



**RED HAT®
TRAINING**



Comprehensive, hands-on training that solves real world problems

Red Hat OpenStack Administration III: Networking & Foundations of NFV

Student Workbook (ROLE)

**RED HAT
OPENSTACK
ADMINISTRATION
III: NETWORKING
& FOUNDATIONS
OF NFV**

Red Hat OpenStack Platform 10.0.5 CL310
Red Hat OpenStack Administration III: Networking &
Foundations of NFV
Edition 1 20180227 20180227

Authors: Fiona Anne Allen, Forrest Taylor, Morgan Weetman, Michael Jarrett,

Snehangshu Karmakar

Editor: Philip Sweany

Copyright © 2018 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are Copyright © 2018 Red Hat, Inc.

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of Red Hat, Inc.

This instructional program, including all material provided herein, is supplied without any guarantees from Red Hat, Inc. Red Hat, Inc. assumes no liability for damages or legal action arising from the use or misuse of contents or details contained herein.

If you believe Red Hat training materials are being used, copied, or otherwise improperly distributed please e-mail training@redhat.com or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, Hibernate, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a registered trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

The OpenStack® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Contributors: Rhys Oxenham, Nir Yechiel, Rob Locke, Rudolf Kastl, Joshua Jack

Document Conventions	ix
Notes and Warnings	ix
Introduction	xi
Red Hat OpenStack Administration III: Networking & Foundations of NFV	xi
Orientation to the Classroom Environment	xii
Internationalization	xxiv
1. Managing Networks in Linux	1
Managing Network Interfaces	2
Guided Exercise: Managing Network Interfaces	11
Implementing Virtual Bridges	14
Guided Exercise: Implementing Virtual Bridges	26
Managing Virtual Networking Devices	33
Guided Exercise: Managing Virtual Networking Devices	40
Implementing Network Namespaces	48
Guided Exercise: Implementing Network Namespaces	53
Quiz: Managing Networks in Linux	57
Summary	59
2. Managing OpenStack Networking Agents	61
Describing OpenStack Networking Architecture	62
Quiz: Describing OpenStack Networking Architecture	67
Managing the L2 and L3 Agents	69
Guided Exercise: Managing the L2 and L3 Agents	77
Administering the DHCP Agent	84
Guided Exercise: Administering the DHCP Agent	87
Managing OpenStack Networking Agents	94
Guided Exercise: Managing OpenStack Networking Agents	96
Lab: Managing OpenStack Networking Agents	101
Summary	111
3. Deploying IPv6 Networks	113
Explaining IPv6	114
Quiz: Explaining IPv6	120
Assigning IPv6 Addresses	122
Guided Exercise: Assigning IPv6 Addresses	131
Deploying IPv6	138
Guided Exercise: Deploying IPv6	144
Lab: Deploying IPv6 Networks	153
Summary	162
4. Provisioning OpenStack Networks	163
Provisioning VLAN Tenant Networks	164
Guided Exercise: Provisioning VLAN Tenant Networks	170
Provisioning VXLAN and GRE Tenant Networks	178
Guided Exercise: Provisioning GRE Tenant Networks	186
Provisioning Provider Networks with Subnet Pools	193
Guided Exercise: Provisioning With Subnet Pools	199
Lab: Provisioning OpenStack Networks	203
Summary	211
5. Implementing Distributed Virtual Routing	213
Describing DVR Architecture	214

Quiz: DVR Architecture	237
Managing DVR Routers	239
Guided Exercise: Managing DVR Routers	244
Quiz: Implementing Distributed Virtual Routing	251
Summary	253
6. Tuning NFV Performance	255
Describing the NFV Architecture	256
Quiz: Describing the NFV Architecture	262
Tuning Cloud Applications	264
Guided Exercise: Tuning Cloud Applications	273
Lab: Tuning NFV Performance	281
Summary	292
7. Implementing NFV Datapaths	293
Describing SR-IOV	294
Quiz: Describing SR-IOV	297
Configuring QoS and DSCP	299
Guided Exercise: Configuring QoS	304
Deploying OVS-DPDK	308
Guided Exercise: Deploying OVS-DPDK	323
Quiz: Implementing NFV Datapaths	331
Summary	333
8. Building Software-defined Networks with OpenDaylight	335
Describing OpenDaylight Architecture	336
Quiz: ODL Architecture	341
Implementing OpenDaylight	343
Guided Exercise: Implementing OpenDaylight	348
Quiz: Building Software-defined Networks with OpenDaylight	353
Summary	355
9. Lab: Comprehensive Review of Red Hat OpenStack Administration III	357
Lab: Tuning OpenStack Networking Performance	358
Lab: Deploying Tenant Networks	368

Document Conventions

Notes and Warnings



Note

"Notes" are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

"Important" boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled "Important" will not cause data loss, but may cause irritation and frustration.



Warning

"Warnings" should not be ignored. Ignoring warnings will most likely cause data loss.



References

"References" describe where to find external documentation relevant to a subject.

Introduction

Red Hat OpenStack Administration III: Networking & Foundations of NFV

Red Hat OpenStack Administration III (CL310) is targeted at network engineers, network operators, cloud operators, and cloud administrators who need to manage and tune OpenStack Networking for performance. In this course you will manage the OpenStack Networking service (Neutron) with Network Functions Virtualization (NFV) to enhance network performance.

You will also configure Distributed Virtual Routers (DVR), Open vSwitch with Data Plane Development Kit (OVS-DPDK) datapath, IPv6 networking in OpenStack, and deploy software-defined networking (SDN) with OpenDaylight (ODL).

Objectives

- Tune OpenStack Networking performance.
- Manage network interfaces, bridges, agents and virtual networking devices.
- Deploy IPv6 networks in OpenStack.
- Implement Distributed Virtual Routing (DVR).
- Implement network functions virtualization (NFV) datapaths.

Audience

- Network Engineers, Network Operators, Cloud Administrators, Cloud Operators

Prerequisites

- Red Hat Certified System Administrator (RHCSA in Red Hat Enterprise Linux) certification or equivalent experience.
- Red Hat Certified System Administrator (RHCSA in Red Hat OpenStack Platform) exam (EX210) or equivalent experience.

Orientation to the Classroom Environment

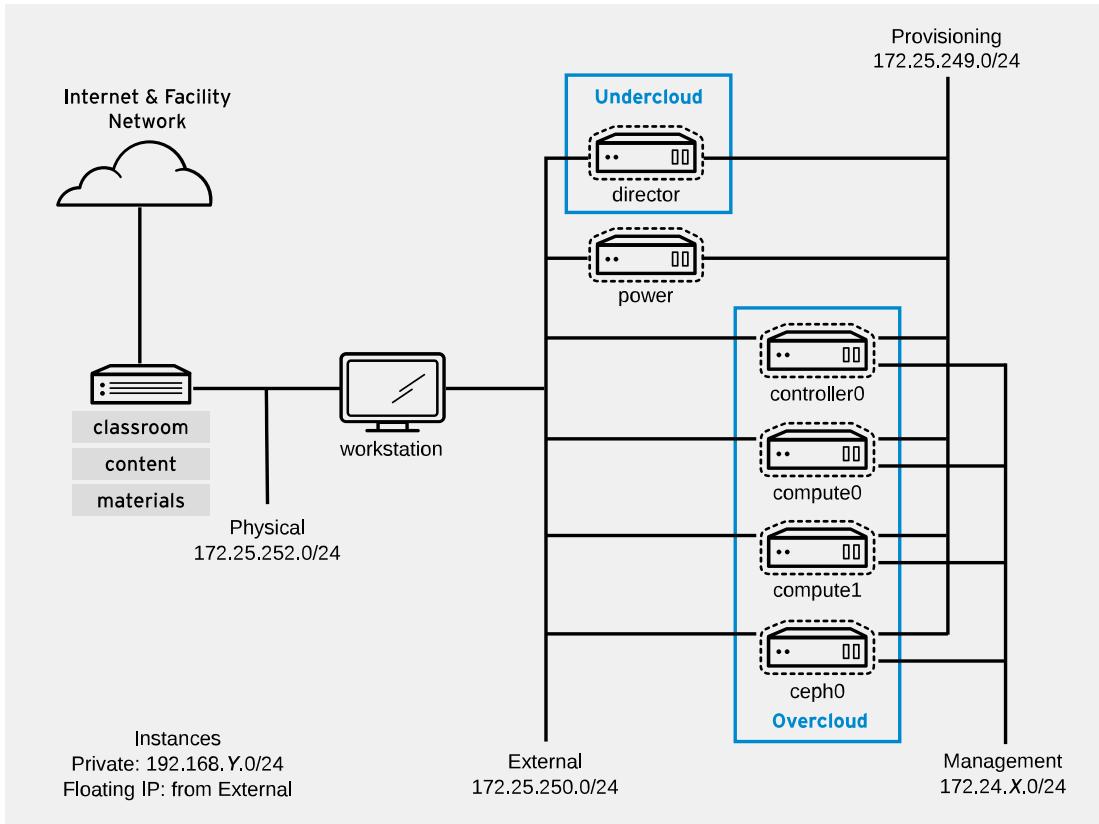


Figure 0.1: CL310 standard classroom architecture

Student systems share an external IPv4 network, **172.25.250.0/24**, with a gateway of **172.25.250.254** (**workstation.lab.example.com**). DNS services for the private network are provided by **172.25.250.254**. The OpenStack overcloud VMs share internal IPv4 networks, **172.24.X.0/24** and connect to the undercloud virtual machine and power interfaces on the **172.25.249.0/24** provisioning network. The tenant networks are commonly **192.168.Y.0/24** and allocate from **172.25.250.0/24** for public access.

The **workstation** virtual machine is the only one that provides a graphical user interface. In most cases, students should log in to the **workstation** virtual machine and use **ssh** to connect to the other virtual machines. A web browser can also be used to log in to the Red Hat OpenStack Platform Dashboard web interface. The following table lists the virtual machines that are available in the classroom environment:

Classroom Machines in the Standard and OpenDaylight Classrooms

Machine name	IP addresses	Role
workstation.lab.example.com , workstationN.example.com	172.25.250.254 172.25.252.N	Graphical workstation
director.lab.example.com	172.25.250.200 172.25.249.200	Undercloud node

Machine name	IP addresses	Role
power.lab.example.com	172.25.250.100 172.25.249.100 172.25.249.101+	IPMI power management of nodes
controller0.overcloud.example.com	172.25.250.1 172.25.249.P 172.24.X.1	Overcloud controller node
compute0.overcloud.example.com	172.25.250.2 172.25.249.R 172.24.X.2	Overcloud first compute node
compute1.overcloud.example.com	172.25.250.12 172.25.249.S 172.24.X.12	Overcloud additional compute node
ceph0.overcloud.example.com	172.25.250.3 172.25.249.T 172.24.X.3	Overcloud storage node
classroom.example.com	172.25.254.254, 172.25.252.254 172.25.253.254	Classroom utility server

The environment runs a central utility server, **classroom.example.com**, which acts as a NAT router for the classroom network to the outside world. It provides DNS, DHCP, HTTP, and other content services to the student lab machines. It uses two alternative names, **content.example.com** and **materials.example.com**, to provide course content used in the hands-on exercises.



Note

Access to the classroom utility server is restricted; shell access is unavailable.

System and Application Credentials

System credentials	User name	Password
Unprivileged shell login	student	student
Privileged shell login	root	redhat
Project member	operatorN, developerN	redhat
Project administrator	architectN	redhat

OpenStack Packages and Documentation

Repositories suitable for PM package installation are available locally in your environment at http://content.example.com/rhosp10.0.5/x86_64/dvd/.

Software documentation is available at <http://materials.example.com/docs/>, which contains subdirectories for files in PDF and single-page HTML format.

Introduction

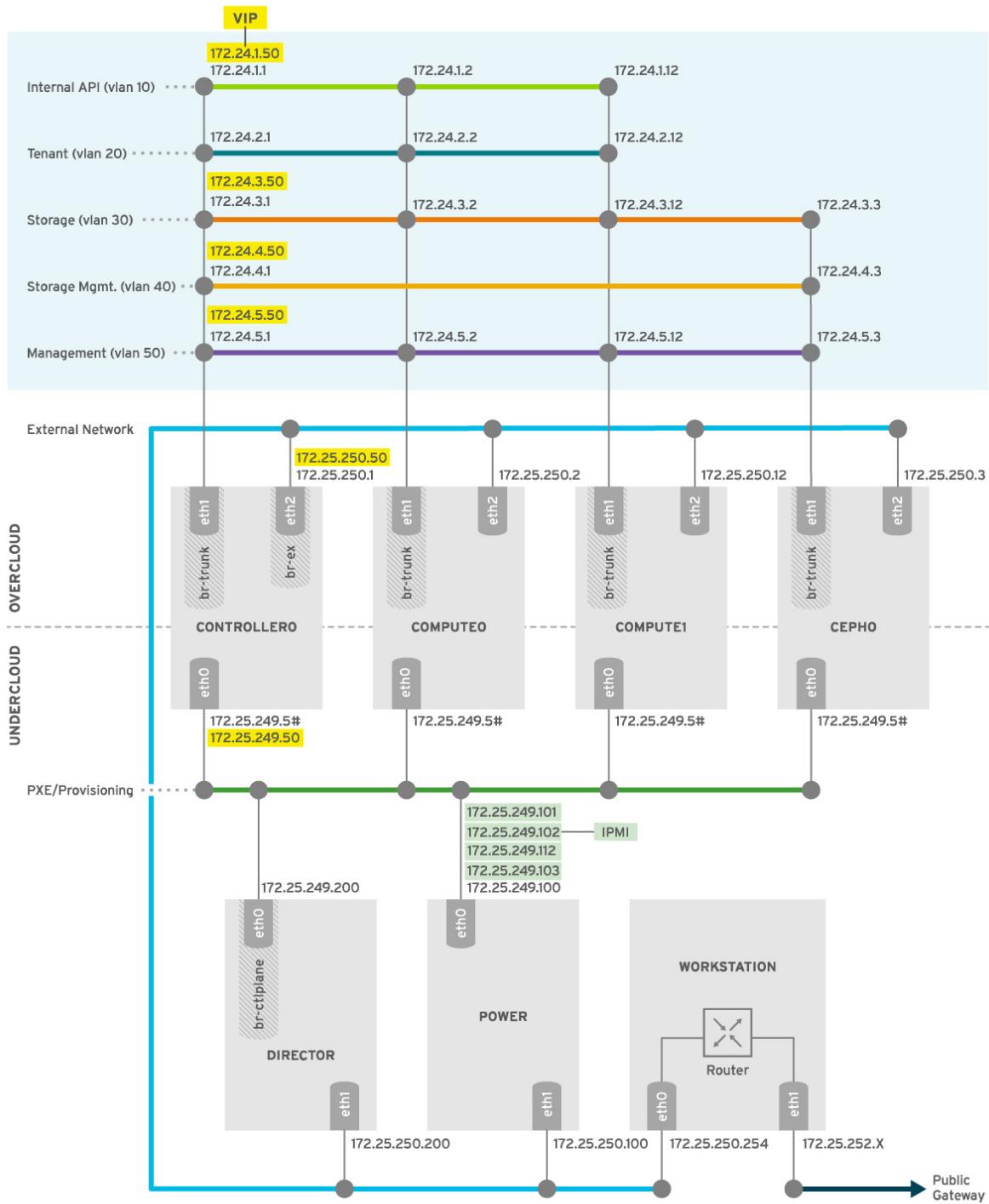


Figure 0.2: CL310 OpenStack infrastructure network layer

Red Hat Online Learning (ROL)

In a Red Hat Online Learning classroom, students are assigned remote computers accessed through a web application hosted at rol.redhat.com [<http://rol.redhat.com>]. Students log in to this machine using the user credentials provided when registering for this course.

CL310 Red Hat OpenStack Administration III

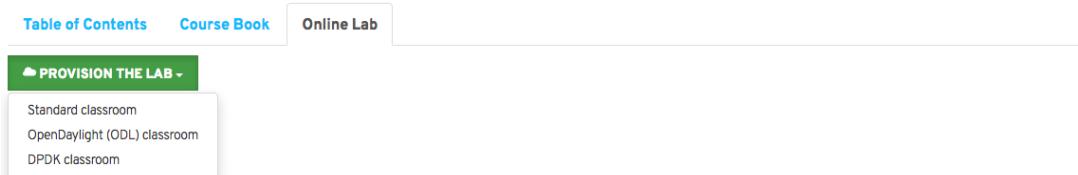


Figure 0.3: CL310 ROL classroom provision menu

CL310 Multiple Overclouds

This CL310 course is built with 3 separate OpenStack classrooms, each focusing on a single cloud architecture technology which is mutually exclusive to the others. The three OpenStack classrooms are:

- The **standard** classroom will be used during the majority of the course and includes all the technologies that are found in a default overcloud environment.
- The **opendaylight** classroom implements the OpenDaylight SDN controller, which replaces the default Open vSwitch SDN controller but retains other OVS components.
- The **dpdk** classroom is an OpenStack installation that showcases the OVS-DPDK technology, which is mutually exclusive with the **Distributed Virtual Routers** in the standard classroom.

Controlling the Virtual Machines

The state of each virtual machine in the classroom is displayed on under the **Online Lab** tab. Each classroom system is managed using controls on this page. Use the **PROVISION THE LAB** button to choose one of the three classrooms. Only one OpenStack classroom will be running at any time.

Introduction

Standard classroom



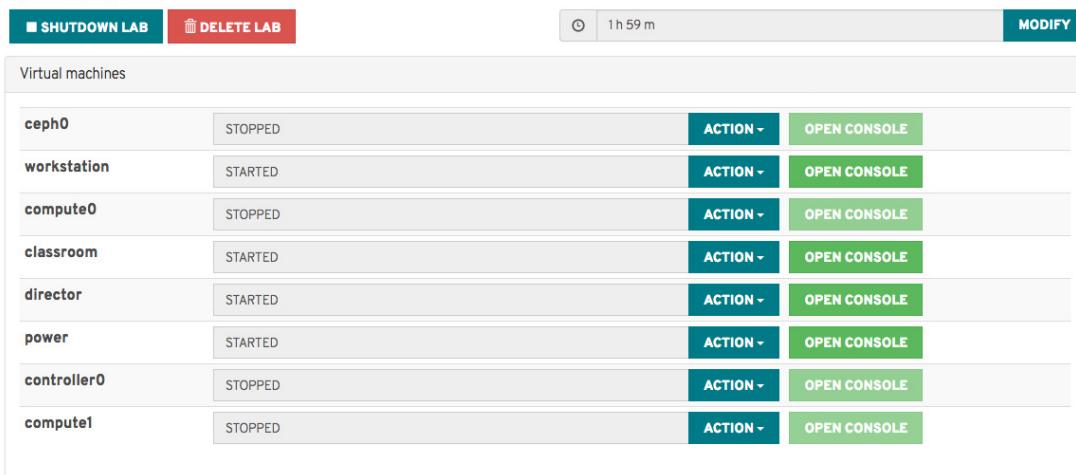
The screenshot shows a list of virtual machines in a standard classroom setup. The interface includes buttons for 'SHUTDOWN LAB' and 'DELETE LAB' at the top left, a timer showing '1 h 28 m' at the top center, and a 'MODIFY' button at the top right. The list of VMs is as follows:

VM Name	Status	Action	Console
ceph0	STOPPED	ACTION	OPEN CONSOLE
workstation	STARTED	ACTION	OPEN CONSOLE
compute0	STOPPED	ACTION	OPEN CONSOLE
classroom	STARTED	ACTION	OPEN CONSOLE
director	STARTED	ACTION	OPEN CONSOLE
power	STARTED	ACTION	OPEN CONSOLE
controller0	STOPPED	ACTION	OPEN CONSOLE
compute1	STOPPED	ACTION	OPEN CONSOLE



Figure 0.4: CL310 ROL standard classroom

OpenDaylight (ODL) classroom



The screenshot shows a list of virtual machines in an ODL classroom setup. The interface is identical to Figure 0.4, with 'SHUTDOWN LAB' and 'DELETE LAB' buttons, a '1 h 59 m' timer, and a 'MODIFY' button. The list of VMs is the same as in Figure 0.4:

VM Name	Status	Action	Console
ceph0	STOPPED	ACTION	OPEN CONSOLE
workstation	STARTED	ACTION	OPEN CONSOLE
compute0	STOPPED	ACTION	OPEN CONSOLE
classroom	STARTED	ACTION	OPEN CONSOLE
director	STARTED	ACTION	OPEN CONSOLE
power	STARTED	ACTION	OPEN CONSOLE
controller0	STOPPED	ACTION	OPEN CONSOLE
compute1	STOPPED	ACTION	OPEN CONSOLE



Figure 0.5: CL310 ROL OpenDaylight classroom

The standard and OpenDaylight classrooms in the two previous figures were built using an undercloud director node and TripleO provisioning of a controller node, compute nodes and a Ceph storage node.

The DPDK classroom in the next figure was built using a simpler PackStack installation of only a controller node and two compute nodes, with no director node or Ceph storage node. The primary reason for this design is to simplify the DPDK resource requirements for a classroom environment.



Figure 0.6: CL310 ROL DPDK classroom

Virtual Machine States

Provisioning or starting a classroom lab automatically starts some or all of the VMs in that classroom. In the TripleO-deployed standard and OpenDaylight classrooms, only the undercloud and supporting machines are started. The overcloud machines will be started manually using a lab script from the **workstation** machine. In the PackStack-installed DPDK classroom, all of the machines are started at once, because there is no need to wait for an undercloud before starting the overcloud.

Virtual Machine States

Virtual Machine State	Description
STARTING	The virtual machine is in the process of booting.
STARTED	The virtual machine is running and available (or, when booting, soon will be).
STOPPING	The virtual machine is in the process of shutting down.
STOPPED	The virtual machine is completely shut down. Upon starting, the virtual machine boots into the same state as when it was shut down (the disk will have been preserved).
PUBLISHING	The initial creation of the virtual machine is being performed.
WAITING_TO_START	The virtual machine is waiting for other virtual machines to start.

A selection of the following actions is available, depending on the current state of a machine.

Classroom or Machine Actions

Button or Action	Description
PROVISION THE LAB	Create the ROL classroom by choosing from the pulldown menu. Creates all of the virtual machines needed for the classroom and starts them. This can take several minutes to complete.
DELETE LAB	Delete the ROL classroom. Destroys all virtual machines in the classroom. Caution: Any work generated on the disks is lost.
START LAB	Start all virtual machines in the classroom.
SHUTDOWN LAB	Stop all virtual machines in the classroom.

Button or Action	Description
OPEN CONSOLE	Open a new tab in the browser and connect to the console of the virtual machine. Students can log in directly to the virtual machine and run commands. In most cases, students should first log in to the workstation virtual machine and then use ssh to connect to the other virtual machines.
ACTION > Start	Start (power on) the virtual machine.
ACTION > Shutdown	Gracefully shut down the virtual machine, preserving the contents of its disk.
ACTION > Power Off	Forcefully shut down the virtual machine, preserving the contents of its disk. This is equivalent to removing the power from a physical machine.
ACTION > Reset	Forcefully shut down the virtual machine and reset the disk to its initial state. Caution: Any work generated on the disk is lost.

At the start of an exercise, if instructed to change the classroom, press the **DELETE LAB** button to delete the classroom environment. After the lab has been deleted, press the **PROVISION THE LAB** button and choose the correct classroom to provision a new set of classroom systems.

The Autostop Timer

The Red Hat Online Learning enrollment entitles students to an agreed amount of computer time. To help conserve your allotted computer time, the ROL classroom has a countdown timer, which shuts down the classroom environment when the timer expires.

To adjust the timer, click **MODIFY**. A **New autostop time (in hours)** dialog box opens. Select an autostop time in hours (with a 4 hour maximum). Click **ADJUST TIME** to accept your selection.

Managing Two Types of OpenStack Installations

TripleO-deployed

OpenStack clouds can be built two ways, both of which have been used in this course. The recommended method is to use an enhanced default TripleO configuration deployed from an undercloud director system. This results in a recognizable classroom with a director system, plus one or more controller, compute and ceph overcloud nodes. In our virtual classroom environment, this design also requires a power node to handle power management for the overcloud nodes.

TripleO OpenStack installations require coordination between director, the power node and the overcloud nodes. Systems should be started and stopped in the correct order and using the correct commands so that nodes are not left in an recognizable or unstartable state. This is a classroom behavior only, because production OpenStack installations are built as redundant, resilient, scaled overclouds that are *never* shut down.

To ease management of a TripleO overcloud, this course includes the **lab overcloud** and **lab overcloud-health-check** commands to instruct the director node to start and stop the whole overcloud as a single entity, in an orderly and safe manner. Two of the three classrooms, standard and OpenDaylight, are TripleO-deployed installs that require these commands. The **lab overcloud** command has an imbedded **lab overcloud-health-check** command, so it is not necessary to run the health check immediately after starting the Overcloud.

The following are examples of the commands to work with either of those TripleO-deployed overclouds.

To start the Overcloud nodes and check the health of the Overcloud, run:

```
[student@workstation ~]$ lab overcloud start
```

You should expect that the Overcloud will continue to run correctly after you start the Overcloud, but you can verify that it is running properly. To check the health of the Overcloud during the time when the Overcloud is already running, execute the following command:

```
[student@workstation ~]$ lab overcloud-health-check setup
```



Note

The **lab setup** scripts always ask if you would like to run the health check. You can choose to run it there or run it separately as you would like.

If you would like to shut down the Overcloud cleanly, run:

```
[student@workstation ~]$ lab overcloud stop
```

PackStack-installed

The second method for installing OpenStack installations is PackStack. For the purpose of this explanation, this results in a cloud with one or more controller and compute nodes only. This method does not start with an undercloud director system, needs no power node, and uses local storage instead of separate ceph nodes. Because there is no director, there is no concept of undercloud and overcloud, and no need for the **lab overcloud** and **lab overcloud-health-check** commands that use director to start, stop, and check the health of the overcloud.

Instead, PackStack-installed clouds are managed using the standard classroom environment commands. The nodes will all be started when you switch to the **dpmk** classroom, so it is not expected that you will need to manage these nodes manually.

Starting the Overcloud from a New Provision

In the online learning environment, you must first select and provision a classroom. The selected classroom will continue to start each time you return to the online environment. The following table lists the expected beginning states for the nodes in each classroom:

Node Beginning States in Each Classroom

Node	standard	opendaylight	dpmk
ceph0	STOPPED	STOPPED	<i>does not exist</i>
classroom	STARTED	STARTED	STARTED
compute0	STOPPED	STOPPED	STARTED
compute1	STOPPED	STOPPED	STARTED
controller0	STOPPED	STOPPED	STARTED

Introduction

Node	standard	.opendaylight	dpldk
director	STARTED	STARTED	<i>does not exist</i>
power	STARTED	STARTED	<i>does not exist</i>
workstation	STARTED	STARTED	STARTED

Each classroom environment automatically starts specific nodes, shown in the preceding table. If expected nodes are not yet STARTED, then start them before continuing.

View the online course dashboard and click the **PROVISION THE LAB** button. From the pulldown menu, choose a classroom. If the lab has already been provisioned, click the **START LAB** button.

Wait until the expected nodes have finished booting and initializing services. The course dashboard displays **STARTED** when the nodes are initialized, but this is not an indication that the nodes have completed their startup procedures. Startup is expected to take up to 15 minutes. View the workstation console to monitor the startup.

When ready, open the workstation console to continue. Log in as **student**, password **student**.

Start your overcloud using the **lab overcloud start** command. Under all normal circumstances, do not use the online course dashboard buttons to start overcloud nodes!

The following example shows expected output when starting the Overcloud. The health check output has been omitted in this example.

```
[student@workstation ~]$ lab overcloud start

Starting the Overcloud for lab exercise work:

. Verifying director.lab.example.com is reachable..... SUCCESS
. Checking status of Nova on director.lab.example.com..... SUCCESS
. Pinging controller0..... NOREPLY
. Finding power state for controller0..... SUCCESS
. Finding task state for controller0..... SUCCESS
. Finding VM state for controller0..... SUCCESS
. Starting controller0..... SUCCESS
. Waiting for 10 seconds for controller0 to settle..... SUCCESS
. Pinging compute0..... NOREPLY
. Finding power state for compute0..... SUCCESS
. Finding task state for compute0..... SUCCESS
. Finding VM state for compute0..... SUCCESS
. Starting compute0..... SUCCESS
. Waiting for 3 seconds for compute0 to settle..... SUCCESS
. Pinging compute1..... NOREPLY
. Finding power state for compute1..... SUCCESS
. Finding task state for compute1..... SUCCESS
. Finding VM state for compute1..... SUCCESS
. Starting compute1..... SUCCESS
. Waiting for 3 seconds for compute1 to settle..... SUCCESS
. Pinging ceph0..... NOREPLY
. Finding power state for ceph0..... SUCCESS
. Finding task state for ceph0..... SUCCESS
. Finding VM state for ceph0..... SUCCESS
. Starting ceph0..... SUCCESS
. Waiting for 3 seconds for ceph0 to settle..... SUCCESS
. Waiting for ceph0 to become available..... SUCCESS
...output omitted...
```

Stopping Cleanly at the End of a Session

When finished for the day, or when you are done practicing for a while, recommended practice is to shut down your course lab environment safely. Start by shutting down the overcloud nodes.

```
[student@workstation ~]$ lab overcloud stop

Stopping the Overcloud:

. Verifying director.lab.example.com is reachable..... SUCCESS
. Checking status of Nova on director.lab.example.com..... SUCCESS
. Pinging compute0..... SUCCESS
. Finding power state for compute0..... SUCCESS
. Finding task state for compute0..... SUCCESS
. Finding VM state for compute0..... SUCCESS
. Stopping compute0..... SUCCESS
. Pinging compute1..... SUCCESS
. Finding power state for compute1..... SUCCESS
. Finding task state for compute1..... SUCCESS
. Finding VM state for compute1..... SUCCESS
. Stopping compute1..... SUCCESS
. Pinging ceph0..... SUCCESS
. Finding power state for ceph0..... SUCCESS
. Finding task state for ceph0..... SUCCESS
. Finding VM state for ceph0..... SUCCESS
. Stopping ceph0..... SUCCESS
. Pinging controller0..... SUCCESS
. Finding power state for controller0..... SUCCESS
. Finding task state for controller0..... SUCCESS
. Finding VM state for controller0..... SUCCESS
. Stopping controller0..... SUCCESS
```

Wait until OpenStack stops all overcloud nodes, then shut down the rest of the environment.

View the online course dashboard and each nodes' status text box. Wait for the overcloud nodes **controller0**, **ceph0**, **compute0**, and **compute1** to reach the **STOPPED** state, then click the **SHUTDOWN LAB** button. When all nodes are **STOPPED**, the course environment is cleanly shut down.

Starting After an Unclean Shutdown

If your classroom system or environment was shut down without using the clean shutdown procedure, it may result in an environment where the undercloud knowledge about the overcloud nodes does not match the physical or running status of the nodes. The **lab overcloud start** command will check the 3 different node states (power state, task state and VM state) and perform correct commands to start nodes successfully.

At this point, it is expected that the overcloud nodes are not running yet, because the course lab environment only auto-starts **classroom**, **workstation**, **power** and **director**. Check the online course dashboard to view each node's status, then start the overcloud.

```
[student@workstation ~]$ lab overcloud start
... output omitted ...
```

Checking the Health of the Overcloud Environment

The **overcloud-health-check** script checks the general status of an overcloud environment at any time. This script is invoked, with a prompt to allow skipping, at the beginning of every exercise setup script. Invoke this script manually at any time to verify the overcloud.

```
[student@workstation ~]$ lab overcloud-health-check setup

Checking the health of the overcloud:

This script's initial task thoroughly validates the overcloud environment,"  

taking a minute or more, but checking is not required before each exercise."  

If you are without overcloud problems, with a stable environment, say (n)o."  

Pressing 'Enter' or allowing the 20 second timeout will default to (n)o."  
  

You should *always* say (y)es if any of the following conditions are true:  

- You have just reset the overcloud nodes using "rht-vmctl reset" in ILT."  

- You have just reset the overcloud nodes from the Online course dashboard."  

- You have restarted or rebooted overcloud nodes or any critical services."  

- You suspect your environment has a problem and would prefer to validate."  
  

[?] Check the overcloud environment? (y|N)

Verifying overcloud nodes

· Retrieving state for controller0..... SUCCESS
· Retrieving state for ceph0..... SUCCESS
· Retrieving state for compute1..... SUCCESS
· Retrieving state for compute0..... SUCCESS
· Waiting for controller0 to be available..... SUCCESS
· Waiting for ceph0 to be available..... SUCCESS
· Waiting for compute1 to be available..... SUCCESS
· Waiting for compute0 to be available..... SUCCESS
· Verifying ceph0 access..... SUCCESS
· Starting ceph0 disk arrays and restarting ceph.target..... SUCCESS
· Verifying ceph0 service, please wait..... SUCCESS
· Checking RabbitMQ (5m timer)..... SUCCESS
· Ensuring the Downloads directory exists..... SUCCESS
· Ensuring OpenStack services are running, please wait..... SUCCESS
```

To Reset Your Environment

Critical concept

When the CL310 coursebook instructs to reset virtual machines, the intention is to reset only the overcloud to an initial state. Unless something else is wrong with any physical system or online environment that is deemed irreparable, there is no reason to reset all virtual machines or to re-provision a new lab environment.

What it means to reset the overcloud

Whether you are working in a physical or online environment, certain systems never need to be reset because they remain materially unaffected by exercises and labs. This table lists the systems never to be reset and those intended to be reset as a group during this course:

Which systems normally should or should not be reset

never to be reset	only reset as a group
· classroom	· controller0
· workstation	· compute0

never to be reset	only reset as a group
• power	• compute1 • ceph0 • director

Technically, the **director** system is the undercloud. However, in the context of resetting the overcloud, **director** must be included because director's services and databases are full of control, management, and monitoring information about the overcloud it is managing. Therefore, to reset the overcloud without resetting **director** is to load a fresh overcloud with **director** retaining stale information about the previous overcloud that was just discarded.

Click ACTION > Reset for **only director** and the undercloud nodes **controller0**, **compute0**, **compute1**, and **ceph0**.



Important

Never reset the power node in the online environment.

The **director** node is configured to start automatically, while the overcloud nodes are configured to not start automatically. This is the same behavior as a newly provisioned lab environment. Give **director** sufficient time to finish booting and initializing services, then run the **lab overcloud start** command from **workstation**.

```
[student@workstation ~]$ lab overcloud start
```

If you are still experiencing problems after performing resets, the remaining option is to re-provision the classroom. To reset **everything**, if deemed necessary, takes time but results in a fresh environment.

In the online environment, click the **DELETE LAB** button, wait, then click the **PROVISION THE LAB** button.

After the environment is re-provisioned, start again with the instructions for a new environment.

Lab Exercise Setup and Grading

Most activities use the **lab** command, executed on **workstation**, to prepare and evaluate exercises. The **lab** command takes two arguments: the activity's name and a verb of **setup**, **grade**, or **cleanup**.

- The **setup** verb is used at the beginning of an exercise or lab. It verifies that the systems are ready for the activity, possibly making some configuration changes to them.
- The **grade** verb is executed at the end of a lab. It provides external confirmation that the activity's requested steps were performed correctly.
- The **cleanup** verb can be used to selectively undo elements of the activity before moving on to later activities.

Internationalization

Language Support

Red Hat Enterprise Linux 7 officially supports 22 languages: English, Assamese, Bengali, Chinese (Simplified), Chinese (Traditional), French, German, Gujarati, Hindi, Italian, Japanese, Kannada, Korean, Malayalam, Marathi, Odia, Portuguese (Brazilian), Punjabi, Russian, Spanish, Tamil, and Telugu.

Per-user Language Selection

Users may prefer to use a different language for their desktop environment than the system-wide default. They may also want to set their account to use a different keyboard layout or input method.

Language Settings

In the GNOME desktop environment, the user may be prompted to set their preferred language and input method on first login. If not, then the easiest way for an individual user to adjust their preferred language and input method settings is to use the **Region & Language** application. Run the command **gnome-control-center region**, or from the top bar, select **(User) > Settings**. In the window that opens, select **Region & Language**. The user can click the **Language** box and select their preferred language from the list that appears. This will also update the **Formats** setting to the default for that language. The next time the user logs in, these changes will take full effect.

These settings affect the GNOME desktop environment and any applications, including **gnome-terminal**, started inside it. However, they do not apply to that account if accessed through an **ssh** login from a remote system or a local text console (such as **tty2**).



Note

A user can make their shell environment use the same **LANG** setting as their graphical environment, even when they log in through a text console or over **ssh**. One way to do this is to place code similar to the following in the user's **~/.bashrc** file. This example code will set the language used on a text login to match the one currently set for the user's GNOME desktop environment:

```
i=$(grep 'Language=' /var/lib/AccountService/users/${USER} \
| sed 's/Language=//')
if [ "$i" != "" ]; then
    export LANG=$i
fi
```

Japanese, Korean, Chinese, or other languages with a non-Latin character set may not display properly on local text consoles.

Individual commands can be made to use another language by setting the **LANG** variable on the command line:

```
[user@host ~]$ LANG=fr_FR.utf8 date
```

jeu. avril 24 17:55:01 CDT 2014

Subsequent commands will revert to using the system's default language for output. The **locale** command can be used to check the current value of **LANG** and other related environment variables.

Input Method Settings

GNOME 3 in Red Hat Enterprise Linux 7 automatically uses the **IBus** input method selection system, which makes it easy to change keyboard layouts and input methods quickly.

The **Region & Language** application can also be used to enable alternative input methods. In the **Region & Language** application's window, the **Input Sources** box shows what input methods are currently available. By default, **English (US)** may be the only available method. Highlight **English (US)** and click the **keyboard** icon to see the current keyboard layout.

To add another input method, click the **+** button at the bottom left of the **Input Sources** window. An **Add an Input Source** window will open. Select your language, and then your preferred input method or keyboard layout.

Once more than one input method is configured, the user can switch between them quickly by typing **Super+Space** (sometimes called **Windows+Space**). A *status indicator* will also appear in the GNOME top bar, which has two functions: It indicates which input method is active, and acts as a menu that can be used to switch between input methods or select advanced features of more complex input methods.

Some of the methods are marked with gears, which indicate that those methods have advanced configuration options and capabilities. For example, the Japanese **Japanese (Kana Kanji)** input method allows the user to pre-edit text in Latin and use **Down Arrow** and **Up Arrow** keys to select the correct characters to use.

US English speakers may find also this useful. For example, under **English (United States)** is the keyboard layout **English (international AltGr dead keys)**, which treats **AltGr** (or the right **Alt**) on a PC 104/105-key keyboard as a "secondary-shift" modifier key and dead key activation key for typing additional characters. There are also Dvorak and other alternative layouts available.



Note

Any Unicode character can be entered in the GNOME desktop environment if the user knows the character's Unicode code point, by typing **Ctrl+Shift+U**, followed by the code point. After **Ctrl+Shift+U** has been typed, an underlined **u** will be displayed to indicate that the system is waiting for Unicode code point entry.

For example, the lowercase Greek letter lambda has the code point U+03BB, and can be entered by typing **Ctrl+Shift+U**, then **03bb**, then **Enter**.

System-wide Default Language Settings

The system's default language is set to US English, using the UTF-8 encoding of Unicode as its character set (**en_US.utf8**), but this can be changed during or after installation.

From the command line, **root** can change the system-wide locale settings with the **localectl** command. If **localectl** is run with no arguments, it will display the current system-wide locale settings.

