ICMP, and the rest of the IPs) on each north traffic port. The counter is given a priority and this constitutes the peacetime traffic floor (to monitor traffic by periodically reading all counter flow entry traffic count values). Because the PN may be re-introduced at restart or a change in network topology may require re-calculation of counter locations, it is possible that some/all counters may already be in place. Only new counters are added. Counters that are no longer are removed. `ODLStatsCollectionRep` configures the flow entries according to the `NetNode` type. For hybrid `NetNodes`, the flow entry action is “send to normal” (proceed to legacy routing), while for native `NetNodes`, the action is to match the output port (in each protected link). `ODLStatsCollectionRep` invokes the ODL to create each specified flow entry. The latter invokes `FlowEntryMgr` and `Connector` to send the request to the ODC.

Attack Detection Flow

Periodically, the `StatsCollector` requests the ODL `StatsCollectionRep` to query the ODC for the latest statistics for each set counter for each configured PN. `ODLStatsCollectionRep` invokes `FlowEntryMgr` to obtain statistics for each flow entry in a counter. The latter invokes the connector to obtain the desired statistics from the ODC.

`ODLStatsCollectionRep` aggregates the obtained results in a vector of stats (latest bytes and packets readings per each protocol) and returns that vector. `StatsCollector` feeds each counter stats vector to `DetectionMgr`, which then forwards the stats vector to the `RateBasedDetector`. The `RateBasedDetector` maintains stats information for every counter as well as aggregated counter stats for every PN. Stats information includes the time of previous reading, and for every protocol the latest rates and exponential averages.

The `RateBasedDetector` checks for significant and prolonged latest rate deviations from the average, and if such deviations are found in the PN aggregated level, it notifies the `AttackDecisionPoint` about attack detection. As long as deviations continue, the `RateBasedDetector` continues notifying the `AttackDecisionPoint` about the detections. It sets an expiration time for every detection notification, and repeatable notifications essentially prolong the detection expiration.

`AttackDecisionPoint` honors all detections. If it has already declared an attack on that protocol-port, then the `AttackDecisionPoint` associates the additional detection with that existing attack. Otherwise, it creates a new attack and notifies the `MitigationMgr` to mitigate that attack (as described below). Periodically, `AttackDecisionPoint` checks the status of all detections of each live attack. If all detections have expired, `AttackDecisionPoint` declares the end of the attack and notifies `MitigationMgr` to stop mitigating the attack.

Attack Mitigation Flow

`MitigationMgr`, upon receiving mitigate notification from the `AttackDecisionPoint`, attempts to find a plugged-in `MitigationDriver` to handle the mitigation. Currently, it requests only its plugged-in `MitigationDriverLocal`.

MitigationDriverLocal checks if there are known, live, and available AMSs to which attacked (or all) traffic can be diverted from NetNodes through which attacked traffic flows. It selects one of the suitable AMSs and configures it prior to diverting attack traffic to the selected AMS. For example, MitigationDriverLocal retrieves from Repo the relevant protocol averages, and configures them in AMS through the AMSRep.

MitigationDriverLocal then requests ODLDvsnRep to divert the attacked PN protocol-port (or all PN) traffic from each of the NetNodes through which the PN traffic flows to the selected AMS.

ODLDvsnRep creates a new highest priority traffic-floor (that contains flow entries with a priority higher than any flow entry in the previously set traffic floors). The traffic floor contains all flow entries to divert and count traffic from every ingress/northbound traffic port into the AMS, and back from the AMS to the relevant output (southbound) ports. Optionally, diversion can be "symmetric" (in both directions), in which case flow entries are added to divert traffic from southbound ports into the AMS, and back from the AMS to northbound ports. Note that the StatsCollector treats this added traffic floor as any other, and passes obtained statistics from this floor to the DetectionMgr/RateBasedDetector. Because traffic floors are aggregated (in the same NetNode as well as across NetNodes) for a given PN the combined rates remain the same as prior to diversion. Just like ODLStatsCollectionRep, ODLDvsnRep also utilizes lower level modules to install the flow entries in desired NetNodes.

Finally, MitigationDriverLocal notifies AMSRep to optionally start monitoring this attack and notify the AttackDecisionPoint if the attack continues or new attacks develop. AMSRep can do that through the AMSBasedDetector module.

If MitigationDriverLocal finds no suitable AMSs, or fails to configure any of its mitigation steps, it aborts the mitigation attempt, asynchronously notifying MitigationMgr. The mitigation then remains in status "no-resources."

When MitigationMgr receives a notification to stop mitigating an attack, it forwards this notification to the relevant (and currently the only) MitigationDriver, MitigationDriverLocal. MitigationDriverLocal reverses the actions in at the start of the mitigation. It notifies AMSRep to stop monitoring for this attack, it cancels diversion for the attacked traffic, and finally notifies AMSRep to optionally remove pre-mitigation configurations.

Continuity

Service Continuity, as opposed to High Availability, is defined here as the ability to deliver a required level of service, at tolerable cost, in the presence of disrupting events, where:

- Disrupting events can be load, changes, logical errors, failures and disasters, administrative actions (such as an upgrade), external attacks, and so on.
- The level of service can include response time, throughput, survivability of data/operations, security/privacy, and so on. The required level of service may differ for every service function, for every type of event, at different event handling phases.
- The cost can include people (number, expertise), equipment (hardware, software), facilities (space, power).

Clustering and Fault-tolerance - Clusters help to address both Scalability and High Availability. If one of the cluster members fails, another cluster member can quickly assume its responsibilities. This overcomes member failures, member hosting machine failures, and member network connectivity failures. Defense4All clustering is scheduled for future releases. In version 1.0, Defense4All runs as a Linux restartable service, so if it fails, the hosting Linux OS revives Defense4All. This enables overcoming intermittent/sporadic Defense4All failures. Failure of the Defense4All hosting machine means longer time and modest additional human effort to revive the machine and its hosted Defense4All. If the machine cannot be brought up, Defense4All can be started on another machine in the network. To ensure that Defense4All resumes its operation (rather than restart from scratch) you must pre-load the Defense4All (latest or earlier) state snapshot on that machine. A non-clustered environment affects the time and the human effort to recover from machine failures. The time factor is less critical, as Defense4All runs out-of-path, so its longer non-availability period means a longer time to detect and mitigate new attacks.

State Persistence - Defense4All persists the state in the Cassandra DB running on the same machine. In version 1.0, only one Cassandra instance cluster is configured. As long as local stable storage does not crash, a Linux restart of the Defense4All service enables Defense4All to quickly retrieve its latest state from Cassandra and resume its latest operation. The same happens at failure and restart of the machine hosting Defense4All. Taking the Defense4All state backup, and restoring on another machine allows for resuming the Defense4All operation on that machine. Multi-node Cassandra clusters (scheduled for future versions) will increase state persistence while reducing recovery time and effort.

Restart Process - When Defense4All (re)starts, it first checks for saved configuration data, and re-plays the configuration steps against all its relevant modules, driving any relevant external programming and/or configuration actions (such as against the PFC or AMS devices), for example, re-adding a PO. The only difference between this configuration replay and original configuration is that any dynamically obtained data is preserved, for example, all PO statistics. This allows for easily reaching internal consistency, especially in cases where Defense4All or its hosting machine has crashed. When configuration action derivatives are replayed against external entities, for example adding missing PO stats counters, and removing no longer necessary ones, consistency with external entities is also reached. Defense4All becomes operational (launching its Web server), lets you or some other component to complete Defense4All missing configurations according to possible changes while Defense4All was down. This results in reaching end-to-end consistency.

Reset - Defense4All lets you reset its dynamically obtained data and configuration information (factory reset). This enables you to overcome many logical errors and mis-configurations. Note that a Defense4All restart or failover would not overcome such problems. This mechanism is therefore complementary to the restart-failover mechanism, and should typically be applied as a last resort.

Failure Isolation and Health Tracker - In Defense4All, failure isolation takes place in the form of a failure of immediate recovery or compensation (as much as possible), and a failure recording in a special module called Health Tracker. Except for a handful of substantial failures (such as a failure to start the Framework), no failure in any module immediately causes Defense4All to stop. Instead, each module records each failure in its scope, providing severity specifications and an indication of failure permanence. If the combined severity (permanent or temporary) of all failures exceeds a globally set threshold, the HealthTracker triggers Defense4All shutdown (and revival by Linux).

Later on, permanent or repeating temporary faults will cause HealthTracker to trigger Defense4All soft and dynamic reset (of dynamically obtained data) or suggest to the administrator to perform a factory reset (that also includes configuration information).

State Backup and Restore - The administrator can snapshot the Defense4All state, save the backup in a different location, and restore to the original or new Defense4All location. This allows overcoming certain logical bugs and mis-configurations, as well as the permanent failure of the machine hosting Defense4All. To snapshot the Defense4All state, do the following:

1. Quiesce (shutdown) Defense4All, causing the current state to flush to stable storage). Avoid performing any configurations changes when it is brought back up, avoiding new state changes.
2. Take the Cassandra snapshot for Defense4All DB - "DF": For backup-restore guidelines, refer to http://www.datastax.com/docs/1.0/operations/backup_restore.
3. Copy the snapshot files to the desired storage archive.

To restore a Defense4All backup to a target machine, do the following:

1. Restore the desired saved snapshot in the target machine (same as backup or different). For Cassandra backup-restore guidelines, refer to http://www.datastax.com/docs/1.0/operations/backup_restore.
2. Bring up Cassandra on that machine.
3. Bring up Defense4All on that machine.

7. DLUX

Table of Contents

| | |
|---------------------|-----|
| Setup and Run | 146 |
| DLUX Modules | 148 |
| Yang Utils | 151 |

Setup and Run

Required Technology Stack

- NodeJS (Http Server, <http://www.nodejs.org>)
- Bower (JavaScript Package Manager, <http://bower.io>)
- GruntJS (JavaScript Task Runner, <http://gruntjs.com>)
- AngularJS (JavaScript client-side framework, <http://www.angularjs.org>)
- Karma (JavaScript Test Runner, <http://karma-runner.github.io/>)
- Other AngularJS/Third-party JS libraries

Install NodeJS

For Windows and Mac without brew:

1. Go to <http://www.nodejs.org>
2. Download and install NodeJS

For Mac with brew installed:

```
$ brew update
$ brew install node
```

Verify NodeJS is installed:

```
$ npm --version
```

Install required Node libraries

Install the following node components using npm. For Mac, you may have to use "sudo"

```
$ sudo npm -g install grunt-cli
$ sudo npm -g install bower
$ sudo npm -g install karma
$ sudo npm -g install karma-cli
```

Get latest DLUX code from git

Anonymous clone.

```
$ git clone http://git.opendaylight.org/gerrit/p/dlux.git
```

If you have a.opendaylight.org account.

```
$ git clone ssh://<username>@git.opendaylight.org:29418/dlux.git
```

Build the DLUX code

```
$ cd dlux/dlux-web

#installs the necessary NodeJS related components for the project - will
create a node_modules directory
$ sudo npm install

#installs necessary components provided by bower
$ bower install

# update dlux-web with all DLUX static resources for each module by running
maven
$ mvn clean install

# run the unit tests and start karma test runner
# this will open up the default browser pointing to karma test running and
run the unit tests
$ grunt watch
```

Hit Ctrl-C to quit

Build DLUX Karaf feature and distribution

Once you have installed all necessary modules mentioned above such as nodesjs, bower etc.. You should be able to build the DLUX feature and distribution. Run following command at DLUX home directory /dlux. Once successful, It will create DLUX Karaf distribution and DLUX Karaf feature. You can find Karaf distribution at dlux/distribution-dlux.

```
$ mvn clean install
```

Enable DLUX Karaf Feature

Get the Opendaylight official helium distribution or use Karaf distribution of DLUX project created above. Unzip the Karaf distribution and go to bin directory of your distribution and run following command to start Karaf. It will start the Karaf console, which may not have any feature installed by default.

```
$ ./karaf
```

Install basic MD-SAL controller features on the Karaf console. I would recommend installing the following features before starting the DLUX feature. L2Switch feature internally enables MD-SAL data broker and openflow plugin service. L2Switch also makes sure that in topology UI, hosts are also visible along with switches. We need the mdsal-apidocs feature for yangUI in DLUX.

```
$ feature:install odl-restconf
$ feature:install odl-l2switch-switch
$ feature:install odl-mdsal-apidocs
```

Install the AD-SAL features on the Karaf console.

```
$ feature:install odl-adsal-all
$ feature:install odl-adsal-northbound
```

Then, install the DLUX feature

```
$ feature:install odl-dlux-core
```

Once done, you should be able to access DLUX UI at

```
http://<IP of your controller machine>:8181/dlux/index.html
```

For login, use admin as both the username and the password. Before login, just make sure that MD-SAL features are enabled. you can check what all features are installed by running following command -

```
$ feature:list -i
```

Run standalone DLUX against the controller

- Start Karaf distribution with installed MD-SAL and AD-SAL features.
- Goto mininet VM and create a topology for the controller

Based on where you controller is running, Update baseUrl in file dlux/dlux-web/config/development.json.

Back in the DLUX terminal, run

```
$ grunt live
```

Open a browser and go to the URL

```
http://localhost:9000/dlux/index.html
```

This should bring up the DLUX UI and pull data from the controller. Use admin as the username and password to access the UI.

DLUX Modules

DLUX modules are the individual features such as nodes, topology etc. Each module has a defined structure and you can find all existing modules under /dlux/modules directory of code.

Module Structure

- module_folder
 - <module_name>.module.js
 - <module_name>.controller.js
 - <module_name>.services.js

- <module_name>.directives.js
- <module_name>.filter.js
- index.tpl.html
- <a_stylesheet>.css

Create New Module

Define the module

First, create an empty file with the module name. Next, we need to surround our module with a define function. This allows RequireJs to see our module.js files. The first argument is an array who contain all the module dependencies. The second is a callback function whose body contain the AngularJs code base. The function parameters correspond with the order of dependencies. Each dependencies is injected into a parameter if it is provided. Finally, we return the angular module to be able to inject it as a parameter in our others modules.

For each new module, you must have at least those two dependencies :

- angularAMD : It's a wrapper arround angularjs to provide an AMD (Asynchronous Module Definition) support. Which is used by RequireJs. For more information click [here](#).
- app/core/core.services : This one is mandatory if you want to add content in the navigation menu, the left bar or the top bar.

The following are not mandatory, but very often used.

- angular-ui-router : A library to provide URL routing
- routingConfig : To set the level access to a page

```
define(['angularAMD', 'app/routingConfig', 'angular-ui-router', 'app/core/
core.services'], function(ng) {
  var module = angular.module('app.a_module', ['ui.router.state',
  'app.core']);
  // module configuration
  module.config(function() {
    [...]
  });
  return module;
});
```

Set the register function

If your module is only required by the main application, you will need register your angular components because the app will be already bootstrapped. Otherwise, it won't see your components on the runtime.



Tip

If your module is only use by an other module, you don't have to do this step.

```
module.config(function($compileProvider, $controllerProvider, $provide) {
  module.register = {
    controller : $controllerProvider.register,
    directive : $compileProvider.directive,
    factory : $provide.factory,
    service : $provide.service
  };
});
```

Set the route

The next step is to set up the route for our module. This part is also done in the configuration method of the module. We have to add **\$stateProvider** as a parameter.

```
module.config(function($stateProvider) {
  var access = routingConfig.accessLevels;
  $stateProvider.state('main.module', {
    url: 'module',
    views : {
      'content' : {
        templateUrl: 'src/app/module/module.tpl.html',
        controller: 'ModuleCtrl'
      }
    }
  });
});
```

Adding element to the navigation menu

To be able to add item to the navigation menu, the module requires the **NavHelperProvider** parameter in the configuration method. This helper has a method to easily add an item to the menu. The first parameter is an id that refers to the level of your menu and the second is a object.

```
var module = angular.module('app.a_module', ['app.core']);
module.config(function(NavMenuHelper) {
  NavMenuHelper.addToMenu('myFirstModule', {
    "link" : "#/module/index",
    "active" : "module",
    "title" : "My First Module",
    "icon" : "icon-sitemap",
    "page" : {
      "title" : "My First Module",
      "description" : "My first module"
    }
  });
});
```

The ID parameter supports, for now, two levels of depth. So if your ID looks like *rootNode.childNode*, the helper will look for a node named *rootNode* and it will append the *childNode* to it. If the root node doesn't exist, it will create it.

Link the controller file

To include the controller file, we will use the NavHelperProvider. It contain a method who will load the given file.

```
[...]
```

```
NavHelperProvider.addControllerUrl('<path_to_module_folder>/  
<module_name>.controller');
```

The module.js file is now complete.

Create the Controllers, factory, directive, etc

Creating the controller and other components are similar to the module.

- First, add the define method
- Second, add the relative path to the module definition
- Last, create your methods as you usually do it with angularJs

```
define(['<relative_path_to_module>/<module_name>.module'], function(module)  
{  
  module.register.controller('ModuleCtrl', function($rootScope, $scope) {  
  });  
});
```

Append to the main file

The last thing to do is to add the path of the module definition file and add the name of the angular module. So, edit the file app.module.js as the follows.

```
//----Temporary-----\\  
var module = [  
  [...]  
  '<relative_path_module>/<module_name>.js',  
  [...]  
var e = [  
  [...]  
  'a_module',  
  [...]  
//-----\\
```

Yang Utils

Yang Utils are used by yang UI to perform all CRUD operations. All of these utilities are present in yangutils.services.js file. It has following factories -

Factories

- **arrayUtils** – defines functions for working with arrays.
- **pathUtils** – defines functions for working with xpath (paths to APIs and subAPIs). It divides xpath string to array of elements, so this array can be later used for search functions.
- **syncFact** – provides synchronization between requests to and from ODL when it's needed.
- **custFunc** – it is linked with apiConnector.createCustomFunctionalityApis in yangui controller in yangui.controller.js. That function makes it possible to create some custom

function called by the click on button in index.tpl.html. All custom functions are stored in array and linked to specific subAPI. When particular subAPI is expanded and clicked, its inputs (linked root node with its child nodes) are displayed in the bottom part of the page and its buttons with custom functionality are displayed also.

- **reqBuilder** – creates object builder = request built from filled inputs on page in JSON format. It is possible with “show preview” button. This request is sent to ODL when button PUT or POST is clicked.
- **yinParser** – factory for reading of .xml files of yang models and creating objects hierarchy. Every statement from yang is represented by node.
- **nodeWrapper** – adds functions to objects in tree hierarchy created with yinParser. These functions provide functionality for every type of node.
- **apiConnector** – the main functionality is filling the main structures and linking them. Structure of APIs and subAPIs which is two level array - first level is filled by main APIs, second level is filled by others sub APIs. Second main structure is array of root nodes, which are objects including root node and its children nodes. Linking these two structures is creating links between every subAPI (second level of APIs array) and its root node, which must be displayed like inputs when subAPI is expanded.
- **yangUtils** – some top level functions which are used by yangui controller for creating the main structures.

8. Group-Based Policy

Table of Contents

| | |
|--|-----|
| Group-Based Policy Architecture Overview | 154 |
| Policy Model | 155 |
| State Repositories | 169 |
| Renderers | 170 |

This chapter describes the Group-Based Policy project. The Group-Based Policy project defines an application-centric policy model for OpenDaylight that separates information about application connectivity requirements from information about the underlying details of the network infrastructure.

Group-Based Policy Architecture Overview

Figure 8.1. Group-Based Policy Architecture



State repositories (blue) communicate using MD-SAL (orange) with external orchestration systems as well as internally with renderers (green) through the renderer common framework (red).

The components of the architecture are divided into two main categories. First, there are components that are responsible for managing the policy, configuration, and related state. These are the components that deal with the higher-order group-based policy that exists independent of the underlying infrastructure. Second, the renderer components that are responsible for applying the policy model to the underlying network infrastructure. The

system can support potentially a variety of renderers that may have very different sets of features and different approaches for enabling the policy that the user has requested.

The key to understanding the architecture is to first understand the policy model — much of the design of the system flows directly from the design of the policy model.

Policy Model

The policy model is built around the idea of placing endpoints into groups that share the same semantics, and then defining what other groups those endpoints need to communicate, and then finally defining how these endpoints need to communicate. In this way, we represent the requirements of the application and then force the infrastructure to figure out how to meet these requirements, rather than defining the policy in terms of the underlying infrastructure.

Policy Model UML Diagrams



Note

The UML diagrams included here are **not** normative. They attempt to provide a pictorial representation of the group-based policy model. In the event of any conflicts, the RESTful interfaces that are generated dynamically from the group-based policy yang models are normative. The group-based policy yang models can be found at `./groupbasedpolicy/groupbasedpolicy/src/main/yang/model/`.

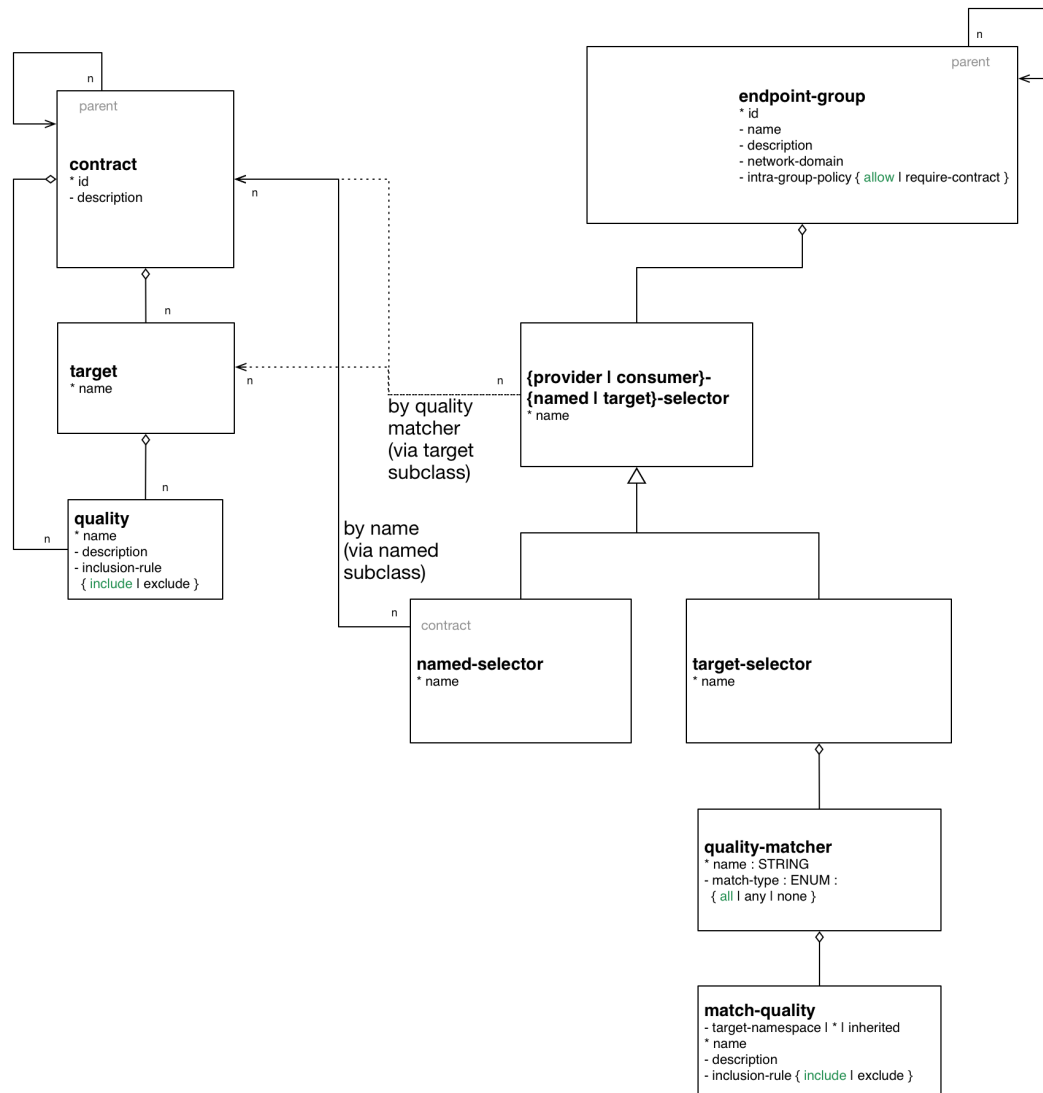
Figure 8.2. Policy Model: Contract Selection

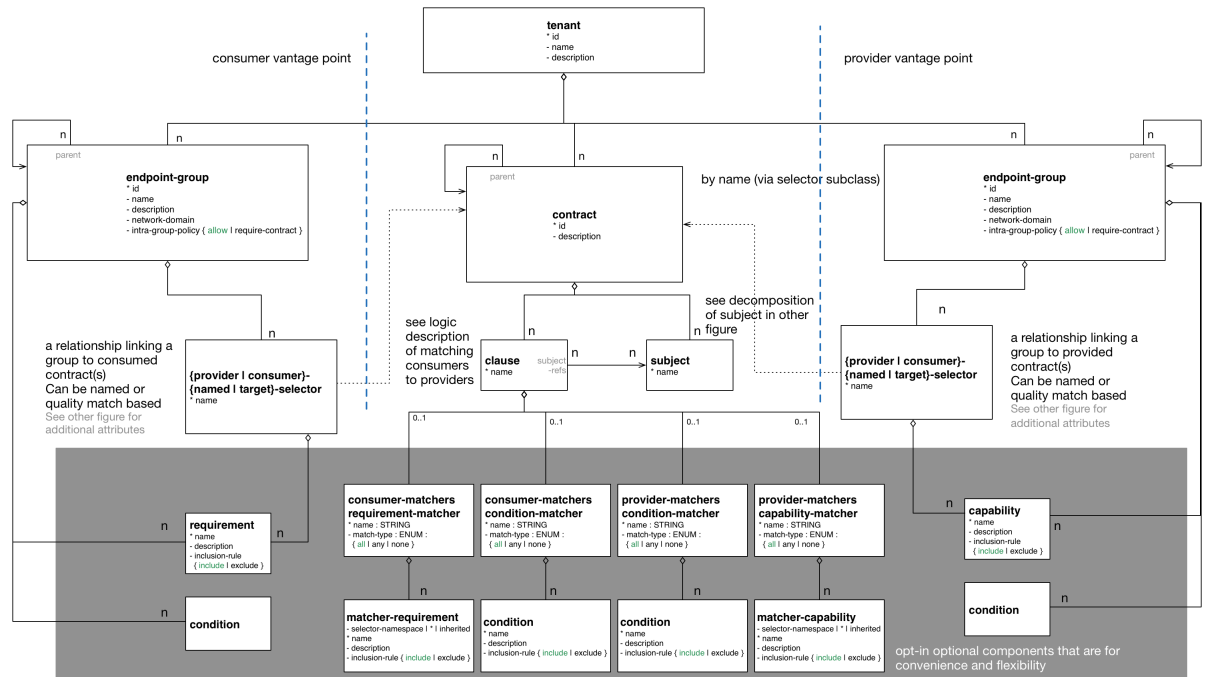
Figure 8.3. Policy Model: Clauses and Subject Selection

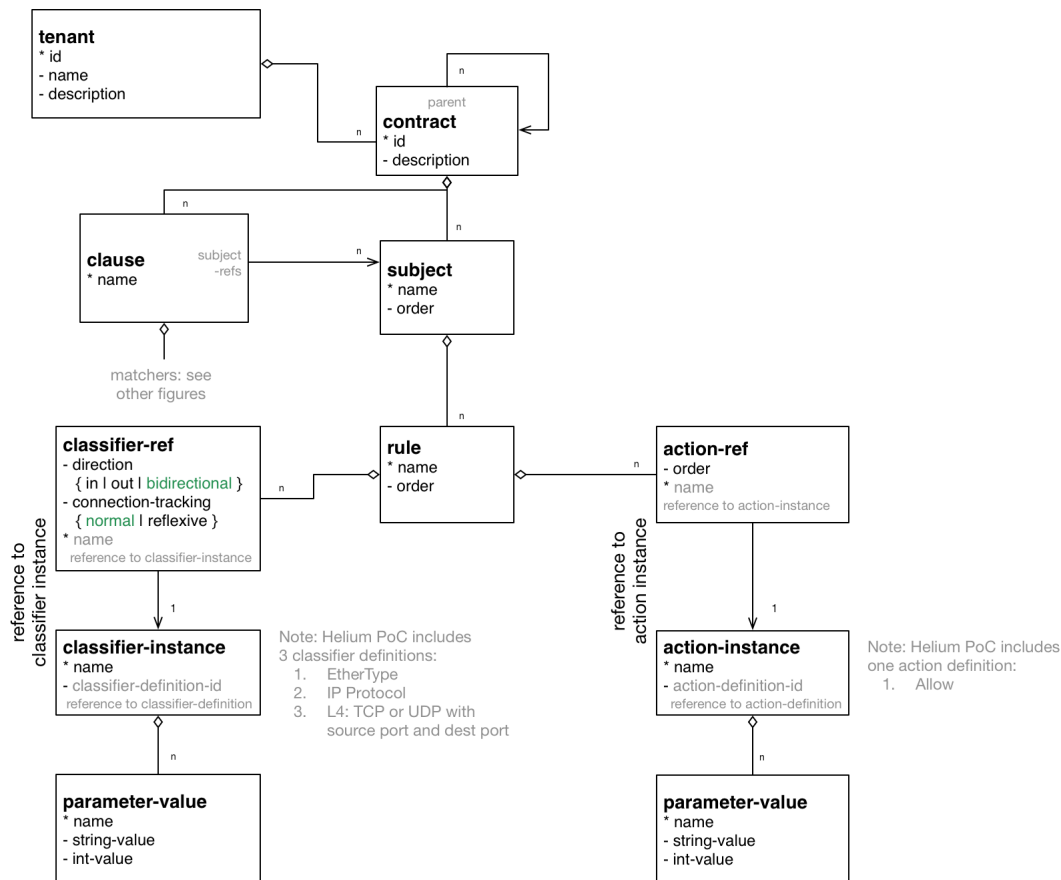
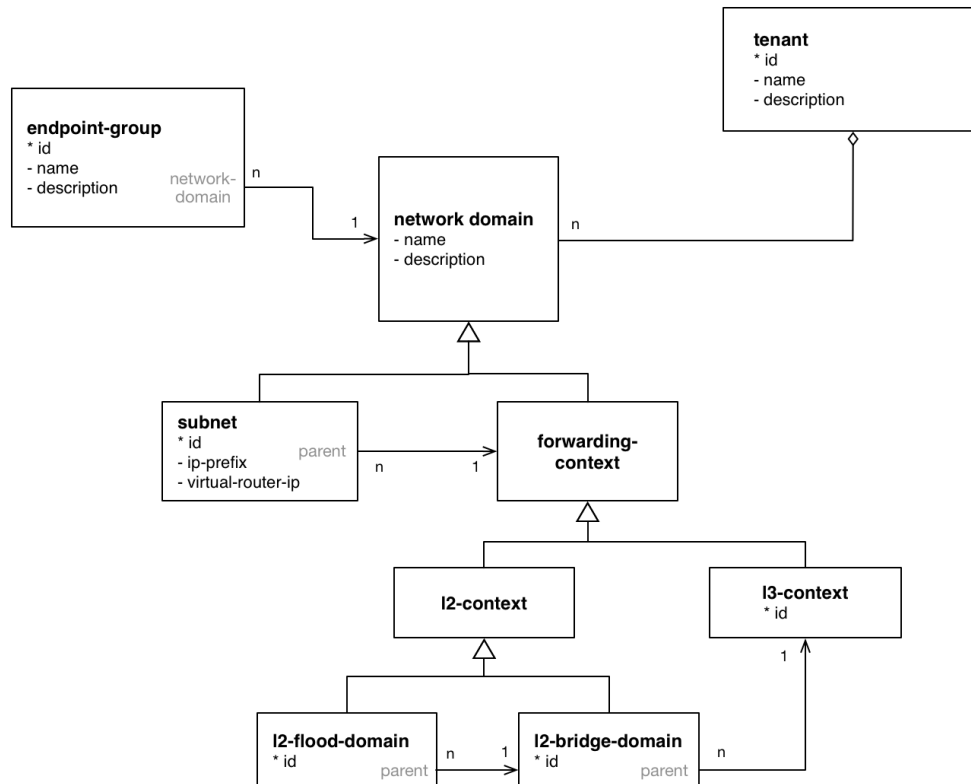
Figure 8.4. Policy Model: Subject Contents

Figure 8.5. Policy Model: Forwarding

Policy Concepts

This section describes some of the most important concepts in the policy model. See the next section on [Policy Resolution](#) for a description of how these fit together to determine how to apply the policy to the network.

| | |
|----------------|---|
| Endpoint | An <i>endpoint</i> is a specific device in the network. It could be a VM interface, a physical interface, or other network device. Endpoints are defined and assigned to endpoint groups through mechanisms that are not specified by the policy model (See Endpoint Repository for more information). Endpoints can have associated <i>conditions</i> that are just labels that represent some potentially-transient status information about an endpoint. |
| Endpoint Group | <i>Endpoint groups</i> are sets of endpoints that share a common set of policies. Endpoint groups can participate in <i>contracts</i> that determine the kinds of communication that is allowed. They also expose both <i>requirements</i> and <i>capabilities</i> , which are labels that help to determine how contracts will be applied. An endpoint group is allowed to specify a parent endpoint group from which it inherits. |
| Contract | <i>Contracts</i> determine which endpoints can communicate and in what way. Contracts between pairs of endpoint groups are selected by the contract selectors defined by the endpoint group. Contracts |

expose *qualities*, which are labels that can help endpoint groups to select contracts. Once the contract is selected, contracts have *clauses* that can match against requirements and capabilities exposed by endpoint groups, as well as any conditions that may be set on endpoints, in order to activate *subjects* that can allow specific kinds of communication. A contract is allowed to specify a parent contract from which it inherits.

| | |
|---------|---|
| Clause | <i>Clauses</i> are defined as part of a contract. Clauses determine how a contract should be applied to particular endpoints and endpoint groups. Clauses can match against requirements and capabilities exposed by endpoint groups, as well as any conditions that may be set on endpoints. Matching clauses define some set of <i>subjects</i> which can be applied to the communication between the pairs of endpoints. |
| Subject | <i>Subjects</i> describe some aspect of how two endpoints are allowed to communicate. Subjects define an ordered list of rules that will match against the traffic and perform any necessary actions on that traffic. No communication is allowed unless a subject allows that communication. |

Introduction to Policy Resolution

There are a lot of concepts to unpack and it can be difficult to see how this all fits together. Let's imagine that we want to analyze a particular flow of traffic in the network and walk through the policy resolution process for that flow. The key here is that the policy resolution process happens logically in three phases. First, we need to select the contracts that are in scope for the endpoint groups of the endpoints of the flow. Next, we select the set of subjects that apply to the endpoints of the flow. Finally, we apply the rules from the applicable subjects to the actual network traffic in the flow.

Note that this description gives a semantic understanding of how the policy model should be applied. The steps described here may or may not correspond to an actual efficient implementation of this policy model.

Contract Selection

The first step in policy resolution is to select the contracts that are in scope. For a particular flow, we look up the endpoint groups for each of the endpoints involved in the flow.

Endpoint groups participate in contracts either as a *provider* or as a *consumer*. Each endpoint group can participate in many contracts at the same time, but for each contract it can be in only one role at a time. In addition, there are two ways for an endpoint group to select a contract: either with a *named selector* or with a *target selector*. Named selectors simply select a specific contract by its contract ID. Target selectors allow for additional flexibility by matching against *qualities* of the contract's *target*.

Thus, there are a total of 4 kinds of contract selector:

| | |
|-------------------------|--|
| provider named selector | Select a contract by contract ID, and participate as a provider. |
|-------------------------|--|

| | |
|--------------------------|--|
| provider target selector | Match against a contract's target with a quality matcher, and participate as a provider. |
| consumer named selector | Select a contract by contract ID, and participate as a consumer. |
| consumer target selector | Match against a contract's target with a quality matcher, and participate as a consumer. |

So to determine which contracts are in scope for our flow, we must find contracts where either the source endpoint group selects a contract as either a provider or consumer, while the destination endpoint group matches against the same contract in the corresponding role. So if endpoint *x* in endpoint group *X* is communicating with endpoint *y* in endpoint group *Y*, a contract *C* is in scope if either *X* selects *C* as a provider and *Y* selects *C* as a consumer, or *X* selects *C* as a consumer and *Y* selects *C* as a provider.

The details of how quality matchers work are described further below in [Matchers](#). For now, we can simply state that quality matchers provide a flexible mechanism for selecting the contract based on labels.

The end result of the contract selection phase can be thought of as a set of tuples representing selected contract scopes. The fields of the tuple are:

- Contract ID
- The provider endpoint group ID
- The name of the selector in the provider endpoint group that was used to select the contract, which we'll call the *matching provider selector*.
- The consumer endpoint group ID
- The name of the selector in the consumer endpoint group that was used to select the contract, which we'll call the *matching consumer selector*.

Subject Selection

The second phase in policy resolution is to determine which subjects are in scope. The subjects allow us to define what kinds of communication are allowed between endpoints in the endpoint groups. For each of the selected contract scopes from the contract selection phase, we'll need to apply the subject selection procedure.

Before we can discuss how the subjects are matched, we need to first examine what we match against to bring those subjects into scope. We match against labels called, capabilities, requirements and conditions. Endpoint groups have capabilities and requirements, while endpoints have conditions.

Requirements and Capabilities

When acting as a provider, endpoint groups expose *capabilities*, which are labels representing specific pieces of functionality that can be exposed to other endpoint groups that may meet functional requirements of those endpoint groups. When acting as a consumer, endpoint groups expose *requirements*, which are labels that represent that fact that the endpoint group requires some specific piece of functionality. As an example, we

might create a capability called "user-database" which indicates that an endpoint group contains endpoints that implement a database of users. We might create a requirement also called "user-database" to indicate an endpoint group contains endpoints that will need to communicate with the endpoints that expose this service. Note that in this example the requirement and capability have the same name, but the user need not follow this convention.

We examine the matching provider selector (that was used by the provider endpoint group to select the contract) to determine the capabilities exposed by the provider endpoint group for this contract scope. The provider selector will have a list of capabilities either directly included in the provider selector or inherited from a parent selector or parent endpoint group (See [Inheritance](#) below). Similarly, the matching consumer selector will expose a set of requirements.

Conditions

Endpoints can have *conditions*, which are labels representing some relevant piece of operational state related to the endpoint. An example of a condition might be "malware-detected," or "authentication-succeeded." We'll be able to use these conditions to affect how that particular endpoint can communicate. To continue with our example, the "malware-detected" condition might cause an endpoint's connectivity to be cut off, while "authentication-succeeded" might open up communication with services that require an endpoint to be first authenticated and then forward its authentication credentials.

Conditions do not actually appear in the policy configuration model other than as a named reference. To determine the set of conditions that apply to a particular endpoint, the endpoint will need to be looked up in the endpoint registry, and its associated condition labels retrieved from there.

Clauses

Clauses are what will do the actual selection of subjects. A clause has four lists of matchers in two categories. In order for a clause to become active, all four lists of matchers must match. A matching clause will select all the subjects referenced by the clause. Note that an empty list of matchers counts as a match.

The first category is the consumer matchers, which match against the consumer endpoint group and endpoints. The consumer matchers are:

| | |
|-----------------------------|--|
| Requirement matchers | matches against requirements in the matching consumer selector. |
| Consumer condition matchers | matches against conditions on endpoints in the consumer endpoint group |

The second category is the provider matchers, which match against the provider endpoint group and endpoints. The provider matchers are:

| | |
|-----------------------------|--|
| Capability matchers | matches against capability in the matching provider selector. |
| Provider condition matchers | matches against conditions on endpoints in the provider endpoint group |

Clauses have a list of subjects that apply when all the matchers in the clause match. The output of the subject selection phase logically is a set of subjects that are in scope for any particular pair of endpoints.

Rule Application

Now that we have a list of subjects that apply to the traffic between a particular set of endpoints, we're ready to describe how we actually apply policy to allow those endpoints to communicate. The applicable subjects from the previous step will each contain a set of rules.

Rules consist of a set of *classifiers* and a set of *actions*. Classifiers match against traffic between two endpoints. An example of a classifier would be something that matches against all TCP traffic on port 80, or one that matches against HTTP traffic containing a particular cookie. Actions are specific actions that need to be taken on the traffic before it reaches its destination. Actions could include tagging or encapsulating the traffic in some way, redirecting the traffic, or applying some service chain. For more information on how classifiers and actions are defined, see below under [Subject Features](#).

If and only if *all* classifiers on a rule matches, *all* the actions on that rule are applied (in order) to the traffic. Only the first matching rule will apply.

Rules, subjects, and actions have an *order* parameter, where a lower order value means that a particular item will be applied first. All rules from a particular subject will be applied before the rules of any other subject, and all actions from a particular rule will be applied before the actions from another rule. If more than item has the same order parameter, ties are broken with a lexicographic ordering of their names, with earlier names having logically lower order.

We've now reached final phase in the three-phases policy resolution process. First, we found the set of contract scopes to apply. Second, we found the set of subjects to apply. Finally, we saw how we apply the subjects to traffic between pairs of endpoints in order to realize the policy. The remaining sections will fill in additional detail for the policy resolution process.

Matchers

Matchers have been mentioned a few times now without really explaining what they are. Matchers specify a set of labels (which include requirements, capabilities, conditions, and qualities) to match against. There are several kinds of matchers that operate similarly:

- Quality matchers are used in target selectors during the contract selection phase. Quality matchers provide a more advanced and flexible way to select contracts compared to a named selector.
- Requirement matchers and capability matchers are used in clauses during the subject selection phase to match against requirements and capabilities on endpoint groups
- Condition matchers are used in clauses during the subject selection phase to match against conditions on endpoints

A matcher is, at its heart, fairly simple. It will contain a list of label names, along with a *match type*. The match type can be either "all," which means the matcher matches when all

of its labels match, "any," which means the matcher matches when any of its labels match, or "none," which means the matcher matches when none of its labels match. Note that a matcher which always matches can be made by matching against an empty set of labels with a match type of "all."

Additionally each label to match can optionally include a relevant "name" field. For quality matchers, this is a target name. For capability and requirement matchers, this is a selector name. If the name field is specified, then the matcher will only match against targets or selectors with that name, rather than any targets or selectors.

There are some additional semantics related to inheritance. Please see the section for [Inheritance](#) for more details.

Quality Matchers

A contract contains *targets* that are just a set of quality labels. A target selector on an endpoint group matches against these targets using quality matchers. A quality matcher is a matcher where the label it matches is a quality, and the name field is a target name.

Requirement and Capability Matchers

The matching selector from the contract selection phase will define either requirements or capabilities for the consumer and provider endpoint groups, respectively. Clauses can match against these labels using requirement and capability matchers. Requirements matchers match against requirements while capability matchers match against capabilities. In both cases, the name field is a selector.

Condition Matcher

Endpoints can have condition labels. The condition matcher can be used in a clause to match against endpoints with particular combinations of conditions.

Tenants

The system allows multiple tenants that are designed to allow separate domains of administration. Contracts and endpoint groups are defined within the context of a particular tenant. Endpoints that belong to endpoint groups in separate tenants cannot communicate with each other except through a special mechanism to allow cross-tenant contracts called *contract references*.

While it would be possible to define semantics for tenant inheritance, as currently defined there is no way for tenants to inherit from each other. There is, however, a limited mechanism through the special *common tenant* (see [Common Tenant](#) below). All references to names are within the scope of that particular tenant, with the limited exceptions of the common tenant and contract references.

Contract References

Contract references are the mechanism by which endpoints in different tenants can communicate. This is especially useful for such common use cases as gateway routers or other shared services. In order for an endpoint group to select a contract in a different tenant, there must first exist a contract reference defined in the endpoint group's local

tenant. The contract reference is just a tenant ID and a contract ID; this will bring that remote contract into the scope of the local contract. Note that this reference may be subject to additional access control mechanisms.

Endpoint groups can participate in such remotely-defined contracts only as consumers, not as providers.

Once the contract reference exists, endpoint groups can now select that contract using either named or target selectors. By defining a contract reference, the qualities and targets in that contract are imported into the namespace of the local tenant for the contract selection phase. Similarly, the requirements and conditions from the local tenant will be used when performing the consumer matches in the subject selection phase.

Common Tenant

The common tenant is an area where definitions that are useful for all tenants can be created. Everything defined in the common tenant behaves exactly as though it were defined individually in every tenant. This applies to resolution of labels for the purposes of contract selection, as well as subject feature instances (see [Subject Features](#) below).

If a name exists in both the common tenant and another tenant, then when resolving names within the context of that tenant the definition in the common tenant will be masked. One special case to consider is if a definition in a tenant defines the common tenant definition as its parent and uses the same name as the parent object. This works as you might expect: the name reference from the child definition will extend the behavior of the definition in the common tenant, but then mask the common tenant definition so that references to the name within the tenant will refer to the extended object.

Subject Features

Subject features are objects that can be used as a part of a subject to affect the communication between pairs of endpoints. This is where the policy model meets the underlying infrastructure. Because different networks will have different sets of features, we need some way to represent to the users of the policy what is possible. Subject features are the answer to this.

There are two kinds of subject features: classifiers and actions. Classifiers match on traffic between endpoints, and actions perform some operation on that traffic (See [Rule Application](#) above for more information on how they are used).

Subject features are defined with a subject feature definition. The definition defines a name and description for the feature, along with a set of parameters that the item can take. This is the most general description for the subject feature, and this definition is global and applies across all tenants. As an example, a classifier definition might be called "tcp_port", and would take an integer parameter "port".

Next, there are subject feature instances. Subject feature instances are scoped to a particular tenant, and reference a subject feature definition, but fill in all required parameters. To continue our example, we might define a classifier instance called "http" that references the "tcp_port" classifier and species the parameter "port" as 80.

Finally, there are subject feature references, which are references to subject feature instances. Subjects contain these subject feature references in order to apply the feature.

These references also contain, as appropriate an order field to determine order of operations and fields for matching the direction of the traffic.

If the underlying network infrastructure is unable to to implement a particular subject, then it must raise an exception related to that subject. It may then attempt to relax the constraints in a way that allows it to implement the policy. However, when doing this it must attempt to avoid allowing traffic that should not be allowed. That is, it should "fail closed" when relaxing constraints.

Forwarding Model

Communication between endpoint groups can happen at layer 2 or layer 3, depending on the policy configuration. We define our model of the forwarding behavior in a way that supports very flexible semantics including overlapping layer 2 and layer 3 addresses.

We define several kinds of *network domains*, which represent some logical grouping or namespace of network addresses:

| | |
|------------------|---|
| L3 Context | A layer 3 context represents a namespace for layer 3 addresses. It represents a domain inside which endpoints can communicate without requiring any address translation. A subtype of a forwarding context, which is a subtype of a network domain. |
| L2 Bridge Domain | A layer 2 bridge domain represents a domain in which layer 2 communication is possible when allowed by policy. Bridge domains each have a single parent L3 context. A subtype of an L2 domain, which is a subtype of a forwarding context. |
| L2 Flood Domain | A layer 2 flood domain represents a domain in which layer 2 broadcast and multicast is allowed. L2 flood domains each have a single parent L2 bridge domain. A subtype of an L2 domain. |
| Subnet | An IP subnet associated with a layer 2 or layer 3 context. Each subnet has a single parent forwarding context. A subtype of a network domain. |

Every endpoint group references a single network domain.

Inheritance

This section contains information on how inheritance works for various objects in the system. This is covered here to avoid cluttering the main sections with a lot of details that would make it harder to see the big picture.

Some objects in the system include references to parents, from which they will inherit definitions. The graph of parent references must be loop free. When resolving names, the resolution system must detect loops and raise an exception. Objects that are part of these loops may be considered as though they are not defined at all.

Generally, inheritance works by simply importing the objects in the parent into the child object. When there are objects with the same name in the child object, then the child

object will override the parent object according to rules which are specific to the type of object. We'll next explore the detailed rules for inheritance for each type of object.

Endpoint Groups

Endpoint groups will inherit all their selectors from their parent endpoint groups. Selectors with the same names as selectors in the parent endpoint groups will inherit their behavior as defined below.

Selectors

Selectors include provider named selectors, provider target selectors, consumer named selectors, and consumer target selectors. Selectors cannot themselves have parent selectors, but when selectors have the same name as a selector of the same type in the parent endpoint group, then they will inherit from and override the behavior of the selector in the parent endpoint group.

Named Selectors

Named selectors will add to the set of contract IDs that are selected by the parent named selector.

Target Selectors

A target selector in the child endpoint group with the same name as a target selector in the parent endpoint group will inherit quality matchers from the parent. If a quality matcher in the child has the same name as a quality matcher in the parent, then it will inherit as described below under Matchers.

Contracts

Contracts will inherit all their targets, clauses and subjects from their parent contracts. When any of these objects have the same name as in the parent contract, then the behavior will be as defined below.

Targets

Targets cannot themselves have a parent target, but it may inherit from targets with the same name as the target in a parent contract. Qualities in the target will be inherited from the parent. If a quality with the same name is defined in the child, then this does not have any semantic effect except if the quality has its inclusion-rule parameter set to "exclude." In this case, then the label should be ignored for the purpose of matching against this target.

Subjects

Subjects cannot themselves have a parent subject, but it may inherit from a subject with the same name as the subject in a parent contract.

The order parameter in the child subject, if present, will override the order parameter in the parent subject.

The rules in the parent subject will be added to the rules in the child subject. However, the rules will *not* override rules of the same name. Instead, all rules in the parent subject will

be considered to run with a higher order than all rules in the child; that is all rules in the child will run before any rules in the parent. This has the effect of overriding any rules in the parent without the potentially-problematic semantics of merging the ordering.

