networking-odl Documentation

Release 19.0.1.dev4

OpenStack Foundation

Jun 02, 2023

CONTENTS

1 Summary **1**

2 Installation **3**
2.1 Installation Guide . . . 3
2.1.1 Installation . . . 3
ODL Installation . . . 3
Networking-odl Installation . . . 4
Networking-odl Configuration . . . 4
Compute/network nodes . . . 6
2.1.2 Enabling in Devstack . . . 6

3 Configuration options **11**
3.1 Configuration Reference . . . 11
3.1.1 ml2_odl . . . 11
3.1.2 Configuration Samples . . . 13
Sample ml2_conf_odl.ini . . . 14

4 Administration Guide **17**
4.1 Administration Guide . . . 17
4.1.1 Reference Architecture . . . 17
Cloud Composition . . . 17
Networking Requirements . . . 17
Minimal Hardware Requirements . . . 18

5 Contributor Guide **21**
5.1 Contributor Guide . . . 21
5.1.1 Contributors Reference . . . 21
Testing Networking-odl + neutron . . . 21
ODL Drivers Architecture . . . 24
Journal Maintenance . . . 27
Usage . . . 28
Contributing . . . 28
Specifications . . . 28
5.1.2 Tutorial . . . 42
Developer Quick-Start . . . 42
5.1.3 Networking OpenDayLight Internals . . . 46
Host Configuration . . . 46

6 Reference Deployment Guide **51**

6.1 Reference Deployment 51
6.1.1 OpenStack Version Reference 51
Pike ODL Reference 51
Ocata ODL Reference 51
Newton ODL Reference 52

CHAPTER

ONE

SUMMARY

OpenStack networking-odl is a library of drivers and plugins that integrates OpenStack Neutron API with OpenDaylight Backend. For example it has ML2 driver and L3 plugin to enable communication of OpenStack Neutron L2 and L3 resources API to OpenDayLight Backend.

To report and discover bugs in networking-odl the following link can be used: https://bugs.launchpad.net/networking-odl

Any new code submission or proposal must follow the development guidelines detailed in HACKING.rst and for further details this link can be checked: https://docs.openstack.org/networking-odl/latest/

The OpenDaylight homepage: https://www.opendaylight.org/

Release notes for the project can be found at: https://docs.openstack.org/releasenotes/networking-odl/

The project source code repository is located at: https://opendev.org/openstack/networking-odl

CHAPTER

TWO

INSTALLATION

2.1 Installation Guide

2.1.1 Installation

The `networking-odl` repository includes integration with DevStack that enables creation of a simple OpenDaylight (ODL) development and test environment. This document discusses what is required for manual installation and integration into a production OpenStack deployment tool of conventional architectures that include the following types of nodes:

- Controller - Runs OpenStack control plane services such as REST APIs and databases.
- Network - Provides connectivity between provider (public) and project (private) networks. Services provided include layer-3 (routing), DHCP, and metadata agents. Layer-3 agent is optional. When using netvirt (vpnservice) DHCP/metadata are optional.
- Compute - Runs the hypervisor and layer-2 agent for the Networking service.

ODL Installation

http://docs.opendaylight.org provides manual and general documentation for ODL

Review the following documentation regardless of install scenario:

- ODL installation.
- OpenDaylight with OpenStack.

Choose and review one of the following installation scenarios:

- GBP with OpenStack. OpenDaylight Group Based Policy allows users to express network configuration in a declarative rather than imperative way. Often described as asking for what you want, rather than how you can do it, Group Based Policy achieves this by implementing an Intent System. The Intent System is a process around an intent driven data model and contains no domain specifics but is capable of addressing multiple semantic definitions of intent.
- OVSDB with OpenStack. OpenDaylight OVSDB allows users to take advantage of Network Virtualization using OpenDaylight SDN capabilities whilst utilizing OpenvSwitch. The stack includes a Neutron Northbound, a Network Virtualization layer, an OVSDB southbound plugin, and an OpenFlow southbound plugin.
- VTN with OpenStack. OpenDaylight Virtual Tenant Network (VTN) is an application that provides multi-tenant virtual network on an SDN controller. VTN Manager is implemented as one

plugin to the OpenDaylight controller and provides a REST interface to create/update/delete VTN components. It provides an implementation of Openstack L2 Network Functions API.

Networking-odl Installation

```
# sudo pip install networking-odl
```

Note: pip need to be installed before running above command.

Networking-odl Configuration

All related neutron services need to be restarted after configuration change.

1. Configure Openstack neutron server. The neutron server implements ODL as an ML2 driver. Edit the `/etc/neutron/neutron.conf` file:

   - Enable the ML2 core plug-in.

   ```
   [DEFAULT]
   ...
   core_plugin = neutron.plugins.ml2.plugin.Ml2Plugin
   ```

   - (Optional) Enable ODL L3 router, if QoS feature is desired, then qos should be appended to service_plugins

   ```
   [DEFAULT]
   ...
   service_plugins = odl-router_v2
   ```

2. Configure the ML2 plug-in. Edit the `/etc/neutron/plugins/ml2/ml2_conf.ini` file:

   - Configure the ODL mechanism driver, network type drivers, self-service (tenant) network types, and enable extension drivers(optional).

   ```
   [ml2]
   ...
   mechanism_drivers = opendaylight_v2
   type_drivers = local,flat,vlan,vxlan
   tenant_network_types = vxlan
   extension_drivers = port_security, qos
   ```

   Note: The enabling of extension_driver qos is optional, it should be enabled if service_plugins for qos is also enabled.

   - Configure the vxlan range.

```
[ml2_type_vxlan]
...
vni_ranges = 1:1000
```

- Optionally, enable support for VLAN provider and self-service networks on one or more physical networks. If you specify only the physical network, only administrative (privileged) users can manage VLAN networks. Additionally specifying a VLAN ID range for a physical network enables regular (non-privileged) users to manage VLAN networks. The Networking service allocates the VLAN ID for each self-service network using the VLAN ID range for the physical network.

  ```
  [ml2_type_vlan]
  ...
  network_vlan_ranges = PHYSICAL_NETWORK:MIN_VLAN_ID:MAX_VLAN_ID
  ```

  Replace `PHYSICAL_NETWORK` with the physical network name and optionally define the minimum and maximum VLAN IDs. Use a comma to separate each physical network.

  For example, to enable support for administrative VLAN networks on the `physnet1` network and self-service VLAN networks on the `physnet2` network using VLAN IDs 1001 to 2000:

  ```
  network_vlan_ranges = physnet1,physnet2:1001:2000
  ```

- Enable security groups.

  ```
  [securitygroup]
  ...
  enable_security_group = true
  ```

- Configure ML2 ODL

  ```
  [ml2_odl]

  ...
  username = <ODL_USERNAME>
  password = <ODL_PASSWORD>
  url = http://<ODL_IP_ADDRESS>:<ODL_PORT>/controller/nb/v2/neutron
  port_binding_controller = pseudo-agentdb-binding
  ```

- Optionally, To enable ODL DHCP service in an OpenDaylight enabled cloud, set *enable_dhcp_service=True* under the *[ml2_odl]* section. It will load the openstack-odl-v2-dhcp-driver which will create special DHCP ports in neutron for use by the OpenDaylight Controllers DHCP Service. Please make sure to set *controller-dhcp-enabled = True* within the OpenDaylight Controller configuration file `netvirt-dhcpservice-config.xml` along with the above configuration.

  OpenDaylight Spec Documentation Link:.

  ```
  [ml2_odl]

  ...
  enable_dhcp_service = True
  ```

Compute/network nodes

Each compute/network node runs the OVS services. If compute/network nodes are already configured to run with Neutron ML2 OVS driver, more steps are necessary. OVSDB with OpenStack can be referred to.

1. Install the `openvswitch` packages.
2. Start the OVS service.

   Using the *systemd* unit:

   ```
   # systemctl start openvswitch
   ```

   Using the `ovs-ctl` script:

   ```
   # /usr/share/openvswitch/scripts/ovs-ctl start
   ```

3. Configure OVS to use ODL as a manager.

   ```
   # ovs-vsctl set-manager tcp:${ODL_IP_ADDRESS}:6640
   ```

   Replace `ODL_IP_ADDRESS` with the IP address of ODL controller node

4. Set host OVS configurations if port_binding_controller is pseudo-agent

   ```
   # sudo neutron-odl-ovs-hostconfig
   ```

5. Verify the OVS service.

   ```
   # ovs-vsctl show
   ```

Note: After setting config files, you have to restart the neutron server if you are using screen then it can be directly started from neutron-api window or you can use service neutron-server restart, latter may or may not work depending on OS you are using.

2.1.2 Enabling in Devstack

1. Download DevStack
2. Copy the sample local.conf over:

   ```
   cp devstack/local.conf.example local.conf
   ```

3. Optionally, to manually configure this:

   Add this repo as an external repository:

   ```
   > cat local.conf
   [[local|localrc]]
   enable_plugin networking-odl http://opendev.org/openstack/networking-odl
   ```

4. Optionally, to enable support for OpenDaylight L3 router functionality, add the below:

```
> cat local.conf
[[local|localrc]]
ODL_L3=True
```

Note: This is only relevant when using old netvirt (ovsdb based, default).