Introduction

To set the system-wide language, run the command **localectl set-locale LANG=locale**, where *locale* is the appropriate **\$LANG** from the "Language Codes Reference" table in this chapter. The change will take effect for users on their next login, and is stored in **/etc/locale.conf**.

```
[root@host ~]# localectl set-locale LANG=fr_FR.utf8
```

In GNOME, an administrative user can change this setting from **Region & Language** and clicking the **Login Screen** button at the upper-right corner of the window. Changing the **Language** of the login screen will also adjust the system-wide default language setting stored in the **/etc/locale.conf** configuration file.



Important

Local text consoles such as **tty2** are more limited in the fonts that they can display than **gnome-terminal** and **ssh** sessions. For example, Japanese, Korean, and Chinese characters may not display as expected on a local text console. For this reason, it may make sense to use English or another language with a Latin character set for the system's text console.

Likewise, local text consoles are more limited in the input methods they support, and this is managed separately from the graphical desktop environment. The available global input settings can be configured through **localectl** for both local text virtual consoles and the X11 graphical environment. See the **localectl(1)**, **kbd(4)**, and **vconsole.conf(5)** man pages for more information.

Language Packs

When using non-English languages, you may want to install additional "language packs" to provide additional translations, dictionaries, and so forth. To view the list of available langpacks, run **yum langavailable**. To view the list of langpacks currently installed on the system, run **yum langlist**. To add an additional langpack to the system, run **yum langinstall code**, where *code* is the code in square brackets after the language name in the output of **yum langavailable**.



References

locale(7), **localectl(1)**, **kbd(4)**, **locale.conf(5)**, **vconsole.conf(5)**,
unicode(7), **utf-8(7)**, and **yum-langpacks(8)** man pages

Conversions between the names of the graphical desktop environment's X11 layouts and their names in **localectl** can be found in the file **/usr/share/X11/xkb/rules/base.lst**.

Language Codes Reference

Language Codes

Language	\$LANG value
English (US)	en_US.utf8
Assamese	as_IN.utf8
Bengali	bn_IN.utf8
Chinese (Simplified)	zh_CN.utf8
Chinese (Traditional)	zh_TW.utf8
French	fr_FR.utf8
German	de_DE.utf8
Gujarati	gu_IN.utf8
Hindi	hi_IN.utf8
Italian	it_IT.utf8
Japanese	ja_JP.utf8
Kannada	kn_IN.utf8
Korean	ko_KR.utf8
Malayalam	ml_IN.utf8
Marathi	mr_IN.utf8
Odia	or_IN.utf8
Portuguese (Brazilian)	pt_BR.utf8
Punjabi	pa_IN.utf8
Russian	ru_RU.utf8
Spanish	es_ES.utf8
Tamil	ta_IN.utf8
Telugu	te_IN.utf8



CHAPTER 1

MANAGING NETWORKS IN LINUX

Overview	
Goal	Manage network interfaces, bridges, virtual networking devices, and network namespaces.
Objectives	<ul style="list-style-type: none">Describe Linux networking concepts and manage Linux network interfaces.Implement Linux and Open vSwitch bridges.Manage TAP, veth, and OVS patch pair devices.Create and manage network namespaces.
Sections	<ul style="list-style-type: none">Managing Network Interfaces (and Guided Exercise)Implementing Virtual Bridges (and Guided Exercise)Managing Virtual Networking Devices (and Guided Exercise)Implementing Network Namespaces (and Guided Exercise)
Quiz	Managing Networks in Linux

Managing Network Interfaces

Objective

After completing this section, students should be able to:

- Describe major Red Hat Linux networking technologies
- Describe Linux networking concepts.
- Manage Linux networking interfaces.

Managing Networks in the Private Cloud

Red Hat OpenStack Platform (RHOSP) is the de-facto choice for a virtualized infrastructure manager (VIM) for leading edge Network Function Virtualization (NFV) deployments. The OpenStack Networking Service's pluggable architecture is a key enabling technology for NFV. Many Red Hat-supported open source projects and communities are focused on accelerating cloud networking performance and NFV with the support of all major network equipment providers and telcos. A short list of relevant Red Hat technologies includes libvirt, Data Plane Development Kit (DPDK), Open vSwitch, QEMU/KVM, and Linux.

Red Hat's NFV focus is on the infrastructure (NFVI), especially the VIM layer for enabling Virtual Network Functions (VNFs). Red Hat partners with ISVs, for NFV management and organization (MANO), and with hardware providers, building the technology that enables advanced network functionality both in Red Hat OpenStack Platform and across the entire network stack. By developing the capabilities into the platform and operating system and leveraging those features and tools up into the SDN layer, high throughput is achieved for virtually all applications and use cases, while being fully integrated, supported and performance-tuned.

Below is a list of Red Hat's primary Red Hat Enterprise Linux networking features, as they apply to Neutron, RHOSP's Networking Service. Each of these technologies is included in this course.

Open vSwitch (OVS)

- A multi-layer software switch to forward traffic between virtual machines and physical or logical networks, OpenStack integrated via a Neutron ML2 driver and agent.
- Supports traffic isolation using overlay technologies (GRE, VXLAN) and 802.1Q VLANs.
- A multi-threaded user space switching daemon for increased scalability, support wildcard flows in kernel datapath, kernel-based hardware offload for GRE and VXLAN, and OpenFlow and OVSDB management protocols.

Data Plane Development Kit (DPDK)

- DPDK is a set of libraries and user-space drivers for fast packet processing, enabling applications to perform their own packet processing directly from/to the NIC.
- Delivers up to wire speed performance in specific use cases, including DPDK-accelerated VNFs in a guest VM.
- Accelerated Open vSwitch is a new package (*openvswitch-dpdk*) available with Red Hat OpenStack Platform, in which DPDK is bundled with OVS for better performance.

Single Root I/O Virtualization (SR-IOV)

- Allows a network adapter to isolate and share access resources among various PCIe hardware functions: both physical function (PF) and multiple virtual functions (VFs).

- Enables network traffic to bypass the software layer of the hypervisor and flow directly between the VF and the application's virtual machine. Achieves near line-rate performance without the need to dedicate a separate NIC to each individual virtual machine.
- In OpenStack, SR-IOV is implemented with a generic ML2 driver (**sriovnicswitch**). Optional advanced drivers, requiring NIC hardware support, are available. Supported network adapters are inherited from Red Hat Enterprise Linux.

Quality of Service (QoS)

- QoS is a resource control system that guarantees certain network requirements such as bandwidth, latency, and reliability.
- Network devices such as switches and routers can mark traffic so that it is handled with a higher priority to fulfill the QoS conditions.

vCPU Pinning, Large Pages, and NUMA awareness

- CPU pinning is the ability to run a specific virtual machine's virtual CPU on a specific physical CPU, in a specific host.
- Processes that use large amounts of memory and/or are otherwise memory intensive, benefit from being configured to use larger page sizes.
- NUMA awareness allows OpenStack to make smart scheduling and placement decisions when launching instances. Customized performance flavors target specialized workloads.

IPv6 support

- OpenStack Networking supports IPv6 tenant subnets in dual-stack configuration, so that projects can dynamically assign IPv6 addresses to virtual machines using Stateless Address Autoconfiguration (SLAAC) or DHCPv6.
- OpenStack Networking is also able to integrate with SLAAC on your physical routers, so that virtual machines can receive IPv6 addresses from an existing infrastructure.

OpenDaylight

- OpenDaylight delivers a common open source framework and platform for Software-Defined Networking (SDN), enabling users to program network layers, separating the data plane from the control plane.
- OpenDaylight allows agile placement of networking services when and where they are needed. By enabling wide programmability, OpenDaylight optimizes network resources, increases network agility, service innovation, and accelerates service time-to-availability.
- Red Hat's OpenDaylight distribution is OpenStack-centric, and only ships the required interfaces to OpenStack Networking and to Open vSwitch (via OVSDB NetVirt).

Linux Networking Concepts

Underneath the Software Defined Networking (SDN) layer of an OpenStack environment are Red Hat Enterprise Linux (RHEL) nodes that comprise the OpenStack infrastructure layer. In a proof-of-concept or development environment for practicing OpenStack, these systems could be either KVM virtual machines or baremetal systems that support RHEL. However, production OpenStack installations will always be baremetal hardware of sufficient quantity, size and configuration to handle the expected cloud workload to be deployed.

OpenStack installations require configuring each node's network interfaces for bridge encapsulation, handing over control of the infrastructure NICs to the Networking Service's Open vSwitch driver mechanisms. The systems in this course are configured similarly, with KVM VMs using virtual interface names, slightly different than a production OpenStack environment that would use a naming convention better suited for permanent enterprise installation.

Network Interface Naming

Red Hat Enterprise Linux 7 is the first release of RHEL which assigns network interface names using systemd's Predictable Network Interface naming scheme. This is described in detail in the RHEL 7 Networking Guide chapter on Consistent Device Naming. Red Hat strongly recommend that the new RHEL7 naming conventions are used.

At boot time, interfaces are *always* assigned ethX style names by the kernel, with the number assigned in the order that devices are detected in this boot event. Since the kernel has no ability to ensure drivers or interfaces are discovered in the same order every time, it is necessary to implement an additional OS function to rename interfaces in a predictable way such that each interface is assigned the same name on subsequent boots. In RHEL 7, this function is provided by systemd's Predictable Network Interface feature.

A previous alternate naming scheme, **biosdevname**, was designed and implemented by DELL using unique network interface information presented by the system BIOS. Only DELL hardware consistently provides the necessary information needed for biosdevname to work. Non-DELL systems can enable this scheme by ensuring that setting the **biosdevname=1** kernel parameter and ensuring the **biosdevname** package is installed. This scheme is disabled by default in RHEL 7, and systemd's Predictable Network Interface feature is preferred.

Without the Predictable Network Interface feature, network interfaces retain their original ethX style names. With no OS function to ensure specific interfaces are given the same name at each boot event, device naming becomes unpredictable and is unacceptable for an OpenStack installation. Predictable device naming uses udev rules to choose device names, in this order:

1. on-board devices with firmware-provided index numbers (example: eno1)
2. add-in devices with firmware-provided PCIe slot index numbers (example: ens1)
3. a device named using a physical connector location (example: enp2s0)
4. a device name incorporating the interface's MAC address (example: enx78e7d1ea46da)
5. finally, the classic, unpredictable kernel-native naming (example: eth0)

The only supported circumstances where it is safe to disable the Predictable Network Interface feature by setting **net.ifnames=0** are listed here:

- If the system only has a single interface and will never have more than a single interface.
- KVM guests exclusively using virtio-net type interfaces can safely set net.ifnames=0.
- If the system is custom configured for unique, non ethX style names using udev rules or the udev properties in the /usr/lib/udev/rules.d/60-net.rules file.

In this course, each node has three network interfaces, named using the ethX style, since the classroom is built with KVM guests using virtio-net. In this configuration, no naming order conflict would exist.

Disabling NetworkManager

NetworkManager is a dynamic network control and configuration system that attempts to keep network devices and connections up and active when they are available. NetworkManager can be used to configure Ethernet, wireless, and other network interface types like network aliases, static routes, DNS information and VPN connections, plus many connection-specific parameters. In enterprise computing, NetworkManager is an indispensable tool for managing and scaling complex networks.

In cloud computing, Software-Defined Networking (SDN) controllers configure and manage both the interfaces of the physical infrastructure, and the virtual interfaces and devices of the virtualized cloud environment, providing a complete view of the network topology and its

current state. NetworkManager's actions would be in conflict with SDN controllers. Therefore, the NetworkManager service is disabled by default as part of an OpenStack installation:

```
[root@demo ~]# systemctl stop NetworkManager
[root@demo ~]# systemctl disable NetworkManager
[root@demo ~]# systemctl mask NetworkManager
```

If NetworkManager is not masked, the service can still be started manually or if the network service is started when individual interface configurations include the setting for NetworkManager control. To ensure NetworkManager does not take control of an interface in case it is accidentally activated, add the **NM_CONTROLLED="no"** parameter to every **/etc/sysconfig/network-scripts/ifcfg-*** file.

Maximum Transmission Unit

Working with maximum transmission unit (MTU) values is a common skill for Linux network administrators. In SDN environments with layered, tunneled and mixed provider environments, understanding how the OpenStack Networking Service supports multiple MTU settings is necessary for successful and efficient network traffic. The MTU is the size of the largest network (layer 3) packet that can be transported across a single network connection. Common Ethernet, with a maximum frame size of 1518 bytes, with 18 bytes of overhead, results in a 1500 byte MTU.

The Networking service uses the MTU of the underlying physical network to calculate the MTU for virtual network components, including instance network interfaces. By default, a standard 1500-byte MTU is assumed for the underlying physical network. The Networking service supports physical networks using jumbo frames, enabling instances to use jumbo frames minus any overlay protocol overhead. For example, an underlying physical network with a 9000-byte MTU yields a 8950-byte MTU for instances using a VXLAN network with IPv4 endpoints. Using IPv6 endpoints for overlay networks adds 20 bytes of overhead for any protocol.

For typical underlying physical network architectures that implement a single MTU value, the physical layer value, configured in **neutron.conf** file, and the SDN layer, configured in the **ml2_conf.ini** file, are set to the same value. Some underlying physical network architectures contain multiple layer 2 networks with different MTU values. Each flat or VLAN provider network in the bridge or interface mapping options of the layer-2 agent to reference a unique MTU value.

Filename	Parameters
neutron.conf	[DEFAULT] global_physnet_mtu = 9000
openvswitch_agent.ini	[ovs] bridge_mappings = provider1:eth1,provider2:eth2,provider3:eth3
ml2_conf.ini	[ml2] physical_network_mtus = provider2:4000,provider3:1500 path_mtu = 9000

Some underlying physical network architectures contain a unique layer-2 network for overlay networks using protocols such as VXLAN and GRE. In this case, the **ml2_conf.ini** file **path_mtu** parameter is set to the MTU for the overlay network.

For deployed instances, the **advertise_mtu** option in the **neutron.conf** file enables the DHCP service to provide an appropriate MTU value to instances using IPv4 and enables the

L3 agent to provide an appropriate MTU value to instances using IPv6. IPv6 uses Router Advertisement via the L3 agent because the DHCP agent only supports IPv4. Instances using IPv4 and IPv6 should obtain the same MTU value regardless of method.

Review Layer 2 and Layer 3 Devices

The four most common network interactions correspond to operations that occur at different levels in the networking stack. In the OpenStack Networking Service, the common resources are bridges, routers, and services. Use the following figure as a reference when discussing software-defined networking, because SDN network stack interactions are identical to the same actions in hardware.

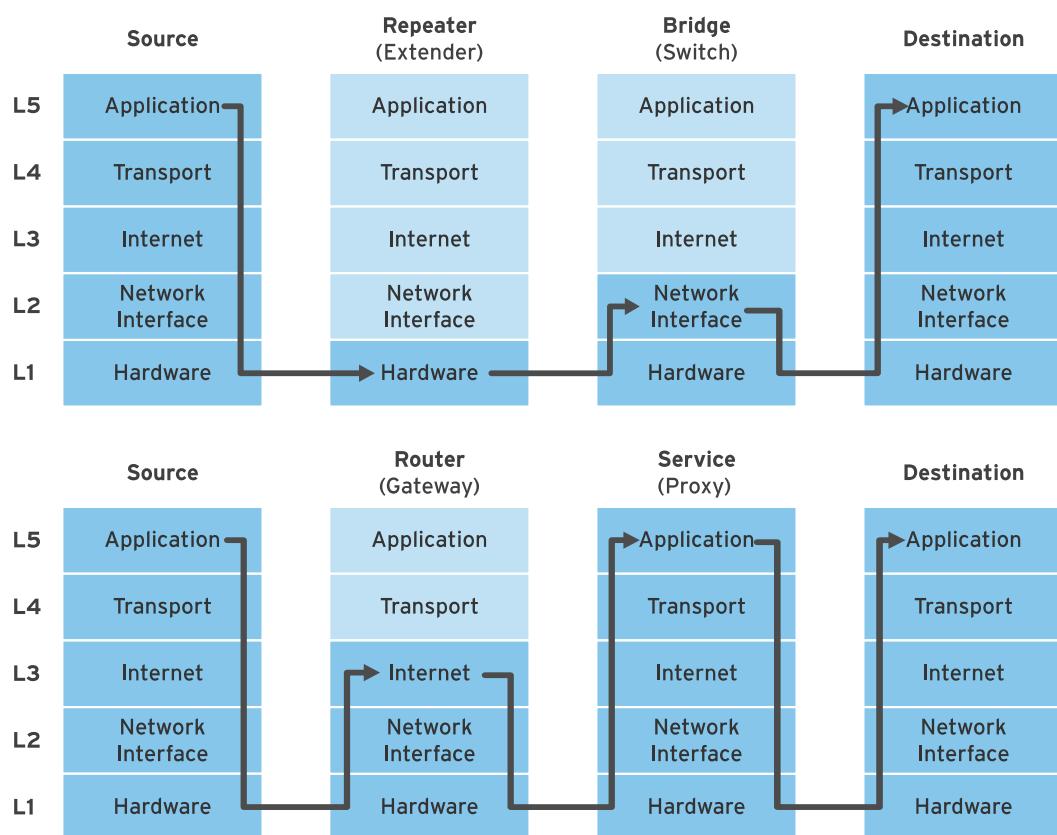


Figure 1.1: Reviewing layer 2 and 3 networking devices

Both Linux and Open vSwitch bridges are layer 2 devices that use VLAN IDs and MAC addresses to move packets to destinations. OVS routers, including NAT namespaces, operate at layer3 by performing packet address field modification. The hardware layer in SDN is virtualized, but the interfaces are visible using the same commands as for hardware.

The ip Command

The ip command has a number of subcommands. The most common will be used during this course: **address**, **link**, **route**, **neighbor discovery (neigh)** and **netns**.

Subcommand: **address** - Display IP Addresses and property information

Subcommand	Description
ip address	Show information for all addresses
ip address show dev em1	Display information for device em1 only

Subcommand: **link** - Manage and display the state of all network interfaces

Subcommand	Description
ip link	Show information for all interfaces
ip link show dev eth0	Display information for device eth0 only
ip -s link	Displays interface statistics

Subcommand: **route** - Display and alter the routing table

Subcommand	Description
ip route	List all of the route entries

Subcommand: **neigh** - Neighbor Discovery, the IPv6 answer to ARP

Subcommand	Description
ip -6 neigh show device_name	Display known or configured IPv6 neighbors
ip -6 neigh add IPv6 address lladdr link-layer address dev device	Manually add an entry
ip -6 neigh del IPv6 address lladdr link-layer address dev device	Manually delete an entry

Subcommand: **netns** - Network namespace management

Subcommand	Description
ip netns list	List all of named network namespaces
ip netns exec namespace command	Execute a command within a specific namespace
ip netns add name	Add a new namespace

The ip Command

Watch this video as the instructor shows how to use tcpdump and wireshark to analyze network traffic.

1. Use the **ip address show** command to view interfaces.
2. Use the **ip link** command to configure, maintain and view interfaces.
3. Use the **ip route** command to configure, maintain, and view routes.
4. The **ip link set** command can set the value of the MTU.

Managing Devices with the ethtool Command

Identifying device types can be achieved with the ethtool command. For example, the driver or the bus type can be retrieved:

```
[root@demo ~]# ethtool -i eth0
driver: virtio_net
version:
firmware-version:
bus-info: virtio0
supports-statistics: no
supports-test: no
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no
```

The **ethtool** command can also be used to change interface settings. Some useful syntax:

Command	Description
ethtool interface	List Ethernet device properties
ethtool -s eth0 speed 100 autoneg off	Change the speed of the Ethernet device
ethtool -i interface	Display Ethernet driver settings
ethtool -S interface	Display network statistics of a specific Ethernet device



Note

All network interfaces can be found under the **/sys/class/net** directory, which provides the active settings for every interface.

Capturing Packets with the tcpdump Command

The **tcpdump** command is a packet analyzer tool used to capture TCP/IP packets.

Command	Description
tcpdump -i enp0s31f6	Capture Packets from a specific interface
tcpdump -c 5 -i enp0s31f6	Capture a specific number of packets from a specific interface
tcpdump -D	Display all available interfaces
tcpdump -w 0001.pcap -i enp0s31f6	Save captured packets to file
tcpdump -r 0001.pcap	Read the captured packets file
tcpdump -n -i enp0s31f6	Capture only IP address packets on a specific interface
tcpdump -i enp0s31f6 tcp	Capture only TCP packets on a specific interface
tcpdump -i enp0s31f6 port 22	Capture packets from a specific port on a specific interface

Command	Description
<code>tcpdump -i enp0s31f6 src 192.168.10.23</code>	Capture packets from a specific source IP address on a specific interface
<code>tcpdump -i enp0s31f6 dst 57.25.45.139</code>	Capture packets from a specific destination IP address on a specific interface

Wireshark

Formerly known as Ethereal, Wireshark™ captures packets in real time and displays them in human readable format. It is an open source, graphical application for capturing, filtering, and inspecting network packets. Wireshark can perform promiscuous packet sniffing when network interface controllers support it. Packets are colorized for easy identification.

Protocol	Color
HTTP	Light green
TCP	Grey
UDP	Light blue
ARP	Light yellow
ICMP	Pink
Errors	Black

Red Hat Enterprise Linux 7 includes the `wireshark-gnome` package. This provides Wireshark functionality on a system with X installed.

```
[root@demo ~]# yum -y install wireshark-gnome
```

Wireshark is launched by selecting **Applications > Internet > Wireshark Network Analyzer** from the GNOME desktop. It can also be launched directly from a shell.

```
[root@demo ~]# wireshark
```

Capturing packets with Wireshark

Wireshark captures network packets. Direct access to the network interfaces requires root privilege. The user can start and stop Wireshark by using the top-level menu. The administrator can capture packets from any selected interface. The *any* interface matches all of the network interfaces.

Captured packets can be written to a file for sharing or for analysis after packet capturing has stopped. Wireshark supports a variety of file formats.

Wireshark can analyze captured packets that have been saved to a file. Analyzing packets from an existing capture file does not require root privilege.

```
[root@demo ~]# wireshark -r yesterday-eth0.pcapng
```

Inspecting packets with Wireshark

Users can filter expressions to limit which packets are displayed in Wireshark. Wireshark recognizes a variety of network protocols. Entering **http** filters captured packets so that only

HTTP TCP packets are displayed. Typing **ip** or **IPv6** filters packets so only IPv4 or IPv6 packets, respectively, are displayed.

The **Expression** button allows users to create more robust filtering expressions. The expression **ip.src == 192.168.10.23** filters the packets so only packets originating from 192.168.10.23 are displayed.

The frames in Wireshark's main display allows the user to inspect packet contents. The top frame displays the list of captured packet headers that have been selected with the current filter. The highlighted packet is the one currently being displayed in the middle and bottom frames. The bottom frame displays the whole packet, both headers and data, in hexadecimal and ASCII format.

The middle frame displays the packet as it was parsed by Wireshark. Each network layer header is displayed in a brief human-readable format. Expanding each header shows more detailed information about that network layer. Each line starts with the header field name followed by its value. Wireshark translates values to strings when possible. For example, when 22 is the value of a TCP port, it is displayed as **ssh** with 22 in parentheses. As each field is selected, the corresponding raw data is highlighted in the bottom frame.

Wireshark can display related packets between a protocol client and server in a easily readable format. Right-click a packet, then select **Follow TCP Stream**. Messages from the client display in one color and responses from the server display in another.

TCPDump and Wireshark

Watch this video as the instructor shows how to use tcpdump and wireshark to analyze network traffic.

1. Using VNC on the Horizon Dashboard, open the instance interface.
2. Using **tcpdump**, capture all DHCP packets.
3. Force interface eth0 down and then back up again.
4. Analyze the captured traffic.
5. Capture ICMP packets using tcpdump, and save them to file. Analyze the captured packets using both tcpdump and Wireshark.
6. Open **Wireshark** as **root**, and start traffic analysis on **all** interfaces. From **workstation**, create some HTTP traffic. Analyze the traffic using Wireshark.



References

ip(8), **ip-address(8)**, and **ip-link(8)** man pages.

Further information is available in the chapter on Configuring IP networking in the *Red Hat Enterprise Linux 7 Networking Guide* at

<https://access.redhat.com/documentation/en-US/index.html>

Guided Exercise: Managing Network Interfaces

In this exercise, you will use the **tcpdump** command to capture and analyze traffic. You will also use Wireshark to analyze captured traffic.

Outcomes

You should be able to:

- Consult the ARP table
- Capture traffic using the tcpdump command
- Analyze traffic using Wireshark

Before you begin

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab linux-interfaces setup** command. This script verifies that the overcloud nodes are accessible and are running the correct OpenStack services. It creates the instances used during this guided exercise.

```
[student@workstation ~]$ lab linux-interfaces setup
```

Steps

1. Using the **openstack server list**, retrieve the floating IP address of **finance-server1**. Log in to the instance using the **ssh** command. View the ARP table using the **arp** command. Log in to **finance-server2** and review the ARP table.
 - 1.1. Source the **developer1-finance** credentials. Use the **openstack server list** to retrieve the floating IP address of **finance-server1**. Use **ssh** to log in to **finance-server1** using **cloud-user**.

```
[student@workstation ~]$ source ~/developer1-finance-rc
[student@workstation ~(developer1-finance)]$ openstack server list \
-c Name -c Networks
+-----+-----+
| Name | Networks |
+-----+-----+
| finance-server1 | finance-network1=192.168.1.Q, 172.25.250.N |
| finance-server2 | finance-network1=192.168.1.R, 172.25.250.P |
+-----+
[student@workstation ~(developer1-finance)]$ ssh cloud-user@172.25.250.N
Last login: Thu Aug 31 08:02:38 2017 from workstation.lab.example.com
[cloud-user@finance-server1 ~]$
```

- 1.2. Review the ARP table.

```
[cloud-user@finance-server1 ~]$ arp
Address      HWtype  HWaddress          Flags Mask   Iface
gateway      ether    fa:16:3e:aa:71:7c  C          eth0
```

- 1.3. Ping **finance-server2** and review the ARP table again, and then exit from the instance.

```
[cloud-user@finance-server1 ~]$ ping -c 5 192.168.1.R
...output omitted...
64 bytes from 192.168.1.R: icmp_seq=3 ttl=64 time=1.8 ms
64 bytes from 192.168.1.R: icmp_seq=4 ttl=64 time=1.9 ms
64 bytes from 192.168.1.R: icmp_seq=5 ttl=64 time=2.3 ms
[cloud-user@finance-server1 ~]$ arp
Address          HWtype  HWaddress          Flags Mask   Iface
gateway         ether    fa:16:3e:aa:71:7c  C      eth0
192.168.1.R     ether    fa:16:3e:ef:32:e4  C      eth0
[cloud-user@finance-server1 ~]$ logout
[student@workstation ~(developer1-finance)]$
```

- 1.4. Log in to **finance-server2** and display the ARP table. You can see an entry for finance-server1 is already present due to the ping traffic.

```
[student@workstation ~(developer1-finance)]$ ssh cloud-user@172.25.250.P
Last login: Thu Aug 31 08:02:38 2017 from workstation.lab.example.com
[cloud-user@finance-server2 ~]$ arp
Address          HWtype  HWaddress          Flags Mask   Iface
gateway         ether    fa:16:3e:2d:f7:d8  C      eth0
192.168.1.Q     ether    fa:16:3e:f2:14:57  C      eth0
[cloud-user@finance-server2 ~]$ logout
[student@workstation ~(developer1-finance)]$
```

2. Using the **tcpdump** command, capture **SSH**, **HTTP**, **ICMP**, and **DNS** packets. Analyze the captured traffic using both **tcpdump** and Wireshark.
- 2.1. On **workstation** using administrator privileges, capture traffic using the **tcpdump** command. Using the **-w** option, write the capture to a file called **ge.pcap**.

```
[student@workstation ~(developer1-finance)]$ sudo tcpdump -i eth0 -w ~/ge.pcap
[sudo] password for student
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535
bytes
```

- 2.2. Open another terminal window. As **student** on **workstation**, run a series of commands to generate traffic.

```
[student@workstation ~]$ curl director
...output omitted...
[student@workstation ~]$ ping -c 3 172.25.250.N
64 bytes from 172.25.250.N: icmp_seq=1 ttl=63 time=1.8 ms
64 bytes from 172.25.250.N: icmp_seq=2 ttl=63 time=1.9 ms
64 bytes from 172.25.250.N: icmp_seq=3 ttl=63 time=2.3 ms
...output omitted...
[student@workstation ~]$ ssh cloud-user@172.25.250.N
[cloud-user@finance-server1 ~]$ exit
[student@workstation ~]$
```

- 2.3. Return to the terminal window running **tcpdump** and stop it with **Ctrl+C**.

```
[student@workstation ~(developer1-finance)]$ sudo tcpdump -i eth0 -w ~/ge.pcap
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535
bytes
```

CTRL+C

```
155 packets received by filter
0 packets dropped by kernel
[student@workstation ~(developer1-finance)]$
```

3. Using the **tcpdump** command, extract information from the captured traffic. The **icmp** option returns only **ICMP** captured packets. The **host 172.25.250.N** option returns only packets pertaining to that specific host.

```
[student@workstation ~(developer1-finance)]$ tcpdump -r ~/ge.pcap icmp
...output omitted...
14:40:41.749160 IP 172.25.250.254 > 172.25.250.107: ICMP echo request, id 2785, seq 1, length 64
14:40:41.749754 IP 172.25.250.107 > 172.25.250.254: ICMP echo reply, id 2785, seq 1, length 64
14:40:42.749476 IP 172.25.250.254 > 172.25.250.107: ICMP echo request, id 2785, seq 2, length 64
14:40:42.749874 IP 172.25.250.107 > 172.25.250.254: ICMP echo reply, id 2785, seq 2, length 64
14:40:46.029551 IP 172.25.250.254 > 172.25.250.106: ICMP echo request, id 2789, seq 1, length 64
...output omitted...
[student@workstation ~(developer1-finance)]$ tcpdump -r ~/ge.pcap host 172.25.250.N
...output omitted...
14:40:54.303354 IP 172.25.250.107.52964 > 172.25.250.254.domain: 13753+ A?
  workstation.lab.example.com. (45)
14:40:54.303380 IP 172.25.250.107.52964 > 172.25.250.254.domain: 54396+ AAAA?
  workstation.lab.example.com. (45)
14:40:54.303409 IP 172.25.250.254.domain > 172.25.250.107.52964: 13753* 1/0/0 A
  172.25.250.254 (61)
14:40:54.303422 IP 172.25.250.254.domain > 172.25.250.107.52964: 54396 0/0/0 (45)
...output omitted...
```

4. Using Wireshark, open the tcpdump file **ge.pcap** and analyze the HTTP traffic.

On **workstation**, go to **Applications > Internet > Wireshark Network Analyzer** and open the Wireshark Application. Click on **File** then **Open**, find and select **ge.pcap** in **student**'s home directory. Find an **HTTP** entry in the top pane, right click on the packet and choose **Follow TCP Stream**. Note the HTTP headers.

Cleanup

From **workstation**, run the **lab linux-interfaces cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab linux-interfaces cleanup
```

This concludes the guided exercise.

Implementing Virtual Bridges

Objectives

After completing this section, students should be able to:

- Manage Linux bridges.
- Manage OVS bridges.
- View network flows through bridges.

Bridges in the Classroom Environment

Refer to the following two OpenStack SDN configuration diagrams when discussing the OpenStack SDN general architecture as implemented in this classroom build. Many of these terms and concepts should already be familiar from field experience or attendance in the prerequisite OpenStack Administration I and II courses. Architecture and devices for distributed routing and provider networks information is introduced in this course.



Note

Refer to the CL310 OpenStack infrastructure diagram in the Introduction Chapter for an implementation diagram for the physical layer underneath the SDN layer. Because our courses are taught using single systems using KVM in an Instructor-led Classroom and in the cloud for the self-paced Red Hat Online Learning environment, our physical layer is actually virtual machines and power management emulation. Except for those details, this classroom OpenStack cloud is a default installation using recommended architecture practices, but without redundant components.

The external network on the left of the diagrams is connected to the public network of the organization. The next hop or destination for packets leaving this OpenStack installation is the enterprise switch or a gateway router. At the physical layer, the **eth2** network interface is implemented as a flat network. However, bridge **br-ex** was also configured to be able to use VLAN for external traffic.

The external network in our classroom, both for Instructor-led and Red Hat Online Learning, does not actually contain a switch or router. Therefore, there is no external device to interpret, tag or strip VLAN IDs. The **eth2** interface is configured as a trunk port so that it will not modify or interpret 802.1Q tags. Guided Exercises that practice provider or tenant VLAN configuration use the **br-ex** bridge because the lack of external verification allows this configuration to work.

The internal network, on the right, has no outside connectivity and is the private management and tenant backbone to the OpenStack installation. The next hop or destination for packets leaving an OpenStack node on the internal private side is another OpenStack node, either locally or through a tunneling protocol to a remote OpenStack installation that is part of this organization.

The internal network interface is **vlan20**, an 802.1Q tagged network shared on the trunked physical eth1 interface underneath. Refer to the Introduction for the infrastructure diagram to see **vlan20** and the **eth1** interface managed by the **br-trunk** Linux bridge. The OVS bridge

managing the **vlan20** interface is **br-tun**, which is configured to support only tunneling protocols, but no 802.1Q tagging.

Inside the OpenStack nodes are the remainder of the SDN resources created and managed as Open vSwitch entities. These diagrams contain a legacy (non-distributed) router on the controller node along with a non-distributed DHCP (dnsmasq) server. DHCP servers can also be located on compute nodes for resiliency. The SNAT namespace is a resource used by the distributed routing configuration.

On the compute node are a distributed router and the floating IP namespace that supplements the router. An example instance is shown here along with the now-deprecated Linux bridge to perform the security group protection for the instance. Current versions of OpenStack now incorporate the iptables functionality into Open vSwitch flow rules on **br-int**, eliminating the Linux **qbr** bridge.

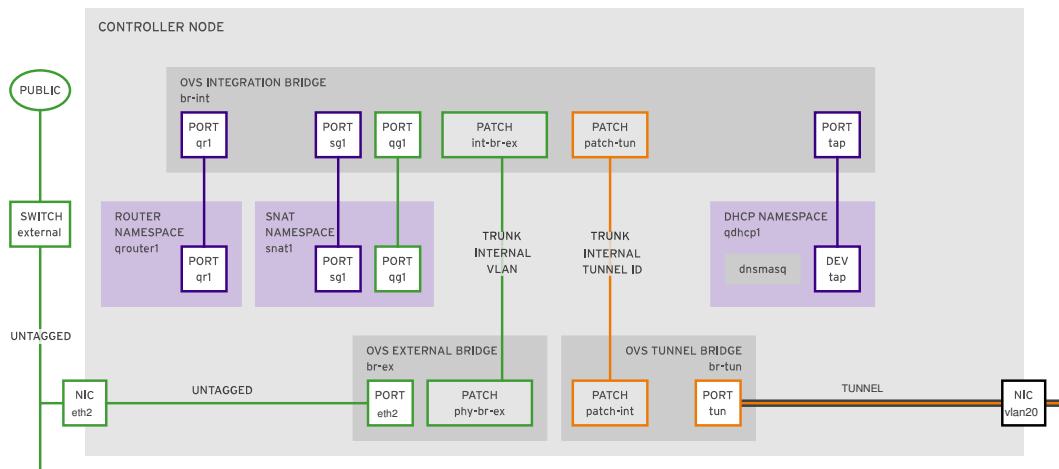


Figure 1.2: OpenStack SDN configuration in classroom environment - controller node

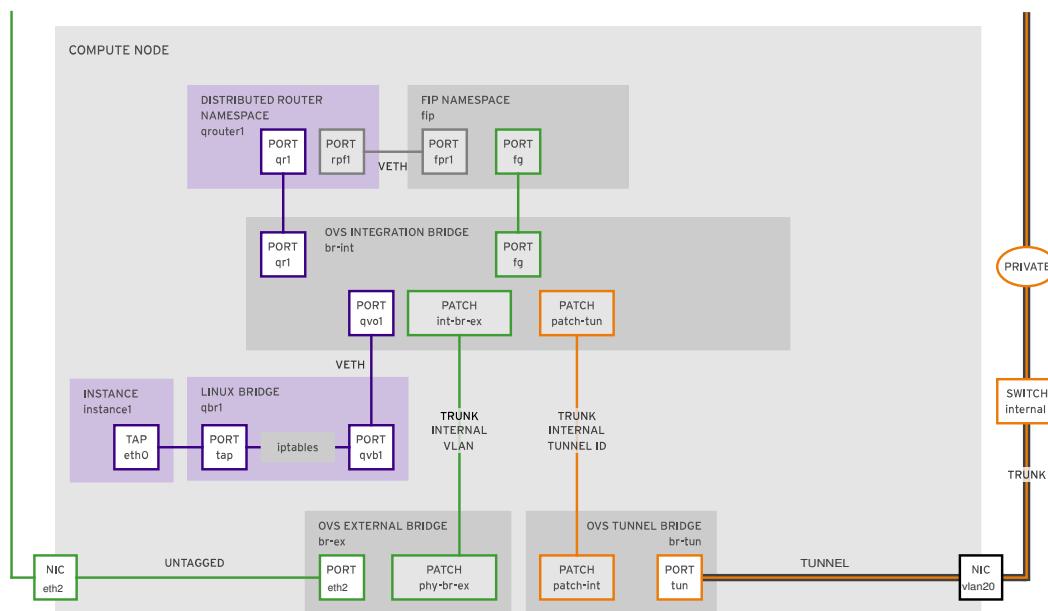


Figure 1.3: OpenStack SDN configuration in classroom environment - compute node

Implementing Linux Bridges

Network devices, switches or bridges, connect several segments with each other to create a local area network. These network bridges are a link layer devices which move traffic between segments based on MAC addresses. The MAC addresses table in these devices is built by learning what hosts are connected to each segment.

Software emulation of such devices on a Linux host is called software bridges or Linux bridges. Linux bridges are used in the virtualization environment to share a single NIC with one or more virtual NICs attached to one or more virtual instances.

Managing an Interface with the **brctl** Command

brctl is a command-line tool used to manage Linux bridges. Linux bridges provide an emulated switch that connects real and virtual interfaces within a kernel module. The bridge kernel module loads when the first virtual bridge is created. All network traffic is managed in the kernel's memory space.

- To create a bridge, use **brctl addbr**.

```
[user@demo ~]$ sudo brctl addbr demo-linbr0
```

- A common network design consists of removing the IP address from a physical interface, and attaching it to its corresponding bridge.

```
[user@demo ~]$ sudo ip addr flush eth1
[user@demo ~]$ sudo brctl addif demo-linbr0 eth1
```

- Once it has been attached, administrators can assign an IP address to the bridge.

```
[user@demo ~]$ sudo ip addr add 192.168.1.2/24 dev demo-linbr0
```

- Verify that the bridge and interfaces are correctly configured using **brctl show**.

```
[user@demo ~]$ brctl show demo-linbr0
bridge name      bridge id      STP enabled   interfaces
demo-linbr0      8000.52540001000c  no          eth1
[user@demo ~]$ brctl showmacs demo-linbr0
port no mac addr      is local?    ageing timer
      1 52:54:00:01:00:0c  yes          0.00
```

The kernel modules are automatically loaded when Linux bridges are created.