Clauses

Clauses cannot themselves have a parent clause, but it may inherit from a clause with the same name as the clause in a parent contract.

The list of subject references in the parent clause will be added to the list of subject references in the child clause. There is no meaningful overriding possible here; it's just a union operation. Note of course though that a subject reference that refers to a subject name in the parent contract might have that name overridden in the child contract.

Each of the matchers in the clause are also inherited by the child clause. Matchers in the child of the same name and type as a matcher from the parent will inherit from and override the parent matcher. See below under [Matchers](#) for more information.

Matchers

Matchers include quality matchers, condition matchers, requirement matchers, and capability matchers. Matchers cannot themselves have parent matchers, but when there is a matcher of the same name and type in the parent object, then the matcher in the child object will inherit and override the behavior of the matcher in the parent object.

The match type, if specified in the child, overrides the value specified in the parent.

Labels are also inherited from the parent object. If there is a label with the same name in the child object, this does not have any semantic effect except if the label has its inclusion-rule parameter set to "exclude." In this case, then the label should be ignored for the purpose of matching. Otherwise, the label with the same name will completely override the label from the parent.

Subject Feature Definitions

Subject features definitions, including classifier definitions and subject definitions can also inherit from each other by specifying a parent object. These are a bit different from the other forms of override because they do not merely affect the policy resolution process, but rather affect how the policy is applied in the underlying infrastructure.

For the purposes of policy resolution, a subject feature will inherit from its parent any named parameters. However, unlike in other cases, if a named parameter with the same name exists in the child as in the parent, this is an invalid parameter and will be ignored in the child. That is, the child *cannot* override the type of a named parameter in a child subject feature.

For the purposes of applying the subject in the underlying infrastructure, the child subject feature is assumed to add some additional functionality to the parent subject feature such that the child feature is a specialization of that parent feature. For example, there might be a classifier definition for matching against a TCP port, and a child classifier definition that allowed for deep packet inspection for a particular protocol that extended the base

classifier definition. In this case, the child classifier would be expected to match the TCP port as well as apply its additional deep packet inspection semantics.

If the underlying infrastructure is unable to apply a particular subject feature, it can attempt to fall back to implementing instead the parent subject feature. The parameter `fallback-behavior` determines how this should apply. If this is set to "strict" then a failure to apply the child is a fatal error and the entire subject must be ignored. If the fallback behavior is "allow-fallback" then the error is nonfatal and it is allowed to apply instead only the parent subject feature.

State Repositories

The state repositories are distributed data stores that provide the configuration and operational data required for renderers to apply the policy as specified by the user. The state repositories all model their state as yang models, and store that state in the MD-SAL data store as either operational or configuration data, as appropriate. The state repositories implement a minimum amount of actual functionality and instead focus on defining the models and supporting the correct querying and subscription semantics. The intelligence is expected to be in the renderers.

Querying and Subscription

State repositories support both simple queries on the data but more important allow subscriptions to the data, so that systems that are responsible for applying the policy model are informed about changes to that policy configuration or operational state that might affect the policy. Those subsystems are expected to continuously reevaluate the policy as these changes come in make the required changes in the underlying infrastructure.

Endpoint Repository

The endpoint repository is responsible for storing metadata about endpoints, including how they are mapped into endpoint groups. Information about endpoints can be added to the repository either by a central orchestration system or by a renderer that performs discovery to learn about new endpoints. In either case, the semantics of how an endpoint is mapped to an endpoint group are not defined here; the system that sets up the information in the endpoint repository must have its own method for assigning endpoints to endpoint groups.

Policy Repository

The policy repository stores the policies themselves. This includes endpoint groups, selectors, contracts, clauses, subjects, rules, classifiers, actions, and network domains (everything in the policy model except endpoints and endpoint-related metadata). The policy repository is populated through the northbound APIs.

Status Repository

The status repository will be added in a future release of group-based policy.

Renderers

One of the key design features of the group-based policy architecture is that it can support a variety of renderers based on very different underlying technology. This is possible because the policy model is based only on high-level user intent, and contains no information about the details of how the network traffic is actually forwarded. However, one consequence of this design choice is that the renderers actually contain most of the complexity in the design of the system, and most of the real problems in building a software-defined virtual network solution will need to be solved by the renderers themselves.

Renderer Common Framework

The renderers have available to them some service and libraries that collectively make up the *renderer common framework*. These are not actually required to implement a renderer, but where convenient functionality that would be generally useful should be placed here.

InheritanceUtils

This class provides a convenient utility to resolve all the complex inheritance rules into a normalized view of the policy for a tenant.

```
/**
 * Fully resolve the specified {@link Tenant}, returning a tenant with all
 * items fully normalized. This means that no items will have parent/child
 * relationships and can be interpreted simply without regard to inheritance
 * rules
 * @param tenantId the {@link TenantId} of the {@link Tenant}
 * @param transaction a {@link DataModificationTransaction} to use for
 * reading the data from the policy store
 * @return the fully-resolved {@link Tenant}
 */
public static Tenant resolveTenant(TenantId tenantId,
                                   DataModificationTransaction transaction)
```

PolicyResolverService

The policy resolver service resolves the policy model into a representation suitable for rendering to an underlying network. It will run through the contract resolution and

The policy resolver is a utility for renderers to help in resolving group-based policy into a form that is easier to apply to the actual network.

For any pair of endpoint groups, there is a set of rules that could apply to the endpoints on that group based on the policy configuration. The exact list of rules that apply to a given pair of endpoints depends on the conditions that are active on the endpoints.

In a more formal sense: Let there be endpoint groups G_n , and for each G_n a set of conditions C_n that can apply to endpoints in G_n . Further, let S be the set of lists of rules defined in the policy. Our policy can be represented as a function $F: (G_n, 2^{C_n}, G_m, 2^{C_m}) \rightarrow S$, where 2^{C_n} represents the power set of C_m . In other words, we want to map all the possible tuples of pairs of endpoints along with their active conditions onto the right list of rules to apply.

We need to be able to query against this policy model, enumerate the relevant classes of traffic and endpoints, and notify renderers when there are changes to policy as it applies to active sets of endpoints and endpoint groups.

The policy resolver will maintain the necessary state for all tenants in its control domain, which is the set of tenants for which policy listeners have been registered.

Open vSwitch-Based Overlay Renderers

This section describes a data plane architecture for building a network virtualization solution using Open vSwitch. This data plane design is used by two renderers: the [OpenFlow Renderer](#) and the [OpFlex Renderer](#).

The design implements an overlay design and is intended to meet the following use cases:

- Routing when required between endpoint groups, including serving as a distributed default gateway.
- Optional broadcast within a bridge domain.
- Management of L2 broadcast protocols including ARP and DHCP to avoid broadcasting.
- Layer 2-4 classifiers for policy between endpoint groups, including connection tracking/reflexive ACLs.
- Service insertion/redirection

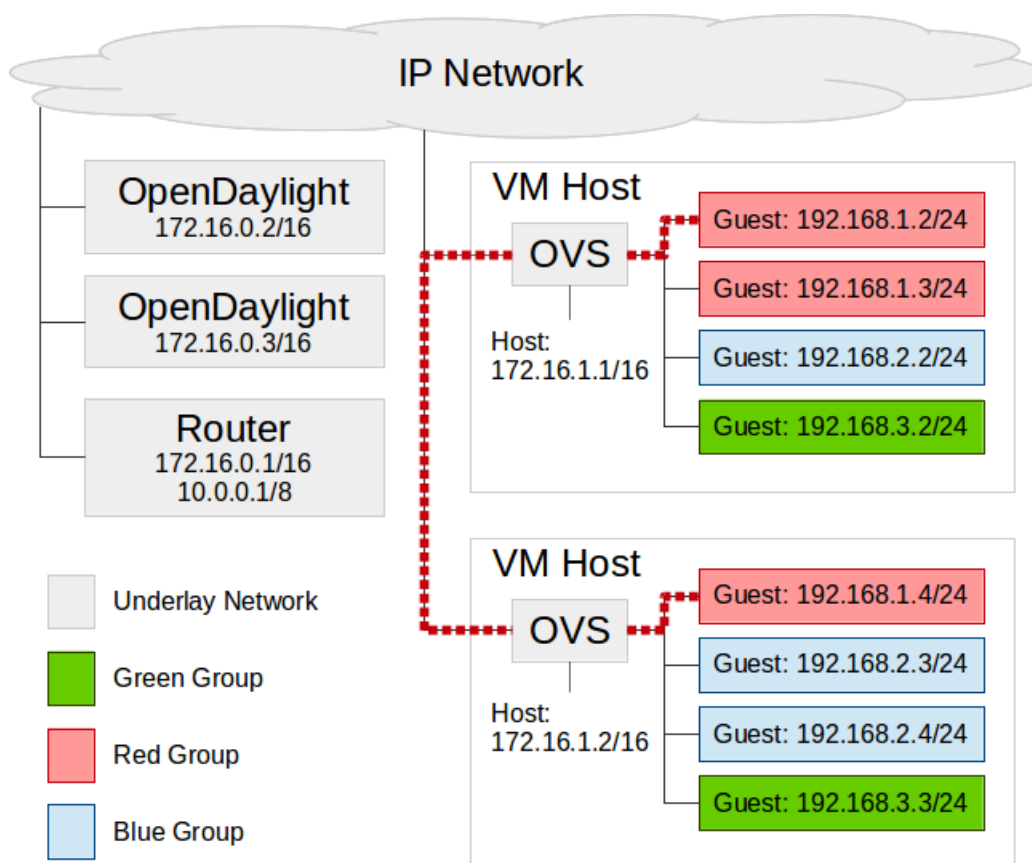
Network Architecture

Network Topology

The network architecture is an overlay network based on VXLAN or similar encapsulation technology, with an underlying IP network that provides connectivity between hypervisors and the controller. The overlay network is a full-mesh set of tunnels that connect each pair of vSwitches.

The "underlay" IP network has no special requirements though it should be set up with ECMP to the top-of-rack switch for the best performance, but this is not a strict requirement for correct behavior. Also, the underlay network should be configured with a path MTU that's large enough to accommodate the overlay tunnel headers. For a typical overlay network with a 1500 byte MTU, a 1600 byte MTU in the underlay network should be sufficient. If this is not configured correctly, the behavior will be correct but it will result in fragmentation which could have a severe negative effect on performance.

Physical devices such as routers on the IP network are trusted entities in the system since these devices would have the ability to forge encapsulated packets.

Figure 8.6. GBP OVS Network Topology Example

The [Network Topology Example](#) figure shows an example of a supported network topology, with an underlying IP network and hypervisors with Open vSwitch. Infrastructure components and elements of the underlay network are shown in grey. Three endpoint groups exist with different subnets in the same layer 3 context, which are shown in red, green, and blue. A tunneled path (dotted red line) is shown between two red virtual machines on different VM hosts.

Control Network

The security of the system depends on keeping a logically isolated control network separate from the data network, so that guests cannot reach the control network. Ideally, the network is kept isolated through an out-of-band control network. This can be accomplished using a separate NIC, a special VLAN, or other mechanism. However, the system is also designed to operate in the case where the control traffic and the data traffic are on the same layer 2 network and isolation is still enforced.

In the [Network Topology Example](#) figure above, the control network is shown as 172.16/16. The VM hosts, and controllers all have addresses on this network, and communicate using OpenFlow and OVSDb on this network. In the example, the router is shown with an interface configured on this network as well; this works but in practice it is preferable to isolate this network by accessing it through a VPN or jump box if needed. Note that there is no requirement that the control network be all in one subnet.

The router is also shown with an interface configured on the 10/8 network. This network will be used for routing traffic destined for internet hosts. Both the 172.16/16 and 10/8 networks here are isolated from the guest address spaces.

Overlay Network

Whenever traffic between two guests is in the network, it will be encapsulated using a VXLAN tunnel (though supporting additional encapsulation formats could be configured in the future). A packet encapsulated as VXLAN contains:

- Outer ethernet header, with source and destination MAC
- Outer IP header, with source and destination IP address
- Outer UDP header
- VXLAN header, with a virtual network identifier (VNI). The virtual network identifier is a 24-bit field that uniquely identifies an endpoint group in our policy model.
- Encapsulated original packet, which includes:
 - Inner ethernet header, with source and destination MAC
 - (Optional) Inner IP header, with source and destination IP address

Delivering Packets

Endpoints can communicate with each other in a number of different ways, and each is processed slightly differently. Endpoint groups exist inside a particular layer 2 or layer 3 context which represents a namespace for their network identifiers. It's only possible for endpoints to address endpoints within the same context, so no communication is possible for endpoints in different layer 3 contexts, and only layer 3 communication is possible for endpoints in different layer 2 contexts.

Overlay Tunnels

The next key piece of information is the location of the destination endpoint. For destinations on the same switch, we can simply apply policy (see below), perform any routing action required (see below), then deliver it to the local port.

When the endpoints are located on different switches, we need to use the overlay tunnel. This is the case shown as a dotted red line in the [Network Topology Example](#) figure. After policy is applied to the packet, we encapsulated it in a tunnel with the tunnel ID set to a unique ID for the destination endpoint group. The outer packet is addressed to the IP address of the OVS host that hosts the destination endpoint. This encapsulated packet is now sent out to the underlay network, which is just a regular IP network that can deliver the packet to the destination switch.

When the encapsulated packet arrives on the other side, the destination vSwitch inspects the metadata of the encapsulation header to see if the policy has been applied already. If the policy has not been applied or if the encapsulation protocol does not support carrying of metadata, the policy must be applied at the destination vSwitch. The packet can now be delivered to the destination endpoint.

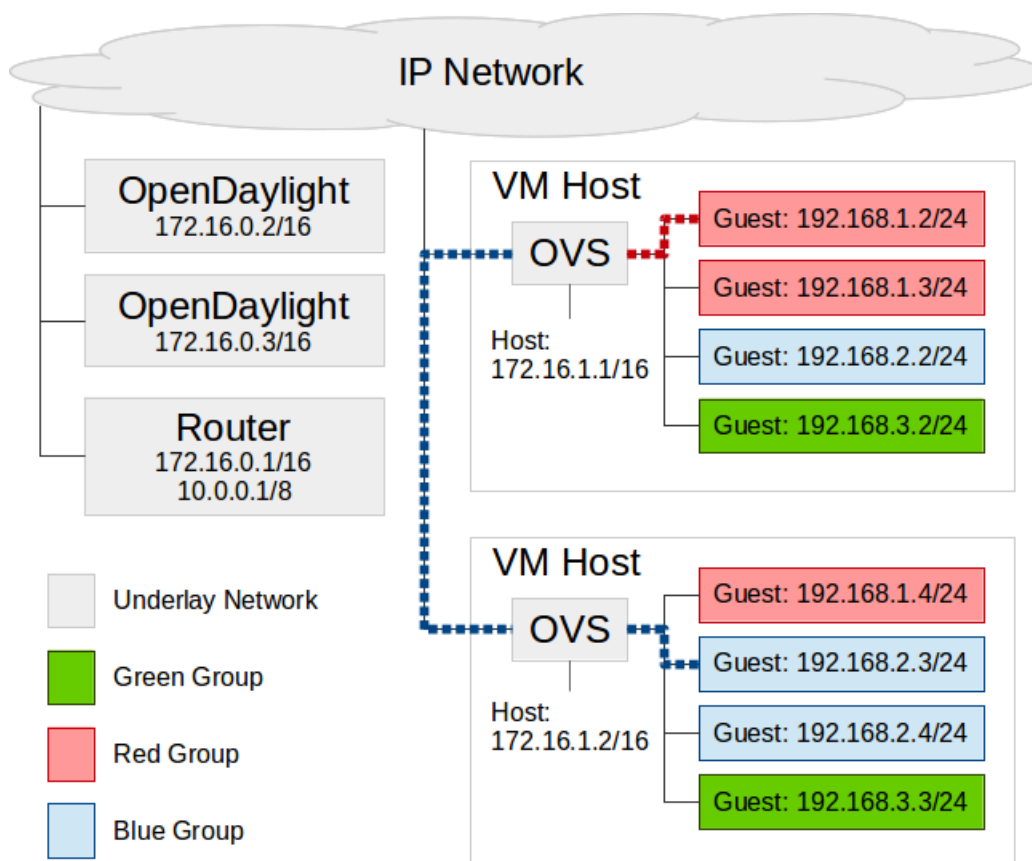
Bridging and Routing

The system will transparently handle bridging or routing as required. Bridging occurs between endpoints in the same layer 2 context, while routing will generally be needed for endpoints in different layer 2 contexts. More specifically, a packet needs to be routed if it is addressed to the gateway MAC address. We can simply use a fixed MAC address to serve as the gateway everywhere. Packets addressed to any other MAC address can be bridged.

Bridged packets are easy to handle, since we don't need to do anything special to them to deliver them to the destination. They can be simply delivered unmodified.

Routing is slightly more complex, though not massively so. When routing locally on a switch, we simply rewrite the destination MAC address to the MAC of the destination endpoint, and set the source MAC to the gateway MAC, decrement the TTL, and then deliver it to the correct local port.

When routing to an endpoint on a different switch, we'll actually perform routing in two steps. On the source switch, we will decrement TTL and rewrite the source MAC address to the MAC of the gateway router (so that both the source and the destination MAC are set to the gateway router's MAC). It's then delivered to the destination switch using the appropriate tunnel. On the destination switch, we perform a second routing action by now rewriting the destination MAC as the MAC address of the destination endpoint and decrementing the TTL again. The reason why do the routing as two hops is that this avoids the need to maintain on every switch the correct MAC address for every endpoint on the network. Each switch needs the mappings only for endpoints that are directly attached to that switch. An example of a communication pathway requiring this routing is shown in the figure below.

Figure 8.7. GBP OVS Routing Example

In this example, we show the path of traffic from the blue guest 192.168.2.3 and the red guest 192.168.1.2. The traffic is encapsulated in a tunnel marked with the blue endpoint group's VNI while in transit between the two switches. Because two endpoints are in different subnets, the traffic is routed in two hops: one the source switch and one on the destination switch.

The vSwitch on each host must respond to local ARP requests for the gateway IP address and return a logical MAC address representing the L3 gateway.

Communicating with Outside Hosts

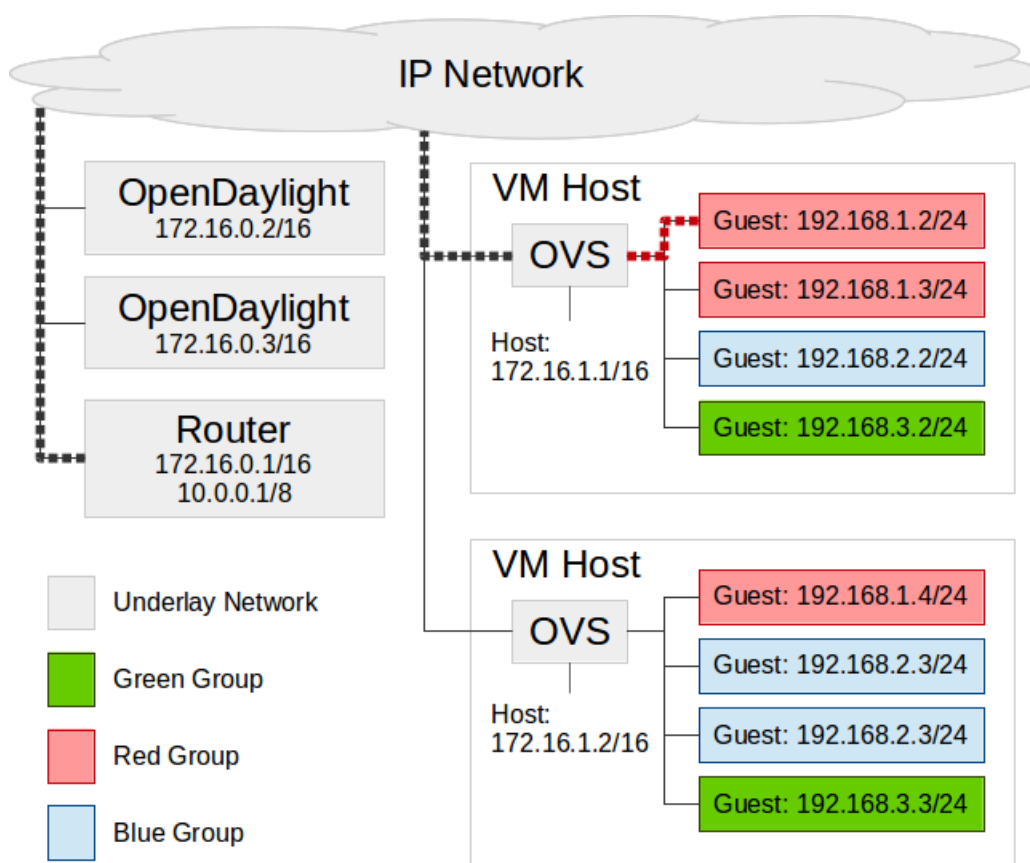
Everything up until now is quite simple, but it's possible to conceive of situations where endpoints in our network need to communicate over the internet or with other endpoints outside the overlay network. There are two broad approaches for handling this. In both cases, we allow such access only via layer 3 communication.

First, we can map physical interfaces on an OVS system into the overlay network. If a router interface is attached either directly to a physical interface or indirectly via an isolated network, then the router interface can be easily exposed as an endpoint in the network. Endpoints can then communicate with this router interface (perhaps after some intermediate routing via the distributed routing scheme described above) and from there get to the rest of the world. Dedicated OVS systems can be thus configured as gateway devices into the overlay network which will then be needed for any of this north/south

communication. This has the advantage of being very conceptually simple but requires special effort to load balance the traffic effectively.

Second, we can use a DNAT scheme to allow access to endpoints that are reachable via the underlay network. In this scheme, for every endpoint that is allowed to communicate to these outside hosts, we allocate an IP address from a dedicated set of subnets on the underlay (each network segment in the underlay network will require a separate DNAT range for switches attached to that subnet). We can perform the DNAT translation on the OVS switch and then simply deliver the traffic to the underlay network to deliver to the internet host or other host, and perform the reverse translation to get back into the overlay network.

Figure 8.8. GBP OVS Example of Communication With Outside Endpoints

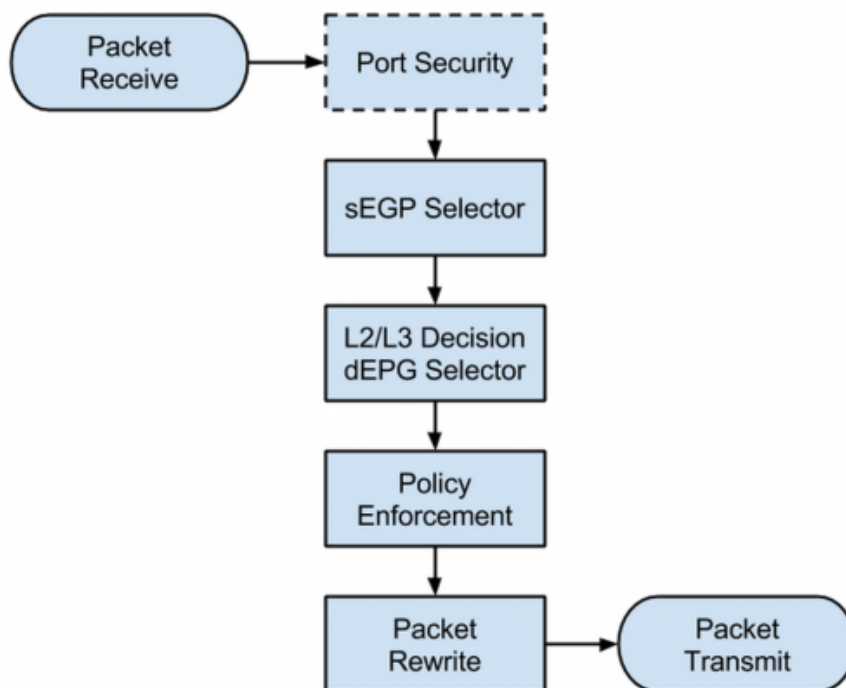


An example of communication with outside endpoints using the DNAT scheme is shown in the figure above. In this example, the red endpoint is communicating with an endpoint on the internet through a gateway router. The traffic goes through a DNAT translation to an IP allocated to the endpoint for this purpose. The IP communication can then be delivered through the IP underlay network.

For the first implementation, we'll stick with the DNAT scheme and consider implementing the gateway-based or other solution.

Packet Processing Pipeline

Figure 8.9. GBP OVS Packet Processing Pipeline



Here is a simplified high-level view of what happens to packets in this network when it hits an OVS instance:

1. If data and management network are shared, determine whether packet is targeted for the host system. If so, reinject into host networking stack.
2. Apply port security rules if enabled on the port to determine if the source identifiers (MAC and IP) are allowed on the port
 - For packets received from the overlay: Determine the source endpoint group (sEPG) based on the tunnel ID from the outer packet header.
 - For packets received from local ports: Determine sEPG based on source port and source identifiers as configured.
 - As an sEPG can only be associated with a single L2 and L3 context, the context is determined in this step as well.
 - Unknown source identifiers may result in a packet-in if the network is doing learning.
3. Handle broadcast and multicast packets while respecting broadcast domains.
4. Catch any special packet types that are handled specially. This could include ARP, DHCP, or LLDP. How these are handled may depend on the specific renderer implementation.
5. Determine whether the packet will be bridged or routed. If the destination MAC address is the default gateway MAC, then the packet will be routed, otherwise it will be bridged.

6. Determine the destination endpoint group (dEPG) and outgoing port or next hop while respecting the L2/L3 context.
 - For bridged packets (L2): Determine based on the destination MAC address.
 - For routed packets (L3): Determine based on the destination IP address.
7. Apply the appropriate set of policy rules based on the active subjects for that flow. We can bypass this step if the tunnel metadata indicates that the policy has been applied at the source.
8. Apply a routing action if needed by modifying the destination and source MAC and decrementing the TTL.
 - For local destination: Rewrite the destination MAC to the MAC address for the connected endpoint, source MAC to the MAC of the default gateway.
 - For remote destinations: Rewrite the destination MAC to the MAC of the next hop, source MAC to the MAC of the default gateway.
9. If the next hop is a local port, then it is delivered as-is. If the next hop is not local, then the packet is encapsulated and the tunnel ID is set to the network identifier for the source endpoint group (sEPG). If the packet is a layer 2 broadcast packet, then it will need to be written to the correct set of ports, which might be a combination of local and multiple remote tunnel endpoints.

Register Usage

The processing pipeline needs to store metadata such as the sEPG, dEPG, and broadcast domain. This metadata can be stored in any way supported by the switch. OpenFlow provides a dedicated 64 bit metadata field, Open vSwitch additionally provides multiple 32 bit registers in form of Nicira Extensions. The following examples will use Nicira extensions for simplicity. The choice of register usage is an implementation detail of the renderer.

Option 1: Register allocation using Nicira Extensions

| Register | Value |
|-------------|---|
| NXM_NX_REG1 | Source Endpoint Group (sEPG) ID |
| NXM_NX_REG2 | L2 context (BD) |
| NXM_NX_REG3 | Destination Endpoint Group (dEPG) ID |
| NXM_NX_REG4 | Port number to send packet to after policy enforcement. This is required because port selection occurs before policy enforcement in the pipeline. |
| NXM_NX_REG5 | L3 context ID (VRF) |

Option 2: Register allocation using OpenFlow metadata

OpenFlow offers a single 64 bit register which can be used to store sEPG, dEPG, and BD throughout the lookup process alternatively. The advantage over using Nicira extensions is better portability and offload capability to hardware.

| Register | Value |
|------------------|--------------------------------------|
| metadata[0..15] | Source Endpoint Group (sEPG) ID |
| metadata[16..31] | Destination Endpoint Group (dEPG) ID |

| Register | Value |
|------------------|---|
| metadata[32..39] | L2 context (BD) |
| metadata[40..47] | L3 context (VRF) |
| metadata[48..63] | Port number to send packet to after policy enforcement. This is required because port selection occurs before policy enforcement in the pipeline. |

Table/Pipeline Names and Order

In order to increase readability, the following table names are used in the following sections. Their order in the pipeline is as follows:

| Table | ID | Description | Flow Hit | Flow Miss |
|-------|-----------------|------------------------------|--|--|
| 1 | PORT_SECURITY | Optional port security table | Proceed to SEPG_FILTER | Drop |
| 2 | SEPG_FILTER | sEPG selection | Remember sEPG, BD, and VRF. Then proceed to DEPG_FILTER | Trigger policy resolution (send to controller) |
| 3 | DEPG_FILTER | dEPG selection | Remember dEPG and output coordinates, proceed to POLICY_ENFORCER | Trigger policy resolution (send to controller) |
| 4 | POLICY_ENFORCER | Policy enforcement | Forward packet | Drop |

OpenFlow ≥ 1.1 capable switches can implement the flow miss policy for each table directly. Pure OpenFlow 1.0 switches will need to have a catch-all flow inserted to enforce the specified policy.

Port Security

An optional port security table can be inserted at the very beginning of the pipeline. It enforces a list of valid sMAC and sIP addresses for a specific port.

```
priority=30, in_port=TUNNEL_PORT, actions=goto_table:SEPG_FILTER
priority=30, in_port=PORT1, dl_src=MAC1, action=goto_table:SEPG_FILTER
priority=30, in_port=PORT2, dl_src=MAC2, ip, nw_src=IP2, actions=
goto_table:SEPG_FILTER
priority=20, in_port=PORT2, dl_src=MAC2, ip, actions=drop
priority=10, in_port=PORT2, dl_src=MAC2, actions=goto_table:SEPG_FILTER
priority=30, in_port=PORT3, actions=goto_table:SEPG_FILTER
```

The port-security flow-miss policy is set to drop in order for packets received on an unknown port or with an unknown sMAC/sIP to be rejected.

The following modes of enforcement are defined:

1. **Whitelisted:** The port is allowed to use any addresses. All tunnel ports must be whitelisted. The filter is enforced with a single flow matching on in_port and redirects to the next table.
2. **L2 enforcement:** Any packet from the port must use a specific sMAC. The filter is enforced with a single flow matching on the in_port and dl_src and redirects to the next table.
3. **L3 enforcement:** Same as L2 enforcement. Additionally, any IP packet from the port must use a specific sIP. The filter is enforced with three flows with different priority.
 - a. Any IP packet with correct sMAC and sIP is redirected to the next table.

- b. Any IP packet left over is dropped.
- c. Any non-IP packet with correct sMAC is redirected to the next table.

Source EPG & L2/L3 Domain Selection

The sEPG is determined based on a separate flow table which maps known OpenFlow port numbers and tunnel identifiers to a locally unique sEPG ID. The sEPG ID is stored in register `NXM_NX_REG1` for later use in the pipeline. At the same time, the L2 and L3 context is determined and stored in register `NXM_NX_REG2`.

| Field | Description |
|-------------------------------|---|
| <code>table=SEPG_TABLE</code> | Flow must be in sEPG selection table |
| <code>in_port=\$OFPORT</code> | Flow must match on incoming port |
| <code>tun_id=\$VNI</code> | If <code>in_port</code> is a tunnel, flow must match on tunnel ID |

The actions performed are:

1. Write sEPG ID corresponding to incoming port or tunnel ID to register
2. Write L2/L3 context ID corresponding to incoming port or tunnel ID to registers
3. Proceed to dEPG selection

An example flow to map a local port to an sEPG:

```
table=SEPG_FILTER, in_port=$OFPORT
actions=load:$SEPG->NXM_NX_REG1[ ],
        load:$BD->NXM_NX_REG2[ ],
        load:$VRF->NXM_NX_REG5[ ],
        goto_table:$DEPG_FILTER
```

An example flow to map a tunnel ID to an sEPG:

```
table=SEPG_FILTER, in_port=TUNNEL_PORT, tun_id=$VNI1,
actions=load:$SEPG1->NXM_NX_REG1[ ],
        load:$BD->NXM_NX_REG2[ ],
        load:$VRF->NXM_NX_REG5[ ],
        goto_table:$DEPG_FILTER
```

A flow hit means that the sEPG is known and the pipeline should proceed to the next stage.

A flow miss means that we have received a packet from an unknown EPG:

1. If the packet was received on a local port then this corresponds to the discovery of a new EP for which the Port to EPG mapping has not been populated yet. If the network is learned, generate a packet-in to trigger policy resolution, otherwise drop the packet.
2. If the packet was received from a tunnel then this corresponds to a packet for which we have not populated the tunnel ID to EGP mapping yet. If the network is learned, generate a packet-in to trigger policy resolution, otherwise drop the packet.

Broadcasting / Multicasting

Packets sent to the MAC broadcast address (`ff:ff:ff:ff:ff:ff`) must be flooded to all ports belonging to the broadcast domain. This is **not** equivalent to the OVS flood action as

multiple broadcast domains reside on the same switch. The respective broadcast domains are modeled using OpenFlow group tables as follows:

1. Upon addition of a new broadcast domain to the local vSwitch:

- Create a new OpenFlow group table, using the BD ID as group ID

```
ovs-ofctl [...] add-group BRIDGE group_id=$BD, type=all
```

- Create a flow in the dEPG selection table matching on broadcast packets and correctly have them flooded to all group members:

```
priority=10, table=$DEPG_TABLE, reg2=$BD, dl_dst=ff:ff:ff:ff:ff:ff,  
actions=group:$BD
```

2. Upon addition/removal of a local port

- Modify group and add/remove output action to port to account for membership change:

```
osvs-ofctl [...] mod-group $BRIDGE [Old entry,] bucket=output:$PORT
```

3. Upon addition/removal of a non-local port to the BD

- Modify group and add/remove output + tunnel action to start/stop flooding packets over overlay

Special Packet Types

ARP Responder

In order for the distributed L3 gateway to be reachable, the vSwitch must respond to ARP requests sent to the default gateway address. For this purpose, a flow is added which translates ARP requests into ARP replies and sends them back out the incoming port.

| Field | Description |
|-------------------|---|
| priority=20 | Must have higher priority than regular, non-ARP dEPG table flows. |
| table=DEPG_FILTER | Flow must be in dEPG selection table |
| reg5=2 | Must match a specific L3 context (NXM_NX_REG5) |
| arp, arp_op=1 | Packet must be ARP request |
| arp_tpa=GW_IP | ARP request must be targeted for IP of gateway |

The actions performed are:

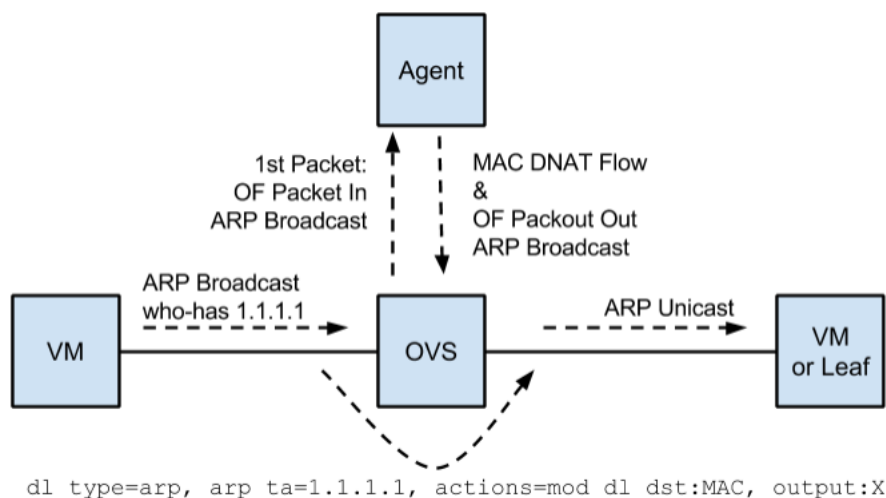
1. Set dMAC to original sMAC of packet to reverse direction
2. Set sMAC to MAC of gateway
3. Set ARP operation to (arp-reply)
4. Set target hardware address to original source hardware address
5. Set source hardware address to MAC of gateway

6. Set target protocol address to original source protocol address
7. Set source protocol address to IP of gateway
8. Transmit packet back out the incoming port

```
priority=20, table=DEPG_FILTER, reg5=$VRF,
arp, arp_op=1, arp_tpa=$GW_ADDRESS,
actions=move:NXM_OF_ETH_SRC[]->NXM_OF_ETH_DST[],
      mod_dl_src:$GW_MAC,
      load:2->NXM_OF_ARP_OP[],
      move:NXM_NX_ARP_SHA[]->NXM_NX_ARP_THA[],
      load:'Hex('$GW_MAC')'->NXM_NX_ARP_SHA[],
      move:NXM_OF_ARP_SPA[]->NXM_OF_ARP_TPA[],
      load:'Hex('$GW_ADDRESS')'->NXM_OF_ARP_SPA[],
in_port
```

ARP Optimization

Figure 8.10. GBP OVS ARP Optimization



As the MAC / IP pairing of endpoints is known in the network. ARP requests can be optimized and translated into unicasts. While it is possible to have a local vSwitch become an ARP responder directly, the unicast translation offers a minimal aliveness check within the scope of the L2 context.

A flow is inserted into the sEPG selection table as follows:

```
priority=10, arp, arp_op=1, dl_dst=ff:ff:ff:ff:ff:ff, actions=controller
```

As the ARP request is received, the packet is sent to the controller. The controller/agent resolves the MAC address to the IP address and inserts a new DNAT flow to translate subsequent ARP requests for the same transport address directly in the vSwitch:

```
priority=15, table=DEPG_FILTER,
arp, arp_op=1, dl_dst=ff:ff:ff:ff:ff:ff,
actions=mod_dl_dst:$MAC,
      load:${DEPG}->NXM_NX_REG3[],
      load:${OFPORT}->NXM_NX_REG4[],
```

```
goto_table:$ENFORCER_TABLE
```

The `OFPORT` is either a local port or the tunnel port. The latter case requires to additionally set the tunnel ID as described in previous sections.



Note

The controller can proactively insert ARP optimization flows for local or even remote endpoints to avoid the one time controller round trip penalty.

The controller/agent then reinjects the original ARP request back into the network via a packet-out OpenFlow message.

Destination EPG Selection (L2)

The dEPG selection is performed after the sEPG has been determined. The mapping occurs in its own flow table which contains both L2 and L3 flow entries. This section explains L2 processing, L3 processing is described in the next section.

The purpose of flow entries in this table is to map known destination MAC addresses in a specific L2 context to a dEPG and to prepare the action set for execution after policy enforcement.

| Field | Description |
|-------------------|--|
| priority=10 | Must have lower priority than L3 flows |
| table=DEPG_FILTER | Flow must be in dEPG selection table |
| reg2=2 | Must match on L2 context (NXM_NX_REG2) |
| dl_dst=MAC | Packet must match on destination MAC of the EP |

The actions performed are:

1. Write dEPG ID corresponding to dMAC to register to allow matching on it during policy enforcement
2. Write expected outgoing port number to register. This can be a local or a tunnel port.
3. If outgoing port is a tunnel, also include an action to set the tunnel ID and tunnel destination to map the sEPG to the tunnel ID.
4. Proceed to policy enforcement

Example flow for a local endpoint mapping:

```
table=$DEPG_FILTER, reg2=$BD, dl_dst=$MAC,
actions=load:$DEPG->NXM_NX_REG3[],
        load:$OFPORT->NXM_NX_REG4[],
        goto_table:$ENFORCER_TABLE
```

Example flow for a remote endpoint mapping:

```
table=$DEPG_FILTER, reg2=$BD, dl_dst=$MAC,
actions=load:$DEPG->NXM_NX_REG3[],
        load:$TUNNEL_PORT->NXM_NX_REG4[],
        move:NXM_NX_REG1[]->NXM_NX_TUN_ID[],
        load:$TUNNEL_DST->NXM_NX_TUN_IPV4_DST[],
```



```
goto_table:$ENFORCER_TABLE
```

A flow hit indicates that both the sEPG and dEPG are known at this point at the packet can proceed to policy enforcement.

A flow miss indicates that the dEPG is not known. If the network is in learning mode, generate a packet-in, otherwise drop the packet.

Destination EPG Selection (L3)

Much like L2 flows in the dEPG selection table, L3 flows map known destination IP addresses to the corresponding dEPG and outgoing port number.

Additionally, flow hits will result in a routing action performed.

| Field | Description |
|-------------------|---|
| priority=15 | Must have higher priority than L2 but lower than ARP flows. |
| table=DEPG_FILTER | Flow must be in dEPG selection table |
| reg5=2 | Must match on L3 context (NXM_NX_REG5) |
| dl_dst=GW_MAC | Packet must match MAC of gateway |
| nw_dst=PREFIX | Packet must match on a IP subnet |

The actions performed are:

1. Write dEPG ID corresponding to destination subnet to register to allow matching on it during policy enforcement
2. Write expected outgoing port number to register. This can be a local or a tunnel port
3. If outgoing port is a tunnel, also include an action to set the tunnel ID and tunnel destination to map the sEPG to the tunnel ID.
4. Modify destination MAC to the nexthop. The nexthop can be the MAC of the EP or another router.
5. Set source MAC to MAC of local default gateway
6. Decrement TTL
7. Proceed to policy enforcement

Example flow for a local endpoint over L3:

```
table=DEPG_TABLE, reg5=$VRF, dl_dst=$ROUTER_MAC, ip, nw_dst=$PREFIX,
actions=load:$DEPG->NXM_NX_REG3[],
        load:$OFPORT->NXM_NX_REG4[],
        mod_dl_dst:$DST_EP_MAC,
        mod_dl_src:$OWN_ROUTER_MAC,
        dec_ttl,
        goto_table:$POLICY_ENFORCER
```

Example flow for a remote endpoint over L3:

```
table=DEPG_TABLE, reg5=$VRF, dl_dst=$ROUTER_MAC, ip, nw_dst=$PREFIX,
```

```
actions=load:$DEPG->NXM_NX_REG3[ ],
        load:$TUNNEL_PORT->NXM_NX_REG4[ ],
        move:NXM_NX_REG1[ ]->NXM_NX_TUN_ID[ ],
        load:$TUNNEL_DST->NXM_NX_TUN_IPV4_DST[ ],
        mod_dl_dst:$NEXTHOP,
        mod_dl_src:$OWN_ROUTER_MAC,
        dec_ttl,
        goto_table:$POLICY_ENFORCER
```

Policy Enforcement

Given the sEPG, BD/VRF, and dEPG are known at this point, the policy is enforced in a separate flow table by matching on the sEPG and dEPG as found in the respective registers. Additional filters may be provided as specified by the policy.

| Field | Description |
|-----------------------|---|
| table=POLICY_ENFORCER | Flow must be in policy enforcement table. |
| reg1=\$SEPG | Must match on sEPG of packet |
| reg3=\$DEPG | Must match on dEPG of packet |

The policy may require to match on additional fields such as L3 ports, TCP flags, labels, conditions, etc.

The actions performed on flow hit depend on the specified policy and are described in the next section.

Example of a flow in the policy enforcement table:

```
table=$POLICY_ENFORCER reg1=$SEPG, reg3=$DEPG, tcp_dst=DPORT/MASK,
actions=output:NXM_NX_REG4[ ]
```

A flow miss indicates that no policy has been specified or the policy has not been populated. Depending on whether the policy population is proactive or reactive, the action on flow miss is either drop or notification of the controller/agent to trigger policy resolution.

Policy Actions & Packet Rewrite

The policy may specify multiple actions which are to be performed on matching policy classifiers. The following actions are supported:

Accept

Forward/route the packet as previously selected in the dEPG selection table. This translates to executing the queued up action set and forwarding the packet to the port number stored in NXM_NX_REG4 which represents the L2 nexthop.

Basic example flow to allow an sEPG talk to a dEPG:

```
table=$POLICY_ENFORCER reg1=$SEPG, reg3=$DEPG,
actions=output:NXM_NX_REG4[ ]
```

Drop

Disregard any previous forwarding or routing decision and drop the packet:

```
table=$POLICY_ENFORCER reg1=$SEPG, reg3=$DEPG,  
actions=clear_actions, drop
```

Log

The logging action is an extension to the drop action. It will send packet to the controller for logging purposes. The controller will then drop the packet.

```
table=$POLICY_ENFORCER reg1=$SEPG, reg3=$DEPG,  
actions=clear_actions, controller:[...]
```

Set QoS

The **Set QoS** action allows to modify the QoS mark of a packet. This includes the DiffServ field as well as ECN information. Note that this action may only be applied to IP packets.

This action is typically followed by an allow or redirect action.

```
table=$POLICY_ENFORCER reg1=$SEPG, reg3=$DEPG,  
actions=mod_nw_tos:TOS, mod_nw_ecn:ECN, ...
```

Redirect / Service Redirection

Service insertion or redirection can be defined as an action between EPGs in the policy. It may occur transparently, i.e. without changing the packet in any way, or non-transparently by explicitly redirecting the packet to the service node.

Non-transparent Service Insertion

Non-transparent service insertion is used to redirect packets to a service such as a web proxy which requires the packet to be addressed to the service. The vSwitch forwarding behavior to achieve this is identical to a L2/L3 switching/routing action to any other EP.

The specific action chain will depend on whether the service is located within the same BD or whether routing is required. The controller/agent is aware of the location of both EPs and will insert the required action set. The following is an example for a L2 non-transparent service redirection:

```
table=$POLICY_ENFORCER reg1=$SEPG, reg3=$DEPG,  
actions=mod_dl_dst:$MAC_OF_SERVICE,  
        load:$TUNNEL_PORT->NXM_NX_REG4[ ],  
        move:NXM_NX_REG1[ ]->NXM_NX_TUN_ID[ ],  
        load:$TUNNEL_DST->NXM_NX_TUN_IPV4_DST[ ],  
        action:output:NXM_NX_REG4[ ]
```

Transparent Service Insertion

Transparent service insertion is used to redirect packets to a service such as a firewall which does not require a packet to be specifically addressed to the service. The service will be applied to all packets on the virtual network. This requires that the service only sees packets to which the service should be applied.

The required forwarding behavior is to encapsulate the packet with the appropriate VNID. There is no need to rewrite any of the L2 headers.

```
table=$POLICY_ENFORCER reg1=$SEPG, reg3=$DEPG,  
actions=load:$TUNNEL_PORT->NXM_NX_REG4[ ],
```

```
move:$VNI_OF_SERVICE->NXM_NX_TUN_ID[ ],
load:$TUNNEL_DST->NXM_NX_TUN_IPV4_DST[ ],
output:$NXM_NX_REG4[ ]
```

The redirect action in the policy will specify the VNID and VTEP to be used.

TBD: Does the pipeline always stop after a redirect action has been processed?

Mirror

This action causes the packet to be cloned and forwarded to an additional port (port mirroring).

OpenFlow/OVS Renderer

The OpenFlow renderer is based on the [OVS Overlay](#) design and implements a network virtualization solution for virtualized compute environments using Open vSwitch, OpenFlow and OVSDDB remotely from the controller.

The OpenFlow renderer architecture consists of the following:

| | |
|----------------------|---|
| Switch Manager | Manage connected switch configuration using OVSDDB. Maintain overlay tunnels. |
| Endpoint Manager | Optionally learn endpoints based on simple rules that map interfaces to endpoint groups. Can add additional rules in the future. Keep endpoint registry up to date. If disabled, then an orchestration system must program all endpoints and endpoint mappings. |
| ARP and DHCP Manager | Convert ARP and DHCP into unicast. |
| Policy Manager | Subscribe to renderer common infrastructure, and switch and endpoint manager. Manage the state of the flow tables in OVS. |

OpFlex Renderer

The OpFlex renderer is based on the [OVS Overlay](#) design and implements a network virtualization solution for virtualized compute environments by communicating with the OpFlex Agent.

The OpFlex renderer architecture consists of the following main pieces:

| | |
|------------------|--|
| Agent Manager | Manage connected agents using OpFlex. |
| RPC Library | Manage serialization/deserialization of JSON RPC with Policy Elements. |
| OpFlex Messaging | Provides definition of OpFlex messages and serialization/deserialization into Managed Objects. |
| Endpoint manager | Optionally learn endpoints based on simple rules that map interfaces to endpoint groups. Can add additional rules in the |

future. Keep endpoint registry up to date. If disabled, then an orchestration system must program all endpoints and endpoint mappings.

Policy manager

Subscribe to render common infrastructure and endpoint registry and provide normalized policy to agents.

9. L2Switch

Table of Contents

| | |
|--|-----|
| Checking out the L2Switch project | 189 |
| Testing your changes to the L2Switch project | 189 |
| Architecture of the L2Switch project | 190 |
| Developer's Guide for Packet Dispatcher | 191 |
| Developer's Guide for Loop Remover | 191 |
| Developer's Guide for Arp Handler | 193 |
| Developer's Guide for Address Tracker | 195 |
| Developer's Guide for Host Tracker | 197 |
| Developer's Guide for L2Switch Main | 197 |

The L2Switch project provides Layer2 switch functionality.

Checking out the L2Switch project

```
git clone https://git.opendaylight.org/gerrit/p/l2switch.git
```

The above command will create a directory called "l2switch" with the project.

Testing your changes to the L2Switch project

Running the L2Switch project

To run the base distribution, you can use the following command

```
./distribution/base/target/distributions-l2switch-base-0.1.0-SNAPSHOT-  
osgipackage/.opendaylight/run.sh
```

If you need additional resources, you can use these command line arguments:

```
-Xms1024m -Xmx2048m -XX:PermSize=512m -XX:MaxPermSize=1024m'
```

To run the karaf distribution, you can use the following command:

```
./distribution/karaf/target/assembly/bin/karaf
```

Create a network using mininet

```
sudo mn --controller=remote,ip=<Controller IP> --topo=linear,3 --switch  
ovsk,protocols=OpenFlow13  
sudo mn --controller=remote,ip=127.0.0.1 --topo=linear,3 --switch  
ovsk,protocols=OpenFlow13
```

The above command will create a virtual network consisting of 3 switches. Each switch will connect to the controller located at the specified IP, i.e. 127.0.0.1

```
sudo mn --controller=remote,ip=127.0.0.1 --mac --topo=linear,3 --switch  
ovsk,protocols=OpenFlow13
```

The above command has the "mac" option, which makes it easier to distinguish between Host MAC addresses and Switch MAC addresses.