5. If you need to route the traffic out of the box (e.g. br-ex), set ODL_PROVIDER_MAPPINGS to map the physical provider network to device mapping, as shown below:

```
> cat local.conf
[[local|localrc]]
ODL_L3=True
ODL_PROVIDER_MAPPINGS=${ODL_PROVIDER_MAPPINGS:-br-ex:eth2}    # for old␣
↪netvirt (ovsdb based)
ODL_PROVIDER_MAPPINGS=${ODL_PROVIDER_MAPPINGS:-physnet1:eth2} # for new␣
↪netvirt (vpnservice based)
```

7. run `stack.sh`
8. Note: In a multi-node devstack environment, for each compute node you will want to add this to the local.conf file:

```
> cat local.conf
[[local|localrc]]
enable_plugin networking-odl http://opendev.org/openstack/networking-odl
ODL_MODE=compute
```

9. Note: In a node using a release of Open vSwitch provided from another source than your Linux distribution you have to enable in your local.conf skipping of OVS installation step by setting *SKIP_OVS_INSTALL=True*. For example when stacking together with networking-ovs-dpdk Neutron plug-in to avoid conflicts between openvswitch and ovs-dpdk you have to add this to the local.conf file:

```
> cat local.conf
[[local|localrc]]
enable_plugin networking-ovs-dpdk http://opendev.org/openstack/networking-
↪ovs-dpdk
enable_plugin networking-odl http://opendev.org/openstack/networking-odl
SKIP_OVS_INSTALL=True
```

10. Note: Optionally, to use the new netvirt implementation (netvirt-vpnservice-openstack), add the following to the local.conf file (only allinone topology is currently supported by devstack, since tunnel endpoints are not automatically configured). For tunnel configurations after loading devstack, please refer to this guide https://wiki.opendaylight.org/view/Netvirt:_L2Gateway_HowTo#Configuring_Tunnels:

```
> cat local.conf
[[local|localrc]]
ODL_NETVIRT_KARAF_FEATURE=odl-restconf-all,odl-aaa-authn,odl-dlux-core,
↪odl-mdsal-apidocs,odl-netvirt-vpnservice-openstack
```

(continues on next page)

(continued from previous page)

```
ODL_BOOT_WAIT_URL=restconf/operational/network-topology:network-topology/
↪# Workaround since netvirt:1 no longer exists in DS!
```

11. Note: To enable Quality Of Service (QoS) with OpenDaylight Backend, add the following lines in neutron.conf:

```
> in /etc/neutron/neutron.conf
service_plugins = qos, odl-router
```

enable qos extension driver in ml2 conf:

```
> in /etc/neutron/plugins/ml2/ml2_conf.ini
extensions_drivers = qos, port_security
```

restart neutron service neutron-api

12. Note: legacy netvirt specific options

   - OVS conntrack support

     variable ODL_LEGACY_NETVIRT_CONNTRACK By default its False for compatibility and version requirements.

     – version requirement

       ODL version Boron release or later. (ODL legacy netvirt support is from Beryllium. But networking-odl devstack supports Boron+)

       OVS version 2.5 or later

     enable OVS conntrack support:

```
> cat local.conf
[[local|localrc]]
ODL_LEGACY_NETVIRT_CONNTRACK=True
```

13. Note: To enable Vlan Aware VMs (Trunk) with OpenDaylight Backend, make the following entries in local.conf:

```
> cat local.conf
[[local|localrc]]
Q_SERVICE_PLUGIN_CLASSES=trunk
```

14. Enabling L2Gateway Backend for OpenDaylight

- The package networking-l2gw must be installed as a pre-requisite.

  So include in your localrc (or local.conf) the following:

```
enable_plugin networking-l2gw http://opendev.org/x/networking-l2gw
enable_service l2gw-plugin
NETWORKING_L2GW_SERVICE_DRIVER=L2GW:OpenDaylight:networking_odl.l2gateway.
↪driver_v2.OpenDaylightL2gwDriver:default
```

- Now stack up Devstack and after stacking completes, we are all set to use l2gateway-as-a-service with OpenDaylight.

15. Note: To enable Service Function Chaining support driven by networking-sfc, the following steps have to be taken:

    - local.conf should contain the following lines:

    ```
    # enable our plugin:
    enable_plugin networking-odl https://github.com/openstack/networking-
    ↪odl.git

    # enable the networking-sfc plugin:
    enable_plugin networking-sfc https://github.com/openstack/networking-
    ↪sfc.git

    # enable the odl-netvirt-sfc Karaf feature in OpenDaylight
    ODL_NETVIRT_KARAF_FEATURE+=,odl-netvirt-sfc

    # enable the networking-sfc OpenDaylight driver pair
    [[post-config|$NEUTRON_CONF]]
    [sfc]
    drivers = odl_v2
    [flowclassifier]
    drivers = odl_v2
    ```

    - A special commit of Open vSwitch should be compiled and installed (containing compatible NSH OpenFlow support). This isnt done automatically by networking-odl or DevStack, so the user has to manually install. Please follow the instructions in: https://wiki.opendaylight.org/view/Service_Function_Chaining:Main#Building_Open_vSwitch_with_VxLAN-GPE_and_NSH_support

    - Fluorine is the recommended and latest version of OpenDaylight to use, you can specify it by adding the following to local.conf:

    ```
    ODL_RELEASE=fluorine-snapshot-0.9.0
    ```

    - To clarify, OpenDaylight doesnt have to be running/installed before stacking with networking-odl (and it shouldnt). The networking-odl DevStack plugin will download and start OpenDaylight automatically. However, it will not fetch the correct Open vSwitch version, so the instructions above and the usage of `SKIP_OVS_INSTALL` are important.

16. To enable BGPVPN driver to use with OpenDaylight controller Include the following lines in your localrc (or local.conf):

    ```
    enable_plugin networking-bgpvpn https://opendev.org/openstack/networking-
    ↪bgpvpn.git

    [[post-config|$NETWORKING_BGPVPN_CONF]]
    [service_providers]
    service_provider=BGPVPN:OpenDaylight:networking_odl.bgpvpn.odl_v2.
    ↪OpenDaylightBgpvpnDriver:default
    ```

    and then stack up your devstack.

17. To enable DHCP Service in OpenDaylight deployments with Openstack, please use:

```
[[local|localrc]]
ODL_DHCP_SERVICE=True
```

18. To enable ODL with OVS hardware Offload support please use:

```
[[local|localrc]]
ODL_OVS_HOSTCONFIGS_OPTIONS="--noovs_dpdk --debug --ovs_sriov_offload"
```

Note: OVS offload support minimal version requirements - Linux kernel from version 4.12 OVS from version 2.8.0 ODL from version Nitrogen (please note that Nitrogen is no longer maintained)

19. For development environment, if opendaylight installation is not required for stack.sh then a parameter ODL_INSTALL should be set to False. By default it is set to True therefore it is backward compatible with gate and already existing scripts:

```
[[local|localrc]]
ODL_INSTALL=False
```

20. To Enable L3 Flavors with ODL, service providers should be added to neutron.conf:

```
[service_providers]
service_provider = L3_ROUTER_NAT:ODL:networking_odl.l3.l3_flavor.
↪ODLL3ServiceProvider:default
```

Note: Service Plugin router should be used in neutron.conf for enabling L3 flavors completely

CHAPTER

THREE

CONFIGURATION OPTIONS

3.1 Configuration Reference

This section provides configuration options for networking-odl, that needs to be set in addition to neutron configuration, for all other configuration examples like neutron.conf and ml2_conf.ini, neutron repo can be referred.

3.1.1 ml2_odl

url

Type string

Default <None>

HTTP URL of OpenDaylight REST interface.

username

Type string

Default <None>

HTTP username for authentication.

password

Type string

Default <None>

HTTP password for authentication.

timeout

Type integer

Default 10

HTTP timeout in seconds.

session_timeout

Type integer

Default 30

Tomcat session timeout in minutes.

`sync_timeout`

Type floating point

Default `10`

Sync thread timeout in seconds or fraction.

`retry_count`

Type integer

Default `5`

Number of times to retry a row before failing.

`maintenance_interval`

Type integer

Default `300`

Journal maintenance operations interval in seconds.

`completed_rows_retention`

Type integer

Default `0`

Time to keep completed rows (in seconds).For performance reasons its not recommended to change this from the default value (0) which indicates completed rows arent kept.This value will be checked every maintenance_interval by the cleanup thread. To keep completed rows indefinitely, set the value to -1

`enable_lightweight_testing`

Type boolean

Default `False`

Test without real ODL.

`port_binding_controller`

Type string

Default `pseudo-agentdb-binding`

Name of the controller to be used for port binding.

`processing_timeout`

Type integer

Default `100`

Time in seconds to wait before a processing row is marked back to pending.

`odl_hostconf_uri`

Type string

Default `/restconf/operational/neutron:neutron/hostconfigs`

Path for ODL host configuration REST interface

restconf_poll_interval

Type integer

Default 30

Poll interval in seconds for getting ODL hostconfig

enable_websocket_pseudo_agentdb

Type boolean

Default False

Enable websocket for pseudo-agent-port-binding.

odl_features_retry_interval

Type integer

Default 5

Wait this many seconds before retrying the odl features fetch

odl_features

Type list

Default <None>

A list of features supported by ODL.

odl_features_json

Type string

Default <None>

Features supported by ODL, in the json format returnedby ODL. Note: This config option takes precedence overodl_features.

enable_dhcp_service

Type boolean

Default False

Enables the networking-odl driver to supply special neutron ports of dhcp type to OpenDaylight Controller for its use in providing DHCP Service.

3.1.2 Configuration Samples

This section provides sample configuration file ml2_conf_odl.ini

Sample ml2_conf_odl.ini

This is sample for ml2_conf_odl.ini.

```
[DEFAULT]


[ml2_odl]

#
# From ml2_odl
#

# HTTP URL of OpenDaylight REST interface. (string value)
#url = <None>

# HTTP username for authentication. (string value)
#username = <None>

# HTTP password for authentication. (string value)
#password = <None>

# HTTP timeout in seconds. (integer value)
#timeout = 10

# Tomcat session timeout in minutes. (integer value)
#session_timeout = 30

# Sync thread timeout in seconds. (integer value)
#sync_timeout = 10

# Number of times to retry a row before failing. (integer value)
#retry_count = 5

# Journal maintenance operations interval in seconds. (integer value)
#maintenance_interval = 300

# Time to keep completed rows (in seconds).
# For performance reasons it's not recommended to change this from the default
# value (0) which indicates completed rows aren't kept.
# This value will be checked every maintenance_interval by the cleanup
# thread. To keep completed rows indefinitely, set the value to -1
# (integer value)
#completed_rows_retention = 0

# Test without real ODL. (boolean value)
#enable_lightweight_testing = false

# Name of the controller to be used for port binding. (string value)
#port_binding_controller = pseudo-agentdb-binding
```

(continues on next page)

(continued from previous page)

```
# Time in seconds to wait before a processing row is
# marked back to pending. (integer value)
#processing_timeout = 100

# Path for ODL host configuration REST interface (string value)
#odl_hostconf_uri = /restconf/operational/neutron:neutron/hostconfigs

# Poll interval in seconds for getting ODL hostconfig (integer value)
#restconf_poll_interval = 30

# Enable websocket for pseudo-agent-port-binding. (boolean value)
#enable_websocket_pseudo_agentdb = false

# Wait this many seconds before retrying the odl features fetch
# (integer value)
#odl_features_retry_interval = 5

# A list of features supported by ODL (list value)
#odl_features = <None>

# Enables the networking-odl driver to supply special neutron ports of
# "dhcp" type to OpenDaylight Controller for its use in providing DHCP
# Service. (boolean value)
#enable_dhcp_service = false
```

CHAPTER

FOUR

ADMINISTRATION GUIDE

4.1 Administration Guide

4.1.1 Reference Architecture

This document lists the minimum reference architecture to get OpenStack installed with OpenDayLight. Wherever possible, additional resources will be stated.