```
[root@demo ~]# lsmod |grep bridge
bridge                  115385  0
stp                      12976  1 bridge
llc                     14552  2 stp,bridge
```

Managing Persistent Linux Bridges

Bridges are made persistent with the creation of a file named **/etc/sysconfig/network-scripts/ifcfg-BRIDGE**. Bridges have an internal interface, which will have an autogenerated MAC address if one is not assigned. Configuration files for a bridge with an internal interface contain a special environment variable: **TYPE=Bridge**. Bridges created this way do not have

any external interfaces attached. Interfaces can be attached and made persistent by adding the variable **BRIDGE=<Bridge Name>** to their configuration files. IP addresses can also be assigned.

```
[root@demo ~]# cat /etc/sysconfig/network-scripts/ifcfg-eth1
DEVICE=eth1
BRIDGE=demo-linbr0
ONBOOT=yes
```

```
[root@demo ~]# cat /etc/sysconfig/network-scripts/ifcfg-demo-linbr0
DEVICE=demo-linbr0
TYPE=Bridge
IPADDR=192.168.1.2
PREFIX=24
ONBOOT=yes
```

Creating and Managing Linux Bridges

Bridges created with **brctl** are immediately available, but are not persistent across reboots. The following steps outline the process for creating and managing Linux bridges.

1. As an administrator, ensure that the *bridge-utils* package is installed.
2. To create a bridge, use the **brctl addbr** command.
3. Use **brctl addif** to add a network interface as a port to the bridge.
4. Assign an IP address to the Linux bridge using the **ip addr add** command. The device name passed as an argument to the command is the bridge name.
5. To verify that the bridge and interfaces are correctly configured, use the **brctl show** command.
6. Activate the bridge using the **ip link set dev** command.
7. Optionally, to make the Linux bridge configuration persistent, create a file in the **/etc/sysconfig/network-scripts** directory. The bridge name and the corresponding file name containing the bridge configuration is defined by the user. The file defines the IP address assigned to the bridge and sets the **TYPE** to **Bridge**.

Edit the network interface configuration file under **/etc/sysconfig/network-scripts** to add the interface as a port on the Linux bridge. Restart or reload the network service.

Linux Bridge on Compute Node

When an instance is launched, a new Linux bridge is created with the name **qbr** on the compute node. The **qbr** Linux bridge has two ports attached to the bridge. The **qvb** is the endpoint connected to the Linux bridge, which connects the **qbr** Linux bridge with the integration bridge. The **tap** virtual TAP device is the network interface of the instance. The ID associated with the Linux bridge name and virtual devices is that of the internal network port attached to the Neutron router. This screen output shows the internal network port ID associated with the instance on the router:

```
[user@demo ~]$ openstack port list -c ID -c 'Fixed IP Addresses' \
--device-owner compute:nova -f json
```

```
[  
  {  
    "Fixed IP Addresses": "ip_address='192.168.1.X', subnet_id='269d1a60-a956-450d-  
    b126-1fd9625a300f'",  
    "ID": "edfc3ab6-2b88-4db6-a568-2db6558804a5"  
  }  
]
```

Run the **brctl** command to review the Linux bridge and its ports.

```
[heat-admin@compute1 ~]$ brctl show  
bridge name      bridge id      STP enabled     interfaces  
qbredfc3ab6-2b1  8000.5ed59317502b no          qvbedfc3ab6-2b2  
                          tapedfc3ab6-2b3
```

- ① The Linux bridge attached to the instance.
- ② The endpoint of the veth pair connected to the Linux bridge.
- ③ The TAP device, which is the network interface of the instance.

Implementing OVS Bridges

Linux bridging does a good job of emulating a real bridge in the operating system. However, modern switches support many extensions to traditional Ethernet standards. The main purpose of Open vSwitch is to provide a switching stack for hardware virtualization environments, while supporting multiple protocols and standards used in computer networks. Open vSwitch can provide support for protocols such as VLAN, VXLAN, GRE, OpenFlow, NetFlow, sFlow, SPAN, RSPAN, CLI, LACP, and 802.1ag. It can also operate in a distributed configuration with centralized controllers provisioning the bridge. Both the Linux bridge module and Open vSwitch can coexist on the same switch. Open vSwitch can operate both as a software-based switch running on a virtual machine's hypervisors and as the control stack for dedicated switching hardware; as a result, it has been ported to multiple virtualization platforms and switching chipsets. Open vSwitch features include:

- Exposed communication between virtual machines by NetFlow, sFlow, IPFIX, SPAN, and RSPAN.
- Link aggregation through LACP.
- Standard 802.1Q VLAN model for network partitioning, with support for trunking.
- Support for the Spanning Tree Protocol.
- Fine-grained quality of service (QoS) control for different applications, users, or data flows.
- Network interface controller bonding, with load balancing by source MAC addresses.
- Support for the OpenFlow protocol, including various virtualization-related extensions.
- Full IPv6 support.
- Support for multiple tunneling protocols: GRE, VXLANs, IPsec.
- Remote configuration protocol.
- Implementation of the packet-forwarding engine in kernel space or user space, allowing additional flexibility as well as providing performance improvements by processing the majority of forwarded packets without leaving kernel space.

Managing an Interface with the **ovs-vsctl** Command

ovs-vsctl is one of several tools in the *openvswitch* package that administrators can use to manage Open vSwitch bridges. Bridges can be created, viewed, and destroyed. Ports can be added using the same utility. Advanced features include the ability to connect to an external

SDN controller. Bridges created with the **ovs-vsctl** command are active immediately and are persistent. The **ifcfg** files, as well as the **ip** command, can also be used to configure internal interfaces for an Open vSwitch.

The **ovs-vsctl** is available in the *openvswitch* package. Consider the following example to manage Open vSwitch bridges:

- Create a bridge.

```
[user@demo ~]$ sudo ovs-vsctl add-br demo-ovsbr0
```

- Prepare an interface before integrating it into an existing bridge.

```
[user@demo ~]$ sudo ip addr flush dev eth1
```

- Add an IP address to the **demo-ovsbr0** device.

```
[user@demo ~]$ sudo ip addr add 192.168.1.2/24 dev demo-ovsbr0
```

- Add the interface to the bridge.

```
[user@demo ~]$ sudo ovs-vsctl add-port demo-ovsbr0 eth1
```

- Bring up the bridge.

```
[user@demo ~]$ sudo ip link set dev demo-ovsbr0 up
```

- Check the bridge and the IP address allocated to the interface.

```
[user@demo ~]$ sudo ovs-vsctl show
    Bridge demo-ovsbr0
        Port demo-ovsbr0
            Interface demo-ovsbr0
                type: internal
        Port "eth1"
            Interface "eth1"
[user@demo ~]$ sudo ip addr show
42: demo-ovsbr0: <BROADCAST,UP,LOWER_UP> mu 1500 disc no queue state UNKNOWN
    link/ether ea:49:69:4e:93:4b bed ff:ff:ff:ff:ff:ff
    net 192.168.1.2/24 scope global demo-ovsbr0
        inet6 fe80::b8fb:8dff:fe35:2f31/64 scope link
            valid_lft forever preferred_lft forever
2: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mu 1500 disc FIFO_fast master demo-ovsbr0
    state UP glen 1000
    link/ether 52:54:00:bf:b7:d2 bed ff:ff:ff:ff:ff:ff
```

Creating and Managing OVS Bridges

OVS bridges created with the **ovs-vsctl** tool are immediately available and persist across reboots. The following steps outline the process for creating and managing OVS bridges.

1. As an administrator, ensure that the *openvswitch* package is installed. Ensure that the **openvswitch** service is running.

2. To create an OVS bridge, use the **ovs-vsctl addr-br** command.
3. Use the **ovs-vsctl add-port** command to add the interface to the bridge.
4. Assign an IP address to the OVS bridge. Bring up the OVS bridge device using the **ip link set dev BRIDGE up** command.
5. Confirm that the bridge is configured correctly using the **ovs-vsctl show** command. List the ports attached to the bridge using the **ovs-vsctl list-ports** command.
6. Optionally, to make the OVS bridge IP configuration persistent, create a file in the **/etc/sysconfig/network-scripts** directory. The file defines the IP address assigned to the bridge and sets the TYPE to **ovs**.

Edit the network interface configuration file under **/etc/sysconfig/network-scripts** to add the interface as a port on the OVS bridge. Restart or reload the network service.

OVS Bridges on Compute Node

The compute nodes deployed by Red Hat OpenStack Platform have four OVS bridges configured: **br-int**, **br-tun**, **br-trunk**, and **br-ex**. The integration bridge, **br-int**, performs VLAN tagging and untagging for traffic coming from and to the instance running on the compute node. The tunnel bridge, **br-tun**, translates VLAN tagged traffic from integration bridge to tunnel IDs using OpenFlow rules. The trunk bridge, **br-trunk**, separates network traffic types into different networks such as internal API, tenant, storage, storage management, and management networks. The external bridge, **br-ex**, forwards the traffic coming to and from the network to allow external access to instances.

External OVS Bridge on Compute Node

Using **br-ex** OVS bridge, the compute nodes connect to the external network and allows instances to communicate directly with the external network. The **br-ex** bridge connects physical ports, like eth2, so that the floating IP traffic for project networks is received from the physical network, and routed to the project network ports.

Run the **ovs-vsctl** command to review the implementation of the external OVS bridge on the compute node.

```
[heat-admin@compute1 ~]$ sudo ovs-vsctl show
...output omitted...
Bridge br-ex①
    Controller "tcp:127.0.0.1:6633"
        is_connected: true
        fail_mode: secure
    Port phy-br-ex
        Interface phy-br-ex②
            type: patch
            options: {peer=int-br-ex}
    Port br-ex
        Interface br-ex
            type: internal
    Port "eth2"
        Interface "eth2"③
...output omitted...
```

① External OVS bridge.

- ❷ Endpoint on the external bridge that connects to the integration bridge.
- ❸ External physical interface that routes the packets to and from an external network.

Integration Bridge on Compute Node

Packets leaving the network interface of the instance goes through the **qbr** Linux bridge using the virtual interface **qvo**. The interface **qvb** is connected to the Linux bridge, and **qvo** is connected to the **br-int** integration OVS bridge. The **qvo** port on the integration bridge has an internal VLAN tag that gets appended to the packet header when a packet reaches the integration bridge.

Apart from the connectivity to the **qbr** Linux bridge, the integration bridge also has interfaces that connect to the tunnel bridge and the external bridge on the compute node.

Review the integration OVS bridge on the compute node.

```
[heat-admin@compute1 ~]$ sudo ovs-vsctl show
...output omitted...
Bridge br-int①
    Controller "tcp:127.0.0.1:6633"
        is_connected: true
        fail_mode: secure
    Port br-int
        Interface br-int
            type: internal
    Port "qvoedfc3ab6-2b"
        tag: 1②
        Interface "qvoedfc3ab6-2b"③
    Port "qr-c29d0875-28"
        tag: 1
        Interface "qr-c29d0875-28"
            type: internal
    Port "fg-6333e87d-a0"
        tag: 2
        Interface "fg-6333e87d-a0"④
            type: internal
    Port patch-tun
        Interface patch-tun
            type: patch
            options: {peer=patch-int}
    Port int-br-ex
        Interface int-br-ex
            type: patch
            options: {peer=phy-br-ex}
...output omitted...
```

- ❶ Integration OVS bridge name.
- ❷ VLAN tag appended to the packet header on reaching the integration bridge.
- ❸ Endpoint on the integration bridge that connects to Linux bridge.
- ❹ Endpoint on the integration bridge that connects to the floating IP namespace of the distributed router.

Tunnel Bridge on Compute Node

The **br-tun** tunnel bridge allows communication between instances on different networks. In a distributed router, an instance with a fixed IP routes the traffic between project and external networks through the tunnel bridge.

Tunneling helps to encapsulate the traffic traveling over insecure networks. The tunnel bridge supports two forms of overlay networks: **Generic Routing Encapsulation** (GRE) and **Virtual extensible LAN** (VXLAN).

The Generic Routing Encapsulation (GRE) is used when transporting incompatible payload addresses, such as using the datagram protocol over IP in the transport layer.

The Virtual Extensible LAN (VXLAN) provides a solution to the scalability problem associated with VLAN usage, which limits the maximum number of concurrent tenant networks to 4096. VXLAN scales it up to 16 million concurrent tenant networks through the addition of a 24-bit segment ID. VXLANs encapsulate L2 networks over L3 networks.

The tunnel bridge translates VLAN-tagged traffic from the integration bridge into GRE or VXLAN tunnels. The translation between VLAN IDs and tunnel IDs is performed by OpenFlow rules installed on the tunnel bridge.

Review the tunnel OVS bridge on the compute node.

```
[heat-admin@compute1 ~]$ sudo ovs-vsctl show
...output omitted...
Bridge br-tun①
    Controller "tcp:127.0.0.1:6633"
        is_connected: true
    fail_mode: secure
    Port br-tun
        Interface br-tun
            type: internal
    Port "vxlan-ac180201"
        Interface "vxlan-ac180201"
            type: vxlan②
            options: {df_default="true", in_key=flow, local_ip="172.24.2.12",
out_key=flow, remote_ip="172.24.2.1"}
        Port patch-int
            Interface patch-int③
                type: patch
                options: {peer=patch-tun}
...output omitted...
```

- ① Tunnel OVS bridge name.
- ② Tunnel type.
- ③ Endpoint on the tunnel bridge connecting to integration bridge.

Trunk Bridge on Compute Node

In a typical Red Hat OpenStack Platform deployment, the number of network types required to be provisioned often exceeds the number of physical network interfaces. The Red Hat OpenStack Platform director provides a method to separate these various traffic types to certain VLANs. The choice of network traffic types is based on the deployment architecture. Commonly the traffic types includes:

- Internal API network
- Storage network
- Tenant network
- Management network

Review the trunk OVS bridge on the compute node. In the classroom Red Hat OpenStack Platform deployment, the trunk bridge isolates the traffic by creating VLANs on the physical interface **eth1**. There are four isolated traffic types on the compute node trunk bridge.

```
[heat-admin@compute1 ~]$ sudo ovs-vsctl show
...output omitted...
    Bridge br-trunk①
        fail_mode: standalone
        Port "eth1"
            Interface "eth1"
        Port "vlan20"②
            tag: 20
            Interface "vlan20"
                type: internal
        Port br-trunk
            Interface br-trunk
                type: internal
        Port "vlan30"③
            tag: 30
            Interface "vlan30"
                type: internal
        Port "vlan50"④
            tag: 50
            Interface "vlan50"
                type: internal
        Port "vlan10"⑤
            tag: 10
            Interface "vlan10"
                type: internal
...output omitted...
```

- ① Trunk bridge name
- ② Tenant network VLAN
- ③ Storage network VLAN
- ④ Management network VLAN
- ⑤ Internal API network VLAN

The storage management traffic runs on VLAN 40. The controller node connects over the storage management network to manage Ceph storage cluster.

Viewing OpenFlow Rules

The OpenFlow switch separates the data path portion on the switch, while high-level routing decisions are moved to a separate controller. An OpenFlow switch contains one or more flow tables. When a packet arrives to the switch, it is processed by the first flow table. If the packet does not match any flow entries in the table, the packet is dropped or passed to another table.

Using **ovs-ofctl** Command

The **ovs-ofctl** command-line interface is a tool for monitoring and managing OpenFlow switches. This tool can be used to show the current state of an OVS bridge, including features, configuration, and table entries. It is compatible with OpenFlow switches created by Open vSwitch or OpenFlow.

The following is a list of some of the current options for managing OpenFlow switches with the **ovs-vsctl** command.

- To display information on the switch and its ports in addition to limits, such as number of tables and number of buffers, use **ovs-ofctl show**.

```
[heat-admin@compute1 ~]$ sudo ovs-ofctl show br-ex
OFPT_FEATURES_REPLY (xid=0x2): dpid:000052540002fa0c
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP①
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src mod_dl_dst
mod_nw_src mod_nw_dst mod_nw_tos mod_tp_src mod_tp_dst②
    1(eth2): addr:52:54:00:02:fa:0c③
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
    2(phy-br-ex): addr:c2:88:5f:58:22:70④
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
    LOCAL(br-ex): addr:52:54:00:02:fa:0c⑤
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
```

- ① Capabilities supported by the OpenFlow switch.
- ② Actions supported by the OpenFlow switch.
- ③ Port with port ID 1 attached to the OpenFlow switch.
- ④ Port with port ID 2 attached to the OpenFlow switch.
- ⑤ OpenFlow switch name.

- To display the entries for all flows on a switch, use **ovs-ofctl dump-flows**.

```
[heat-admin@compute1 ~]$ sudo ovs-ofctl dump-flows br-ex
NXST_FLOW reply (xid=0x4):
..., table=0, ..., priority=4,in_port=2,dl_vlan=2 actions=strip_vlan,NORMAL①
..., table=0, ..., priority=2,in_port=2 actions=resubmit(,1)
..., table=0, ..., priority=0 actions=NORMAL
..., table=0, ..., priority=1 actions=resubmit(,3)
..., table=1, ..., priority=0 actions=resubmit(,2)
..., table=2, ..., priority=2,in_port=2 actions=drop②
..., table=3, ..., priority=2,dl_src=fa:16:3f:2a:b5:78 actions=output:2
..., table=3, ..., priority=2,dl_src=fa:16:3f:30:a6:af actions=output:2③
..., table=3, ..., priority=1 actions=NORMAL④
```

- ① The packet that arrives through port 2 (in_port=2) and has the destination VLAN ID set to 2 (dl_vlan=2) matches the rule. The rule has the priority of 4, which means if a packet matched several flow entries, with lower priorities this rule will have precedence. The rule performs the action to remove the VLAN tag and the OpenFlow switch will act as a traditional layer 2 switch (actions=strip_vlan, NORMAL).
- ② The packet arriving through port 2 (in_port=2) matches this rule and has the priority of 2. The rule performs the action to drop the packet (actions=drop).

- ③ The packet having source MAC address of **FA:16:3F:30:A6:Af** (dl_src=fa:16:3f:30:a6:af) matches this rule and has the priority of 2. The rule performs the action to output the packet through port 2 (actions=outputs:2).
- ④ The rule has the priority of 1. The **NORMAL** means the Open vSwitch will act as a traditional layer 2 switch (actions=NORMAL).
- The OpenFlow command line can be used to add rules to OpenFlow switch, using the **ovs-vsctl add-flow** command.

```
[heat-admin@compute1 ~]$ sudo ovs-ofctl add-flow demo-ovsbr0 \
'table=0,priority=4,in_port=1,actions=output:2'
```

Viewing Existing Virtual Bridges on OpenStack Compute Node

The following steps outline the process for viewing the configuration and flow information of existing virtual bridges on the OpenStack compute node.

1. From the OpenStack compute node, view the Linux bridges and their associated ports using the **brctl show**. The Linux bridge attached to the instance running on the compute node is **qbr**.
2. View the OVS bridges that exist on the compute node using the **ovs-vsctl show**.
The three OVS bridges that commonly exist in an OpenStack deployment are: integration bridge (**br-int**), tunnel bridge (**br-tun**), and external bridge (**br-ex**).
3. On the compute node, view the configuration files associated with the bridges in the **/etc/sysconfig/network-scripts** directory. View the network interface configuration file under the **/etc/sysconfig/network-scripts** directory that are defined as the ports on the bridge.
4. View the ports associated with the integration, tunneling, and the external bridges using the **ovs-vsctl list-ports** command.
5. For an existing Open vSwitch bridge, use **ovs-ofctl show** to list OpenFlow features and port descriptions.
6. Use the **ovs-ofctl dump-flows** command to print all the flow entries from the OpenFlow switch.



References

ip(8), **ovs-vsctl(8)**, **brctl(8)**, and **ovs-ofctl(8)** man pages

Troubleshooting OpenStack Networking
<https://access.redhat.com/articles/904403>

Guided Exercise: Implementing Virtual Bridges

In this exercise, you will observe packet flow through various elements within the OpenStack network bridges.

Outcomes

You should be able to:

- Create and view a Linux bridge.
- Create and view a OVS bridge.
- View OVS bridges used to transfer data to and from a compute node.
- Review the packet flow within OpenStack network bridges.

Before you begin

Log in to **workstation** as **student** using **student** as the password.

Run the **lab linux-virtual-bridges setup** command. This script verifies that the overcloud nodes are accessible and running the correct OpenStack services. The script also launches an instance on the **compute1** overcloud node.

```
[student@workstation ~]$ lab linux-virtual-bridges setup
```

Steps

1. From **workstation**, log in to the **compute1** overcloud node as the **root** user.

```
[student@workstation ~]$ ssh root@compute1  
[root@compute1 ~]#
```

2. On **compute1**, create a Linux bridge named **lin-br0**.

- 2.1. Create a Linux bridge named **lin-br0**.

```
[root@compute1 ~]# brctl addbr lin-br0
```

- 2.2. View the device state of the **lin-br0** Linux bridge.

```
[root@compute1 ~]# ip address show dev lin-br0  
14: lin-br0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000  
link/ether f2:81:4b:cc:68:f8 brd ff:ff:ff:ff:ff:ff</BROADCAST,MULTICAST>
```

The **lin-br0** device state is in a **DOWN** state, as the bridge is not set to be up.

- 2.3. View the bridge information and attached ports of the **lin-br0** bridge.

```
[root@compute1 ~]# brctl show lin-br0  
bridge name bridge id STP enabled interfaces  
lin-br0 8000.000000000000 no
```

The output shows no attached ports, as no interfaces are attached to the **lin-br0** bridge.

3. On **compute1**, view all the Linux bridges and their associated ports that exist on the node. In the screen output, the **tap39655e76-69** interface is attached to the instance's **eth0** and the **qvb39655e76-69** interface is attached to the integration OVS bridge.

```
[root@compute1 ~]# brctl show
bridge name     bridge id      STP enabled   interfaces
lin-br0         8000.000000000000  no
qbr39655e76-69 8000.ae8a3d6e6c69  no          qvb39655e76-69
                                         tap39655e76-69
```

4. Create and view the **ovs-br0** OVS bridge.

- 4.1. Create the **ovs-br0** OVS bridge.

```
[root@compute1 ~]# ovs-vsctl add-br ovs-br0
```

- 4.2. View the device state of the **ovs-br0** OVS bridge.

```
[root@compute1 ~]# ip addr show dev ovs-br0
15: ovs-br0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether 72:49:68:74:a3:4d brd ff:ff:ff:ff:ff:ff</BROADCAST,MULTICAST>
```

The **ovs-br0** device state is in a **DOWN** state, as the bridge is not set to be up.

- 4.3. View the bridge information and attached ports of the **ovs-br0** bridge.

```
[root@compute1 ~]# ovs-vsctl show | grep -1 ovs-br0
...output omitted...
    Bridge "ovs-br0"
        Port "ovs-br0"
            Interface "ovs-br0"
                type: internal
[root@compute1 ~]# ovs-vsctl list-ports ovs-br0
[root@compute1 ~]#
```

The output shows no attached ports, as no interfaces are attached to the **ovs-br0** bridge.

5. On **compute1**, view all the bridges and list their respective ports.

- 5.1. View the OVS bridges that exist on the **compute1** node.

```
[root@compute1 ~]# ovs-vsctl list-br
br-ex
br-int
br-trunk
br-tun
ovs-br0
```

- 5.2. View the ports associated with the **br-int**, **br-tun**, and **br-ex** bridges.

```
[root@compute1 ~]# ovs-vsctl list-ports br-int
int-br-ex
patch-tun
qvo3965e76-69
[root@compute1 ~]# ovs-vsctl list-ports br-tun
patch-int
vxlan-ac180201
[root@compute1 ~]# ovs-vsctl list-ports br-ex
eth2
phy-br-ex
```

6. Review the OpenFlow information of the **br-ex** bridge.

- 6.1. Determine the port ID of the **phy-br-ex** port using the **ovs-ofctl** command. The output lists the ports in the **br-ex** bridge. In the screen below, the **phy-br-ex** port has a value of **2**.

```
[root@compute1 ~]# ovs-ofctl show br-ex
OFPT_FEATURES_REPLY (xid=0x2): dpid:000052540002fa0c
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src
        mod_dl_dst mod_nw_src mod_nw_dst mod_nw_tos mod_tp_src mod_tp_dst
1(eth2): addr:52:54:00:02:fa:0c
        config:    0
        state:     0
        speed: 0 Mbps now, 0 Mbps max
2(phy-br-ex): addr:3a:83:d1:8b:e0:62
        config:    0
        state:     0
        speed: 0 Mbps now, 0 Mbps max
LOCAL(br-ex): addr:52:54:00:02:fa:0c
        config:    0
        state:     0
        speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
```

6.2. Dump the flows for the external bridge, **br-ex**.

```
[root@compute1 ~]# ovs-ofctl dump-flows br-ex
NXST_FLOW reply (xid=0x4):
cookie=0xaf...f7fc, duration=13382.535s, table=0, n_packets=0, n_bytes=0,
idle_age=13473, priority=2,in_port=2 actions=resubmit(,1)
cookie=0xaf...f7fc, duration=13473.810s, table=0, n_packets=63, n_bytes=3521,
idle_age=13382, priority=0 actions=NORMAL
cookie=0xaf...f7fc, duration=13382.535s, table=0, n_packets=15307,
n_bytes=993026, idle_age=0, priority=1 actions=resubmit(,3)
cookie=0xaf...f7fc, duration=13382.534s, table=1, n_packets=0, n_bytes=0,
idle_age=13382, priority=0 actions=resubmit(,2)
cookie=0xaf...f7fc, duration=13382.534s, table=2, n_packets=0, n_bytes=0,
idle_age=13382, priority=2,in_port=2 actions=drop
cookie=0xaf...f7fc, duration=13382.523s, table=3, n_packets=0, n_bytes=0,
idle_age=13382, priority=2,dl_src=fa:16:3f:0d:56:d3 actions=output:2
cookie=0xaf...f7fc, duration=13382.520s, table=3, n_packets=0, n_bytes=0,
idle_age=13382, priority=2,dl_src=fa:16:3f:91:cd:f6 actions=output:2
cookie=0xaf...f7fc, duration=13382.533s, table=3, n_packets=15307,
n_bytes=993026, idle_age=0, priority=1 actions=NORMAL
```

7. When creating virtual networks, the translation between VLAN IDs and tunnel IDs is performed by OpenFlow rules running on the **br-tun** tunnel bridge. Use the **ovs-ofctl dump-flows** command to dump the OpenFlow rules on the **br-tun** tunnel bridge.
- 7.1. Determine the port ID of the associated ports on the **br-tun** OVS bridge. The screen output shows that the port ID associated with **patch-int** is **1** and that the port ID associated to **vxlan** port is **2**. The **patch-int** port connects the tunneling bridge with the integration bridge.

```
[root@compute1 ~]# ovs-ofctl show br-tun
OFPT_FEATURES_REPLY (xid=0x2): dpid:00000ab9d78adf42
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src
mod_dl_dst mod_nw_src mod_nw_dst mod_nw_tos mod_tp_src mod_tp_dst
1(patch-int): addr:66:a0:38:1d:3e:bb
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
2(vxlan-ac180201): addr:aa:62:c0:66:a2:ab
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
LOCAL(br-tun): addr:0a:b9:d7:8a:df:42
    config: PORT_DOWN
    state: LINK_DOWN
    speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
```

- 7.2. Dump the flows for the tunneling bridge, **br-tun**. Review the entries to locate the flow for the outgoing packets from the instance running on the **compute1** node. Locate the rule that handles packets going through the **patch-int** port with the port ID **1**.

The following output shows how OpenFlow handles packets coming in from the patch-int port of the OpenFlow switch (**in_port=1**) with VLAN ID 1 (**dl_vlan=1**). The OpenFlow rules set the tunnel ID to 31 (**strip_vlan, load:0x1f->NXM_NX_TUN_ID[]**) before sending it out the VXLAN tunnel with the port ID 2 (**output:2**).

The VLAN ID and tunnel ID returned in the output may vary.

```
[root@compute1 ~]# ovs-ofctl dump-flows br-tun
NXST_FLOW reply (xid=0x4):
cookie=0x96...74be, duration=10949.921s, table=0, n_packets=847, n_bytes=83762,
idle_age=51, priority=1,in_port=1 actions=resubmit(,1)
...output omitted...
cookie=0x96...74be, duration=10949.919s, table=1, n_packets=847, n_bytes=83762,
idle_age=51, priority=0 actions=resubmit(,2)
...output omitted...
cookie=0x96...74be, duration=10949.930s, table=2, n_packets=833, n_bytes=82130,
idle_age=51, priority=0,dl_dst=00:00:00:00:00:00/01:00:00:00:00:00
actions=resubmit(,20)
cookie=0x96...74be, duration=10949.929s, table=2, n_packets=14, n_bytes=1632,
idle_age=10773, priority=0,dl_dst=01:00:00:00:00:00/01:00:00:00:00:00
actions=resubmit(,22)
...output omitted...
cookie=0x96...74be, duration=10800.445s, table=20, n_packets=0,
n_bytes=0, idle_age=10800, priority=2,dl_vlan=1,dl_dst=fa:16:3e:13:4c:e9
actions=strip_vlan,load:0x1f->NXM_NX_TUN_ID[],output:2
```

```

cookie=0x96...74be, duration=10800.443s, table=20, n_packets=833,
n_bytes=82130, idle_age=51, priority=2,dl_vlan=1,dl_dst=fa:16:3e:4a:67:b8
actions=strip_vlan,load:0x1f->NXM_NX_TUN_ID[],output:2
cookie=0x96...74be, duration=56.531s, table=20, n_packets=0,
n_bytes=0, hard_timeout=300, idle_age=56, hard_age=51,
priority=1,vlan_tci=0x0001/0x0fff,dl_dst=fa:16:3e:4a:67:b8 actions=load:0-
>NXM_OF_VLAN_TCI[],load:0x1f->NXM_NX_TUN_ID[],output:2
cookie=0x96...74be, duration=10949.925s, table=20, n_packets=0, n_bytes=0,
idle_age=10949, priority=0 actions=resubmit(,22)
cookie=0x96...74be, duration=10800.446s, table=22, n_packets=11, n_bytes=1374,
idle_age=10773, priority=1,dl_vlan=1 actions=strip_vlan,load:0x1f-
>NXM_NX_TUN_ID[],output:2
cookie=0x96...74be, duration=10949.925s, table=22, n_packets=3, n_bytes=258,
idle_age=10801, priority=0 actions=drop

```

- 7.3. Dump the flows for the tunnel bridge, **br-tun**. Review the entries to locate the flow for the incoming packets to the instance running on the **compute1** node. Locate the rule that handles packets going through the **vxlan** port with the port ID **2**.

The following output shows how the OpenFlow switch handles packets coming in from VXLAN port (**in_port=2**) with tunnel ID 31 (**tun_id=0x1f**). The OpenFlow rule modifies the VLAN ID to 1 (**action=mod_vlan_vid:1**) before sending it out using the **patch-int** interface with the port ID 1 (**output:1**).

The VLAN ID and tunnel ID returned in the output may vary.

```

[root@compute1 ~]# ovs-ofctl dump-flows br-tun
NXST_FLOW reply (xid=0x4):
cookie=0x96...74be, duration=10800.449s, table=0, n_packets=775, n_bytes=79138,
idle_age=51, priority=1,in_port=2 actions=resubmit(,4)
...output omitted...
cookie=0x96...74be, duration=10801.302s, table=4, n_packets=775, n_bytes=79138,
idle_age=51, priority=1,tun_id=0x1f actions=mod_vlan_vid:1,resubmit(,9)
cookie=0x96...74be, duration=10949.928s, table=4, n_packets=0, n_bytes=0,
idle_age=10949, priority=0 actions=drop
cookie=0x96...74be, duration=10949.927s, table=6, n_packets=0, n_bytes=0,
idle_age=10949, priority=0 actions=drop
cookie=0x96...74be, duration=10949.903s, table=9, n_packets=0, n_bytes=0,
idle_age=10949, priority=1,dl_src=fa:16:3f:0d:56:d3 actions=output:1
cookie=0x96...74be, duration=10949.898s, table=9, n_packets=0, n_bytes=0,
idle_age=10949, priority=1,dl_src=fa:16:3f:91:cd:f6 actions=output:1
cookie=0x96...74be, duration=10949.920s, table=9, n_packets=775, n_bytes=79138,
idle_age=51, priority=0 actions=resubmit(,10)
cookie=0x96...74be, duration=10949.926s, table=10,
n_packets=775, n_bytes=79138, idle_age=51, priority=1
actions=learn(table=20,hard_timeout=300,priority=1,
cookie=0x96...74be,NXM_OF_VLAN_TCI[0..11],
NXM_OF_ETH_DST[]>NXM_OF_ETH_SRC[],load:0-
>NXM_OF_VLAN_TCI[],load:NXM_NX_TUN_ID[]-
>NXM_NX_TUN_ID[],output:0XM_OF_IN_PORT[],output:1
...output omitted...

```

8. Incoming packets going to instances from the external network first reach the **eth2** network device. They are then forwarded to the **br-ex** bridge. From the **br-ex** bridge, packets are moved to the integration bridge, **br-int**. Use the **ovs-ofctl** command to view the flow information on the **br-int** integration bridge.

- 8.1. Determine the port ID of the **int-br-ex** port, using the **ovs-ofctl** command. The output lists the ports in the **br-int** bridge. In the screen below, the **int-br-ex** port has port ID **1**, the **patch-tun** port has port ID **2**, and the **qvo39655e76-69** port has port ID **3**.

```
[root@compute1 ~]# ovs-ofctl show br-int
OFPT_FEATURES_REPLY (xid=0x2): dpid:000082733d24a946
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src
mod_dl_dst mod_nw_src mod_nw_dst mod_nw_tos mod_tp_src mod_tp_dst
1(int-br-ex): addr:56:7d:9d:fc:0e:6d
    config:      0
    state:       0
    speed: 0 Mbps now, 0 Mbps max
2(patch-tun): addr:6e:3e:cb:6a:0f:a5
    config:      0
    state:       0
    speed: 0 Mbps now, 0 Mbps max
3(qvo39655e76-69): addr:ce:31:a4:73:6c:ff
    config:      0
    state:       0
    current:    10GB-FD COPPER
    speed: 10000 Mbps now, 0 Mbps max
...output omitted...
LOCAL(br-int): addr:82:73:3d:24:a9:46
    config:      PORT_DOWN
    state:       LINK_DOWN
    speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
```

- 8.2. Run the **ovs-ofctl dump-flows br-int** command to view the flows in the **br-int** integration bridge.

The following output shows the flow rules on the bridge, which forwards the packets received on **qvo39655e76-69 (in_port=3)** interface to the instance (**dl_src=fa:16:3e:93:cf:b0**), where **fa:16:3e:93:cf:b0** is the MAC address of the instance.

```
[root@compute1 ~]# ovs-ofctl dump-flows br-int
NXST_FLOW reply (xid=0x4):
...output omitted...
cookie=0x8c...c500, duration=12645.579s, table=25, n_packets=847,
n_bytes=83678, idle_age=677, priority=2,in_port=3,dl_src=fa:16:3e:93:cf:b0
actions=NORMAL
...output omitted...
```

9. Log out of **compute1** node.

```
[root@compute1 ~]# logout
[student@workstation ~]$
```

Cleanup

From **workstation**, run the **lab linux-virtual-bridges cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab linux-virtual-bridges cleanup
```

This concludes the guided exercise.

Managing Virtual Networking Devices

Objectives

After completing this section, students should be able to:

- Manage TAP devices.
- Manage veth pairs.
- Manage OVS patch pairs.

Virtual Devices

A virtual networking device is similar to a physical network interface: it can have an IP address and a MAC address. However, no hardware is associated with it. Sending a packet to, through, or from it does not send it to a physical interface by default. These virtual interfaces can be connected to external devices through kernel routing, kernel bridging, Open vSwitch bridging, and network namespaces.

Virtual interfaces are created, configured, and assigned to a user by an administrator. The regular user then attaches these virtual interfaces to namespaces or virtual bridges. The virtual interfaces are destroyed only by an administrator.

TAP Devices

TAP devices are virtual Ethernet network devices that simulate a link layer device. TAP devices operate on layer 2 Ethernet frames. Packets sent by an operating system over a TAP device are delivered to a userspace program attached to the virtual device. In OpenStack, TAP devices are created by supported hypervisors, such as KVM and Xen. They implement a virtual network interface card, typically called a VIF, or vNIC. An Ethernet frame sent to a TAP device is received by the guest operating system, enabling the virtual machine to emulate physical machines from a networking perspective.

TAP devices are transient in nature and are created and destroyed by the same program. Once a TAP device is created, an IP address can be assigned, firewall rules can be created, and routes can be defined, unlike a other physical interface.

In the following diagram, a deployed instance on a compute node connects its **eth0** interface to a port on a Linux bridge (**qbr1**) using a TAP device named **tap**.

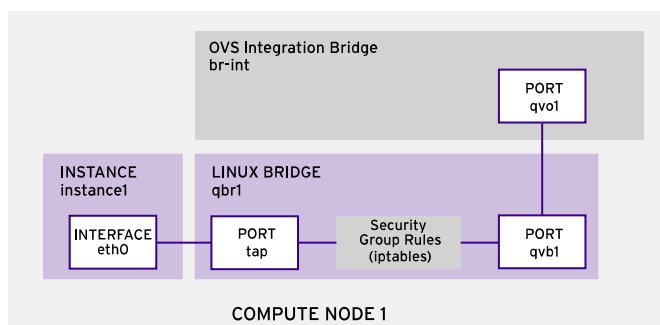


Figure 1.4: A TAP Device connecting an instance to a Linux bridge



Note

OpenStack uses iptables rules on the TAP device to implement security groups. Open vSwitch was not originally compatible with Netfilter rules applied directly on TAP devices. Instead, a Linux bridge implemented the iptables rules. Newer Open vSwitch versions now support firewall flow rules, deprecating the use of Linux bridges.

Creating TAP Devices

TAP devices can be created with the **ip tuntap** command provided by the *iproute* package.

- Create a TAP device named **demo-tap0** using **ip tuntap**. TAP devices created with the **ip tuntap** are not persistent across reboots.

```
[root@demo ~]# ip tuntap add dev demo-tap0 mode tap
```

- A TAP device is added to an OVS bridge as an interface using the **ovs-vsctl add-port** command.

```
[root@demo ~]# ovs-vsctl add-port demo-ovsbr0 demo-tap0 \
-- set interface demo-tap0 type=internal
```

A TAP device can be added to a Linux bridge using the **brctl addif** command.

```
[root@demo ~]# brctl addif demo-linbr0 demo-tap0
```

- Assign an IP address to the TAP interface.

```
[root@demo ~]# ip addr add 172.25.250.150/24 dev demo-tap0
```

- Activate the TAP device using **ip link set** command.

```
[root@demo ~]# ip link set demo-tap0 up
```

- Verify that the TAP device is connected properly using **ip addr show**. In the screen output, the **demo-tap0** TAP device is attached as an interface to a Linux bridge.

```
[root@demo ~]# ip addr show demo-tap0
45: demo-tap0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast master
    demo-linbr0 state DOWN mode DEFAULT qlen 1000
        link/ether 1e:d9:a2:9a:14:37 brd ff:ff:ff:ff:ff:ff
        inet 172.25.250.150/24 scope global demo-tap0
            valid_lft forever preferred_lft forever
```

- To delete a TAP device, remove the TAP port from the bridge if it is attached as an interface to a bridge. Delete the TAP device using the **ip tuntap del** command.

```
[root@demo ~]# ip tuntap del dev demo-tap0 mode tap
```

Managing TAP Devices

The following steps outline the process for creating and managing TAP devices.