Generating network traffic using mininet

```
h1 ping h2
```

The above command will cause host1 (h1) to ping host2 (h2)

```
pingall
```

pingall will cause each host to ping every other host.

Miscellaneous mininet commands

```
link s1 s2 down
```

This will bring the link between switch1 (s1) and switch2 (s2) down

```
link s1 s2 up
```

This will bring the link between switch1 (s1) and switch2 (s2) up

```
link s1 h1 down
```

This will bring the link between switch1 (s1) and host1 (h1) down

Architecture of the L2Switch project

- Packet Handler
 - Decodes the packets coming to the controller and dispatches them appropriately
- Loop Remover
 - Removes loops in the network
- Arp Handler
 - Handles the decoded ARP packets
- Address Tracker
 - Learns the Addresses (MAC and IP) of entities in the network
- Host Tracker
 - Tracks the locations of hosts in the network
- L2Switch Main
 - Installs flows on each switch based on network traffic

Developer's Guide for Packet Dispatcher Classes

- **AbstractPacketDecoder**
 - Defines the methods that all decoders must implement
- **EthernetDecoder**
 - The base decoder which decodes the packet into an Ethernet packet
- **ArpDecoder**, **Ipv4Decoder**, **Ipv6Decoder**
 - Decodes Ethernet packets into the either an ARP or IPv4 or IPv6 packet

Further development

There is a need for more decoders. A developer can write

- A decoder for another EtherType, i.e. LLDP.
- A higher layer decoder for the body of the IPv4 packet or IPv6 packet, i.e. TCP and UDP.

How to write a new decoder

- extends **AbstractDecoder<A, B>**
 - A refers to the notification that the new decoder consumes
 - B refers to the notification that the new decoder produces
- implements **xPacketListener**
 - The new decoder must specify which notification it is listening to
- **canDecode** method
 - This method should examine the consumed notification to see whether the new decoder can decode the contents of the packet
- **decode** method
 - This method does the actual decoding of the packet

Developer's Guide for Loop Remover Classes

- **LoopRemoverModule**
 - Reads config subsystem value for *is-install-lldp-flow*

- If *is-install-lldp-flow* is true, then an **InitialFlowWriter** is created
- Creates and initializes the other LoopRemover classes
- **InitialFlowWriter**
 - Only created when *is-install-lldp-flow* is true
 - Installs a flow, which forwards all LLDP packets to the controller, on each switch
- **TopologyLinkDataChangeHandler**
 - Listens to data change events on the Topology tree
 - When these changes occur, it waits *graph-refresh-delay* seconds and then tells **NetworkGraphImpl** to update
 - Writes an STP (Spanning Tree Protocol) status of "forwarding" or "discarding" to each link in the Topology data tree
 - Forwarding links can forward packets.
 - Discarding links cannot forward packets.
- **NetworkGraphImpl**
 - Creates a loop-free graph of the network

Configuration

- *graph-refresh-delay*
 - Used in TopologyLinkDataChangeHandler
 - A higher value has the advantage of doing less graph updates, at the potential cost of losing some packets because the graph didn't update immediately.
 - A lower value has the advantage of handling network topology changes quicker, at the cost of doing more computation.
- *is-install-lldp-flow*
 - Used in LoopRemoverModule
 - "true" means a flow that sends all LLDP packets to the controller will be installed on each switch
 - "false" means this flow will not be installed
- *lldp-flow-table-id*
 - The LLDP flow will be installed on the specified flow table of each switch
- *lldp-flow-priority*

- The LLDP flow will be installed with the specified priority
- `lldp-flow-idle-timeout`
 - The LLDP flow will timeout (removed from the switch) if the flow doesn't forward a packet for *x* seconds
- `lldp-flow-hard-timeout`
 - The LLDP flow will timeout (removed from the switch) after *x* seconds, regardless of how many packets it is forwarding

Further development

No suggestions at the moment.

Validating changes to Loop Remover

STP Status information is added to the Inventory data tree.

- A status of "forwarding" means the link is active and packets are flowing on it.
- A status of "discarding" means the link is inactive and packets are not sent over it.

The STP status of a link can be checked through a browser or a REST Client.

```
http://10.194.126.91:8080/restconf/operational/.opendaylight-inventory:nodes/  
node/openflow:1/node-connector/openflow:1:2
```

The STP status should still be there after changes are made.

Developer's Guide for Arp Handler

Classes

- **ArpHandlerModule**
 - Reads config subsystem value for *is-proactive-flood-mode*
 - If *is-proactive-flood-mode* is true, then a `ProactiveFloodFlowWriter` is created
 - If *is-proactive-flood-mode* is false, then an `InitialFlowWriter` is created
- **ProactiveFloodFlowWriter**
 - Only created when *is-proactive-flood-mode* is true
 - Installs a flood flow on each switch. With this flood flow, a packet that doesn't match any other flows will be flooded/broadcast from that switch.
- **InitialFlowWriter**

- Only created when *is-proactive-flood-mode* is false
- Installs a flow, which sends all ARP packets to the controller, on each switch
- **ArpPacketHandler**
 - Only created when *is-proactive-flood-mode* is false
 - Handles and processes the controller's incoming ARP packets
 - Uses **PacketDispatcher** to send the ARP packet back into the network
- **PacketDispatcher**
 - Only created when *is-proactive-flood-mode* is false
 - Sends packets out to the network
 - Uses **InventoryReader** to determine which node-connector to send a packet on
- **InventoryReader**
 - Only created when *is-proactive-flood-mode* is false
 - Maintains a list of each switch's node-connectors

Configuration

- *is-proactive-flood-mode*
 - "true" means that flood flows will be installed on each switch. With this flood flow, each switch will flood a packet that doesn't match any other flows.
 - Advantage: Fewer packets are sent to the controller because those packets are flooded to the network.
 - Disadvantage: A lot of network traffic is generated.
 - "false" means the previously mentioned flood flows will not be installed. Instead an ARP flow will be installed on each switch that sends all ARP packets to the controller.
 - Advantage: Less network traffic is generated.
 - Disadvantage: The controller handles more packets (ARP requests & replies) and the ARP process takes longer than if there were flood flows.
- *flood-flow-table-id*
 - The flood flow will be installed on the specified flow table of each switch
- *flood-flow-priority*
 - The flood flow will be installed with the specified priority
- *flood-flow-idle-timeout*

- The flood flow will timeout (removed from the switch) if the flow doesn't forward a packet for *x* seconds
- flood-flow-hard-timeout
 - The flood flow will timeout (removed from the switch) after *x* seconds, regardless of how many packets it is forwarding
- arp-flow-table-id
 - The ARP flow will be installed on the specified flow table of each switch
- arp-flow-priority
 - The ARP flow will be installed with the specified priority
- arp-flow-idle-timeout
 - The ARP flow will timeout (removed from the switch) if the flow doesn't forward a packet for *x* seconds
- arp-flow-hard-timeout
 - The ARP flow will timeout (removed from the switch) after *arp-flow-hard-timeout* seconds, regardless of how many packets it is forwarding

Further development

The **ProactiveFloodFlowWriter** needs to be improved. It does have the advantage of having less traffic come to the controller; however, it generates too much network traffic.

Developer's Guide for Address Tracker

Classes

- AddressTrackerModule
 - Reads config subsystem value for *observe-addresses-from*
 - If *observe-addresses-from* contains "arp", then an AddressObserverUsingArp is created
 - If *observe-addresses-from* contains "ipv4", then an AddressObserverUsingIpv4 is created
 - If *observe-addresses-from* contains "ipv6", then an AddressObserverUsingIpv6 is created
- AddressObserverUsingArp
 - Registers for ARP packet notifications
 - Uses **AddressObservationWriter** to write address observations from ARP packets
- AddressObserverUsingIpv4

- Registers for IPv4 packet notifications
- Uses **AddressObservationWriter** to write address observations from IPv4 packets
- AddressObserverUsingIpv6
 - Registers for IPv6 packet notifications
 - Uses **AddressObservationWriter** to write address observations from IPv6 packets
- AddressObservationWriter
 - Writes new Address Observations to the Inventory data tree
 - Updates existing Address Observations with updated "last seen" timestamps
 - Uses the *timestamp-update-interval* configuration variable to determine whether or not to update

Configuration

- timestamp-update-interval
 - A last-seen timestamp is associated with each address. This last-seen timestamp will only be updated after *timestamp-update-interval* milliseconds.
 - A higher value has the advantage of performing less writes to the database.
 - A lower value has the advantage of knowing how fresh an address is.
- observe-addresses-from
 - IP and MAC addresses can be observed/learned from ARP, IPv4, and IPv6 packets. Set which packets to make these observations from.

Further development

Further improvements can be made to the **AddressObservationWriter** so that it (1) doesn't make any unnecessary writes to the DB and (2) is optimized for multi-threaded environments.

Validating changes to Address Tracker

Address Observations are added to the Inventory data tree.

The Address Observations on a Node Connector can be checked through a browser or a REST Client.

```
http://10.194.126.91:8080/restconf/operational/.opendaylight-inventory:nodes/  
node/openflow:1/node-connector/openflow:1:1
```

The Address Observations should still be there after changes.

Developer's Guide for Host Tracker

Validating changes to Host Tracker

Host information is added to the Topology data tree.

- Host address
- Attachment point (link) to a node/switch

This host information and attachment point information can be checked through a browser or a REST Client.

```
http://10.194.126.91:8080/restconf/operational/network-topology:network-topology/topology/flow:1/
```

Host information should still be there after changes.

Developer's Guide for L2Switch Main

Classes

- L2SwitchMainModule
 - Reads config subsystem value for *is-install-dropall-flow*
 - If *is-install-dropall-flow* is true, then an **InitialFlowWriter** is created
 - Reads config subsystem value for *is-learning-only-mode*
 - If *is-learning-only-mode* is false, then a **ReactiveFlowWriter** is created
- InitialFlowWriter
 - Only created when *is-install-dropall-flow* is true
 - Installs a flow, which drops all packets, on each switch. This flow has low priority and means that packets that don't match any higher-priority flows will simply be dropped.
- ReactiveFlowWriter
 - Reacts to network traffic and installs MAC-to-MAC flows on switches. These flows have matches based on MAC source and MAC destination.
 - Uses **FlowWriterServiceImpl** to write these flows to the switches
- FlowWriterService / FlowWriterServiceImpl
 - Writes flows to switches

Configuration

- is-install-dropall-flow

- "true" means a drop-all flow will be installed on each switch, so the default action will be to drop a packet instead of sending it to the controller
- "false" means this flow will not be installed
- dropall-flow-table-id
 - The dropall flow will be installed on the specified flow table of each switch
 - This field is only relevant when "is-install-dropall-flow" is set to "true"
- dropall-flow-priority
 - The dropall flow will be installed with the specified priority
 - This field is only relevant when "is-install-dropall-flow" is set to "true"
- dropall-flow-idle-timeout
 - The dropall flow will timeout (removed from the switch) if the flow doesn't forward a packet for x seconds
 - This field is only relevant when "is-install-dropall-flow" is set to "true"
- dropall-flow-hard-timeout
 - The dropall flow will timeout (removed from the switch) after x seconds, regardless of how many packets it is forwarding
 - This field is only relevant when "is-install-dropall-flow" is set to "true"
- is-learning-only-mode
 - "true" means that the L2Switch will only be learning addresses. No additional flows to optimize network traffic will be installed.
 - "false" means that the L2Switch will react to network traffic and install flows on the switches to optimize traffic. Currently, MAC-to-MAC flows are installed.
- reactive-flow-table-id
 - The reactive flow will be installed on the specified flow table of each switch
 - This field is only relevant when "is-learning-only-mode" is set to "false"
- reactive-flow-priority
 - The reactive flow will be installed with the specified priority
 - This field is only relevant when "is-learning-only-mode" is set to "false"
- reactive-flow-idle-timeout
 - The reactive flow will timeout (removed from the switch) if the flow doesn't forward a packet for x seconds

- This field is only relevant when "is-learning-only-mode" is set to "false"
- reactive-flow-hard-timeout
 - The reactive flow will timeout (removed from the switch) after x seconds, regardless of how many packets it is forwarding
 - This field is only relevant when "is-learning-only-mode" is set to "false"

Further development

The **ReactiveFlowWriter** needs to be improved to install the MAC-to-MAC flows faster. For the first ping, the ARP request and reply are successful. However, then the ping packets are sent out. The first ping packet is dropped sometimes because the MAC-to-MAC flow isn't installed quickly enough. The second, third, and following ping packets are successful though.

10. Lisp Flow Mapping

Table of Contents

| | |
|--|-----|
| OpenDaylight Locator/ID Separation Protocol (LISP) Flow Mapping Overview | 200 |
| LISP Flow Mapping Service | 201 |
| LISP Service Architecture | 201 |
| LISP APIs | 203 |
| LISP Configuration Options | 203 |
| Developer Tutorial | 203 |
| LISP Support | 210 |
| Installing LISP Flow Mapping | 210 |

OpenDaylight Locator/ID Separation Protocol (LISP) Flow Mapping Overview

[Locator/ID Separation Protocol \(LISP\)](#) is a technology that provides a flexible map-and-encap framework that can be used for overlay network applications such as data center network virtualization and Network Function Virtualization (NFV).

LISP provides the following name spaces:

- [Endpoint Identifiers \(EIDs\)](#)
- [Routing Locators \(RLOCs\)](#)

In a virtualization environment EIDs can be viewed as virtual address space and RLOCs can be viewed as physical network address space.

The LISP framework decouples network control plane from the forwarding plane by providing:

- A data plane that specifies how the virtualized network addresses are encapsulated in addresses from the underlying physical network.
- A control plane that stores the mapping of the virtual-to-physical address spaces and the associated forwarding policies and serves this information to the data plane on demand.

Network programmability is achieved by programming forwarding policies such as transparent mobility, service chaining, and traffic engineering in the mapping system; where the data plane elements can fetch these policies on demand as new flows arrive. This chapter describes the LISP Flow Mapping project in OpenDaylight and how it can be used to enable advanced SDN and NFV use cases.

LISP data plane Tunnel Routers are available at LISPmob.org in the open source community on the following platforms:

- Linux

- Android
- OpenWRT

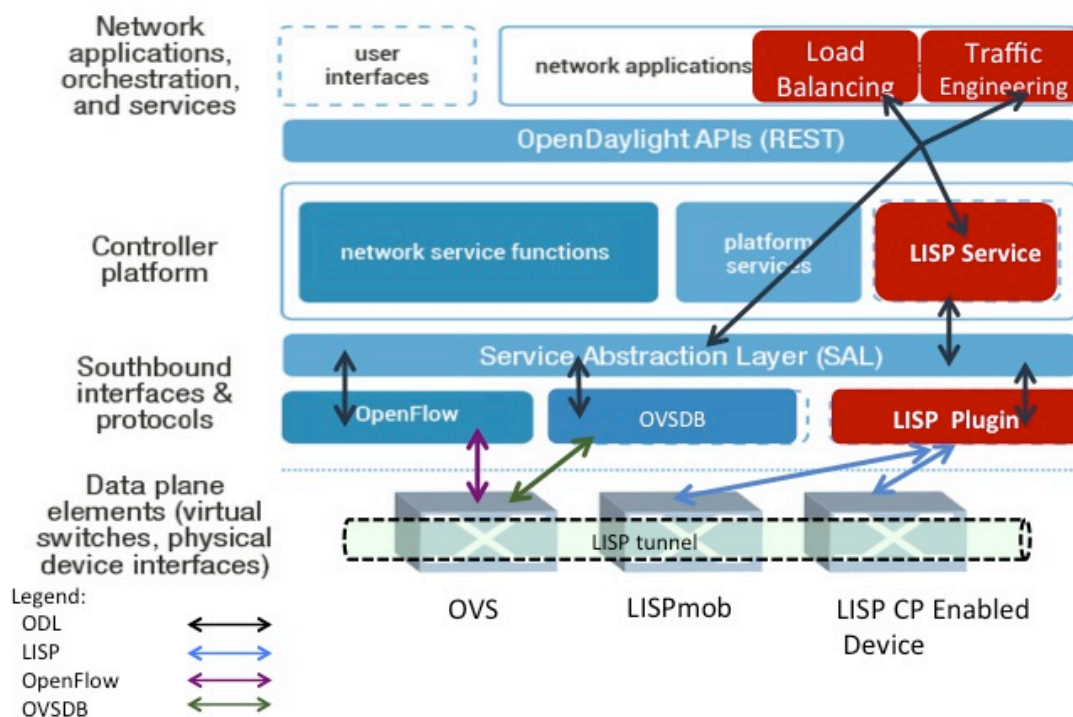
For more details and support for LISP data plane software please visit [the LISPmob web site](#).

LISP Flow Mapping Service

The LISP Flow Mapping service provides LISP Mapping System services. This includes LISP Map-Server and LISP Map-Resolver services to store and serve mapping data to data plane nodes as well as to OpenDaylight applications. Mapping data can include mapping of virtual addresses to physical network address where the virtual nodes are reachable or hosted at. Mapping data can also include a variety of routing policies including traffic engineering and load balancing. To leverage this service, OpenDaylight applications and services can use the northbound REST API to define the mappings and policies in the LISP Mapping Service. Data plane devices capable of LISP control protocol can leverage this service through a southbound LISP plugin via the LISP control protocol (Map-Register, Map-Request, Map-Reply messages).

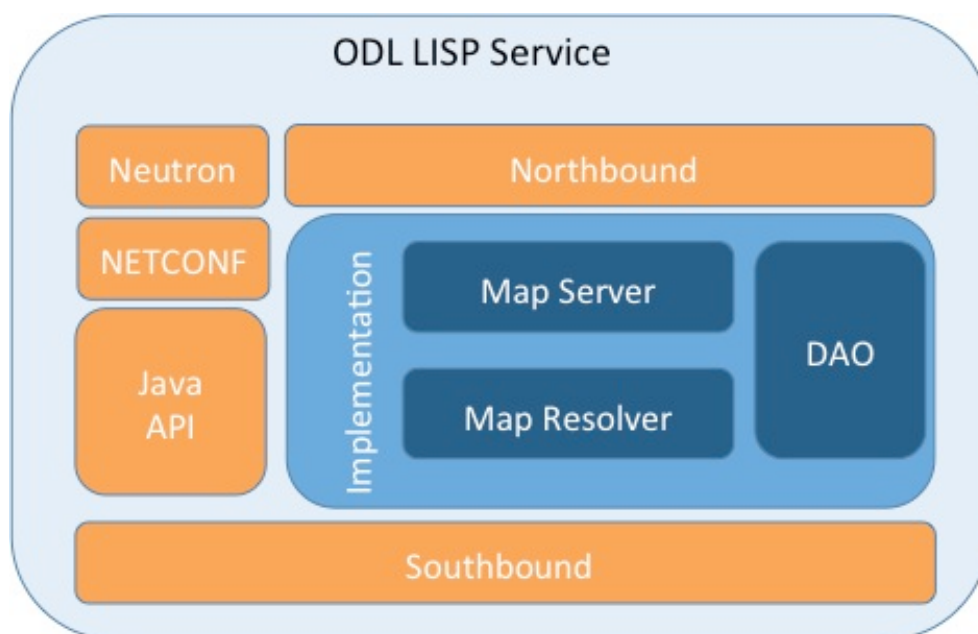
The following figure depicts the described components:

Figure 10.1. Architecture Overview



LISP Service Architecture

The following figure shows the various LISP Flow Mapping modules.

Figure 10.2. LISP Mapping Service Internal Architecture

A brief description of each module is as follows:

- **DAO:** This layer separates the LISP logic from the database, so that we can separate the map server and map resolver from the specific implementation of the DHT (Distributed Hash Table). Currently we have an implementation of this layer with the controller cluster service as a DHT, but it can be switched to any other DHT and you only need to implement the ILISPDAO interface.
- **Map Server:** This module processes the adding or registration of keys and mappings. For a detailed specification of LISP Map Server, see [LISP](#).
- **Map Resolver:** This module receives and processes the mapping lookup queries and provides the mappings to requester. For a detailed specification of LISP Map Server, see [LISP](#).
- **Northbound API:** This is part of the ODL northbound API. This module enables defining key-EID associations as well as adding mapping information through the Map Server. Key-EID associations can also be queried via this API. The Northbound API also provides capability of querying the mapping information for an EID prefix.
- **Neutron:** This module implements the ODL Neutron Service APIs. It provides integration between the LISP service and the ODL Neutron service.
- **NETCONF:** This module enables the LISP service to communicate to NETCONF-enabled devices through ODL's NETCONF plugin.
- **Java API:** The API module exposes the Map Server and Map Resolver capabilities via Java API.
- **LISP Southbound Plugin:** This plugin enables data plane devices that support LISP control plane protocol (see LISP) to register and query mappings to the LISP Flow Mapping via the LISP control plane protocol.

LISP APIs

The LISP Flow Mapping service has JAVA APIs and REST APIs. The Java API reference documentation is auto-generated from the Java build and is available at:

- [JAVA APIs](#)

Below you will find the detailed information about the module's REST resources and their verbs (description, URI, parameters, responses, and status codes), schemas, example XML, example JSON, as well as programming examples.

- [REST APIS](#)

LISP Configuration Options

The `etc/custom.properties` file in the Karaf distribution allows configuration of several OpenDaylight parameters. The LISP service has two properties that can be adjusted: `lisp.mappingOverwrite` and `lisp.smr`.

`lisp.mappingOverwrite` (default: *true*) Configures handling of mapping updates. When set to *true* (default) a mapping update (either through the southbound plugin via a Map-Register message or through a northbound API PUT REST call) the existing RLOC set associated to an EID prefix is overwritten. When set to *false*, the RLOCs of the update are merged to the existing set.

`lisp.smr` (default: *false*) Enables/disables the [Solicit-Map-Request \(SMR\)](#) functionality. SMR is a method to notify changes in an EID-to-RLOC mapping to "subscribers". The LISP service considers all Map-Request's source RLOC as a subscriber to the requested EID prefix, and will send an SMR control message to that RLOC if the mapping changes.

Developer Tutorial

This section provides instructions to set up a LISP network of three nodes (one "client" node and two "server" nodes) using LISPmob and Open vSwitch (OVS) as data plane LISP nodes and the LISP Flow Mapping project from ODL as the LISP programmable mapping system for the LISP network. The steps shown below will demonstrate performing a failover between the two "server" nodes. The three LISP data plane nodes and the LISP mapping system are assumed to be running in Linux virtual machines using the following IPv4 addresses on their eth0 interfaces (please adjust configuration files, JSON examples, etc. accordingly if you're using another addressing scheme):

Table 10.1. Nodes in the tutorial

| Node | Node Type | IP Address |
|------------|--------------|-------------|
| controller | OpenDaylight | 10.33.12.32 |
| client | LISPmob | 10.33.12.35 |
| server1 | LISPmob | 10.33.12.37 |

| Node | Node Type | IP Address |
|---------|--------------|-------------|
| server2 | Open vSwitch | 10.33.12.44 |



Note

While the tutorial uses LISPmob and OVS as the data plane, they could be any LISP-enabled HW or SW router (commercial/open source).

The below steps are using the command line tool cURL to talk to the LISP Flow Mapping northbound REST API. This is so that you can see the actual request URLs and body content on the page.



Note

It is more convenient to use the Postman Chrome browser plugin to edit and send the requests. The project git repository hosts a collection of the requests that are used in this tutorial in the `resources/tutorial/ODL_Summit_LISP_Demo.json` file. You can import this file to Postman by following *Collections#Import a collection#Import from URL* and then entering the following link: https://git.opendaylight.org/gerrit/gitweb?p=lispflowmapping.git;a=blob_plain;f=resources/tutorial/ODL_Summit_LISP_Demo.json;hb=refs/heads/develop. Alternatively, you can save the file on your machine, or if you have the repository checked out, you can import from there. You will need to define some variables to point to your OpenDaylight controller instance.



Note

It is assumed that commands are executed as the *root* user.



Note

To set up a basic LISP network overlay (no fail-over) without dealing with OVS, you can skip steps 7 and 8 and just use LISPmob as your dataplane. If you do want to test fail-over, but not using OVS, skip steps 7 and 8, but set up LISPmob on **server2** as well, with identical configuration.

1. Install and run OpenDaylight Helium release on the controller VM. Please follow the general OpenDaylight Helium Installation Guide for this step. Once the OpenDaylight controller is running install the *odl-openflowplugin-all*, *odl-adsal-compatibility-all*, *odl-ovsdb-all*, and *odl-lispflowmapping-all* features from the CLI:

```
feature:install odl-openflowplugin-all odl-adsal-compatibility-all odl-ovsdb-all odl-lispflowmapping-all
```



Note

If you're not planning on using OVS you can skip the first three and install *odl-lispflowmapping-all* only.

It takes quite a while to load and initialize all features and their dependencies. It's worth running the command `log:tail` in the Karaf console to see when is the log output winding down, and continue after that.

2. Install LISPmob on the **client** and **server1** VMs following the installation instructions from the [LISPmob README file](#).
3. Configure the LISPmob installations from the previous step. Starting from the `lispd.conf.example` file in the distribution, set the EID in each `lispd.conf` file from the IP address space selected for your virtual/LISP network. In this tutorial the EID of the **client** is set to 1.1.1.1/32, and that of **server1** to 2.2.2.2/32. Set the RLOC interface in each `lispd.conf`. LISP will determine the RLOC (IP address of the corresponding VM) based on this interface. Set the Map-Resolver address to the IP address of the **controller**, and on the **client** the Map-Server too. On **server1** set the Map-Server to something else, so that it doesn't interfere with the mappings on the controller, since we're going to program them manually. Modify the "key" parameter in each `lispd.conf` file to a key/password of your choice, *asdf* in this tutorial. The `resources/tutorial` directory in the *develop* branch of the project git repository has the files used in the tutorial checked in: [lispd.conf.client](#) and [lispd.conf.server1](#). Copy the files to `/root/lispd.conf` on the respective VMs.
4. Define a key and EID prefix association in ODL using the northbound API for both EIDs (1.1.1.1/32 and 2.2.2.2/32). Run the below commands on the **controller** (or any machine that can reach **controller**, by replacing *localhost* with the IP address of **controller**).

```
curl -u "admin":"admin" -H "Content-type: application/json" -X PUT \
  http://localhost:8080/lispflowmapping/nb/v2/default/key \
  --data @key1.json
curl -u "admin":"admin" -H "Content-type: application/json" -X PUT \
  http://localhost:8080/lispflowmapping/nb/v2/default/key \
  --data @key2.json
```

where the content of the *key1.json* and *key2.json* files is the following (with different "ipAddress"):

```
{
  "key" : "asdf",
  "maskLength" : 32,
  "address" :
  {
    "ipAddress" : "1.1.1.1",
    "afi" : 1
  }
}
```

5. Verify that the key is added properly by requesting the following URL:

```
curl -u "admin":"admin" http://localhost:8080/lispflowmapping/nb/v2/default/
key/0/1/1.1.1.1/32
curl -u "admin":"admin" http://localhost:8080/lispflowmapping/nb/v2/default/
key/0/1/2.2.2.2/32
```

6. Run the lispd LISPmob daemon on the **client** and **server1** VMs:

```
lispd -f /root/lispd.conf
```

7. Prepare the OVS environment on **server2**:

- a. Start the `ovsdb-server` and `ovs-vswitchd` daemons (or check that your distribution's init scripts already started them)

- b. Start listening for OVSDB manager connections on the standard 6640 TCP port:

```
ovs-vsctl set-manager "ptcp:6640"
ovs-vsctl show
```

- c. Create a TAP port for communications with the guest VM. We'll have another VM inside the **server2** VM, that will be set up with the 2.2.2.2/24 EID. It also needs a fictitious gateway, and a static ARP entry for that gateway, with any MAC address.

```
tunctl -t tap0
ifconfig tap0 up
```

- d. Start the guest VM:

```
modprobe kvm
kvm -daemonize -vnc :0 -m 128 -net nic,macaddr=00:00:0C:15:C0:A1 \
    -net tap,ifname=tap0,script=no,downscript=no \
    -drive file=ubuntu.12-04.x86-64.20120425.static_ip_2.2.2.2.qcow2
```

8. Set up the OVS environment on **server2** using the ODL northbound API

- a. Connect to the OVSDB management port from ODL:

```
curl -u "admin":"admin" -X PUT \
    http://localhost:8080/controller/nb/v2/connectionmanager/node/
server2/address/10.33.12.44/port/6640
```

You can check if this and the next requests have the desired effect on OVS by running the following on **server2**

```
ovs-vsctl show
```

It should now show the "Manager" connection as connected

- b. Create the bridge br0:

```
curl -u "admin":"admin" -H "Content-type: application/json" -X POST \
    http://localhost:8080/controller/nb/v2/networkconfig/bridgedomain/
bridge/OVS/server2/br0 -d "{}"
```

- c. Add tap0 to br0:

```
curl -u "admin":"admin" -H "Content-type: application/json" -X POST \
    http://localhost:8080/controller/nb/v2/networkconfig/bridgedomain/
port/OVS/server2/br0/tap0 -d "{}"
```

- d. Add the lisp0 LISP tunneling virtual port to br0:

```
curl -u "admin":"admin" -H "Content-type: application/json" -X POST \
    http://localhost:8080/controller/nb/v2/networkconfig/bridgedomain/
port/OVS/server2/br0/lisp0 -d @lisp0.json
```

where *lisp0.json* has the following content:

```
{
  "type": "tunnel",
  "tunnel_type": "lisp",
  "dest_ip": "10.33.12.35"
}
```

The **dest_ip** parameter sets the tunnel destination to the **client** VM. This has to be done manually (from the controller), since OVS doesn't have a LISP control plane to fetch mappings.

- e. We will now need to set up flows on **br0** to steer traffic received on the LISP virtual port in OVS to the VM connected to **tap0** and vice-versa. For that we will need the node id of the bridge, which is based on its MAC address, which is generated at creation time. So we look at the list of connections on the controller:

```
curl -u "admin":"admin" http://localhost:8080/controller/nb/v2/
connectionmanager/nodes
```

The response should look similar to this:

```
{ "id": "00:00:62:71:36:30:7b:44", "type": "OF" } ],
{ "id": "10.33.12.35", "type": "LISP" }, { "id": "server2", "type": "OVS" } ] }
```

There are three types of nodes connected to ODL: one "OF" node (the OpenFlow connection to **br0** on **server2**), one "LISP" node (the **client** VM sending LISP Map-Register control messages to the controller which is acting as a LISP Map-Server), and one "OVS" node (this is the OVSDDB connection to **server2**). We will need the id of the "OF" node in order to set up flows.

- f. The first flow will decapsulate traffic received from the client VM on **server2** and send it to the guest VM through the **tap0** port.

```
curl -u "admin":"admin" -H "Content-type: application/json" -X PUT \
  http://localhost:8080/controller/nb/v2/flowprogrammer/default/node/
OF/00:00:62:71:36:30:7b:44/staticFlow/Decap -d @flow_decap.json
```

Make sure that the bridge id after the OF path component of the URL is the id from the previous step. It should also be the same on line 6 in *flow_decap.json* file (see below), which should have the MAC address of the KVM instance started on **server2** on line 11 (**SET_DL_DST**):

```
{
  "installInHw": "true",
  "name": "Decap",
  "node": {
    "type": "OF",
    "id": "00:00:62:71:36:30:7b:44"
  },
  "priority": "10",
  "dlDst": "02:00:00:00:00:00",
  "actions": [
    "SET_DL_DST=00:00:0c:15:c0:a1",
    "OUTPUT=1"
  ]
}
```


- g. The second flow will encapsulate traffic received from the guest VM on **server2** through the **tap0** port.

```
curl -u "admin":"admin" -H "Content-type: application/json" -X PUT \
  http://localhost:8080/controller/nb/v2/flowprogrammer/default/node/
  OF/00:00:62:71:36:30:7b:44/staticFlow/Encap -d @flow_encap.json
```

The *flow_encap.json* file should look like this:

```
{
  "installInHw": "true",
  "name": "Decap",
  "node": {
    "type": "OF",
    "id": "00:00:62:71:36:30:7b:44"
  },
  "priority": "5",
  "ingressPort": "1",
  "etherType": "0x0800",
  "vlanId": "0",
  "nwDst": "1.1.1.1/32",
  "actions": [
    "OUTPUT=2"
  ]
}
```

- h. Check if the flows have been created correctly. First, in ODL

```
curl -u "admin":"admin" http://localhost:8080/controller/nb/v2/
flowprogrammer/default
```

And most importantly, on **server2**

```
ovs-ofctl dump-flows br0 -O OpenFlow13
```

9. The **client** LISPmob node should now register its EID-to-RLOC mapping in ODL. To verify you can lookup the corresponding EIDs via the northbound API

```
curl -u "admin":"admin" http://localhost:8080/lispflowmapping/nb/v2/default/
mapping/0/1/1.1.1.1/32
```

10. Register the EID-to-RLOC mapping of the server EID 2.2.2.2/32 to the controller, pointing to **server1** and **server2** with a higher priority for **server1**

```
curl -u "admin":"admin" -H "Content-type: application/json" -X PUT \
  http://localhost:8080/lispflowmapping/nb/v2/default/mapping \
  -d @mapping.json
```

where the *mapping.json* file looks like this

```
{
  "key" : "asdf",
  "mapregister" :
  {
    "proxyMapReply" : true,
    "eidToLocatorRecords" :
    [
      {
        "authoritative" : true,

```

```
"prefixGeneric" :
{
  "ipAddress" : "2.2.2.2",
  "afi" : 1
},
"mapVersion" : 0,
"maskLength" : 32,
"action" : "NoAction",
"locators" :
[
  {
    "multicastPriority" : 1,
    "locatorGeneric" :
    {
      "ipAddress" : "10.33.12.37",
      "afi" : 1
    },
    "routed" : true,
    "multicastWeight" : 0,
    "rlocProbed" : false,
    "localLocator" : false,
    "priority" : 126,
    "weight" : 1
  },
  {
    "multicastPriority" : 1,
    "locatorGeneric" :
    {
      "ipAddress" : "10.33.12.44",
      "afi" : 1
    },
    "routed" : true,
    "multicastWeight" : 0,
    "rlocProbed" : false,
    "localLocator" : false,
    "priority" : 127,
    "weight" : 1
  }
],
"recordTtl" : 5
}
],
"keyId" : 0
}
```

Here the priority of the second RLOC (10.33.12.44 - **server2**) is 127, a higher numeric value than the priority of 10.33.12.37, which is 126. This policy is saying that **server1** is preferred to **server2** for reaching EID 2.2.2.2/32. Note that lower priority has higher preference in LISP.

11. Verify the correct registration of the 2.2.2.2/32 EID:

```
curl -u "admin":"admin" http://localhost:8080/lispflowmapping/nb/v2/default/
mapping/0/1/2.2.2.2/32
```

12. Now the LISP network is up. To verify, log into the **client** VM and ping the server EID:

```
ping 2.2.2.2
```

13. Let's test fail-over now. Suppose you had a service on **server1** which became unavailable, but **server1** itself is still reachable. LISP will not automatically fail over, even if the mapping for 2.2.2.2/32 has two locators, since both locators are still reachable and uses the one with the higher priority (lowest priority value). To force a failover, we need to set the priority of **server2** to a lower value. Using the file mapping.json above, swap the priority values between the two locators and repeat the request from step 10. You can also repeat step 11 to see if the mapping is correctly registered. Note that the previous locators are still present, so you should see a list of four locators. If you leave the ping on, and monitor the traffic using Wireshark you can see that the ping traffic will be diverted from **server1** to **server2**.

With the default ODL configuration this may take some time, because the mapping stays in the **client** map-cache until the TTL expires. LISP has a [Solicit-Map-Request \(SMR\) mechanism](#) that can ask a LISP data plane element to update its mapping for a certain EID. This is disabled by default, and is controlled by the `lisp.smr` variable in `etc/custom.properties`. When enabled, any mapping change from the northbound will trigger an SMR packet to all data plane elements that have requested the mapping in a certain time window.

If you used the Postman collection, you will notice an "ELP" mapping. This is for supporting service chaining, but it requires a Re-encapsulating Tunnel Router (RTR). Support for RTR functionality in LISPmob is in progress, and we will update the tutorial to demonstrate service chaining when it becomes available.

LISP Support

For support please contact the lispflowmapping project at:

- LISP Flow Mapping users mailing list: lispflowmapping-users@lists.opendaylight.org
- LISP Flow Mapping dev mailing list: lispflowmapping-dev@lists.opendaylight.org

You can also reach us at the following channel on IRC:

- #opendaylight-lispflowmapping on irc.freenode.net

Additional information is also available on the wiki:

- [LISP Flow Mapping wiki](#)

Installing LISP Flow Mapping

This chapter contains installation instructions for Locator ID Separation Protocol (LISP) and provides guidelines for installation from the lispflowmapping repository.

Setting up Gerrit

Code reviews are enabled through Gerrit. For setting up Gerrit, see [Set up Gerrit](#). >>>>>>>
a8dd6f1... added installation section into the lisp doc



Note

You will need to perform the Gerrit Setup before you can access git via ssh as described below.

Pulling code via Git CLI

Pull the code by cloning the LispFlowMapping repository.

```
git clone ssh://<username>@git.opendaylight.org:29418/lispflowmapping.git
```

or if you just want to do an anonymous git clone, you can use:

```
git clone https://git.opendaylight.org/gerrit/p/lispflowmapping.git
```

Setting up Gerrit Change-id Commit Message Hook

This command inserts a unique Change-Id tag in the footer of a commit message. This step is optional but highly recommended for tracking changes.

```
cd lispflowmapping
scp -p -P 29418 <username>@git.opendaylight.org:hooks/commit-msg .git/hooks/
chmod 755 .git/hooks/commit-msg
```

Install and setup gitreview. The instructions can be found at [here](#).

Hacking the Code

The following tasks are used to help you hack the code.

Setup Eclipse

1. Run Eclipse (Kepler is the current version).
2. Open Git Repository perspective.
3. Add an existing repository and choose the Lisp Flow Mapping repository that was pulled earlier.
4. Import existing Maven projects and choose the following under the lispflowmapping directory:
 - api/pom.xml
 - implementation/pom.xml

Build the code

```
mvn clean install
```

To run without unitests you can skip building those tests running the following:

```
mvn clean install -DskipTests
```

```
/* instead of "mvn clean install" */
```

Run the controller

```
cd distribution-karaf/target/assembly/bin  
./karaf
```

At this point the ODL controller is running. Open a web browser and point your browser at <http://localhost:8080/>

For complete documentation on running the controller, see the ODL Helium Installation Guide.

Commit the code using Git CLI



Note

To be accepted, all code must come with a [developer certificate of origin](#) as expressed by having a Signed-off-by. This means that you are asserting that you have made the change and you understand that the work was done as part of an open-source license.

Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

- (a) The contribution was created in whole or in part by me and I have the right to submit it under the open source license indicated in the file; or
- (b) The contribution is based upon previous work that, to the best of my knowledge, is covered under an appropriate open source license and I have the right under that license to submit that work with modifications, whether created in whole or in part by me, under the same open source license (unless I am permitted to submit under a different license), as indicated in the file; or
- (c) The contribution was provided directly to me by some other person who certified (a), (b) or (c) and I have not modified it.
- (d) I understand and agree that this project and the contribution are public and that a record of the contribution (including all personal information I submit with it, including my sign-off) is maintained indefinitely and may be redistributed consistent with this project or the open source license(s) involved.

Mechanically you do it this way:

```
git commit --signoff
```

You will be prompted for a commit message. If you are fixing a bugzilla bug you can add the associated bug number to your commit message and it will get linked from Gerrit:

For Example:

```
Fix for bug 2.

Signed-off-by: Ed Warnicke <eaw@cisco.com>
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch develop
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README
#
```

Pushing the Code via Git CLI

Use gitreview to push your changes back to the remote repository using:

```
git review
```

You can set a topic for your patch by:

```
git review -t <topic>
```

The Jenkins Controller User will verify your code.

Pulling the Code changes via Git CLI

Use git pull to get the latest changes from the remote repository

```
git pull origin HEAD:refs/for/develop
```

Pushing the Code via Git CLI

Use git push to push your changes back to the remote repository.

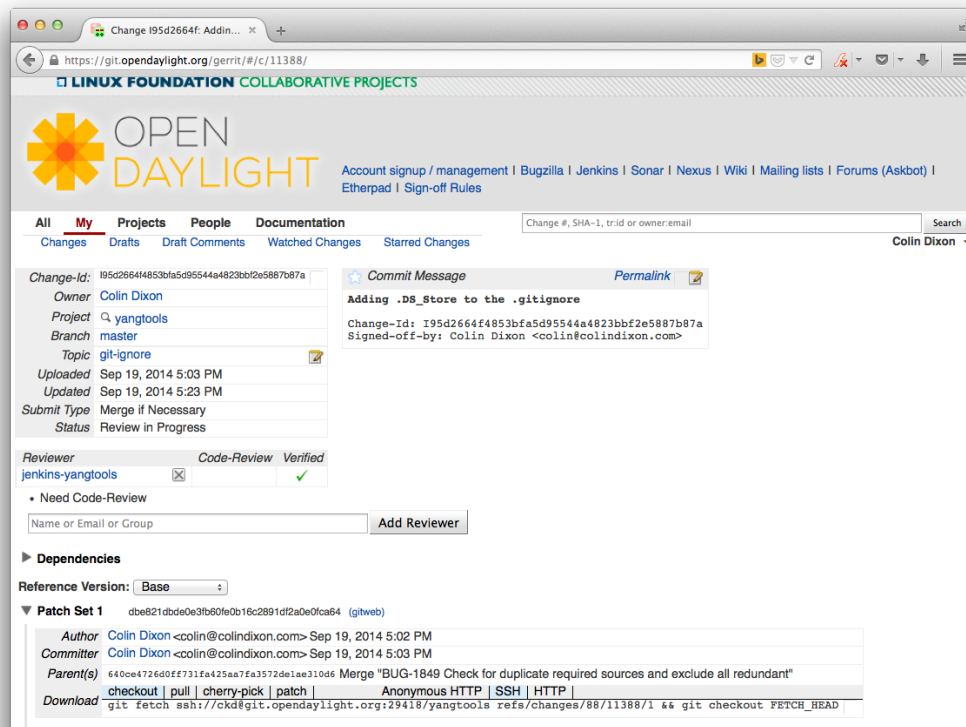
```
git push origin HEAD:refs/for/develop
```

You will get a message pointing you to your gerrit request like:

```
=====
remote: Resolving deltas: 100% (2/2) +
remote: Processing changes: new: 1, refs: 1, done      +
remote: +
remote: New Changes: +
remote:   http://git.opendaylight.org/gerrit/64 +
remote: +
=====
```

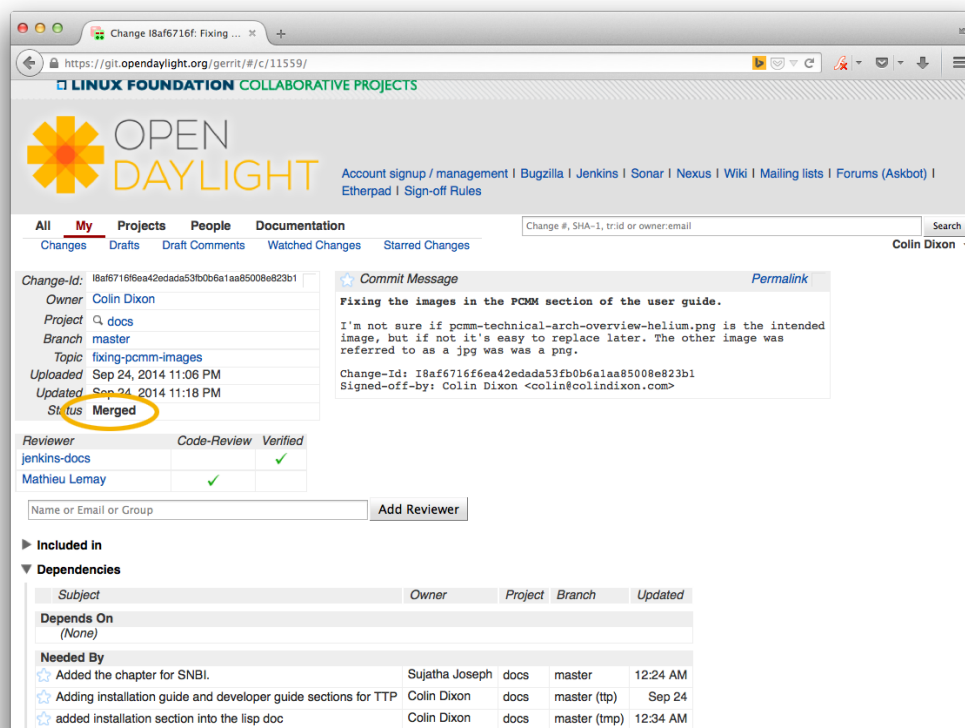
Viewing your Changes in Gerrit

Follow the link you got above to see your commit in Gerrit:

Figure 10.3. Gerrit Code Review Sample

Note that the Jenkins Controller User has verified your code and at the bottom is a link to the Jenkins build.

Once your code has been reviewed and submitted by a committer it will be merged into the authoritative repo, which would look like this:

Figure 10.4. Gerrit Code Merge Sample

Troubleshooting

1. What to do if your Firewall blocks port 29418

There have been reports that many corporate firewalls block port 29418. If that's the case, please follow the [Setting up HTTP in Gerrit](#) instructions and use git URL:

```
git clone https://<your_username>@git.opendaylight.org/gerrit/p/lispflowmapping.git
```

You will be prompted for the password you generated in [Setting up HTTP in Gerrit](#).

All other instructions on this page remain unchanged.