Cloud Composition

The basic cloud will have 3 types of nodes:

- Controller Node - Runs OpenStack services and the ODL controller.
- Network Node - Runs the DHCP agent, the metadata agent, and the L3 agent (for SNAT).
- Compute Node - VMs live here.

Usually each of the first 2 types of nodes will have a cluster of 3 nodes to support HA. Its also possible to run the ODL controller on separate hardware than the OpenStack services, but this isnt mandatory.

The last type of nodes can have as many nodes as scale requirements dictate.

Networking Requirements

There are several types of networks on the cloud, the most important for the reference architecture are:

- Management Network - This is the network used to communicate between the different management components, i.e. Nova controller to Nova agent, Neutron to ODL, ODL to OVS, etc.
- External Network - This network provides VMs with external connectivity (i.e. internet) usually via virtual routers.
- Data Network - This is the network used to connect the VMs to each other and to network resources such as virtual routers.

The Control Nodes usually are only connected to the Management Network, unless they have an externally reachable IP on the External Network.

The other node types are connected to all the networks since ODL uses a distributed routing model so that each Compute Node hosts a virtual router responsible for connecting the VMs from that node to other networks (including the External Network).

This diagram illustrates how these nodes might be connected:

```
            Controller Node
           +-----------------+
           |                 |
+----------+192.168.0.251    |
|          |                 |
|          +-----------------+
|
|           Compute Node         +----------------+
|          +---------------+     | Legend         |
|          |               |     +----------------+
+----------+192.168.0.1    |     |                |
|          |               |     | --- Management |
| +~~~~~~~~+10.0.0.1       |     |                |
| |        |               |     | ~~~ Data       |
| | +======+br-int         |     |                |
| | |      |               |     | === External   |
| | |      +---------------+     |                |
| | |                            +----------------+
| | |       Network Node
| | |      +-----------------+
| | |      |                 |
+----------+192.168.0.100    |
  | |      |                 |
  +~~~~~~~~+10.0.0.100       |
    |      |                 |
    |======+br-int           |
    |      |                 |
    |      +-----------------+
+----+---+
|        |
| Router |
|        |
+--------+
```

Minimal Hardware Requirements

The rule of thumb is the bigger the better, more RAM and more cores will translate to a better environment. For a POC environment the following is necessary:

Management Node

CPU: 2 cores

Memory: 8 GB

Storage: 100 GB

Network: 1 * 1 Gbps NIC

Network Node

CPU: 2 cores

Memory: 2 GB

Storage: 50 GB

Network: 1 Gbps NIC (Management Network), 2 * 1+ Gbps NICs

Compute Node

CPU: 2+ cores

Memory: 8+ GB

Storage: 100 GB

Network: 1 Gbps NIC (Management Network), 2 * 1+ Gbps NICs

CHAPTER FIVE

CONTRIBUTOR GUIDE

5.1 Contributor Guide

In the Developer/Contributor Guide, you will find information on networking-odls lower level design and implementation details. We will cover only essential details related to just networking-odl and we wont repeat neutron devref here, for details in neutron, neutrons devref can be checked: https://docs.openstack.org/neutron/latest/contributor/index.html

For details regarding OpenStack Neutrons Api: https://docs.openstack.org/api-ref/network/

5.1.1 Contributors Reference

Testing Networking-odl + neutron

Overview

The unit tests (networking_odl/tests/unit/) are meant to cover as much code as possible and should be executed without the service running. They are designed to test the various pieces of the neutron tree to make sure any new changes dont break existing functionality.

TODO (Manjeet): Update functional testing doc.

Development process

It is expected that any new changes that are proposed for merge come with tests for that feature or code area. Ideally any bugs fixes that are submitted also have tests to prove that they stay fixed! In addition, before proposing for merge, all of the current tests should be passing.

Virtual environments

Testing OpenStack projects, including Neutron, is made easier with DevStack.

Create a machine (such as a VM or Vagrant box) running a distribution supported by DevStack and install DevStack there. For example, there is a Vagrant script for DevStack at https://github.com/bcwaldon/vagrant_devstack.

Note: If you prefer not to use DevStack, you can still check out source code on your local machine and develop from there.

Running unit tests

There are two mechanisms for running tests: tox, and nose. Before submitting a patch for review you should always ensure all test pass; a tox run is triggered by the jenkins gate executed on gerrit for each patch pushed for review.

With these mechanisms you can either run the tests in the standard environment or create a virtual environment to run them in.

By default after running all of the tests, any pep8 errors found in the tree will be reported.

With *nose*

You can use nose to run individual tests, as well as use for debugging portions of your code:

```
. .venv/bin/activate
pip install nose
nosetests
```

There are disadvantages to running Nose - the tests are run sequentially, so race condition bugs will not be triggered, and the full test suite will take significantly longer than tox & testr. The upside is that testr has some rough edges when it comes to diagnosing errors and failures, and there is no easy way to set a breakpoint in the Neutron code, and enter an interactive debugging session while using testr.

With *tox*

Networking-odl, like other OpenStack projects, uses tox for managing the virtual environments for running test cases. It uses Testr for managing the running of the test cases.

Tox handles the creation of a series of virtualenvs that target specific versions of Python (2.6, 2.7, 3.3, etc).

Testr handles the parallel execution of series of test cases as well as the tracking of long-running tests and other things.

Running unit tests is as easy as executing this in the root directory of the Neutron source code:

```
tox
```

Running tests for syntax and style check for written code:

```
tox -e pep8
```

For more information on the standard Tox-based test infrastructure used by OpenStack and how to do some common test/debugging procedures with Testr, see this wiki page: https://wiki.openstack.org/wiki/Testr

Tests written can also be debugged by adding pdb break points. Normally if you add a break point and just run the tests with normal flags they will end up in failing. There is debug flag you can use to run after adding pdb break points in the tests.

Set break points in your test code and run:

```
tox -e debug networking_odl.tests.unit.db.test_db.DbTestCase.test_validate_
↪updates_same_object_uuid
```

The package oslotest was used to enable debugging in the tests. For more information see the link: https://docs.openstack.org/oslotest/latest/user/features.html

Running individual tests

For running individual test modules or cases, you just need to pass the dot-separated path to the module you want as an argument to it.

For executing a specific test case, specify the name of the test case class separating it from the module path with a colon.

For example, the following would run only the TestUtils tests from networking_odl/tests/unit/common/test_utils.py

```
$ tox -e py37 networking_odl.tests.unit.common.test_utils.TestUtils
```

Adding more tests

There might not be full coverage yet. New patches for adding tests which are not there are always welcome.

To get a grasp of the areas where tests are needed, you can check current coverage by running:

```
$ tox -e cover
```

Debugging

Its possible to debug tests in a tox environment:

```
$ tox -e venv -- python -m testtools.run [test module path]
```

Tox-created virtual environments (venvs) can also be activated after a tox run and reused for debugging:

```
$ tox -e venv
$ . .tox/venv/bin/activate
$ python -m testtools.run [test module path]
```

Tox packages and installs the neutron source tree in a given venv on every invocation, but if modifications need to be made between invocation (e.g. adding more pdb statements), it is recommended that the source tree be installed in the venv in editable mode:

```
# run this only after activating the venv
$ pip install --editable .
```

Editable mode ensures that changes made to the source tree are automatically reflected in the venv, and that such changes are not overwritten during the next tox run.

Running functional tests

Neutron defines different classes of test cases. One of them is functional test. It requires pre-configured environment. But its lighter than running devstack or openstack deployment. For definitions of functional tests, please refer to: https://docs.openstack.org/neutron/latest/contributor/index.html

The script is provided to setup the environment. At first make sure the latest version of pip command:

```
# ensure you have the latest version of pip command
# for example on ubuntu
$ sudo apt-get install python-pip
$ sudo pip --upgrade pip
```

And then run functional test as follows:

```
# assuming devstack is setup with networking-odl
$ cd networking-odl
$ ./tools/configure_for_func_testing.sh /path/to/devstack
$ tox -e dsvm-functional
```

For setting up devstack, please refer to neutron documentation:

- https://wiki.openstack.org/wiki/NeutronDevstack
- https://docs.openstack.org/neutron/latest/contributor/index.html
- https://docs.openstack.org/neutron/latest/contributor/testing/testing.html

ODL Drivers Architecture

This document covers architectural concepts of the ODL drivers. Although driver is an ML2 term, its used widely in ODL to refer to any implementation of APIs. Any mention of ML2 in this document is solely for reference purposes.

V1 Driver Overview (Removed in Rocky)

Note: This architecture has been deprecated in Queens and removed in Rocky. The documentation is kept as a reference to understand the necessity of a different architecture.

The first driver version was a naive implementation which synchronously mirrored all calls to the ODL controller. For example, a create network request would first get written to the DB by Neutrons ML2 plugin, and then the ODL driver would send the request to POST the network to the ODL controller.

Although this implementation is simple, it has a few problems:

- ODL is not really synchronous, so if the REST call succeeds it doesnt mean the action really happened on ODL.
- The synchronous call can be a bottleneck under load.
- Upon failure the V1 driver would try to full sync the entire Neutron DB over on the next call, so the next call could take a very long time.
- It doesnt really handle race conditions:
  - For example, create subnet and then create port could be sent in parallel by the driver in an HA Neutron environment, causing the port creation to fail.
  - Full-sync could possibly recreate deleted resources if the deletion happens in parallel.

V2 Driver Design

The V2 driver set upon to tackle problems encountered in the V1 driver while maintaining feature parity. The major design concept of the V2 driver is *journaling* - instead of passing the calls directly to the ODL controller, they get registered in the journal table which keeps a sort of queue of the various operations that occurred on Neutron and should be mirrored to the controller.

The journal is processed mainly by a journaling thread which runs periodically and checks if the journal table has any entries in need of processing. Additionally the thread is triggered in the postcommit hook of the operation (where applicable).

If we take the example of create network again, after it gets stored in the Neutron DB by the ML2 plugin, the ODL driver stores a journal entry representing that operation and triggers the journaling thread to take care of the entry.

The journal entry is recorded in the pre-commit phase (whenever applicable) so that in case of a commit failure the journal entry gets aborted along with the original operation, and theres nothing extra needed.