1. As an administrator, ensure that the *iproute* package is installed.
2. Create a TAP device using the **ip tuntap** command.
3. Ensure that the device is successfully created.
4. Activate and add an IP address to the TAP device using the **ip** command.
5. Add the TAP device to a Linux bridge using the **brctl** command or to an OVS bridge using the **ovs-vsctl** command.
6. Verify the connectivity by using **ping** to the IP address assigned to the TAP device.
7. Optionally, to delete the TAP device, use **ip tuntap del** command.

veth Devices

A veth virtual network device can connect to other virtual network devices. They are essentially *virtual patch cables*. For example, two bridges are connected together by adding one of the veth device interfaces. OpenStack uses veth pairs to implement connections between a Linux bridge and an OVS bridge. The veth pair is provisioned by OpenStack when an instance is launched to connect the tenant network to the infrastructure network.

Communication using veth pair happens either as a point-to-point direct link by assigning IPs to both ends and routing on the host, or by the use of bridging. An Ethernet frame sent to one end of a veth pair is received by the other end of a veth pair.

Creating veth Devices

Create a veth pair using the **ip link add** command. While creating a veth pair, the **type** option is required to be set as **veth**. The administrators, when creating veth pair device, are required to provide the name of the first port and the name of the peer port.

In the screen below, the **demo-veth0** is the name of the first port and **demo-veth1** is the name of the peer port.

```
[root@demo ~]# ip link add demo-veth0 type veth peer name demo-veth1
[root@demo ~]# ip link set up dev demo-veth1
[root@demo ~]# ip link set down dev demo-veth0
[root@demo ~]# ip link show
...output omitted...
20: demo-veth1@demo-veth0: <NO-CARRIER,BROADCAST,MULTICAST,UP,M-DOWN> mtu 1500 qdisc noop state LOWERLAYERDOWN mode DEFAULT qlen 1000
    link/ether e2:36:13:75:43:0d brd ff:ff:ff:ff:ff:ff
21: demo-veth0@demo-veth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT qlen 1000
    link/ether aa:a7:f8:d2:a1:60 brd ff:ff:ff:ff:ff:ff
```

As data link layer devices, a veth pair implements carrier detection. In the screen below, the endpoint interface indicates **NO-CARRIER** because its peer endpoint is in a **DOWN** state. The **demo-veth1** interface will indicate **NO-CARRIER** until the other endpoint interface is up. When both endpoints are **DOWN**, the veth pair endpoints will only indicate that they are down.

```
[root@demo ~]# ip link set up dev demo-veth0
```

```
[root@demo ~]# ip link show
...output omitted...
20: demo-veth1@demo-veth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noop
    state UP mode DEFAULT qlen 1000
        link/ether e2:36:13:75:43:0d brd ff:ff:ff:ff:ff:ff
21: demo-veth0@demo-veth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noop
    state UP mode DEFAULT qlen 1000
        link/ether aa:a7:f8:d2:a1:60 brd ff:ff:ff:ff:ff:ff
```

veth Devices in OpenStack

When an instance is launched on OpenStack, the Linux bridge **qbr** is connected to the **br-int** OVS integration bridge using a veth pair device. This veth pair device has one end connected to the **qbr** Linux bridge using the interface **qvb**. The other end of the veth pair is connected to the **br-int** OVS bridge using the interface **qvo**.

```
[root@demo ~]# brctl show qbr3103c297-96
bridge name      bridge id      STP enabled interfaces
qbr3103c297-96   8000.12aa29708202 no      qvb3103c297-96
                  tap3103c297-96
[root@demo ~]# ovs-vsctl list-ports br-int
...output omitted...
qvo3103c297-96
[root@demo ~]# ip link show
...output omitted...
18: qvo3103c297-96@qvb3103c297-96: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1446
    qdisc noqueue master ovs-system state UP mode DEFAULT qlen 1000
        link/ether fe:38:ce:37:a1:ee brd ff:ff:ff:ff:ff:ff
19: qvb3103c297-96@qvo3103c297-96: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1446
    qdisc noqueue master qbr3103c297-96 state UP mode DEFAULT qlen 1000
        link/ether 12:aa:29:70:82:02 brd ff:ff:ff:ff:ff:ff
```

Managing veth Pair Virtual Devices

Linux TAP interfaces created with **ip tuntap** cannot be used to attach network namespaces to Linux bridges or OVS bridges. A veth pair device connects two Linux network namespaces. The following steps outline the process for creating and managing veth pair devices to connect two namespaces using an OVS bridge or Linux bridge.

1. Create a veth pair device using the **ip link add** command.
2. Connect one end of the veth pair device to a Linux network namespace using the **ip link set** command. Then connect the other end of the veth pair device to an OVS bridge using the **ovs-vsctl** command or to a Linux bridge using the **brctl** command.
Another veth pair device is connected in a similar way to another Linux network namespace.
3. Bring the links up for the veth pair devices using the **ip link set** command.
4. Add an IP address to the end of the veth pair devices attached to the network namespaces using the **ip** command.
5. Verify the connectivity by using **ping** using the IP address assigned to the other veth pair.
6. Optionally, to delete a veth pair device, use **ip link del** command. Deleting either end of the device causes the peer interface to also be deleted.

OVS Patch Pair Devices

In earlier OpenStack network implementations, veth pairs were used for connecting OVS bridges except when VXLAN or GRE tunnels were used. All recent implementations use OVS patch ports for flat and VLAN deployments.

OVS bridges use OpenFlow rules specified in flow tables to determine packet routing. To process a packet arriving on an OVS port, the Open vSwitch kernel module uses packet metadata to find a rule match in the flow table. If found, the OpenFlow rule action is performed. If a match is not found, the packet metadata is used to create a new rule in the flow table in kernel space. Subsequent packets received with the same metadata can then be processed in kernel space, avoiding context switching to userspace. Using OVS patch pairs over veth pairs allows higher performance because of reduced context switching between userspace and kernel space. Linux bridge ports and veth pairs can also interconnect OVS bridges, but with lower performance due to the need to context switch from the kernel to userspace.

Using OVS patch ports are critical for specialized use cases such as Network Function Virtualization (NFV).

Open vSwitch can operate entirely in userspace without assistance from a kernel module. Using Open vSwitch in userspace requires the TUN/TAP kernel module driver to be loaded. To use **ovs-switchd** in userspace mode, create bridges with **datapath_type=netdev**. To operate using kernel datapaths, set **datapath_type=system**.

```
[root@demo ~]# ovs-vsctl add-br demo-ovsbr0
[root@demo ~]# ovs-vsctl set bridge demo-ovsbr0 datapath_type=netdev
```

Creating OVS Patch Pairs

Create an OVS patch pair using the **ovs-vsctl add-port** command to attach to an existing bridge. Set the interface to use the type **patch** and provide a peer endpoint name.

When using **ovs-vsctl**, multiple commands can be executed as a single command by use of **--**. For example, to create **demo-patch0** as a patch pair with a peer **demo-patch1** and attach **demo-patch0** to the **demo-ovsbr0** bridge, use the following command.

```
[root@demo ~]# ovs-vsctl add-port demo-ovsbr0 demo-patch0 \
-- set interface demo-patch0 type=patch \
-- set interface demo-patch0 options:peer=demo-patch1
```



Note

In the previous example, the patch ports were created, set and attached in a single command. If the commands were run one at a time, an intermediary warning message would have displayed, "Error detected while setting up 'demo-patch0'." This occurs when the endpoint interfaces being created are not backed by physical devices, which is common when creating patch pairs to connect two virtual objects. This message can be safely ignored.

Verify that the patch pair was successfully created.

```
[root@demo ~]# ovs-vsctl show
```

```
Bridge "demo-br0"
  Port "demo-patch0"
    Interface "demo-patch0"
      type: patch
      options: {peer="demo-patch1"}
  Port "demo-br0"
    Interface "demo-br0"
      type: internal
```

Repeat these commands for each bridge to be connected to another bridge.

OVS Patch Pairs in OpenStack

OpenStack networking uses the OVS patch pairs to chain multiple OVS bridges. The integration bridge, **br-int**, also uses OVS patch pair device to connect to **br-tun** tunnel bridge. It uses the **patch-tun** attached as a port on the integration bridge and its peer **patch-int** attached to the tunnel bridge.

```
[root@demo ~]# ovs-vsctl list-ports br-int
...output omitted...
patch-tun
[root@demo ~]# ovs-vsctl list-ports br-tun
...output omitted...
patch-int
```

Similarly, the **br-int** bridge use the OVS patch pair device to connect to the **br-ex** external bridge. It connects the compute and network nodes to the external network and provides external access to instances running on the compute node. The **int-br-ex** OVS port on the **br-int** bridge is connected to **br-ex** using the **phy-br-ex** OVS port.

```
[root@demo ~]# ovs-vsctl list-ports br-int
...output omitted...
int-br-ex
[root@demo ~]# ovs-vsctl list-ports br-ex
...output omitted...
phy-br-ex
```

Managing OVS Patch Pairs Virtual Device

The following steps outline the process for creating and managing OVS patch pair devices.

1. As an administrator, ensure that the *openvswitch* package is installed.
2. Create a port on an existing OVS bridge using the **ovs-vsctl add-port** and set the port type as **internal**. Repeat the process to add a port to the other OVS bridge that is to be connected using the OVS patch pair.
3. Set the port's type to use **patch** using **ovs-vsctl set interface** command.
4. Set the patch ports connected to the first OVS bridge and to the second OVS bridge to be a peer of one another. Use the **ovs-vsctl set interface** with the **options:peer=PORT** option.
5. View the OVS bridge ports to verify that the patch ports are attached.
6. Optionally, to delete the OVS patch pair device, use **ovs-vsctl del-port**.



References

ip(8), ovs-ctl(8), brctl(8) man pages

Further information is available in the chapter on Connect an instance to the physical network in the *Networking Guide* for Red Hat OpenStack Platform 10 at

| <https://access.redhat.com/documentation/en-US/index.html>

| Tun/Tap interface tutorial

| <http://backreference.org/2010/03/26/tuntap-interface-tutorial>

Guided Exercise: Managing Virtual Networking Devices

In this exercise, you will manage TAP devices, veth pairs, OVS patch pairs, and view datapaths and flows through existing virtual devices:

Outcomes

You should be able to:

- Manage TAP devices.
- Manage veth pairs.
- Manage OVS patch pairs.
- View datapaths and flows through existing virtual devices.

Before you begin

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab linux-virtual-devices setup** command. This script verifies that the overcloud nodes are accessible and running the correct OpenStack services. The script also ensures the existence of the **ovs-br0** OVS bridge and the **finance-server1** instance launched in the previous exercise.

```
[student@workstation ~]$ lab linux-virtual-devices setup
```

Steps

1. In this step, you will create and view a TAP device.

- 1.1. From **workstation**, use the **ssh** command to log in to the **compute1** virtual machine as the **heat-admin** user. Become the **root** user.

```
[student@workstation ~]$ ssh heat-admin@compute1
[heat-admin@compute1 ~]# sudo -i
[root@compute1 ~]#
```

- 1.2. Create a TAP device named **tap-dev0**.

```
[root@compute1 ~]# ip tuntap add dev tap-dev0 mode tap
```

- 1.3. View information about the TAP device.

```
[root@compute1 ~]# ethtool -i tap-dev0
driver: tun
version: 1.6
firmware-version:
expansion-rom-version:
bus-info: tap
supports-statistics: no
supports-test: no
```

```
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no
[root@compute1 ~]# ip address show tap-dev0
17: tap-dev0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop master ovs-system state
DOWN qlen 1000
    link/ether 72:91:6c:97:62:ed brd ff:ff:ff:ff:ff:ff
```

2. In this step, you will attach a TAP device to an OVS bridge and test the network connectivity of the TAP device.

- 2.1. Assign the **172.25.250.150/24** IP address to the **tap-dev0** TAP device.

```
[root@compute1 ~]# ip address add 172.25.250.150/24 dev tap-dev0
```

- 2.2. Activate the link on the **ovs-br0** OVS bridge to enable networking.

```
[root@compute1 ~]# ip link set up dev ovs-br0
```

- 2.3. Add the **tap-dev0** device to the **ovs-br0** OVS bridge created in an earlier exercise.

```
[root@compute1 ~]# ovs-vsctl add-port ovs-br0 tap-dev0
```

- 2.4. View the ports on the **ovs-br0** bridge to verify that the TAP device is attached to the OVS bridge.

```
[root@compute1 ~]# ovs-vsctl list-ports ovs-br0
tap-dev0
```

- 2.5. Ping the IP address to verify the network configuration is working.

```
[root@compute1 ~]# ping -c 3 172.25.250.150
PING 172.25.250.150 (172.25.250.150) 56(84) bytes of data.
64 bytes from 172.25.250.150: icmp_seq=1 ttl=64 time=0.027 ms
64 bytes from 172.25.250.150: icmp_seq=2 ttl=64 time=0.032 ms
64 bytes from 172.25.250.150: icmp_seq=3 ttl=64 time=0.032 ms
...output omitted...
```

3. Using the **virsh** command, review the network configuration for the **finance-server1** instance running on the **compute1** node. The **finance-server1** instance was launched by the lab script.

- 3.1. Using the **ip** command, review the interfaces created for the **finance-server1** instance.

```
[root@compute1 ~]# ip link show
...output omitted...
15: ①qbr9c9e4323-db: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue
    state UP mode DEFAULT qlen 1000
        link/ether 4a:1a:d2:b3:57:e5 brd ff:ff:ff:ff:ff:ff
...output omitted...
```

```
18: ②tap9c9e4323-db: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc pfifo_fast master qbr9c9e4323-db state UNKNOWN mode DEFAULT qlen 1000 link/ether fe:16:3e:93:a6:24 brd ff:ff:ff:ff:ff:ff  
...output omitted...
```

- ① The Linux bridge created for the **finance-server1** instance.
- ② The TAP device attached to the instance (eth0).

3.2. Retrieve the domain name and ID of the instance.

```
[root@compute1 ~]# virsh list  
Id   Name           State  
---  
1    instance-00000001      running
```

3.3. List the virtual interfaces attached to the **finance-server1** instance. Use the **ID** or **Name** retrieved in the previous step. Exit from the compute node.

```
[root@compute1 ~]# virsh domiflist instance-00000001  
Interface  Type      Source      Model      MAC  
-----  
tap9c9e4323-db  bridge     qbr9c9e4323-db  virtio     fa:16:3e:93:a6:24  
[root@compute1 ~]# exit  
[heat-admin@compute1 ~]# exit
```

Notice that the network device for the instance is the TAP device displayed by **ip link show** in the earlier step.

4. In this step, you will view the network packets pass through the existing TAP device and the Linux bridge attached to the instance running on **compute1** node.

4.1. From **workstation**, open a new terminal and ping the floating IP address of the **finance-server1**.

```
[student@workstation ~]$ source ~/architect1-finance-rc  
[student@workstation ~(architect1-finance)]$ openstack server show \  
finance-server1 \  
-c addresses -f value  
finance-network1=192.168.1.P, 172.25.250.N
```

```
[student@workstation ~(architect1-finance)]$ ping 172.25.250.N  
PING 172.25.250.N (172.25.250.N) 56(84) bytes of data.  
64 bytes from 172.25.250.N: icmp_seq=1 ttl=63 time=4.53 ms  
...output omitted...
```

To view the network packets pass through the TAP device and the bridge, let this command run while you perform the next step.

- 4.2. On **workstation**, open a new terminal and use the **ssh** command to log in to the **compute1** virtual machine as the **heat-admin** user. Become the **root** user.

```
[student@workstation ~]$ ssh -X heat-admin@compute1
[heat-admin@compute1 ~]# sudo -i
[root@compute1 ~]#
```

- 4.3. Open Wireshark so you can view the network packets coming from the **ping** command.

```
[root@compute1 ~]# XAUTHORITY=/home/heat-admin/.Xauthority wireshark &
```

- 4.4. In the top menu, choose **Capture > Interfaces**. Select the **tap9c9e4323-db** device and press the **Start** button. Watch a few of the ICMP packets come through. You should see something like the following output:

No.	Time	Source	Destination	Protocol	Length
1	0.0000000000	172.25.250.254	192.168.1.1	ICMP	98
	Echo (ping) request	id=0x1129, seq=2767/53002, ttl=63			
2	0.000019918	172.25.250.254	192.168.1.1	ICMP	98
	Echo (ping) request	id=0x1129, seq=2767/53002, ttl=63			
	...output omitted...				

- 4.5. Stop the **ping** command in the previous terminal.

```
[student@workstation ~(architect1-finance)]$ ping 172.25.250.N
...output omitted...
64 bytes from 172.25.250.N: icmp_seq=10 ttl=63 time=0.694 ms
Ctrl+C
...output omitted...
```

5. In your original terminal, create and validate a veth pair on the **compute1** node.

- 5.1. On **compute1**, create a veth pair named **veth-pair0** and **veth-pair1**.

```
[root@compute1 ~]# ip link add \
veth-pair0 type veth peer name veth-pair1
```

- 5.2. Activate the link on both endpoints of the veth pair.

```
[root@compute1 ~]# ip link set veth-pair0 up
[root@compute1 ~]# ip link set veth-pair1 up
```

- 5.3. Using the **ip** command, review the veth pair created.

```
[root@compute1 ~]# ip link show
...output omitted...
21: veth-pair1@veth-pair0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT qlen 1000
    link/ether a6:cb:04:ae:78:23 brd ff:ff:ff:ff:ff:ff
22: veth-pair0@veth-pair1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT qlen 1000
    link/ether 42:93:d2:3b:da:0f brd ff:ff:ff:ff:ff:ff
```

- 5.4. Assign the **172.25.250.151/24** IP address to the **veth-pair0** device.

```
[root@compute1 ~]# ip address add 172.25.250.151/24 dev veth-pair0
```

- 5.5. Ping the IP address to verify that the network configuration is working.

```
[root@compute1 ~]# ping -c 3 172.25.250.151
PING 172.25.250.151 (172.25.250.151) 56(84) bytes of data.
64 bytes from 172.25.250.151: icmp_seq=1 ttl=64 time=0.021 ms
64 bytes from 172.25.250.151: icmp_seq=2 ttl=64 time=0.039 ms
64 bytes from 172.25.250.151: icmp_seq=3 ttl=64 time=0.042 ms
...output omitted...
```

6. Using the **ip** command, review the veth pair configuration for the instance running on the **compute1** node.

- 6.1. Using the **ip** command, review the veth pair interfaces created for the instance.

```
[root@compute1 ~]# ip link show
...output omitted...
16: ①qvo9c9e4323-db@qvb9c9e4323-db: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP>
    mtu 1446 qdisc noqueue master ovs-system state UP mode DEFAULT qlen 1000
    link/ether ba:55:7b:d4:ad:88 brd ff:ff:ff:ff:ff:ff
17: ②qvb9c9e4323-db@qvo9c9e4323-db: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP>
    mtu 1446 qdisc noqueue master qbr9c9e4323-db state UP mode DEFAULT qlen 1000
    link/ether 4a:1a:d2:b3:57:e5 brd ff:ff:ff:ff:ff:ff
...output omitted...
```

- ① The first interface of the veth pair, connected to the Open vSwitch bridge
- ② The second interface of the veth pair, connected to the Linux bridge.

- 6.2. From **workstation**, ping the floating IP address of the **finance-server1**.

```
[student@workstation ~(architect1-finance)]$ ping 172.25.250.N
PING 172.25.250.N (172.25.250.N) 56(84) bytes of data.
64 bytes from 172.25.250.N: icmp_seq=1 ttl=63 time=4.53 ms
...output omitted...
```

Let this command run while you perform the next step to view the network packets pass through the veth pair devices.

- 6.3. On **workstation**, from the terminal where X11 forwarding to the **compute1** node is already established, open **Wireshark** if it is not running, view the network packets coming from the **ping** command.

```
[root@compute1 ~]# XAUTHORITY=/home/heat-admin/.Xauthority wireshark &
```

- 6.4. In the top menu, choose **Capture > Interfaces**. Select the **qvo9c9e4323-db** and **qvb9c9e4323-db** device and press **Start** button. Watch a few of the ICMP packets come through. You should see something like the following output:

No.	Time	Source	Destination	Protocol
Length Info				
1	0.000000000	172.25.250.254	192.168.1.P	ICMP 98
Echo (ping) request	id=0x1129, seq=2767/53002, ttl=63			
2 0.000019918 172.25.250.254 192.168.1.P ICMP 98				
Echo (ping) request	id=0x1129, seq=2767/53002, ttl=63			
...output omitted...				
5 0.006600207 fa:16:3e:17:69:66	fa:16:3e:93:a6:24			ARP 42
Who has 192.168.1.P?	Tell 192.168.1.1			
...output omitted...				

Notice that the same ICMP packets are captured on the TAP device that is associated with the instance. The packets are forwarded from the **br-int** OVS bridge to the **qbr** Linux bridge using the **qvo9c9e4323-db** and **qvb9c9e4323-db** veth pair. These veth pairs connect the **br-int** OVS bridge with **qbr** Linux bridge.

6.5. Stop the **ping** command in the previous terminal.

```
[student@workstation ~(architect1-finance)]$ ping 172.25.250.N
...output omitted...
64 bytes from 172.25.250.N: icmp_seq=10 ttl=63 time=0.694 ms
Ctrl+C
...output omitted...
```

7. In this step, you will create another OVS bridge and TAP device, which you will later connect to the previous OVS bridge using OVS patch pairs.

- 7.1. On **compute1**, create an OVS bridge named **ovs-br1**. Set the bridge **datapath_type=netdev** to use the userspace datapath.

```
[root@compute1 ~]# ovs-vsctl add-br ovs-br1 \
-- set bridge ovs-br1 datapath_type=netdev
```

- 7.2. Create a TAP device named **tap-dev1**.

```
[root@compute1 ~]# ip tuntap add dev tap-dev1 mode tap
```

- 7.3. Assign the **172.25.250.152/24** IP address to the **tap-dev1** TAP device.

```
[root@compute1 ~]# ip address add 172.25.250.152/24 dev tap-dev1
```

- 7.4. Activate the link on the **ovs-br1** OVS bridge.

```
[root@compute1 ~]# ip link set dev ovs-br1 up
```

- 7.5. Add the **tap-dev1** TAP device to the **ovs-br1** OVS bridge.

```
[root@compute1 ~]# ovs-vsctl add-port ovs-br1 tap-dev1
```

8. In this step, you will create a pair of OVS patch ports and connect the two OVS bridges.

- 8.1. Create a port on **ovs-br0** named **ovs-patch0**. Set the port type as **internal**.

```
[root@compute1 ~]# ovs-vsctl add-port ovs-br0 ovs-patch0 \
-- set interface ovs-patch0 type=internal
```

- 8.2. Set the **ovs-patch0** port as a patch port.

```
[root@compute1 ~]# ovs-vsctl set interface ovs-patch0 type=patch
```

- 8.3. Set the **ovs-patch0** patch port as a peer to **ovs-patch1**

```
[root@compute1 ~]# ovs-vsctl set interface \
ovs-patch0 options:peer=ovs-patch1
```

- 8.4. Create a port on **ovs-br1** named **ovs-patch1**. Set the port type as **internal**.

```
[root@compute1 ~]# ovs-vsctl add-port ovs-br1 ovs-patch1 \
-- set interface ovs-patch1 type=internal
```

- 8.5. Set the **ovs-patch1** port as a patch port.

```
[root@compute1 ~]# ovs-vsctl set interface ovs-patch1 type=patch
```

- 8.6. Set the **ovs-patch1** patch port as a peer to **ovs-patch0**

```
[root@compute1 ~]# ovs-vsctl set interface \
ovs-patch1 options:peer=ovs-patch0
```

- 8.7. View the OVS bridge ports to verify that the patch ports are attached.

```
[root@compute1 ~]# ovs-vsctl show
...output omitted...
Bridge "ovs-br1"
  Port "ovs-patch1"
    Interface "ovs-patch1"
      type: patch
      options: {peer="ovs-patch0"}
  Port "ovs-br1"
    Interface "ovs-br1"
      type: internal
  Port "tap-dev1"
    Interface "tap-dev1"
Bridge "ovs-br0"
  Port "ovs-br0"
    Interface "ovs-br0"
      type: internal
  Port "ovs-patch0"
    Interface "ovs-patch0"
      type: patch
      options: {peer="ovs-patch1"}
  Port "tap-dev0"
    Interface "tap-dev0"
```

```
...output omitted...
[root@compute1 ~]# ovs-vsctl list-ports ovs-br0
ovs-patch0
tap-dev0
[root@compute1 ~]# ovs-vsctl list-ports ovs-br1
ovs-patch1
tap-dev1
```

9. Log out from all the opened remote sessions to **compute1** node.

```
[root@compute1 ~]# exit
[heat-admin@compute1 ~]$ exit
[student@workstation ~]$
```

Cleanup

From **workstation**, run the **lab linux-virtual-devices cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab linux-virtual-devices cleanup
```

This concludes the guided exercise.

Implementing Network Namespaces

Objectives

After completing this section, students should be able to:

- Define a network namespace.
- View firewall NAT tables in a network namespace.
- Manage VETH pairs.

Linux Namespaces

Namespaces are an integral feature of Red Hat Enterprise Linux that isolate system resources used by select processes. There are multiple namespace types, each named for the resource being virtualized: process identifiers (PIDs), user identifiers (UIDs), and user privileges, network stacks, mount points, control groups (cgroups), interprocess communication (ipc), and others. This section focuses on OpenStack Network Service network namespaces.

A namespace is described as an abstract kernel environment allowing unconflicted use and restriction of resource identifiers. Each namespace contains uniquely named processes and resources that cannot be seen or reached by processes from other namespaces. A primary characteristic of namespaces is that no coordination is required between namespaces when resource identifiers are created; the use of identical names in different namespaces is never a conflict.

Red Hat Enterprise Linux starts a default global namespace at boot, containing all host system resources and identifiers. Additional namespaces are initialized, as needed, to isolate system resources for new virtualized structures when created. For example, namespaces are the core framework of virtual machines and containers. Each virtual machine recognizes its own set of users, network interfaces and processes by using resource identifiers unique to that virtual machine. Other virtual machines on the same virtualization host may use the same resource identifiers for their users, network interfaces and processes, but no resource identifier conflicts occur.

Network Namespaces

Virtual machines and containers are one example where namespaces are used in Red Hat OpenStack Platform. To facilitate multitenancy, namespaces virtualize each tenant project's network services and resource entities, allowing each project's administrators to declare any network addressing and interface naming scheme they choose, without conflicting with other projects. Projects may use overlapping IPv4 addresses scopes, since each project's subnets and addresses are isolated and use Network Address Translation (NAT) to route packets to external networks and subnets.

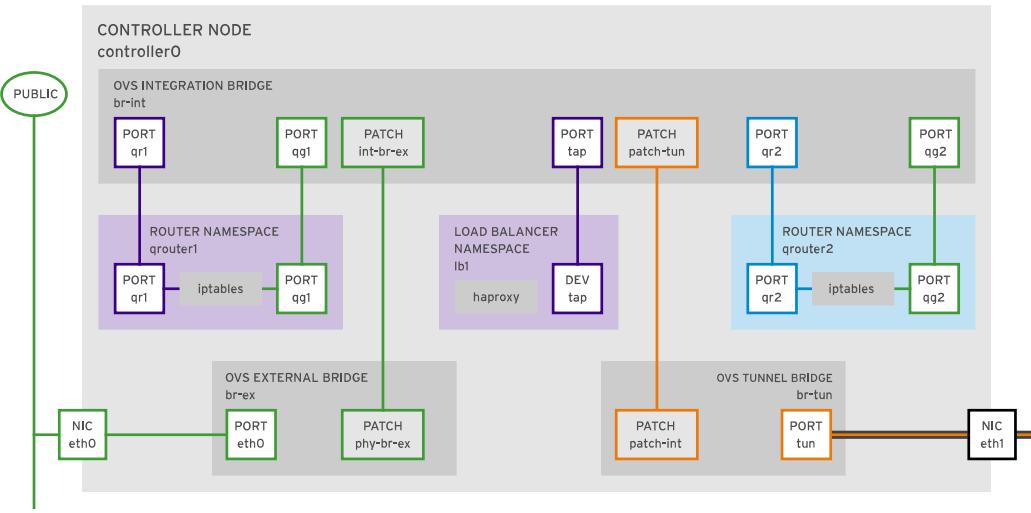


Figure 1.5: Network namespaces on a controller node

An OpenStack Networking L3 agent (neutron-l3-agent) uses a network namespace to isolate each project's layer 3 routers, which direct traffic and provide gateway services for the project's layer 2 networks. The router namespace contains iptables rules for the project network's routing needs, including NAT rules to translate a project's fixed, local IPv4 addresses to external floating IP addresses, and other IPv4 and IPv6 rules to allow and restrict network traffic.

An OpenStack Networking Load Balancing agent (neutron-lbaas-agent) provisions LBaaS routers using the default HAProxy network namespace. The namespace route table distributes incoming requests among pooled instances according to a configured policy. A newer approach is the Octavia project, deploying load balancers as VMs instead of HAProxy namespaces.

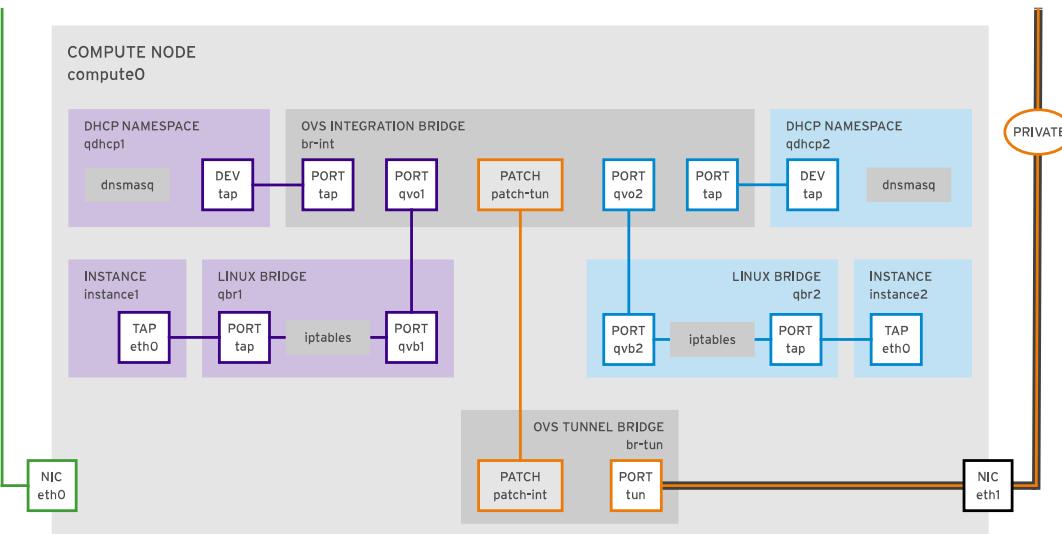


Figure 1.6: Network namespaces on a compute node

An OpenStack Networking DHCP agent (neutron-dhcp-agent) manages the creation of network namespaces for each project subnet that enabled DHCP IP allocation. Each namespace runs a dnsmasq process, which allocates IP addresses to virtual machines deployed on the corresponding subnet. A subnet has DHCP allocation enabled by default if the DHCP agent is enabled and running when the subnet is created. The current OpenStack default is to place

DHCP namespaces on compute nodes, as shown here. Red Hat OpenStack Platform installations performed by Director place DHCP namespaces on the controller node, or on separate network nodes when specified through deployment templates.



Note

Projects may also use provider networks, in which internal project networks and subnets are configured directly on the external physical network using addresses taken from the external network address scope. Provider networks, because they are external networks, do not use or need Neutron L3 Agent routers and namespaces.

There is a similar IP technology called virtual routing and forwarding (VRF), which allows multiple instances of a routing table to exist on the same physical or virtual router. VRF functionality in legacy routers is conceptually similar to that provided by network namespaces in Software Defined Networking (SDN).

Managing Namespaces

Each resource namespace has a unique name beginning with namespace type, followed by the unique ID of the relevant resource being managed:

grouter-xxxxxxxx

A route table namespace uses the managed legacy router's ID, as viewed with an **openstack router list** command. Router namespaces run iptables.

qdhcp-xxxxxxxx

A DHCP server namespace uses the managed subnet's network ID, as viewed with an **openstack network list** command. DHCP namespaces run dnsmasq.

qlbaas-xxxxxxxx

An LBaaS proxy namespace uses the managed load balancing pool's ID, as viewed with a **neutron lb-pool-list** command. LBaaSv2 namespaces run HAProxy.

snat-xxxxxxxx

AA source NAT namespace uses the managed DVR router's ID, as viewed with an **openstack router list** command. SNAT namespaces run iptables.

To locate nodes containing namespaces, list the network agents to view their host parameter.

-c "Agent Type"	-c Host	-c State
L3 agent	compute0.overcloud.example.com	UP
Open vSwitch agent	compute0.overcloud.example.com	UP
Metadata agent	controller0.overcloud.example.com	UP
Loadbalancerv2 agent	controller0.overcloud.example.com	UP
Metadata agent	compute0.overcloud.example.com	UP
Open vSwitch agent	controller0.overcloud.example.com	UP
Metadata agent	compute1.overcloud.example.com	UP
L3 agent	compute1.overcloud.example.com	UP
L3 agent	controller0.overcloud.example.com	UP
Open vSwitch agent	compute1.overcloud.example.com	UP
DHCP agent	controller0.overcloud.example.com	UP

A Linux process runs in a specific network namespace, usually the same namespace inherited from the process's parent. The processes running inside a namespace cannot be seen from any other namespace. A logged in user, including root, exists in the general Linux namespace and cannot list or interact with other namespace processes unless a privileged request is made to pass a command into another namespace.

Running Commands In Another Namespace

The **ip** command is used to request visibility into other namespaces. The syntax used will depend on whether the requested command is an **ip** subcommand or if it is any command other than **ip**.

For **ip** subcommands, use the **-n** option to specify the namespace in which to run the subcommand. In the following example, after locating the desired namespaces, an **ip address** command was run in the selected namespace:

```
[demo@controller0 ~]$ ip netns list
qdhcp-6fce40cf-7da3-4242-938a-3c7043dc2b6a
snat-779631a4-1dea-4fb6-8db4-181ebd0d9248
qrouter-b07cc523-105d-418b-ac2c-cb620a2ada49
qrouter-779631a4-1dea-4fb6-8db4-181ebd0d9248
[demo@controller0 ~]$ sudo ip -n qdhcp-6fce40cf-7da3-4242-938a-3c7043dc2b6a address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
53: tapfc9df83e-a7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1496 qdisc noqueue state UNKNOWN qlen 1000
    link/ether fa:16:3e:c5:01:2a brd ff:ff:ff:ff:ff:ff
    inet 192.168.4.2/24 brd 192.168.4.255 scope global tapfc9df83e-a7
        valid_lft forever preferred_lft forever
```

For any command other than an **ip** subcommand, use the **ip netns exec** command to request execution within the namespace. In the following example, the **iptables** command is executed within the selected namespace:

```
[demo@controller0 ~]$ sudo ip exec qrouter-b07cc523-105d-418b-ac2c-cb620a2ada49 iptables
24 1632 DNAT all -- * * 0.0.0.0/0      172.25.250.28  to:192.168.0.11
  8  672 SNAT all -- * * 192.168.0.11  0.0.0.0/0      to:172.25.250.28
... output omitted ...
```

Listing the namespaces did not require root access, but running commands from the default global namespace into another namespace does. Note above that **sudo** was placed before the **ip netns exec**, not the command being requested from inside the namespace.

Network namespaces isolate only network resources, not processes or users. Running **ps ax**, a process command, inside a network namespace returns all processes running in the *global* namespace, but running **ip route**, a network command, returns the rules present only in the selected namespace.

```
[demo@controller0 ~]$ sudo ip -n qrouting-b07cc523-105d-418b-ac2c-cb620a2ada49 route
default via 172.25.250.254 dev qg-589c787a-b9
172.25.250.0/24 dev qg-589c787a-b9 proto kernel scope link src 172.25.250.180
```

```
192.168.1.0/24 dev qr-80819b82-a7 proto kernel scope link src 192.168.1.1
```

Implementing Network Namespaces

1. Create two namespaces and verify that they exist in the namespace list.
2. Create a veth pair. Verify that they exist in the global namespace.
3. Observe that interfaces in veth pairs exhibit normal layer 2 behavior.
4. Connect each interface into its assigned namespace, one at a time.
5. Confirm that the interfaces no longer exist in the global namespace, but do exist in their assigned namespace.
6. Assign an IP address to each interface endpoint in their namespace.
7. Bring both link interfaces up.
8. Ping in each direction to verify that each endpoint communicates with its peer.
9. When done, delete the test namespaces.

References

Further information is available in the *Networking Guide* for Red Hat OpenStack Platform at

<https://access.redhat.com/documentation/en/red-hat-openstack-platform/>

Guided Exercise: Implementing Network Namespaces

In this exercise, you will manage network namespaces, and view firewall rules and NAT routing tables from within the router namespace.

Outcomes

You should be able to:

- Manage a network namespace.
- Show iptables rules in a network namespace.
- Assign a floating IP Address to an instance.

Before you begin

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab provisioning-namespaces setup** command. The script verifies that the overcloud nodes are accessible and are running the correct OpenStack services, and creates the instance used during this guided exercise.

```
[student@workstation ~]$ lab provisioning-namespaces setup
```

Steps

1. Source the **developer1-finance** credentials. Create a network called **test-network1**. Create a subnet called **test-subnet1**. Attach **finance-router1** to **test-subnet1**.
 - 1.1. Source the **developer1-finance-rc** credential file.

```
[student@workstation ~]$ source ~/developer1-finance-rc
```

- 1.2. Create a network called **test-network1**.

```
[student@workstation ~(developer1-finance)]$ openstack network create \
test-network1
+-----+-----+
| Field | Value |
+-----+-----+
...output omitted...
| id | bc610b6e-a1ba-4451-921e-791169a441d5 |
| ipv4_address_scope | None |
| ipv6_address_scope | None |
| mtu | 1446 |
| name | test-network1 |
...output omitted...
```

- 1.3. Create the subnet **test-subnet1**. The subnet range is **192.168.3.0/24**, use **172.25.250.254** for the DNS server and ensure that **dhcp** is enabled.

```
[student@workstation ~(developer1-finance)]$ openstack subnet create \
```

```
--network test-network1 \
--subnet-range=192.168.3.0/24 \
--dns-nameserver=172.25.250.254 \
--dhcp test-subnet1
+-----+-----+
| Field | Value |
+-----+-----+
| allocation_pools | 192.168.3.2-192.168.3.254
| cidr | 192.168.3.0/24
| created_at | 2017-10-04T07:23:21Z
| description |
| dns_nameservers | 172.25.250.254
| enable_dhcp | True
| gateway_ip | 192.168.3.1
| headers |
| host_routes |
| id | edab030e-23e6-4956-9987-a22aa51c68d8
...output omitted...
```

- 1.4. Using the **openstack router add** command, attach **finance-router1** to **test-subnet1**.

```
[student@workstation ~(developer1-finance)]$ openstack router add \
subnet finance-router1 test-subnet1
```

2. Open a second terminal window. Using SSH, log in to **controller0** and view the corresponding namespace for **test-subnet1**. The **qdhcp** namespace uses the network ID of **test-network1**. Use the **ip netns** command to show the interfaces within the Linux namespace.