To download pre-built images with ODP bootstraps see the following Github project:

[Pre-Built OpenDaylight VM Images](#)

11. ODL-SDNi

Content on using ODL SDNi can be found on the OpenDaylight wiki here: https://wiki.opendaylight.org/view/ODL-SDNiApp:Developer_Guide

12. OpenFlow Protocol Library

Content on developing using the OpenFlow Protocol Library can be found on the OpenDaylight wiki here: https://wiki.opendaylight.org/view/Openflow_Protocol_Library:Documentation

13. OpenFlow Plugin

Table of Contents

| | |
|---|-----|
| OpenFlow Plugin: Sequence diagrams | 219 |
| OpenFlow Plugin:Config subsystem | 223 |
| Message Spy in OF Plugin | 229 |
| OpenFlow Plugin:Mininet | 232 |
| Installation | 232 |
| Usage | 235 |
| Coding tips for OpenFlow Plugin | 235 |
| OpenFlow Plugin: Wiring up notifications | 237 |
| OpenFlow Plugin:Python test scripts | 239 |
| General | 240 |
| ODL Test (odl_crud_tests.py) | 241 |
| Parameters | 242 |
| Stress Test (stress_test.py) | 243 |
| Operational Data Test (oper_data_test.py) | 243 |
| Switch restart (sw_restart_test.py) | 243 |
| OpenFlow Plugin: Robot framework tests | 244 |
| TLS support for OF Plugin | 245 |
| Configuring the ODL OpenFlow plugin | 247 |
| Configuring openvswitch SSL | 247 |
| Configuring a hardware switch with TLS | 248 |
| Open Flow Plugin: Support for extensibility | 249 |
| Overload protection in the OF Plugin | 251 |

Table 13.1. OpenFlow plugin: Component map

| Artifact ID | Component | Description |
|--|-----------------|--|
| openflowplugin | openflowplugin | Main implementation of OFPlugin |
| openflowplugin-it test-provider drop-test test-scripts | test | Support for end-to-end, integration, and regression testing |
| openflowplugin-controller-config | configSubsystem | Default configuration files for config subsystem |
| distributions-openflowplugin-base | distribution | OFPlugin distribution, based on the distribution of the controller, but the old (OF-1.0 only) plugin is replaced with the new plugin(OF-1.0+1.3) |
| learning-switch sample-consumer | sample | Sample projects demonstrating MD-SAL usage |
| vagrant | util | Materialize testing virtual machine containing mininet+ovs |

OpenFlow Plugin: Sequence diagrams

Figure 13.1. Message Lifecycle

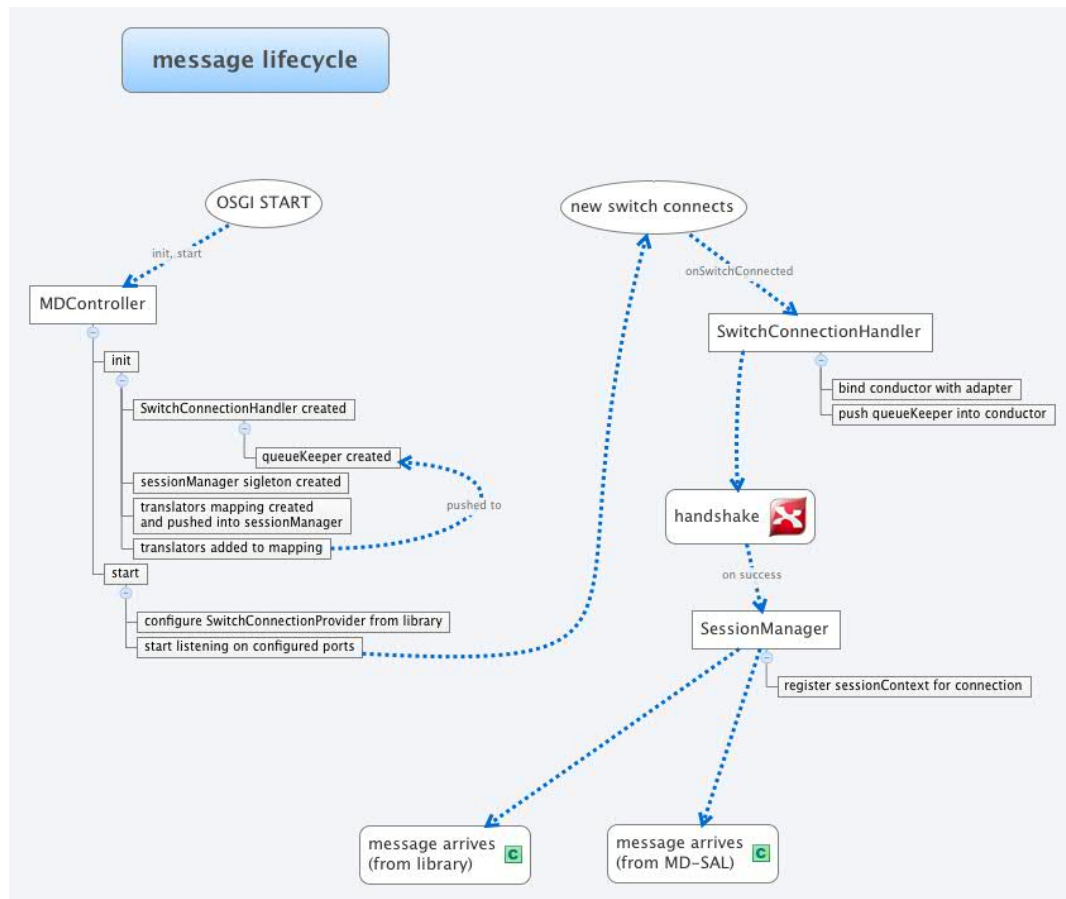
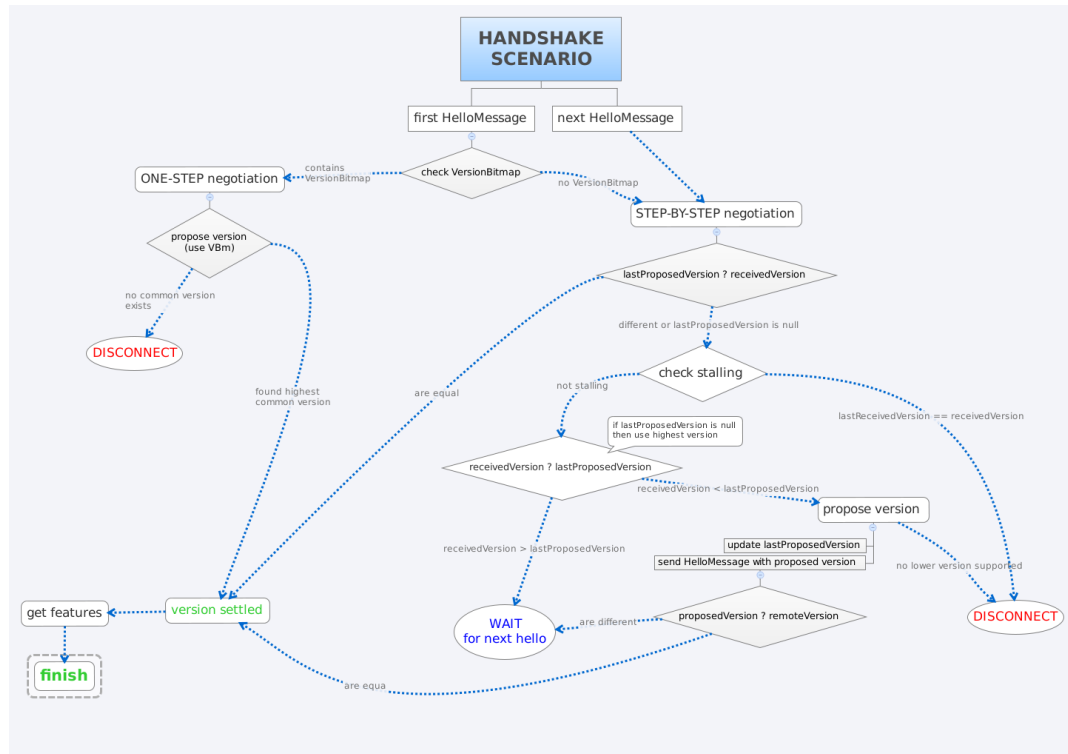


Figure 13.2. Handshake Scenario

OF1.0 Switch Handshake Sequence

```

sequenceDiagram
    participant Switch
    participant Controller
    participant ConnectionConductor
    participant HandshakeManager
    participant DOM

    Switch-->>Controller: Connect
    Controller->>ConnectionConductor: onConnectionReady()
    ConnectionConductor->>ConnectionConductor: setState(HANDSHAKING)
    ConnectionConductor->>HandshakeManager: startHandshake()
    alt [controller sends first HELLO]
        HandshakeManager-->>Controller: Hello( xid=M, version=0x4 )
    else [switch sends first HELLO]
        Controller->>ConnectionConductor: onHelloMessage
        ConnectionConductor->>HandshakeManager: continueHandshake()
        HandshakeManager->>HandshakeManager: stepByStepNegotiation
        HandshakeManager-->>HandshakeManager: 
        HandshakeManager-->>Controller: Hello( xid=N+1, version=0x01 )
    else [Switch sends a followup HELLO]
        Controller->>ConnectionConductor: onHelloMessage
        ConnectionConductor->>HandshakeManager: continueHandshake()
        HandshakeManager->>HandshakeManager: stepByStepNegotiation
        HandshakeManager-->>HandshakeManager: 
    else 
        HandshakeManager->>HandshakeManager: postHandshake
        HandshakeManager-->>HandshakeManager: 
        HandshakeManager-->>Switch: getFeatures
        ConnectionConductor->>ConnectionConductor: setState(WORKING)
        ConnectionConductor->>HandshakeManager: onHandshakeSuccessful
        ConnectionConductor->>DOM: registerDataPathID
    end
  
```

The diagram illustrates the OF1.0 Switch Handshake Sequence. It begins with the Switch sending a **Connect** message to the Controller. The Controller then calls **onConnectionReady()** on the ConnectionConductor, which sets its state to **HANDSHAKING**. The ConnectionConductor then calls **startHandshake()** on the HandshakeManager. An **alt** block handles the initial handshake: if the controller sends the first HELLO, the HandshakeManager responds with **Hello(xid=M, version=0x4)**; if the switch sends the first HELLO, the Controller calls **onHelloMessage** on the ConnectionConductor, which then calls **continueHandshake()** on the HandshakeManager. The HandshakeManager performs **stepByStepNegotiation** and responds with **Hello(xid=N+1, version=0x01)**. Another **alt** block handles a followup HELLO from the switch, where the Controller calls **onHelloMessage** on the ConnectionConductor, which then calls **continueHandshake()** on the HandshakeManager. The HandshakeManager again performs **stepByStepNegotiation**. Finally, the HandshakeManager calls **postHandshake** on itself, then sends **getFeatures** to the Switch. The ConnectionConductor sets its state to **WORKING** and calls **onHandshakeSuccessful** on the HandshakeManager. The ConnectionConductor also registers the data path ID with the DOM via **registerDataPathID**.

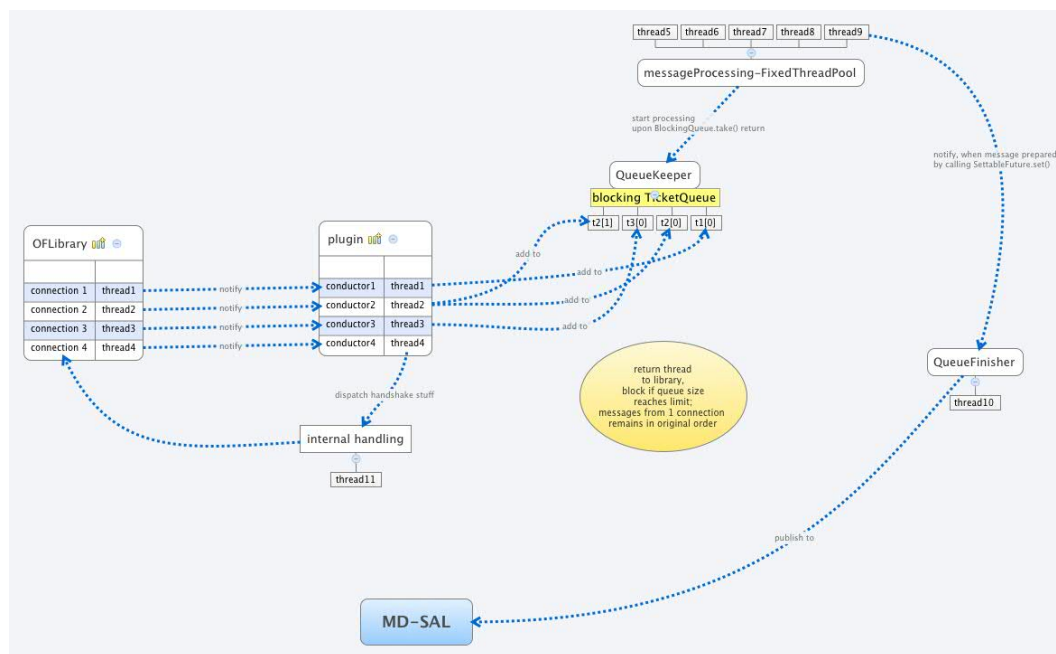
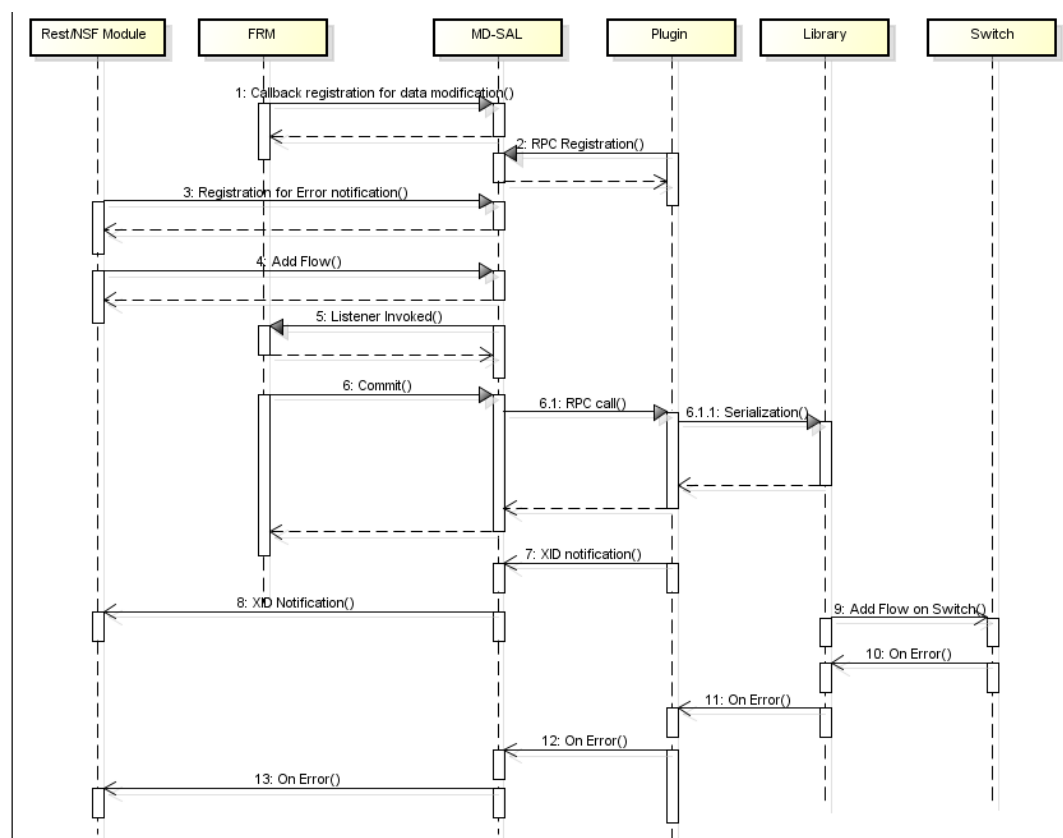
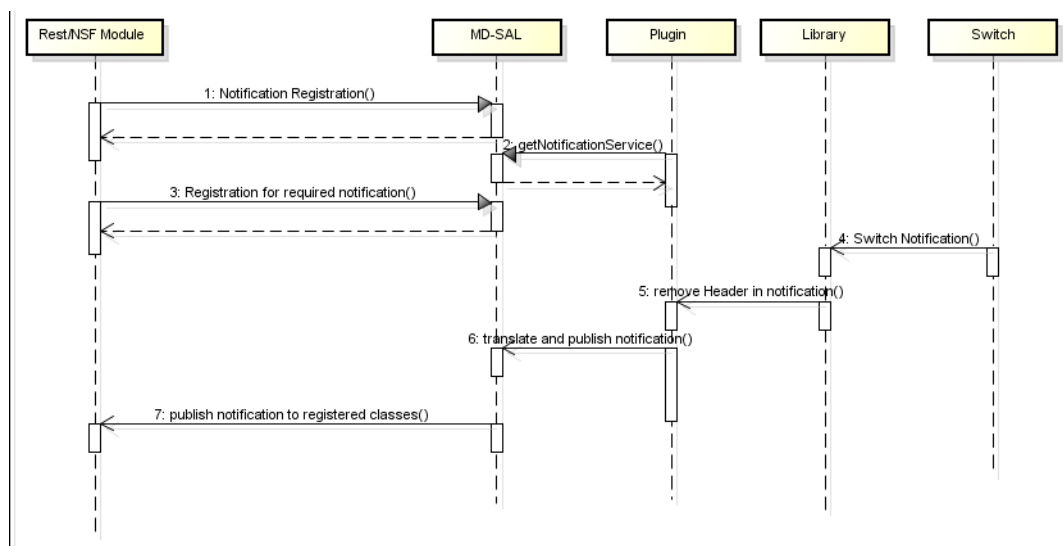
Figure 13.4. Message Order Preservation**Figure 13.5. Add Flow Sequence**

Figure 13.6. Generic Notification Sequence

OpenFlow Plugin:Config subsystem

Model provided modules by yang

General model (interfaces) - openflow-plugin-cfg.yang.

- The provided module is defined (identity openflow-provider)
- The target implementation is assigned (...OpenflowPluginProvider)

```

module openflow-provider {
  yang-version 1;
  namespace "urn:opendaylight:params:xml:ns:yang:openflow:common:config";
  prefix "ofplugin-cfg";

  import config {prefix config; revision-date 2013-04-05; }
  description
    "openflow-plugin-custom-config";
  revision "2014-03-26" {
    description
      "Initial revision";
  }
  identity openflow-provider{
    base config:service-type;
    config:java-class "org.opendaylight.openflowplugin.openflow.md.core.
sal.OpenflowPluginProvider";
  }
}
  
```

Implementation model - openflow-plugin-cfg-impl.yang

- The implementation of module is defined (identity openflow-provider-impl).
- The class name of the generated implementation is defined (ConfigurableOpenFlowProvider).

- The configuration of the module is defined through augmentation:
- This module requires an instance of a binding-aware-broker (container binding-aware-broker).
- Also required is a list of openflow-switch-connection-providers. (Those are provided by openflowjava: one plugin instance will orchestrate multiple openflowjava modules.)

```
module openflow-provider-impl {
  yang-version 1;
  namespace
    "urn:opendaylight:params:xml:ns:yang:openflow:common:config:impl";
  prefix "ofplugin-cfg-impl";

  import config {prefix config; revision-date 2013-04-05;}
  import openflow-provider {prefix openflow-provider;}
  import openflow-switch-connection-provider {prefix openflow-switch-
    connection-provider; revision-date 2014-03-28;}
  import opendaylight-md-sal-binding { prefix md-sal-binding; revision-date
    2013-10-28;}

  description
    "openflow-plugin-custom-config-impl";

  revision "2014-03-26" {
    description
      "Initial revision";
  }

  identity openflow-provider-impl {
    base config:module-type;
    config:provided-service openflow-provider:openflow-provider;
    config:java-name-prefix ConfigurableOpenFlowProvider;
  }

  augment "/config:modules/config:module/config:configuration" {
    case openflow-provider-impl {
      when "/config:modules/config:module/config:type = 'openflow-
        provider-impl'";

        container binding-aware-broker {
          uses config:service-ref {
            refine type {
              mandatory true;
              config:required-identity md-sal-binding:binding-broker-
osgi-registry;
            }
          }
        }

        list openflow-switch-connection-provider {
          uses config:service-ref {
            refine type {
              mandatory true;
              config:required-identity openflow-switch-connection-
provider:openflow-switch-connection-provider;
            }
          }
        }
      }
    }
  }
```

```

    }
  }
}

```

Generating config and sal classes from yangs



Note

Suitable code generators, needed in pom, are involved.

```

<build> ...
  <plugins>
    <plugin>
      <groupId>org.opendaylight.yangtools</groupId>
      <artifactId>yang-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>generate-sources</goal>
          </goals>
          <configuration>
            <codeGenerators>
              <generator>
                <codeGeneratorClass>
                  org.opendaylight.controller.config.yangjmxgenerator.plugin.
JMXGenerator
                </codeGeneratorClass>
                <outputBaseDir>${project.build.directory}/generated-sources/
config</outputBaseDir>
                <additionalConfiguration>
                  <namespaceToPackage1>
                    urn:opendaylight:params:xml:ns:yang:controller==org.
opendaylight.controller.config.yang
                  </namespaceToPackage1>
                </additionalConfiguration>
              </generator>
              <generator>
                <codeGeneratorClass>
                  org.opendaylight.yangtools.maven.sal.api.gen.plugin.
CodeGeneratorImpl
                </codeGeneratorClass>
                <outputBaseDir>${project.build.directory}/generated-sources/
sal</outputBaseDir>
              </generator>
              <generator>
                <codeGeneratorClass>org.opendaylight.yangtools.yang.unified.
doc.generator.maven.DocumentationGeneratorImpl</codeGeneratorClass>
                <outputBaseDir>${project.build.directory}/site/models</
outputBaseDir>
              </generator>
            </codeGenerators>
            <inspectDependencies>true</inspectDependencies>
          </configuration>
        </execution>
      </executions>
      <dependencies>
        <dependency>
          <groupId>org.opendaylight.controller</groupId>
          <artifactId>yang-jmx-generator-plugin</artifactId>

```

```
<version>0.2.5-SNAPSHOT</version>
</dependency>
<dependency>
  <groupId>org.opendaylight.yangtools</groupId>
  <artifactId>maven-sal-api-gen-plugin</artifactId>
  <version>${yangtools.version}</version>
  <type>jar</type>
</dependency>
</dependencies>
</plugin>
...
```

- JMX generator (target/generated-sources/config)
- sal CodeGeneratorImpl (target/generated-sources/sal)
- Documentation generator (target/site/models): [openflow generator](#) and [openflow provider impl](#).

Altering generated files

Those files were generated under `src/main/java` in the package as referred in yangs (if they exist, the generator will not overwrite them):

- ConfigurableOpenFlowProviderModuleFactory

The **instantiateModule** methods are extended in order to capture and inject the `osgi BundleContext` into module, so it can be injected into final implementation:

OpenflowPluginProvider `module.setBundleContext(bundleContext);`

- ConfigurableOpenFlowProviderModule

The **createInstance** method is extended in order to inject `osgi BundleContext` into the module implementation: `pluginProvider.setContext(bundleContext);`

Configuration xml file

The configuration file contains:

- Required capabilities
 - Modules definitions from `openflowjava`
 - Definitions from `openflowplugin`
- Modules definition
 - `openflow:switch:connection:provider:impl` (listening on port 6633, name=`openflow-switch-connection-provider-legacy-impl`)
 - `openflow:switch:connection:provider:impl` (listening on port 6653, name=`openflow-switch-connection-provider-default-impl`)
 - `openflow:common:config:impl` (having 2 services (wrapping those 2 previous modules) and `binding-broker-osgi-registry` injected)
- Provided services

- openflow-switch-connection-provider-default
- openflow-switch-connection-provider-legacy
- openflow-provider

```

<snapshot>
  <required-capabilities>

    <capability>urn:opendaylight:params:xml:ns:yang:openflow:switch:connection:provider:impl?
module=openflow-switch-connection-provider-impl&revision=2014-03-28</
capability>

    <capability>urn:opendaylight:params:xml:ns:yang:openflow:switch:connection:provider?
module=openflow-switch-connection-provider&revision=2014-03-28</capability>

    <capability>urn:opendaylight:params:xml:ns:yang:openflow:common:config:impl?
module=openflow-provider-impl&revision=2014-03-26</capability>
    <capability>urn:opendaylight:params:xml:ns:yang:openflow:common:config?
module=openflow-provider&revision=2014-03-26</capability>
  </required-capabilities>

  <configuration>

    <modules xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
      <module>
        <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:openflow:switch:connection:provider:impl">
          prefix=openflow-switch-connection-provider-impl
        </type>
        <name>openflow-switch-connection-provider-default-impl</name>
        <port>6633</port>
        <switch-idle-timeout>15000</switch-idle-timeout>
      </module>
      <module>
        <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:openflow:switch:connection:provider:impl">
          prefix=openflow-switch-connection-provider-impl
        </type>
        <name>openflow-switch-connection-provider-legacy-impl</name>
        <port>6653</port>
        <switch-idle-timeout>15000</switch-idle-timeout>
      </module>

      <module>
        <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:openflow:common:config:impl">
          prefix=openflow-provider-impl
        </type>
        <name>openflow-provider-impl</name>

        <openflow-switch-connection-provider>
          <type xmlns:ofSwitch=
"urn:opendaylight:params:xml:ns:yang:openflow:switch:connection:provider">
            ofSwitch=openflow-switch-connection-provider
          </type>
          <name>openflow-switch-connection-provider-default</name>

```

```

        </openflow-switch-connection-provider>
        <openflow-switch-connection-provider>
            <type xmlns:ofSwitch=
"urn:opendaylight:params:xml:ns:yang:openflow:switch:connection:provider">
                ofSwitch:openflow-switch-connection-provider
            </type>
            <name>openflow-switch-connection-provider-legacy</name>
        </openflow-switch-connection-provider>

        <binding-aware-broker>
            <type xmlns:binding=
"urn:opendaylight:params:xml:ns:yang:controller:md:sal:binding">
                binding:binding-broker-osgi-registry
            </type>
            <name>binding-osgi-broker</name>
        </binding-aware-broker>
    </module>
</modules>

    <services xmlns="urn:opendaylight:params:xml:ns:yang:controller:config">
        <service>
            <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:openflow:switch:connection:provider">
                prefix:openflow-switch-connection-provider
            </type>
            <instance>
                <name>openflow-switch-connection-provider-default</name>
                <provider>/modules/module[type='openflow-switch-connection-
provider-impl'][name='openflow-switch-connection-provider-default-impl']</
provider>
            </instance>
            <instance>
                <name>openflow-switch-connection-provider-legacy</name>
                <provider>/modules/module[type='openflow-switch-connection-
provider-impl'][name='openflow-switch-connection-provider-legacy-impl']</
provider>
            </instance>
        </service>

        <service>
            <type xmlns:prefix=
"urn:opendaylight:params:xml:ns:yang:openflow:common:config">prefix:openflow-
provider</type>
            <instance>
                <name>openflow-provider</name>
                <provider>/modules/module[type='openflow-provider-impl'][name=
'openflow-provider-impl']</provider>
            </instance>
        </service>
    </services>

</configuration>
</snapshot>

```

API changes

In order to provide multiple instances of modules from openflowjava, there is an API change. Previously, the OFPlugin got access to the SwitchConnectionProvider exposed by OFJava, and injected the collection of configurations so that for every configuration, a new instance of the TCP listening server was created. Now, those configurations are provided by the configSubsystem, and the configured modules (wrapping the original SwitchConnectionProvider) are injected into the OFPlugin (wrapping SwitchConnectionHandler).

Providing config file (IT, local distribution/base, integration/distributions/base)

openflowplugin-it

The whole configuration is contained in one file (controller.xml). The entries needed in order to start up and wire the OEPlugin + OFJava are simply added there.

OFPlugin/distribution/base

The new config file is added (src/main/resources/configuration/initial/42-openflow-protocol-impl.xml), and copied to the config/initial subfolder of the build.

Integration/distributions/build

In order to push the actual config into the config/initial subfolder of distributions/base in the integration project, a new artifact was created in OFPlugin. The openflowplugin-controller-config contains only the config xml file under src/main/resources. Another change was committed into the integration project. During a build, this config xml is extracted and copied to the final folder in order to be accessible during the controller run.

Message Spy in OF Plugin

With the intent to debug, the OpenFlow plugin implements a Message Spy to monitor controller communications. The Message Spy collects and displays message statistics.

Message statistics collection

Message statistics are grouped according to message type and checkpoint. The counter assigned to a checkpoint and message class increases by 1 when a message passes through.

The following checkpoints count passing messages:

```
/**
 * statistic groups overall in OFPlugin
 */
enum STATISTIC_GROUP {
    /** message from switch, enqueued for processing */
    FROM_SWITCH_ENQUEUED,
    /** message from switch translated successfully - source */
    FROM_SWITCH_TRANSLATE_IN_SUCCESS,
    /** message from switch translated successfully - target */
    FROM_SWITCH_TRANSLATE_OUT_SUCCESS,
    /** message from switch where translation failed - source */
}
```

```

FROM_SWITCH_TRANSLATE_SRC_FAILURE,
/** message from switch finally published into MD-SAL */
FROM_SWITCH_PUBLISHED_SUCCESS,
/** message from switch - publishing into MD-SAL failed */
FROM_SWITCH_PUBLISHED_FAILURE,

/** message from MD-SAL to switch via RPC enqueued */
TO_SWITCH_ENQUEUED_SUCCESS,
/** message from MD-SAL to switch via RPC NOT enqueued */
TO_SWITCH_ENQUEUED_FAILED,
/** message from MD-SAL to switch - sent to OFJava successfully */
TO_SWITCH_SUBMITTED_SUCCESS,
/** message from MD-SAL to switch - sent to OFJava but failed*/
TO_SWITCH_SUBMITTED_FAILURE
}

```

Message statistics display

Access the message statistics by means of logs, osgi, and jmx.

- **osgi command (on demand):** This method is considered deprecated. : `osgi> dumpMsgCount`
- **From the controller console where statistics are refreshed every 10 seconds:** Required logback settings: `<logger name="org.opendaylight.openflowplugin.openflow.md.queue.MessageSpyCounterImpl" level="DEBUG" />`
- **As JMX from the jconsole:**
 - Start the OFplugin with the `-jmx` parameter.
 - Tab MBeans contains `org.opendaylight.controller`.
 - `RuntimeBean` has a `msg-spy-service-impl`.
 - `Operations` provides `makeMsgStatistics` report functionality.

Sample results

```

DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_ENQUEUED:
MSG[PortStatusMessage] -> +0 | 1
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_ENQUEUED:
MSG[MultipartReplyMessage] -> +24 | 81
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_ENQUEUED:
MSG[PacketInMessage] -> +8 | 111
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_IN_SUCCESS:
MSG[PortStatusMessage] -> +0 | 1
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_IN_SUCCESS:
MSG[MultipartReplyMessage] -> +24 | 81
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_IN_SUCCESS:
MSG[PacketInMessage] -> +8 | 111
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[QueueStatisticsUpdate] -> +3 | 7
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[NodeUpdated] -> +0 | 3
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[NodeConnectorStatisticsUpdate] -> +3 | 7

```

```
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[GroupDescStatsUpdated] -> +3 | 7
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[FlowsStatisticsUpdate] -> +3 | 19
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[PacketReceived] -> +8 | 111
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[MeterFeaturesUpdated] -> +0 | 3
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[GroupStatisticsUpdated] -> +3 | 7
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[GroupFeaturesUpdated] -> +0 | 3
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[MeterConfigStatsUpdated] -> +3 | 7
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[MeterStatisticsUpdated] -> +3 | 7
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[NodeConnectorUpdated] -> +0 | 12
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_OUT_SUCCESS:
MSG[FlowTableStatisticsUpdate] -> +3 | 8
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_TRANSLATE_SRC_FAILURE: no
activity detected
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[QueueStatisticsUpdate] -> +3 | 7
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[NodeUpdated] -> +0 | 3
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[NodeConnectorStatisticsUpdate] -> +3 | 7
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[GroupDescStatsUpdated] -> +3 | 7
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[FlowsStatisticsUpdate] -> +3 | 19
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[PacketReceived] -> +8 | 111
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[MeterFeaturesUpdated] -> +0 | 3
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[GroupStatisticsUpdated] -> +3 | 7
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[GroupFeaturesUpdated] -> +0 | 3
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[MeterConfigStatsUpdated] -> +3 | 7
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[MeterStatisticsUpdated] -> +3 | 7
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[NodeConnectorUpdated] -> +0 | 12
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_SUCCESS:
MSG[FlowTableStatisticsUpdate] -> +3 | 8
DEBUG o.o.o.s.MessageSpyCounterImpl - FROM_SWITCH_PUBLISHED_FAILURE: no
activity detected
DEBUG o.o.o.s.MessageSpyCounterImpl - TO_SWITCH_ENQUEUED_SUCCESS:
MSG[AddFlowInput] -> +0 | 12
DEBUG o.o.o.s.MessageSpyCounterImpl - TO_SWITCH_ENQUEUED_FAILED: no activity
detected
DEBUG o.o.o.s.MessageSpyCounterImpl - TO_SWITCH_SUBMITTED_SUCCESS:
MSG[AddFlowInput] -> +0 | 12
DEBUG o.o.o.s.MessageSpyCounterImpl - TO_SWITCH_SUBMITTED_FAILURE: no activity
detected
```


OpenFlow Plugin:Mininet

Mininet on debian wheezy(7), x86_64

Requirements

Openvswitch

1. Install all requirements.

```
apt-get install build-essential fakeroot  
apt-get install debhelper autoconf automake libssl-dev pkg-config bzip2  
openssl python-all procps python-qt4 python-zopeinterface python-twisted-  
conch
```

1. Install a few helper applications.

```
apt-get -y install screen sudo vim etckeeper mlocate autoconf2.13 libssl-dev  
graphviz tcpdump gdebi-core
```

Test the Python environment

Python pip

1. Install setuptools.

```
wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py  
sudo python ez_setup.py
```

1. Install pip.

```
wget https://raw.githubusercontent.com/pypa/pip/master/contrib/get-pip.py  
sudo python get-pip.py
```

1. Post install the python libraries required by the ODL testing script.

```
sudo pip install netaddr
```

Installation

Openvswitch 2.0.0

1. Remove the old packages, as root:

```
sudo -i  
apt-get remove openvswitch-common openvswitch-datapath-dkms openvswitch-  
controller openvswitch-pki openvswitch-switch
```

1. Download and unpack OpenV Switch 2.0.0.

```
wget http://openvswitch.org/releases/openvswitch-2.0.0.tar.gz
```

```
tar zxvf openvswitch-2.0.0.tar.gz
```

Build and install

1. Install the openvswitch package. Deploy it using the module assistant at: <https://wiki.debian.org/ModuleAssistant>

```
cd ../
gdebi openvswitch-datapath-source_2.0.0-1_all.deb
module-assistant auto-install openvswitch-datapath
gdebi openvswitch-common_2.0.0-1_amd64.deb
gdebi openvswitch-switch_2.0.0-1_amd64.deb
gdebi openvswitch-pki_2.0.0-1_all.deb
gdebi openvswitch-controller_2.0.0-1_amd64.deb
```

Post installation settings

```
service openvswitch-controller stop
update-rc.d openvswitch-controller disable
```

Test installation

```
ovs-vsctl show
ovs-vsctl --version
ovs-ofctl --version
ovs-dpctl --version
ovs-controller --version
```

Mininet 2.1.0

1. Download and checkout the required version.

```
git clone git://github.com/mininet/mininet
cd mininet
git checkout -b 2.1.0 2.1.0
```

1. Compile and install mininet.

```
gcc mnexec.c -o mnexec
mv mnexec /usr/bin/
python setup.py install
```

1. Test the installation.

```
mn --version
mn --test pingall
```

Expected result

```
root@debian:~/mininet# mn --version
2.1.0
root@debian:~/mininet# mn --test pingall
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
```

```

s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
*** Starting 1 switches
s1
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
*** Stopping 1 switches
s1 ..
*** Stopping 2 hosts
h1 h2
*** Stopping 1 controllers
c0
*** Done
completed in 0.269 seconds

```

Post installation additions

- Modify the source code of the mininet node.py file as described in [Stage 3](#).

```

--- /root/mininet/build/lib.linux-x86_64-2.7/mininet/node.py      2013-11-22
03:35:12.000000000 -0800
+++ /usr/local/lib/python2.7/dist-packages/mininet-2.1.0-py2.7.egg/mininet/
node.py      2013-11-22 06:17:07.350574387 -0800
@@ -952,6 +952,10 @@
        datapath: userspace or kernel mode (kernel|user)"""
        Switch.__init__( self, name, **params )
        self.failMode = failMode
+       protKey = 'protocols'
+       if self.params and protKey in self.params:
+           print 'have protocol params!'
+           self.opts += protKey + '=' + self.params[protKey]
        self.datapath = datapath

@@ -1027,8 +1031,9 @@
        if self.datapath == 'user':
            self.cmd( 'ovs-vsctl set bridge', self, 'datapath_type=netdev' )
            int( self.dpid, 16 ) # DPID must be a hex string
+       print 'OVSSwitch opts: ', self.opts
        self.cmd( 'ovs-vsctl -- set Bridge', self,
-               'other_config:datapath-id=' + self.dpid )
+               self.opts + ' other_config:datapath-id=' + self.dpid )
        self.cmd( 'ovs-vsctl set-fail-mode', self, self.failMode )
        for intf in self.intfList():
            if not intf.IP():

```

Start and test the modified mininet

1. Start the mn session:

```

sudo mn --topo single,3 --controller 'remote,ip=<your controller IP>' --
switch ovsk,protocols=OpenFlow10

```

1. Alternatively, use this command:

```
sudo mn --topo single,3 --controller 'remote,ip=<your controller IP>' --  
switch ovsk,protocols=OpenFlow13
```

1. Test the version of the protocol used by switch "s1":

```
ovs-ofctl -O OpenFlow10 show s1  
ovs-ofctl -O OpenFlow13 show s1
```

Usage

REST tests openflowplugin

```
sudo python odl_tests.py --xmls 1,2
```

- For more option informations, use:

```
sudo python odl_tests.py --help
```

Coding tips for OpenFlow Plugin

If you use Eclipse, the following compiler settings might be useful either during coding or while fixing errors. The following errors are noteworthy:

- name shadowing.
- null checks.
- missing case in switch block.
- missing break in case.
- unused variables/parameters.
- annotations checks (@override).
- access to non accessible member of enclosing type.
- If overriding hashCode or equals, both must be overridden.

Also useful are warnings upon missing javadoc comments for public classes, members, and methods.

Figure 13.7. Configure Compiler Errors and Warnings

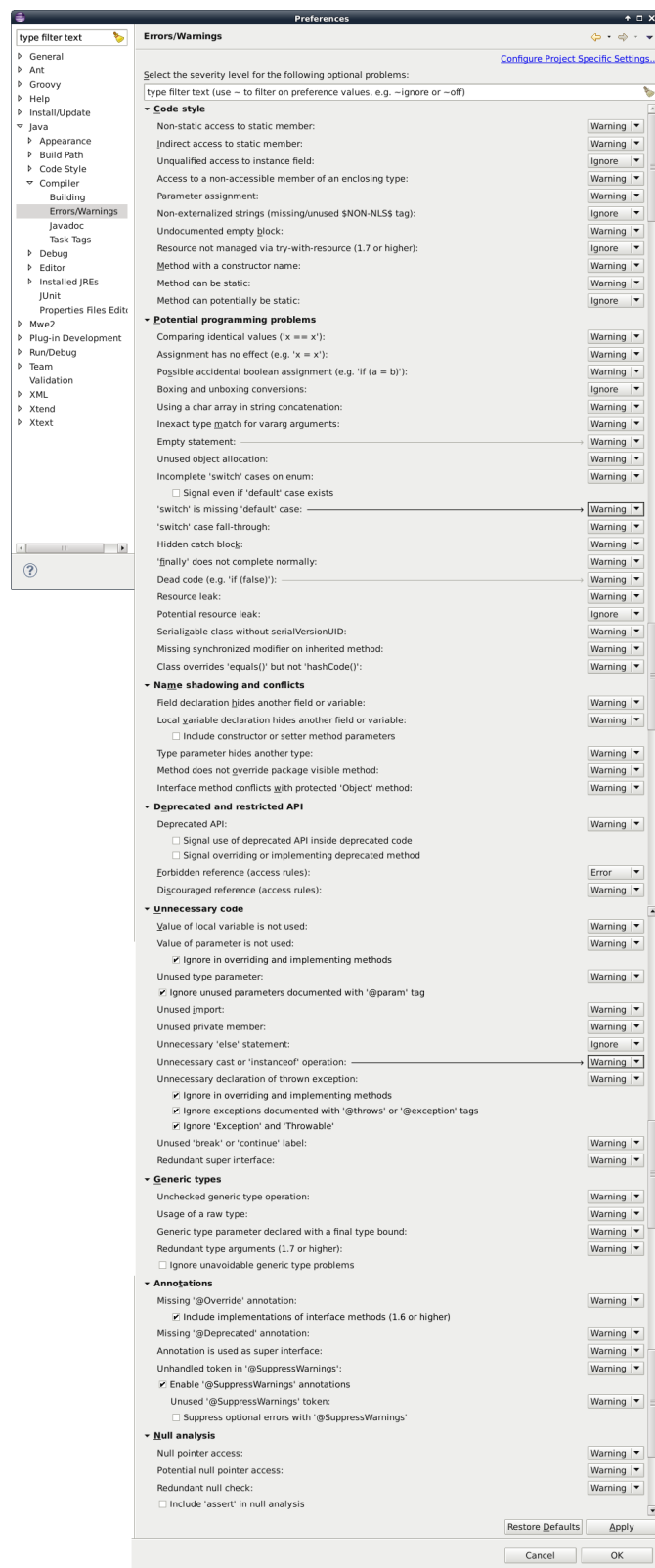
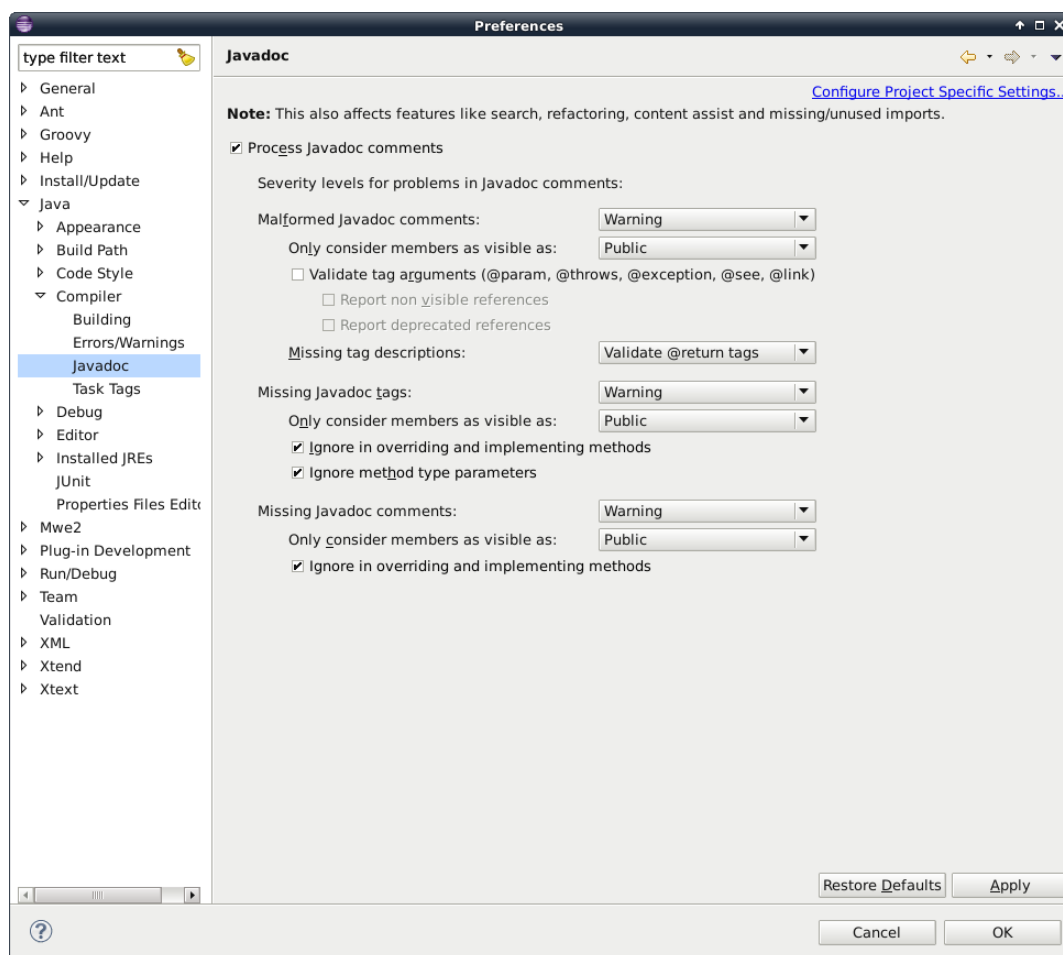


Figure 13.8. Configure Javadoc

OpenFlow Plugin: Wiring up notifications

Introduction

OpenFlow messages coming from the OpenflowJava plugin into MD-SAL Notification objects must be translated, and then published to the MD-SAL.

To create and register a Translator

1. Create a Translator class.
2. Register the Translator.
3. Register the notificationPopListener to handle Notification Objects.

Creating a Translator Class

An example is available in [PacketInTranslator.java](#).

1. Create the class.

```
public class PacketInTranslator implements IMDMessageTranslator<OfHeader,  
List<DataObject>> {
```

1. Implement the translate function:

```
public class PacketInTranslator implements IMDMessageTranslator<OfHeader,  
List<DataObject>> {  
  
    protected static final Logger LOG = LoggerFactory  
        .getLogger(PacketInTranslator.class);  
    @Override  
    public PacketReceived translate(SwitchConnectionDistinguisher cookie,  
        SessionContext sc, OfHeader msg) {  
        ...  
    }  
}
```

1. Ensure that the type is the expected one, and cast it:

```
if(msg instanceof PacketInMessage) {  
    PacketInMessage message = (PacketInMessage)msg;  
    List<DataObject> list = new CopyOnWriteArrayList<DataObject>();
```

1. Complete the translation and return.

```
PacketReceived pktInEvent = pktInBuilder.build();  
list.add(pktInEvent);  
return list;
```

Registeing the Translator Class

- Go to [MDController.java](#) and in `init()` add register your Translator:

```
public void init() {  
    LOG.debug("Initializing!");  
    messageTranslators = new ConcurrentHashMap<>();  
    popListeners = new ConcurrentHashMap<>();  
    //TODO: move registration to factory  
    addMessageTranslator(ErrorMessage.class, OF10, new ErrorTranslator());  
    addMessageTranslator(ErrorMessage.class, OF13, new ErrorTranslator());  
    addMessageTranslator(PacketInMessage.class, OF10, new  
        PacketInTranslator());  
    addMessageTranslator(PacketInMessage.class, OF13, new  
        PacketInTranslator());  
}
```



Note

There is a separate registration for each of the OF10 and OF13. Basically, you indicate the type of openflowjava message you wish to translate for, the OF version, and an instance of your Translator.

Registering your MD-SAL message for notification to the MD-SAL

- In `MDController.init()` register to have the `notificationPopListener` handle your MD-SAL Message:

```
addMessagePopListener(PacketReceived.class, new  
    NotificationPopListener<DataObject>());
```

When a message comes from the openflowjava plugin, it will be translated and published to the MD-SAL.

OpenFlow Plugin:Python test scripts

Prerequisites for Python test-scripts

- Linux based OS (these instructions cover debian 7 - wheezy)
- Java 1.7+
- Python (v 2.6)
- Openvswitch (v 2.0.0)
- Mininet (v 2.1.0)
- Controller (supporting openflow 1.3)

Installing python tools



Note

Build python tools with python2.6, not the default python.

- wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py
- python2.6 ez_setup.py
- wget <https://raw.github.com/pypa/pip/master/contrib/get-pip.py>
- python2.6 get-pip.py

See [the section called "OpenFlow Plugin:Mininet" \[232\]](#)

Installing Wireshark

1. apt-get install wireshark
2. Make yourself a standard user again (CTRL^D)
3. sudo dpkg-reconfigure wireshark-common
4. sudo usermod -a -G wireshark \$USER
5. sudo reboot

Adding openflow13 dissector to wireshark

1. mkdir /home/mininet/.wireshark/plugins/

2. Copy the file openflow.so to this directory //TODO add attachment.

Controller

Install Java JDK and set JAVA_HOME

1. apt-get install openjdk-7-jdk
2. Export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64/jre/bin/java

Download, unzip, and run the integration build

1. Find the latest integration/distribution/base build on nexus.
2. Download it (using for example, wget <url to artifact.zip>) and unzip it (using for example, unzip <artifact.zip>)
3. Start the controller:

```
cd opendaylight
./run.sh -of13
```

Clone openflowplugin project

- git clone <https://git.opendaylight.org/gerrit/p/openflowplugin.git>

Tests

- locations: openflowplugin/test-scripts
- content directory
 - xmls (switch configuration input in xml form)
 - openvswitch
 - **runnable files:**
 - odl_crud_tests.py
 - stress_test.py
 - oper_data_test.py
 - sw_restart_test.py

General

The tests are designed for running on Linux based machines with installed ovs and mininet python scripts. All scripts has to be started with same permission as mininet (**sudo**). Otherwise the scripts can not start mininet. All runnable scripts contains a **help** description for input parameters for a quick orientation.

Basic parameters for all runnable scripts:

- `--help`: dump help
- `--mnport`: A controller port listener for the openflow switch communications. The parameter is used for configuration startup of the Mininet. A default value is **6653**.
- `--odlhost`: A controller IP address. The parameter is used for configuration startup of the Mininet and for the rest address builders. A default value is **127.0.0.1** (localhost).
- `--odlport`: A controller port listener for a http REST communication. The parameter is used for the rest address builders.

ODL Test (odl_crud_tests.py)

The test scripts are designed like CRUD (Create Read Update Delete) End-to-End black-box test suite for testing the switch configuration inputs/outputs via RESTconf. (It could work with mininet [opf13] by CPqD,OVS only.)

All inputs are read from xml files:

- file prefix f*.xml # Flow ;
- file prefix g*.xml # Group ;
- file prefix m*.xml # Meter ;



Note

Only the Groups and the Meters are supported by CPqD.

The test uses:

- RESTfull (GET, PUT, POST (create data only), DELETE)
- RESTconf POST sal-services

Test life cycle

1. Read input and put in to controller via REST (PUT | POST | POST sal-add).
2. Get the stored data via REST from config DataStore and compare input vs output (GET).
3. Get the stored data via REST from operational DataStore and compare input vs output (GET).
4. Modify the input and the update put in to controller via REST (PUT | POST sal-update).
5. Delete the input via REST (DELETE | POST sal-remove).
6. Validate the delete process in config DS and operational DS (GET).

Parameters

- `--odlhost`: odl controller host (default value is 127.0.0.1)
- `--odlport`: odl RESTconf listening port (default value is 8080)
- `--loglev`: tlogging level definition (default value is DEBUG) debug level contains request/response payload
- `--mininet`: OpenVSwitch or CPqD (default OVS)
- `--fxmls`:The number specifies a Flow test xml file from xmls directory (pattern: f{nr}.xml) (e.g. 1,3,34). This parameter has no default value. The script is testing all f_.xml files from xmls directory without `-fxmls` parameter. 0 means no test. The parameter is relevant for (OVS and CPqD)
- `--mxmls`:The number specifies a Meter test xml file from xmls directory (pattern: m{nr}.xml) (e.g. 1,3). This parameter has no default value. The script is testing all m_.xml files from xmls directory without `-mxmls` parameter. 0 means no test. The parameter is relevant for (CPqD only)
- `--gxmls`:The number specifies a Group test xml file from xmls directory (pattern: g{nr}.xml) (e.g. 1,3). This parameter has no default value. The script is testing all g_.xml files from xmls directory without `-gxmls` parameter. 0 means no test. The parameter is relevant for (CPqD only)
- `--confresp`: (configuration response) - define a delay to the Configuration Data Store (default = 0 sec.) Increase this value is important for a weaker controller machine
- `--operresp`: (operation response) - define a delay to the Operation Data Store (default = 3 sec.) Increase this value is important for a weaker controller machine or a weaker network
- `--coloring`: switcher for enable/disable coloring logged output



Note

The script has a file and the console logging output handlers (file `crud_test.log`).

cmd example:

```
python odl_crud_tests.py --mininet 2 --fxmls 1 --gxmls 0 --mxmls 3 --loglev 2
```

cmd means: The script expects ODL Controller RESTconf listener in 127.0.0.1:8080; the script expects Mininet by CPqD (gxmls and mxmls params are not ignored); and the script create the tests for f1.xml, and m3.xml and the script shows only INFO and ERROR logging messages which are colourized.



Note

The device Errors listener is not supported yet. We recommend that you use a wireshark tool for the investigation of an unexpected behaviour.