The *get_resources_for_full_sync* method is defined in the ResourceBaseDriver class, it fetches all the resources needed for full sync, based on resource type. To override the default behaviour of *get_resources_for_full_sync* define it in driver class, For example L2 gateway driver needs to provide customized method for filtering of fetched gateway connection information from database. Neutron defines *l2_gateway_id* for a l2 gateway connection but ODL expects *gateway_id*, these kind of pre or post processing can be done in this method.

Journal Entry Lifecycle

The first state in which a journal entry is created is the pending state. In this state, the entry is awaiting a thread to pick it up and process it. Multiple threads can try to grab the same journal entry, but only one will succeed since the selection is done inside a select for update clause. Special care is taken for GaleraDB since it reports a deadlock if more than one thread selects the same row simultaneously.

Once an entry has been selected it will be put into the processing state which acts as a lock. This is done in the same transaction so that in case multiple threads try to lock the same entry only one of them will succeed. When the winning thread succeeds it will continue with processing the entry.

The first thing the thread does is check for dependencies - if the entry depends on another one to complete. If a dependency is found, the entry is put back into the queue and the thread moves on to the next entry.

When there are no dependencies for the entry, the thread analyzes the operation that occurred and performs the appropriate call to the ODL controller. The call is made to the correct resource or collection and the type of call (PUT, POST, DELETE) is determined by the operation type. At this point if the call was successful (i.e. got a 200 class HTTP code) the entry is marked completed.

In case of a failure the thread determines if this is an expected failure (e.g. network connectivity issue) or an unexpected failure. For unexpected failures a counter is raised, so that a given entry wont be retried more than a given amount of times. Expected failures dont change the counter. If the counter exceeds the configured amount of retries, the entry is marked as failed. Otherwise, the entry is marked back as pending so that it can later be retried.

Full Sync & Recovery

```
file: networking_odl/journal/base_driver.py

ALL_RESOURCES = {}

class ResourceBaseDriver(object):
    # RESOURCES is dictionary of resource_type and resource_suffix to
    # be defined by the drivers class.
    RESOURCES = {}

    def __init__(self, plugin_type, *args, **kwargs):
        super(ResourceBaseDriver, self).__init__(*args, **kwargs)
        self.plugin_type = plugin_type
        # All the common methods to be used by full sync and recovery
        # specific to driver.

        # Only driver is enough for all the information. Driver has
        # plugin_type for fetching the information from db and resource
        # suffix is available through driver.RESOURCES.
        for resource, resource_suffix in self.RESOURCES.items():
            ALL_RESOURCES[resource] = self

    def get_resource_for_recovery(self, resource_type, resource_id):
        # default definition to be used, if get_resource method is not
        # defined then this method gets called by recovery

    def get_resources_for_full_sync(self, resource_type):
        # default definition to be used, if get_resources method is not
        # defined then this method gets called by full sync

    @staticmethod
    def get_method_name_by_resource_suffix(method_suffix):
        # Returns method name given resource suffix

    @staticmethod
    def get_method(plugin, method_name):
        # Returns method for a specific plugin
```

(continues on next page)

(continued from previous page)

```
file: networking_odl/<driver-name>/<driver-file>.py

class XXXXDriver(ResourceBaseDriver, XXXXDriverBase):
    RESOURCES = {
        odl_const.XXXX: odl_const.XXXY,
        odl_const.XXXY: odl_const.XXYY
    }

    def __init__(self, *args, **kwargs):
        super(XXXXDriver, self)(plugin_type, *args, **kwargs)
        # driver specific things

    # get_resources_for_full_sync and get_resource_for_recovery methods are
    # optional and they have to be defined, if customized behaviour is
    # required. If these methods are not defined in the driver then default
    # methods defined in ResourceBaseDriver is used.
    def get_resources_for_full_sync(self, resource_type):
        # returns resource for full sync

    def get_resource_for_recovery(self, resource_type, resource_id):
        # returns resource for recovery
```

Journal Maintenance

Overview

The V2 ODL driver is Journal based[1], which means that theres a journal of entries detailing the various operations done on a Neutron resource. The driver has a thread which is in charge of processing the journal of operations which entails communicating the operation forward to the ODL controller.

The journal entries can wind up in several states due to various reasons:

- PROCESSING - Stale lock left by a thread due to thread dying or other error
- COMPLETED - After the operation is processed successfully
- FAILED - If there was an unexpected error during the operation

These journal entries need to be dealt with appropriately, hence a maintenance thread was introduced that takes care of journal maintenance and other related tasks. This thread runs in a configurable interval and is HA safe using a shared state kept in the DB.

Currently the maintenance thread performs:

- Stale lock release
- Completed entries clean up
- Failed entries are handled by the recovery mechanism
- Full sync detect when ODL is tabula rasa and syncs all the resources to it

[1] See *V2 Driver Design* for details.

Creating New Maintenance Operations

Creating a new maintenance operation is as simple as writing a function that receives the database session object and registering it using a call to:

```
MaintenanceThread.register_operation
```

The best place to do so would be at the _start_maintenance_thread method of the V2 OpenDaylight-MechanismDriver class.

Usage

To use networking-odl in a project:

```
import networking_odl
```

Contributing

If you would like to contribute to the development of OpenStack, you must follow the steps documented at: https://docs.openstack.org/infra/manual/developers.html

Once those steps have been completed, changes to OpenStack should be submitted for review via the Gerrit tool, following the workflow documented at: https://docs.openstack.org/infra/manual/developers.html#development-workflow

Pull requests submitted through GitHub will be ignored.

Bugs should be filed on Launchpad, not GitHub: https://bugs.launchpad.net/networking-odl

Specifications

Pike specs

Dependency Validations on Create

https://blueprints.launchpad.net/networking-odl/+spec/dep-validations-on-create

Right now V2 driver entry dependency validations happen when a journal entry is picked for processing. This spec proposes that this be moved to entry creation time, in order to have a clear understanding of the entry dependencies and conserve journal resources.

Problem Description

Dependency validations are necessary in the V2 driver because each operation gets recorded in a journal entry and sent to ODL asynchronously. Thus, a consecutive operation might be sent to ODL before the first one finishes, while relying on the first operation. For example, when a subnet gets created it references a network, but if the network was created right before the subnet was then the subnet create shouldnt be sent over until the network create was sent.

Currently these checks are performed each time an entry is selected for processing - if the entry passes the dependency checks then it gets processed and if the dependency check fails (i.e. finds a previous unhandled entry that needs to execute before this one) then the entry gets sent back to the queue.

Generally this is not optimal for several reasons:

- No clear indication of relations between the entries.
  - The logic is hidden in the code and theres no good way to know why an entry fails a dependency check.
  - Difficult to debug in case of problems.
  - Difficult to spot phenomenon such as a cyclic dependency.
- Wasted CPU effort.
  - An entry can be checked multiple times for dependencies.
  - Lots of redundant DB queries to determine dependencies each time.

Proposed Change

The proposed solution is to move the dependency calculation to entry creation time.

When a journal entry is created the dependency management system will calculate the dependencies on other entries (Similarly to how it does now) and if there are journal entries the new entry should depend on, their IDs will be inserted into a link table.

Thus, when the journal looks for an entry to pick up it will only look for entries that no other entry depends on by making sure there arent any entries in the dependency table.

When a journal entry is done processing (either successfully or reaches failed state), the dependency links will be removed from the dependency table so that dependent rows can be processed.

The proposed table:

```
+------------------------+
| odl_journal_dependency |
+------------------------+
| parent_id              |
| dependent_id           |
+------------------------+
```

The table columns will be foreign keys to the seqnum column in the journal table. The constraints will be defined as ON DELETE CASCADE so that when a journal entry is removed any possible rows will be removed as well. The primary key will be made from both columns of the table as this is a link table and not an actual entity. If we face DB performance issues (highly unlikely, since this table should normally

have a very small amount of rows if any at all) then an index can be constructed on the dependent_id column.

The dependency management mechanism will locate parent entries for the given entry and will populate the table so that the parent entrys seqnum will be set as the parent_id, and the dependent entry id will be set as dependent_id. When the journal picks up an entry for processing it will condition it on not having any rows with the parent_id in the dependency table. This will ensure that dependent rows get handled after the parent rows have finished processing.

Performance Considerations

Generally the performance shouldnt be impacted as were moving the part of code that does dependency calculations from the entry selection time to entry creation time. This will assure that dependency calculations happen only once per journal entry.

However, some simple benchmarks should be performed before & after the change:

- Average Tempest run time.
- Average CPU consumption on Tempest.
- Full sync run time (Start to finish of all entries).

If performance suffers a severe degradation then we should consider alternative solutions.

Questions

Q: Should entries in failed state block other entries?

A: Currently failed rows are not considered as blocking for dependency validations, but we might want to change this as it makes little sense to process a dependent entry that failed processing.

Q: How will this help debug-ability?

A: It will be easy to query the table contents at any time to figure out which entries depend on which other entries.

Q: How will we be able to spot cyclic dependencies?

A: Currently this isnt planned as part of the spec, but a DB query (or a series of them) can help determine if this problem exists.

Neutron Port Allocation per Subnet for OpenDaylight DHCP Proxy Service

This spec describes the proposal to allocate a Neutron DHCP Port just for use by OpenDaylight Controller on Subnets that are created or updated with enable-dhcp to True.

When in OpenDaylight controller, the controller-dhcp-enabled configuration flag is set to true, these Neutron DHCP Ports will be used by the OpenDaylight Controller to provide DHCP Service instead of using the subnet-gateway-ip as the DHCP Server IP as it stands today.

The networking-odl driver is not aware about the above OpenDaylight controller parameter configuration. When controller-dhcp-enabled configuration flag is set to false the DHCP port will be created and destroyed without causing any harm to either OpenDaylight controller or networking-odl driver.

Problem Statement

The DHCP service within OpenDaylight currently assumes availability of the subnet gateway IP address. The subnet gateway ip is not a mandatory parameter for an OpenStack subnet, and so it might not be available from OpenStack orchestration. This renders the DHCP service in OpenDaylight to not be able to serve DHCP offers to virtual endpoints requesting for IP addresses, thereby resulting in service unavailability. Even if subnet-gateway-ip is available in the subnet, it is not a good design in OpenDaylight to hijack that ip address and use that as the DHCP Server IP Address.

Problem - 1: L2 Deployment with 3PP gateway

There can be deployment scenario in which L2 network is created with no distributed Router/VPN functionality. This deployment can have a separate gateway for the network such as a 3PP LB VM, which acts as a TCP termination point and this LB VM is configured with a default gateway IP. It means all inter-subnet traffic is terminated on this VM which takes the responsibility of forwarding the traffic.