```
[student@workstation ~(developer1-finance)]$ ssh heat-admin@controller0
Last login: Tue Oct  3 13:21:08 2017 from 172.25.250.254
[heat-admin@controller0 ~]$ ip netns
...output omitted...
qdhcp-bc610b6e-a1ba-4451-921e-791169a441d5
...output omitted...
[heat-admin@controller0 ~]$ sudo ip -n \
qdhcp-bc610b6e-a1ba-4451-921e-791169a441d5 address
...output omitted...
40: tap1ba928c4-86: >BROADCAST,MULTICAST,UP,LOWER_UP< mtu 1446 qdisc noqueue state UNKNOWN qlen 1000
    link/ether fa:16:3e:21:96:aa brd ff:ff:ff:ff:ff:ff
    inet 192.168.3.2/24 brd 192.168.3.255 scope global tap1ba928c4-86
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe21:96aa/64 scope link
        valid_lft forever preferred_lft forever
...output omitted...
```

3. Back on **workstation**, determine the ID of **finance-router1**. In the router namespace, execute the **iptables** command to show the firewall NAT table and rules before and after a floating IP has been configured for **finance-server1**.
- 3.1. In the first terminal, use the **openstack router list** command to determine the ID of **finance-router1**.

```
[student@workstation ~(developer1-finance)]$ openstack router list -c ID -c Name
+-----+-----+
```

ID	Name
152de7f4-ea63-4b12-b50a-1861f15a2840	finance-dvr-router1
7ee5fe29-ddf7-424d-aea9-f5e04f02d549	finance-router1

- 3.2. In the second terminal, use the **ip netns** command to list the firewall rules for **finance-router1**. Note the lack of any IP addresses in the **l3-agent** rule sets.

```
[heat-admin@controller0 ~]$ ip netns
...output omitted...
qrouter-7ee5fe29-ddf7-424d-aea9-f5e04f02d549
...output omitted...
[heat-admin@controller0 ~]$ sudo ip netns exec \
qrouter-7ee5fe29-ddf7-424d-aea9-f5e04f02d549 iptables -t nat -L -n -v
...output omitted...
Chain neutron-l3-agent-OUTPUT (1 references)
target    prot opt in     out   source   destination
          ...output omitted...

Chain neutron-l3-agent-PREROUTING (1 references)
target    prot opt in     out   source   destination
REDIRECT  tcp  --  qr-+  *  0.0.0.0/0  169.254.169.254  tcp...9697
...output omitted...

Chain neutron-l3-agent-float-snat (1 references)
target    prot opt in     out   source   destination
...output omitted...
```

4. Back on **workstation**, add a floating IP address to **finance-server1**. On **controller0**, use the **ip netns** command to list firewall rules and the NAT table for **finance-router1**. Notice the differences in the **OUTPUT**, **PREROUTING**, and **float-snat** tables.

- 4.1. On workstation, list the floating IP addresses available and attach one to **finance-server1**.

```
[student@workstation ~(developer1-finance)]$ openstack floating ip list \
-c "Floating IP Address" -c Port
+-----+-----+
| Floating IP Address | Port |
+-----+-----+
| 172.25.250.113      | None  |
| 172.25.250.114      | None  |
| 172.25.250.111      | None  |
| 172.25.250.112      | None  |
| 172.25.250.115      | None  |
+-----+-----+
[student@workstation ~(developer1-finance)]$ openstack server add \
floating ip finance-server1 172.25.250.P
```

- 4.2. In the **controller0** terminal, use the **ip netns exec** command to show the iptables firewall and NAT table rules in the qrouting namespace.

```
[heat-admin@controller0 ~]$ sudo ip netns exec \
qrouting-7ee5fe29-ddf7-424d-aea9-f5e04f02d549 iptables -t nat -L -n -v
Chain neutron-l3-agent-OUTPUT (1 references)
```

```

target prot opt in  out source      destination
DNAT    all  --  *   *  0.0.0.0/0  172.25.250.P to:192.168.1.N

...output omitted...

Chain neutron-13-agent-PREROUTING (1 references)
target    prot opt in      out     source      destination
REDIRECT  tcp   --  qr-+   *  0.0.0.0/0  169.254.169.254      tcp...9697
DNAT      all   --  *   *  0.0.0.0/0  172.25.250.P      to:192.168.1.N

Chain neutron-13-agent-float-snat (1 references)
target    prot opt in  out  source      destination
SNAT      all   --  *   *  192.168.1.N  0.0.0.0/0      to:172.25.250.P

```

- 4.3. In the **workstation** terminal, ensure that the IP addresses in the iptables rules and NAT table are the same as those configured for **finance-server1**.

```

[student@workstation ~(developer1-finance)]$ openstack server list \
-c Name -c Networks
+-----+-----+
| Name        | Networks          |
+-----+-----+
| finance-server1 | finance-network1=192.168.1.N, 172.25.250.P|
+-----+-----+

```

Cleanup

From **workstation**, run the **lab provisioning-namespaces cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab provisioning-namespaces cleanup
```

This concludes the guided exercise.

Quiz: Managing Networks in Linux

Choose the correct answer(s) to the following questions:

1. Which virtual bridge connects the instance running on a OpenStack compute node to the **br-int** OVS bridge?

The **3103c297-96** in the options is the first 11 characters of the associated OpenStack neutron port.

- a. Linux bridge (qbr3103c297-96)
- b. OVS bridge (br-ex)
- c. veth pair (qb3103c297-96 and qvo3103c297-96)
- d. TAP device (tap3103c297-96)

2. Which two virtual device connects OpenStack neutron routers to the **br-int** OVS bridge? (Choose two.)

The **750841ec-03** is the first 11 characters of the associated OpenStack neutron port.

- a. qr-750841ec-03
- b. qg-750841ec-03
- c. qbr750841ec-03
- d. qvb750841ec-03

3. Which virtual device connects the integration bridge (**br-int**) with the **br-tun** tunnel bridge?

- a. veth pair (**patch-tun** and **patch-int**)
- b. TAP device (**tap-tun**)
- c. OVS patch pairs (**patch-tun** and **patch-int**)

4. Which virtual device connects the external bridge (**br-ex**) with the **br-int** bridge?

- a. veth pair **phy-br-ex** on **br-int** bridge connects to **int-br-ex** on **br-ex** bridge.
- b. OVS patch pair **int-br-ex** on **br-int** bridge connects to the peer **phy-br-ex** on **br-ex** bridge.
- c. TAP device **int-br-ex** on **br-int** bridge connects to **phy-br-ex** on **br-ex** bridge.

Solution

Choose the correct answer(s) to the following questions:

1. Which virtual bridge connects the instance running on a OpenStack compute node to the **br-int** OVS bridge?

The **3103c297-96** in the options is the first 11 characters of the associated OpenStack neutron port.

- a. Linux bridge (qbr3103c297-96)
- b. OVS bridge (br-ex)
- c. veth pair (qb3103c297-96 and qvo3103c297-96)
- d. TAP device (tap3103c297-96)

2. Which two virtual device connects OpenStack neutron routers to the **br-int** OVS bridge? (Choose two.)

The **750841ec-03** is the first 11 characters of the associated OpenStack neutron port.

- a. qr-750841ec-03
- b. qg-750841ec-03
- c. qbr750841ec-03
- d. qvb750841ec-03

3. Which virtual device connects the integration bridge (**br-int**) with the **br-tun** tunnel bridge?

- a. veth pair (**patch-tun** and **patch-int**)
- b. TAP device (**tap-tun**)
- c. **OVS patch pairs (**patch-tun** and **patch-int**)**

4. Which virtual device connects the external bridge (**br-ex**) with the **br-int** bridge?

- a. veth pair **phy-br-ex** on **br-int** bridge connects to **int-br-ex** on **br-ex** bridge.
- b. **OVS patch pair **int-br-ex** on **br-int** bridge connects to the peer **phy-br-ex** on **br-ex** bridge.**
- c. TAP device **int-br-ex** on **br-int** bridge connects to **phy-br-ex** on **br-ex** bridge.

Summary

In this chapter, you learned:

- Network configuration files are used to manage linux interfaces. Configuration files ensure that changes made persist across reboots. The Network Manager service runs by default on Red Hat Enterprise Linux and may conflict OpenStack SDN management. Managing network interfaces using the **ip** and **ethtool** commands. Linux network namespaces are a clone of a Linux network stack. Namespaces ensure that tenant or project traffic remains independent and to avoid address conflicts between networks.
- Designing virtual requires network administrators to anticipate traffic routing and flow. OpenStack MTU configuration is important for correct tunnel configuration. MTU settings must be consistent throughout all networks in OpenStack. A typical ethernet packet has a default MTU of 1500 bytes. Traffic analysis tools, such as tcmpdump and Wireshark, are useful when troubleshooting.
- Network bridges are Link Layer devices to connect segments together. Device MAC address tables are built by learning which hosts are connected to each network. Linux bridges are used in virtualized environments to share a single physical NIC with more than one virtual NICs. Implement and manage virtual bridges with the **brctl** command. Manage Linux bridges persistently by altering configuration files.
- Open vSwitch provides a switching stack for hardware virtualization environments. The Linux bridge module and Open vSwitch can coexist on the same environment and are managed by the **ovs-vsctl** command. To troubleshoot OVS, first understand how packets flow through the different bridges and how to view flows.
- The **br-ex** bridge forwards traffic to and from the external network. The **br-int** bridge performs VLAN tagging and untagging. The **br-tun** bridge processes tunnel traffic using OpenFlow rules. The **br-trunk** bridge isolates infrastructure management vlans. The **ovs-ofctl** command is used to monitor and manage OpenFlow switches and view the current state of an OVS bridge.
- Virtual networking devices are similar to a physical network interface. Virtual devices can have an IP address and a MAC address and communicate just like a physical device. Virtual devices are connected to external devices through kernel routing, kernel bridging, Open vSwitch bridging, and network namespaces. TAP devices are virtual Ethernet devices simulating a link layer device. TAP devices are transient and are created and destroyed by the same program.
- A veth virtual device is a virtual patch cable. Two bridges can be connected together by adding veth device interfaces. In OpenStack, a virtual cable can connect a tenant network to the infrastructure network.
- Network Namespaces facilitate multi-tenancy, allowing OpenStack projects and network resources to operate in isolated network TCP/IP stacks. With namespaces, each project can utilize routing rules, IP addressing schemes and VM configuration without creating conflicts with other projects. OpenStack has namespaces to provide isolation for many types of Networking Service resource objects: routers, DHCP servers and NAT table processing daemons.



CHAPTER 2

MANAGING OPENSTACK NETWORKING AGENTS

Overview	
Goal	Manage L2, L3, DHCP, and other OpenStack Networking agents.
Objectives	<ul style="list-style-type: none">• Describe OpenStack Networking architecture.• Manage OpenStack Networking L2 and L3 agents.• Administer the OpenStack Networking DHCP agent.• Manage the load balancer agent, metadata agent, and metering agent.
Sections	<ul style="list-style-type: none">• Describing OpenStack Networking Architecture (and Quiz)• Managing the L2 and L3 Agents (and Guided Exercise)• Administering the DHCP Agent (and Guided Exercise)• Managing OpenStack Networking Agents (and Guided Exercise)
Lab	Managing OpenStack Networking Agents

Describing OpenStack Networking Architecture

Objectives

After completing this section, students should be able to:

- Define OpenStack Neutron.
- Describe Neutron components.
- Explain Neutron provisioning with modules.

Neutron Networking Overview

Neutron is the networking component of OpenStack. As with most OpenStack components, it is designed to run independently of other components while retaining a high level of integration. Neutron reads and writes messages to the same message queue used by other OpenStack components. It has an endpoint registered with Keystone in order for other services and users to discover the Neutron service. Neutron can authenticate users with Keystone so that tenants can request provisioning of network services in a platform-neutral way. Neutron can then instruct agents and plug-ins to provision those services in a platform-specific way.

Neutron provides a technology-agnostic way for a non-administrative user to provision network services with complete isolation from other tenants. The core Neutron model is based on instances in an isolated layer 2 network connected with layer 3 gateways providing NAT services.. The benefit of this model is the built-in security of connecting only servers that need to be connected to external networks. Additionally, deploying network filters is simplified, and authority to provision is delegated to the tenant that owns the networks. Running DHCP on the tenant network is not required.

Neutron Interface

Neutron's interface is a REST API, which can be accessed in many ways. An authenticated user with sufficient quota and privileges can provision network services such as L2 VLANs, L3 gateways, DHCP services, load balancing, using the **openstack** command. The legacy **neutron** command is still required for some tasks, and its capabilities are still being migrated to the **openstack** unified CLI. The graphical Horizon dashboard, operates by making REST API calls to all OpenStack components. The primary network objects in any Neutron deployment are:

- Network: an isolated layer 2 broadcast domain; a named segment.
- Subnet: an IP network prefix; assigned to an underlying network segment.
- Port: a connection object; displays address information for attached resource.
- Router: a layer 3 forwarding table; for moving traffic between networks.

Neutron Provisioning with Plug-ins

Network services are provisioned on OpenStack hosts as SDN, as well as in hardware routers and switches. Hardware vendors provide custom driver mechanisms for their proprietary provisioning interfaces. Neutron uses plug-ins and agents to implement extra functionality. Core plug-ins and agents are included to provide functionality, such as L2 network deployment or Load Balancing

as a Service (LBaaS). However, there are many agents to provide L2 services, and external vendors have LBaaS services to integrate with Neutron. Core plugins and services include:

- Neutron Server: Core daemon that provides API services
- DHCP Agent: DHCP services used by tenant networks
- L3 Agent: Provides NAT and routing capabilities to tenant networks
- OVS Agent: Allows Neutron to integrate with Open vSwitch and manage layer 2 networks

Optional plug-ins to increase connectivity and integration support include:

- Neutron ML2 framework: Framework that allows OpenStack Networking to simultaneously utilize multiple layer 2 networking technologies, such as:
 - Open vSwitch
 - Linux bridge
 - OpenDaylight
 - L2 Population
 - SR-IOV
- VPNaaS plugin: allows the creation of VPN tunnels
- FWaaS plugin: provides firewall services using iptables
- LBaaS plugin: provides load-balancing services using HAProxy

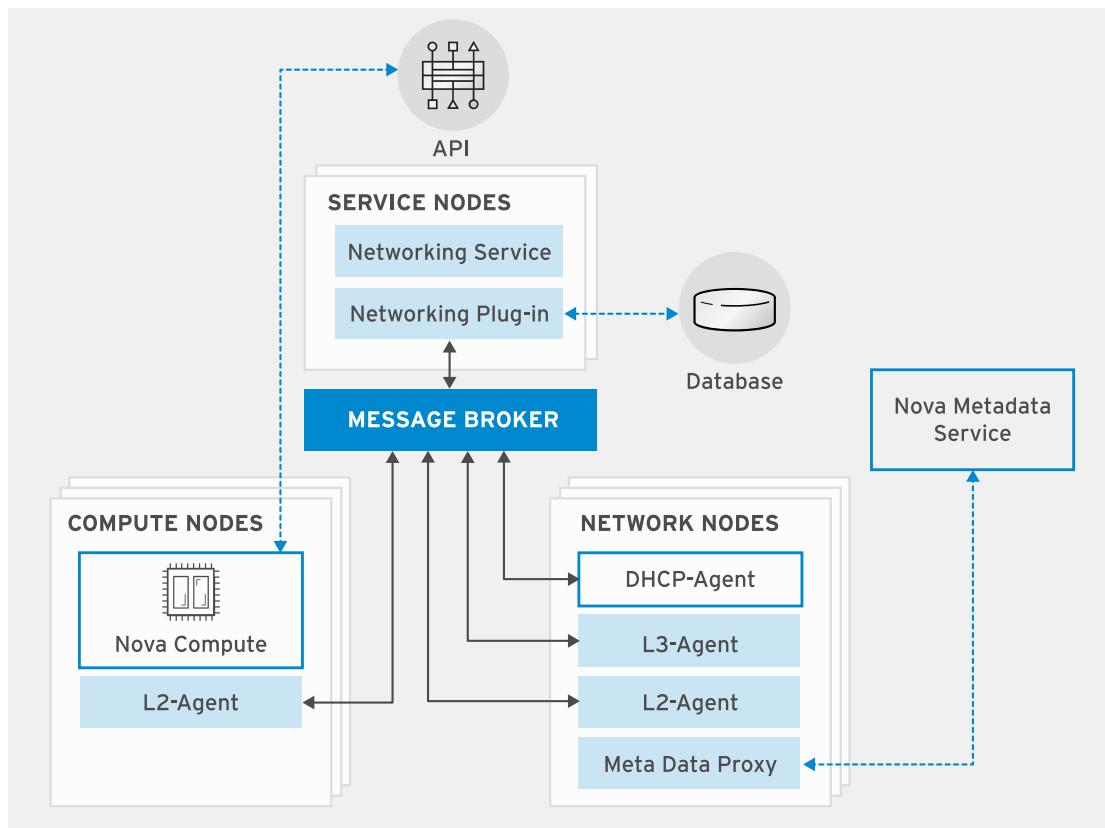


Figure 2.1: Neutron architecture

Neutron Agents

Openstack neutron server acts as the centralized controller, the actual network configuration are executed on either compute or network nodes. The neutron agents are software entities that

carry out configuration changes on compute or network nodes. Neutron agents communicate with the main Neutron service using the Neutron API and message queues.

Neutron DHCP Agent

The Neutron DHCP agent communicates with the OpenStack networking service over RPC. The DHCP agent ensures network isolation between various tenant networks using namespace, such that each tenant network has its own DHCP namespace created. Inside the DHCP namespace it runs the **dnsmasq** process that servers the DHCP request from instances. The Neutron DHCP agent configures the **dnsmasq** process using a DHCP lease file. The agent needs L2 connectivity with tenant networks.

Neutron LBaaS Agent

The Neutron LBaaS v2 agent provisions HAProxy load balancers on behalf of users. The HAProxy-based service plugin shipped with LBaaS v2 lacks deployment with Red Hat OpenStack Platform director deployment as well as high availability support. The LBaaS project in the upstream OpenStack community was replaced by Octavia which is a new LBaaS implementation for OpenStack. With the Octavia the LBaaS service is managed by fleet of virtual machines, containers or even baremetal servers. These fleet of servers are provisioned on demand to support horizontal scaling of load balancing service. Therefore, Red Hat have chosen to focus on Octavia and support it with Red Hat OpenStack Platform superseding LBaaS in future releases.

Neutron Metadata Agent

The Neutron Metadata agent is part of the mechanism that enables instances to obtain metadata about themselves, as a method of customization. The metadata agent handles the metadata request originating from an instance by sending the request using metadata proxy service to the OpenStack compute service. This allows the instance to retrieve additional configuration information from within the instance using the following URL:

```
[cloud-user@instance ]$ curl http://169.254.169.254/latest/meta-data/  
ami-id  
ami-launch-index  
ami-manifest-path  
block-device-mapping/  
hostname  
instance-action  
instance-id  
instance-type  
local-hostname  
local-ipv4  
placement/  
public-hostname  
public-ipv4  
public-keys/  
reservation-id  
[cloud-user@instance ]$ curl http://169.254.169.254/latest/meta-data/public-ipv4  
172.25.250.X
```

Neutron Metering Agent

Neutron has an API to measure bandwidth utilization for each client. The Neutron Metering agent is responsible for tracking router updates, collecting traffic counters from routers, and sending metering notifications.

Troubleshooting Neutron

When troubleshooting Neutron networking issues start with checking services, followed by investigating the log files for the relevant agent. Use the **openstack network agent list** command to see the status of Neutron agents.

```
[user@demo ~(admin-admin)]$ openstack network agent list \
-c 'Agent Type' -c Host -c Alive -c State
+-----+-----+-----+
| Agent Type | Host | Alive | State |
+-----+-----+-----+
| Open vSwitch agent | compute0.overcloud.example.com | True | UP |
| Metadata agent | controller0.overcloud.example.com | True | UP |
| L3 agent | controller0.overcloud.example.com | True | UP |
| DHCP agent | controller0.overcloud.example.com | True | UP |
| Loadbalancerv2 agent | controller0.overcloud.example.com | True | UP |
| Metadata agent | compute0.overcloud.example.com | True | UP |
| Metadata agent | compute1.overcloud.example.com | True | UP |
| Open vSwitch agent | controller0.overcloud.example.com | True | UP |
| L3 agent | compute1.overcloud.example.com | True | UP |
| L3 agent | compute0.overcloud.example.com | True | UP |
| Open vSwitch agent | compute1.overcloud.example.com | True | UP |
+-----+-----+-----+
```

The following table describes Neutron services.

Neutron Services

Service	Description
neutron-dhcp-agent	Provides DHCP services for internal networks and supports metadata operations for isolated networks
neutron-l3-agent	Provides routing and NAT services
neutron-metadata-agent	Supports instance customization by providing access to metadata
neutron-metering-agent	Collects traffic counters for reporting bandwidth usage by clients
neutron-openvswitch-agent	Allows Neutron to manage Open vSwitch
neutron-ovs-cleanup	Cleans up stale entries in the Open vSwitch database
neutron-server	Provides the API for networking requests

Log files may be located on controller or compute nodes, depending on the location of the agent or service. Location of agents is also dependent on whether centralized routing or a Distributed Virtual Router is in use. The following table lists log file locations for Neutron services.

Neutron Log Files

Service	Log File
neutron-dhcp-agent	/var/log/neutron/dhcp-agent.log
neutron-l3-agent	/var/log/neutron/l3-agent.log
neutron-metadata-agent	/var/log/neutron/metadata-agent.log
neutron-metering-agent	/var/log/neutron/metering-agent.log
neutron-openvswitch-agent	/var/log/neutron/openvswitch-agent.log

Service	Log File
neutron-ovs-cleanup	/var/log/neutron/ovs-cleanup.log
neutron-server	/var/log/neutron/server.log



References

Further information is available in the chapter on OpenStack networking concepts in the *Red Hat OpenStack Platform 10 Networking Guide* at

| <https://access.redhat.com/documentation/en/red-hat-openstack-platform/>

Quiz: Describing OpenStack Networking Architecture

Choose the correct answer(s) to the following questions:

1. Which one of the following was the predecessor to Neutron networking in Red Hat OpenStack Platform?
 - a. Quasar networking
 - b. Novis networking
 - c. Nova networking
 - d. Stack networking

2. Which three of the following features are provided by Neutron? (Choose three.)
 - a. FWaaS (Firewall as a Service)
 - b. DNSaaS (Domain Naming System as a Service)
 - c. LBaaS (Load Balancing as a Service)
 - d. VPNaas (Virtual Private Network as a Service)

3. Which five of the following network technologies are supported by Neutron? (Choose five.)
 - a. L2 Population
 - b. Open vSwitch
 - c. SR-IOV
 - d. Linux Bridge
 - e. OpenDaylight
 - f. Broadcast Relay

Solution

Choose the correct answer(s) to the following questions:

1. Which one of the following was the predecessor to Neutron networking in Red Hat OpenStack Platform?
 - a. Quasar networking
 - b. Novis networking
 - c. **Nova networking**
 - d. Stack networking
2. Which three of the following features are provided by Neutron? (Choose three.)
 - a. FWaaS (Firewall as a Service)
 - b. DNSaaS (Domain Naming System as a Service)
 - c. LBaaS (Load Balancing as a Service)
 - d. VPNaas (Virtual Private Network as a Service)
3. Which five of the following network technologies are supported by Neutron? (Choose five.)
 - a. L2 Population
 - b. Open vSwitch
 - c. SR-IOV
 - d. Linux Bridge
 - e. **OpenDaylight**
 - f. Broadcast Relay

Managing the L2 and L3 Agents

Objectives

After completing this section, students should be able to:

- Describe the L2 and L3 OpenStack Networking agents.
- Troubleshoot the L2 and L3 OpenStack Networking agents.

Managing Neutron L2 Agents

The ML2 plug-in uses an Open vSwitch agent, by default, to provision L2 networking on Neutron nodes. Its capability to provision is almost identical to the Open vSwitch plug-in. However, it has the additional capability of running multiple agents to manage various hardware switches. With ML2, hardware vendors have moved from providing Neutron agents to providing ML2 plug-ins. It is possible to develop a variety of network designs with ML2 using VLAN, GRE, VXLAN, and OpenFlow.

If the Open vSwitch agent is used, a special bridge called an integration bridge is used to connect the instance's TAP ports to the rest of the servers running Neutron with an L2 agent and with a resource to share. Instances connected to the same integration bridge cannot see each other. Open vSwitch uses VLAN, VXLAN, GRE tagging, or tunneling to isolate instances to the networks specified by the user. The bridge needs to be created in Open vSwitch, and Neutron needs to be configured with the name of the bridge. All servers needing an L2 plug-in need this bridge and configuration.

The **neutron-ovs-cleanup** service runs as needed to remove stale bridge entries from the Open vSwitch database on startup.

Managing Neutron L3 Agents

The L3 agent is included with the *openstack-neutron* package. The agent allows administrators and tenants to create routers to interconnect L2 networks and create floating IP addresses to map to internal addresses. Just like physical routers, each interface on a router must be on a different subnet. Tenants reusing the same subnet on multiple networks need multiple routers.

The **neutron-l3-agent** uses the Linux IP stack and Netfilter to perform L3 forwarding and NAT. In order to support multiple routers with potentially overlapping IP addresses, **neutron-l3-agent** defaults to using Linux network namespaces to provide an isolated context for each tenant router. As a result, the IP addresses of routers are not visible simply by running **ip addr list** or **ifconfig** on the node. Similarly, users are not able to directly reach fixed IPs using the **ping** command. To do either of these things, users must run the command within a particular router's network namespace. The namespace has the name **qrouter-UUID**, where *UUID* is the ID of the router.



Important

Since Red Hat OpenStack Platform 5, all features have been ported to the ML2 plug-in in the form of mechanism drivers. ML2 currently provides Linux Bridge, Open vSwitch, and Hyper-V mechanism drivers.

Neutron Configuration File

Given the default value of `ml2` for the `core_plugin` directive in `/etc/neutron/neutron.conf`, the startup scripts start the daemons by using the corresponding plug-in configuration file directory. For example, the configuration file for the Open vSwitch plugin is `/etc/neutron/plugins/ml2/openvswitch_agent.ini`. OpenStack Networking plug-ins can be referenced in `/etc/neutron/neutron.conf` by their short names.

`/etc/neutron/neutron.conf - DEFAULT section`

<i>Plug-in Short name</i>	<i>Class name</i>
bigswitch	neutron.plugins.bigswitch.plugin.NeutronRestProxyV2
brocade	neutron.plugins.brocade.NeutronPlugin.BrocadePluginV2
cisco	neutron.plugins.cisco.network_plugin.PluginV2
hyperv	neutron.plugins.hyperv.hyperv_neutron_plugin.HyperVNeutronPlugin
linuxbridge	neutron.plugins.linuxbridge.lb_neutron_plugin.LinuxBridgePluginV2
ml2	neutron.plugins.ml2.plugin.ML2Plugin
nec	neutron.plugins.nec.nec_plugin.NECPluginV2
openvswitch	neutron.plugins.openvswitch.ovs_neutron_plugin.OVSNeutronPluginV2

Centralized (Legacy) Routing

Originally, OpenStack networking was designed with a centralized routing in which tenant routers are deployed in a dedicated node (referred to as the network node, but usually the controller node). The traffic in east/west or north/south routing scenarios in case of centralized routing traverses through the dedicated network node. This introduce several challenges with scalability and high availability of network node:

- When two VMs need to communicate with each other located on the same compute node. The VM traffic has to leave the compute node, flow through network node and then the traffic is routed back to the other VM on the same compute node. This affects the network performance.
- VMs with floating IP address in centralized routing requires the traffic to be routed through the centralized external gateway available only on the network node. This routing model is true from traffic originating from a VM or destined to a VM. In a large environment, the network node performance, scalability, and availability is affected when the node is subjected to high network traffic.

The L3-HA overcomes the availability challenge posed by centralized routing by distributing the routing function across multiple network nodes. Currently L3-HA is the default routing model used in Red Hat OpenStack Platform. Distributed routing model aims to provide high-availability and network traffic optimization by deploying L3 agent and routers on each compute nodes.

Currently, distributed routing and L3-HA cannot be configured together, even though it is possible to mix legacy (centralized) and distributed (DVR) routers. Distributed Virtual Routers are redundant by design, allowing VMs to migrate to other compute nodes with functioning DVR in case of failure. Future L3-HA development is expected to add redundancy for the SNAT service.

To create legacy centralized router when distributed router is configured as the default routing model. Use the `neutron router-create` command with the `--distributed False` option.

The `python-openstackclient` package shipped with Red Hat OpenStack Platform 10 do not allow creation of centralized router using `openstack router create`.

```
[user@demo ~(admin-admin)]$ neutron router-create --distributed False demo-router1
```

L3-Agent and VRRP (L3-HA)

The Virtual Router Redundancy Protocol (VRRP) provides high availability and failover of routing in OpenStack networking. VRRP requires at least two Network Service nodes, for a minimum architecture of a master and a backup router instance. VRRP uses **keepalived** to transmit a heartbeat that connects all VRRP routers in a project. The backup router promotes itself to master should **keepalived** stops receiving heartbeats from the master. When promoted, the router re-configures necessary IP addresses on the **qrouter namespace** interfaces. If there is more than one backup router, the one with the highest IP address promotes itself to master.



Note

Failure of the L3-Agent, or manually restarting it, does not trigger a failover as long as **keepalived** continues working.

VRRP uses hidden networks for **keepalived** traffic, isolated per project. The network type used for hidden networks can be any of the **tenant_network_types** configured in **m12_conf.ini**. Set the hidden network type and name using **l3_ha_network_type** and **l3_ha_network_name** in **neutron.conf**.

The VRRP Protocol is only used on tenant networks or projects using legacy L3 routers. Provider networks do not need L3-HA because they do not use L3-Agent routers; by definition, they use external physical routers. It is possible to distribute the master router instances over several network node to reduce the amount of traffic on one particular network node. In production environments, recommended practice is to use a minimum of three Networking Service nodes.

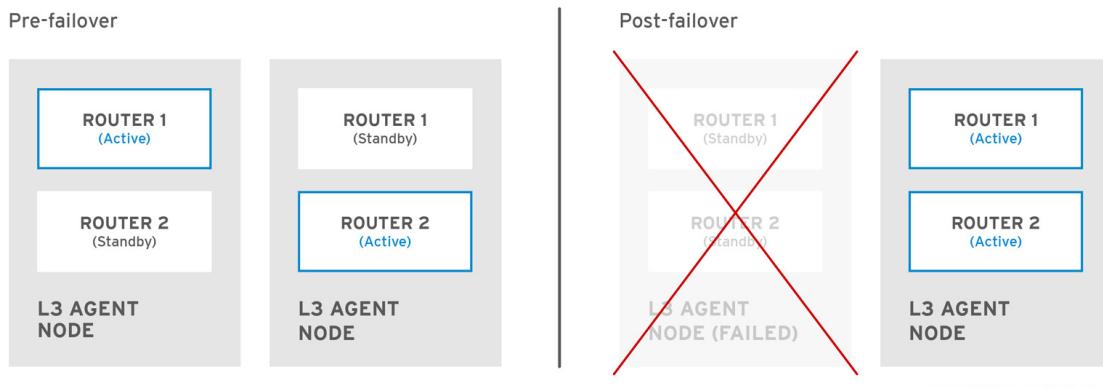


Figure 2.2: VRRP failover architecture

Troubleshooting L2 and L3 Agents

L2 Agent

The L2 Agent runs on a compute node. The L2 Agent's main duties are wiring new devices and configuring local switches. It communicates with the Neutron service using RPC. Security groups

are also applied by the L2 Agent, implemented using iptables and IP sets. The L2 Agent has log files on both controller and compute nodes. The **nova-compute.log** log file is located on the compute node in the **/var/log/nova/** directory. The **openvswitch-agent.log** log file is located in the **/var/log/neutron** directory. The **server.log** log file is in the **/var/log/neutron/** directory on the controller node.

L3 Agent

The main responsibility of the L3 agent is to provide Layer 3 networking and tenant network connectivity. It is also responsible for NAT, network namespace management, and external network connectivity.

End-to-end operational troubleshooting in OpenStack is challenging. Each instance has a **TAP** device acting as its network interface. The TAP devices are bridged using a Linux bridge into an Open vSwitch bridge. The Linux bridge is required because that is where security groups are implemented.

Gather basic information before troubleshooting as the same information is commonly used repetitively during diagnostics. Basic information includes the **instance name**, **GUID**, **hypervisor**, and **IP addresses** for the instance. Use **openstack server show server_name** as an admin user to be able to view all necessary information.

[user@demo ~ (admin-admin)]\$ openstack server show finance-server2	
Field	Value
OS-DCF:diskConfig	MANUAL
OS-EXT-AZ:availability_zone	nova
OS-EXT-SRV-ATTR:host	compute0.overcloud.example.com
OS-EXT-SRV-ATTR:hypervisor_hostname	compute0.overcloud.example.com
OS-EXT-SRV-ATTR:instance_name	instance-0000000b
OS-EXT-STS:power_state	Running
OS-EXT-STS:task_state	None
OS-EXT-STS:vm_state	active
OS-SRV-USG:launched_at	2017-10-12T08:49:33.000000
OS-SRV-USG:terminated_at	None
accessIPv4	
accessIPv6	
addresses	finance-network1= 192.168.1.12, 172.25.250.113
config_drive	
created	2017-10-12T08:49:14Z
flavor	default (6113...98ad)
hostId	1c3d...46e4
id	8510...0f1e
image	rhel7 (e59a...5978)
key_name	example-keypair
name	finance-server2
os-extended-volumes:volumes_attached	[]
progress	0
project_id	0ed0...5dd8
properties	
security_groups	[{"name": "default"}]
status	ACTIVE
updated	2017-10-12T08:49:33Z
user_id	a560...a916

The network name and ID can also be learned using the **openstack server show** command. The **segmentation_id** is a key field, displayed in OpenFlow in hexadecimal form. Convert the

decimal integer to hexadecimal using the command `printf "%x\n" N`. In this case the decimal value 3 equals hexadecimal **0x3**.

[user@demo ~(admin-admin)]\$ openstack network show finance_network1	
Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	nova
created_at	2017-10-07T05:41:43Z
description	
id	03f83f61-5ca0-44dd-84cd-fd4ab02f6c0b
ipv4_address_scope	None
ipv6_address_scope	None
mtu	1446
name	finance-network1
port_security_enabled	True
project_id	0ed0cb3b180d46afb510b12a09675dd8
provider_id	0ed0cb3b180d46afb510b12a09675dd8
provider:network_type	vxlan
provider:physical_network	None
provider:segmentation_id	3
qos_policy_id	None
revision_number	5
router:external	Internal
shared	False
status	ACTIVE
subnets	a4781a7a-734c-4e3d-8d7d-23598f18a4f2
tags	[]
updated_at	2017-10-07T05:41:45Z

Using the `openstack network agent list` command, find the host of the **DHCP agent**.

[user@demo ~(admin-admin)]\$ openstack network agent list \ -c ID -c 'Agent Type' -c Host		
ID	Agent Type	Host
0371...0210	Metadata agent	compute0.overcloud.example.com
17e5...9a9e	Open vSwitch agent	compute0.overcloud.example.com
3393...15dd	Metadata agent	controller0.overcloud.example.com
846c...e1b9	L3 agent	controller0.overcloud.example.com
85bd...0e31	Open vSwitch agent	controller0.overcloud.example.com
a784...a9e9	L3 agent	compute0.overcloud.example.com
c9c2...2239	L3 agent	compute1.overcloud.example.com
d969...22be	Open vSwitch agent	compute1.overcloud.example.com
dd91...5fad	DHCP agent	controller0.overcloud.example.com
f144...d744	Metadata agent	compute1.overcloud.example.com

Basic Troubleshooting

Most networking problems are due to misconfiguration. It is essential to check that the basic configuration is correct. The following list contains the first things to check when troubleshooting OpenStack networking.

- On the compute and controller nodes, ensure that the interfaces are in the correct network and that they have IP addresses. Ensure that you can ping from one to the other.

- Ensure that the compute node has the correct Open vSwitch bridges: **br-int** and **br-tun**. Check the tunnels that connect the virtual switches. Do they have the correct IP addresses? Are they on the correct subnet?
- Check the log files for nova-compute, neutron-server, openvswitch-agent and OVS.
- Verify that the DHCP agent is present and running.
- In the network namespace, ensure that **dnsmasq** is running.
- Using the **ovs-vsctl** command, check the virtual switch configuration.

Advanced Troubleshooting

If the basic configuration is correct but the problem persists, try the following advanced troubleshooting.

- Using the **sudo virsh dumpxml *ID* | grep tap** command, retrieve the name of the TAP device for the instance. This command is run on the host compute node.
- Using the **iptables -S | grep tap-*ID*** command on the compute node, check the firewall rules. Then using the **ip netns exec qdhcp-*ID* iptables -L** command on the **network** node, check the firewall rules in the network namespace.
- On the **compute** node, run the **brctl show** command. Ensure that the TAP device and interfaces are correct. Then use the **ovs-vsctl show** command to ensure that the interfaces are listed as ports of **br-int**. If there are any problems with this configuration, check the Nova and Neutron log files.
- Check the OpenFlow traffic using the **ovs-ofctl dump-flows br-tun** command. Use the segmentation ID that was collected and translated to HEX earlier. In this example, ID=3, which is HEX **0x3**. Use **grep** to make the output more readable. The **mod_vlan_vid** parameter will have the same ID as the **port tag** from the output of the **ovs-vsctl show** command.

```
[heat-admin@compute1 ~]$ sudo ovs-vsctl show
Bridge br-int
  Controller "tcp:127.0.0.1:6633"
  is_connected: true
  fail_mode: secure
  ...output omitted...
  Port "qvod689be46-a4"
    tag: 1
    Interface "qvod689be46-a4"
[heat-admin@compute1 ~]$ sudo ovs-ofctl dump-flows br-tun | grep 0x3
cookie=0x8765cfe011f38a19, duration=516916.145s, table=4, n_packets=6323,
  n_bytes=531183, idle_age=91, hard_age=65534, priority=1,tun_id=0x3
  actions=mod_vlan_vid:1,resubmit(,9)
  ...output omitted...
```

You can also check the OpenFlow table for more information using the **ovs-ofctl dump-flows** command.

```
[heat-admin@compute1 ~]$ sudo ovs-ofctl dump-flows \
br-tun table=4
NXST_FLOW reply (xid=0x4):
```

```
cookie=0x9039e8880106dbc8, duration=518211.561s, table=4, n_packets=0,
n_bytes=0, idle_age=65534, hard_age=65534, priority=1,tun_id=0x64
actions=mod_vlan_vid:1,resubmit(,9)
```

- On the controller node, retrieve the **TAP** device of the network namespace. Use the **ip netns** command. Use the **ovs-vsctl** command to retrieve the tag information for the TAP device. Using the **ovs-ofctl dump flows** command, retrieve the OpenFlow information. Note the flow information for the TAP device.

```
[heat-admin@compute1 ~]$ ip -n qdhcp-ID a
37: tapd75c6e33-2d: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue state UNKNOWN mode DEFAULT qlen 1000
    link/ether fa:16:3e:37:e5:fe brd ff:ff:ff:ff:ff:ff
[heat-admin@compute1 ~]$ ovs-vsctl show | grep -A1 d75c6e33-2d
Port "tapd75c6e33-2d"
tag: 5
  Interface "tapd75c6e33-2d"
    type: internal
...output omitted...
[heat-admin@compute1 ~]$ ovs-ofctl dump-flows \
br-tun table=4 | grep 0x3
[root@controller0 ~]# ovs-ofctl dump-flows br-tun table=4 | grep 0x3
357 cookie=0x9039e8880106dbc8, duration=517870.521s, table=4, n_packets=15324,
n_bytes=1118186, idle_age=63, hard_age=65534, priority=1,tun_id=0x3
actions=mod_vlan_vid:5,resubmit(,9)
```

Troubleshooting Tools

Here are the main commands to use when troubleshooting networking problems in Red Hat OpenStack Platform.