Stress Test (stress_test.py)

The test simulates multiple connections for the repeatable END-TO-END add flow test scenario. The flow pattern is the same (look at `openvswitch.flow_tools.py`). The script changes only a `flow_id` value.

The test life cycle:

- Initialize mininet and thread pool
- The incremental add flow's group (in every thread from thread pool)
- Check nr. of flows (validate numbers of flows with expected calculated values and make report)
- Get all flows from switch directly by command line
- Get all flows from configuration DataStore
- Get all flows from the operational DataStore
- Incrementally delete the groups of the flow (in every thread from thread pool) final report

Parameters:

- `--threads`: number of threads which should be used for multiple connection simulation in the thread pool. The default value is 50
- `--flows`: number of flows which should be used for add connection samples

Operational Data Test (oper_data_test.py)

The test checks the operational store of the controller. The Flow addition action and deletion action from the Data Store. When a flow is added via REST, it is added to the config store and then pushed to the switch. When it is successfully pushed to the switch, it is also moved to the operational store. Deletion also happens the same way.

You can specify the number of flows added by the parameter:

```
--flows : number of the flows which are add to switch. The default value is 100
```

Switch restart (sw_restart_test.py)

The test is for a flow addition to a switch after the switch has been restarted. After the switch is restarted, it should get the flow configuration from the controller operational datastore. The speed at which the configuration is pushed to the restarted switch may vary. So, you can specify the wait time; and the number of retries by wait time; and the number of retries by:

```
sw_restart_test.py --wait WAIT_TIME (default is 30)
sw_restart_test.py --retry NO_RETRIES (default is 1)
```

You can also specify that flows are added by xmls from the /xmls folder. If you do not specify this parameter, the default xml template will be used.

```
sw_restart_test.py --xmls XMLS (default is generic template)
```

OpenFlow Plugin: Robot framework tests

Prerequisites for robot tests

- Virtual machine with Mininet for OF1.0 and OF1.3 and with OpenSwitch
- Current version of ODL Controller
- Python (v 2.6 and higher)
- Robot framework
- GIT

Installation

There are in three puzzle pieces:

- ODL controller
- Mininet with ovs
- Robot framework + tests



Note

Use VMs to run them on the same machine or distribute them.

All-in-one strategy: Advantages and disadvantages

- Easy to transfer whole setup (if running on VM)
- No network issues (especially between VMs)
- However, there is no simple way to switch or update mininet or ovs

Distributed strategy: Robot + ODL controller on one VM, mininet on another

- Modularity
- Transfer of the whole set-up involves two VMs
- VMs need network access to one another (This can be achieved by the *internal network* of virtualBox.)

VM with Mininet

There are three options to create a VM:

- Follow instructions on this Opendaylight wiki page at: [Install Mininet for OF1.0 and OF1.3](#)
- Download [Preinstalled VMs](#) or there is also a possibility to create mininet VM from scratch (based on debian distribution)



Important

In order for robot framework to be able to control mininet through ssh the prompt on mininet VM has to end with ">" character.

| Component | Topic | Included in Guide |
|-----------|---|-------------------|
| MD-SAL | Southbound Protocol Plugin | Developer guide |
| MD-SAL | Plugin Types: <ul style="list-style-type: none">• Southbound Protocol Plugin• Manager-type Application• Protocol Library• Connector Plugin | User Guide |

TLS support for OF Plugin

SDN separates the data plane from the control plane of networks. It is imperative that communication between the two planes is secure. Secure communications between the data plane switches and controllers on the control plane require the authentication of switches and controllers. Authentication ensures that no unsecured switch connects to a controller, and that no unsecured controller manages a switch. When a controller with TLS configured is opened, the OpenFlow port only accepts Transport Layer Security (TLS) communications. Any switch without TLS configured will fail in its connection attempt.

Open Secure Sockets Layer (SSL) provides the tools for the public key infrastructure (PKI) management required to establish secure connections between a controller and switches. Information on 'SSL on Open vSwitch and ovs controller' is available at: <https://github.com/mininet/mininet/wiki/SSL-on-Open-vSwitch-and-ovs-controller>

In a lab environment, the private key of the controller resides on the mininet host that also acts as the Certification authority (CA) signing host. In a production environment, the key generation for the controller would be separate from that of the switches; only the public controller key is shared with the switches.



Note

While in a lab environment, TLS may be configured with the keystore shipped with the controller, the TLS configuration in a production environment must choose a different keystore.

Creating and signing private and public key certificates Use ovs pki to create private keys and public certificate files for the switches and the controller.

1. On the mininet host, verify whether PKI is initialized: `ls /var/lib/openvswitch/pki/controllerca/cacert.pem`

2. If PKI is not initialized, use: `ovs-pki init`

3. To generate the signed certificates, use the request certificates `sc-req.pem` and `ctl-req.pem`:

```
$ ls /etc/openvswitch
conf.db ctl-cert.pem ctl-privkey.pem ctl-req.pem sc-cert.pem sc-privkey.pem
sc-req.pem
system-id.conf
```

1. To create private keys and public cert files for the switches and the controller, run the `ovs-pki`:

```
cd /etc/openvswitch
sudo ovs-pki req+sign sc switch
sudo ovs-pki req+sign ctl controller
```

1. From `.pem` files, create an intermediate Open SSL PKCS 12 formatted keystore to hold the private key for the controller.

```
sudo openssl pkcs12 -export -in ctl-cert.pem -inkey ctl-privkey.pem \
-out ctl.p12 -name odlserver \
-CAfile /var/lib/openvswitch/pki/controllerca/cacert.pem -caname root -chain
You'll be prompted for a password, use "opendaylight"
Enter Export Password:
Verifying - Enter Export Password:
```

1. Copy the intermediate keystore, which has the private key of the controller, and the switches public key cert file (`ctl.p12` and `sc-cert.pem`) from the mininet host to any work directory on the controller machine. Import the PKCS 12 format to a Java compatible format that the controller can use:

```
sftp mininet@mininetipaddress
mininet
sftp get ctl.p12 sc-cert.pem
quit
```

1. For use in the steps that follow, find a `keytool` in a `jdk bin` directory, and add it to the path:

```
keytool -importkeystore \
-deststorepass opendaylight -destkeypass opendaylight -destkeystore
ctl.jks \
-srckeystore ctl.p12 -srcstoretype PKCS12 -srcstorepass opendaylight \
-alias odlserver
```

1. Store the public key of the switch in a truststore:

```
keytool -importcert -file sc-cert.pem -keystore truststore.jks -storepass
opendaylight
# when prompted "Trust this certificate? [no]:" enter "yes"
# Certificate was added to keystore
```

1. Copy the two keystores to the ssl configuration directory:

```
mkdir ODLINSTALL/configuration/ssl
cp ctl.jks truststore.jks ODLINSTALL/configuration/ssl
```

Configuring the ODL OpenFlow plugin

- Configure the OF plugin using the following:

```
cd configuration/initial
vi configuration/initial/42-openflowplugin.xml
# add the <tls> blocks as shown to each of the existing OF-switch-connection-
provider modules

    <!-- default OF-switch-connection-provider (port 6633) -->
    <module>
      <type xmlns:prefix=
"urn:.opendaylight:params:xml:ns:yang:openflow:switch:connection:provider:impl">
        prefix:openflow-switch-connection-provider-impl
      </type>
      <name>openflow-switch-connection-provider-default-impl</name>
      <port>6633</port>
      <switch-idle-timeout>15000</switch-idle-timeout>
      <tls>
        <keystore>configuration/ssl/ctl.jks</keystore>
        <keystore-type>JKS</keystore-type>
        <keystore-path-type>PATH</keystore-path-type>
        <keystore-password>opendaylight</keystore-password>
        <truststore>configuration/ssl/truststore.jks</truststore>
        <truststore-type>JKS</truststore-type>
        <truststore-path-type>PATH</truststore-path-type>
        <truststore-password>opendaylight</truststore-password>
        <certificate-password>opendaylight</certificate-password>
      </tls>

    </module>
    <!-- default OF-switch-connection-provider (port 6653) -->
    <module>
      <type xmlns:prefix=
"urn:.opendaylight:params:xml:ns:yang:openflow:switch:connection:provider:impl">
        prefix:openflow-switch-connection-provider-impl
      </type>
      <name>openflow-switch-connection-provider-legacy-impl</name>
      <port>6653</port>
      <switch-idle-timeout>15000</switch-idle-timeout>
      <tls>
        <keystore>configuration/ssl/ctl.jks</keystore>
        <keystore-type>JKS</keystore-type>
        <keystore-path-type>PATH</keystore-path-type>
        <keystore-password>opendaylight</keystore-password>
        <truststore>configuration/ssl/truststore.jks</truststore>
        <truststore-type>JKS</truststore-type>
        <truststore-path-type>PATH</truststore-path-type>
        <truststore-password>opendaylight</truststore-password>
        <certificate-password>opendaylight</certificate-password>
      </tls>

    </module>
```

Configuring openvswitch SSL

To configure openswitch SSL

1. Set ovs ssl options.

```
sudo ovs-vsctl set-ssl \  
    /etc/openvswitch/sc-privkey.pem \  
    /etc/openvswitch/sc-cert.pem \  
    /var/lib/openvswitch/pki/controllerca/cacert.pem
```

1. Start a mininet with SSL connections to the ODL controller.

a. Open the `ssl_switch_tests.py` file

```
#!/usr/bin/python  
from mininet.net import Mininet  
from mininet.node import Controller, RemoteController  
from mininet.cli import CLI  
from mininet.log import setLogLevel, info  
  
def emptyNet():  
    net = Mininet( controller=RemoteController )  
    net.addController( 'c0' )  
    h1 = net.addHost( 'h1' )  
    h2 = net.addHost( 'h2' )  
    s1 = net.addSwitch( 's1' )  
    net.addLink( h1, s1 )  
    net.addLink( h2, s1 )  
  
    net.start()  
    s1.cmd( 'ovs-vsctl set-controller s1 ssl:YOURODLCONTROLLERIPADDRESS:6633' )  
  
    CLI( net )  
    net.stop()  
  
if __name__ == '__main__':  
    setLogLevel( 'info' )  
    emptyNet()
```

1. Start mininet with TLS:

```
chmod +x ssl_switch_test.py  
sudo ./ssl_switch_test.py
```

Configuring a hardware switch with TLS

The configuration example that follows uses a Brocade MLX device. **To configure a hardware switch**

1. Set up a tftp server.

```
telnet@NetIron MLX-4 Router#enable  
<enter config password>.
```

1. Copy the sc-cert.pem and sc-privkey.pem files to the tftp sever on the controller:

```
telnet@NetIron MLX-4 Router(config)#copy tftp flash 10.0.0.1 sc-cert.pem  
client-certificate  
telnet@NetIron MLX-4 Router(config)#copy tftp flash 10.0.0.1 sc-privkey.pem  
client-private-key  
telnet@NetIron MLX-4 Router(config)#openflow controller ip-address 10.0.0.1
```


Converters (semantic level)

Converters aid communication between applications and devices by making possible the communication between southbound APIs and their North-bound counterparts. They translate MD-SAL models to OFJava-API models. The default set of converters reside in: `openflowplugin/src/main/java/org/.opendaylight/openflowplugin/openflow/md/core/sal/convertor`

Converters act upon models from and to the MD-SAL. Inputs for **action converter from MD-SAL** are instances of the MD-SAL model: for example, in the case of action, `OutputActionCase`. The output contains OFJava-API models of Action transferred from applications to devices. Working in reverse, **action converters to MD-SAL** translate OFJava-API models (Action) to MD-SAL models (Action).

After a vendor bundle is activated, converters are registered with the OF plugin so that they can work. Registration is based on the augmentation type and version. Once the converters are registered, the OF Plugin can convert MD-SAL action to OF Java actions.

Approaches to action conversion

The sample that follows shows two approaches to converting action (`ActionConvertor.java`). The first approach relies on a key field in a generalExtension augmentation. The second approach directly creates the converter lookup key out of the action type.

```
else if (action instanceof GeneralExtensionGrouping) {
    /**
     * TODO: EXTENSION PROPOSAL (action, MD-SAL to OFJava)
     * - we might need sessionContext as converter input
     */
    GeneralExtensionGrouping extensionCaseGrouping =
        (GeneralExtensionGrouping) action;
    Extension extAction = extensionCaseGrouping.getExtension();
    ConverterExtensionKey<? extends ExtensionKey> key = new
        ConverterExtensionKey<>(extensionCaseGrouping.getExtensionKey(), version);
    ConvertorToOFJava<Action> convertor =
        OFSessionUtil.getExtensionConvertorProvider().
getConverter(key);
    if (convertor != null) {
        ofAction = convertor.convert(extAction);
    }
} else {
    // try vendor codecs
    TypeVersionKey<org.opendaylight.yang.gen.v1.urn.opendaylight.
action.types.rev131112.action.Action> key =
        new TypeVersionKey<>(
            (Class<? extends org.opendaylight.yang.
gen.v1.urn.opendaylight.action.types.rev131112.action.Action>) action.
getImplementedInterface(),
            version);
    ConvertorActionToOFJava<org.opendaylight.yang.gen.v1.urn.
opendaylight.action.types.rev131112.action.Action, Action> convertor =
```

```
OFSessionUtil.getExtensionConvertorProvider().
getConverter(key);
    if (convertor != null) {
        ofAction = convertor.convert(action);
    }
}
```

Encoders and decoders for augment messages (low level)

Augments are encoded using encoders. Vendor bundles register the encoders so that the OpenFlow Java Library can support the vendor actions. Default sets of encoders and decoders reside in `/openflow-protocol-impl/src/main/java/org/.opendaylight/openflowjava/protocol/impl/serialization` and `/openflow-protocol-impl/src/main/java/org/.opendaylight/openflowjava/protocol/impl/deserialization`. The OF plugin uses encoders to create the binary (wire protocol) form of a message object, and write it to the buffer.

Decoders on the other hand are responsible for the following tasks:

- Read binary buffer
- Detect the type of message (encoded in the header)
- Create the corresponding objects, and populate them with values from the buffer

Master decoder

Vendor decoders cannot be directly registered if the actual message type is outside the general header, and only vendor-provided logic can take decisions. Then a master decoder, which is also provided by the vendor, is used. The master decoder contains logic to register decoders and to distinguish between vendor actions. The same work-flow persists: the lookup decoder by key containing version, actionClass, vendorActionSubtype. (For example, the experimenter action makes it appear as if all actions from one vendor have the same header, and the subtype of the actual action lies somewhere further in the buffer.)

The OFJava extensions provide the space for registering vendor encoders and master decoders. They also provide the lookup mechanism to pick the right decoder or encoder for work with a message or buffer.

Overload protection in the OF Plugin

Overload protection in the OpenFlow (OF) Plugin works in the following way:

1. The `ConnectionConductor` is the source from where all incoming messages are pushed to queues for asynchronous processing. It is the part of the OF Plugin closest to OFJava, and has `on*Message` methods (listeners to incoming messages). The `ConnectionConductorImpl` pushes messages to the `QueueKeeper`. Every `ConnectionConductor` has a local instance of the `QueueKeeper`. The `QueueKeeper` has two queues:
 - Unordered queues (for packetIn messages)
 - Ordered queues (for other messages) Both queue types are limited and blocking.

2. If a particular queue is full, the messages pushed to it will be dropped. Upon a successful push, the harvester is pinged to be roused from hibernation.
3. A QueueZipper wraps the two queues, and provides the poll method. This poll method rotates regularly through the underlying queues. If the currently polled queue is empty, it polls the next queue. (See QueueKeeperFairImpl).
4. Each QueueKeeper gets registered by the QueueKeeperHarvester. The Harvester runs upon one thread; iterates through all the registered QueueKeepers; and polls them. The polled message is then queued into the QueueProcessor. If all the registered queueKeepers are empty, the harvester hibernates.
5. At the QueueProcessor are several threads translating messages from OFJava-API models to MD-SAL models (preserving order). The QueueProcessor uses two threadPools:
 - One threadPool to process the queue items
 - Another threadPool (containing one thread) to publish messages to the MD-SAL

A queue gets filled for different reasons:

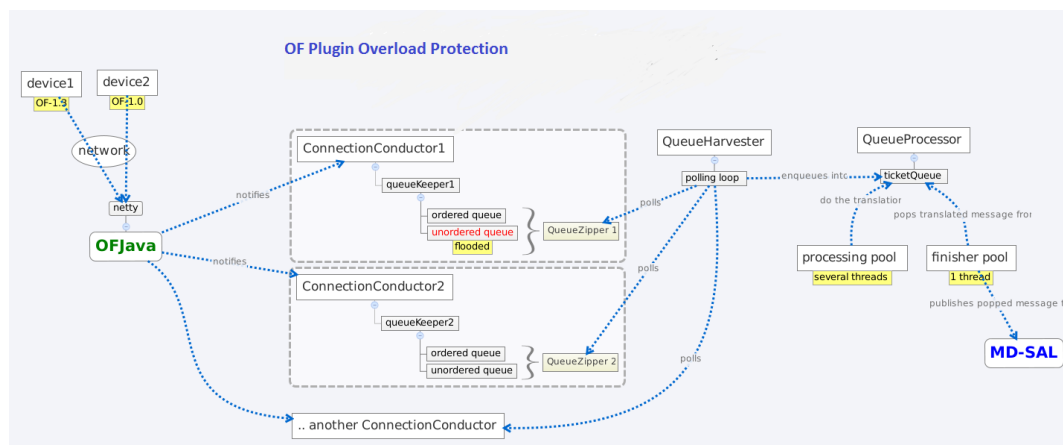
- The MD-SAL is overloaded.
- A node is flooding, or something has generally slowed down the processing pipeline. If the queue in the QueueProcessor is full, it blocks the harvester. If the harvester is blocked, the queues in the QueueKeeper will not be emptied.



Note

The current implementation of the feature offers no checking of the memory or CPU load to actively throttle messages.

Figure 13.10. Overload protection



Effects of overload protection

- When a node floods the controller, it will not block messages from other nodes.

- The processing of messages is fair: *Floody* node messages are neither prioritized, nor do they infest queues outside the ConnectionConductor.
- Memory is not exhausted on the controller side as messages gets dropped immediately upon an unsuccessful push to the local queue.
- The functionality cannot create back pressure at the netty level. Pressure affects the echo message, and might cause a connection close action on the switch side.

14. OVSDB Integration

Table of Contents

| | |
|--|-----|
| OpenDaylight OVSDB integration | 254 |
| Building and running OVSDB | 257 |
| OVSDB integration design | 260 |
| OpenDaylight OVSDB southbound plugin architecture and design | 260 |
| OVSDB southbound plugin | 261 |
| Connection service | 261 |
| Network Configuration Service | 263 |
| OpenDaylight OVSDB Developer Getting Started Video Series | 267 |
| OVSDB integration: New features | 267 |

The Open vSwitch database (OVSDB) Plugin component for OpenDaylight implements the OVSDB [RFC 7047](#) management protocol that allows the southbound configuration of vSwitches. The component comprises a library and various plugin usages. The OVSDB protocol uses JSON/RPC calls to manipulate a physical or virtual switch that has OVSDB attached to it. Almost all vendors support OVSDB on various hardware platforms. The OpenDaylight controller uses the native OVSDB implementation to manipulate the Open vSwitch database.



Note

Read the OVSDB User Guide before you begin development.

OpenDaylight OVSDB integration

For information on how the GRE-endpoint destination IP address is communicated to the controller, see [Neutron and ODL interactions](#).

The OpenStack integration architecture uses the following technologies:

- [RFC 7047](#) and [The Open vSwitch Database Management Protocol](#)
- OpenFlow v1.3
- OpenStack Neutron ML2 Plugin

Getting the code

```
export ODL_USERNAME=<username for the account you created at OpenDaylight>
git clone ssh://{ODL_USERNAME}@git.opendaylight.org:29418/ovsdb.git;(cd
ovsdb; scp -p -P 29418 ${ODL_USERNAME}@git.opendaylight.org:hooks/commit-
msg .git/hooks/;chmod 755 .git/hooks/commit-msg;git config remote.origin.push
HEAD:refs/for/master)
```

OpenDaylight Mechanism Driver for Openstack Neutron ML2

This code is a part of OpenStack and is available at: https://github.com/openstack/neutron/blob/master/neutron/plugins/ml2/drivers/mechanism_odl.py

To make changes to this code, please read about [Neutron Development](#).

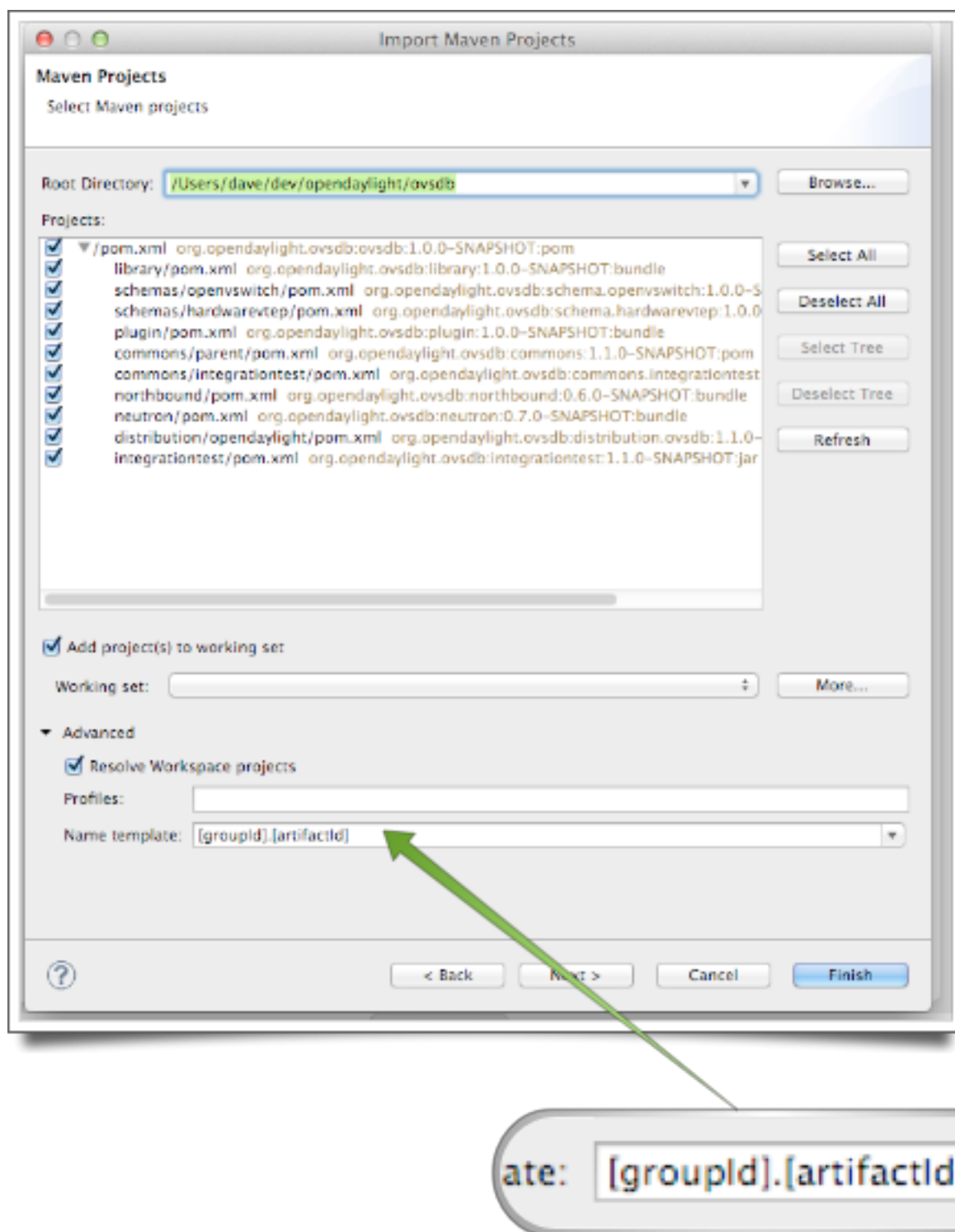
Before submitting the code, run the following tests:

```
tox -e py27
tox -e pep8
```

Importing the code in to Eclipse or IntelliJ

Check out either of the following:

- [Getting started with Eclipse](#)
- [Developing with IntelliJ](#)

Figure 14.1. Avoid conflicting project names

- To ensure that a project in Eclipse does not have a conflicting name in the workspace, select Advanced > Name Template > [groupId].[artifactId] when importing the project.

Browsing the code

The code is mirrored to [GitHub](#) to make reading code online easier.

Source code organization

The OVSDb project generates 3 x Karaf Modules:

```
ovsdb -- all ovsdb related artifacts
of-nxm-extensions -- openflow nxm extensions
ovs-sfc -- openflow service chaining function
```

Both of-nxm-extensions and ovs-sfc are expected to be moved out of the ovsdb source tree in the future.

Following are a brief descriptions on directories you will find at the root ovsdb/ directory:

- *commons* contains the parent POM file for Maven project which is used to get consistency of settings across the project.
- *distribution* contains the OVSDb distribution. For OSGI, this is the latest Virtualization Edition pulled from the Integration project with your local OVSDb artifacts added. This gives developers the ability to run the controller for testing. For Karaf, this is the latest Karaf bundle pulled from the Integration project, with your local OVSDb Karaf bundles mentioned above.
- *openstack* contains the northbound handlers for Neutron used by OVSDb, as well as their providers.
- *resources* contains useful scripts, How-To's and other resources.
- *schemas* contains the OVSDb schemas that are implemented in ODL.
- *utils* contains helper functions used for handling MD-SAL OpenFlow implementation.

Building and running OVSDb

Prerequisites

- JDK 1.7+
- Maven 3+

Building a Karaf feature and deploying it in an Opendaylight Karaf distribution

This is a new method for Opendaylight distribution wherein there are no defined editions such as Base, Virtualization, or SP editions. The end-customer can choose to deploy the required feature based on user deployment needs.

1. From the root ovsdb/ directory, run **mvn clean install**.

2. Unzip the distribution-karaf-<VERSION_NUMBER>-SNAPSHOT.zip file created from step 1 in the directory ovsdb/distribution/opendaylight-karaf/target:

```
unzip distribution-karaf-<VERSION_NUMBER>-SNAPSHOT.zip
```

Downloading OVSDb's Karaf distribution

Instead of building, you can download the latest OVSDb distribution from the Nexus server. The link for that is:

```
http://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/org/opendaylight/ovsdb/distribution.ovsdb/1.2.0-SNAPSHOT/
```

Running Karaf feature from OVSDb's Karaf distribution

1. Start ODL, from the unzipped directory

```
bin/karaf
```

1. Once karaf has started, and you see the Opendaylight ascii art in the console, the last step is to start the OVSDb plugin framework with the following command in the karaf console:

```
feature:install odl-ovsdb-openstack
```

For ovsdb northbound, you will also need to invoke

```
feature:install odl-ovsdb-northbound
```

Sample output from the Karaf console

```
opendaylight-user@root>feature:list | grep -i ovsdb
odl-ovsdb-all | 1.0.0-SNAPSHOT | | ovsdb-1.0.0-SNAPSHOT
OpenDaylight :: OVSDb :: all
odl-ovsdb-library | 1.0.0-SNAPSHOT | x | ovsdb-1.0.0-SNAPSHOT
OVSDb :: Library
odl-ovsdb-schema-openvswitch | 1.0.0-SNAPSHOT | x | ovsdb-1.0.0-SNAPSHOT
OVSDb :: Schema :: Open_vSwitch
odl-ovsdb-schema-hardwarevtep | 1.0.0-SNAPSHOT | x | ovsdb-1.0.0-SNAPSHOT
OVSDb :: Schema :: hardware_vtep
odl-ovsdb-plugin | 1.0.0-SNAPSHOT | x | ovsdb-1.0.0-SNAPSHOT
OpenDaylight :: OVSDb :: Plugin
odl-ovsdb-northbound | 0.6.0-SNAPSHOT | | ovsdb-1.0.0-SNAPSHOT
OpenDaylight :: OVSDb :: Northbound
odl-ovsdb-openstack | 1.0.0-SNAPSHOT | x | ovsdb-1.0.0-SNAPSHOT
OpenDaylight :: OVSDb :: OpenStack Network Virtual
odl-ovsdb-ovssfc | 0.0.1-SNAPSHOT | | ovsdb-0.0.1-SNAPSHOT
OpenDaylight :: OVSDb :: OVS Service Function Chain
odl-openflow-nxm-extensions | 0.0.3-SNAPSHOT | x | ovsdb-0.0.3-SNAPSHOT
```

```
OpenDaylight :: Openflow :: Nicira Extensions
```

Testing patches

It is recommended that you test your patches locally before submission.

Neutron integration

To test patches to the Neutron integration, you need a [Multi-Node Devstack Setup](#). The ``resources`` folder contains sample ``local.conf`` files.

Open vSwitch

To test patches to the library, you will need a working [Open vSwitch](#). Packages are available for most Linux distributions. If you would like to run multiple versions of Open vSwitch for testing you can use [docker-ovs](#) to run Open vSwitch in [Docker](#) containers.

Mininet

[Mininet](#) is another useful resource for testing patches. Mininet creates multiple Open vSwitches connected in a configurable topology.

Vagrant

The Vagrant file in the root of the OVSDb source code provides an easy way to create VMs for tests.

- To install Vagrant on your machine, follow the steps at: [Installing Vagrant](#).

Testing with Devstack

1. Start the controller.

```
vagrant up devstack-control
vagrant ssh devstack-control
cd devstack
./stack.sh
```

1. Run the following:

```
vagrant up devstack-compute-1
vagrant ssh devstack-compute-1
cd devstack
./stack.sh
```

1. To start testing, create a new VM.

```
nova boot --flavor m1.tiny --image $(nova image-list | grep 'cirros-0.3.1-x86_64-uec\s' | awk '{print $2}') --nic net-id=$(neutron net-list | grep private | awk '{print $2}') test
```

To create three, use the following:

```
nova boot --flavor m1.tiny --image $(nova image-list | grep 'cirros-0.3.1-x86_64-uec\s' | awk '{print $2}') --nic net-id=$(neutron net-list | grep private | awk '{print $2}') --num-instances 3 test
```

To get a mininet installation for testing:

```
vagrant up mininet  
vagrant ssh mininet
```

1. Use the following to clean up when finished:

```
vagrant destroy
```

OVSDB integration design

Resources

See the following:

- [Network Heresy](#)

See the OVSDB YouTube Channel for getting started videos and other tutorials:

- [ODL OVSDB Youtube Channel](#)
- [Mininet OVSDB Tutorial](#)

OpenDaylight OVSDB southbound plugin architecture and design

OpenVSwitch (OVS) is generally accepted as the unofficial standard for Virtual Switching in the Open hypervisor based solutions. Every other Virtual Switch implementation, propriety or otherwise, uses OVS in some form. For information on OVS, see [Open vSwitch](#).

In Software Defined Networking (SDN), controllers and applications interact using two channels: OpenFlow and OVSDB. OpenFlow addresses the forwarding-side of the OVS functionality. OVSDB, on the other hand, addresses the management-plane. A simple and concise overview of Open Virtual Switch Database(OVSDB) is available at: <http://networkstatic.net/getting-started-ovsdb/>

Overview of OpenDaylight Controller architecture

The OpenDaylight controller platform is designed as a highly modular and plugin based middleware that serves various network applications in a variety of use-cases. The modularity is achieved through the Java OSGi framework. The controller consists of many Java OSGi bundles that work together to provide the required controller functionalities.

The bundles can be placed in the following broad categories:

- Network Service Functional Modules (Examples: Topology Manager, Inventory Manager, Forwarding Rules Manager, and others)
- NorthBound API Modules (Examples: Topology APIs, Bridge Domain APIs, Neutron APIs, Connection Manager APIs, and others)

- Service Abstraction Layer(SAL)- (Inventory Services, DataPath Services, Topology Services, Network Config, and others)
- SouthBound Plugins (OpenFlow Plugin, OVSDb Plugin, OpenDove Plugin, and others)
- Application Modules (Simple Forwarding, Load Balancer)

Each layer of the Controller architecture performs specified tasks, and hence aids in modularity. While the Northbound API layer addresses all the REST-Based application needs, the SAL layer takes care of abstracting the SouthBound plugin protocol specifics from the Network Service functions.

Each of the SouthBound Plugins serves a different purpose, with some overlapping. For example, the OpenFlow plugin might serve the Data-Plane needs of an OVS element, while the OVSDb plugin can serve the management plane needs of the same OVS element. As the Openflow Plugin talks OpenFlow protocol with the OVS element, the OVSDb plugin will use OVSDb schema over JSON-RPC transport.

OVSDb southbound plugin

The [Open vSwitch Database Management Protocol-draft-02](#) and [Open vSwitch Manual](#) provide theoretical information about OVSDb. The OVSDb protocol draft is generic enough to lay the groundwork on Wire Protocol and Database Operations, and the OVS Manual currently covers 13 tables leaving space for future OVS expansion, and vendor expansions on proprietary implementations. The OVSDb Protocol is a database records transport protocol using JSON RPC1.0. For information on the protocol structure, see [Getting Started with OVSDb](#). The OpenDaylight OVSDb southbound plugin consists of one or more OSGi bundles addressing the following services or functionalities:

- Connection Service - Based on Netty
- Network Configuration Service
- Bidirectional JSON-RPC Library
- OVSDb Schema definitions and Object mappers
- Overlay Tunnel management
- OVSDb to OpenFlow plugin mapping service
- Inventory Service

Connection service

One of the primary services that most southbound plugins provide to SAL in OpenDaylight and NSF is Connection Service. The service provides protocol specific connectivity to network elements, and supports the connectivity management services as specified by the OpenDaylight Connection Manager. The connectivity services include:

- Connection to a specified element given IP-address, L4-port, and other connectivity options (such as authentication,...)

- Disconnection from an element
- Handling Cluster Mode change notifications to support the OpenDaylight Clustering/High-Availability feature

By default, the `ovsdb-server` process running on the hypervisor listens on TCP port 6632 (This is configurable.). The Connection Service takes the connectivity parameters from the connection manager, including the IP-address and TCP-Port for connections. Owing to the many benefits it provides, Connection Service will use the Netty framework (<http://netty.io/>) for connectivity purposes. Every successful connection to a network element will result in a Node object (Refer to OpenDaylight SAL Node.java) with the type = "OVSDB" and value = User-Readable Name of the Connection as specified by the Connection Manager. This Node object is returned to the OpenDaylight Connection Manager and the application that invoked the `Connect()` functionality.

```
IPluginInConnectionService : public Node connect(String identifier,
Map<ConnectionConstants, String> params)
```

Any subsequent interaction with this network element through any of the SAL services (Connection, Configuration, and others) will be by means of this Node Object. This Node object will be added to the Inventory maintained and managed by the Inventory Service of the plugin. The Node object will also assist with the OVSDB to Openflow mapping.

The Node and its "Name" holds the key to the stateful Netty Socket handler maintained under the Connection Object created during the `connect()` call. The Channel concept of the Netty framework provides the much needed abstraction on the pipelining. With this Channel Pipelining and the asynchronous event handling, the message handling process gets better streamlined and understood. It also makes easier the replacement or manipulation of the pipeline functions in a more controlled fashion.

Figure 14.2. Connection to OVSDB server

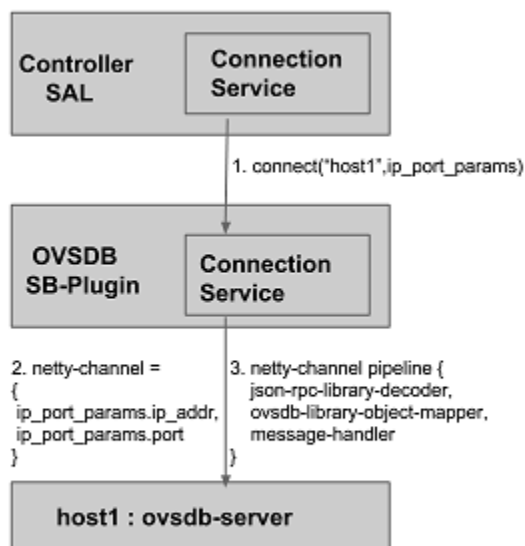
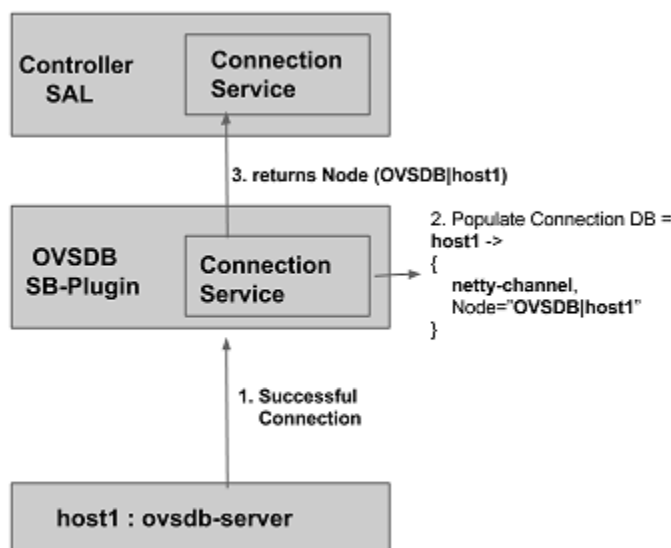


Figure 14.3. Successful connection handling

Network Configuration Service

The goal of the OpenDaylight Network Configuration services is to provide complete management plane solutions needed to successfully install, configure, and deploy the various SDN based network services. These are generic services which can be implemented in part or full by any south-bound protocol plugin. The south-bound plugins can be either of the following:

- The new network virtualization protocol plugins such as OVSDB JSON-RPC
- The traditional management protocols such as SNMP or any others in the middle.

The above definition, and more information on Network Configuration Services, is available at : https://wiki.opendaylight.org/view/OpenDaylight_Controller:NetworkConfigurationServices

The current default OVSDB schemas support the Layer2 Bridge Domain services as defined in the Networkconfig.bridgedomain component.

- Create Bridge Domain: `createBridgeDomain(Node node, String bridgeIdentifier, Map<ConfigConstants, Object> params)`
- Delete Bridge Domain: `deleteBridgeDomain(Node node, String bridgeIdentifier)`
- Add configurations to a Bridge Domain: `addBridgeDomainConfig(Node node, String bridgeIdentifier, Map<ConfigConstants, Object> params)`

- Delete Bridge Domain Configuration: `removeBridgeDomainConfig(Node node, String bridgeIdentifier, Map<ConfigConstants, Object> params)`
- Associate a port to a Bridge Domain: `addPort(Node node, String bridgeIdentifier, String portIdentifier, Map<ConfigConstants, Object> params);`
- Disassociate a port from a Bridge Domain: `deletePort(Node node, String bridgeIdentifier, String portIdentifier)`
- Add configurations to a Node Connector / Port: `addPortConfig(Node node, String bridgeIdentifier, String portIdentifier, Map<ConfigConstants, Object> params)`
- Remove configurations from a Node Connector: `removePortConfig(Node node, String bridgeIdentifier, String portIdentifier, Map<ConfigConstants, Object> params)`

The above services are defined as generalized entities in SAL in order to ensure their compatibility with all relevant southBound plugins equally. Hence, the OVSDb plugin must derive appropriate specific configurations from a generalized request. For example: `addPort()` or `addPortConfig()` SAL service call takes in a `params` option which is a `Map` structure with a Constant Key. These `ConfigConstants` are defined in SAL network configuration service:

```
public enum ConfigConstants {
    TYPE("type"),
    VLAN("Vlan"),
    VLAN_MODE("vlan_mode"),
    TUNNEL_TYPE("Tunnel Type"),
    SOURCE_IP("Source IP"),
    DEST_IP("Destination IP"),
    MACADDRESS("MAC Address"),
    INTERFACE_IDENTIFIER("Interface Identifier"),
    MGMT("Management"),
    CUSTOM("Custom Configurations");
}
```

These are mapped to the appropriate OVSDb configurations. So, if the request is to create a VXLAN tunnel with `src-ip=x.x.x.x`, `dst-ip=y.y.y.y`, then the `params` `Map` structure may contain:

```
{
  TYPE = "tunnel",
  TUNNEL_TYPE = "vxlan",
  SOURCE_IP="x.x.x.x",
  DEST_IP="y.y.y.y"
}
```



Note

All of the APIs take in the `Node` parameter which is the `Node` value returned by the `connect()` method explained in [the section called "Connection service" \[261\]](#).

Bidirectional JSON-RPC library

The OVSDb plugin implements a Bidirectional JSON-RPC library. It is easy to design the library as a module that manages the Netty connection towards the Element.

The main responsibilities of this Library are:

- Demarshal and marshal JSON Strings to JSON objects
- Demarshal and marshal JSON Strings from and to the Network Element.

OVSDB Schema definitions and Object mappers

The OVSDB Schema definitions and Object Mapping layer sits above the JSON-RPC library. It maps the generic JSON objects to OVSDB schema POJOs (Plain Old Java Object) and vice-versa. This layer mostly provides the Java Object definition for the corresponding OVSDB schema (13 of them) and also will provide much more friendly API abstractions on top of these object data. This helps in hiding the JSON semantics from the functional modules such as Configuration Service and Tunnel management.

On the demarshaling side, the mapping logic differentiates the Request and Response messages as follows :

- Request messages are mapped by its "method"
- Response messages are mapped by their IDs which were originally populated by the Request message. The JSON semantics of these OVSDB schema is quite complex. The following figures summarize two of the end-to-end scenarios:

Figure 14.4. End-to-end handling of a Create Bridge request

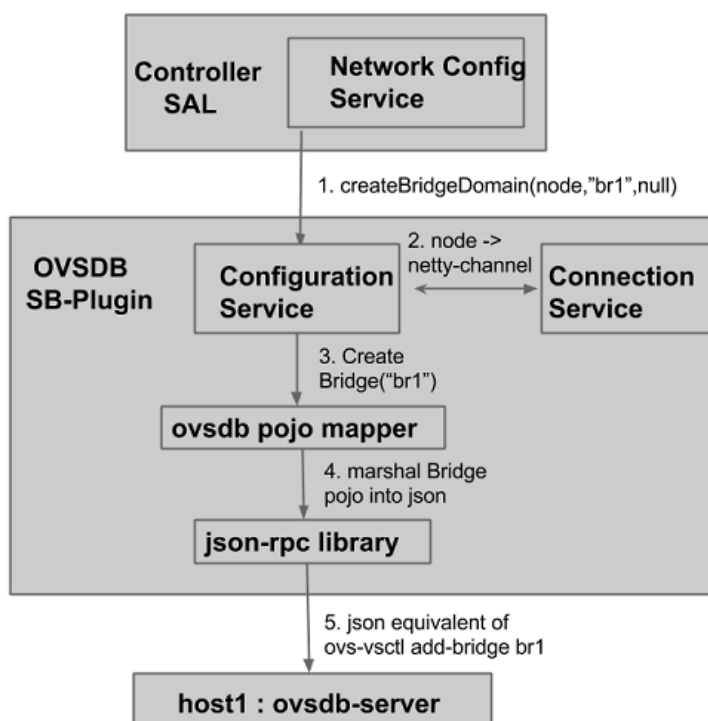
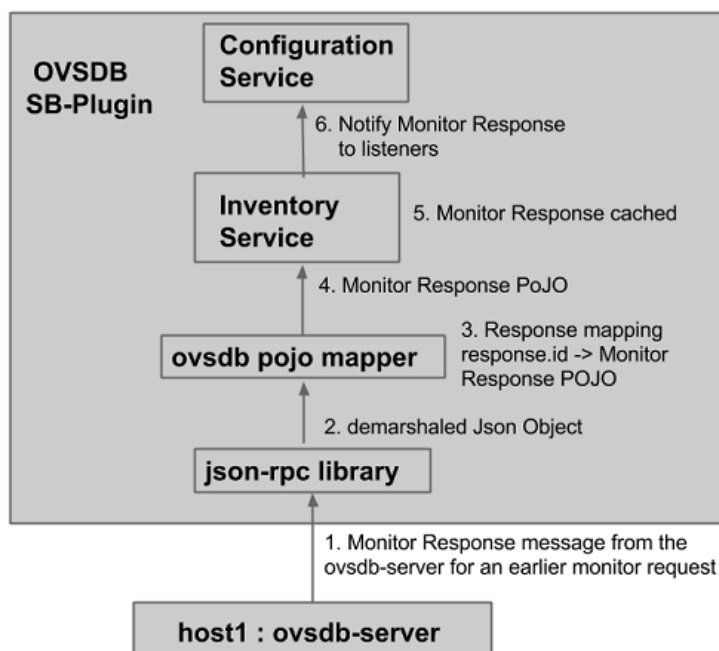


Figure 14.5. End-to-end handling of a monitor response

Overlay tunnel management

Network Virtualization using OVS is achieved through Overlay Tunnels. The actual Type of the Tunnel may be GRE, VXLAN, or STT. The differences in the encapsulation and configuration decide the tunnel types. Establishing a tunnel using configuration service requires just the sending of OVSDb messages towards the ovssdb-server. However, the scaling issues that would arise on the state management at the data-plane (using OpenFlow) can get challenging. Also, this module can assist in various optimizations in the presence of Gateways. It can also help in providing Service guarantees for the VMs using these overlays with the help of underlay orchestration.

OVSDb to OpenFlow plugin mapping service

The connect() of the ConnectionService would result in a Node that represents an ovssdb-server. The CreateBridgeDomain() Configuration on the above Node would result in creating an OVS bridge. This OVS Bridge is an OpenFlow Agent for the OpenDaylight OpenFlow plugin with its own Node represented as (example) OF|xxx.yy.zzz. Without any help from the OVSDb plugin, the Node Mapping Service of the Controller platform would not be able to map the following:

```
{OVSDb_NODE + BRIDGE_IDENTIFIER} <---> {OF_NODE}.
```

Without such mapping, it would be extremely difficult for the applications to manage and maintain such nodes. This Mapping Service provided by the OVSDb plugin would essentially

help in providing more value added services to the orchestration layers that sit atop the Northbound APIs (such as OpenStack).

Inventory service

Inventory Service provides a simple database of all the nodes managed and maintained by the OVSDb plugin on a given controller. For optimization purposes, it can also provide enhanced services to the OVSDb to OpenFlow mapping service by maintaining the following mapping owing to the static nature of this operation.

```
{OVSDb_NODE + BRIDGE_IDENTIFIER} <---> {OF_NODE}
```

OpenDaylight OVSDb Developer Getting Started Video Series

The video series were started to help developers bootstrap into OVSDb development.

- [OpenDaylight OVSDb Developer Getting Started](#)
- [OpenDaylight OVSDb Developer Getting Started - Northbound API Usage](#)
- [OpenDaylight OVSDb Developer Getting Started - Java APIs](#)
- [OpenDaylight OVSDb Developer Getting Started - OpenStack Integration OpenFlow v1.0](#)

Other developer tutorials

- [OVSDb OpenFlow v1.3 Neutron ML2 Integration](#)
- [Open vSwitch Database Table Explanations and Simple Jackson Tutorial](#)

OVSDb integration: New features

Schema independent library

The OVS connection is a node which can have multiple databases. Each database is represented by a schema. A single connection can have multiple schemas. OVSDb supports multiple schemas. Currently, these are two schemas available in the OVSDb, but there is no restriction on the number of schemas. Owing to the Northbound v3 API, no code changes in ODL are needed for supporting additional schemas.

Schemas:

- openvswitch : Schema wrapper that represents <http://openvswitch.org/ovs-vswitchd.conf.db.5.pdf>
- hardwarevtep: Schema wrapper that represents <http://openvswitch.org/docs/vtep.5.pdf>

Northbound API v3

OVSDb supports Northbound API v3 which allows external access to all ODL OVSDb databases or schemas. The general syntax for that API follows this format:

```
http://{controllerHost}:{controllerPort}/ovsdb/nb/v3/node/{OVS|HOST}/database
```

For more information on Northbound REST API see: https://docs.google.com/spreadsheets/d/11Rp5KSNTcrvOD4HadCnXDCUdJq_TZ5RgoQ6qSHf_xkw/edit?usp=sharing

The key differences between Northbound API v2 and v3 include:

- Support for schema independence
- Formal restful style API, which includes consistent URL navigation for nodes and tables
- Ability to create interfaces and ports within a single rest call. To allow that, the JSON in the body can include distinct parts like interface and port

Port security

Based on the fact that security rules can be obtained from a port object, OVSDb can apply Open Flow rules. These rules will match on what types of traffic the Openstack tenant VM is allowed to use.

Support for security groups is very experimental. There are limitations in determining the state of flows in the Open vSwitch. See [Open vSwitch and the Intelligent Edge](#) from Justin Petit for a deep dive into the challenges we faced creating a flow based port security implementation. The current set of rules that will be installed only supports filtering of the TCP protocol. This is because via a Nicira TCP_Flag read we can match on a flows TCP_SYN flag, and permit or deny the flow based on the Neutron port security rules. If rules are requested for ICMP and UDP, they are ignored until greater visibility from the Linux kernel is available as outlined in the OpenStack presentation mentioned earlier.

Using the port security groups of Neutron, one can add rules that restrict the network access of the tenants. The OVSDb Neutron integration checks the port security rules configured, and apply them by means of openflow rules.

Through the ML2 interface, Neutron security rules are available in the port object, following this scope: Neutron Port # Security Group # Security Rules.

The current rules are applied on the basis of the following attributes: ingress/egress, tcp protocol, port range, and prefix.