But the current DHCP service in OpenDaylight controller hijacks gateway IP address for serving DHCP discover/request messages. If the LB is up, this can continue to work, DHCP broadcasts will get hijacked by the OpenDaylight, and responses sent as PKT_OUTs with SIP = GW IP.

However, if the LB is down, and the VM ARPs for the same IP as part of a DHCP renew workflow, the ARP resolution can fail, due to which renew request will not be generated. This can cause the DHCP lease to lapse.

Problem - 2: Designated DHCP for SR-IOV VMs via HWVTEP

In this Deployment scenario, L2 network is created with no distributed Router/ VPN functionality, and HWVTEP for SR-IOV VMs. DHCP flood requests from SR-IOV VMs(DHCP discover, request during bootup), are flooded by the HWVTEP on the L2 Broadcast domain, and punted to the controller by designated vswitch. DHCP offers are sent as unicast responses from Controller, which are forwarded by the HWVTEP to the VM. DHCP renews can be unicast requests, which the HWVTEP may forward to an external Gateway VM (3PPLB VM) as unicast packets. Designated vswitch will never receive these pkts, and thus not be able to punt them to the controller, so renews will fail.

Proposed Change

In general as part of implementation of this spec, we are introducing a new configuration parameter create_opendaylight_dhcp_port whose truth value determines whether the dhcp-proxy-service within the openstack-odl framework need to be made functional. This service will be responsible for managing the create/update/delete lifecycle for a new set of Neutron DHCP Ports which will be provisioned specifically for use by the OpenDaylight Controllers existing DHCP Service Module.

Detailed Design

Introduce a driver config parameter(create_opendaylight_dhcp_port) to determine if OpenDaylight based DHCP service is being used. Default setting for the parameter is false.

When create_opendaylight_dhcp_port is set to True, it triggers the networking -odl ml2 driver to hook on to OpenStack subnet resource lifecycle and use that to manage a special DHCP port per subnet for OpenDaylight Controller use. These special DHCP ports will be shipped to OpenDaylight controller, so that DHCP Service within the OpenDaylight controller can make use of these as DHCP Server ports themselves. The port will be used to service DHCP requests for virtual end points belonging to that subnet.

These special DHCP Ports (one per subnet), will carry unique device-id and device-owner values.

- device-owner(network:dhcp)
- device-id(OpenDaylight-<subnet-id>)

OpenDaylight DHCP service will also introduce a new config parameter controller -dhcp-mode to indicate if the above DHCP port should be used for servicing DHCP requests. When the parameter is set to use-odl-dhcp-neutron-port, it is recommended to enable the create_opendaylight_dhcp_port flag for the networking -odl driver.

Alternative 1

The creation of Neutron OpenDaylight DHCP port will be invoked within the OpenDaylight mechanism Driver subnet-postcommit execution.

Any failures during the neutron dhcp port creation or allocation for the subnet should trigger failure of the subnet create operation with an appropriate failure message in logs. On success the subnet and port information will be persisted to Journal DB and will subsequently synced with the OpenDaylight controller.

The plugin should initiate the removal of allocated dhcp neutron port at the time of subnet delete. The port removal will be handled in a subnet-delete- post-commit execution and any failure during this process should rollback the subnet delete operation. The subnet delete operation will be allowed only when all other VMs launched on this subnet are already removed as per existing Neutron behavior.

A subnet update operation configuring the DHCP state as enabled should allocate such a port if not previously allocated for the subnet. Similarly a subnet update operation configuring DHCP state to disabled should remove any previously allocated OpenDaylight DHCP neutron ports.

Since the invocation of create/delete port will be synchronous within subnet post-commit, a failure to create/delete port will result in an exception being thrown which makes the ML2 Plugin to fail the subnet operation and not alter Openstack DB.

Alternative 2

The OpenDaylight Neutron DHCP Port creation/deletion is invoked asynchronously driven by a journal entry callback for any Subnet resource state changes as part of create/update/delete. A generic journal callback mechanism to be implemented. Initial consumer of this callback would be the OpenDaylight DHCP proxy service but this could be used by other services in future.

The Neutron DHCP Port (for OpenDaylight use) creation is triggered when the subnet journal-entry is moved from PENDING to PROCESSING. On a failure of port-creation, the journal will be retained in PENDING state and the subnet itself wont be synced to the OpenDaylight controller. The journal-entry state is marked as COMPLETED only on successful port creation and successful synchronization of that subnet resource to OpenDaylight controller. The same behavior is applicable for subnet update and delete operations too.

The subnet create/update operation that allocates an OpenDaylight DHCP port to always check if a port exists and allocate new port only if none exists for the subnet.

Since the invocation of create/delete port will be within the journal callback and asynchronous to subnet-postcommit, the failure to create/delete port will result in the created (or updated) subnet to remain in PENDING state. Next journal sync of this pending subnet will again retry creation/deletion of port and this cycle will happen until either create/delete port succeeds or the subnet is itself deleted by the orchestrating tenant. This could result in piling up of journal PENDING entries for these subnets when there is an unexpected failure in create/delete DHCP port operation. It is recommended to not keep retrying the port operation and instead failures would be indicated in OpenDaylight as DHCP offers/renews will not be honored by the dhcp service within the OpenDaylight controller, for that subnet.

Recommended Alternative

All of the following cases will need to be addressed by the design.

- Neutron server can crash after submitting information to DB but before invoking post-commit during a subnet create/update/delete operation. The dhcp-proxy-service should handle the DHCP port creation/deletion during such failures when the service is enabled.
- A subnet update operation to disable-dhcp can be immediately followed by a subnet update operation to enable-dhcp, and such a situation should end up in creating the neutron-dhcp-port for consumption by OpenDaylight.
- A subnet update operation to enable-dhcp can be immediately followed by a subnet update operation to disable-dhcp, and such a situation should end up in deleting the neutron-dhcp-port that was created for use by OpenDaylight.
- A subnet update operation to enable-dhcp can be immediately followed by a subnet delete operation,and such a situation should end up deleting the neutron-dhcp-port that was about to be provided for use by OpenDaylight.
- A subnet create operation (with dhcp enabled) can be immediately followed by a subnet update operation to disable-dhcp, and such a situation should end up in deleting the neutron-dhcp-port that was created for use by OpenDaylight.

Design as per Alternative 2 meets the above cases better and is what we propose to take as the approach that we will pursue for this spec.

Dependencies

Feature is dependent on enhancement in OpenDaylight DHCP Service as per the Spec in [1]

Impact

None

Assignee(s)

- Achuth Maniyedath (achuth.m@altencalsoftlabs.com)
- Karthik Prasad(karthik.p@altencalsoftlabs.com)

References

- [1] OpenDaylight spec to cover this feature https://git.opendaylight.org/gerrit/#/c/52298/

Ocata specs

Journal Recovery

https://blueprints.launchpad.net/networking-odl/+spec/journal-recovery

Journal entries in the failed state need to be handled somehow. This spec will try to address the issue and propose a solution.

Problem Description

Currently there is no handling for Journal entries that reach the failed state. A journal entry can reach the failed state for several reasons, some of which are:

- Reached maximum failed attempts for retrying the operation.
- Inconsistency between ODL and the Neutron DB.
  - For example: An update fails because the resource doesnt exist in ODL.
- Bugs that can lead to failure to sync up.

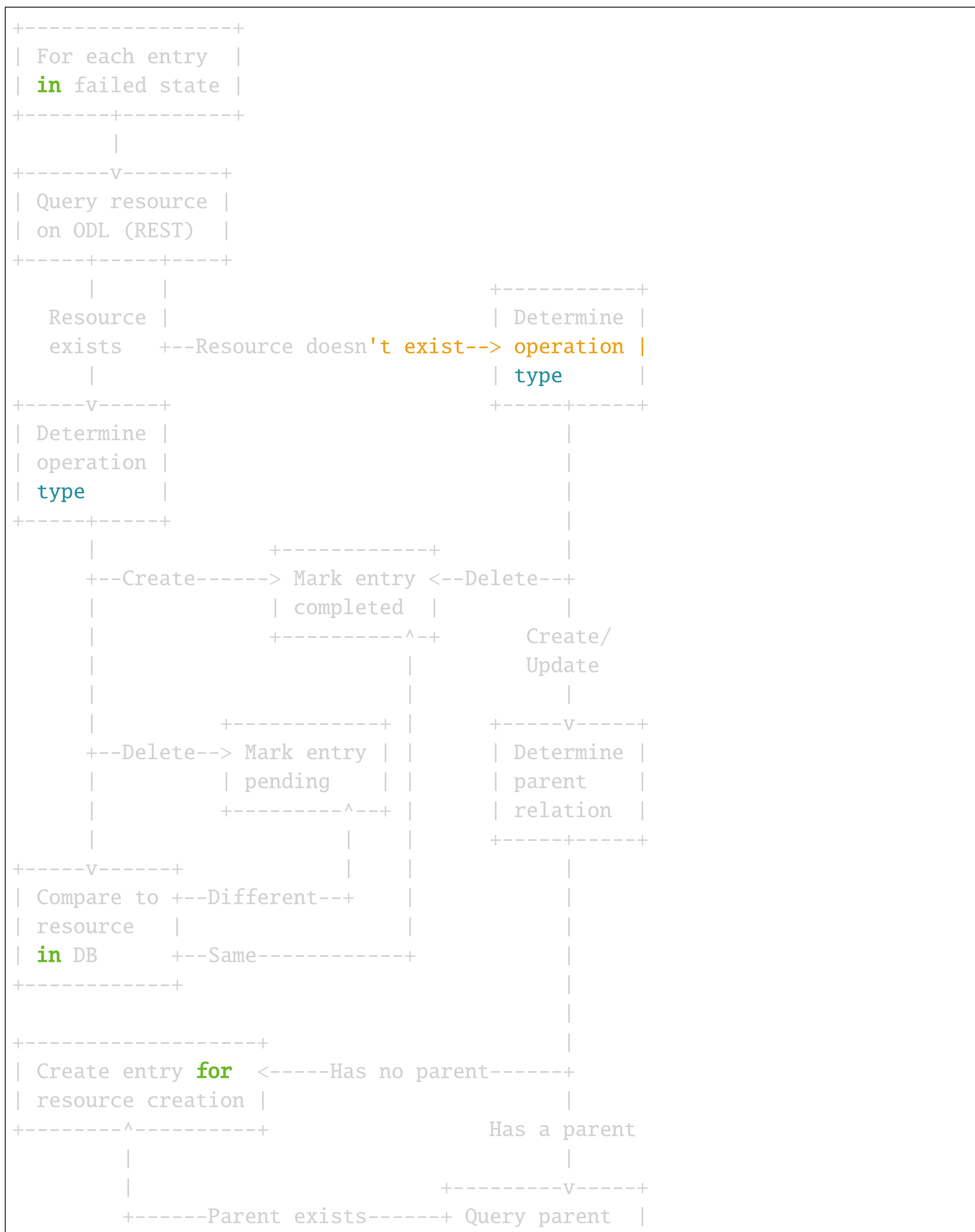
These entries will be left in the journal table forever which is a bit wasteful since they take up some space on the DB storage and also affect the performance of the journal table. Albeit each entry has a negligible effect on its own, the impact of a large number of such entries can become quite significant.