Command	Description
ip addr	Inspect devices on the node or within a namespace. Show names, status, IP address and MTU settings.
ip route	Show the routing table. For viewing the packet path between networks.
iptables -L	Show firewall rules within a namespace. Dropped packets may be caused by a deny or missing rules.
tcpdump	Capture specific types of packets on specified interfaces. Analyze packets that have been saved to a file.
wireshark	Graphic interface to capture specific types of packets on specified interfaces. Analyze packets that have been saved to a file.
ip netns	Executes commands within namespaces.
ovs-vsctl show	Show the configuration of Open vSwitch bridges.
ovs-ofctl show	Show Open vSwitch data paths.
ovs-ofctl dump-flows	Provide dumps of the Open vSwitch flow table for a specified bridge.
brctl show	Show the configuration of a specified Linux bridge.

Managing L2 and L3 Agents

Watch this video as the instructor shows how to manage and troubleshoot L2 and L3 Agents in OpenStack.

1. Inspect the interfaces of the compute node.
2. Create a new server instance, ensuring that **compute0** is the host server. Inspect the new devices created on the compute node.
3. Inspect the Neutron and Nova log files to see the actions taken by the L2 and L3 agents when the instance was created.
4. Add a floating IP address to the instance and review the Neutron log file.
5. Look at the different configuration files for the L2 and L3 Agents.
6. Delete the instance and reinspect the devices on the compute node.

References

Further information is available in the chapter on Configuring layer 3 high availability in the *Red Hat OpenStack Platform 10 Networking Guide* at

<https://access.redhat.com/documentation/en/red-hat-openstack-platform/>

Guided Exercise: Managing the L2 and L3 Agents

In this exercise, you will:

- Launch an instance and follow the L2 and L3 agent logs as they do their work
- View the devices created by the L2 and L3 agents when you launch an instance

Outcomes

You should be able to locate Neutron network issues during instance provisioning

Before you begin

Log in to workstation as **student** using **student** as the password. On **workstation**, run the **lab agents-12-13 setup** command. This script verifies that the overcloud nodes are accessible and the OpenStack services are running, and sets up the prerequisites for this guided exercise.

```
[student@workstation ~]$ lab agents-12-13 setup
```

Steps

1. Source the **architect1** user credentials for the **finance** project.

```
[student@workstation ~]$ source ~/architect1-finance-rc
[student@workstation ~(architect1-finance)]$
```

2. Log in to **compute0** as the **root** user and list the network interfaces. You should not find any instance networking or bridge devices yet. You will run this command again after you create an instance and view the differences.

2.1. Connect using SSH to **compute0**.

```
[student@workstation ~(architect1-finance)]$ ssh root@compute0
Last login: Thu Sep 14 04:30:15 2017 from workstation.lab.example.com
[root@compute0 ~]#
```

2.2. List the network interfaces.

```
[root@compute0 ~]# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    qlen 1
        link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    mode DEFAULT qlen 1000
        link/ether 52:54:00:00:f9:02 brd ff:ff:ff:ff:ff:ff
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master ovs-
    system state UP mode DEFAULT qlen 1000
        link/ether 52:54:00:01:00:02 brd ff:ff:ff:ff:ff:ff
4: eth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master ovs-
    system state UP mode DEFAULT qlen 1000
        link/ether 52:54:00:02:fa:02 brd ff:ff:ff:ff:ff:ff
```

```

5: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT
  qlen 1000
    link/ether 0a:5a:7f:35:f9:c6 brd ff:ff:ff:ff:ff:ff
6: br-int: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT
  qlen 1000
    link/ether 7a:71:8e:08:37:45 brd ff:ff:ff:ff:ff:ff
7: br-ex: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN
  mode DEFAULT qlen 1000
    link/ether 52:54:00:02:fa:02 brd ff:ff:ff:ff:ff:ff
8: br-tun: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT
  qlen 1000
    link/ether c2:a8:5e:b5:74:4d brd ff:ff:ff:ff:ff:ff
9: vlan20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
  UNKNOWN mode DEFAULT qlen 1000
    link/ether f2:6f:b0:d1:16:12 brd ff:ff:ff:ff:ff:ff
10: vlan10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
  UNKNOWN mode DEFAULT qlen 1000
    link/ether c2:92:a5:28:e8:3f brd ff:ff:ff:ff:ff:ff
11: vlan50: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
  UNKNOWN mode DEFAULT qlen 1000
    link/ether e6:1e:0b:15:3a:d5 brd ff:ff:ff:ff:ff:ff
12: vlan30: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
  UNKNOWN mode DEFAULT qlen 1000
    link/ether 6a:e4:62:98:71:cb brd ff:ff:ff:ff:ff:ff
13: br-trunk: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
  UNKNOWN mode DEFAULT qlen 1000
    link/ether 52:54:00:01:00:02 brd ff:ff:ff:ff:ff:ff
[root@compute0 ~]# exit
[student@workstation ~(architect1-finance)]$
```

3. From workstation, create an instance named **finance-server1** using existing resources. Using the **--availability-zone** option specify that the instance should be installed on the **compute0** node.

```
[student@workstation ~(architect1-finance)]$ openstack server create \
--image rhel7 \
--nic net-id=finance-network1 \
--key-name example-keypair \
--flavor default \
--availability-zone nova:compute0.overcloud.example.com \
--wait finance-server1
...output omitted...
```

4. As the **root** user, log in to **compute0**. List the network interfaces and observe the differences from the earlier output.

4.1. Connect using SSH to **compute0**.

```
[student@workstation ~(architect1-finance)]$ ssh root@compute0
Last login: Thu Sep 14 04:30:15 2017 from workstation.lab.example.com
[root@compute0 ~]#
```

4.2. List the network interfaces.

```
[root@compute0 ~]# ip link
...output omitted...
19: qbr3b24df55-74: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue
  state UP mode DEFAULT qlen 1000
```

```

link/ether 16:43:d0:56:20:ca brd ff:ff:ff:ff:ff:ff
20: qvo3b24df55-74@qvb3b24df55-74: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
1446 qdisc noqueue master ovs-system state UP mode DEFAULT qlen 1000
    link/ether 2a:56:a7:56:33:b0 brd ff:ff:ff:ff:ff:ff
21: qvb3b24df55-74@qvo3b24df55-74: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu
1446 qdisc noqueue master qbr3b24df55-74 state UP mode DEFAULT qlen 1000
    link/ether 16:43:d0:56:20:ca brd ff:ff:ff:ff:ff:ff
22: tap3b24df55-74: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc pfifo_fast
    master qbr3b24df55-74 state UNKNOWN mode DEFAULT qlen 1000
    link/ether fe:16:3e:ca:88:9d brd ff:ff:ff:ff:ff:ff
23: vxlan_sys_4789: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 65470 qdisc noqueue
    master ovs-system state UNKNOWN mode DEFAULT qlen 1000
    link/ether e2:e1:52:f9:d7:28 brd ff:ff:ff:ff:ff:ff

```

In the command output, you can see the TAP device, the **qvb** and **qvo** veth pair, and the **qbr** Linux bridge.

4.3. List the Linux bridges.

```

[root@compute0 ~]# brctl show
bridge name      bridge id      STP enabled  interfaces
qbr3b24df55-74  8000.1643d05620ca  no          qvb3b24df55-74
                                         tap3b24df55-74

```

You can see the TAP interface connecting the instance to the Linux bridge, and the **qvb** interface connecting the Linux bridge to the OVS integration bridge.

4.4. List the ports on the OVS integration bridge.

```

[root@compute0 ~]# ovs-vsctl list-ports br-int
int-br-ex
patch-tun
qvo3b24df55-74
[root@compute0 ~]# exit
[student@workstation ~(architect1-finance)]$ 

```

Note the **qvo** interface connecting the OVS integration bridge to the Linux bridge.

- From workstation, open a second terminal and use SSH to connect to **compute0**. Examine the actions of the Open vSwitch agent in **/var/log/neutron/openvswitch-agent.log**.

5.1. Here you can see the membership for a security group being updated.

```

2017-09-14 12:24:16.326 266152 INFO neutron.agent.securitygroups_rpc
[req-9d9570b2-28d3-418a-b047-b13217356b0d c538335020244c0b9ea92d06864f49c6
90fd25f9104d4487a689a5bac14f3fd4 - - -] Security group member updated
[u'e0961cd5-89b6-4498-a14f-d6e965c22268']

```

- On workstation source the **developer1-finance** credentials. Using the **openstack security group list** command determine which security group was updated.

```

[student@workstation ~(architect1-finance)]$ source ~/developer1-finance-rc
[student@workstation ~(developer1-finance)]$ openstack security group list \
-c ID -c Name
+-----+-----+
| ID   | Name  |

```

e0961cd5-89b6-4498-a14f-d6e965c22268 default

5.3. Here you can see a port being updated.

```
2017-09-14 12:24:30.644 266152 INFO
neutron.plugins.ml2.drivers.openvswitch.agent.ovs_neutron_agent
[req-83cd9995-4cc9-480f-b838-16687744e3d6 - - - - ]
Port 551c4b7a-4e8e-4e65-810e-a06f3a0ee470 updated. Details:
{u'profile': {}, u'network_qos_policy_id': None, u'qos_policy_id': None,
u'allowed_address_pairs': [], u'admin_state_up': True, u'network_id':
u'5c7bcbfd-9308-450a-93c8-e30c316ccaa', u'segmentation_id': 1,
u'device_owner': u'compute:None', u'physical_network': None, u'mac_address':
u'fa:16:3e:c9:87:01', u'device': u'551c4b7a-4e8e-4e65-810e-a06f3a0ee470',
u'port_security_enabled': True, u'port_id': u'551c4b7a-4e8e-4e65-810e-
a06f3a0ee470', u'fixed_ips': [{u'subnet_id': u'3c72295b-f300-4e9c-
b6d4-7fdbca20b7af2', u'ip_address': u'192.168.1.5'}], u'network_type': u'vxlan'}
```

5.4. On workstation, correlate the resource IDs in the previous step to their names.

```
[student@workstation ~ (developer1-finance)]$ openstack port list \
-c ID -c 'Fixed IP Addresses'
+-----+
| ID | Fixed IP Addresses |
+-----+
| 12ae...6189 | ip_address='192.168.1.2', subnet_id='3c72...7af2' |
| 3326...fe10 | ip_address='192.168.1.1', subnet_id='3c72...7af2' |
| 551c...e470 | ip_address='192.168.1.5', subnet_id='3c72...7af2' |
| ef83...5165 | ip_address='192.168.2.2', subnet_id='e409...064c' |
+-----+
[student@workstation ~ (developer1-finance)]$ openstack network list \
-c ID -c Name
+-----+
| ID | Name |
+-----+
| 5ad5c71f-a848-45e3-800d-f379b1f111e7 | finance-network2 |
| 5c7bcbfd-9308-450a-93c8-e30c316ccaa | finance-network1 |
| f8198bbb-5f83-4b79-af27-c267b4d9c72c | provider-datacentre |
+-----+
[student@workstation ~ (developer1-finance)]$ openstack subnet list \
-c ID -c Name
+-----+
| ID | Name |
+-----+
| 3c72295b-f300-4e9c-b6d4-7fdbca20b7af2 | finance-subnet1 |
| 96c22e97-0482-4794-a4d6-9c143d1aeb30 | provider-subnet-172.25.250 |
| e409ee7e-22fd-43c5-bafb-69936de5064c | finance-subnet2 |
+-----+
[student@workstation ~ (developer1-finance)]$ openstack server list \
-c Name -c Networks
+-----+
| Name | Networks |
+-----+
| finance-server1 | finance-network1=192.168.1.5 |
+-----+
```

5.5. In the second terminal you can see several other events.

```

2017-09-14 12:24:30.645 266152 INFO neutron.plugins.ml2.drivers.openvswitch.agent.ovs_neutron_agent
[req-83cd9995-4cc9-480f-b838-16687744e3d6] Assigning 1 as local vlan
for net-id=5c7bcbfd-9308-450a-93c8-e30c316ccaaaf
2017-09-14 12:24:30.752 266152 INFO neutron.agent.l2.extensions.qos
[req-83cd9995-4cc9-480f-b838-16687744e3d6] QoS extension did have no
information about the port 551c4b7a-4e8e-4e65-810e-a06f3a0ee470 that we were
trying to reset
2017-09-14 12:24:30.856 266152 INFO neutron.agent.securitygroups_rpc
[req-83cd9995-4cc9-480f-b838-16687744e3d6] Preparing filters for
devices set([u'551c4b7a-4e8e-4e65-810e-a06f3a0ee470'])
2017-09-14 12:24:31.309 266152 INFO neutron.plugins.ml2.drivers.openvswitch.agent.ovs_neutron_agent
[req-83cd9995-4cc9-480f-b838-16687744e3d6] Configuration for devices
up [u'551c4b7a-4e8e-4e65-810e-a06f3a0ee470'] and devices down [] completed.

```

- A local VLAN ID of 1 is assigned for the **finance-network1** network. This confirms that traffic for a given network cannot transit Open vSwitch until an instance with a connection to that network is running on this node.
- No QoS configuration was found for this port.
- The security group filters were applied to the port.

5.6. Exit from the second terminal.

6. Delete the **finance-server1** instance.

```
[student@workstation ~(developer1-finance)]$ openstack server delete finance-server1
```

7. On **compute0**, change the **firewall_driver** in **/etc/neutron/plugins/ml2/openvswitch_agent.ini** from using the iptables firewall driver to the Open vSwitch firewall driver. Restart Neutron services. You must be root to perform this action.



Note

In Red Hat OpenStack Platform 10, the OVS firewall driver is in tech preview, it is shown here simply to highlight the differences in bridge and interface layout.

- 7.1. Log in to **compute0** as **root** user. Update the **/etc/neutron/plugins/ml2/openvswitch_agent.ini** file as shown in the screen below. Just comment out the original line because you will be reverting your changes at the end of this exercise.

```

...output omitted...
[securitygroup]
#
# From neutron.ml2.ovs.agent
#
# Driver for security groups firewall in the L2 agent (string value)
#firewall_driver = <None>
#firewall_driver =
neutron.agent.linux.iptables_firewall.OVSHybridIptablesFirewallDriver
firewall_driver = openvswitch

```

```
...output omitted...
```

7.2. Restart Neutron services.

```
[root@compute0 ~]# systemctl restart neutron\*
[root@compute0 ~]# exit
[student@workstation ~(developer1-finance)]$
```

8. From workstation, recreate the **finance-server1** instance. Source the **architect1-finance** credentials.

```
[student@workstation ~(developer1-finance)]$ source ~/architect1-finance-rc
[student@workstation ~(architect1-finance)]$ openstack server create \
--image rhe17 \
--nic net-id=finance-network1 \
--key-name example-keypair \
--flavor default \
--availability-zone nova:compute0.overcloud.example.com \
--wait finance-server1
...output omitted...
```

9. Log in to **compute0** as the **root** user and list the network interfaces.

9.1. Connect using SSH to **compute0**.

```
[student@workstation ~(architect1-finance)]$ ssh root@compute0
[root@compute0 ~]#
```

9.2. List the network interfaces.

```
[root@compute0 ~]# ip link
...output omitted...
13: br-trunk: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN mode DEFAULT qlen 1000
    link/ether 52:54:00:01:00:0c brd ff:ff:ff:ff:ff:ff
45: tap551c4b7a-4e: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc pfifo_fast
    master ovs-system state UNKNOWN mode DEFAULT qlen 1000
    link/ether fe:16:3e:c9:87:01 brd ff:ff:ff:ff:ff:ff
46: vxlan_sys_4789: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 65470 qdisc noqueue
    master ovs-system state UNKNOWN mode DEFAULT qlen 1000
    link/ether e6:ae:6c:b6:62:39 brd ff:ff:ff:ff:ff:ff
```

In the command output you now see only the TAP device.

9.3. List the Linux bridges.

```
[root@compute0 ~]# brctl show
bridge name bridge id          STP enabled interfaces
```

There are no Linux bridges.

9.4. List the ports on the OVS integration bridge.

```
[root@compute0 ~]# ovs-vsctl list-ports br-int
```

```
int-br-ex
patch-tun
tap551c4b7a-4e
```

The TAP device is now connected directly to the OVS integration bridge. This illustrates that the Linux bridge is only required to support iptables.

Log out of **compute0**.

10. Delete the **finance-server1** instance.

```
[student@workstation ~ (architect1-finance)]$ openstack server delete finance-server1
```

11. On **compute0**, revert your changes to **/etc/neutron/plugins/ml2/openvswitch_agent.ini**, and then restart Neutron services.
 - 11.1. Log in to **compute0** as **root** user. Revert the firewall driver to use **neutron.agent.linux.iptables_firewall.OVSHybridIptablesFirewallDriver**.

```
...output omitted...
[securitygroup]
#
# From neutron.ml2.ovs.agent
#
# Driver for security groups firewall in the L2 agent (string value)
#firewall_driver = <None>
firewall_driver =
neutron.agent.linux.iptables_firewall.OVSHybridIptablesFirewallDriver
..output omitted...
```

- 11.2. Restart Neutron services.

```
[root@compute0 ~]# systemctl restart neutron*
[root@compute0 ~]# exit
[student@workstation ~ (architect1-finance)]$
```

Cleanup

From **workstation**, run the **lab agents-12-13 cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab agents-12-13 cleanup
```

This concludes the guided exercise.

Administering the DHCP Agent

Objective

After completing this section, students should be able to administer the OpenStack Networking DHCP agent.

Implementing the Neutron DHCP Agent

The DHCP agent uses the message queue to communicate with Neutron. It requires L2 connectivity with the tenant network. This connectivity is provided by the L2 agent, which must be installed along with the DHCP agent. The agent can be deployed alongside any other OpenStack service and running it is optional. Multiple agents can be used in the same deployment.

There is one DHCP Agent per controller. It is not responsible for assigning or configuring IP addresses. When a subnet is created, the DHCP Agent is responsible for creating the Linux namespace for that subnet. When a subnet is deleted, the DHCP agent removes the Linux namespace of that subnet. When a Linux namespace is created, the DHCP Agent starts the dnsmasq process for that namespace, and it stops the dnsmasq process when a Linux namespace is removed.

Neutron DHCP Agent Configuration

The `/etc/neutron/dhcp_agent.ini` configuration file is used to configure the DHCP agent. There are many configuration options available.

DHCP Agent Configuration Options

Configuration Option	Description
<code>interface_driver = neutron.agent.linux.interface.OVSIInterfaceDriver</code>	The driver used to manage the virtual interface
<code>resync_interval = 30</code>	The number of seconds between each attempt to resync the DHCP Agent with Neutron to recover from RPC errors
<code>dhcp_driver = neutron.agent.linux.dhcp.Dnsmasq</code>	The driver used to manage the DHCP server
<code>dnsmasq_config_file =/etc/dnsmasq-ironic.conf</code>	Overrides the default dnsmasq settings
<code>enable_isolated_metadata = True</code>	The DHCP server appends specific host routes to the DHCP request to assist with providing metadata support on isolated networks
<code>enable_metadata_network = False</code>	Allows for serving metadata requests from a dedicated network whose CIDR is 169.254.169.254/16

High Availability for DHCP

In OpenStack, it is possible to make DHCP agents scalable and highly available. The agent management and agent alias extensions are invoked in this configuration.

Use the **neutron ext-list** command to verify that these extensions are enabled.

```
[stack@director ~]$ neutron ext-list -c name -c alias
+-----+-----+
| name           | alias      |
+-----+-----+
...output omitted...
| agent          | agent      |
| Subnet Allocation | subnet_allocation |
| DHCP Agent Scheduler | dhcp_agent_scheduler |
...output omitted...
+-----+-----+
```

To understand the concept of a highly available DHCP agent please consider the following setup.

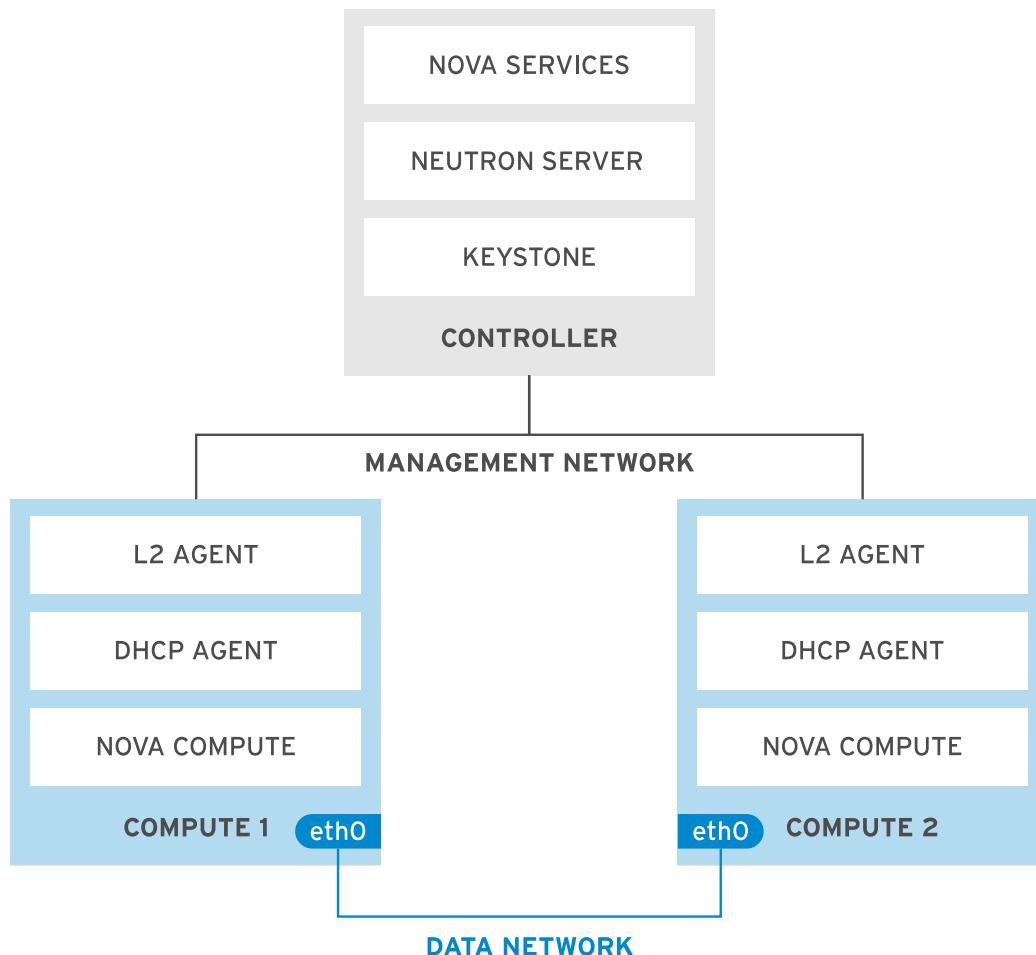


Figure 2.3: Highly available DHCP

Host Configuration

Host	Purpose
Controller	Runs Networking, Compute, and Identity Services. Must have at least 1 NIC connected to the management network
Compute_1	Runs nova-compute, the L2 agent, and the DHCP agent
Compute_2	Runs nova-compute, the L2 agent, and the DHCP agent

Controller Configuration

- Set the `/etc/neutron/neutron.conf` parameter `allow_overlapping_ips` to `True`. Set the `agent_down_time` parameter to `5`.
- Set the `/etc/neutron/plugins/linuxbridge/linuxbridge_conf.ini` parameter `network_vlan_ranges` to `1000:2999`. Verify the SQL connection port as `3306`, and the `retry_interval` is `2`.

Compute Configuration

- On all compute nodes, verify that the `/etc/neutron/neutron.conf` parameters `rabbit_host` and `rabbit_password` are correct.
- The `/etc/neutron/plugins/linuxbridge/linuxbridge_conf.ini` should match that configured above on the controller node.
- Verify that the `/etc/nova/nova.conf` parameter `neutron_admin_auth_url` is enabled on port `35357`, that the URL-embedded version number is `2.0`, and the `neutron_url` has the correct IP address and port.
- Set the `/etc/neutron/dhcp_agent.ini` parameter `interface_driver = neutron.agent.linux.interface.BridgeInterfaceDriver`.

Administering the DHCP Agent

The following steps outline the process for administering the DHCP agent.

- Create a network and a subnet with *no* DHCP
- Create a network and a subnet with DHCP
- Using `ip netns list`, look at the different namespaces that have been created
- Show the `ovs-bridge` bridge information associated with the TAP device
- Locate the `dnsmasq` process associated with the namespace

References

Further information is available in the chapter on OpenStack networking concepts in the *Red Hat OpenStack Platform 10 Networking Guide* at

<https://access.redhat.com/documentation/en/red-hat-openstack-platform/>

Guided Exercise: Administering the DHCP Agent

In this exercise you will view networks, the DHCP agents and namespaces, and the dnsmasq process.

Outcomes

You should be able to:

- Create a centralized router.
- Create a network and show the DHCP network namespace and the dnsmasq process.
- Create a network without DHCP and prove the lack of an IP address.

Before you begin

Log in to **workstation** as **student** using **student** as the password.

On **workstation**, run the **lab agents-dhcp setup** command. This script verifies that the overcloud nodes are accessible and running the correct OpenStack services.

```
[student@workstation ~]$ lab agents-dhcp setup
```

Steps

1. Source the **developer1-research-rc** credential file, create the network called **research-network1**. Then create the subnet **subnet-research1**.
 - 1.1. Source the **developer1-research-rc** credential file.

```
[student@workstation ~]$ source ~/developer1-research-rc  
[student@workstation ~(developer1-research)]$
```

- 1.2. Create the network called **research-network1**.

```
[student@workstation ~(developer1-research)]$ openstack network create \  
research-network1  
...output omitted...
```

- 1.3. Create the subnet **research-subnet1**. The subnet range is **192.168.1.0/24**. Use **172.25.250.254** for the DNS server.

```
[student@workstation ~(developer1-research)]$ openstack subnet create \  
--network research-network1 \  
--subnet-range=192.168.1.0/24 \  
--dns-nameserver=172.25.250.254 \  
--dhcp research-subnet1  
...output omitted...
```

2. Create a centralized router named **research-legacy-router1**. Attach the **research-legacy-router1** router to the external and internal networks.

- 2.1. As **architect1** with the **admin** role, create the **research-legacy-router1** centralized router.

```
[student@workstation ~(developer1-research)]$ source ~/architect1-research-rc
[student@workstation ~(architect1-research)]$ neutron router-create \
--distributed False research-legacy-router1
+-----+-----+
| Field | Value |
+-----+-----+
| admin_state_up | True
| availability_zone_hints |
| availability_zones |
| created_at | 2017-09-13T14:48:32Z
| description |
| distributed | False
| external_gateway_info |
| flavor_id |
| ha |
| id | af8f9cac-90ae-46ba-9df8-7d7f08973a81
| name | research-legacy-router1
| project_id | 6b2eb5c2e59743b9b345ee54a7f87321
| revision_number | 3
| routes |
| status | ACTIVE
| tenant_id | 6b2eb5c2e59743b9b345ee54a7f87321
| updated_at | 2017-09-13T14:48:32Z
+-----+-----+
```

- 2.2. As **developer1**, attach **research-legacy-router1** to the external network **provider-datacentre**.

```
[student@workstation ~(architect1-research)]$ source ~/developer1-research-rc
[student@workstation ~(developer1-research)]$ neutron router-gateway-set \
research-legacy-router1 provider-datacentre
Set gateway for router research-router1
```

- 2.3. Attach **research-legacy-router1** to the internal subnet **research-subnet1**.

```
[student@workstation ~(developer1-research)]$ openstack router add \
subnet research-legacy-router1 research-subnet1
```

3. Review the **DHCP** processes for **research-network1**. In Red Hat OpenStack Platform, DHCP services are provided by **dnsmasq**.

- 3.1. Retrieve the ID of **research-network1**.

```
[student@workstation ~(developer1-research)]$ openstack network show \
research-network1 -c id
+-----+-----+
| Field | Value |
+-----+-----+
| id | 16df85f4-367d-4a13-a806-df7d64e4c1e7 |
+-----+-----+
```

- 3.2. Log in to **controller0** using the **heat-admin** user. Using the **ps** command, and the ID of **research-network1**, list the DHCP processes. Note the TAP device that the **dnsmasq** process listens on.

```
[student@workstation ~(developer1-research)]$ ssh heat-admin@controller0
[heat-admin@controller0 ~]$ ps -ef | \
grep 16df85f4-367d-4a13-a806-df7d64e4c1e7
nobody    16074      1  0 Sep13 ?          00:00:00 dnsmasq
--no-hosts --no-resolv --strict-order --except-interface=lo
--pid-file=/var/lib/neutron/dhcp/16df85f4-367d-4a13-a806-df7d64e4c1e7.pid
--dhcp-hostsfile=/var/lib/neutron/dhcp/16df85f4-367d-4a13-a806-df7d64e4c1e7/host
--addn-hosts=/var/lib/neutron/dhcp/16df85f4-367d-4a13-a806-df7d64e4c1e7/
addn_hosts
--dhcp-optspath=/var/lib/neutron/dhcp/16df85f4-367d-4a13-a806-df7d64e4c1e7/opts
--dhcp-leasefile=/var/lib/neutron/dhcp/16df85f4-367d-4a13-a806-df7d64e4c1e7/
leases
--dhcp-match=set:ipxe,175
--bind-interfaces
--interface=tap17c56c38-b4
--dhcp-range=set:tag0,192.168.11.0,static,86400s
--dhcp-option-force=option:mtu,1446
--dhcp-lease-max=256 --conf-file= --domain=openstacklocal
```

4. Review the DHCP namespace for **research-network1**.
- 4.1. Use the **ip netns** command to list the DHCP namespaces. Find the **qdhcp** namespace for **research-network1** by comparing the **ID** retrieved in the previous step with the **ID** of the namespaces.

```
[heat-admin@controller0 ~]$ ip netns
qdhcp-16df85f4-367d-4a13-a806-df7d64e4c1e7
snat-152de7f4-ea63-4b12-b50a-1861f15a2840
qrouter-152de7f4-ea63-4b12-b50a-1861f15a2840
qrouter-be5f5de3-ce84-4701-889f-1bf616434e44
qrouter-7799dd96-6281-4316-a079-1839c3bcd362
qrouter-7ee5fe29-ddf7-424d-aea9-f5e04f02d549
qdhcp-c1c50c02-af95-45a0-854b-1041a685fbba
qdhcp-0b82e4eb-77d4-4016-bc15-a575e39f4842
qdhcp-3cd52ebc-c017-4531-97ed-160d13507285
```

- 4.2. Execute the **ip address** command inside the **research-network1** namespace. Note the assigned IP address and the TAP device used by **dnsmasq**. Then exit from **controller0**.

```
[heat-admin@controller0 ~]$ sudo ip -n \
qdhcp-16df85f4-367d-4a13-a806-df7d64e4c1e7 address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
23: tap17c56c38-b4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue
    state UNKNOWN qlen 1000
    link/ether fa:16:3e:f2:08:ac brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.2/24 brd 192.168.1.255 scope global tap17c56c38-b4
        valid_lft forever preferred_lft forever
```

```
inet6 fe80::f816:3eff:fe:8ac/64 scope link  
      valid_lft forever preferred_lft forever  
[heat-admin@controller0 ~]$ exit  
[student@workstation ~(developer1-research)]$
```

5. On **workstation** as **developer1** in the **research** project, create a new server instance. Use the **default** flavor, the **example-keypair** key pair, the **research-network1** network, and the **rhel7** image. Name the server **research-server1**.

```
[student@workstation ~(developer1-research)]$ openstack server create \  
--flavor default \  
--key-name example-keypair \  
--nic net-id=research-network1 \  
--image rhel7 \  
--wait research-server1  
...output omitted...
```

6. Show the IP address allocated by DHCP using the **dnsmasq** process.

- 6.1. Using the **openstack subnet show** command, show the details for **research-subnet1**. Note the IP range in the **allocation_pools** attribute.

```
[student@workstation ~(developer1-research)]$ openstack subnet show \  
research-subnet1  
+-----+-----+  
| Field | Value |  
+-----+-----+  
| allocation_pools | 192.168.1.2-192.168.1.254 |  
| cidr | 192.168.1.0/24 |  
...output omitted...
```

- 6.2. Now run the **openstack server list** command. Note the IP address of the instance **research-server1**.

```
[student@workstation ~(developer1-research)]$ openstack server list \  
-c Name -c Networks  
+-----+-----+  
| Name | Networks |  
+-----+-----+  
| research-server1 | research-network1=192.168.1.N |  
+-----+-----+
```

7. Create a second network called **research-network2**. Then create a subnet called **research-subnet2**.

- 7.1. Create the network called **research-network2**

```
[student@workstation ~(developer1-research)]$ openstack network create \  
research-network2  
...output omitted...
```

- 7.2. Create the subnet **research-subnet2**. The network range is **192.168.2.0/24**. Use **172.25.250.254** as the DNS server. Ensure that **dhcp** is not used.

```
[student@workstation ~(developer1-research)]$ openstack subnet create \
--network research-network2 \
--subnet-range=192.168.2.0/24 \
--dns-nameserver=172.25.250.254 \
--no-dhcp research-subnet2
+-----+-----+
| Field      | Value   |
+-----+-----+
| allocation_pools | 192.168.2.2-192.168.2.254 |
| cidr       | 192.168.2.0/24 |
| created_at  | 2017-09-14T10:02:43Z |
| description |          |
| dns_nameservers | 172.25.250.254 |
| enable_dhcp  | False   |
| gateway_ip   | 192.168.2.1 |
| headers      |          |
| host_routes  |          |
| id          | 50af94f-69f4-4921-9be3-d4a593145dab |
| ip_version   | 4       |
| ipv6_address_mode | None   |
| ipv6_ra_mode  | None   |
| name         | research-subnet2 |
| network_id   | 648d02c7-71f4-4bc5-a3f6-8bfedb04f858 |
| project_id   | 81b2aa8cadea447d92ec78da6076b985 |
| project_id   | 81b2aa8cadea447d92ec78da6076b985 |
| revision_number | 2     |
| service_types | []    |
| subnetpool_id | None   |
| updated_at   | 2017-09-14T10:02:43Z |
+-----+-----+
```

7.3. Attach **research-legacy-router1** to the internal **research-subnet2** subnet.

```
[student@workstation ~(developer1-research)]$ openstack router add \
subnet research-legacy-router1 research-subnet2
```

8. Create a new server instance. Use the **default** flavor, the **example-keypair** key pair, the **research-network2** network, and the **rhel7** image. Name the instance **research-server2**.

```
[student@workstation ~(developer1-research)]$ openstack server create \
--flavor default \
--key-name example-keypair \
--nic net-id=research-network2 \
--image rhel7 \
--wait research-server2
...output omitted...
```

9. Access the console of **research-server2** and compare the IP configuration of the actual server to the configuration in OpenStack. Use the **ip address** command to manually configure an IP address for **research-server2**.

9.1. Obtain the console URL for **research-server2** and open the URL in Firefox.

```
[student@workstation ~(developer1-research)]$ openstack console url show \
-f json research-server2
```

```
{
  "url": "http://172.25.250.50:6080/vnc_auto.html?token=a1dd...096b",
  "type": "novnc"
}
[student@workstation ~](developer1-research)]$ firefox \
http://172.25.250.50:6080/vnc_auto.html?token=a1dd...096b
```



Note

You may also want to view the console log as the instance boots because the instance will hang as cloud-init times out with no network connection:

```
[student@workstation ~](developer1-research)]$ openstack console log \
show research-server2
[    0.000000] Initializing cgroup subsys cpuset
...output omitted...
```

- 9.2. After the **research-server2** instance has booted, log in as user **root** with the password **redhat**. Run the **ip address** command. Interface **eth0** has no configured IP address.

```
[root@localhost ~]# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
  link/loopback 00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
      valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
      valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
  qlen 1000
  link/ether 52:54:00:00:fa:fe brd ff:ff:ff:ff:ff:ff
```

- 9.3. Compare the IP configuration with **research-server1** by opening its console and running the **ip address** command. Observe that interface **eth0** on instance **research-server1** has an IP address.

```
[root@research-server1 ~]# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
  link/loopback 00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
      valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
      valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
  qlen 1000
  link/ether 52:54:00:00:fa:fe brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.11/24 brd 192.168.1.255 scope global eth0
      valid_lft forever preferred_lft forever
    inet6 fe80::5054:ff:fe00:faf6/64 scope link
      valid_lft forever preferred_lft forever
```

- 9.4. OpenStack has assigned an IP address to **research-server2** from the allocated_pools of **research-subnet2** regardless that no DHCP service is running. Back on **workstation**, you can see the assigned IP address.

```
[student@workstation ~](developer1-research)]$ openstack server list \
-c Name -c Networks
+-----+-----+
| Name | Networks |
+-----+-----+
| research-server2 | research-network2=192.168.2.P |
| research-server1 | research-network1=192.168.1.N |
+-----+-----+
```

- 9.5. It is possible to configure an IP address for instance **research-server2**. Return to the console of **research-server2** in the Horizon dashboard. Using the **ip address** command, manually configure an IP address.

```
[root@localhost ~]# ip address add 192.168.2.P dev eth0
[root@localhost ~]# ip address
...output omitted...
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
qlen 1000
    link/ether 52:54:00:00:fa:fe brd ff:ff:ff:ff:ff:ff
        inet 192.168.2.P/24 brd 192.168.1.255 scope global eth0
...output omitted...
```

Cleanup

From **workstation**, run the **lab agents-dhcp cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab agents-dhcp cleanup
```

This concludes the guided exercise.

Managing OpenStack Networking Agents

Objectives

After completing this section, students should be able to:

- Describe the Load Balancer as a Service (LBaaS) agent.
- Describe the Metadata agent.
- Describe the Metering agent.

Neutron LBaaS Agent

Neutron LBaaS v2 allows users to create load balancers to sit in front of groups of instances serving the same role. A load balancing service consists of a load balancer, a listener, a pool, pool members, and a health monitor. The LBaaS agent performs the provisioning of load balancers when requests are submitted to the Neutron API. The load balancing capability is currently implemented using HAProxy.

Neutron Metadata Agent

To perform instance customization, cloud-init requires metadata unique to the instance. The metadata can include a hostname, IP address, instance id, or SSH public key to inject into the **authorized_keys** file for the cloud-user account. Neutron enables instances to reach the metadata server by providing the metadata-proxy. In a routed network, requests to 169.254.169.254 that reach the I3 agent are redirected by iptables to an instance of HAProxy listening in each router namespace. HAProxy passes incoming requests to a metadata-proxy process, which then forwards the requests over RPC to the Neutron Metadata agent. The Metadata agent then forwards the requests to Nova for servicing, and responses are send back through the same path.

For instances attached to an isolated network, the Neutron DHCP agent can be configured to support metadata operations. When configured, a host route DHCP option is passed to the instance, allowing it to connect to a metadata-proxy process running in the DHCP namespace. The remainder of the communication is the same as in a routed network.