OpenStack workflow

1. Create a stack.
2. Add the network and subnet.
3. Add the Security Group and Rules.



Note

This is no different than what users normally do in regular openstack deployments.

```
neutron security-group-create group1 --description "Group 1"
```

```
neutron security-group-list
neutron security-group-rule-create --direction ingress --protocol tcp group1
```

1. Start the tenant, specifying the security-group.

```
nova boot --flavor ml.tiny \
--image $(nova image-list | grep 'cirros-0.3.1-x86_64-uec\s' | awk '{print $2}') \
--nic net-id=$(neutron net-list | grep 'vxlan2' | awk '{print $2}') vxlan2 \
--security-groups group1
```

Examples: Rules supported

```
neutron security-group-create group2 --description "Group 2"
neutron security-group-rule-create --direction ingress --protocol tcp --port-range-min 54 group2
neutron security-group-rule-create --direction ingress --protocol tcp --port-range-min 80 group2
neutron security-group-rule-create --direction ingress --protocol tcp --port-range-min 1633 group2
neutron security-group-rule-create --direction ingress --protocol tcp --port-range-min 22 group2
```

```
neutron security-group-create group3 --description "Group 3"
neutron security-group-rule-create --direction ingress --protocol tcp --remote-ip-prefix 10.200.0.0/16 group3
```

```
neutron security-group-create group4 --description "Group 4"
neutron security-group-rule-create --direction ingress --remote-ip-prefix 172.24.0.0/16 group4
```

```
neutron security-group-create group5 --description "Group 5"
neutron security-group-rule-create --direction ingress --protocol tcp group5
neutron security-group-rule-create --direction ingress --protocol tcp --port-range-min 54 group5
neutron security-group-rule-create --direction ingress --protocol tcp --port-range-min 80 group5
neutron security-group-rule-create --direction ingress --protocol tcp --port-range-min 1633 group5
neutron security-group-rule-create --direction ingress --protocol tcp --port-range-min 22 group5
```

```
neutron security-group-create group6 --description "Group 6"
neutron security-group-rule-create --direction ingress --protocol tcp --remote-ip-prefix 0.0.0.0/0 group6
```

```
neutron security-group-create group7 --description "Group 7"
neutron security-group-rule-create --direction egress --protocol tcp --port-range-min 443 --remote-ip-prefix 172.16.240.128/25 group7
```

Reference gist:<https://gist.github.com/anonymous/1543a410d57f491352c8>[Gist]

Security group rules supported in ODL

The following rules formata are supported in the current implementation. The direction (ingress/egress) is always expected. Rules are implemented such that tcp-syn packets that do not satisfy the rules are dropped.

| Proto | Port | IP Prefix |
|-------|------|-----------|
| TCP | x | x |

| Proto | Port | IP Prefix |
|-------|------|-----------|
| Any | Any | x |
| TCP | x | Any |
| TCP | Any | Any |

Limitations

- Soon, conntrack will be supported by OVS. Until then, TCP flags are used as way of checking for connection state. Specifically, that is done by matching on the TCP-SYN flag.
- The param `--port-range-max` in `security-group-rule-create` is not used until the implementation uses conntrack.
- No UDP/ICMP specific match support is provided.
- No IPv6 support is provided.

L3 forwarding

OVSDb extends support for the usage of an ODL-Neutron-driver so that OVSDb can configure OF 1.3 rules to route IPv4 packets. The driver eliminates the need for the router of the L3 Agent. In order to accomplish that, OVS 2.1 or a newer version is required. OVSDb also supports inbound/outbound NAT, floating IPs.

Starting OVSDb and OpenStack

1. Build or download OVSDb distribution, as mentioned in [building a Karaf feature section](#).
2. [Install Vagrant](#).

1. Enable the L3 Forwarding feature:

```
echo 'ovsdb.l3.fwd.enabled=yes' >> ./opendaylight/configuration/config.ini
echo 'ovsdb.l3gateway.mac=${GATEWAY_MAC}' >> ./configuration/config.ini
```

1. Run the following commands to get the odl neutron drivers:

```
git clone https://github.com/dave-tucker/odl-neutron-drivers.git
cd odl-neutron-drivers
vagrant up devstack-control devstack-compute-1
```

1. Use ssh to go to the control node, and clone odl-neutron-drivers again:

```
vagrant ssh devstack-control
git clone https://github.com/dave-tucker/odl-neutron-drivers.git
cd odl-neutron-drivers
sudo python setup.py install
*leave this shell open*
```

1. Start odl, as mentioned in [running Karaf feature section](#).
2. To see processing of neutron event related to L3, do this from prompt:

```
log:set debug org.opendaylight.ovsdb.openstack.netvirt.impl.NeutronL3Adapter
```

1. From shell, do one of the following: open on ssh into control node or vagrant ssh devstack-control.

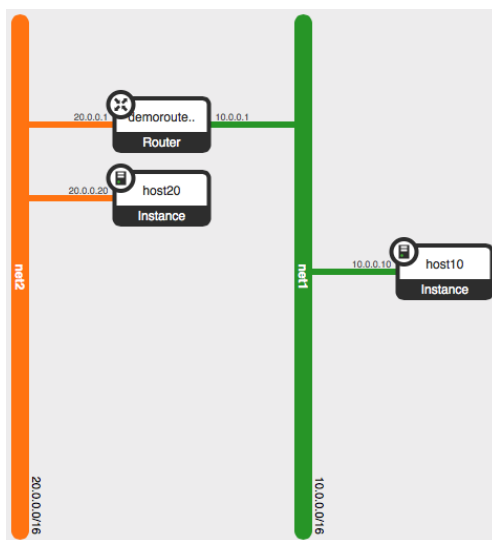
```
cd ~/devstack && ./stack.sh
```

1. From a new shell in the host system, run the following:

```
cd odl-neutron-drivers
vagrant ssh devstack-compute-1
cd ~/devstack && ./stack.sh
```

OpenStack workflow

Figure 14.6. Sample workflow



Use the following steps to set up a workflow like the one shown in figure above.

1. Set up authentication. From shell on stack control or vagrant ssh devstack-control:

```
source openrc admin admin
```

```
rm -f id_rsa_demo* ; ssh-keygen -t rsa -b 2048 -N -f id_rsa_demo
nova keypair-add --pub-key id_rsa_demo.pub demo_key
# nova keypair-list
```

1. Create two networks and two subnets.

```
neutron net-create net1 --tenant-id $(keystone tenant-list | grep '\sadmin |
awk '{print $2}') \
--provider:network_type gre --provider:segmentation_id 555
```

```
neutron subnet-create --tenant-id $(keystone tenant-list | grep '\sadmin |
awk '{print $2}') \
net1 10.0.0.0/16 --name subnet1 --dns-nameserver 8.8.8.8
```

```
neutron net-create net2 --tenant-id $(keystone tenant-list | grep '\sadmin |
awk '{print $2}') \
--provider:network_type gre --provider:segmentation_id 556
```



```
neutron subnet-create --tenant-id $(keystone tenant-list | grep '\sadmin |  
awk '{print $2}')) \  
net2 20.0.0.0/16 --name subnet2 --dns-nameserver 8.8.8.8
```

1. Create a router, and add an interface to each of the two subnets.

```
neutron router-create demorouter --tenant-id $(keystone tenant-list | grep '\sadmin | awk '{print $2}'))  
neutron router-interface-add demorouter subnet1  
neutron router-interface-add demorouter subnet2  
# neutron router-port-list demorouter
```

1. Create two tenant instances.

```
nova boot --poll --flavor m1.nano --image $(nova image-list | grep 'cirros-0.3.2-x86_64-uec\s' | awk '{print $2}')) \  
--nic net-id=$(neutron net-list | grep -w net1 | awk '{print $2}'),v4-fixed-ip=10.0.0.10 \  
--availability-zone nova:devstack-control \  
--key-name demo_key host10
```

```
nova boot --poll --flavor m1.nano --image $(nova image-list | grep 'cirros-0.3.2-x86_64-uec\s' | awk '{print $2}')) \  
--nic net-id=$(neutron net-list | grep -w net2 | awk '{print $2}'),v4-fixed-ip=20.0.0.20 \  
--availability-zone nova:devstack-compute-1 \  
--key-name demo_key host20
```

Limitations

- To use this feature, you need OVS 2.1 or newer version.
- Owing to OF limitations, icmp responses due to routing failures, like ttl expired or host unreachable, are not generated.
- The MAC address of the default route is not automatically mapped. In order to route to L3 destinations outside the networks of the tenant, the manual configuration of the default route is necessary. To provide the MAC address of the default route, use `ovsdb.l3gateway.mac` in file `configuration/config.ini` ;
- This feature is Tech preview, which depends on later versions of OpenStack to be used without the provided neutron-driver.
- No IPv6 support is provided.

More information on L3 forwarding:

- odl-neutron-driver: <https://github.com/dave-tucker/odl-neutron-drivers>
- OF rules example: <http://dtucker.co.uk/hack/building-a-router-with-openvswitch.html>

LBaaS

Load-Balancing-as-a-Service (LBaaS) creates an Open vSwitch powered L3-L4 stateless load-balancer in a virtualized network environment so that individual TCP connections destined to a designated virtual IP (VIP) are sent to the appropriate servers (that is to say, serving

app VMs). The load-balancer works in a session-preserving, proactive manner without involving the controller during flow setup.

A Neutron northbound interface is provided to create a VIP which will map to a pool of servers (that is to say, members) within a subnet. The pools consist of members identified by an IP address. The goal is to closely match the API to the OpenStack LBaaS v2 API: http://docs.openstack.org/api/openstack-network/2.0/content/lbaas_ext.html.

Creating an OpenStack workflow

1. Create a subnet.
2. Create a floating VIP A that maps to a private VIP B.
3. Create a Loadbalancer pool X.

```
neutron lb-pool-create --name http-pool --lb-method ROUND_ROBIN --protocol HTTP --subnet-id XYZ
```

1. Create a Loadbalancer pool member Y and associate with pool X.

```
neutron lb-member-create --address 10.0.0.10 --protocol-port 80 http-pool
neutron lb-member-create --address 10.0.0.11 --protocol-port 80 http-pool
neutron lb-member-create --address 10.0.0.12 --protocol-port 80 http-pool
neutron lb-member-create --address 10.0.0.13 --protocol-port 80 http-pool
```

1. Create a Loadbalancer instance Z, and associate pool X and VIP B with it.

```
neutron lb-vip-create --name http-vip --protocol-port 80 --protocol HTTP --subnet-id XYZ http-pool
```

Implementation

The current implementation of the proactive stateless load-balancer was made using "multipath" action in the Open vSwitch. The "multipath" action takes a max_link parameter value (which is same as the number of pool members) as input, and performs a hash of the fields to get a value between (0, max_link). The value of the hash is used as an index to select a pool member to handle that session.

Open vSwitch rules

Assuming that table=20 contains all the rules to forward the traffic destined for a specific destination MAC address, the following are the rules needed to be programmed in the LBaaS service table=10. The programmed rules makes the translation from the VIP to a different pool member for every session.

- Proactive forward rules:

```
sudo ovs-ofctl -O OpenFlow13 add-flow s1 "table=10,reg0=0,ip,nw_dst=10.0.0.5,actions=load:0x1->NXM_NX_REG0[[[]],multipath(symmetric_14, 1024, modulo_n, 4, 0, NXM_NX_REG1[0..12]),resubmit(,10)"
sudo ovs-ofctl -O OpenFlow13 add-flow s1 table=10,reg0=1,nw_dst=10.0.0.5,ip,reg1=0,actions=mod_dl_dst:00:00:00:00:00:10,mod_nw_dst:10.0.0.10,goto_table:20
sudo ovs-ofctl -O OpenFlow13 add-flow s1 table=10,reg0=1,nw_dst=10.0.0.5,ip,reg1=1,actions=mod_dl_dst:00:00:00:00:00:11,mod_nw_dst:10.0.0.11,goto_table:20
```

```
sudo ovs-ofctl -O OpenFlow13 add-flow s1 table=10,reg0=1,nw_dst=10.0.0.5,ip,reg1=2,actions=mod_dl_dst:00:00:00:00:00:12,mod_nw_dst:10.0.0.12,goto_table:20
sudo ovs-ofctl -O OpenFlow13 add-flow s1 table=10,reg0=1,nw_dst=10.0.0.5,ip,reg1=3,actions=mod_dl_dst:00:00:00:00:00:13,mod_nw_dst:10.0.0.13,goto_table:20
```

- Proactive reverse rules:

```
sudo ovs-ofctl -O OpenFlow13 add-flow s1 table=10,ip,tcp,tp_src=80,actions=mod_dl_src:00:00:00:00:00:05,mod_nw_src:10.0.0.5,goto_table:20
```

OVSDB project code

The current implementation handles all neutron calls in the net-virt/LBaaSHandler.java code, and makes calls to the net-virt-providers/LoadBalancerService to program appropriate flowmods. The rules are updated whenever there is a change in the Neutron LBaaS settings. There is no cache of state kept in the net-virt or providers.

Limitations

Owing to the inflexibility of the multipath action, the existing LBaaS implementation comes with some limitations:

- TCP, HTTP or HTTPS are supported protocols for the pool. (Caution: You can lose access to the members if you assign {Proto:TCP, Port:22} to LB)
- Member weights are ignored.
- The update of an LB instance is done as a delete + add, and not an actual delta.
- The update of an LB member is not supported (because weights are ignored).
- Deletion of an LB member leads to the reprogramming of the LB on all nodes (because of the way multipath does link hash).
- There is only a single LB instance per subnet because the pool-id is not reported in the create load-balancer call.

15. Packet Cable MultiMedia (PCMM)

Table of Contents

| | |
|---|-----|
| Checking out the Packetcable PCMM project | 275 |
| System Overview | 275 |
| Dependency Map | 276 |
| Packetcable Components | 276 |
| Download and Install | 277 |
| Preparing to Work with the Packetcable PCMM Service | 277 |
| Explore and exercise the PacketCable REST API | 281 |
| RESTCONF API Explorer | 281 |
| Postman | 282 |
| Custom Testsuite | 282 |
| Using Wireshark to Trace PCMM | 282 |
| Debugging and Verifying DQoS Gate (Flows) on the CMTS | 283 |
| Find the Cable Modem | 283 |
| Arris | 285 |
| RESTCONF API for Packetcable PCMM | 285 |

Checking out the Packetcable PCMM project

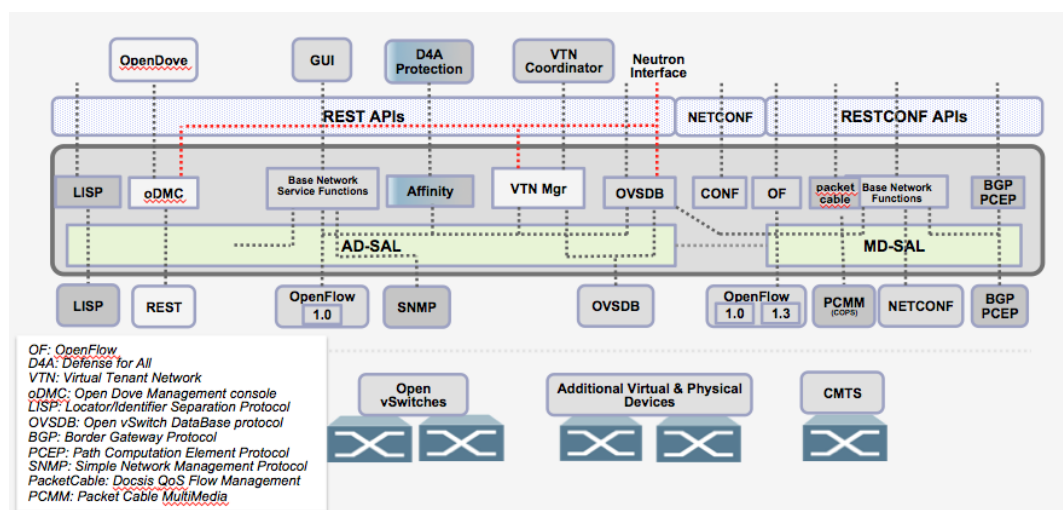
```
git clone https://git.opendaylight.org/gerrit/p/packetcable.git
```

The above command will create a directory called "packetcable" with the project.

System Overview

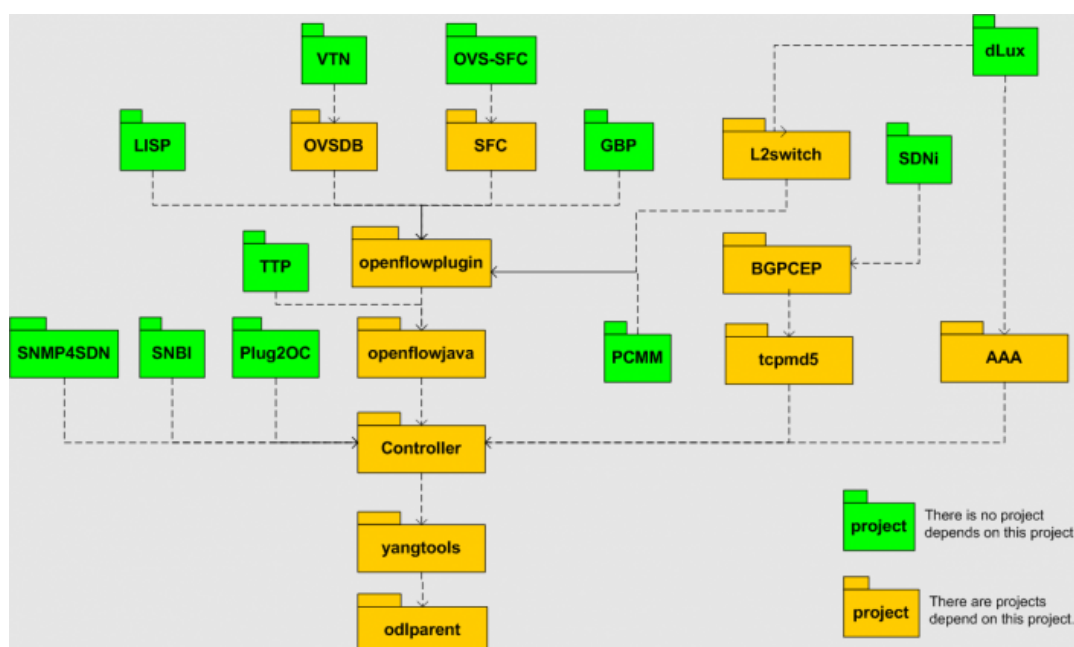
These components introduce a DOCSIS QoS Service Flow management using the PCMM protocol. The driver component is responsible for the PCMM/COPS/PDP functionality required to service requests from PacketCable Provider and FlowManager. Requests are transposed into PCMM Gate Control messages and transmitted via COPS to the CMTS. This plugin adheres to the PCMM/COPS/PDP functionality defined in the CableLabs specification. PacketCable solution is an MD-SAL compliant component.

Figure 15.1. System Overview



Dependency Map

Figure 15.2. Dependency Map



Packetcable Components

packetcable is comprised of three OpenDaylight bundles

Table 15.1. Table of Bundle and Components

| Bundle | Description |
|-------------------|---|
| packetcable-model | Contains the YANG information model for flows and nodes |

| Bundle | Description |
|----------------------|---|
| packetcable-provider | Provider hosts the model processing, RESTCONF, API implementation, and brokers requests to consumer |
| packetcable-driver | Driver manages PCMM Gate message over COPS for flows and CMTS connections |
| packetcable-consumer | Consumer is the codec for for transforming the model of nodes and flows to COPS Gate messages |

See [YANG Model](#)

Download and Install

Current instructions

Download

[Download](#)

<http://nexus.opendaylight.org/content/groups/staging/org/opendaylight/integration/distribution-karaf/>

Unzip

```
unzip distribution-karaf-0.2.0-Helium.zip
```

Run Karaf

```
cd distribution-karaf-0.2.0-Helium/bin/  
./karaf
```

Preparing to Work with the Packetcable PCMM Service

Minimum install procedure

```
opendaylight-user@root>feature:install odl-packetcable-all
```

Useful Features to Start with PCMM

```
opendaylight-user@root>feature:install odl-restconf odl-l2switch-switch odl-  
dlux-core odl-mdsal-apidocs odl-packetcable-all
```

Auto Starting a Series of Bundles using Karaf

Edit etc/org.apache.karaf.features.cfg 'featuresBoot'

```
#  
# Comma separated list of features to install at startup  
#  
featuresBoot=config,standard,region,package,kar,ssh,management,odl-restconf,  
odl-l2switch-switch,odl-dlux-core,odl-mdsal-apidocs,odl-packetcable-all
```

Starting Karaf as System Service

```
cd distribution-karaf-0.2.0-Helium/  
sudo bin/start
```

Accessing the Karaf Console

```
ssh -p 8101 karaf@localhost
```

Add These Directives to Your Operating System Profile to Change the Karaf Startup Parameters for Troubleshooting

```
export KARAF_DEBUG=true  
export JAVA_DEBUG_OPTS="-Xdebug -Xnoagent -Djava.compiler=NONE -  
Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5005"
```

Tell a Bundle to Log Debug

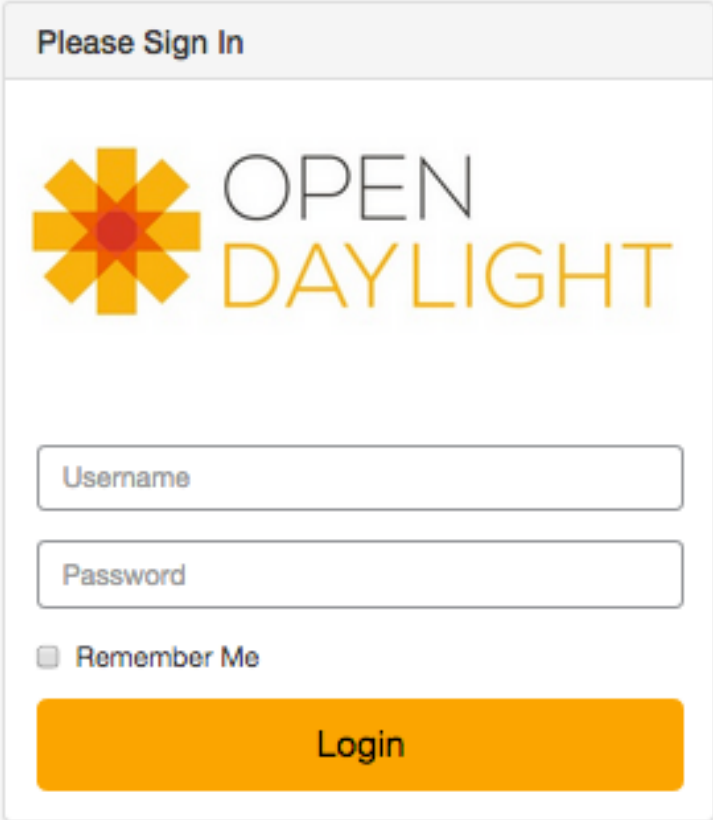
```
log:set org.opendaylight.packetcable
```

Management UI

<http://localhost:8181/dlux/index.html>

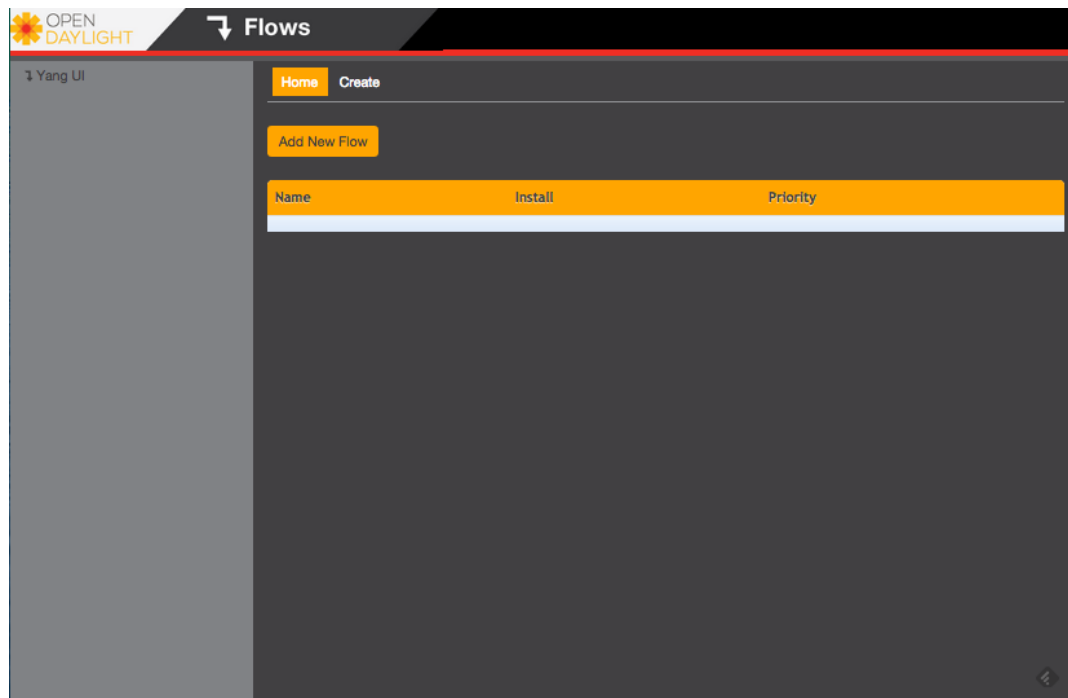
| | |
|----------|-------|
| user | admin |
| password | admin |

Sign in

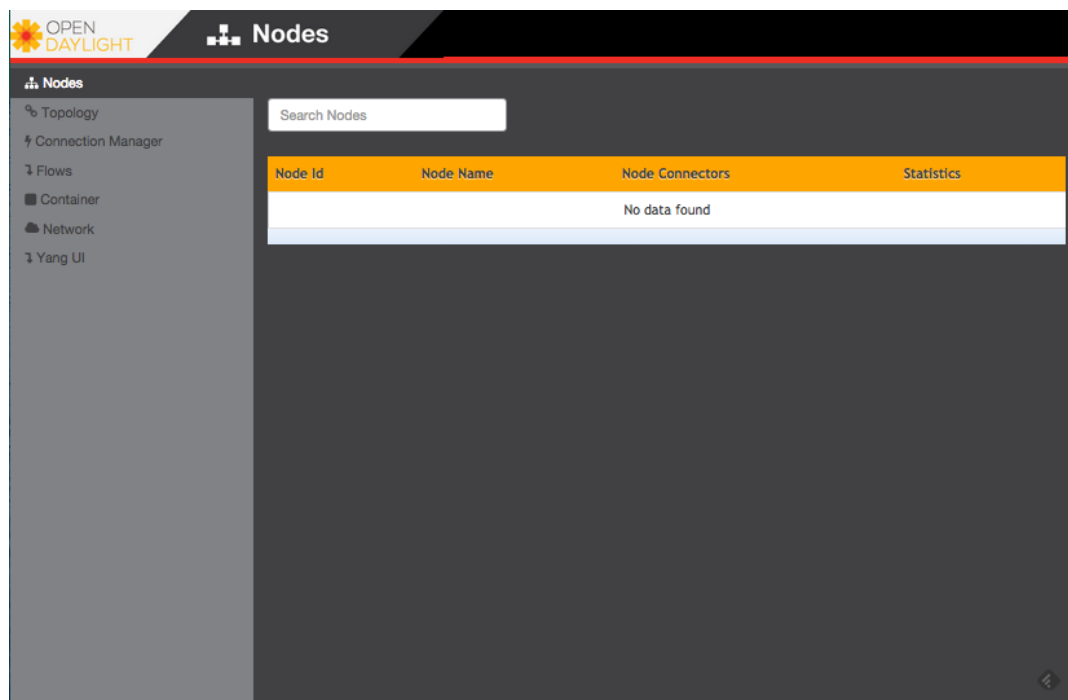
Figure 15.3. Sign in to Dlux UI

The image shows a sign-in form titled "Please Sign In". It features the OpenDaylight logo, which consists of a stylized orange flower icon and the text "OPEN DAYLIGHT". Below the logo are two input fields: "Username" and "Password". There is a checkbox labeled "Remember Me" and a large orange "Login" button at the bottom.

Manage Flows

Figure 15.4. View and Manage Flows in Dlux

Manage Nodes

Figure 15.5. View and Manage Nodes in Dlux

Explore and exercise the PacketCable REST API

<http://localhost:8181/apidoc/explorer/index.html>

RESTCONF API Explorer

<http://localhost:8181/apidoc/explorer/index.html>

Add a CMTS to Opendaylight Inventory

Figure 15.6. Add CMTS using RESTCONF Explorer

GET

/config/opendaylight-inventory:nodes/node/{id}/packetcable-cmts:cmts-node/

Response Class

Model | Model Schema

```
{
  "packetcable-cmts:port": "integer"
}
```

Response Content Type

application/json

Parameters

| Parameter | Value | Description | Parameter Type | Data Type |
|-----------|-------|-------------------------------------|----------------|-----------|
| id | | The unique identifier for the node. | path | string |

Try it out

PUT

/config/opendaylight-inventory:nodes/node/{id}/packetcable-cmts:cmts-node/

Response Class

Model | Model Schema

```
(config)cmts-node {
  packetcable-cmts:port (integer, optional): TCP port number to connect,
  packetcable-cmts:address (undefined, optional): IP Address of CMTS
}
```

Response Content Type

application/json

Parameters

| Parameter | Value | Description | Parameter Type | Data Type |
|-------------------|--|-------------------------------------|----------------|----------------------|
| id | 1 | The unique identifier for the node. | path | string |
| (config)cmts-node | <pre>{ "cmts-node": { "address": "10.200.90.3", "port": "3918" } }</pre> | | body | Model Model Schema |

Parameter content type:

application/json

Try it out

Hide Response

Request URL

http://localhost:8181/restconf/config/opendaylight-inventory:nodes/node/1/packetcable-cmts:cmts-node/

Response Body

no content

Response Code

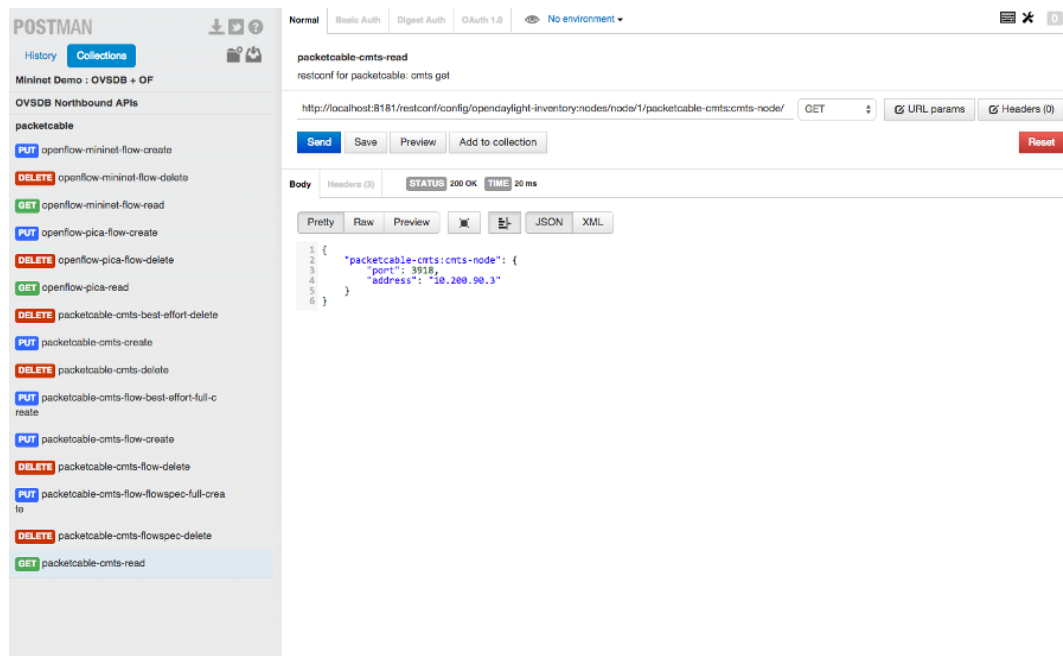
200

Postman

[Configure the Chrome browser](#)

Download and import sample [packetcable collection](#) for Postman.

Figure 15.7. Postman Collection for Packetcable PCMM



Custom Testsuite

Most of the tests for RESTCONF can be adapted for PCMM and service flow testing. The following list of Packetcable client testing. Browse this folder for tests and examples used for testing.

restconfapi.py

Scripted series of packetcable actions testing compliance. Other flows can be formulated and added to create a regression test of what kind of flows are interesting for use cases.

flow_config_perf_pcmm.py

For load testing there is this nice tool that could be repurpose to load test a CMTS.

Using Wireshark to Trace PCMM

To start wireshark with privileges issue the following command:

```
sudo wireshark &
```

Select the interface to monitor.

Use the Filter to only display COPS messages by applying "cops" in the filter field. .Using Wireshark to View COPS image::pcmm-wireshark.png["Wireshark",width=500]

Debugging and Verifying DQoS Gate (Flows) on the CMTS

Below are some of the most useful CMTS commands to verify flows have been enabled on the CMTS.

Cisco

[Cisco CMTS Cable Command Reference](#)

Find the Cable Modem

```
10k2-DSG#show cable modem
```

| D Num I | MAC Address | IP Address | I/F | MAC State | Prim Sid | RxPwr (dBmv) | Timing Offset |
|----------------|---------------|------------|----------|--------------|-------------|-----------------|------------------|
| CPE P | | | | | | | |
| 0010.188a.faf6 | 0.0.0.0 | C8/0/0/U0 | offline | 1 | 0.00 | 1482 | |
| 0 N | | | | | | | |
| 74ae.7600.01f3 | 10.32.115.150 | C8/0/10/U0 | online | 1 | -0.50 | 1431 | |
| 0 Y | | | | | | | |
| 0010.188a.fad8 | 10.32.115.142 | C8/0/10/UB | w-online | 2 | -0.50 | 1507 | |
| 1 Y | | | | | | | |
| 000e.0900.00dd | 10.32.115.143 | C8/0/10/UB | w-online | 3 | 1.00 | 1677 | |
| 0 Y | | | | | | | |
| e86d.5271.304f | 10.32.115.168 | C8/0/10/UB | w-online | 6 | -0.50 | 1419 | |
| 1 Y | | | | | | | |

Show PCMM Plugin Connection

```
10k2-DSG#show packetcabl ?
  cms      Gate Controllers connected to this PacketCable client
  event    Event message server information
  gate     PacketCable gate information
  global   PacketCable global information

10k2-DSG#show packetcable cms
```

| GC-Addr | GC-Port | Client-Addr | COPS-handle | Version | PSID | Key | PDD-Cfg |
|-------------|---------|-------------|--------------|---------|------|-----|---------|
| 10.32.0.240 | 54238 | 10.32.15.3 | 0x4B9C8150/1 | 4.0 | 0 | 0 | 0 |

```
10k2-DSG#show packetcable cms
```

| GC-Addr | GC-Port | Client-Addr | COPS-handle | Version | PSID | Key | PDD-Cfg |
|-------------|---------|-------------|--------------|---------|------|-----|---------|
| 10.32.0.240 | 54238 | 10.32.15.3 | 0x4B9C8150/1 | 4.0 | 0 | 0 | 0 |

Show COPS Messages

```
debug cops details
```

Use CM Mac Address to List Service Flows

```
10k2-DSG#show cable modem
```

| D | MAC Address | IP Address | I/F | MAC | Prim | RxPwr | Timing |
|----------------|---------------|------------|------------|----------|------|--------|--------|
| Num | I | | | State | Sid | (dBmV) | Offset |
| CPE | P | | | | | | |
| 0010.188a.faf6 | --- | | C8/0/0/UB | w-online | 1 | 0.50 | 1480 |
| 1 | N | | | | | | |
| 74ae.7600.01f3 | 10.32.115.150 | | C8/0/10/U0 | online | 1 | -0.50 | 1431 |
| 0 | Y | | | | | | |
| 0010.188a.fad8 | 10.32.115.142 | | C8/0/10/UB | w-online | 2 | -0.50 | 1507 |
| 1 | Y | | | | | | |
| 000e.0900.00dd | 10.32.115.143 | | C8/0/10/UB | w-online | 3 | 0.00 | 1677 |
| 0 | Y | | | | | | |
| e86d.5271.304f | 10.32.115.168 | | C8/0/10/UB | w-online | 6 | -0.50 | 1419 |
| 1 | Y | | | | | | |

```
10k2-DSG#show cable modem 000e.0900.00dd service-flow
```

SUMMARY:

| MAC Address | IP Address | Host | MAC | Prim | Num | Primary |
|----------------|---------------|-----------------|----------|------|-----|-----------|
| DS | | Interface | State | Sid | CPE | |
| | | Downstream RfId | | | | |
| 000e.0900.00dd | 10.32.115.143 | C8/0/10/UB | w-online | 3 | 0 | Mo8/0/2:1 |
| | 2353 | | | | | |

| Sfid | Dir | Curr | Sid | Sched | Prio | MaxSusRate | MaxBrst | MinRsvRate |
|------------|-----|-------|------|-------|------|------------|---------|------------|
| Throughput | | State | Type | | | | | |
| 23 | US | act | 3 | BE | 0 | 0 | 3044 | 0 |
| 30 | US | act | 16 | BE | 0 | 500000 | 3044 | 0 |
| 24 | DS | act | N/A | N/A | 0 | 0 | 3044 | 0 |

UPSTREAM SERVICE FLOW DETAIL:

| SFID | SID | Requests | Polls | Grants | Delayed Grants | Dropped Grants | Packets |
|------|-----|----------|-------|--------|----------------|----------------|---------|
| 23 | 3 | 784 | 0 | 784 | 0 | 0 | 784 |
| 30 | 16 | 0 | 0 | 0 | 0 | 0 | 0 |

DOWNSTREAM SERVICE FLOW DETAIL:

| SFID | RP_SFID | QID | Flg | Policer | Drops | Scheduler | Drops | FrwdIF |
|------|---------|--------|-----|---------|-------|-----------|-------|-----------|
| | | | | Xmits | | Xmits | | |
| 24 | 33019 | 131550 | | 0 | 0 | 777 | 0 | Wi8/0/2:2 |

Flags Legend:

\$: Low Latency Queue (aggregated)

~: CIR Queue

Deleting a PCMM Gate Message from the CMTS

```
10k2-DSG#test cable dsd 000e.0900.00dd 30
```

Find service flows

All gate controllers currently connected to the PacketCable client are displayed

```
show cable modem 00:11:22:33:44:55 service flow ????  
show cable modem
```

Debug and display PCMM Gate messages

```
debug packetcable gate control  
debug packetcable gate events  
show packetcable gate summary  
show packetcable global  
show packetcable cms
```

Debug COPS messages

```
debug cops detail  
debug packetcable cops  
debug cable dynamic_qos trace
```

Arris

Pending

RESTCONF API for Packetcable PCMM

CMTS

CMTS can be read, created, updated and deleted by a user having the correct role. An ID is used to identify where to read or save the CMTS node.

Read

| | |
|---------------|--|
| URL | /restconf/config/.opendaylight-inventory:nodes/node/[id]/packetcable-cmts:cmts-node/ |
| Method | GET |
| Request Body | {} |
| Response Body | {} |
| Return Codes | 201 |

Create

| | |
|--------|--|
| URL | /restconf/config/.opendaylight-inventory:nodes/node/[id]/packetcable-cmts:cmts-node/ |
| Method | PUT |

| | |
|----------------------|---|
| Request Body | <pre>{ "packetcable-cmts:cmts-node": { "port": "3918", "address": "10.200.90.3" } }</pre> |
| Response Body | <pre>{}</pre> |
| Return Codes | 201 |

Delete

| | |
|----------------------|---|
| URL | /restconf/config/opendaylight-inventory:nodes/node/[id]/packetcable-cmts:cmts-node/ |
| Method | DELETE |
| Request Body | <pre>{}</pre> |
| Response Body | <pre>{}</pre> |
| Return Codes | 201 |

Flows

Flows can be read, created, updated and deleted by a user having the correct role. A CMTS ID is used to identify which CMTS node to read or save the flow. Note: The Table ID is not used.

Read

| | |
|----------------------|--|
| URL | /restconf/config/opendaylight-inventory:nodes/node/[cmts id]/table/0/flow/[flow id] |
| Method | GET |
| Request Body | <pre>{}</pre> |
| Response Body | <pre>{ "flow": { "cookie": "101", "cookie_mask": "255", "flow-name": "FooXf7", "hard-timeout": "1200", "id": "256", "idle-timeout": "3400", "installHw": "false", "instructions": { "instruction": { "apply-actions": { "action": { "order": "0", "traffic-profile": "best-effort" } }, "order": "0" } }, "match": { "ethernet-match": { "ethernet-type": { "type": "34525" } }, "ip-match": { "ip-dscp": "60", "ip-ecn": "3", "ip-protocol": "6" }, "ipv6-destination": "fe80:2acf:e9ff:fe21::6431/94", "ipv6-source": "1234:5678:9ABC:DEF0:FDCD:A987:6543:210F/76", "tcp-destination-port": "8080", "tcp-source-port": "183" } } }</pre> |

```

    "priority": "2",
    "strict": "false",
    "table_id": "2"
  }
}

```

Create

| | |
|---------------------|---|
| URL | /restconf/config/opendaylight-inventory:nodes/node/[cmts id]/table/0/flow/[flow id] |
| Method | PUT |
| Request Body | <pre> { "flow": { "barrier": "false", "flow-name": "FooXCableFlowCrazyTrafficProfileFBesteffort1", "id": "115", "installHw": "false", "instructions": { "instruction": { "apply-actions": { "action": { "traffic-profile": "best-effort", "be-authorized-envelope": { "traffic-priority": "0", "reserved0": "0", "reserved1": "0", "request-transmission-policy": "0", "maximum-sustained-traffic-rate": "0", "maximum-traffic-burst": "3044", "maximum-reserved-traffic-rate": "0", "traffic-rate-packet-size-maximum-concatenated- burst": "0", "assumed-minimum-reserved": "1522", "required-attribute-mask": "0", "forbidden-attribute-mask": "0", "attribute-aggregation-rule-mask": "0", }, "be-reserved-envelope": { "traffic-priority": "0", "reserved0": "0", "reserved1": "0", "request-transmission-policy": "0", "maximum-sustained-traffic-rate": "0", "maximum-traffic-burst": "3044", "maximum-reserved-traffic-rate": "0", "traffic-rate-packet-size-maximum-concatenated- burst": "0", "assumed-minimum-reserved": "1522", "required-attribute-mask": "0", "forbidden-attribute-mask": "0", "attribute-aggregation-rule-mask": "0", }, "be-committed-envelope": { "traffic-priority": "0", "reserved0": "0", "reserved1": "0", "request-transmission-policy": "0", "maximum-sustained-traffic-rate": "0", "maximum-traffic-burst": "3044", "maximum-reserved-traffic-rate": "0", "traffic-rate-packet-size-maximum-concatenated- burst": "0", "assumed-minimum-reserved": "1522", "required-attribute-mask": "0", "forbidden-attribute-mask": "0", "attribute-aggregation-rule-mask": "0", } }, "order": "0" } }, "order": "0" }, "match": { "ethernet-match": { "ethernet-type": { </pre> |

| | |
|----------------------|--|
| | <pre> "type": "2048" }, "ipv4-destination": "10.0.0.1/24" }, "priority": "2", } }</pre> |
| Response Body | {} |
| Return Codes | 201 |

Delete

| | |
|----------------------|--|
| URL | /restconf/config/.opendaylight-inventory:nodes/node/[cmts id]/table/0/flow/[flow id] |
| Method | DELETE |
| Request Body | {} |
| Response Body | {} |
| Return Codes | 201 |

Specifications and References

The packetcable-driver was written to the [PacketCable Specification Multimedia Specification PKT-SP-MM-I05-091029](#)

16. Plugin for OpenContrail

The Developer Guide for the Plugin for OpenContrail can be found on the OpenDaylight wiki here: https://wiki.opendaylight.org/view/Southbound_Plugin_to_the_OpenContrail_Platform:Developer_Guide

17. Service Function Chaining

More information on Service Function Chaining can be found on the OpenDaylight wiki here: https://wiki.opendaylight.org/view/Service_Function_Chaining:Main

18. SNBI Developers' Guide

Table of Contents

| | |
|--|-----|
| Defining characteristics of SNBI bootstrapping | 291 |
| SNBI components | 291 |
| How SNBI works | 292 |

The Secure Network Bootstrapping Infrastructure (SNBI) component of OpenDaylight automatically creates secure IP connectivity between a set of forwarding elements (devices) and the controller.

Defining characteristics of SNBI bootstrapping

In the SNBI context, network bootstrapping involves discovering the device, authenticating a device, and installing the device domain certificate so that it becomes part of an administrative domain ("SNBI domain").

SNBI bootstrapping is:

- **Secure:** Only devices on the white list of the SNBI registrar are allowed into a domain. The RA (Registrar Authority) and the CA (Certificate Authority) ensure that a secure channel of communication is established between the SNBI registrar and the devices, and also between the devices. SNBI uses Bouncy Castle to run the CA and sign certificates.
- **Automatic:** Normal network bootstrapping involves the manual configuration of network connectivity. To secure any control protocol connecting to the device, one typically needs to manually install certificates. SNBI fully automates the configuration of network connectivity (incl. IP address assignment, routing protocol configuration, and others) as well as the distribution and installation of certificates.

SNBI components

An SNBI implementation includes SNBI controllers and forwarding elements.

- **Forwarding element component (SNBI agent)** The software package for secure discovery service is created and integrated with the network container reference platform for the devices.
- **Controller components:**
 - **SNBI Registrar:** The north-bound configuration manager that configures the south-bound SNBI plugin.

The registrar establishes trust in a network domain thereby anchoring it.

The SNBI registrar does the following:

- Maintains the white list of devices which belong to a domain. An administrator sets a white list for the registrar for every domain.
- Decides in accordance with policy rules as to which devices are admitted to a domain.
- Manages certificates: Issues, renews, and revokes certificates as a CA. Certificate management is fully self-contained in the SNBI solution.
- SNBI plugin The secure discovery service is a southbound plugin that runs the SNBI protocol.

Forwarding element components

The SNBI functions in the Forwarding Elements (FEs) are implemented inside lightweight portable foundations.

Portable Foundation

The SNBI portable foundation can use any light weight portable foundation technology that provides a protected and isolated application execution environment. The current SNBI implementation utilizes Docker, a light weight portable foundation mechanism supported by the current Linux kernels.

How SNBI works

An administrator plugs in a device thereby introducing it into a domain. When a forwarding element discovers the new device, it acts as an intermediary between the new device and the registrar, and proxies all device requests to the registrar.

A device gets a neighbour invite request from the registrar which is forwarded by a proxy forwarding element. The device presents its Unique Device Identifier (UDI) to the registrar through the proxy. The UDI could be anything, a serial number, an 802.1AR compliant identifier, or others. The proxy sends the credentials to the registrar for validation. Upon validation, the device sends a Certificate Sign Request (CSR) PKCS10 request and gets it signed by the CA running at the SNBI Registrar. The CA enrolls and signs an x.509 certificate.

The device gets a domain name and ID. The device uses the domain name and ID to also derive its IPv6 which it will use to communicate with other SNBI agents over the secure channel.

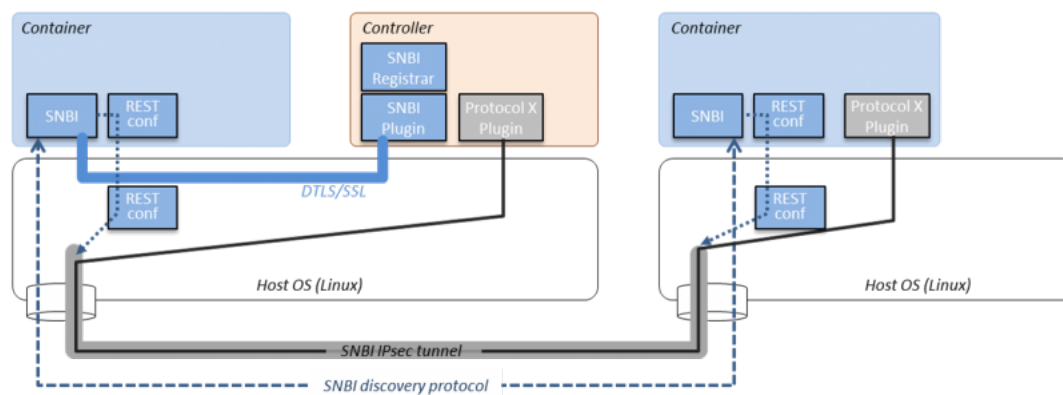
Bootstrapping a device using SNBI

To bootstrap a device using SNBI:

1. In the Yang model of the REST API for SNBI, enter the names of devices per domain to be bootstrapped. The registrar includes this information in its white list: s/Yang/YANG/.
2. Plug in the device to be bootstrapped.

Controller and FE communications

Figure 18.1. Communication between the controller and FE



SNBI between controller and portable foundation The SNBI-plugin on the Controller and the SNBI agent on the "first hop" FE establishes a DTLS/SSL connection to secure their communication. It is assumed that the device or server which runs the Controller runs an instance of the portable foundation or container. This allows for SNBI to automatically establish a secure IP connectivity throughout the network without the need to pre-configure any IP connectivity between the devices in the network. If the Controller is hosted on a device which does not run an instance of an SNBI-agent within a portable foundation, then IP connectivity between the Controller and the "first hop" FE which runs an instance of an SNBI-agent within a portable foundation needs to be configured by other means (for example, manually).

It is recommended that the Controller always be hosted on a device (server) which also runs an instance of the SNBI-agent within the portable foundation.

SNBI agent discovery SNBI-agents discover each other through a discovery protocol.