Proposed Change

A journal recovery routine will run as part of the current journal maintenance process. This routine will scan the journal table for rows in the failed state and will try to sync the resource for that entry.

The procedure can be best described by the following flow chart:

asciiflow:



```
+-----------------+
| For each entry  |
| in failed state |
+-------+---------+
        |
+-------v--------+
| Query resource |
| on ODL (REST)  |
+-----+-----+----+
      |     |                       +-----------+
   Resource |                       | Determine |
   exists   +--Resource doesn't exist--> operation |
      |                             | type      |
+-----v-----+                       +-----+-----+
| Determine |                             |
| operation |                             |
| type      |                             |
+-----+-----+                             |
      |              +------------+       |
      +--Create------> Mark entry <--Delete--+
      |              | completed  |       |
      |              +----------^-+    Create/
      |                         |      Update
      |                         |         |
      |          +------------+ |   +-----v-----+
      +--Delete--> Mark entry | |   | Determine |
      |          | pending    | |   | parent    |
      |          +---------^--+ |   | relation  |
      |                    |    |   +-----+-----+
+-----v------+             |    |         |
| Compare to +--Different--+    |         |
| resource   |                  |         |
| in DB      +--Same------------+         |
+------------+                            |
                                          |
+-------------------+                     |
| Create entry for  <-----Has no parent------+
| resource creation |                     |
+---------^---------+               Has a parent
          |                               |
          |                     +---------v-----+
          +------Parent exists------+ Query parent  |
```

(continues on next page)

(continued from previous page)

```
                                           | on ODL (REST) |
                                           +--------+------+
+------------------+                                |
| Create entry for <---Parent doesn't exist--+
| parent creation  |
+------------------+
```

For every error during the process the entry will remain in failed state but the error shouldnt stop processing of further entries.

The implementation could be done in two phases where the parent handling is done in a second phase. For the first phase if we detect an entry that is in failed for a create/update operation and the resource doesnt exist on ODL we create a new create resource journal entry for the resource.

This proposal utilises the journal mechanism for its operation while the only part that deviates from the standard mode of operation is when it queries ODL directly. This direct query has to be done to get ODLs representation of the resource.

Performance Impact

The maintenance thread will have another task to handle. This can lead to longer processing time and even cause the thread to skip an iteration. This is not an issue since the maintenance thread runs in parallel and doesnt directly impact the responsiveness of the system.

Since most operations here involve I/O then CPU probably wont be impacted.

Network traffic would be impacted slightly since we will attempt to fetch the resource each time from ODL and we might attempt to fetch its parent. This is however negligible as we do this only for failed entries, which are expected to appear rarely.

Alternatives

The partial sync process could make this process obsolete (along with full sync), but its a far more complicated and problematic process. Its better to start with this process which is more lightweight and doable and consider partial sync in the future.

Assignee(s)

Primary assignee: mkolesni <mkolesni@redhat.com>

Other contributors: None

References

https://goo.gl/IOMpzJ

Newton specs

Quality of Service Driver for OpenDaylight

This spec describes the plan to implement quality of service driver for OpenDaylight Controller.

Problem Statement

OpenStack networking project (neutron [1]) have a extension plugin implemented and which expose api for quality of service that can be also be implemented by any backend networking service provider to support QoS. These APIs provide a way to integrate OpenStack Neutron QoS with any of the backend QoS providers. OpenDaylight will provide backend for existing functionalities in neutron-QoS. A notification driver is needed for integration of existing api in Openstack neutron for QoS with OpenDaylight backend.

Proposed Change

This change will introduce a new notification driver in networking-odl that will take CRUD requests data for QoS policies from OpenStack neutron and notify the OpenDaylight controller about the respective operation.

Detailed Design

To enable the formal end to end integration between OpenStack QoS and OpenDaylight requires an networking-odl QoS notification driver. QoS driver will act as a shim layer between OpenStack and OpenDaylight that will carry out following task:

1. **After getting QoS policy request data from neutron, It will log a operation** request in opendaylightjournal table.
2. **The operation will be picked from opendaylightjournal table and a rest call** for notifying OpenDaylight server will be prepared and sent.

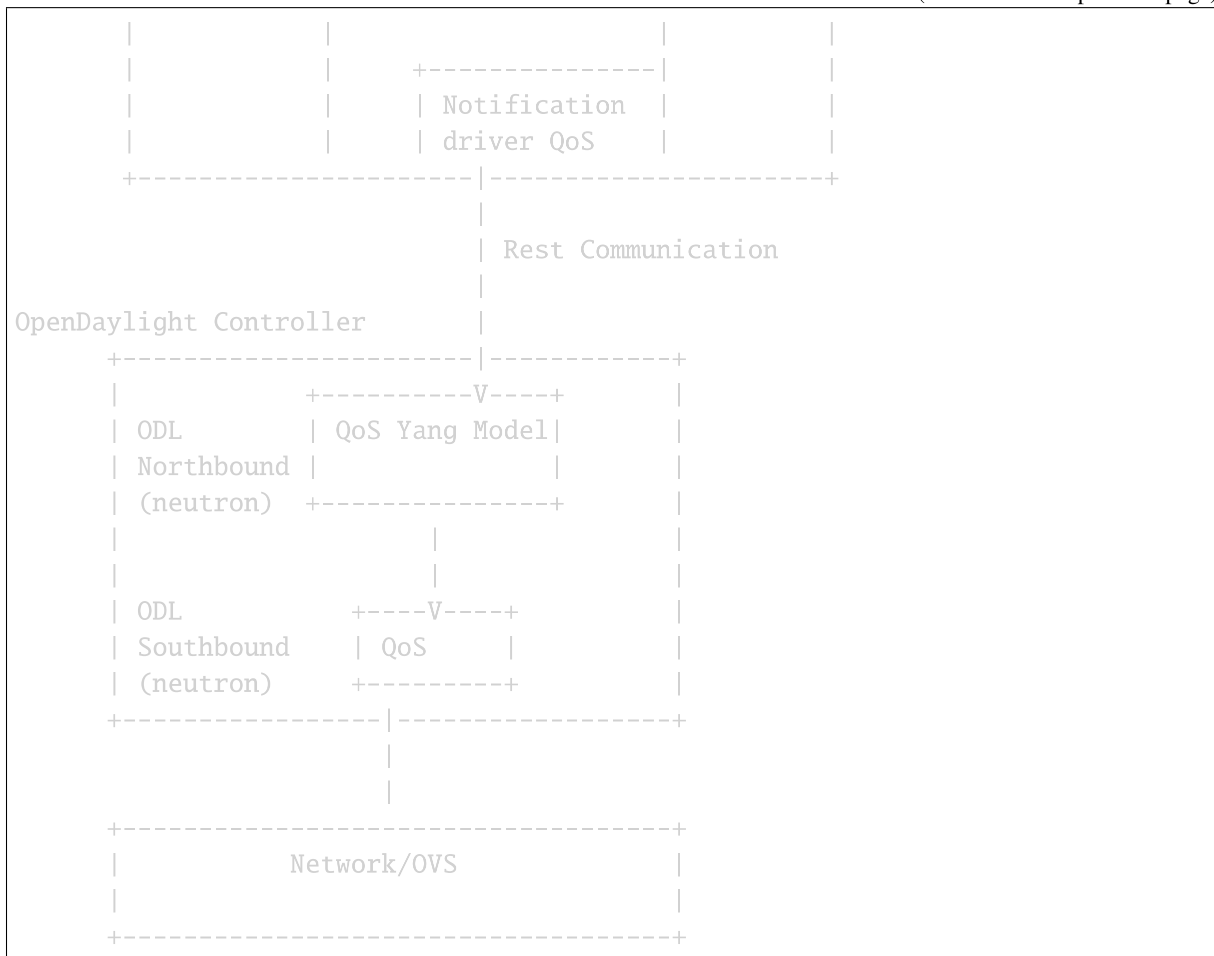
#. This request will processed by neutron northbound in OpenDaylight. The OpenDaylight neutron northbound project. These models will be based on the existing neutron qos plugin APIs.

QoS providers in OpenDaylight can listen to these OpenDaylight Neutron Northbound QoS models and translate it to their specific yang models for QoS. The following diagram shows the high level integration between OpenStack and the OpenDaylight QoS provider:

```
+---------------------------------------------+
| OpenStack Network Server (neutron qos)      |
|                                             |
|             +---------------------+         |
|             | networking-odl      |         |
```

(continues on next page)

(continued from previous page)



In the above diagram, the OpenDaylight components are shown just to understand the overall architecture, but its out of scope of this specs work items. This spec will only track progress related to networking-odl notification QoS driver work.

Dependencies

It has a dependency on OpenDaylight Neutron Northbound QoS yang models, but that is out of scope of this spec.

Impact

None

Assignee(s)

Following developers will be the initial contributor to the driver, but we will be happy to have more contributor on board.

- Manjeet Singh Bhatia (manjeet.s.bhatia@intel.com, irc: manjeets)

References

- [1] https://docs.openstack.org/neutron/latest/contributor/internals/quality_of_service.html
- [2] https://wiki.opendaylight.org/view/NeutronNorthbound:Main

Service Function Chaining Driver for OpenDaylight

This spec describes the plan to implement OpenStack networking-sfc[1] driver for OpenDaylight Controller.

Problem Statement

OpenStack SFC project (networking-sfc [1]) exposes generic APIs[2] for Service Function Chaining (SFC) that can be implemented by any backend networking service provider to support SFC. These APIs provide a way to integrate OpenStack SFC with any of the backend SFC providers. OpenDaylight SFC project provides a very mature implementation of SFC [3], but currently there is no formal integration mechanism present to consume OpenDaylight as an SFC provider for networking-sfc.