Neutron Metering Agent

The Neutron Metering agent enables charge-back based on bandwidth usage. It uses a driver to insert iptables rules in routers to label outgoing traffic. Once a router has metering rules added, the agent polls the router regularly to collect traffic counters. The traffic information is collected into a report which includes items such as the label, project ID, number of packets, number of bytes, and first and last update times.

Managing OpenStack Networking Agents

The following steps outline the process for examining the Neutron metadata agent.

1. Log in to a compute node.
2. Check the status of the neutron-metadata-agent service.
3. View the metadata agent processes.

4. Look at the directives set in the metadata agent config file.
5. View the metadata agent log file.
6. Provision an instance.
7. View the instance's metadata requests in the agent log file.



References

Further information is available in the chapter on Configure load-balancing-as-a-service (LBAAS) in the *Red Hat OpenStack Platform 10 Networking Guide* at

| <https://access.redhat.com/documentation/en/red-hat-openstack-platform/>

Guided Exercise: Managing OpenStack Networking Agents

In this exercise, you will:

- Create a load balancer.
- Configure a virtual IP (VIP) for the load balancer.
- Create a load balancer pool.
- Configure a health monitor for the load balancer pool.

Outcomes

You should be able to create a load balancer, with an associated VIP, pool, and health monitor.

Before you begin

Log in to workstation as **student** using **student** as the password. Open a terminal, then run the **lab agents-load-balancer setup** command. This script verifies that the overcloud nodes are accessible and the OpenStack services are running, and sets up the prerequisites for this guided exercise.

```
[student@workstation ~]$ lab agents-load-balancer setup
```

Steps

1. Source the credentials for the **developer1** user in the **finance** project. Create an LBaaS load balancer named **finance-lb1** in the **finance-subnet1** subnet.

```
[student@workstation ~]$ source ~/developer1-finance-rc
[student@workstation ~(developer1-finance)]$ neutron lbaas-loadbalancer-create \
--name finance-lb1 finance-subnet1
Created a new loadbalancer:
+-----+-----+
| Field | Value |
+-----+-----+
| admin_state_up | True |
| description | |
| id | bd4d4e61-57bb-4e88-9938-a064843e1d1b |
| listeners | |
| name | finance-lb1 |
| operating_status | OFFLINE |
| pools | |
| provider | haproxy |
| provisioning_status | PENDING_CREATE |
| tenant_id | 0ed0cb3b180d46afb510b12a09675dd8 |
| vip_address | 192.168.1.11 |
| vip_port_id | 7666f902-e8bb-4e47-998b-179856e3eb01 |
| vip_subnet_id | a4781a7a-734c-4e3d-8d7d-23598f18a4f2 |
+-----+-----+
```

2. Create an LBaaS listener named **finance-listener1** attached to the **finance-lb1** load balancer using HTTP on port 80.

```
[student@workstation ~ (developer1-finance)]$ neutron lbaas-listener-create \
--loadbalancer finance-lb1 \
--protocol HTTP \
--protocol-port 80 \
--name finance-listener1
Created a new listener:
+-----+
| Field | Value |
+-----+
| admin_state_up | True |
| connection_limit | -1 |
| default_pool_id | |
| default_tls_container_ref | |
| description | |
| id | 862bda78-7b84-47ac-a342-ab142ef43f0d |
| loadbalancers | {"id": "bd4d4e61-57bb-4e88-9938-a064843e1d1b"} |
| name | finance-listener1 |
| protocol | HTTP |
| protocol_port | 80 |
| sni_container_refs | |
| tenant_id | 0ed0cb3b180d46afb510b12a09675dd8 |
+-----+
```

3. Create an LBaaS pool named **finance-pool1** using the **ROUND_ROBIN** load-balancing algorithm, and serving the **finance-listener1** listener over HTTP.

```
[student@workstation ~ (developer1-finance)]$ neutron lbaas-pool-create \
--lb-algorithm ROUND_ROBIN \
--listener finance-listener1 \
--protocol HTTP \
--name finance-pool1
Created a new pool:
+-----+
| Field | Value |
+-----+
| admin_state_up | True |
| description | |
| healthmonitor_id | |
| id | f94e78b4-70a0-460a-a5d8-649168bfb36e |
| lb_algorithm | ROUND_ROBIN |
| listeners | {"id": "862bda78-7b84-47ac-a342-ab142ef43f0d"} |
| loadbalancers | {"id": "bd4d4e61-57bb-4e88-9938-a064843e1d1b"} |
| members | |
| name | finance-pool1 |
| protocol | HTTP |
| session_persistence | |
| tenant_id | 0ed0cb3b180d46afb510b12a09675dd8 |
+-----+
```

4. Add the **finance-server1** instance as a member of the **finance-pool1** pool, using port 80.

```
[student@workstation ~ (developer1-finance)]$ openstack server show \
-c addresses -f value finance-server1
finance-network1=192.168.1.1
[student@workstation ~ (developer1-finance)]$ neutron lbaas-member-create \
--subnet finance-subnet1 \
--address 192.168.1.1 \
```

```
--protocol-port 80 \
finance-pool1
Created a new member:
+-----+
| Field      | Value
+-----+
| address    | 192.168.1.P
| admin_state_up | True
| id         | c59f34be-4897-46b5-90bd-0eef001e7cc
| name       |
| protocol_port | 80
| subnet_id   | a4781a7a-734c-4e3d-8d7d-23598f18a4f2
| tenant_id   | 0ed0cb3b180d46afb510b12a09675dd8
| weight      | 1
+-----+
```

5. Add the **finance-server2** instance as a member of the **finance-pool1** pool, using port 80.

```
[student@workstation ~ (developer1-finance)]$ openstack server show \
-c addresses -f value finance-server2
finance-network1=192.168.1.Q
[student@workstation ~ (developer1-finance)]$ neutron lbaas-member-create \
--subnet finance-subnet1 \
--address 192.168.1.Q \
--protocol-port 80 \
finance-pool1
Created a new member:
+-----+
| Field      | Value
+-----+
| address    | 192.168.1.Q
| admin_state_up | True
| id         | e41f6ace-9436-4393-b6c9-084906255bf0
| name       |
| protocol_port | 80
| subnet_id   | a4781a7a-734c-4e3d-8d7d-23598f18a4f2
| tenant_id   | 0ed0cb3b180d46afb510b12a09675dd8
| weight      | 1
+-----+
```

6. Add an LBaaS health monitor named **finance-health1** for the **finance-pool1** pool using HTTP. Configure a delay of 5, a timeout of 2, and max-retries of 3.

```
[student@workstation ~ (developer1-finance)]$ neutron lbaas-healthmonitor-create \
--delay 5 \
--type HTTP \
--max-retries 3 \
--timeout 2 \
--pool finance-pool1 \
--name finance-health1
Created a new healthmonitor:
+-----+
| Field      | Value
+-----+
| admin_state_up | True
| delay        | 5
| expected_codes | 200
| http_method  | GET
| id           | 0daa0ff9-0fb1-40ff-b780-fba6aee17a40
+-----+
```

max_retries	3
max_retries_down	3
name	finance-health1
pools	{"id": "f94e78b4-70a0-460a-a5d8-649168bfb36e"}
tenant_id	0ed0cb3b180d46afb510b12a09675dd8
timeout	2
type	HTTP
url_path	/

7. Find the **finance-lb1** load balancer VIP port ID.

[student@workstation ~ (developer1-finance)]\$ neutron lbaas-loadbalancer-show \ finance-lb1	
Field	Value
admin_state_up	True
description	
id	bd4d4e61-57bb-4e88-9938-a064843e1d1b
listeners	{"id": "862bda78-7b84-47ac-a342-ab142ef43f0d"}
name	finance-lb1
operating_status	ONLINE
pools	{"id": "f94e78b4-70a0-460a-a5d8-649168bfb36e"}
provider	haproxy
provisioning_status	ACTIVE
tenant_id	0ed0cb3b180d46afb510b12a09675dd8
vip_address	192.168.1.N
vip_port_id	7666f902-e8bb-4e47-998b-179856e3eb01
vip_subnet_id	a4781a7a-734c-4e3d-8d7d-23598f18a4f2

8. Find an available floating IP, then associate it with the load balancer VIP port ID.

[student@workstation ~ (developer1-finance)]\$ neutron floatingip-list \ -c id -c floating_ip_address -c port_id		
id	floating_ip_address	port_id
5e0342ec-f7b2-4148-bef8-27b032f93169	172.25.250.N	
...output omitted...		
[student@workstation ~ (developer1-finance)]\$ neutron floatingip-associate \ 5e0342ec-f7b2-4148-bef8-27b032f93169 7666f902-e8bb-4e47-998b-179856e3eb01		
Associated floating IP 5e0342ec-f7b2-4148-bef8-27b032f93169		

9. Test the load balancer using the **curl** command.

```
[student@workstation ~ (developer1-finance)]$ curl http://172.25.250.N
finance-server1
[student@workstation ~ (developer1-finance)]$ curl http://172.25.250.N
finance-server2
[student@workstation ~ (developer1-finance)]$ curl http://172.25.250.N
finance-server1
[student@workstation ~ (developer1-finance)]$ curl http://172.25.250.N
finance-server2
```

Cleanup

From **workstation**, run the **lab agents-load-balancer cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab agents-load-balancer cleanup
```

This concludes the guided exercise.

Lab: Managing OpenStack Networking Agents

In this lab, you will:

- Launch two database servers.
- Configure a load balancer with a virtual IP address (VIP).
- Configure the load balancer using the LEAST_CONNECTIONS policy.
- Configure a health monitor for the load balancer.
- Verify the LEAST_CONNECTIONS policy.

Outcomes

You should be able to create customized instances fronted by a load balancer.

Before you begin

Log in to workstation as **student** using **student** as the password. On **workstation**, run the **lab agents-managing-networking setup** command. This script verifies that the overcloud nodes are accessible and that the OpenStack services are running, and sets up the prerequisites for this lab.

```
[student@workstation ~]$ lab agents-managing-networking setup
```



Note

This lab uses databases behind the load balancer to give you enough time to complete the exercise without sessions timing out. Using Apache HTTPD would have been simpler but the HTTP sessions by their nature are too short lived.

Steps

1. Create a cloud-init user-data file that installs the *mariadb* and *mariadb-server* packages, enable and start the mariadb service, and allow remote logins to the database from anywhere. The SQL command to allow remote logins is shown below.


```
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY 'redhat' WITH GRANT OPTION;
```
2. As **operator1** in the **production** project, launch an instance named **production-server1** using the standard resources available. Include the user-data file created in the previous step.
3. As **operator1** in the **production** project, launch an instance named **production-server2** using the standard resources available. Include the user-data file created earlier.
4. Create an LBaaS load balancer named **production-lb1** in the **production-subnet1** subnet.

5. Create an LBaaS listener named **production-listener1** attached to the **production-lb1** load balancer using TCP on port 3306.
6. Create an LBaaS pool named **production-pool1** using the **LEAST_CONNECTIONS** load-balancing algorithm, and serving the **production-listener1** listener over TCP.
7. Add the **production-server1** instance as a member of the **production-pool1** pool, using port 3306.
8. Add the **production-server2** instance as a member of the **production-pool1** pool, using port 3306.
9. Add an LBaaS health monitor named **production-health1** for the **production-pool1** pool using TCP. Configure a delay of 5, a timeout of 2, and max-retries of 3.
10. Find the **production-lb1** load balancer VIP port ID.
11. Find an available floating IP, then associate it with the load balancer VIP port ID.
12. Confirm that the load balancer is functioning. Using the **mysql** command, connect to the load balancer floating IP.
13. Confirm that the load balancer is using the **LEAST_CONNECTIONS** method to distribute new connections.
 - Suspend the **production-server2** instance. The load balancer takes approximately 10 seconds to mark the pool member as unavailable.
 - Open three new terminals on workstation. In two of them, connect to the load balancer floating IP using the **mysql** command, then execute the query **select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES where VARIABLE_NAME='hostname' \G**. This confirms that you are connected to **production-server1**.
 - Resume the **production-server2** instance.
 - From the third new terminal, connect to the load balancer floating IP using the **mysql** command, then execute the query **select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES where VARIABLE_NAME='hostname' \G**. This confirms that you are connected to **production-server2**.
 - From your original terminal, connect to the load balancer floating IP using the **mysql** command, then execute the query **select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES where VARIABLE_NAME='hostname' \G**. This confirms that you are connected to **production-server2**.

Evaluation

From **workstation**, run the **lab agents-managing-networking grade** command to confirm the success of this exercise. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab agents-managing-networking grade
```

Cleanup

From **workstation**, run the **lab agents-managing-networking cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab agents-managing-networking cleanup
```

This concludes the lab.

Solution

In this lab, you will:

- Launch two database servers.
- Configure a load balancer with a virtual IP address (VIP).
- Configure the load balancer using the LEAST_CONNECTIONS policy.
- Configure a health monitor for the load balancer.
- Verify the LEAST_CONNECTIONS policy.

Outcomes

You should be able to create customized instances fronted by a load balancer.

Before you begin

Log in to workstation as **student** using **student** as the password. On **workstation**, run the **lab agents-managing-networking setup** command. This script verifies that the overcloud nodes are accessible and that the OpenStack services are running, and sets up the prerequisites for this lab.

```
[student@workstation ~]$ lab agents-managing-networking setup
```



Note

This lab uses databases behind the load balancer to give you enough time to complete the exercise without sessions timing out. Using Apache HTTPD would have been simpler but the HTTP sessions by their nature are too short lived.

Steps

1. Create a cloud-init user-data file that installs the *mariadb* and *mariadb-server* packages, enable and start the mariadb service, and allow remote logins to the database from anywhere. The SQL command to allow remote logins is shown below.

```
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY 'redhat' WITH GRANT OPTION;
```

```
#!/usr/bin/bash
yum -y install mariadb mariadb-server
systemctl enable mariadb --now
mysql -u root -e "GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY 'redhat'
WITH GRANT OPTION;"
```

2. As **operator1** in the **production** project, launch an instance named **production-server1** using the standard resources available. Include the user-data file created in the previous step.

- 2.1. Source the **/home/student/operator1-production-rc** credential file.

```
[student@workstation ~]$ source ~/operator1-production-rc
```

```
[student@workstation ~(operator1-production)]$
```

- 2.2. Create the **production-server1** instance.

```
[student@workstation ~(operator1-production)]$ openstack server create \
--image rhel7 \
--nic net-id=production-network1 \
--key-name example-keypair \
--flavor default \
--user-data ~/mariadb.sh \
--wait production-server1
...output omitted...
```

3. As **operator1** in the **production** project, launch an instance named **production-server2** using the standard resources available. Include the user-data file created earlier.

```
[student@workstation ~(operator1-production)]$ openstack server create \
--image rhel7 \
--nic net-id=production-network1 \
--key-name example-keypair \
--flavor default \
--user-data ~/mariadb.sh \
--wait production-server2
...output omitted...
```

4. Create an LBaaS load balancer named **production-lb1** in the **production-subnet1** subnet.

```
[student@workstation ~(operator1-production)]$ neutron lbaas-loadbalancer-create \
--name production-lb1 production-subnet1
Created a new loadbalancer:
+-----+-----+
| Field | Value |
+-----+-----+
| admin_state_up | True |
| description | |
| id | 05efbde7-602e-4c65-86f3-86f40e975027 |
| listeners | |
| name | production-lb1 |
| operating_status | OFFLINE |
| pools | |
| provider | haproxy |
| provisioning_status | PENDING_CREATE |
| tenant_id | 6f1546beb50f4cbc9e9feb4f23ed7f55 |
| vip_address | 192.168.1.N |
| vip_port_id | 8947ab94-87f3-4709-81b8-9b7cd4dfe6c5 |
| vip_subnet_id | c21c0968-561f-49f5-acdf-46233989cd82 |
+-----+-----+
```

5. Create an LBaaS listener named **production-listener1** attached to the **production-lb1** load balancer using TCP on port 3306.

```
[student@workstation ~(operator1-production)]$ neutron lbaas-listener-create \
--loadbalancer production-lb1 \
--protocol TCP \
--protocol-port 3306 \
```

```
--name production-listener1
Created a new listener:
+-----+
| Field | Value |
+-----+
| admin_state_up | True
| connection_limit | -1
| default_pool_id |
| default_tls_container_ref |
| description |
| id | f8593321-1a05-42e8-876e-d15f6a0dfd83
| loadbalancers | {"id": "05efbde7-602e-4c65-86f3-86f40e975027"}
| name | production-listener1
| protocol | TCP
| protocol_port | 3306
| sni_container_refs |
| tenant_id | 6f1546beb50f4cbc9e9feb4f23ed7f55
+-----+
```

6. Create an LBaaS pool named **production-pool1** using the **LEAST_CONNECTIONS** load-balancing algorithm, and serving the **production-listener1** listener over TCP.

```
[student@workstation ~(operator1-production)]$ neutron lbaas-pool-create \
--lb-algorithm LEAST_CONNECTIONS \
--listener production-listener1 \
--protocol TCP \
--name production-pool1
+-----+
| Field | Value |
+-----+
| admin_state_up | True
| description |
| healthmonitor_id |
| id | cf76f3c5-c67d-4f55-9cf5-86a16ee5148c
| lb_algorithm | LEAST_CONNECTIONS
| listeners | {"id": "f8593321-1a05-42e8-876e-d15f6a0dfd83"}
| loadbalancers | {"id": "05efbde7-602e-4c65-86f3-86f40e975027"}
| members |
| name | production-pool1
| protocol | TCP
| session_persistence |
| tenant_id | 6f1546beb50f4cbc9e9feb4f23ed7f55
+-----+
```

7. Add the **production-server1** instance as a member of the **production-pool1** pool, using port 3306.

```
[student@workstation ~(operator1-production)]$ neutron lbaas-member-create \
--subnet production-subnet1 \
--address 192.168.1.P \
--protocol-port 3306 \
production-pool1
+-----+
| Field | Value |
+-----+
| address | 192.168.1.P
| admin_state_up | True
| id | 41bc9753-5529-494e-94f8-f931b2fc26c
| name |
| protocol_port | 3306
+-----+
```

subnet_id	c21c0968-561f-49f5-acdf-46233989cd82
tenant_id	6f1546beb50f4cbc9e9feb4f23ed7f55
weight	1
+	

8. Add the **production-server2** instance as a member of the **production-pool1** pool, using port 3306.

```
[student@workstation ~(operator1-production)]$ neutron lbaas-member-create \
--subnet production-subnet1 \
--address 192.168.1.0 \
--protocol-port 3306 \
production-pool1
+-----+
| Field      | Value
+-----+
| address    | 192.168.1.0
| admin_state_up | True
| id         | 720173ff-2044-40f8-972c-660be625bf41
| name       |
| protocol_port | 3306
| subnet_id   | c21c0968-561f-49f5-acdf-46233989cd82
| tenant_id   | 6f1546beb50f4cbc9e9feb4f23ed7f55
| weight      | 1
+-----+
```

9. Add an LBaaS health monitor named **production-health1** for the **production-pool1** pool using TCP. Configure a delay of 5, a timeout of 2, and max-retries of 3.

```
[student@workstation ~(operator1-production)]$ neutron lbaas-healthmonitor-create \
--type TCP \
--delay 5 \
--max-retries 3 \
--timeout 2 \
--pool production-pool1 \
--name production-health1
+-----+
| Field      | Value
+-----+
| admin_state_up | True
| delay       | 5
| id          | ebbd3880-ddcd-4eea-9bec-af6e3ab55f92
| max_retries | 3
| max_retries_down | 3
| name        | production-health1
| pools       | {"id": "cf76f3c5-c67d-4f55-9cf5-86a16ee5148c"} |
| tenant_id   | 6f1546beb50f4cbc9e9feb4f23ed7f55
| timeout     | 2
| type        | TCP
+-----+
```

10. Find the **production-lb1** load balancer VIP port ID.

```
[student@workstation ~(operator1-production)]$ neutron lbaas-loadbalancer-show \
production-lb1
+-----+
| Field      | Value
+-----+
```

admin_state_up	True
description	
id	05efbd7-602e-4c65-86f3-86f40e975027
listeners	{"id": "f8593321-1a05-42e8-876e-d15f6a0dfd83"}
name	production-lb1
operating_status	ONLINE
pools	{"id": "cf76f3c5-c67d-4f55-9cf5-86a16ee5148c"}
provider	haproxy
provisioning_status	ACTIVE
tenant_id	6f1546beb50f4cbc9e9feb4f23ed7f55
vip_address	192.168.1.N
vip_port_id	8947ab94-87f3-4709-81b8-9b7cd4dfe6c5
vip_subnet_id	c21c0968-561f-49f5-acdf-46233989cd82
+-----+	

11. Find an available floating IP, then associate it with the load balancer VIP port ID.

```
[student@workstation ~(operator1-production)]$ neutron floatingip-associate \
036b346d-2509-4623-817b-48bd1d188a15 8947ab94-87f3-4709-81b8-9b7cd4dfe6c5
Associated floating IP 036b346d-2509-4623-817b-48bd1d188a15
```

12. Confirm that the load balancer is functioning. Using the **mysql** command, connect to the load balancer floating IP.

```
[student@workstation ~(operator1-production)]$ mysql -uroot -predhat \
-h 172.25.250.N
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 16
Server version: 5.5.56-MariaDB MariaDB Server

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> \q
Bye
[student@workstation ~(operator1-production)]$
```

13. Confirm that the load balancer is using the LEAST_CONNECTIONS method to distribute new connections.

- Suspend the **production-server2** instance. The load balancer takes approximately 10 seconds to mark the pool member as unavailable.
- Open three new terminals on workstation. In two of them, connect to the load balancer floating IP using the **mysql** command, then execute the query **select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES where VARIABLE_NAME='hostname'\G**. This confirms that you are connected to **production-server1**.
- Resume the **production-server2** instance.
- From the third new terminal, connect to the load balancer floating IP using the **mysql** command, then execute the query **select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES where**

VARIABLE_NAME='hostname' \G. This confirms that you are connected to **production-server2**.

- From your original terminal, connect to the load balancer floating IP using the **mysql** command, then execute the query **select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES where VARIABLE_NAME='hostname' \G.** This confirms that you are connected to **production-server2**.

13.1. Suspend the **production-server2** instance.

```
[student@workstation ~] $ openstack server suspend \
production-server2
```

Wait approximately 10 seconds for the instance to be marked as unavailable by the load balancer.

13.2. Open a new terminal, connect to the load balancer floating IP using the **mysql** command, then execute the query **select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES where VARIABLE_NAME='hostname' \G.** The hostname should be **production-server1**.

```
[student@workstation ~] $ mysql -uroot -predhat -h 172.25.250.N
...output omitted...
MariaDB [(none)]> select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES
where VARIABLE_NAME='hostname' \G
***** 1. row *****
variable_value: production-server1.novalocal
1 row in set (0.00 sec)

MariaDB [(none)]>
```

13.3. Open a second terminal and connect to the load balancer floating IP using the **mysql** command. Execute the query **select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES where VARIABLE_NAME='hostname' \G.** The hostname should be **production-server1**.

```
[student@workstation ~] $ mysql -uroot -predhat -h 172.25.250.N
...output omitted...
MariaDB [(none)]> select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES
where VARIABLE_NAME='hostname' \G
***** 1. row *****
variable_value: production-server1.novalocal
1 row in set (0.00 sec)

MariaDB [(none)]>
```

13.4. From your original terminal, resume the **production-server2** instance.

```
[student@workstation ~] $ openstack server resume \
production-server2
```

13.5. At this time, you should expect the next two connections to be serviced by **production-server2**, as **production-server1** already has two active connections.

Open a third terminal and connect to the load balancer floating IP using the **mysql** command. Execute the query **select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES where VARIABLE_NAME='hostname' \G**. The hostname should be **production-server2**.

```
[student@workstation ~]$ mysql -uroot -predhat -h 172.25.250.N  
...output omitted...  
MariaDB [(none)]> select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES  
where VARIABLE_NAME='hostname'\G  
***** 1. row *****  
variable_value: production-server2.novalocal  
1 row in set (0.00 sec)  
  
MariaDB [(none)]>
```

13.6. Back in your original terminal, connect to the load balancer floating IP using the **mysql** command. Execute the query **select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES where VARIABLE_NAME='hostname' \G**. The hostname should be **production-server2**.

```
[student@workstation ~(operator1-production)]$ mysql -uroot -predhat \  
-h 172.25.250.N  
...output omitted...  
MariaDB [(none)]> select variable_value from INFORMATION_SCHEMA.GLOBAL_VARIABLES  
where VARIABLE_NAME='hostname'\G  
***** 1. row *****  
variable_value: production-server2.novalocal  
1 row in set (0.00 sec)  
  
MariaDB [(none)]>
```

As the next two connections went to **production-server2**, you have proved that the LEAST_CONNECTIONS algorithm is working correctly.

Evaluation

From **workstation**, run the **lab agents-managing-networking grade** command to confirm the success of this exercise. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab agents-managing-networking grade
```

Cleanup

From **workstation**, run the **lab agents-managing-networking cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab agents-managing-networking cleanup
```

This concludes the lab.

Summary

In this chapter, you learned:

- To describe Neutron OpenStack and its components and how to explain Neutron provisioning and modules.
- That the primary objects in any Neutron deployment are an isolated layer 2 broadcast domain, a subnet assigned to the network, a virtual switch port, and a virtual router.
- That Neutron uses plug-ins and agents to implement extra functionality.
- That the metadata server requests are handled by the l3 agent and redirected by iptables to an instance of HAProxy listening in each router namespace.
- That when Open vSwitch is present, an integration bridge is used to connect all the instance TAP ports to the rest of the servers running Neutron.
- That network namespaces are used to provide isolated context for each tenant or project.
- How to troubleshoot the L2 and L3 agents using various commands, log files, and configuration files.
- It is the job of the DHCP agent to create the Linux namespace for subnets.
- That Neutron LBaaS allows users to create load balancers to sit in front of groups of instances serving the same role.
- That for instances attached to an isolated network, the Neutron DHCP agent can be configured to support metadata operations.
- The Neutron Metering agent enables charge-back based on bandwidth usage.



CHAPTER 3

DEPLOYING IPV6 NETWORKS

Overview	
Goal	Deploy IPv6 networks in OpenStack.
Objectives	<ul style="list-style-type: none">Explain IPv6 concepts.Assign IPv6 addresses with SLAAC, DHCPv6, and RA and provide IPv6 routing.Deploy IPv6 addresses in OpenStack.
Sections	<ul style="list-style-type: none">Explaining IPv6 (and Quiz)Assigning IPv6 Addresses (and Guided Exercise)Deploying IPv6 (and Guided Exercise)
Lab	Deploying IPv6 Networks

Explaining IPv6

Objectives

After completing this section, students should be able to:

- Explain IPv6 concepts.
- Compare and contrast the various IPv6 scopes.

IPv6 Overview

IPv6 is the latest version of the Internet Protocol and is intended as the replacement for the IPv4 network protocol. Even though it functions similarly to IPv4, it also provides a number of enhancements and new features for network configuration management and support for future protocol changes.

However, it does have one major difference, as IPv6 protocol utilizes 128-bit addresses. The major problem IPv6 solves is the exhaustion of IPv4 addresses by using a much larger network address space.

These are the following advantages of using IPv6 protocol:

- Larger address space.
- Support for stateless address auto-configuration.
- Built-in security with IPSec.
- Multicast address support is a standard feature of the IPv6 protocol, making the process of communication faster, easier, and more reliable.
- Quality of Service (QoS) is a technology that helps ease the congestion in packet-switched networks. QoS divides network traffic into different classes during times of congestion, and helps in prioritizing information transfer. IPv6 gains QoS advantages over IPv4 when the flow label is used.
- Mobility provides a mechanism for the host to roam around different links without losing connection and its IP address. IPv6 provides support for mobility.

Understanding IPv6 Addresses

An IPv6 address is a 128-bit number, normally expressed as eight colon-separated groups of four hexadecimal nibbles (half-bytes). Each nibble represents four bits of the IPv6 address, so each group represents 16 bits of the IPv6 address.

```
2001:0db8:0000:0010:0000:0000:0000:0001
```

To make it easier to write IPv6 addresses, leading zeros in a colon-separated group do not need to be written. However, at least one nibble must be written in each field. Zeros which follow a nonzero nibble in the group do need to be written.

```
2001:db8:0:10:0:0:0:1
```

Since addresses with long strings of zeros are common, one or more groups of consecutive zeros may be combined with exactly one :: block.

```
2001:db8:0:10::1
```

IPv6 Subnets

A normal IPv6 address is divided into two parts: the *network prefix* and *interface ID*. The network prefix identifies the subnet to which the IPv6 address belongs. The interface ID identifies a particular interface on the subnet. No two IPv6 address on the same subnet can have the same interface ID.

Typically, the network provider allocates a shorter prefix to an organization, such as **2001:db8:1::/48**. This leaves the rest of the network part for assigning subnets from that allocated prefix. For a **2001:db8:1::/48** prefix allocation, that leaves 16 bits for subnets (up to 65536 subnets). Thus a subnet can be created with a CIDR block as **2001:db8:1:4248::/64**. The remaining 64-bit is the interface ID and results in an IPv6 address, such as **2001:db8:1:4248:f816:3eff:fead:8c3f**.

EUI-64 Format in IPv6

One of the IPv6 address advantages over IPv4 is auto-configuration of addresses. This is implemented using IEEE's *Extended Unique Identified (EUI-64)* specification. Using the EUI-64 format, a host automatically assigns itself a unique 64-bit IPv6 interface ID using the network interface MAC address. This eliminates the need for manual configuration, or for an extra server managing DHCP.

The MAC address of a network interface is an unique 48-bit address. The MAC address is reformatted using EUI-64 specification to get a unique interface ID. The first step in this process is to convert 48-bit MAC address to a 64-bit value. The MAC address is divided into two 24 bit halves and then **0xFFEE** hexadecimal value is inserted between these two halves. The first 24-bit half is the *Organizational Unique Identifier (OUI)* and the second 24-bit half is the *Network Interface Controller (NIC)* part.

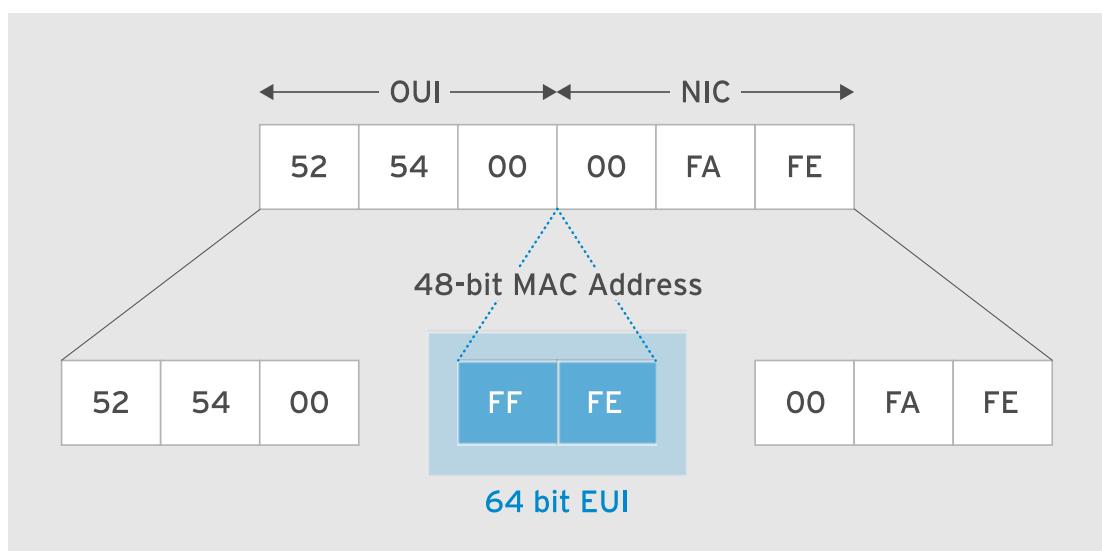


Figure 3.1: Converting MAC address to 64-bit value

The MAC address of a network device is either a universally administered address (UAA) or a locally administered address (LAA). A universally administered address is assigned to the device by the manufacturer and is unique. The second least significant bit in the first octet of the OUI portion is also referred to as the U/L (Universal/Local) bit. The U/L bit identifies how the MAC address of the network device is administered. If the U/L bit is 0, the address is universally administered, while 1 indicates the address is locally administered.

The second step in the process of getting an EUI-64 formatted interface ID is to invert the U/L bit in the OUI portion of the MAC address. The U/L bit is inverted when using an EUI-64 address as an IPv6 interface ID.



Figure 3.2: Inverting the U/L bit of a MAC address

Creating an IPv6 address with the network prefix as **2001:db8::/64**, and with a MAC address of **52:54:00:00:fa:fe**, yields the address **2001:db8::5054:ff:fe00:fafe**.

Types and Categories of IPv6 Addresses

There are three categories of IPv6 addresses. They are classified based on addressing and routing methodologies used to deliver packets:

Unicast

A unicast address is an identifier for a single network interface. When a packet is sent to a unicast address, it is delivered to the interface with the address only. The scope of the address can be **link-local** or **global** scope.

Multicast

A multicast address is an identifier for a group of network interfaces. When a packet is sent to a multicast address, it is delivered to all interfaces identified by the address. Multicast plays a larger role in IPv6 than in IPv4 because there is no broadcast address in IPv6. One key multicast address in IPv6 is **ff02::1**, the all-nodes link-local address. Pinging this address sends traffic to all nodes on the link.

Anycast

An anycast address is an identifier for a group of network interfaces. When a packet is sent to an anycast address, it is delivered to one interface that is closest according to the router protocol.

Link-local Scope Addresses

In IPv4, using **Automatic Private IP Addressing** (APIPA) a host selects an IPv4 address from the **169.254.0.0/16** CIDR block whenever it loses network connectivity. APIPA addressing is also known as **Zeroconf** or **Zero Configuration Networking**. Zeroconf helps to communicate with various hosts on the same network link without requiring to configure special servers such as DHCP or DNS server. One major problem with using APIPA IPv4 address is that it can not be implemented in a wide-area network with complex routing.

The IPv6 link-local unicast address is equivalent to APIPA IPv4 address. Every IPv6 interface automatically configures a link-local address and is used to communicate with all hosts without the requirement of a router. A link-local unicast address in IPv6 is an unroutable address which is used only to talk to hosts on a specific network link. Every network interface on the system is automatically configured with a link-local address on the **fe80::** network. Since every link has an **fe80::/64** network on it, the routing table is not referred to when selecting the outbound interface.

The link used when talking to a link-local address must be specified with a scope identifier at the end of the address. The scope identifier consists of % followed by the name of the network interface. In the below screen, **%eth0** is the scope identifier.

```
[user@host ~]$ ping6 -c3 fe80::5054:ff:fe00:fafe%eth0
PING fe80::5054:ff:fe00:fafe%eth0(fe80::5054:ff:fe00:fafe%eth0) 56 data bytes
64 bytes from fe80::5054:ff:fe00:fafe%eth0: icmp_seq=1 ttl=64 time=0.052 ms
64 bytes from fe80::5054:ff:fe00:fafe%eth0: icmp_seq=2 ttl=64 time=0.057 ms
64 bytes from fe80::5054:ff:fe00:fafe%eth0: icmp_seq=3 ttl=64 time=0.055 ms

--- fe80::5054:ff:fe00:fafe%eth0 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1999ms
rtt min/avg/max/mdev = 0.052/0.054/0.057/0.008 ms
```

Global Scope Addresses

A global unicast address in IPv6 is similar to IPv4 public address. Unlike link-local scope address, the global scope address is globally routable. The global scope address structure consists of a global 48-bit routing-prefix, a 16-bit subnet ID, and a 64-bit interface ID. The IPv6 global unicast address range starts from **2000::/3**. As the global scope addresses are globally routable, no Network Address Translation (NAT) is required with IPv6 addresses.

The global addresses select their outbound interfaces from the routing table. No scope identifier is required when contacting a global scope address.

The Internet Assigned Numbers Authority (IANA) has reserved the **2001:db8::/32** IPv6 address prefix for use in documentation.

```
[user@host ~]$ ping6 -c3 2001:db8:1:1fa1:f816:3eff:feda:9a5a
PING 2001:db8:1:1fa1:f816:3eff:feda:9a5a(2001:db8:1:1fa1:f816:3eff:feda:9a5a) 56 data bytes
64 bytes from 2001:db8:1:1fa1:f816:3eff:feda:9a5a: icmp_seq=1 ttl=63 time=0.827 ms
64 bytes from 2001:db8:1:1fa1:f816:3eff:feda:9a5a: icmp_seq=2 ttl=63 time=0.909 ms
64 bytes from 2001:db8:1:1fa1:f816:3eff:feda:9a5a: icmp_seq=3 ttl=63 time=0.647 ms

--- 2001:db8:1:1fa1:f816:3eff:feda:9a5a ping statistics ---
rtt min/avg/max/mdev = 0.052/0.054/0.057/0.008 ms
```

The IPv6 unicast loopback address, **::1**, is a link-local scope and is attached to a loopback interface.

IPv6 Neighbor Discovery

Neighbor discovery is a protocol also known as *Neighbor Discovery Protocol* (NDP). The protocol allows different nodes on the same link to advertise their existence to their neighbors, and to learn about the existence of their neighbors. The capabilities provided by NDP are similar to those in IPv4 such as Address Resolution Protocol (ARP) and ICMPv4 redirect. The following capabilities are provided by **Neighbor Discovery Protocol**:

Router discovery

Using the two ICMPv6 messages **Router Solicitation** (RS) and **Router Advertisement** (RA), IPv6 hosts can automatically locate the routers on the link. A host sends RS messages to locate the routers on an attached link. A router either sends an RA message on receipt of an RS message, or sends RA messages periodically.

Prefix discovery

Using router advertisement messages, hosts can discover the published IPv6 prefix by the router. A host with an IPv6 prefix is reachable on the link and can communicate directly with other hosts on the same link.

Parameter discovery

Link parameters, such as MTU, are also advertised using RA messages. A host configures the network interface based on the parameters advertised by the router.

Address resolution

Using the two ICMPv6 messages, **Neighbor Solicitation** (NS) and **Neighbor Advertisement** (NA), an IPv6 host maps IP addresses to the link-layer addresses of its neighbors. These messages are equivalent to an ARP request and an ARP reply in IPv4. A host seeking the link-layer address of a neighbor sends an NS message using the multicast address. Its neighbor responds with its link-layer address in an NA message.

Address auto-configuration

A host configures its IPv6 address by using the prefix learned from prefix discovery from the router, and the EUI-64 address from its network interface MAC address.

Neighbor Unreachability Detection (NUD)

Neighbor unreachability detection helps to determine if a neighbor is reachable by sending an NS messages.

Duplicate Address Detection (DAD)

Duplicate address detection helps a host find duplicate addresses on the link. When a host joins the network link for the first time, it sends an NS message before attempting to use the address to communicate. If the host receives an NA message in response to its NS message, it detects that the address is already in use by another host. This helps find duplicate addresses to avoid any IP address conflict.

Next-hop determination

The next-hop determination helps determine the next-hop routers for a destination.

Router redirection

Using the ICMPv6 **Redirect** message enables a router to inform hosts of a better next-hop route for certain destinations.



References

RFC 4291: IP Version 6 Addressing Architecture
<https://tools.ietf.org/html/rfc4291>

Internet Protocol Version 6 Address Space
<https://www.iana.org/assignments/ipv6-address-space/ipv6-address-space.xhtml>

IPv6 Address Prefix Reserved for Documentation
<https://tools.ietf.org/html/rfc3849>

Quiz: Explaining IPv6

Choose the correct answer(s) to the following questions:

1. What is the result of the following **ping** command?