Secure communication between devices SNBI agents establish a secure channel among themselves, which is typically an IPsec connection. Once the secure channel is established, other services running on the same host (be it a forwarding element or a controller) can leverage the secure IP connectivity for their means. In Figure 1, an example "protocol x plugin" leverages the secure channel to communicate between different instances of protocol x. Example protocols which could use the secure channel include OpenFlow, and Netconf. The protocols need not establish their own secure transport (for example, using DTLS/SSL). Any protocol can, of course, establish its own additional secure transport on top of the already secure connectivity provided by SNBI.

Configuration control between SNBI-agent and underlying host OS An SNBI-agent hosted in a portable foundation controls and retrieves certain configuration parameters through a RESTconf/Netconf interface. This includes:

- The establishment and configuration of the secure channel (that is to say, the IPsec connection).
- Routing table control.
- The retrieval of a UDI.

The configuration interface between portable foundation and underlying host is based on standard IETF YANG models ([RFC 7223](#) for interface configuration, draft-ietf-netmod-routing-cfg for route management, and others).

This approach decouples the underlying host and its configuration specifics from the portable foundation hosting environment, and allows for the simplified portability of the portable foundation.

Benefits of SNBI discovery

The automatic discovery between SNBI devices and controllers:

- Reveals the physical topology of the network thus supporting network management
- Exposes a device as either a forwarding element or a controller
- Associates a device to an administrative domain
- Makes possible the initiation of controller federation processes through device type and domain information

The SNBI component of the OpenDaylight Controller automatically creates secure network connectivity between devices. This connectivity can be leveraged by other features and functions to for example to install, control and manage the life cycle of additional software components hosted within the portable foundation of a forwarding element.. The portable foundation built on container technology can be extended to support additional orchestration and configuration management functions.

SNBI: Non-ODL technologies used

- Yang models: The SNBI APIs are defined through Yang.

RFC 6020 'YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF) is available at: <http://tools.ietf.org/html/rfc6020>

- Docker SNBI uses lightweight portable foundations to implement SNBI functions in FEs. The SNBI portable foundation in the current implementation uses Docker and Linux kernels. SNBI uses Docker to start the portable foundation in a host, and pass needed parameters, such as the CID, by means of environment variables into the container.

Information on the Docker open platform is available at: <https://www.docker.com/>

SNBI terms and definitions

| | |
|-------------|---|
| SNBI Domain | A logical set of devices with common goals |
| Registrar | SNBI software that acts as a domain trust anchor, incorporating both RA and CA functions to bootstrap new devices |
| UDI | Unique device identifier |
| FE | Forwarding element |

| | |
|---------------------|--|
| Portable foundation | Reference environment to host network functions, like the SNBI-agent, on devices. The PF provides infrastructure to help host network-centric software components on devices while decoupling them from the Linux distribution and software load of the underlying host. |
| SNBI RA | The Registration Authority module that authenticates new devices |
| SNBI CA | The Certificate Authority module that signs device certificates |

19. SNMP4SDN

The Developer Guide for SNMP4SDN can be found on the OpenDaylight wiki here: https://wiki.opendaylight.org/view/SNMP4SDN:Helium_Developer_Guide

20. TCP-MD5

The Developer Guide for TCP-MD5 can be found on the OpenDaylight wiki here: https://wiki.opendaylight.org/view/TCPMD5:Helium_Developer_Guide

21. Table Type Patterns

Table of Contents

| | |
|---------------------------|-----|
| Introduction | 298 |
| Using The REST APIs | 299 |
| Limitations | 309 |



Important

This section assumes you have already downloaded the Karaf distribution of OpenDaylight Helium and followed the instructions in the Table Type Patterns section of the Installation guide. If not, do that first.

Introduction

Table Type Patterns are a specification developed by the [Open Networking Foundation](#) to enable the description and negotiation of subsets of the OpenFlow protocol. This is particularly useful for hardware switches that support OpenFlow as it enables the to describe what features they do (and thus also what features they do not) support. More details can be found in the full specification listed on the [OpenFlow specifications page](#).

Support in Helium

In the Helium release, Table Type Patterns (TTPs) are exposed as a YANG model for TTPs themselves which can be loaded into the MD-SAL Data Store in three places:

1. As one of a list of TTPs in the `.opendaylight-ttps` top-level container.
2. Attached to a node in the `.opendaylight-inventory` model as an `active_ttp` via the `ttp-capable-node` augmentation.
3. Attached to a node in the `.opendaylight-inventory` model as one of a list of `supported_ttps` via the `ttp-capable-node` augmentation. // link to the inventory docs somehow?

Each of these points can be accessed either through the RESTCONF-based REST APIs or via the Java interface to the MD-SAL Data Store. This discussion will focus on the REST APIs.



Note

Developers who wish to use the Java interfaces are encouraged to first read and understand using the MD-SAL Data Store's APIs including importing the appropriate bundles for to get models, dealing with transactions, and constructing instance identifiers.

After that, it should be somewhat straightforward to translate the REST API calls here into appropriate instance identifiers which can be used in MD-SAL Data Store transactions.

Using The REST APIs

As stated above there are 3 locations where a Table Type Patter can be placed into the MD-SAL Data Store. They correspond to 3 different REST API URIs:

1. `restconf/config/onf-ttp:opendaylight-ttps/onf-ttp:table-type-patterns/`
2. `restconf/config/opendaylight-inventory:nodes/node/{id}/ttp-inventory-node:active_ttp/`
3. `restconf/config/opendaylight-inventory:nodes/node/{id}/ttp-inventory-node:supported_ttps/`



Note

Typically, these URIs are running on the machine the controller is on at port 8181. If you are on the same machine they can thus be accessed at <http://localhost:8181/<uri>>

Setting REST HTTP Headers

Authentication

The REST API calls require authentication by default. The default method is to use basic auth with a user name and password of 'admin'.

Content-Type and Accept

RESTCONF supports both xml and json. This example focuses on JSON, but xml can be used just as easily. When doing a PUT or POST be sure to specify the appropriate Content-Type header: either `application/json` or `application/xml`.

When doing a GET be sure to specify the appropriate Accept header: again, either `application/json` or `application/xml`.

Content

The contents of a PUT or POST should be a OpenDaylight Table Type Pattern. An example of one is provided below. The example can also be found at [parser/sample-TTP-from-tests.ttp in the TTP git repository](#).

Sample Table Type Pattern (json).

```
{
  "table-type-patterns": {
    "table-type-pattern": [
      {
        "security": {
          "doc": [
            "This TTP is not published for use by ONF. It is an",
            "example and for",
            "illustrative purposes only.",
            "If this TTP were published for use it would include",

```

```

        "guidance as to any security considerations in this
doc member."
    ],
    },
    "NDM_metadata": {
        "authority": "org.opennetworking.fawg",
        "OF_protocol_version": "1.3.3",
        "version": "1.0.0",
        "type": "TTPv1",
        "doc": [
            "Example of a TTP supporting L2 (unicast, multicast,
flooding), L3 (unicast only),",
            "and an ACL table."
        ],
        "name": "L2-L3-ACLs"
    },
    "identifiers": [
        {
            "doc": [
                "The VLAN ID of a locally attached L2 subnet on a
Router."
            ],
            "var": "<subnet_VID>"
        },
        {
            "doc": [
                "An OpenFlow group identifier (integer)
identifying a group table entry",
                "of the type indicated by the variable name"
            ],
            "var": "<<group_entry_types/name>>"
        }
    ],
    "features": [
        {
            "doc": [
                "Flow entry notification Extension - notification
of changes in flow entries"
            ],
            "feature": "ext187"
        },
        {
            "doc": [
                "Group notifications Extension - notification of
changes in group or meter entries"
            ],
            "feature": "ext235"
        }
    ],
    "meter_table": {
        "meter_types": [
            {
                "name": "ControllerMeterType",
                "bands": [
                    {
                        "type": "DROP",
                        "rate": "1000..10000",
                        "burst": "50..200"
                    }
                ]
            }
        ]
    }
}

```

```

    },
    {
      "name": "TrafficMeter",
      "bands": [
        {
          "type": "DSCP_REMARK",
          "rate": "10000..500000",
          "burst": "50..500"
        },
        {
          "type": "DROP",
          "rate": "10000..500000",
          "burst": "50..500"
        }
      ]
    }
  ],
  "built_in_meters": [
    {
      "name": "ControllerMeter",
      "meter_id": 1,
      "type": "ControllerMeterType",
      "bands": [
        {
          "rate": 2000,
          "burst": 75
        }
      ]
    },
    {
      "name": "AllArpMeter",
      "meter_id": 2,
      "type": "ControllerMeterType",
      "bands": [
        {
          "rate": 1000,
          "burst": 50
        }
      ]
    }
  ]
},
"table_map": [
  {
    "name": "ControlFrame",
    "number": 0
  },
  {
    "name": "IngressVLAN",
    "number": 10
  },
  {
    "name": "MacLearning",
    "number": 20
  },
  {
    "name": "ACL",
    "number": 30
  },
  {

```

```

        "name": "L2",
        "number": 40
    },
    {
        "name": "ProtoFilter",
        "number": 50
    },
    {
        "name": "IPv4",
        "number": 60
    },
    {
        "name": "IPv6",
        "number": 80
    }
],
"parameters": [
    {
        "doc": [
            "documentation"
        ],
        "name": "Showing-curt-how-this-works",
        "type": "type1"
    }
],
"flow_tables": [
    {
        "doc": [
            "Filters L2 control reserved destination addresses
and",
            "may forward control packets to the controller.",
            "Directs all other packets to the Ingress VLAN
table."
        ],
        "name": "ControlFrame",
        "flow_mod_types": [
            {
                "doc": [
                    "This match/action pair allows for
flow_mods that match on either",
                    "ETH_TYPE or ETH_DST (or both) and send
the packet to the",
                    "controller, subject to metering."
                ],
                "name": "Frame-To-Controller",
                "match_set": [
                    {
                        "field": "ETH_TYPE",
                        "match_type": "all_or_exact"
                    },
                    {
                        "field": "ETH_DST",
                        "match_type": "exact"
                    }
                ],
                "instruction_set": [
                    {
                        "doc": [
                            "This meter may be used to limit
the rate of PACKET_IN frames",

```

```

        "sent to the controller"
    ],
    "instruction": "METER",
    "meter_name": "ControllerMeter"
  },
  {
    "instruction": "APPLY_ACTIONS",
    "actions": [
      {
        "action": "OUTPUT",
        "port": "CONTROLLER"
      }
    ]
  }
]
}
],
"built_in_flow_mods": [
  {
    "doc": [
      "Mandatory filtering of control frames
with C-VLAN Bridge reserved DA."
    ],
    "name": "Control-Frame-Filter",
    "priority": "1",
    "match_set": [
      {
        "field": "ETH_DST",
        "mask": "0xfffffffffff0",
        "value": "0x0180C2000000"
      }
    ]
  },
  {
    "doc": [
      "Mandatory miss flow_mod, sends packets to
IngressVLAN table."
    ],
    "name": "Non-Control-Frame",
    "priority": "0",
    "instruction_set": [
      {
        "instruction": "GOTO_TABLE",
        "table": "IngressVLAN"
      }
    ]
  }
]
}
],
"group_entry_types": [
  {
    "doc": [
      "Output to a port, removing VLAN tag if needed.",
      "Entry per port, plus entry per untagged VID per
port."
    ],
    "name": "EgressPort",
    "group_type": "INDIRECT",
    "bucket_types": [

```



```

        {
            "name": "OutputTagged",
            "action_set": [
                {
                    "action": "OUTPUT",
                    "port": "<port_no>"
                }
            ]
        },
        {
            "name": "OutputUntagged",
            "action_set": [
                {
                    "action": "POP_VLAN"
                },
                {
                    "action": "OUTPUT",
                    "port": "<port_no>"
                }
            ]
        },
        {
            "opt_tag": "VID-X",
            "name": "OutputVIDTranslate",
            "action_set": [
                {
                    "action": "SET_FIELD",
                    "field": "VLAN_VID",
                    "value": "<local_vid>"
                },
                {
                    "action": "OUTPUT",
                    "port": "<port_no>"
                }
            ]
        }
    ]
},
"flow_paths": [
    {
        "doc": [
            "This object contains just a few examples of flow
paths, it is not",
            "a comprehensive list of the flow paths required
for this TTP. It is",
            "intended that the flow paths array could include
either a list of",
            "required flow paths or a list of specific flow
paths that are not",
            "required (whichever is more concise or more
useful."
        ],
        "name": "L2-2",
        "path": [
            "Non-Control-Frame",
            "IV-pass",
            "Known-MAC",
            "ACLskip",
            "L2-Unicast",

```

```
        "EgressPort"
    ]
},
{
    "name": "L2-3",
    "path": [
        "Non-Control-Frame",
        "IV-pass",
        "Known-MAC",
        "ACLskip",
        "L2-Multicast",
        "L2Mcast",
        "[EgressPort]"
    ]
},
{
    "name": "L2-4",
    "path": [
        "Non-Control-Frame",
        "IV-pass",
        "Known-MAC",
        "ACL-skip",
        "VID-flood",
        "VIDflood",
        "[EgressPort]"
    ]
},
{
    "name": "L2-5",
    "path": [
        "Non-Control-Frame",
        "IV-pass",
        "Known-MAC",
        "ACLskip",
        "L2-Drop"
    ]
},
{
    "name": "v4-1",
    "path": [
        "Non-Control-Frame",
        "IV-pass",
        "Known-MAC",
        "ACLskip",
        "L2-Router-MAC",
        "IPv4",
        "v4-Unicast",
        "NextHop",
        "EgressPort"
    ]
},
{
    "name": "v4-2",
    "path": [
        "Non-Control-Frame",
        "IV-pass",
        "Known-MAC",
        "ACLskip",
        "L2-Router-MAC",
        "IPv4",
```

```
{
    {
        ]
    }
}

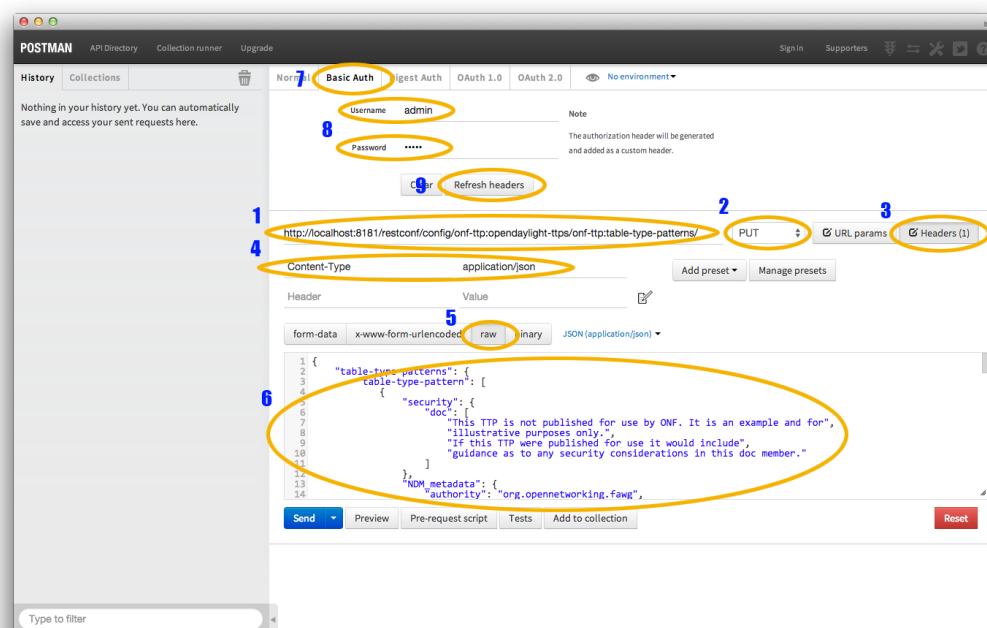
{
    "v4-Unicast-ECMP",
    "L3ECMP",
    "NextHop",
    "EgressPort"
}
```

Making a REST Call

In this example we'll do a PUT to install the sample TTP from above into OpenDaylight and then retrieve it both as json and as xml. We'll use the [Postman - REST Client](#) for Chrome in the examples, but any method of accessing REST should work.

First, we'll fill in the basic information:

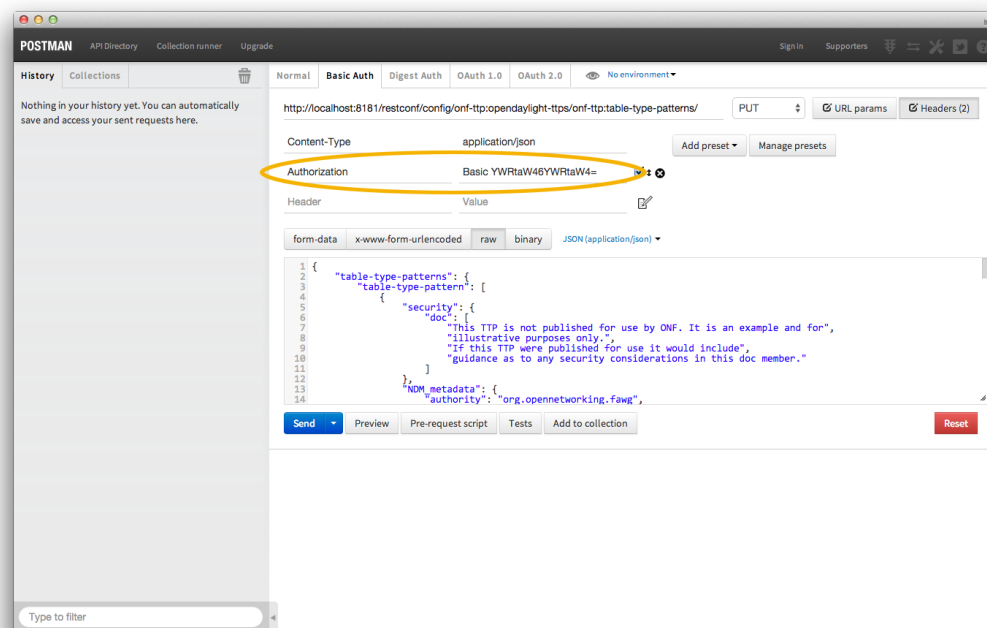
Figure 21.1. Filling in URL, content, Content-Type and basic auth



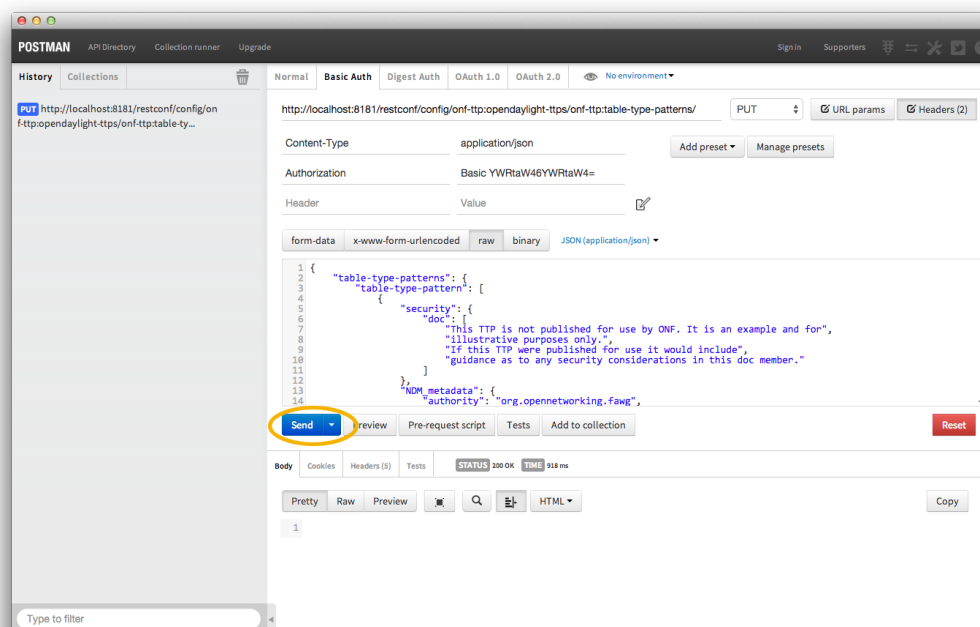
1. Set the URL to `http://localhost:8181/restconf/config/onf-
ttp:opendaylight-ttps/onf-ttp:table-type-patterns/`
2. Set the action to PUT
3. Click Headers and
4. Set a header for Content-Type to application/json
5. Make sure the content is set to raw and

6. Copy the sample TTP from above into the content
7. Click the Basic Auth tab and
8. Set the username and password to admin
9. Click Refresh headers

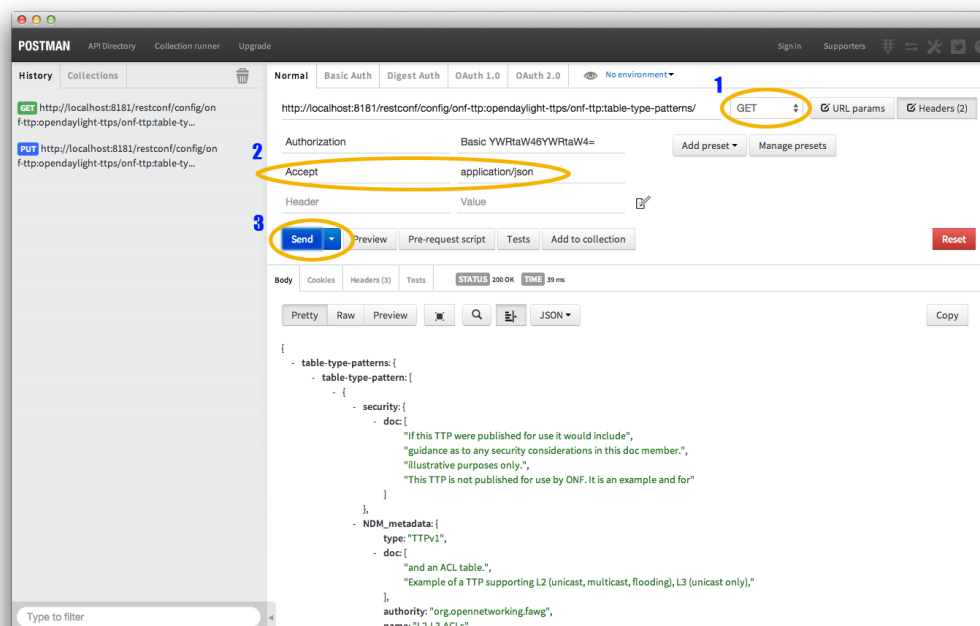
Figure 21.2. Refreshing basic auth headers



After clicking Refresh headers, we can see that a new header (Authorization) has been created and this will allow us to authenticate to make the rest call.

Figure 21.3. PUTting a TTP

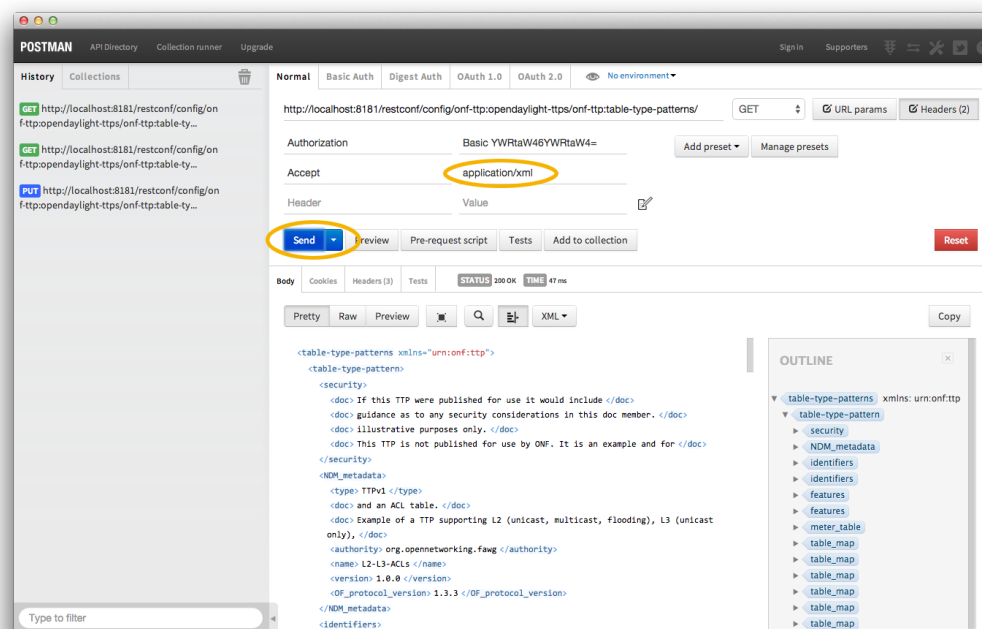
At this point, clicking send should result in a Status response of 200 OK indicating we've successfully PUT the TTP into OpenDaylight.

Figure 21.4. Retrieving the TTP as json via a GET

We can now retrieve the TTP by:

1. Changing the action to GET
2. Setting an Accept header to application/json and
3. Pressing send

Figure 21.5. Retrieving the TTP as xml via a GET



The same process can retrieve the content as xml by setting the Accept header to application/xml.

Limitations

Differences between OpenDaylight TTP and ONF TTP

The OpenDaylight YANG specification for TTPs differs from the ONF's specification in a few areas. These differences are due to limitations in the subsets of JSON that YANG schemas can be used to describe.

- **doc members must always be lists and cannot be just a string**

For example, this is not allowed:

```
"doc": "The VLAN ID of a locally attached L2 subnet on a Router."
```

While this is:

```
"doc": ["The VLAN ID of a locally attached L2 subnet on a Router."]
```

- **table_map formats differ**

In the ONF spec, the table_map looks like this

```
"table_map": {  
  "ControlFrame": 0,  
  "IngressVLAN": 1,  
  "MacLearning": 2,  
  "L2": 3  
},
```

In the ODL TTP YANG definition, it would instead look like this:

```
"table_map": [  
  { "name": "ControlFrame", "number": 0 },  
  { "name": "IngressVLAN", "number": 1 },  
  { "name": "MacLearning", "number": 2 },  
  { "name": "L2", "number": 3 },  
],
```

- **Limited meta member keywords**

The meta member keywords (`all`, `one_or_more`, `zero_or_more`, `exactly_one`, and `zero_or_one`) are allowed anywhere in a TTP according to the ONF specification, but they are only allowed in specific locations in the ODL YANG schema. Specifically:

- a. `all`, `one_or_more`, and `zero_or_more` are allowed in the `flow_mod_types` member
- b. `exactly_one` and `zero_or_one` are allowed in the `match_set` and `instruction_set` members as well as in lists of actions.

- **`flow_paths` repeated table syntax differs**

In the ONF TTP specification, the ability to repeat a table in a path traversal of the tables is done by having the table be a list containing the table name as a string. Like this:

```
"flow_paths": [ "path": [ "table1", [ "table2" ] ] ]
```

In the ODL YANG schema this is instead represented by moving the square brackets inside the string as follows:

```
flow_paths": [ "path": [ "table1", "[table2]" ] ]
```

- `*priority` and `priority_rank` must be strings and can't be numbers.

The `priority` and `priority_rank` members are allowed to be either strings or numbers in the ONF specification, but must be strings in the ODL YANG schema.

- **Empty lists must be omitted**

Lists like this:

```
"match_set": [ ]
```

must instead be omitted from the TTP.

Strictly Informational

At this point in time the only operations available with TTPs are storing and retrieving TTPs from the data store in the three previously mentioned places.

Additional features that make use of and populate this information are planned for future releases.

Known issues

1. Strings containing some special characters result in REST calls returning a 400 `BadRequest` code. A string that contains both an opening angle bracket (<) and a colon (:) with the angle bracket appearing first is known to trigger this behavior.

For example this is known to break RESTCONF:

```
"var": "<<group_entry_types:name>>"
```

While these two work

```
"var": "group_entry_types:name"
```

```
"var": "<<group_entry_types/name>>"
```


22. VTN

Table of Contents

| | |
|---|-----|
| Virtual Tenant Network (VTN) | 312 |
| Hacking VTN Coordinator | 326 |
| Hacking VTN Manager | 329 |
| Hacking VTN Manager(Helium) | 330 |
| Openstack Support Developer Guide | 332 |

Virtual Tenant Network (VTN)

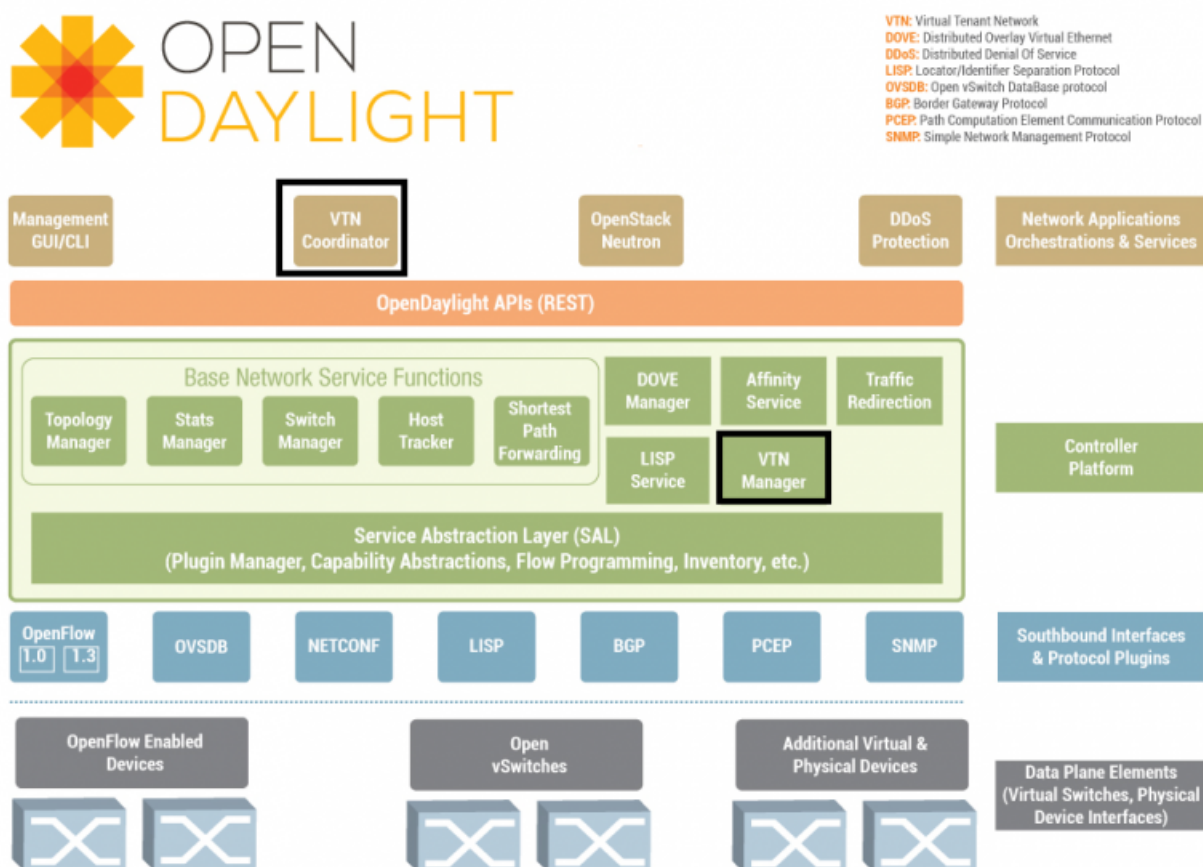
OpenDaylight Virtual Tenant Network (VTN) Overview

OpenDaylight Virtual Tenant Network (VTN) is an application that provides multi-tenant virtual network on an SDN controller.

Conventionally, huge investment in the network systems and operating expenses are needed because the network is configured as a silo for each department and system. Therefore various network appliances must be installed for each tenant and those boxes cannot be shared with others. It is a heavy work to design, implement and operate the entire complex network. The uniqueness of VTN is a logical abstraction plane. This enables the complete separation of logical plane from physical plane. Users can design and deploy any desired network without knowing the physical network topology or bandwidth restrictions. VTN allows the users to define the network with a look and feel of conventional L2/L3 network. Once the network is designed on VTN, it will automatically be mapped into underlying physical network, and then configured on the individual switch leveraging SDN control protocol. The definition of logical plane makes it possible not only to hide the complexity of the underlying network but also to better manage network resources. It achieves reducing reconfiguration time of network services and minimizing network configuration errors. OpenDaylight Virtual Tenant Network (VTN) is an application that provides multi-tenant virtual network on an SDN controller. It provides API for creating a common virtual network irrespective of the physical network.

It is implemented as two major components

- [the section called "VTN Coordinator" \[313\]](#)
- [the section called "VTN Manager" \[315\]](#)

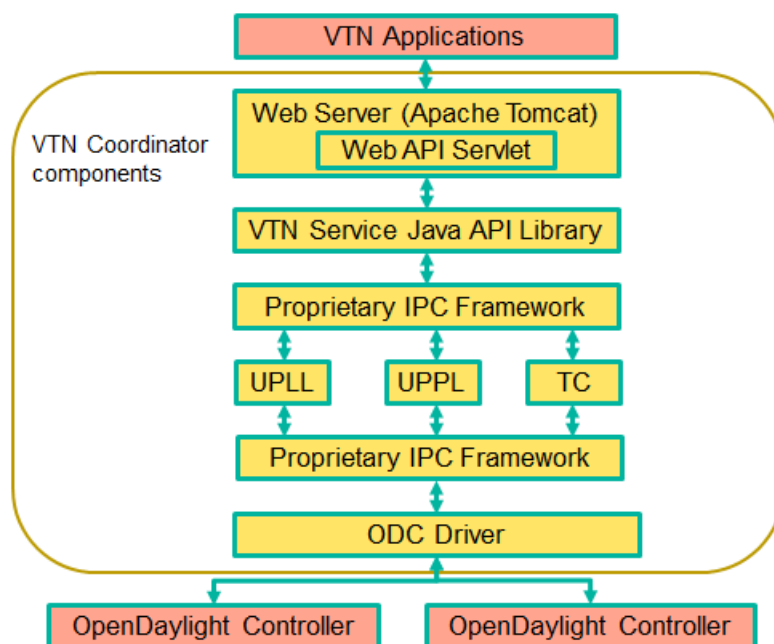
Figure 22.1. VTN Architecture

VTN Coordinator

The VTN coordinator is an external application that provides a REST interface to user to use the VTN Virtualization. It interacts with VTN Manager plugin to implement the user configuration. It is also capable of multiple controller orchestration. realizes Virtual Tenant Network (VTN) provisioning in OpenDaylight Controllers (ODC). In the OpenDaylight architecture VTN Coordinator is part of the network application, orchestration and services layer. VTN Co ordinator has been implemented as an external application to the OpenDaylight controller. This component is responsible for the VTN virtualization. VTN Coordinator will use the REST interface exposed by the VTN Manger to realize the virtual network using the OpenDaylight controller. It uses OpenDaylight APIs (REST) to construct the virtual network in ODCs. It provides REST APIs for northbound VTN applications and supports virtual networks spanning across multiple ODCs by coordinating across ODCs.

Figure 22.2. VTN Coordinator Architecture

VTN Coordinator Architecture



VTN Coordinator has the following components:

- **VTN API** - VTN Web API module provides the north bound REST API interface for VTN applications. For more information about VTN API, see [the section called "VTN Service Java API Library" \[317\]](#).
- **Transaction Coordinator (TC)** - The TC module is a two Phase commit coordinator module that provides the two phase commit coordination functionality for VTN coordinator components.
- **Unified Provider Physical Layer (UPPL)** - The UPPL module is a physical network provisioning/monitoring module. This module is a sub component of the VTN coordinator and provides the physical network provisioning and monitoring functionality.
- **Unified Provider Logical Layer (UPLL)** - The UPLL is a virtual network provisioning/monitoring module. This module is a sub component of the VTN coordinator and provides the Virtual network provisioning and monitoring functionality.
- **OpenDaylight Controller Driver (ODC Driver)** - The OpenDaylight controller interface Module is a sub component of the VTN coordinator and provides mechanisms to provision and monitor virtual networks in the OpenDaylight controller.

Communication Framework

You could communication with and from the VTN using the following:

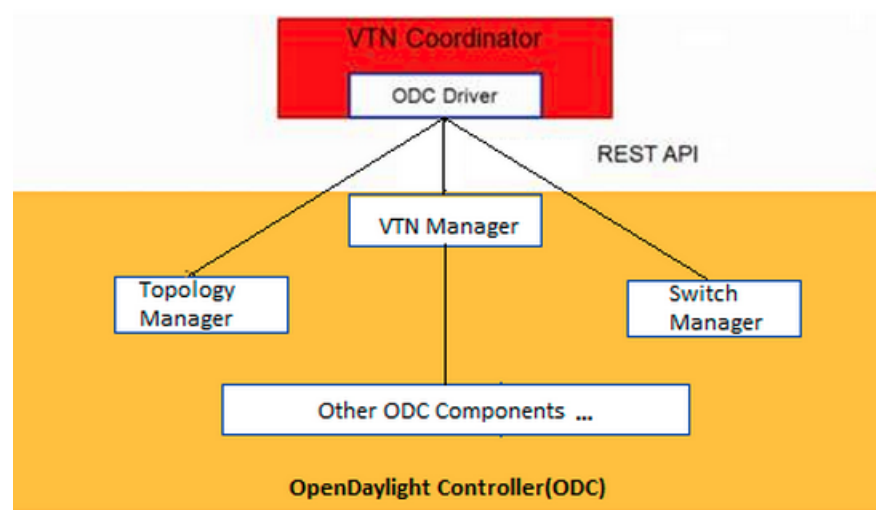
- **Internal communication**

- Proprietary IPC framework
- **External communication**
 - North Bound – REST API
 - South Bound – OpenDaylight API

VTN Manager

A OpenDayLight Controller Plugin that interacts with other modules to implement the components of the VTN model. It also provides a REST interface to configure VTN components in ODL controller. VTN Manager is implemented as one plugin to the OpenDayLight controller. This provides a REST interface to create/update/delete VTN components. The user command in VTN Coordinator is translated as REST API to VTN Manager by the ODC Driver component. In addition to the above mentioned role, it also provides an implementation to the Openstack L2 Network Functions API.

Figure 22.3. VTN Manager Architecture



Function Outline

The table identifies the functions and the interface used by VTN Components:

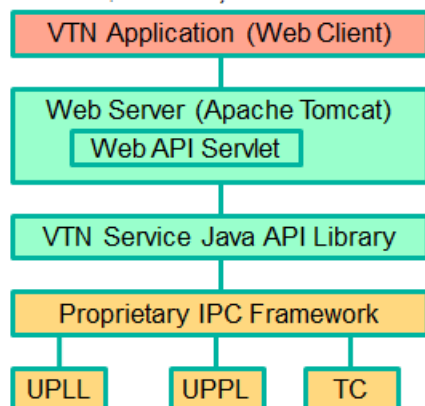
| Component | Interface | Purpose |
|------------------|----------------------------|--|
| VTN Manager | RESTful API | Configure VTN Virtualization model components in OpenDayLight |
| VTN Manager | Neutron API implementation | Handle Networks API from OpenStack (Neutron Interface) |
| VTN Co ordinator | RESTful API | (1) Uses the Restful interface of VTN Manager and configures VTN Virtualization model components in OpenDayLight. (2) Handles multiple controller orchestration. (3) Provides API to read the physical network details. See samples for usage. — |

OpenDaylight Virtual Tenant Network (VTN) API Overview

The VTN API module is a sub component of the VTN coordinator and provides the north bound REST API interface for VTN applications. It consists of two subcomponents:

- the section called “Web Server” [316]
- the section called “VTN Service Java API Library” [317]

Figure 22.4. VTN Coordinator Architecture



Web Server

The Web Server module handles the REST APIs received from the VTN applications. It translates the REST APIs to the appropriate Java APIs.

The main functions of this module are:

- Starts via the startup script catalina.sh.
- VTN Application sends HTTP request to Web server in XML or JSON format.
- Creates a session and acquire a read/write lock.
- Invokes the the section called “VTN Service Java API Library” [317] corresponding to the specified URI.
- Returns the response to the VTN Application.

WebServer Class Details

The table below lists the classes available for Web Server module and its descriptions:

| Class Name | Description |
|-------------|--|
| InitManager | It is a singleton class for executing the acquisition of configuration information from properties file, log initialization, initialization of the section called “VTN Service Java API Library” [317]. Executed by init() of VtnServiceWebAPIServlet. |

| Class Name | Description |
|----------------------------|---|
| ConfigurationManager | Maintains the configuration information acquired from properties file. |
| VtnServiceCommonUtil | Utility class |
| VtnServiceWebUtil | Utility class |
| VtnServiceWebAPIServlet | Receives HTTP request from VTN Application and calls the method of corresponding VtnServiceWebAPIHandler. Inherits class HttpServlet, and overrides doGet(), doPost(), doDelete(), doPost(). |
| VtnServiceWebAPIHandler | Creates JsonObject(com.google.gson) from HTTP request, and calls method of corresponding VtnServiceWebAPIController. |
| VtnServiceWebAPIController | Creates RestResource() class and calls UPLL API/UPPL API through Java API. At the time of calling UPLL API/UPPL API, performs the creation/deletion of session, acquisition/release of configuration mode, acquisition/release of read lock by TC API through Java API. |
| DataConverter | Converts HTTP request to JsonObject and JsonXML to JSON. |

VTN Service Java API Library

It provides the Java API library to communicate with the lower layer modules in the VTN coordinator.

The main functions of this library are:

- Creates an IPC client session to the lower layer.
- Converts the request to IPC framework format.
- Invokes the lower layer API (i.e. UPPL API, UPLL API, TC API).
- Returns the response from the lower layer to the web server
- VTN Service Java API Library Class Details*

The table below lists the classes available for VTN Service Java API library module and its descriptions:

| Class Name | Description |
|-------------------------|---|
| VtnServiceInitManager | It is a Singleton class for executing the acquisition of configuration information from properties file, log initialization. Executed by init() of Web API Servlet. |
| VtnServiceConfiguration | Class to maintain the configuration information acquired from properties file. |
| IpcConnPool | Class that mains Connection pool of IPC. |
| IpcChannelConnection | Class that mains Connections of IPC. |
| RestResource | The class that will be interface for Web API Servlet. Implementation of Interface VtnServiceResource. |
| AnnotationReflect | Performs the mapping of path filed value of RestRsource class and xxxResource class. |
| xxxResource | The class that is created according to the path filed value of RestResource. (vtnResource, VBridgeResource etc) Inherits abstract class AbstractResource. |

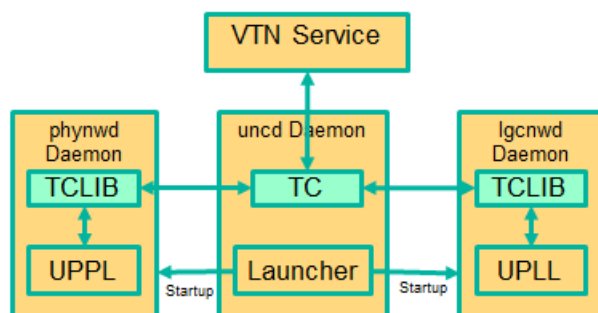
| Class Name | Description |
|--------------------------------------|---|
| xxxResourceValidator CommonValidator | The class that performs the appropriateness check of values specified in the path, query, request field of RestResource class. |
| IpcPhysicalResponseFactory | The class to create JsonObject from the response received from the section called "VTN Unified Provider Logical Layer (UPLL)" [322]. |
| IpcRequestProcessor | Sends request to the section called "VTN Unified Provider Logical Layer (UPLL)" [322] or the section called "VTN Unified Provider Logical Layer (UPLL)" [322] through proprietary IPC Framework. UPLL API and UPPL APIs are implemented on proprietary IPC Framework, and request/response is defined by special interface called as Key Interface. |
| IpcRequestPacket | The class that maintains the request to be sent to the section called "VTN Unified Provider Logical Layer (UPLL)" [322]/the section called "VTN Unified Provider Logical Layer (UPLL)" [322]. |
| IpcStructFactory | The class to create Key Structure and Value Structure that will be included in the request to be sent to the section called "VTN Unified Provider Logical Layer (UPLL)" [322]/the section called "VTN Unified Provider Logical Layer (UPLL)" [322]. |

VTN Transaction Coordinator (TC) Overview

The TC module provides the two phase commit coordination functionality for VTN coordinator components. It consists of two subcomponents

- Transaction Coordinator (TC)
- Transaction Coordinator Library (TCLIB)

Figure 22.5. VTN Transaction Co ordinator (TC) Architecture



Transaction Coordinator (TC)

The Transaction Coordinator module implements the two phase commit operation.

The main functions of this module are:

- TC is started from uncd daemon during startup of VTN coordinator.
- Responsible for two phase commit operation in VTN

- Receives requests from [the section called “VTN Service Java API Library” \[317\]](#) during Commit and Audit operations.
- Invokes lower layer TCLIB API (i.e. UPLL API, UPPL API or ODC Driver API) via IPC framework.

Transaction Coordinator (TC) Class Details

The table below lists the classes available for TC module and its descriptions:

| Class Name | Description |
|--------------|---|
| TcModule | Main interface which offers the services to VTN Service library. It also handles state transitions. |
| TcOperations | Base class that services every operation request in TC. |
| TcMsg | The message to be sent for every operation has different characteristics based on the type of message. This base class will provide methods to handle different types of messages to the intended recipients. |
| TcLock | The exclusion control class, an object of TcLock is contained in TcModule and used for every operation. |
| TcDbHandler | Utility class for TC database operations. |
| TcTaskqUtil | Utility class for taskq used in TC for driver triggered audit and read operations. |

Transaction Coordinator Library

It provides the Java API library to communicate with the lower layer modules in the VTN coordinator.

The main functions of this library are:

- TCLIB will be loaded as a module in UPLL, UPPL and ODC Driver daemon.
- Responsible for handling messages to the daemons from TC.
- The daemons will install their handler with TCLIB, the handlers will be invoked on receiving messages from TC.

Transaction Co ordinator Library Class Details

The table below lists the classes available for Transaction Co ordinator library module and its descriptions:

| Class Name | Description |
|----------------|---|
| TcLibModule | Main class which handles requests from TC module. |
| TcLibInterface | Abstract class which every module implements to interact with TC module. Member of TcLibModule. |
| TcLiBMsgUtil | Internal utility class for extracting session attributes of every request from TC. |

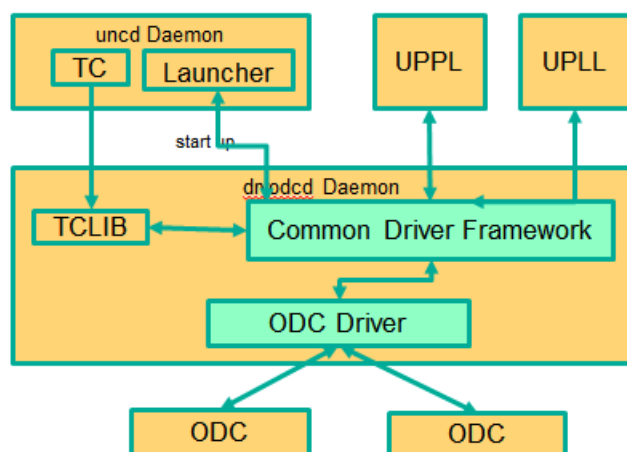
VTN OpenDaylight Controller Driver (ODC Driver) Overview

The ODC driver module is a sub component of the VTN coordinator and provides mechanisms to provision and monitor virtual networks and monitor physical networks in

the OpenDaylight controller. ODC driver is started during startup of VTN coordinator. It consists of two sub components:

- Common Driver Framework (CDF)
- ODC Driver

Figure 22.6. VTN ODC Driver Architecture



Common Driver Framework (CDF)

CDF provides a controller independent processing of the messages sent from UPLL and UPPL modules.

The main functions of the CDF module are:

- Isolate the driver modules from processing messages sent by UPLL and UPPL modules.
- Provide interfaces to the driver module to install their commands for various operations on the controller (eg: VTN creation).
- Provide controller management and support different types of controllers.
- Parse messages and invoke driver methods with appropriate parameters.
- Provide interface for different drivers to install command handlers.
- Simplify transaction processing with simplified transaction functions for vote and commit operations.
- Support for parallel update operation across many controllers.
- The framework can be extended to support all driver modules in a common daemon or individual daemons.

CDF is implemented using the following modules:

- **vtndrvintf**: Implements the features of CDF listed above.

Class Details The following table lists the class details for vtndrvintf module:

| Class Name | Description |
|---------------------|---|
| VtnDrvIntf | Inherited from Module class and provides the entry point for messages from platform. Provides interfaces to add drivers for different types of controllers. |
| KtHandler | Abstract interface for handling different message types. |
| KtRequestHandler | Template implementation of KtHandler to process all messages from platform. |
| DriverTxnInterface | Common transaction handling for drivers. |
| ControllerFramework | Provides methods to add/delete/update Controllers to the VTN Coordinator. Periodic monitoring of controllers |

- **vtncacheutil**: Utility module that provides interfaces for caching configuration entries to validate as a whole and then later commit

Class Details The following table lists the class details for vtncacheutil module:

| Class Name | Description |
|----------------|---|
| keytree | Cache container that provides interfaces to append config to cache. |
| CommonIterator | Provides methods to iterate the elements in cache, the option to iterate in VTN hierarchical order is also available. |

ODC Driver

The ODC driver module implements the interfaces for controller connection management and virtual network provisioning and monitoring in the ODC controller. The request will be translated to the appropriate REST APIs and sent to the controller. ODC driver is capable of translating the VTN Operations as Commands to VTN Manager in the ODL.