Recently Tacker project [4] has been approved as an official project in OpenStack, that opens many possibilities to realize the NFV use cases (e.g SFC) using OpenStack as a platform. Providing a formal end to end integration between OpenStack and OpenDaylight for SFC use case will help NFV users leverage OpenStack, Tacker and OpenDaylight as a solution. A POC for this integration work has already been implemented [5][6] by Tim Rozet, but in this POC work, Tacker directly communicates to OpenDaylight SFC & classifier providers and not through OpenStack SFC APIs (networking-sfc).

Proposed Change

Implementation of this spec will introduce a networking-sfc[1] driver for OpenDaylight Controller in networking-odl project that will pass through the networking-sfc APIs call to the OpenDaylight Controller.

Detailed Design

To enable the formal end to end integration between OpenStack SFC and OpenDaylight requires an SFC driver for OpenDaylight. ODL SFC driver will act as a shim layer between OpenStack and OpenDaylight that will carry out following two main tasks:

- Translation of OpenStack SFC Classifier API to ODL SFC classifier yang models**.
- Translation of OpenStack SFC APIs to OpenDaylight Neutron Northbound SFC models** [8].

** This work is not yet done, but the OpenDaylight neutron northbound project needs to come up with yang models for SFC classification/chain. These models will be based on the existing networking-sfc APIs. This work is out of scope of networking-odl work and will be collaborated in the scope of OpenDaylight Neutron Northbound project.

SFC providers (E.g Net-Virt, GBP, SFC) in OpenDaylight can listen to these OpenDaylight Neutron Northbound SFC models and translate it to their specific yang models for classification/sfc. The following diagram shows the high level integration between OpenStack and the OpenDaylight SFC provider:

```
            +---------------------------------------------+
            | OpenStack Network Server (networking-sfc)   |
            |            +-------------------+            |
            |            | networking-odl    |            |
            |            |   SFC Driver      |            |
            |            +-------------------+            |
            +----------------------|----------------------+
                                   | REST Communication
                                   |
                        -----------------------
OpenDaylight Controller |                     |
+-----------------------|---------------------|---------------+
|           +----------v----+              +---v---+          |
| Neutron   | SFC Classifier|              |SFC    | Neutron  |
| Northbound|    Models     |              |Models | Northbound|
| Project   +---------------+              +-------+ Project  |
|              /        \                      |              |
|            /           \                    |              |
|          /              \                   |              |
|     +-----V--+        +---V----+        +---V---+          |
|     |Net-Virt|  ...   |  GBP   |        |  SFC  |  ...     |
|     +---------+       +--------+        +-------+          |
+-----------|----------------|-----------------|---------------+
            |                |                 |
            |                |                 |
+-----------V----------------V-----------------V---------------+
|                     Network/OVS                              |
|                                                              |
+--------------------------------------------------------------+
```

In the above architecture, the opendaylight components are shown just to understand the overall architecture, but its out of scope of this specs work items. This spec will only track progress related to networking-odl OpenStack sfc driver work.

Given that OpenStack SFC APIs are port-pair based APIs and OpenDaylight SFC APIs are based on

IETF SFC yang models[8], there might be situations where translation might requires API enhancement from OpenStack SFC. Networking SFC team is open for these new enhancement requirements given that they are generic enough to be leveraged by other backend SFC providers[9]. This work will be leveraging the POC work done by Tim [10] to come up with the first version of SFC driver.

Dependencies

It has a dependency on OpenDaylight Neutron Northbound SFC classifier and chain yang models, but that is out of scope of this spec.

Impact

None

Assignee(s)

Following developers will be the initial contributor to the driver, but we will be happy to have more contributor on board.

- Anil Vishnoi (vishnoianil@gmail.com, irc: vishnoianil)
- Tim Rozet (trozet@redhat.com, irc: trozet)

References

[1] https://docs.openstack.org/networking-sfc/latest/

[2] https://github.com/openstack/networking-sfc/blob/master/doc/source/contributor/api.rst

[3] https://wiki.opendaylight.org/view/Service_Function_Chaining:Main

[4] https://wiki.openstack.org/wiki/Tacker

[5] https://github.com/trozet/tacker/tree/SFC_brahmaputra/tacker/sfc

[6] https://github.com/trozet/tacker/tree/SFC_brahmaputra/tacker/sfc_classifier

[7] https://tools.ietf.org/html/draft-ietf-netmod-acl-model-05

[8] https://wiki.opendaylight.org/view/NeutronNorthbound:Main

[9] http://eavesdrop.openstack.org/meetings/service_chaining/2016/service_chaining.2016-03-31-17.00.log.html

[10] https://github.com/trozet/tacker/blob/SFC_brahmaputra/tacker/sfc/drivers/opendaylight.py

5.1.2 Tutorial

Developer Quick-Start

This is a quick walkthrough to get you started developing code for networking-odl. This assumes you are already familiar with submitting code reviews to an OpenStack project.

Setup Dev Environment

Install OS-specific prerequisites:

```
# Ubuntu/Debian 14.04:
sudo apt-get update
sudo apt-get install python-dev libssl-dev libxml2-dev curl \
                     libmysqlclient-dev libxslt1-dev libpq-dev git \
                     libffi-dev gettext build-essential

# CentOS/RHEL 7.2:
sudo yum install python-devel openssl-devel mysql-devel curl \
                 libxml2-devel libxslt-devel postgresql-devel git \
                 libffi-devel gettext gcc

# openSUSE/SLE 12:
sudo zypper --non-interactive install git libffi-devel curl \
                    libmysqlclient-devel libopenssl-devel libxml2-devel \
                    libxslt-devel postgresql-devel python-devel \
                    gettext-runtime
```

Install pip:

```
curl -s https://bootstrap.pypa.io/get-pip.py | sudo python
```

Install common prerequisites:

```
sudo pip install virtualenv flake8 tox testrepository git-review
```

You may need to explicitly upgrade virtualenv if youve installed the one from your OS distribution and it is too old (tox will complain). You can upgrade it individually, if you need to:

```
sudo pip install -U virtualenv
```

Networking-odl source code should be pulled directly from git:

```
# from your home or source directory
cd ~
git clone https://opendev.org/openstack/networking-odl
cd networking-odl
```

For installation of networking-odl refer to *Installation Guide*. For testing refer to *Testing* guide.

Verifying Successful Installation

There are some checks you can run quickly to verify that networking-odl has been installed successfully.

1. Neutron agents must be in running state, if you are using pseudo-agent for port binding then output of **openstack network agent list** should be something like:

```
ubuntu@ubuntu-14:~/devstack$ openstack network agent list
+-----------------------------+----------------+-----------+------------
↪------+-------+-------+----------------------------+
| ID                          | Agent Type     | Host      |␣
↪Availability Zone | Alive | State | Binary                         |
+-----------------------------+----------------+-----------+------------
↪------+-------+-------+----------------------------+
| 00628905-6550-43a5-9cda-    | ODL L2         | ubuntu-14 | None        ␣
↪      | True  | UP    | neutron-odlagent-          |
| 175a309ea538                |                |           |             ␣
↪      |       |       | portbinding                |
| 37491134-df2a-              | DHCP agent     | ubuntu-14 | nova        ␣
↪      | True  | UP    | neutron-dhcp-agent         |
| 45ab-8373-e186154aebee      |                |           |             ␣
↪      |       |       |                            |
| 8e0e5614-4d68-4a42-aacb-    | Metadata agent | ubuntu-14 | None        ␣
↪      | True  | UP    | neutron-metadata-agent     |
| d0a10df470fb                |                |           |             ␣
↪      |       |       |                            |
+-----------------------------+----------------+-----------+------------
↪------+-------+-------+----------------------------+

Your output of this command may vary depending on the your environment,
for example hostname etc.
```

2. **You can check that opendaylight is running by executing following** command:

```
ubuntu@ubuntu-14:~/devstack$ ps -eaf | grep opendaylight
```

Launching Instance and floating IP

1. Gather paramters required for launching instance. We need flavor Id, image Id and network id, following comand can be used for launching an instance:

```
openstack server create --flavor <flavor(m1.tiny)> --image \
<image(cirros)> --nic net-id=<Network ID> --security-group \
<security group(default)> --key-name <keyname(mykey)> \
<server name(test-instance)>
```

For details on creating instances refer to[1] and[2].

2. Attaching floating IPs to created server can be done by following command:

[1] https://docs.openstack.org/mitaka/install-guide-rdo/launch-instance-selfservice.html
[2] https://docs.openstack.org/draft/install-guide-rdo/launch-instance.html

```
openstack server add floating ip <INSTANCE_NAME_OR_ID(test-instance) \
<FLOATING_IP_ADDRESS(203.20.2.12)>
```

For details on attaching floating IPs refer to[3].

Useful Commands

1. For verifying status try following command:

```
ubuntu@ubuntu-14:<Location of opendaylight directory>/distribution-karaf-
↪0.6.0-SNAPSHOT/bin$ ./karaf status
```

You should receive following output:

```
Running ...
```

2. You can login using available client:

```
ubuntu@ubuntu-14:<Location of opendaylight directory>/distribution-karaf-
↪0.6.0-SNAPSHOT/bin$ ./client
```

You will receive output in following format:

```
Logging in as karaf
3877 [sshd-SshClient[6dbb137d]-nio2-thread-3] WARN org.apache.sshd.client.
↪keyverifier.AcceptAllServerKeyVerifier - Server at [/0.0.0.0:8101, RSA,␣
↪56:41:48:1c:38:3b:73:a8:a5:96:8e:69:a5:4c:93:e0] presented unverified {}
↪ key: {}
 ________                       ________                .__  .__       .__
↪     __
 \_____  \ ______   ____   ____ \______ \ _____  ___.__.|  | |__| ____ |  ␣
↪|___/  |_
  /   |   \\____ \_/ __ \ /    \ |    |  \\__  \<   |  ||  | |  |/ ___\|  ␣
↪|  \   __\
 /    |    \  |_> >  ___/|   |  \|    `   \/ __ \\___  ||  |_|  / /_/  >  ␣
↪ Y  \  |
 \_______  /   __/ \___  >___|  /_______  (____  / ____||____/__\___  /|__
↪_|  /__|
         \/|__|        \/     \/        \/     \/\/            /_____/     ␣
↪  \/

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown␣
↪OpenDaylight.
```

Now you can run commands as per your for example:

[3] https://docs.openstack.org/user-guide/cli-manage-ip-addresses.html

```
opendaylight-user@root>subnet-show
No SubnetOpData configured.
Following subnetId is present in both subnetMap and subnetOpDataEntry



Following subnetId is present in subnetMap but not in subnetOpDataEntry

Uuid [_value=2131f292-732d-4ba4-b74e-d70c07eceeb4]

Uuid [_value=7a03e5d8-3adb-4b19-b1ec-a26691a08f26]

Uuid [_value=7cd269ea-e06a-4aa3-bc11-697d71be4cbd]

Uuid [_value=6da591bc-6bba-4c8a-a12b-671265898c4f]


Usage 1: To display subnetMaps for a given subnetId subnet-show --
↪subnetmap [<subnetId>]

Usage 2: To display subnetOpDataEntry for a given subnetId subnet-show --
↪subnetopdata [<subnetId>]
```

To get help on some command:

```
opendaylight-user@root>help feature
COMMANDS
info          Shows information about selected feature.
install       Installs a feature with the specified name and version.
list          Lists all existing features available from the defined␣
↪repositories.
repo-add      Add a features repository.
repo-list     Displays a list of all defined repositories.
repo-refresh Refresh a features repository.
repo-remove   Removes the specified repository features service.
uninstall     Uninstalls a feature with the specified name and version.
version-list Lists all versions of a feature available from the␣
↪currently available repositories.
```

There are other helpful commands, for example, log:tail, log:set, shutdown to get tail of logs, set log levels and shutdown.