The above features are implemented using these modules

- **restjsonutil**: Utility module that provides services for JSON build/parse and handling REST Request/Response.

The following table lists the class details for restjsonutil module:

| Class Name | Description |
|----------------|--|
| HttpClient | Interface to set up and maintain a connection to an HTTP Web service |
| RestClient | Interface to handle request/response on a REST Interface |
| JsonBuildParse | Interface for building/parsing the JSON strings for communication |

- **odcdriver**:
 - Implements the interfaces exposed by CDF
 - Registers the driver for controllers of type : ODC (OpenDaylight Controllers)
 - Uses the restjsonutil to communicate

The following table lists the class details for restjsonutil module:

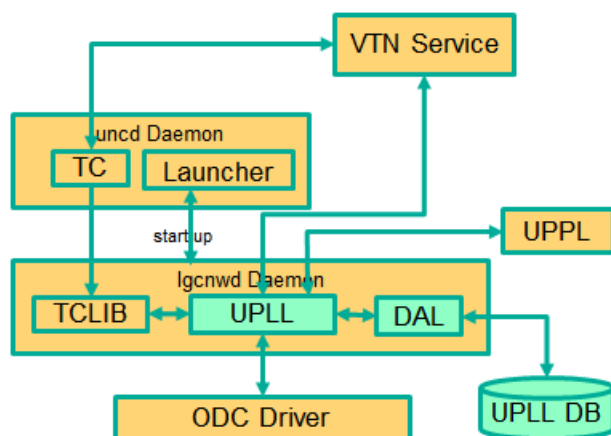
| Class Name | Description |
|-----------------|--|
| OdcModule | Module implementation of odc driver, registers itself as diver for controllers of ODL type |
| ODCController | Implements the various methods according to the features of the ODL Controller. |
| ODCVTNCommand | Handle Create/Update/Delete/Read requests for VTN. |
| ODCVBRCommand | Handle Create/Update/Delete/Read requests for vBridge . |
| ODCVBRIfCommand | Handle Create/Update/Delete/Read requests for vBridge interfaces. |

VTN Unified Provider Logical Layer (UPLL)

The UPLL module is a sub component of the VTN coordinator and provides the Virtual network provisioning and monitoring functionality. It consists of two sub components:

- UPLL
- DAL

Figure 22.7. VTN UPLL Architecture



UPLL Functionalities

The main functions of this module are:

- UPLL is started from lgcnwd daemon during startup of VTN coordinator.
- Interacts with TC, UPLL and ODC Driver using IPC framework.
- Receives virtual network configuration Create/Update/Delete/Read requests from VTN service.
- Maintains the startup, candidate, and running configurations and state information in an external database
- Performs the Setup/Commit/Abort operations as instructed by TC.

- Connects to southbound controllers via ODC Driver.
- Constructs and maintains the virtual network topology using the configuration and notifications (events and alarms) received from controller platforms.
- Supports Audit and Import functionality for the virtual network configurations.

UPLL Class Details

The table below lists the classes available for UPLL module and its descriptions:

| Class Name | Description |
|---------------------------|---|
| UpplConfigSvc | UpplConfigService is a service layer implementation for UPLL. It provides UPLL service to VTN Service and handles all service requests. It also registers with UPPL and Drivers for notifications. |
| UpplIpcEventHandler | Handler for IPC events. |
| UpplConfigMgr | UpplConfigMgr is the core implementation class for configuration services and transaction services including audit and import. |
| TcLibIntfImpl | This an implementation class which implements the TcLibInterface provided by TC. This implementation class, for each virtual function, will invoke corresponding UpplConfigMgr function. |
| MoCfgServiceIntf | Interface class for Edit/Read/Control operations. |
| MoTxServiceIntf | Interface class for normal transaction operations. |
| MoAuditServiceIntf | Interface class for audit operations. |
| MoImportServiceIntf | Interface class for import operations. |
| MoDbServiceIntf | Interface class for database operations. |
| MoManager | Base class for Key tree specific implementation. |
| CtrlrMgr | Stores the controllers as notified by Physical. UPLL stores the controller type and "invalid config" alarm status against each known controller type. |
| ConfigVal | Class for value structure of any key type. This class allows list of values to be specified. |
| ConfigKeyVal | Handler for IPC events |
| UpplConfigMgr | Class for additional data after the request/response header in messages corresponding to configuration operations. This class allows nesting of key types and values. For one key type many values can be specified and sequence of such <key, value, ...> tuples can be encapsulated with one ConfigKeyVal |
| ConfigNotification | Implements config notification. |
| ConfigNotifier | Implements buffering and sending of config notifications. Also provides API for OperStatus change notification. |
| IpcUtil | Provides various IPC wrappers over the IPC framework. |
| IpctSt | Provides wrappers for data sent over IPC. |
| Key type specific classes | Implements the Key type handling functionality for all key types. |

DAL Functionalities

The DAL Module implements the abstraction layer for the Database.

DAL Class Details

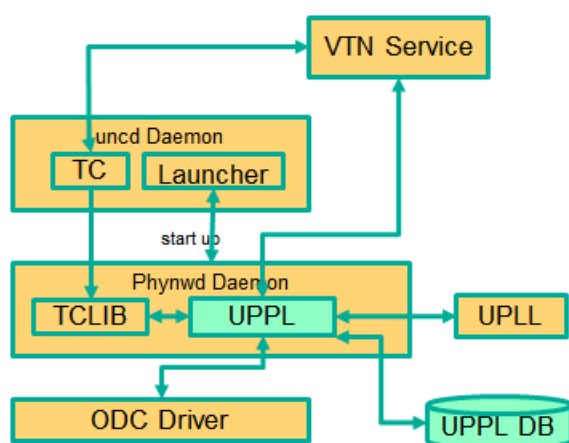
The table below lists the classes available for DAL module and its descriptions:

| Class Name | Description |
|-------------------|---|
| DalBindColumnInfo | Contains column_info for each column_index (column_index, app_data_type, dal_data_type, app_array_size). Contains bind_info (app_out_addr, db_in_out_addr, db_match_addr, io_type). Allocates memory in DB and copies input/match application data. Copies result from DB to application data. |
| DalBindInfo | Contains bind_info for all columns in a table (table_index, list of DalBindColumnInfo. Provides API to UPLL to bind the input/output/match address to DB And to copy result back to application. |
| DalCursor | Holds cursor information. Holds cursor data to fetch result one by one in case of multi-result query. Provides API to UPLL to fetch the result from cursor and destroy the cursor. Creation of cursor will be done in DalOdbcMgr based on the Query API. |
| DalQueryBuilder | Contains list of Query Templates and generates Query based on user inputs. |
| DalErrorHandler | Process SQL errors and maps to corresponding DB result code. |
| DalOdbcMgr | Provides APIs to UPLL for Connection/Disconnection, Commit/Rollback operation, Cursor fetch/Close cursor, All Single/Multiple result queries Diff, Copy Queries. |

VTN Unified Provider Physical Layer (UPPL)

The UPPL module is a sub component of the VTN coordinator and provides the Physical network provisioning and monitoring functionality.

Figure 22.8. VTN UPPL Architecture



UPPL Functionalities

UPPL provides the following functionalities:

- UPPL is started from phynwd daemon during startup of VTN coordinator.
- Interacts with TC, UPLL and ODC Driver using IPC framework

- Receives Controller, Domain and Boundary Create/Update/Delete/Read requests from VTN Services
- Maintains the startup, candidate, and running configurations and state information in an external database
- Performs the setup/commit/abort operations as instructed by TC.
- Connects to southbound controllers via ODC Driver
- Constructs physical topology using the notifications (events and alarms) from controller platform.
- Informs UPLL about the controller addition/deletion and operational status changes of physical topology objects.

UPPL Class Details

The table below lists the classes available for UPPL module and its descriptions:

| Class Name | Description |
|--|--|
| PhysicalLayer | It's a singleton class which will instantiate other UPPL's classes. This class will be inherited from base module in order to use the Core features and IPC service handlers. |
| PhysicalCore | Class that is responsible for processing requests from VTN Transaction Coordinator . It also: * Processes the configuration and capability file. * Responsible for sending alarm to node manager. * Responsible for receiving requests from north bound. |
| IPCConnectionManager | It is responsible for processing the requests received via IPC framework. It contains separate classes to process request from VTN_Service_Java_API_library, Unified Provider Logical Layer (UPLL), OpenDaylight Controller Driver. For more information about the modules mentioned, see VTN Coordinator Architecture |
| ODBCManager | It is a singleton class which performs all database services. |
| InternalTransactionCoordinator | It is responsible for parsing the IPC structures and forward it to the various request classes like ConfigurationRequest, ReadRequest, ImportRequest etc. |
| ConfigurationRequest | It is responsible to process the Create, Delete and Update operations received from the section called "VTN Service Java API Library" [317] . |
| ReadRequest | It is responsible to process all the read operations. |
| Kt_Base, Kt_State_Base and respective Kt classes | These classes perform the functionality required for individual key type. |
| TransactionRequest | It is responsible for performing the various functions required for each phase of the Transaction Request received from Transaction Coordinator during User Commit/Abort. |
| AuditRequest | It is responsible for performing functions related to audit request. |
| ImportRequest | It is responsible for performing functions related to import request. |
| SystemStateChangeRequest | It is responsible for performing functions when the section called "VTN Coordinator" [313] state is moved to active or standby. |

| Class Name | Description |
|------------------------|--|
| DBConfigurationRequest | It is responsible for processing various Database operations like Save/Clear/Abort |

Usage Examples

- [L2 Network using Single Controller](#)

Hacking VTN Coordinator

Prior preparation

1. Arrange a server with any one of the supported 64-bit OS environment.

- RHEL 6 / 7
- CentOS 6 / 7
- Fedora(19/20)
- Ubuntu (12.04/12.10/13.04/14.04)
- Install necessary packages.
- RHEL/Fedora/Cent OS

```
yum install make glibc-devel gcc gcc-c++ boost-devel openssl-devel \  
ant perl-ExtUtils-MakeMaker unixODBC-devel perl-Digest-SHA uuid libxslt  
libcurl libcurl-devel git
```

- Ubuntu 13.10/14.04

```
apt-get install pkg-config gcc make ant g++ maven git libboost-dev  
libcurl4-openssl-dev \  
libjson0-dev libssl-dev openjdk-7-jdk unixodbc-dev xmlstarlet
```

- Ubuntu 12.04

```
apt-get install pkg-config gcc make ant g++ maven git libboost-dev  
libcurl4-openssl-dev \  
libssl-dev openjdk-7-jdk unixodbc-dev
```



Note

Install libjson0-dev from packages of ubuntu versions (>12.04)

- Install JDK 7, and add the JAVA_HOME environment variable (Only for RHEL/Cent OS/Fedora)
- RHEL 6.1/Cent OS 6.1

1. Download Oracle JDK 7 from the following page, and install it. <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

2. Set JAVA_HOME to the location of the JDK.

```
export JAVA_HOME=/usr/java/default
```

- RHEL 6.4,Cent OS 6.4 ,Fedora (17,20)

1.Install OpenJDK 7.

```
yum install java-1.7.0-openjdk-devel
```

2. Set JAVA_HOME to the location of the JDK.

```
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk.x86_64
```

- Preperation for Execution
- RHEL 6/Fedora/Cent OS 6

```
Download the following PostgreSQL 9.1 files (latest versions) from http://  
yum.postgresql.org/9.1/redhat/rhel-6.4-x86_64/ (RHEL 6.4) or http://yum.  
postgresql.org/9.1/redhat/rhel-6.1-x86_64/ (RHEL 6.1)and install.  
postgresql91-libs  
postgresql91  
postgresql91server  
postgresql91-contrib  
postgresql91-odbc
```



Note

The above procedure caters only for RHEL 6, For Fedora and CentOS, Please visit <http://yum.postgresql.org> and install the corresponding rpm for the particular version of Linux distro

- RHEL 7/Cent OS 7

1. Download and install the Postgres 9.3 by installing the rpm NOTE: For RHEL 7, Please install the below rpm

```
rpm -ivh http://yum.postgresql.org/9.3/redhat/rhel-7-x86_64/pgdg-  
redhat93-9.3-1.noarch.rpm  
yum install postgresql93-libs postgresql93 postgresql93-server postgresql93-  
contrib postgresql93-odbc
```



Note

For Cent OS 7, Please install the below rpm

```
rpm -ivh http://yum.postgresql.org/9.3/redhat/rhel-7-x86_64/pgdg-centos93-9.  
3-1.noarch.rpm  
yum install postgresql93-libs postgresql93 postgresql93-server postgresql93-  
contrib postgresql93-odbc
```

- Ubuntu 13.10/12.04

```
apt-get install postgresql-9.1 postgresql-client-9.1 postgresql-client-  
common postgresql-contrib-9.1 odbc-postgresql
```


- Install Maven. (RHEL/Cent OS/Fedora)
 1. Download Maven from the following page and install it following the instruction in the page.
 2. <http://maven.apache.org/download.cgi>
 3. Install gtest-devel, json-c libraries
- RHEL/Fedora/Cent OS

```
wget <nowiki>http://dl.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm</nowiki>
rpm -Uvh epel-release-6-8.noarch.rpm
yum install gtest-devel json-c json-c-devel
```



Note

For RHEL7 and CentOS7, the json-c is part of the default repo, Please install json-c and json-c-devel from the default repo and gtest-devel alone from the EPEL repo (http://dl.fedoraproject.org/pub/epel/7/x86_64/)

- Ubuntu 13.10/Ubuntu 12.04

```
apt-get install cmake libgtest-dev
cp -R /usr/src/gtest gtest-work
cd gtest-work
cmake CMakeLists.txt
make
sudo cp *.a /usr/lib
cd ..
rm -rf gtest-work
```

Build



Note

User is not required to be mandatorily root, but the user must own the directory /usr/local/vtn

- Example : .The directory should appear as below (assuming the user as "vtn"):

```
# ls -l /usr/local/
drwxr-xr-x. 12 vtn  vtn  4096 Mar 14 21:53 vtn
```

- Download the code from git.

```
git clone ssh://<username>@git.opendaylight.org:29418/vtn.git
git clone https://git.opendaylight.org/gerrit/p/vtn.git
```

- Build and install VTN Coordinator.

```
cd vtn/coordinator
mvn -f dist/pom.xml install
tar -C/ -jxvf dist/target/distribution.vtn-coordinator-6.1.0.0-SNAPSHOT-bin.tar.bz2
```

Run VTN Coordinator

Prior preparation

To change the port

1. By Default coordinator will listen on port 8083
2. To change the listening port modify the TOMCAT_PORT in below file

```
/usr/local/vtn/tomcat/conf/tomcat-env.sh.
```

1. Set up the DB.

```
/usr/local/vtn/sbin/db_setup
```

If any problem in setting up db, Please visit: * [https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Network_\(VTN\):Installation:Troubleshooting#After_executing_db_setup.2C](https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Network_(VTN):Installation:Troubleshooting#After_executing_db_setup.2C)

Start

1. Start VTN Coordinator.

```
/usr/local/vtn/bin/vtn_start
```

2. Execute the following commands while stopping.

```
/usr/local/vtn/bin/vtn_stop
```

WebAPI

- VTN Coordinator version information will be displayed if following command is executed when VTN has started successfully.

```
curl --user admin:adminpass -H 'content-type: application/json' -X GET \
'http://127.0.0.1:8083/vtn-webapi/api_version.json'
```

- The expected response message:

```
{"api_version":{"version":"V1.2"}}
```

Hacking VTN Manager

Installing VTN Manager from source code

Prior preparation. VTN Manager is a set of OSGi bundles running in OpenDaylight controller in Karaf platform, so prior preparation for installing VTN Manager is the same as OpenDaylight controller.

Please see https://wiki.opendaylight.org/view/OpenDaylight_Controller:Installation.

Get VTN Source

The below procedure assumes, You are installing OpenDaylight Controller with VTN Manager on your local Linux machine.

- Download the code from the Git repository of VTN Project.

```
git clone ssh://<username>@git.opendaylight.org:29418/vtn.git
or
git clone https://git.opendaylight.org/gerrit/p/vtn.git
```

The following instructions assume you put the code in directory `${vtn_dir}`.

```
${VTN_DIR}=<Top of VTN source tree>
```

- Build the code of VTN Manager.

```
cd ${VTN_DIR}/manager
mvn clean install
```

NOTE: This command will build the local Karaf distribution in the folder `manager/dist/karaf/target/assembly`.

Build and Run the Controller with VTN Manager

Build



Note

You will need direct internet connection for this build to be successful.

```
cd ${VTN_DIR}/manager
mvn clean install -DskipTests
```

Note: SkipTests will not execute the Unit Tests and Integration Tests while compilation

Hacking VTN Manager(Helium)

Installing VTN Manager from source code

Prior preparation. VTN Manager is a set of OSGi bundles running in OpenDaylight controller in Karaf platform, so prior preparation for installing VTN Manager is the same as OpenDaylight controller.

Get VTN Source

The below procedure assumes, You are installing OpenDaylight Controller with VTN Manager on your local Linux machine.

- Download the code from the Git repository of VTN Project.

```
git clone ssh://<username>@git.opendaylight.org:29418/vtn.git
or
git clone https://git.opendaylight.org/gerrit/p/vtn.git
```

The following instructions assume you put the code in directory `${vtn_dir}`.

```
${VTN_DIR}=<Top of VTN source tree>
```

- Build the code of VTN Manager.

```
cd ${VTN_DIR}/manager
mvn clean install
```



Note

This command will build the local Karaf distribution in the folder manager/dist/karaf/target/assembly.

Build and Run the Controller with VTN Manager

Build karaf distribution



Build

You will need direct internet connection for this build to be successful.

```
cd ${VTN_DIR}/manager/dist-karaf for karaf binaries
mvn clean install
```

Run:

- `cd ${vtn_dir}/manager/dist-karaf/target/assembly`
- `./bin/karaf` Note: By default the VTN packages will be installed



Note

* After successful build, the directory `${vtn_dir}/manager/dist-karaf` will have assembly and tar and zip versions of binaries * User can either choose to run from assembly or copy the tar or zip to required locations and execute there. * Test if VTN Manager is running * In the karaf prompt, type the below command to ensure that vtn packages are installed.

Build Non-Karaf distribution

Build.

```
cd ${VTN_DIR}/manager/dist to build the non-karaf binaries
mvn clean install
cd ${VTN_DIR}/manager/dist/target/distribution.vtn-manager-0.2.2-SNAPSHOT-
osgipackage/opendaylight
```

Run.

```
./run.sh of10 (If you want to enable only OpenFlow 1.0)
./run.sh of13 (If you want to enable all supported openflow versions)
```

Verify if VTN is running

- In the karaf prompt, type the below command to ensure that vtn packages are installed.
`feature:list i | grep vtn`

REST API

VTN Manager provides REST API for virtual network functions.

For detailed information about REST API specifications, see [https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Network_\(VTN\):VTN_Manager:RestApi](https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Network_(VTN):VTN_Manager:RestApi)

Here is an example of how to create a virtual tenant network.

```
curl --user "admin":"admin" -H "Accept: application/json" -H \
"Content-type: application/json" -X POST \
http://localhost:8080/controller/nb/v2/vtn/default/vtns/Tenant1 \
-d '{"description": "My First Virtual Tenant Network"}'
```

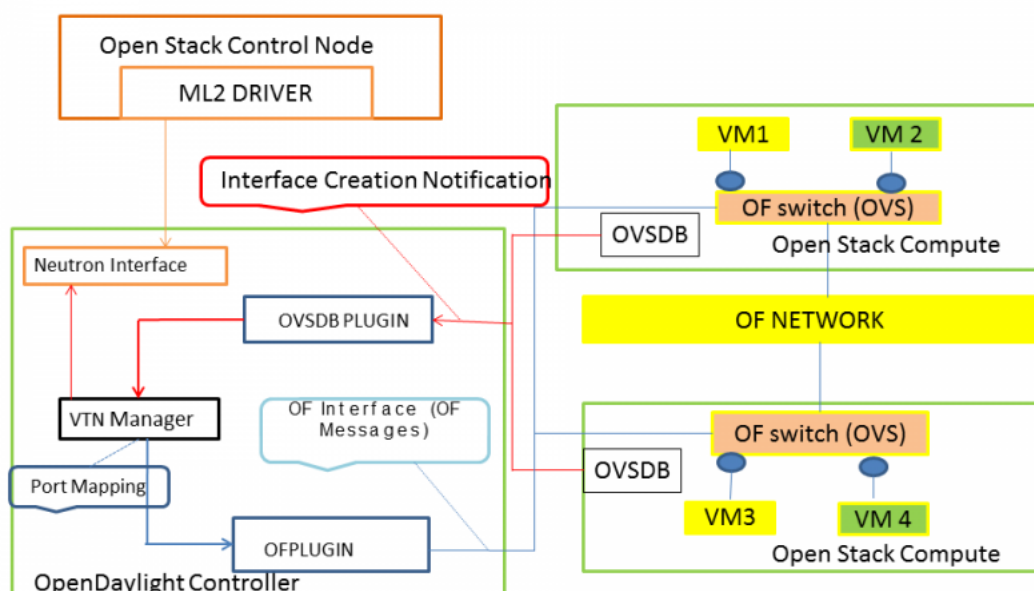
You can check the list of all tenants by executing the following command.

```
curl --user "admin":"admin" -H "Accept: application/json" -H \
"Content-type: application/json" -X GET \
http://localhost:8080/controller/nb/v2/vtn/default/vtns
```

Openstack Support Developer Guide

The implementation describes ODL controller with VTN manager feature provides the network service for OpenStack. VTN manager utilizes the OVSDb southbound service and the neutron for this implementation. The below diagram depicts the communication of ODL Controller and two virtual network connected by a OpenFlow switch using this implementation.

Figure 22.9. OpenStack Architecture



Configuration

Please refer https://wiki.opendaylight.org/view/Release/Helium/VTN/User_Guide/OpenStack_Support for the use and configuration.

OpenStack Configuration Details

Follow the below steps to configure devstack with ODL controller:

Control Node

- Disable nova network and enable neutron.
- Enable service odl-compute (Optionally you can enable odl-server also, which will download and deploy ODL in the particular node).
- Set ODL_MGR_IP with IP Address of ODL.
- Make all network types as local, as there is no support for vlan in Helium.
- Add the ODL neutron URL for devstack to post requests.

Compute Node

- To configure Compute node follows the steps b,c and d of Control Node.
- Very few services are enabled like neutron,n-cpu and rabbit.
- The neutron URL is not needed to be set in compute node.
- The Controller node services are added as the service hosts for various services.

For more details: Please refer to <https://wiki.openstack.org/wiki/NeutronDevstack>

ODL Configuration Details

The implementation makes use of a configuration file vtn.ini in the OpenDaylight configuration directory. The name of the bridge in Openstack OVS, the physical interface device name to be used for VM's are mentioned in this file. The vtn.ini is created with the below parameters and the explanation for them is as follows

```
bridgename=br-int
portname=eth1
protocols=OpenFlow13
failmode=secure
```

- The values of the configuration parameters must be changed based on the user environment. Especially, portname should be carefully configured. Because if the value is wrong, OpenDaylight controller fails to forward packets. Other parameters works fine as is for general use cases. .bridgename
- The name of the bridge in Open-v-Switch, that will be created by OpenDaylight Controller.

- It must be "br-int". .portname
- The name of the port that will be created in the vbridge in Open vSwitch.
- This must be the same name of the interface of OpenStack Nodes which is used for interconnecting OpenStack Nodes in data plane.(in our case:eth1)
- By default, if vtn.ini is not created, VTN uses ens33 as portname. .protocols
- OpenFlow protocol through which OpenFlow Switch and Controller communicate.
- The values can be OpenFlow13 or OpenFlow10. .failmode
- The value can be "standalone" or "secure".
- Please use "secure" for general use cases.

If the file is not found, the following default values are assumed

```
bridgename=br-int
portname=eth1
protocols=OpenFlow13
failmode=secure
```

Implementation details

VTN Manager:

It subscribes to the events from OVS and also implements the neutron requests received by ODL Controller.

Functional Behavior

StartUp:

- The ML2 implementation for ODL Controller will ensure that when OVS is started, the ODL_IP_ADDRESS configured will be set as manager.
- When ODL Controller receives the update of the OVS in the port:6640, (Manager port) VTN Manager handles the event and adds a bridge with required port mappings to the Open-V-Switch at openstack node.
- When neutron starts up, a New Network create is POSTED to ODL, for which VTN Manager creates a VTN.
- Network and Sub-Network Create: Whenever a new sub network is created, VTN Manager will handle the same and create a vbridge under the VTN.
- VM Creation in openstack: The interface mentioned as integration bridge in the configuration file, will be added with more interfaces on creation of a new VM in Openstack and network is provisioned for it by VTN neutron bundle.The addition of new PORT is captured by VTN Manager and it creates a vbridge interface with port mapping for the particular port.Now, when the VM starts to communicate with other

VM's created, VTN Manger will install flows in the OVS and other openflow switches to facilitate communication between VM(s).

Note.

To use this feature, VTN feature should be installed

Reference

https://wiki.opendaylight.org/images/5/5c/Integration_of_vtn_and_ovsdb_for_helium.pdf

23. YANG Tools

Table of Contents

| | |
|--|-----|
| Prerequisites for YANG Tools Project | 336 |
| Pulling code using ssh | 336 |
| Pulling code using https | 336 |
| Building the code | 337 |
| Mapping YANG to Java | 337 |
| Additional Packages | 338 |
| Data Interface | 340 |
| Service Interface | 340 |

YANG is a data modelling language used to model configuration and state data manipulated by the Network Configuration Protocol(NETCONF), NETCONF remote procedure calls, and NETCONF notifications. YANG is used to model the operations and content layers of NETCONF.

Prerequisites for YANG Tools Project

- OpenDayLight account [Get an account](#) to push or edit code on the wiki. You can however pull code anonymously.
- Gerrit Setup for code review To use ssh follow instructions on Opendaylight wiki page at: https://wiki.opendaylight.org/view/OpenDaylight_Controller:Gerrit_Setup To use https follow instructions on Opendaylight wiki page at: https://wiki.opendaylight.org/view/OpenDaylight_Controller:Setting_up_HTTP_in_Gerrit
- Maven 3 to import Maven project from Opendaylight Git repository To clone the controller follow instructions at: <https://git.opendaylight.org/gerrit/p/controller.git> To clone the yangtools repositories follow instructions at: <https://git.opendaylight.org/gerrit/p/yangtools.git>



Note

You need to setup Gerrit to access GIT using ssh.

Pulling code using ssh

To pull code using ssh use the following command:

```
git clone ssh://{ODL_USERNAME}@git.opendaylight.org:29418/yangtools.git;(cd
yangtools; scp -p -P 29418 ${ODL_USERNAME}@git.opendaylight.org:hooks/commit-
msg .git/hooks/;chmod 755 .git/hooks/commit-msg;git config remote.origin.push
HEAD:refs/for/master)
```

Pulling code using https

To pull code using https, use the following command:

```
git clone https://git.opendaylight.org/gerrit/p/yangtools.git;(cd yangtools;  
curl -o .git/hooks/commit-msg https://git.opendaylight.org/gerrit/tools/  
hooks/commit-msg;chmod 755 .git/hooks/commit-msg;git config remote.origin.push  
HEAD:refs/for/master)
```

Building the code

To build the code, increase the memory available for Maven. The settings on the Jenkins build server are:

```
export MAVEN_OPTS="-Xmx1024m -XX:MaxPermSize=256m"
```



Important

The top level build line for yangtools project is: `cd yangtools;mvn clean install`

Mapping YANG to Java

This chapter covers the details of mapping YANG to Java.

Name of the Package

To configure your project and generate source code from YANG, edit your projects **pom.xml** and add.opendaylight SNAPSHOT repository for snapshot releases. Currently, only snapshots are available. The name of the package is *org.opendaylight.yang.gen.v1.urn:2:case#module.rev201379*. After replacing digits and JAVA keywords the package name is *org.opendaylight.yang.gen.v1.urn._2._case.module.rev201379*.

The package name consists of the following parts:

- **OpenDaylight prefix** - Specifies the.opendaylight prefix. Every package name starts with the prefix *org.opendaylight.yang.gen.v* that is hardcoded in **BindingGeneratorUtil.moduleNamespaceToPackageName()**.
- **YANG version** - Specifies the YANG version. YANG version is updated through *module* substatement *yang-version*.
- **Namespace** - Specifies the value of *module* subelement and the *namespace* argument value. The namespace characters are *:/:-@\$#'*+,;=*. *character group:/* are replaced with periods (*.*).
- **Revision** - Specifies the concatenation of word *rev* and value of *module* subelement *revision* argument value without leading zeros before month and day. For example: *rev201379*

After the package name is generated check it in if it contains any JAVA key words or digits. If it is so then before the token add an underscore (*_*).

List of key words which are prefixed with underscore:

```
abstract, assert, boolean, break, byte, case, catch, char, class, const,  
continue, default, double, do, else, enum, extends, false, final, finally,
```

```
float, for, goto, if, implements, import, instanceof, int, interface,
long, native, new, null, package, private, protected, public, return,
short, static, strictfp, super, switch, synchronized, this, throw, throws,
transient, true, try, void, volatile, while
```

As an example suppose following yang model:

```
module module {
  namespace "urn:2:case#module";
  prefix "sbd";
  organization "OPEN DAYLIGHT";
  contact "http://www.whatever.com/";
  revision 2013-07-09 {
  }
}
```

Additional Packages

In cases where the superior YANG elements contain specific subordinate YANG elements additional packages are generated. Table below provides details of superior and subordinate elements:

| Superior Element | Subordinate Element |
|--------------------------|---|
| list | list, container, choice |
| container | list, container, choice |
| choice | leaf, list, leaf-list, container, case |
| case | list, container, choice |
| rpc.output and rpc.input | list, container, (choice isn't supported) |
| notification | list, container, (choice isn't supported) |
| augment | list, container, choice, case |

Subordinate elements are not mapped only to JAVA Getter methods in the interface of superior element but, they also generate packages with names consisting of superior element package name and superior element name. In the example YANG model considers the container element *cont* as the direct subelement of the module.

```
container cont {
  container cont-inner {
  }
  list outter-list {
    list list-in-list {
    }
  }
}
```

Container *cont* is the superior element for the subordinate elements *cont-inner* and *outter-list*.

JAVA code is generated in the following structure:

- org.opendaylight.yang.gen.v1.urn.module.rev201379 - package contains element which are subordinate of module
 - Cont.java

- org.opendaylight.yang.gen.v1.urn.module.rev201379.cont - package contains subordinate elements of cont container element
 - ContInner.java
 - OutterList.java

```
container cont {  
  container cont-inner {  
  }  
  list outter-list {  
    list list-in-list {  
    }  
  }  
}
```

list outter-list is superior element for subordinate element *list-in-list*

JAVA code is generated in the following structure:

- org.opendaylight.yang.gen.v1.urn.module.rev201379.cont.outter.list - package contains subordinate elements of outter-list list element
 - ListInList.java

Class and interface name

Some YANG elements are mapped to JAVA classes and interfaces. The name of YANG element may contain various characters which aren't permitted in JAVA class names. Firstly whitespaces are trimmed from YANG name. Next characters space, -, _ are deleted and subsequent letter is capitalized. At the end first letter is capitalized. Transformation example: example-name without capitalization is mapped to ExampleNameWithoutCapitalization

Getters and setters name

In some cases are YANG elements generated as getter or setter methods. This methods are created through class `MethodSignatureBuilder` The process for getter is:

- name of YANG element is converted to JAVA class name style
- the word get is added as prefix
- return type of the getter method is set to element's type substatement value

The process for setter is:

- name of YANG element is converted to JAVA class name style
- word set is added as prefix
- input parameter name is set to element's name converted to JAVA parameter style
- return parameter is set to void

Module

YANG module is converted to JAVA as two JAVA classes. Each of the class is in the separate JAVA file. The names of JAVA files are composed as follows: `<YANG_module_name><Sufix>.java` where `<sufix>` can be data or service.

Data Interface

Data Interface has a mapping similar to container, but contains only top level nodes defined in module.

Service Interface

Service Interface serves to describe RPC contract defined in the module. This RPC contract is defined by rpc statements.

Typedef

YANG typedef statement is mapped to JAVA class. Typedef may contain following substatement:

| Substatement | Argument Mapped to JAVA |
|--------------|-------------------------|
| type | class attribute |
| descripton | is not mapped |
| units | is not mapped |
| default | is not mapped |

Valid Arguments Type

Simple values of type argument are mapped as follows:

| Argument Type | Mapped to JAVA |
|---------------|---|
| boolean | Boolean |
| empty | Boolean |
| int8 | Byte |
| int16 | Short |
| int32 | Integer |
| int64 | Long |
| string | String or, class (if pattern substatement is specified) |
| decimal64 | Double |
| uint8 | Short |
| uint16 | Integer |
| uint32 | Long |
| uint64 | BigInteger |
| binary | byte[] |

Complex values of type argument are mapped as follows:

| Argument Type | Mapped to JAVA |
|---------------|----------------|
| enumeration | enum |

| Argument Type | Mapped to JAVA |
|---------------------|----------------|
| bits | class |
| leafref | ?? |
| identityref | ?? |
| union | class |
| instance-identifier | ?? |

Enumeration Substatement Enum

The YANG enumeration type has to contain some enum substatements. Enumeration is mapped as JAVA enum type (standalone class) and every YANG enum subelement is mapped to JAVA enum's predefined values. Enum substatement can have following substatements:

| Enum's Substatement | Mapped to JAVA |
|---------------------|--|
| description | is not mapped |
| value | mapped as input parameter for every predefined value of enum |

Example of mapping of YANG enumeration to JAVA:

| YANG | JAVA |
|---|---|
| <pre> typedef typedef-enumeration { type enumeration { enum enum1 { description "enum1 description"; value 18; } enum enum2 { value 16; } enum enum3 { } } } </pre> | <pre> public enum TypedefEnumeration { Enum1(18), Enum2(16), Enum3(19); int value; private TypedefEnumeration(int value) { this.value = value; } } </pre> |

Bits's Substatement Bit

The YANG bits type has to contain some bit substatements. YANG Bits is mapped to JAVA class (standalone class) and every YANG bits subelement is mapped to class boolean attributes. In addition are overridden Object methods hash, toString, equals.

| YANG | JAVA | JAVA overridden Object methods |
|---|--|---|
| <pre> typedef typedef-bits { type bits { bit first-bit { description "first-bit description"; position 15; } bit second-bit; } } </pre> | <pre> public class TypedefBits { private Boolean firstBit; private Boolean secondBit; public TypedefBits() { super(); } public Boolean getFirstBit() { return firstBit; } public void setFirstBit(Boolean firstBit) { this.firstBit = firstBit; } } </pre> | <pre> @Override public int hashCode() { final int prime = 31; int result = 1; result = prime * result + ((firstBit == null) ? 0 : firstBit.hashCode()); result = prime * result + ((secondBit == null) ? 0 : secondBit.hashCode()); return result; } @Override public boolean equals(Object obj) { if (this == obj) { return true; } if (obj == null) { </pre> |

| YANG | JAVA | JAVA overridden Object methods |
|------|---|--|
| | <pre> public Boolean getSecondBit() { return secondBit; } public void setSecondBit(Boolean secondBit) { this.secondBit = secondBit; } </pre> | <pre> return false; } if (getClass() != obj. getClass()) { return false; } TypedefBits other = (TypedefBits) obj; if (firstBit == null) { if (other.firstBit != null) { return false; } } else if (!firstBit. equals(other.firstBit)) { return false; } if (secondBit == null) { if (other.secondBit != null) { return false; } } else if (!secondBit. equals(other.secondBit)) { return false; } return true; } @Override public String toString() { StringBuilder builder = new StringBuilder(); builder.append("TypedefBits [firstBit="); builder.append(firstBit); builder.append(", secondBit="); builder.append(secondBit); builder.append("]"); return builder.toString(); } </pre> |

Union's Substatement Type

If type of typedef is union it has to contain type substatements. Union typedef is mapped to class and its type subelements are mapped to private class attributes. For every YANG union subtype si generated own JAVA constructor with a parameter which represent just one attribute. Example to union mapping:

| YANG | JAVA | JAVA overridden Object methods |
|--|--|---|
| <pre> typedef typedef-union { type union { type int32; type string; } } </pre> | <pre> public class TypedefUnion { private Integer int32; private String string; public TypedefUnion(Integer int32) { super(); this.int32 = int32; } public TypedefUnion(String string) { super(); this.string = string; } public Integer getInt32() { return int32; } } </pre> | <pre> @Override public int hashCode() { final int prime = 31; int result = 1; result = prime * result + ((int32 == null) ? 0 : int32. hashCode()); result = prime * result + ((string == null) ? 0 : string. hashCode()); return result; } @Override public boolean equals(Object obj) { if (this == obj) { return true; } if (obj == null) { return false; } } </pre> |

| YANG | JAVA | JAVA overridden Object methods |
|------|--|--|
| | <pre> } public String getString() { return string; } </pre> | <pre> } if (getClass() != obj. getClass()) { return false; } TypedefUnion other = (TypedefUnion) obj; if (int32 == null) { if (other.int32 != null) { return false; } } else if (!int32. equals(other.int32)) { return false; } if (string == null) { if (other.string != null) { return false; } } else if (!string. equals(other.string)) { return false; } return true; } @Override public String toString() { StringBuilder builder = new StringBuilder(); builder.append("TypedefUnion [int32="); builder.append(int32); builder.append(", string="); builder.append(string); builder.append("]"); return builder.toString(); } </pre> |

String Mapping

YANG String can be detailed specified through type subelements length and pattern which are mapped as follows:

| Type subelement | Mapping to JAVA |
|-----------------|--|
| length | not mapped |
| pattern | . list of string constants = list of patterns . list of Pattern objects . static initialization block where list of Patterns is initialized from list of string of constants |

Example of YANG string mapping

| YANG | JAVA | JAVA Overriden Object Methods |
|--|---|--|
| <pre> typedef typedef-string { type string { length 44; pattern "[a][.]*" } } </pre> | <pre> public class TypedefString { private static final List<Pattern> patterns = new ArrayList<Pattern>(); public static final List<String> PATTERN_CONSTANTS = Arrays.asList("[a][.]*"); static { for (String regEx : PATTERN_CONSTANTS) { patterns. add(Pattern.compile(regEx)); } } } </pre> | <pre> @Override public int hashCode() { final int prime = 31; int result = 1; result = prime * result + ((typedefString == null) ? 0 : typedefString.hashCode()); return result; } @Override public boolean equals(Object obj) { if (this == obj) { return true; } } </pre> |

| YANG | JAVA | JAVA Overriden Object Methods |
|------|---|---|
| | <pre> private String typedefString; public TypedefString(String typedefString) { super(); this.typedefString = typedefString; } public String getTypedefString() { return typedefString; } </pre> | <pre> if (obj == null) { return false; } if (getClass() != obj. getClass()) { return false; } TypedefString other = (TypedefString) obj; if (typedefString == null) { if (other.typedefString ! = null) { return false; } } else if (!typedefString. equals(other.typedefString)) { return false; } return true; } @Override public String toString() { StringBuilder builder = new StringBuilder(); builder.append("TypedefString [typedefString="); builder. append(typedefString); builder.append("]"); return builder.toString(); } </pre> |

Container

YANG Container is mapped to JAVA interface which extends interfaces `DataObject`, `Augmentable<container_interface>`, where `container_interface` is name of mapped interface. Example of mapping:

| YANG | JAVA |
|---------------------------------|--|
| <pre> container cont { } </pre> | <pre> public interface Cont extends DataObject, Augmentable<Cont> { } </pre> |

Leaf

Each leaf has to contain at least one type substatement. The leaf is mapped to getter method of superior element with return type equal to type substatement value. Example of mapping:

| YANG | JAVA |
|---|--|
| <pre> module module { namespace "urn:module"; prefix "sbd"; organization "OPEN DAYLIGHT"; contact "http://www.whatever.com/"; revision 2013-07-09 { } leaf lf { type string; } } </pre> | <pre> package org.opendaylight.yang.gen.v1.urn.module. rev201379; public interface ModuleData { String getLf(); } </pre> |

Example of leaf mapping at container level:

| YANG | JAVA |
|--|--|
| <pre> container cont { leaf lf { type string; } } </pre> | <pre> public interface Cont extends DataObject, Augmentable<Cont> { String getLf(); } </pre> |

Leaf-list

Each leaf-list has to contain one type substatement. The leaf-list is mapped to getter method of superior element with return type equal to List of type substatement value. Example of mapping of leaf-list.

| YANG | JAVA |
|--|---|
| <pre> container cont { leaf-list lf-lst { type typedef-union; } } </pre> | <pre> public interface Cont extends DataObject, Augmentable<Cont> { List<TypedefUnion> getLfLst(); } </pre> |

YANG typedef-union and JAVA TypedefUnion are the same as in union type.

List

YANG list element is mapped to JAVA interface. In superior element is generated as getter method with return type List of generated interfaces. Mapping of list substatement to JAVA:

| Substatement | Mapping to JAVA |
|--------------|-----------------|
| Key | Class |

Example of list mapping *outter-list* is mapped to JAVA interface *OutterList* and in *Cont* interface (superior of *OutterList*) contains getter method with return type List<OutterList>

| YANG | JAVA | JAVA Overriden Object Methods |
|--|--|---|
| <pre> container cont { list outter-list { leaf leaf-in-list { type uint64; } leaf-list leaf-list-in-list { type string; } list list-in-list { leaf-list inner-leaf-list { type int16; } } } } </pre> | <pre> ListInList.java package org.opendaylight.yang. gen.v1.urn.module.rev201379. cont.outter.list; import org.opendaylight. yangtools.yang.binding. DataObject; import org.opendaylight. yangtools.yang.binding. Augmentable; import java.util.List; public interface ListInList extends DataObject, Augmentable<ListInList> { List<Short> getInnerLeafList(); } OutterListKey.java package org.opendaylight.yang. gen.v1.urn.module.rev201379. cont; import org.opendaylight.yang. gen.v1.urn.module.rev201379. cont.OutterListKey; import java.math.BigInteger; </pre> | <pre> OutterListKey.java @Override public int hashCode() { final int prime = 31; int result = 1; result = prime * result + ((LeafInList == null) ? 0 : LeafInList.hashCode()); return result; } @Override public boolean equals(Object obj) { if (this == obj) { return true; } if (obj == null) { return false; } if (getClass() != obj. getClass()) { return false; } OutterListKey other = (OutterListKey) obj; if (LeafInList == null) { if (other.LeafInList != null) { return false; } } } </pre> |

| YANG | JAVA | JAVA Overriden Object Methods |
|------|---|---|
| | <pre> public class OutterListKey { private BigInteger LeafInList; public OutterListKey(BigInteger LeafInList) { super(); this.LeafInList = LeafInList; } public BigInteger getLeafInList() { return LeafInList; } } OutterList.java package org.opendaylight.yang. gen.v1.urn.module.rev201379. cont; import org.opendaylight. yangtools.yang.binding. DataObject; import org.opendaylight. yangtools.yang.binding. Augmentable; import java.util.List; import org.opendaylight.yang. gen.v1.urn.module.rev201379. cont.outter.list.ListInList; public interface OutterList extends DataObject, Augmentable<OutterList> { List<String> getLeafListInList(); List<ListInList> getListInList(); /* Returns Primary Key of Yang List Type */ OutterListKey getOutterListKey(); } Cont.java package org.opendaylight.yang. gen.v1.urn.module.rev201379; import org.opendaylight. yangtools.yang.binding. DataObject; import org.opendaylight. yangtools.yang.binding. Augmentable; import java.util.List; import org.opendaylight.yang. gen.v1.urn.module.rev201379. cont.OutterList; public interface Cont extends DataObject, Augmentable<Cont> { List<OutterList> getOutterList(); </pre> | <pre> } else if(!LeafInList. equals(other.LeafInList)) { return false; } return true; } @Override public String toString() { StringBuilder builder = new StringBuilder(); builder.append("OutterListKey [LeafInList="); builder.append(LeafInList); builder.append("]"); return builder.toString(); } </pre> |

| YANG | JAVA | JAVA Overriden Object Methods |
|------|------|-------------------------------|
| | } | |

Choice and Case

Choice element is mapped similarly as list element. Choice element is mapped to interface (marker interface) and in the superior element is created using getter method with the return type List of this marker interfaces. Case substatements are mapped to the JAVA interfaces which extend mentioned marker interface. Example of choice mapping:

| YANG | JAVA |
|--|--|
| <pre> container cont { choice choice-test { case case1 { } case case2 { } } } </pre> | <p>Case1.java</p> <pre> package org.opendaylight.yang.gen.v1.urn.module.rev201379.cont.choice.test; import org.opendaylight.yangtools.yang.binding.DataObject; import org.opendaylight.yangtools.yang.binding.Augmentable; import org.opendaylight.yang.gen.v1.urn.module.rev201379.cont.ChoiceTest; public interface Case1 extends DataObject, Augmentable<Case1>, ChoiceTest { } </pre> <p>Case2.java</p> <pre> package org.opendaylight.yang.gen.v1.urn.module.rev201379.cont.choice.test; import org.opendaylight.yangtools.yang.binding.DataObject; import org.opendaylight.yangtools.yang.binding.Augmentable; import org.opendaylight.yang.gen.v1.urn.module.rev201379.cont.ChoiceTest; public interface Case2 extends DataObject, Augmentable<Case2>, ChoiceTest { } </pre> <p>ChoiceTest.java</p> <pre> package org.opendaylight.yang.gen.v1.urn.module.rev201379.cont; import org.opendaylight.yangtools.yang.binding.DataObject; public interface ChoiceTest extends DataObject { } </pre> |

Grouping and Uses

Grouping is mapped to JAVA interface. Uses used in some element (using of concrete grouping) are mapped as extension of interface for this element with the interface which represents grouping. Example of grouping and uses mapping.

| YANG | JAVA |
|-------------------------------|---|
| <pre> grouping grp { } </pre> | <p>Cont.java</p> <pre> package org.opendaylight.yang.gen.v1.urn.module.rev201379; </pre> |

| YANG | JAVA |
|---|--|
| <pre> container cont { uses grp; } </pre> | <pre> import org.opendaylight.yangtools.yang.binding. DataObject; import org.opendaylight.yangtools.yang.binding. Augmentable; public interface Cont extends DataObject, Augmentable<Cont>, Grp { } Grp.java package org.opendaylight.yang.gen.v1.urn.module. rev201379; import org.opendaylight.yangtools.yang.binding. DataObject; public interface Grp extends DataObject { } </pre> |

Rpc

Rpc is mapped to JAVA as method of class `ModuleService.java`. Rpc's substatement are mapped as follows:

| Rpc Substatement | Mapping to JAVA |
|------------------|-----------------|
| input | interface |
| output | interface |

Example of rpc mapping:

| YANG | JAVA |
|--|--|
| <pre> rpc rpc-test1 { output { leaf lf-output { type string; } } input { leaf lf-input { type string; } } } </pre> | <pre> ModuleService.java package org.opendaylight.yang.gen.v1.urn.module. rev201379; import java.util.concurrent.Future; import org.opendaylight.yangtools.yang.common. RpcResult; public interface ModuleService { Future<RpcResult<RpcTest1Output>> rpcTest1(RpcTest1Input input); } RpcTest1Input.java package org.opendaylight.yang.gen.v1.urn.module. rev201379; public interface RpcTest1Input { String getLfInput(); } RpcTest1Output.java package org.opendaylight.yang.gen.v1.urn.module. rev201379; public interface RpcTest1Output { String getLfOutput(); } </pre> |

Notification

`Notification` is mapped to the JAVA interface which extends `Notification` interface. Example of notification mapping:

| YANG | JAVA |
|---------------------------------------|---|
| <pre>notification notif { }</pre> | <pre>package org.opendaylight.yang.gen.v1.urn.module. rev201379; import org.opendaylight.yangtools.yang.binding. DataObject; import org.opendaylight.yangtools.yang.binding. Augmentable; import org.opendaylight.yangtools.yang.binding. Notification; public interface Notif extends DataObject, Augmentable<Notif>, Notification { }</pre> |

Augment

`Augment` is mapped to the JAVA interface. The interface starts with the same name as the name of augmented interface. The suffix is order number of augmenting interface. The augmenting interface also extends `Augmentation<>` with actual type parameter equal to augmented interface. Example of augment mapping. In this example is augmented interface `Cont` so whole parametrized type is `Augmentation<Cont>`.

| YANG | JAVA |
|--|---|
| <pre>container cont { } augment "/cont" { }</pre> | <pre>Cont.java package org.opendaylight.yang.gen.v1.urn.module. rev201379; import org.opendaylight.yangtools.yang.binding. DataObject; import org.opendaylight.yangtools.yang.binding. Augmentable; public interface Cont extends DataObject, Augmentable<Cont> { } Cont1.java package org.opendaylight.yang.gen.v1.urn.module. rev201379; import org.opendaylight.yangtools.yang.binding. DataObject; import org.opendaylight.yangtools.yang.binding. Augmentation; public interface Cont1 extends DataObject, Augmentation<Cont> { }</pre> |

Identity

The purpose of the identity statement is to define a new globally unique, abstract, and untyped identity. YANG substatement `base` considers an argument a string; the name of existing identity from which the new identity is derived. Hence, the identity statement

is mapped to JAVA abstract class and base substatement is mapped as extends JAVA keyword. The identity name is translated to class name.

| YANG | JAVA |
|--|---|
| <pre>identity toast-type { }</pre> | <pre>public abstract class ToastType extends BaseIdentity { protected ToastType() { super(); } }</pre> |
| <pre>identity white-bread { base toast-type; }</pre> | <pre>WhiteBread.java public abstract class WhiteBread extends ToastType { protected WhiteBread() { super(); } }</pre> |