For checking neutron bundle is installed:

```
opendaylight-user@root>feature:list -i | grep neutron
odl-neutron-service                         | 0.8.0-SNAPSHOT    | x     ␣
↪     | odl-neutron-0.8.0-SNAPSHOT              | OpenDaylight ::␣
↪Neutron :: API
odl-neutron-northbound-api                  | 0.8.0-SNAPSHOT    | x     ␣
↪     | odl-neutron-0.8.0-SNAPSHOT              | OpenDaylight ::␣
↪Neutron :: Northbound
```

(continues on next page)

(continued from previous page)

```
odl-neutron-spi                                  | 0.8.0-SNAPSHOT   | x    ␣
↪     | odl-neutron-0.8.0-SNAPSHOT                 | OpenDaylight ::␣
↪Neutron :: API
odl-neutron-transcriber                          | 0.8.0-SNAPSHOT   | x    ␣
↪     | odl-neutron-0.8.0-SNAPSHOT                 | OpenDaylight ::␣
↪Neutron :: Implementation
odl-neutron-logger                               | 0.8.0-SNAPSHOT   | x    ␣
↪     | odl-neutron-0.8.0-SNAPSHOT                 | OpenDaylight ::␣
↪Neutron :: Logger
```

For checking netvirt bundle is installed:

```
opendaylight-user@root>feature:list -i | grep netvirt
odl-netvirt-api                                  | 0.4.0-SNAPSHOT   | x    ␣
↪     | odl-netvirt-0.4.0-SNAPSHOT                 | OpenDaylight ::␣
↪NetVirt :: api
odl-netvirt-impl                                 | 0.4.0-SNAPSHOT   | x    ␣
↪     | odl-netvirt-0.4.0-SNAPSHOT                 | OpenDaylight ::␣
↪NetVirt :: impl
odl-netvirt-openstack                            | 0.4.0-SNAPSHOT   | x    ␣
↪     | odl-netvirt-0.4.0-SNAPSHOT                 | OpenDaylight ::␣
↪NetVirt :: OpenStack
```

3. For exploration of APIs following links can be used:

```
API explorer:
  http://localhost:8080/apidoc/explorer

Karaf:
  http://localhost:8181/apidoc/explorer/index.html
```

Detailed information can be found[4].

References

5.1.3 Networking OpenDayLight Internals

Host Configuration

Overview

ODL is agentless configuration. In this scenario Host Configuration is used to specify the physical host type and other configurations for the host system. This information is populated by the Cloud Operator is in OVSDB in Open_vSwitch configuration data in the external_ids field as a key value pair. This information is then read by ODL and made available to networking-odl through REST API. Networking-odl populates this information in agent_db in Neutron and is then used by Neutron scheduler. This information is required for features like Port binding and Router scheduling.

[4] https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Restconf_API_Explorer

Refer to this link for detailed design for this feature.

https://docs.google.com/presentation/d/1kq0elysCDEmIWs3omTi5RoXTSBbrewn11Je2d26cI4M/edit?pref=2&pli=1#slide=id.g108988d1e3_0_6

Related ODL changes:

https://git.opendaylight.org/gerrit/#/c/36767/

https://git.opendaylight.org/gerrit/#/c/40143/

Host Configuration fields

- **host-id**

  This represents host identification string. This string will be stored in external_ids field with the key as odl_os_hostconfig_hostid. Refer to Neutron config definition for host field for details on this field.

  https://docs.openstack.org/kilo/config-reference/content/section_neutron.conf.html

- **host-type**

  The field is for type of the node. This value corresponds to agent_type in agent_db. Example value are ODL L2 and ODL L3 for Compute and Network node respectively. Same host can be configured to have multiple configurations and can therefore can have both L2, L3 and other configurations at the same time. This string will be populated by ODL based on the configurations available on the host. See example in section below.

- **config**

  This is the configuration data for the host type. Since same node can be configured to store multiple configurations different external_ids key value pair are used to store these configuration. The external_ids with keys as odl_os_hostconfig_config_odl_XXXXXXXX store different configurations. 8 characters after the suffix odl_os_hostconfig_config_odl are host type. ODL extracts these characters and store that as the host-type fields. For example odl_os_hostconfig_config_odl_l2, odl_os_hostconfig_config_odl_l3 keys are used to provide L2 and L3 configurations respectively. ODL will extract ODL L2 and ODL L3 as host-type field from these keys and populate host-type field.

Config is a Json string. Some examples of config:

OVS configuration example:

```
{"supported_vnic_types": [{
         "vnic_type": "normal",
         "vif_type": "ovs",
         "vif_details": "{}"
     }]
     "allowed_network_types": ["local", "flat", "gre", "vlan", "vxlan"]",
     "bridge_mappings": {"physnet1":"br-ex"}
}"
```

OVS SR-IOV Hardware Offload configuration example:

```
{"supported_vnic_types": [{
        "vnic_type": "normal",
        "vif_type": "ovs",
        "vif_details": "{}"},
        {"vnic_type": "direct",
        "vif_type": "ovs",
        "vif_details": "{}"}
    }]
    "allowed_network_types": ["local", "flat", "gre", "vlan", "vxlan"]",
    "bridge_mappings": {"physnet1":"br-ex"}
}"
```

OVS_DPDK configuration example:

```
{"supported_vnic_types": [{
        "vnic_type": "normal",
        "vif_type": "vhostuser",
        "vif_details": {
            "uuid": "TEST_UUID",
            "has_datapath_type_netdev": True,
            "support_vhost_user": True,
            "port_prefix": "vhu",
            # Assumption: /var/run mounted as tmpfs
            "vhostuser_socket_dir": "/var/run/openvswitch",
            "vhostuser_ovs_plug": True,
            "vhostuser_mode": "client",
            "vhostuser_socket": "/var/run/openvswitch/vhu$PORT_ID"}
    }]
    "allowed_network_types": ["local", "flat", "gre", "vlan", "vxlan"]",
    "bridge_mappings": {"physnet1":"br-ex"}
}"
```

VPP configuration example:

```
{ {"supported_vnic_types": [
    {"vnic_type": "normal",
     "vif_type": "vhostuser",
     "vif_details": {
         "uuid": "TEST_UUID",
         "has_datapath_type_netdev": True,
         "support_vhost_user": True,
         "port_prefix": "socket_",
         "vhostuser_socket_dir": "/tmp",
         "vhostuser_ovs_plug": True,
         "vhostuser_mode": "server",
         "vhostuser_socket": "/tmp/socket_$PORT_ID"
     }}],
    "allowed_network_types": ["local", "flat", "vlan", "vxlan", "gre"],
    "bridge_mappings": {"physnet1": "br-ex"}}}
```

Host Config URL

Url : https://ip:odlport/restconf/operational/neutron:neutron/hostconfigs/

Commands to setup host config in OVSDB

```
export OVSUUID=$(ovs-vsctl get Open_vSwitch . _uuid)
ovs-vsctl set Open_vSwitch $OVSUUID external_ids:odl_os_hostconfig_
↪hostid=test_host
ovs-vsctl set Open_vSwitch $OVSUUID external_ids:odl_os_hostconfig_config_odl_
↪l2 =
"{"supported_vnic_types": [{"vnic_type": "normal", "vif_type": "ovs", "vif_
↪details": {} }], "allowed_network_types": ["local"], "bridge_mappings": {
↪"physnet1":"br-ex"}}"
```

Example for host configuration

```
{
"hostconfigs": {
  "hostconfig": [
    {
      "host-id": "test_host1",
      "host-type": "ODL L2",
      "config":
      "{"supported_vnic_types": [{
          "vnic_type": "normal",
          "vif_type": "ovs",
          "vif_details": {}
      }]
      "allowed_network_types": ["local", "flat", "gre", "vlan", "vxlan"],
      "bridge_mappings": {"physnet1":"br-ex"}}"
    },
    {
      "host-id": "test_host2",
      "host-type": "ODL L3",
      "config": {}
    }]
  }
}
```

CHAPTER

SIX

REFERENCE DEPLOYMENT GUIDE

6.1 Reference Deployment

This document is intended to guide for versions of OpenStack and OpenDaylight components to use when OpenStack is deployed with OpenDaylight.

6.1.1 OpenStack Version Reference

Pike ODL Reference

Contents

- *Pike ODL Reference*
  - *OpenDaylight Components*

OpenDaylight Components

OpenDaylight Componetns	
Boron Snapshot	No
Carbon Snapshot	Yes
Nitrogen Snapshot	Yes
Netvirt	odl-openstack-netvirt

Ocata ODL Reference

Contents

- *Ocata ODL Reference*
  - *OpenDaylight Components*

OpenDaylight Components

With ocata legacy netvirt is recommended to use with boron snapshot. However legacy netvirt may not work properly with carbon snapshot onwards.

OpenDaylight Componetns	
Boron Snapshot	Yes
Carbon Snapshot	Yes
Nitrogen Snapshot	No
Netvirt	odl-openstack-netvirt

Newton ODL Reference

Contents

- *Newton ODL Reference*
  - *OpenDaylight Components*

OpenDaylight Components

OpenDaylight Components	
Boron Snapshot	Yes
Carbon Snapshot	No
Nitrogen Snapshot	No
Netvirt	odl-ovsdb-openstack