```
[user@demo]$ ping6 ff02::1%eth0
```

- will send traffic to all nodes on the link.
 - will send traffic to the host whose IPv6 address is **ff02::1**.
 - will result in host unreachable.
 - will send traffic to all nodes with global scope IPv6 address.
2. Assuming an interface has a MAC address of **53:52:24:01:fa:fe** and the IPv6 global prefix as **2001:db8::/64**. What is the EUI-64 format IPv6 address assigned to the interface?
 - 2001:db8::5054:24ff:fe01:fafe
 - 2001:db8::5352:24ff:fe01:fafe
 - 2001:db8::5152:24ff:fe01:fafe
 - 2001:db8::5152:ff24::01fe:fafe
 3. Which two statements describe characteristics of IPv6 unicast addressing? (Choose two.)
 - Send packets to link-local addresses that start with fe80::/64.
 - If a global IPv6 address is assigned to an interface, then that is the only allowable address that is assigned.
 - Send packets to any global IPv6 addresses.
 - Send packets only to the loopback address ::1.
 4. Which of the following are functions provided by the Neighbor Discovery Protocol (NDP) used in IPv6? (Choose five.)
 - Hosts can locate routers residing on attached links.
 - Only authorized routers can periodically advertise their presence with various link and Internet parameters.
 - A hosts can discover the prefixes for all other hosts on the link.
 - Mapping between IP addresses and link-layer addresses.
 - Hosts can find next-hop routers for a destination.
 - Hosts can determine that a neighbor is no longer reachable on the link.

Solution

Choose the correct answer(s) to the following questions:

1. What is the result of the following **ping** command?

```
[user@demo]$ ping6 ff02::1%eth0
```

- a. will send traffic to all nodes on the link.
 - b. will send traffic to the host whose IPv6 address is **ff02::1**.
 - c. will result in host unreachable.
 - d. will send traffic to all nodes with global scope IPv6 address.
2. Assuming an interface has a MAC address of **53:52:24:01:fa:fe** and the IPv6 global prefix as **2001:db8::/64**. What is the EUI-64 format IPv6 address assigned to the interface?
 - a. 2001:db8::5054:24ff:fe01:fafe
 - b. 2001:db8::5352:24ff:fe01:fafe
 - c. **2001:db8::5152:24ff:fe01:fafe**
 - d. 2001:db8::5152:ff24::01fe:fafe
3. Which two statements describe characteristics of IPv6 unicast addressing? (Choose two.)
 - a. **Send packets to link-local addresses that start with fe80::/64.**
 - b. If a global IPv6 address is assigned to an interface, then that is the only allowable address that is assigned.
 - c. **Send packets to any global IPv6 addresses.**
 - d. Send packets only to the loopback address ::1.
4. Which of the following are functions provided by the Neighbor Discovery Protocol (NDP) used in IPv6? (Choose five.)
 - a. **Hosts can locate routers residing on attached links.**
 - b. Only authorized routers can periodically advertise their presence with various link and Internet parameters.
 - c. **A hosts can discover the prefixes for all other hosts on the link.**
 - d. Mapping between IP addresses and link-layer addresses.
 - e. Hosts can find next-hop routers for a destination.
 - f. Hosts can determine that a neighbor is no longer reachable on the link.

Assigning IPv6 Addresses

Objectives

After completing this section, students should be able to:

- Describe router advertisement (RA).
- Describe neighbor advertisement (NA).
- Compare and contrast SLAAC, stateless DHCPv6, and stateful DHCPv6.
- Implement a dual stack network within OpenStack.

IPv6 Stateless Auto-configuration

The IPv6 stateless auto-configuration does not require any manual intervention or any additional servers for generating link-local or global addresses. A router is required, however, to advertise the IPv6 prefix. Without this, the global scope addresses can not be assigned to a network interface. Stateless auto-configuration allows the IPv6 hosts to generate their own address by using the local interface ID and router advertised prefix. An IPv6 address is formed by combining the prefix advertised by the router, containing information about the subnet that an interface belongs to, and the interface ID generated from the MAC address of a host's network interface. In absence of a router, a global scope address is not assigned to the interface. The interface, however, is assigned a link-local scope address, sufficient for communicating with other hosts on the same link.

The Neighbor Discovery Protocol (NDP) is the primary protocol in the IPv6 suite. The two main functionalities provided by NDP are neighbor discovery, and stateless auto-configuration of IPv6 addresses. The ICMPv6 messages that perform the functions offered by NDP are:

Router Solicitation (RS)

A host, when attached or reattached to a network link, sends a message to all routers on the link using the multicast address. This ICMPv6 message, sent to all routers, is a Router Solicitation (RS) message. The host waits for the routers on the link to respond with a message to get the default router information to be used by the host, or the IPv6 prefix advertised.

Router Advertisement (RA)

Upon receipt of an RS message, a router sends a message to the soliciting host, or all hosts on the link, periodically. This message is an ICMPv6 message of the type Router Advertisement (RA). The RA message contains information needed by hosts to determine the link MTU, prefix, specific routes, and duration.

Neighbor Solicitation (NS)

A host, in order to communicate with neighbors on the same link or to discover the link-layer address, sends a message using a unicast address to confirm that it is reachable. For link-layer address resolution, the message is sent using the solicited-node multicast address. This message, sent by the host, is an ICMPv6 message of the type Neighbor Solicitation (NS). The host then waits for its neighbor on the link to respond to the message.

Neighbor Advertisement (NA)

Upon receipt of an NS message, a host's neighbor advertises its link-layer address to the host that sent the NS message. If the NS message address is not a multicast address, then the link-layer address is not included in the message. This message, sent by the neighbor, is an ICMPv6 message of the type Neighbor Advertisement (NA). The neighbor is considered reachable if the host has received a confirmation, in the form of an NA message, that packets sent to the neighbor were received by its IP layer. In case of address resolution, the link-layer address of the neighbor is inserted in the host neighbor cache entry.

Redirect

Routers send messages to inform the hosts of a better next-hop host or router for a destination. These messages, sent by the routers, are ICMPv6 messages of the type Redirect. A redirect message contains the link-layer address of the new first-hop router.

The table below summarizes ICMPv6 message types, number, and NDP functionalities provided. These numbers are used to capture message types using packet capture tools, such as `tcpdump`.

Neighbor Discovery (NDP) Message Types and Functions

Message Name	ICMPv6 Message Number	NDP Functions
Router Solicitation	133	router discovery, prefix discovery, parameter discovery, and stateless address auto-configuration
Router Advertisement	134	
Neighbor Solicitation	135	address resolution, Neighbor Unreachability Detection (NUD), and Duplicate Address Detection (DAD)
Neighbor Advertisement	136	
Redirect	137	Better next-hop host or router

Router Discovery Process

An IPv6 host trying to discover the routers available on the same link is referred to as the Router Discovery. When an IPv6 host joins the link, it uses a multicast address to send an RS message to all routers in the multicast group. Upon receiving an RS message, each router on the link sends an RA message. The RA message contains information to perform various NDP functions, such as prefix discovery, parameter discovery, and stateless auto-configuration.

RA messages are sent periodically to all nodes.

This output shows RS and RA messages captured using `tcpdump`:

```

❶ ... IP6 (hlim 255, next-header ICMPv6 (58) payload length: 8) demo-server1.novalocal
> ff02::2: [icmp6 sum ok] ICMP6, router solicitation, length 8
❷ ... IP6 (hlim 255, next-header ICMPv6 (58) payload length: 64) gateway > ff02::1:
[icmp6 sum ok] ICMP6, router advertisement, length 64
    ❸ hop limit 64, Flags [none], pref medium, router lifetime 300s, reachable time
    0ms, retrans time 0ms
    ❹ prefix info option (3), length 32 (4): 2001:db8:1:X::/64, Flags
    [onlink, auto], valid time 86400s, pref. time 14400s
        0x0000: 40c0 0001 5180 0000 3840 0000 0000 2001
        0x0010: 0db8 0001 X 0000 0000 0000 0000
    ❺ mtu option (5), length 8 (1): 1446
        0x0000: 0000 0000 05a6
    ❻ source link-address option (1), length 8 (1): fa:16:3e:c3:c3:48

```

```
0x0000: fa16 3ec3 c348
```

- ➊ The **demo-server1** instance sends an RS message to all routers on the link using the **ff02::2** multicast address.
- ➋ The router named **gateway** sends RA message to all nodes on the link using the **ff02::1** multicast address.
- ➌ The RA message contains information such as current hop limit, flags, router lifetime, reachable time, and re-transmit timer. The **Flags [none]** indicates that the RA message is only giving out the IPv6 prefix.
- ➍ The RA message contains the prefix information advertised by the router. The RA message contains the **2001:db8:1:X::/64** prefix advertised by the router with a valid time of 86400 seconds. The prefix information in the RA message is ignored if the **auto** flag is not set.
- ➎ The RA message contains the MTU value of the link. The interface on **demo-server1** is set to **1446** as the MTU value.
- ➏ The RA message contains the source link-layer address. Using the RA message, the router advertises its link-layer address to the hosts on the link, which is updated by the host in the neighbor cache. This prevents an extra round trip on the network to find the link-layer address of the router.

Address Resolution Process

An IPv6 host seeking the link-layer address of a neighbor on the same link exchanges NS and NA messages. This is referred to as address resolution. An NS message uses the solicited-node multicast address corresponding to the neighbor address. The solicited-node multicast address is created by taking the last 24 bits of a unicast or anycast address and appending them to the prefix **ff02::1:ff00:0/104**. For example, if the link-local address of a host is **fe80::f816:3eff:fecc:a679**, then the solicited-node multicast address is **ff02::1:ffcc::a679**.

The NS message contains the link-layer address of the sending host. When an NS message is received by the target neighbor host, it first updates its own neighbor cache using the link-layer address sent in the NS message. Thereafter, an NA message is sent by the neighbor host to the host that sent the NS message. The NA message contains the link-layer address of the neighbor host as target link-layer address. Once the NA message is received by the host seeking the link-layer address of its neighbor, it updates its neighbor cache with the link-layer address of the neighbor host. After this exchange of messages, both hosts can exchange packets on the link.

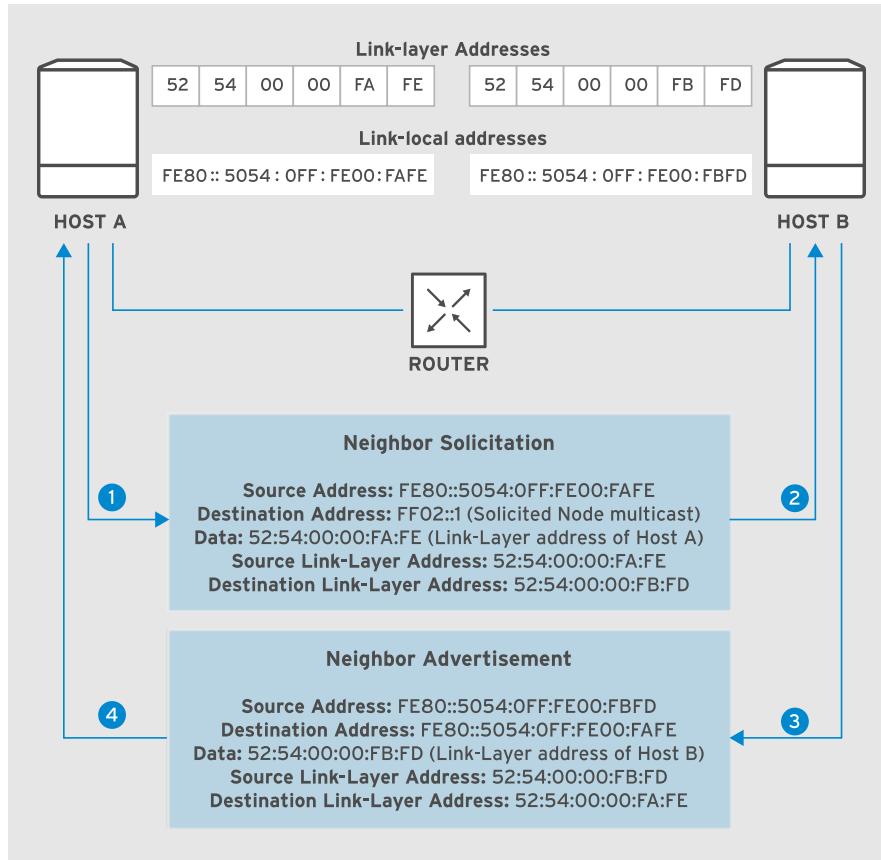


Figure 3.3: Address resolution using NA and NS message

IPv6 Addressing Schemes

IPv6 supports three addressing schemes for address configuration and for providing optional network information:

Stateless Address Auto Configuration (SLAAC)

Using **Stateless Address Auto Configuration** (SLAAC) addressing, the host brings up its interface with a link-local **fe80::/64** address normally. It then uses RA messages to setup the IPv6 global address using the prefix advertised. The router periodically sends multicast updates through RA messages to confirm or update the information provided by the router.

In OpenStack networking, SLAAC uses the **Router Advertisement Daemon** (radvd) to send RA messages in response to RS messages or the RA messages are sent periodically to the IPv6 instances.

Stateless DHCPv6

In **Stateless DHCPv6** addressing, the IPv6 address is auto-configured using the RA message. It uses DHCPv6 to propagate the optional network information such as the DNS search list, nameserver address, and NTP server address.

In OpenStack networking, the **radvd** service is used to send out the RA message containing prefix information. An instance, upon receiving an RA message, sends a DHCPv6 information request to get the other optional network information from the DHCPv6 server. The instance then receives the DHCP reply with the network information. The instance's network interface should be configured to get the optional network information from the DHCPv6 server.

```
IPV6INIT="yes"
DHCPV6C="yes"
DHCPV6C_OPTIONS="-S"
```

DHCPV6C

The **yes** value sets the interface to use DHCPv6 to obtain an IPv6 address and network configuration.

DHCPV6C_OPTION

The **-S** value sets the interface to obtain stateless network configuration only from the DHCPv6 server. The interface will not use the IPv6 address from the DHCPv6 server. Because in case of **Stateless DHCPv6** addressing, the IPv6 address is auto-configured using the RA message.

Stateful DHCPv6

In **Stateful DHCPv6** addressing both the IPv6 address and optional network information are provided by the DHCPv6 server. In response to the RS message, an IPv6 host receives an RA message with the ICMPv6 flag set to managed. The host then sends the DHCPv6 solicit message to all DHCPv6 servers on the link using the multicast address. In response to the solicit message, it then receives a DHCPv6 advertise message sent from the DHCPv6 server. The host then replies with a DHCP request using the multicast address. The DHCPv6 server finishes the exchange with a DHCP reply containing the IPv6 address and other network information.

The instance's network interface should be configured to receive an IPv6 address from a DHCPv6 server.

```
IPV6INIT="yes"
DHCPV6C="yes"
```

Implementing IPv6 within OpenStack

Dual stack allows a host or a router to have both IPv4 and IPv6 protocol stacks implemented. This approach allows a host to send and receive packets belonging to both protocols and thus communicate with every host in the IPv4 and IPv6 network. In OpenStack, there are two types of approaches for implementing IPv6 support:

- Services, API endpoints, and project networks are configured with both IPv4 and IPv6 networks.
- Only project networks are configured with both IPv4 and IPv6 networks, whereas the services and API endpoints use a IPv4 network.

Creating OpenStack Networks with IPv6 Support

Enabling a dual stack in OpenStack networking requires creating a subnet with the IP version set to **IPv6**. Either the subnet allocation pool is provided while creating a subnet or a default subnet pool ID is provided. OpenStack networking supports all three IPv6 addressing schemes namely SLAAC, stateful DHCPv6, and stateless DHCPv6.

From OpenStack dashboard, in the **Create Subnet** dialog box, choose the **IP Version** as **IPv6** in the **Subnet** tab. In the **Subnet Details** tab, ensure that **Enable DHCP** is selected. In the **Subnet Details** tab, **IPv6 Address Configuration Mode** provides the following choices to select from various combinations of IPv6 addressing modes:

- **SLAAC: Address discovered from OpenStack Router**

- SLAAC: Address discovered from an external router
- DHCPv6 stateful: Address discovered from OpenStack DHCP
- DHCPv6 stateless: Address discovered from OpenStack Router and additional information from OpenStack DHCP

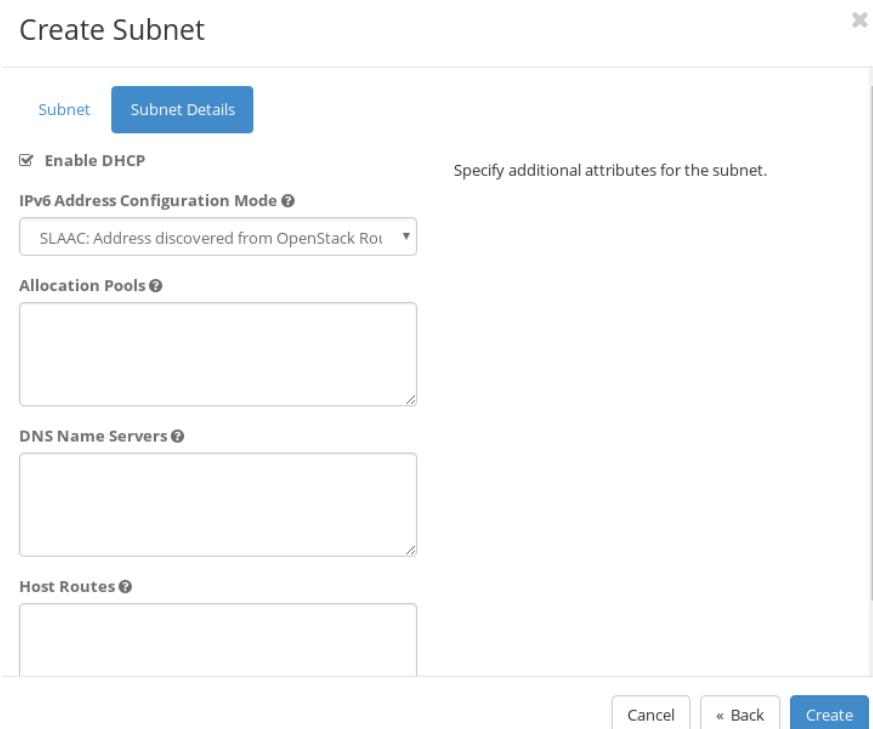


Figure 3.4: IPv6 subnet addressing modes in OpenStack dashboard

When using OpenStack CLI, create the IPv6 subnet on the project network using the **openstack subnet create** command with **ipv6-address-mode** and **ipv6-ra-mode** options. The **ipv6-ra-mode** option determines how RA messages are sent to instances using the subnet. The **ipv6-address-mode** option determines how the instances obtain an IPv6 address, route, and other optional network information. The **enable_dhcp** attribute of the subnet must be set to **True**.

Valid modes for the **ipv6-ra-mode** and the **ipv6-address-mode** options are: **dhcpv6-stateful**, **dhcpv6-stateless**, and **slaac**.

When using SLAAC IPv6 addressing scheme for the IPv6 subnet in a project network, use the following command:

```
[user@demo ~]$ openstack subnet create demo-ipv6-subnet1 \
--ip-version 6 \
--ipv6-ra-mode slaac \
--ipv6-address-mode slaac \
--use-default-subnet-pool \
--network demo-dualstack-network1
+-----+-----+
| Field | Value |
+-----+-----+
```

allocation_pools	::2-::ffff:ffff:ffff:ffff
cidr	::/64
created_at	2017-11-08T08:42:25Z
description	
dns_nameservers	
enable_dhcp	True
gateway_ip	::1
ip_version	6
ipv6_address_mode	slaac
ipv6_ra_mode	slaac
name	demo-ipv6-subnet1
subnetpool_id	prefix_delegation
updated_at	2017-11-09T11:30:36Z

In the above output, the subnet is initially created with a temporary CIDR before an IPv6 prefix is assigned by the prefix delegation server. The **::/64** temporary CIDR is provided by the **prefix_delegation** subnet pool ID. Using the **--use-default-subnet-pool** option allocates the default subnet pool to the subnet.

In OpenStack networking, various combinations of how IPv6 addresses are assigned and optional network information is advertised are supported using the **ipv6-address-mode** and **ipv6_ra_mode** options. This table shows the results based on these combinations:

IPv6 Supported Modes in OpenStack

IPv6 address mode	IPv6 RA mode	Result
SLAAC	SLAAC	IPv6 address is auto-configured using RA messages sent by radvd managed by OpenStack networking, and an external router provides the routing.
SLAAC	Not specified	IPv6 address is auto-configured using RA messages sent by radvd managed by OpenStack networking, and an external router provides the routing.
Not specified	SLAAC	IPv6 address and routing are auto-configured by OpenStack networking using radvd .
Stateless DHCP	Stateless DHCP	IPv6 address is auto-configured by radvd , and optional network information is advertised by dnsmasq using DHCPv6 stateless server managed by OpenStack networking.
Stateless DHCP	Not specified	IPv6 address is auto-configured using RA messages sent by an external router, and optional network information is provided by OpenStack networking.
Not specified	Stateless DHCP	IPv6 address is auto-configured using RA messages sent by OpenStack networking using radvd , and optional network information is provided from an external DHCPv6 stateless server.

IPv6 address mode	IPv6 RA mode	Result
Stateful DHCP	Stateful DHCP	IPv6 address and optional network information is advertised by dnsmsaq using DHCPv6 stateful server managed by OpenStack networking.
Stateful DHCP	Not specified	IPv6 address and optional network information are provided by OpenStack networking using dnsmsaq .
Not specified	Stateful DHCP	IPv6 address and optional network information are provided by an external DHCPv6 stateful server.

IPv6 Networks and Router

In OpenStack networking, IPv6 subnets are connected to an internal router. These IPv6 subnets have multiple IPv6 addresses assigned to the IPv6 internal router interface from their own CIDR block. In the case of a dual stack network, IPv4 subnets have IPv4 internal router interfaces attached with IPv4 addresses. The router ports, when using dual stack network, have both IPv6 and IPv4 address assigned.

Project networks are assigned a global prefix. IPv6 addresses do not require floating IPs for external access. An external network does not require a IPv6 subnet to provide access to the project instances externally. In the classroom environment, the external network with an IPv6 subnet attached to the internal router as a gateway is only used to reach the prefix delegation server. The IPv6 prefix delegation server is used to provide IPv6 prefix requested by the internal/downstream router from an external/upstream router (br-ex).

Launching an Instance using a Dual Stack Network

The following steps outline the process for launching an instance in OpenStack using a dual stack network. Using a dual stack network, an instance is equipped to communicate with both IPv4 and IPv6 protocol stacks in the operating system.

1. Create an IPv6 external subnet in the external network.
2. Create a non-distributed router.
3. Set the external network as the gateway for the router. Optionally, assign the fixed IPv6 address to the gateway.
4. Create a project network.
5. Create IPv4 and IPv6 subnets in the project network. Use **SLAAC** for the IPv6 subnet so that IPv6 addresses are assigned using EUI-64, and the OpenStack Neutron router provides routing to send RA packets.
Optionally, **dhcpv6-stateless** and **dhcpv6-stateful** can be used for IPv6 address configuration and for providing other network information.
6. Attach IPv4 and IPv6 subnets as an interface to the router.
7. Verify that an IPv6 prefix is delegated to the IPv6 subnet.
8. Launch an instance using the project network and other standard resources.

9. Verify that the instance is reachable using both the IPv6 and IPv4 addresses. View the link and global scope IPv6 address assigned to the instance.

References

RFC 4862: IPv6 Stateless Address Autoconfiguration

<http://tools.ietf.org/html/rfc4862>

RFC 4291: IP Version 6 Addressing Architecture

<http://tools.ietf.org/html/rfc4291>

RFC 3736: Stateless Dynamic Host Configuration Protocol (DHCP) Service for IPv6

<http://tools.ietf.org/html/rfc3736>

Further information is available in the chapter on Tenant networking with IPv6 in the

Red Hat OpenStack Platform 10 Networking Guide at

<https://access.redhat.com/documentation/en/red-hat-openstack-platform/>

Guided Exercise: Assigning IPv6 Addresses

In this exercise, you will view router solicitation, router advertisement, neighbor solicitation, and neighbor advertisement messages. You will also compare IPv6 global and link scope addresses.

Outcomes

You should be able to:

- View router solicitation and router advertisement messages using **tcpdump**.
- View neighbor solicitation and neighbor advertisement messages using **tcpdump**.
- Compare IPv6 global and link scope addresses.

Before you begin

Log in to **workstation** as **student** using **student** as the password.

Run the **lab ipv6-addresses setup** command. This script verifies that the overcloud nodes are accessible and running the correct OpenStack services. The script also creates required resources and launches two instances using a dual stack network. The instances are named **finance-server1** and **finance-server2**.



Note

Please note, due to a bug in Dibbler server, if you reboot **controller0** or shutdown your environment in the middle of this exercise, you will need to run **lab ipv6-addresses setup** and then restart the exercise steps.

```
[student@workstation ~]$ lab ipv6-addresses setup
```

Steps

1. Source the **developer1-finance-rc** credentials. Use the **openstack server list** command to retrieve IPv6 addresses of **finance-server1** and **finance-server2**. Take note of the retrieved IPv6 addresses for use during this guided exercise.

```
[student@workstation ~]$ source ~/developer1-finance-rc
[student@workstation ~(developer1-finance)]$ openstack server list \
-c Name -c Networks -f json
[
    {
        "Name": "finance-server2",
        "Networks": "finance-dualstack-network1=2001:db8:1:X:PPPP, ..."
    },
    {
        "Name": "finance-server1",
        "Networks": "finance-dualstack-network1=2001:db8:1:X:NNNN, ..."
    }
]
```

2. From **finance-server1**, run the **tcpdump** command to capture IPv6 router advertisement and router solicitation messages.

- 2.1. Log in to **finance-server1** using **cloud-user** and the IPv6 address retrieved previously.

```
[student@workstation ~](developer1-finance)]$ ssh cloud-user@2001:db8:1:X:NNNN
Warning: Permanently added '2001:db8:1:X:NNNN' (ECDSA) to the list of known
hosts.
[cloud-user@finance-server1 ~]$
```

- 2.2. Run **tcpdump** to capture IPv6 router advertisement and router solicitation messages in the **RA.pcap** file on **finance-server1**.

```
[cloud-user@finance-server1 ~]$ sudo tcpdump -vvvv -ttt \
-i eth0 '(icmp6 && (ip6[40] >= 133 && ip6[40] <= 134))' \
-w RA.pcap
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144
bytes
Got 0
```

The above **tcpdump** command captures incoming packets on **eth0** interface. The command only captures ICMPv6 packet types **133** and **134**. The packet type **133** is associated with router solicitation and **134** is associated with router advertisement.

3. From **workstation**, open a new terminal window and log in to **finance-server1** instance. Use the **cloud-user** user and the IPv6 address associated with **finance-server1**.

Use the **ndptool** command to send a router solicitation message using the **eth0** interface.

```
[student@workstation ~]$ ssh cloud-user@2001:db8:1:X:NNNN
[cloud-user@finance-server1 ~]$ sudo ndptool -i eth0 -t rs send
[cloud-user@finance-server1 ~]$
```

4. Stop the running **tcpdump** command. Using the **tcpdump** command with the **-r** option, extract information from the captured traffic.

- 4.1. From the terminal window running **tcpdump** on **finance-server1**. Wait for some packets to be captured by **tcpdump**.

Press **Ctrl+C** to stop the packet capture.

```
[cloud-user@finance-server1 ~]$ sudo tcpdump -vvvv -ttt \
-i eth0 '(icmp6 && (ip6[40] >= 133 && ip6[40] <= 134))' \
-w RA.pcap
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144
bytes
Got 3
Ctrl+C
```

- 4.2. Extract the information from the **RA.pcap** file. View the router solicitation and advertisement messages captured on **finance-server1**.

```
[cloud-user@finance-server1 ~]$ sudo tcpdump -vvv -r RA.pcap
reading from file RA.pcap, link-type EN10MB (Ethernet)
```

```

... IP6 (hlim 255, next-header ICMPv6 (58) payload length: 8) finance-server1.novalocal > ff02::2: [icmp6 sum ok] ICMP6, router solicitation, length 8
... IP6 (hlim 255, next-header ICMPv6 (58) payload length: 64) gateway
> ff02::1: [icmp6 sum ok] ICMP6, router advertisement, length 64
    hop limit 64, Flags [none], pref medium, router lifetime 300s, reachable
    time 0ms, retrans time 0ms
    prefix info option (3), length 32 (4): 2001:db8:1:X::/64, Flags [onlink,
    auto], valid time 86400s, pref. time 14400s
        0x0000: 40c0 0001 5180 0000 3840 0000 0000 2001
        0x0010: 0db8 0001 X 0000 0000 0000 0000
    mtu option (5), length 8 (1): 1446
        0x0000: 0000 0000 05a6
    source link-address option (1), length 8 (1): fa:16:3e:c3:c3:48
        0x0000: fa16 3ec3 c348

```

The **finance-server1** instance sends the router solicitation message to all routers on the link using the **ff02::2** multicast address. The router named **gateway** in the output sends an advertisement to all nodes on the link using the **ff02::1** multicast address. The router advertisement message contains the **2001:db8:1:X::/64** prefix advertised by the router with a valid time of 86400 seconds. The message also contains an MTU value and the layer 2 MAC address of the router.

- Run **tcpdump** to capture IPv6 neighbor advertisement and neighbor solicitation messages in the **NA.pcap** file on **finance-server1**.

```

[cloud-user@finance-server1 ~]$ sudo tcpdump -vvvv -ttt \
-i eth0 '(icmp6 && (ip6[40] >= 135 && ip6[40] <= 136))' \
-w NA.pcap
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
Got 0

```

The above **tcpdump** command captures incoming packets on **eth0** interface. The command only captures ICMPv6 packet types **135** and **136**. The packet type **135** is associated with neighbor solicitation and **136** is associated with neighbor advertisement.

- From the other terminal window with a remote session connected to **finance-server1**, initiate the neighbor solicitation by using the **ping** command from **finance-server1** to **finance-server2**. View the IPv6 neighbors of **finance-server1** using the **ip -6 neighbor** command.

 - View the IPv6 neighbors of **finance-server1** before initiating the neighbor solicitation by using **ip -6 neighbor**.

```

[cloud-user@finance-server1 ~]$ ip -6 neighbor
fe80::RRRR dev eth0 lladdr fa:16:3e:c3:c3:48 router REACHABLE

```

- Ping the IPv6 address of **finance-server2**, retrieved previously.

```

[cloud-user@finance-server1 ~]$ ping6 -c3 2001:db8:1:X:PPPP
PING 2001:db8:1:X:PPPP(2001:db8:1:X:PPPP) 56 data bytes
64 bytes from 2001:db8:1:X:PPPP: icmp_seq=1 ttl=64 time=1.99 ms
64 bytes from 2001:db8:1:X:PPPP: icmp_seq=2 ttl=64 time=0.732 ms
64 bytes from 2001:db8:1:X:PPPP: icmp_seq=3 ttl=64 time=0.629 ms
--- 2001:db8:1:X:PPPP ping statistics ---

```

```
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
```

6.3. Use **ip -6 neighbor** to view addresses learned and stored in the cache.

Your output might differ based on the hosts that were reachable recently.

```
[cloud-user@finance-server1 ~]$ ip -6 neighbor
fe80::PPPP dev eth0 lladdr fa:16:3e:9b:be:c3 DELAY
fe80::RRRR dev eth0 lladdr fa:16:3e:cc:a6:79 REACHABLE
2001:db8:1:X:PPPP dev eth0 lladdr fa:16:3e:cc:a6:79 REACHABLE
```

7. Stop the running **tcpdump** command. Using the **tcpdump** command with the **-r** option, extract information from the captured traffic.
 - 7.1. From the terminal window running **tcpdump** on **finance-server1**. Wait for some packets to be captured by **tcpdump**.

Press **Ctrl+C** to stop the packet capture.

```
[cloud-user@finance-server1 ~]$ sudo tcpdump -vvv -ttt \
-i eth0 '(icmp6 && (ip6[40] >= 135 && ip6[40] <= 136))' \
-w NA.pcap
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144
bytes
Got 4
Ctrl+C
```

- 7.2. Extract the information from the **NA.pcap** file. View the neighbor solicitation and advertisement messages received by the **finance-server1**.

```
[cloud-user@finance-server1 ~]$ sudo tcpdump -vvv -r NA.pcap
reading from file NA.pcap, link-type EN10MB (Ethernet)
... IP6 (hlim 255, next-header ICMPv6 (58) payload length: 32) gateway >
  finance-server1.novalocal: [icmp6 sum ok] ICMP6, neighbor solicitation, length
  32, who has finance-server1.novalocal
    source link-address option (1), length 8 (1): fa:16:3e:c3:c3:48
      0x0000: fa16 3ec3 c348
... IP6 (hlim 255, next-header ICMPv6 (58) payload length: 24) finance-
  server1.novalocal > gateway: [icmp6 sum ok] ICMP6, neighbor advertisement,
  length 24, tgt is finance-server1.novalocal, Flags [solicited]
... IP6 (hlim 255, next-header ICMPv6 (58) payload length: 32) finance-
  server1.novalocal > ff02::1:ffcc:a679: [icmp6 sum ok] ICMP6, neighbor
  solicitation, length 32, who has 2001:db8:1:X:PPPP
    source link-address option (1), length 8 (1): fa:16:3e:64:16:5d
      0x0000: fa16 3e64 165d
... IP6 (hlim 255, next-header ICMPv6 (58) payload length: 32) 2001:db8:1:X:PPPP
  > finance-server1.novalocal: [icmp6 sum ok] ICMP6, neighbor advertisement,
  length 32, tgt is 2001:db8:1:X:PPPP, Flags [solicited, override]
destination link-address option (2), length 8 (1): fa:16:3e:cc:a6:79
  0x0000: fa16 3ecc a679
... IP6 (hlim 255, next-header ICMPv6 (58) payload length: 32) fe80::PPPP >
  finance-server1.novalocal: [icmp6 sum ok] ICMP6, neighbor solicitation, length
  32, who has finance-server1.novalocal
    source link-address option (1), length 8 (1): fa:16:3e:cc:a6:79
      0x0000: fa16 3ecc a679
```

```
... IP6 (hlim 255, next-header ICMPv6 (58) payload length: 24) finance-
server1.novalocal > fe80::PPPP: [icmp6 sum ok] ICMP6, neighbor advertisement,
length 24, tgt is finance-server1.novalocal, Flags [solicited]
```

The **finance-server1** instance uses the **ff02::1:ffcc:a679** multicast address to send the neighbor solicitation message. The neighbor solicitation message contains the layer 2 MAC address of the host that sends the solicitation message. The host with **2001:db8:1:X:PPPP** global IPv6 address sends a neighbor advertisement message to **finance-server1**. The neighbor advertisement message contains the layer 2 MAC address of the advertised neighbor. Similarly, the link scope addresses are also solicited and advertised. After the neighbor advertisement message is received, both hosts can now exchange packets on the link.

- 7.3. Log out of the **finance-server1** instance in the second terminal.

```
[cloud-user@finance-server1 ~]$ logout
Connection to 2001:db8:1:X:NNNN closed.
[student@workstation ~]$
```

8. From **finance-server1**, ping **finance-server2** using the link scope IPv6 address.

- 8.1. On **workstation**, from the second terminal, log in to **finance-server2** using the IPv6 address and the **cloud-user** user.

```
[student@workstation ~]$ ssh cloud-user@2001:db8:1:X:PPPP
Warning: Permanently added '2001:db8:1:X:PPPP' (ECDSA) to the list of known
hosts.
[cloud-user@finance-server2 ~]$
```

- 8.2. Note the link scope IPv6 address assigned to **eth0** interface of **finance-server2**.

```
[cloud-user@finance-server2 ~]$ ip -6 addr show eth0 scope link
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 state UP qdisc 1000
    inet6 fe80::PPPP/64 scope link
        valid_lft forever preferred_lft forever
```

- 8.3. From **finance-server1**, ping **finance-server2** using link scope IPv6 address. The instance should be reachable, as **finance-server1** and **finance-server2** instances are in the same network segment.

```
[cloud-user@finance-server1 ~]$ ping6 -c3 fe80::PPPP%eth0
PING fe80::PPPP%eth0(fe80::PPPP%eth0) 56 data bytes
64 bytes from fe80::PPPP%eth0: icmp_seq=1 ttl=64 time=1.28 ms
64 bytes from fe80::PPPP%eth0: icmp_seq=2 ttl=64 time=0.667 ms
64 bytes from fe80::PPPP%eth0: icmp_seq=3 ttl=64 time=0.730 ms

--- fe80::PPPP%eth0 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 0.667/0.895/1.289/0.280 ms
```

- 8.4. Log out of the **finance-server2** instance.

```
[cloud-user@finance-server2 ~]$ logout
```

```
Connection to 2001:db8:1:X:PPPP closed.  
[student@workstation ~]$
```

9. From **finance-server1**, ping **workstation** using its global scope and link scope IPv6 addresses. The **workstation** node has **2001:db8::5054:ff:fe00:fafe** as the global scope IPv6 address and its DNS name is **workstation**. The **workstation** node has the link scope IPv6 address **fe80::5054:ff:fe00:fafe**.

View the neighbor solicitation and neighbor advertisement messages received on **workstation** using **ndptool**.

- 9.1. On **workstation**, from the second terminal, run the **ndptool** command to monitor IPv6 packets received on the **eth0** interface.

```
[student@workstation ~]$ sudo ndptool -i eth0 monitor  
[sudo] password for student: student
```

- 9.2. From **finance-server1**, use **ping** to verify that **workstation** is reachable. The **workstation** should be reachable using the global scope IPv6 address.

```
[cloud-user@finance-server1 ~]$ ping6 -c3 workstation  
PING workstation(workstation.lab.example.com (2001:db8::5054:ff:fe00:fafe)) 56  
data bytes  
64 bytes from workstation.lab.example.com (2001:db8::5054:ff:fe00:fafe):  
icmp_seq=1 ttl=62 time=0.577 ms  
64 bytes from workstation.lab.example.com (2001:db8::5054:ff:fe00:fafe):  
icmp_seq=2 ttl=62 time=0.760 ms  
64 bytes from workstation.lab.example.com (2001:db8::5054:ff:fe00:fafe):  
icmp_seq=3 ttl=62 time=0.706 ms  
  
--- workstation ping statistics ---  
3 packets transmitted, 3 received, 0% packet loss, time 2001ms  
rtt min/avg/max/mdev = 0.678/0.725/0.783/0.043 ms
```

- 9.3. Back on **workstation**, view the messages received using the **ndptool** command running on **workstation**. It may take several seconds for the router advertisement (RA) messages to show up.

```
[student@workstation ~]$ sudo ndptool -i eth0 monitor  
NDP payload len 32, from addr: fe80::RRRR, iface: eth0  
    Type: NS  
NDP payload len 24, from addr: fe80::RRRR, iface: eth0  
    Type: NA  
NDP payload len 32, from addr: fe80::RRRR, iface: eth0  
    Type: NS  
NDP payload len 56, from addr: fe80::5054:ff:fe00:fafe, iface: eth0  
    Type: RA  
    Hop limit: 64  
    Managed address configuration: no  
    Other configuration: no  
    Default router preference: medium  
    Router lifetime: 300s  
    Reachable time: unspecified  
    Retransmit time: unspecified  
    Source linkaddr: 52:54:00:00:fa:fe
```

```
Prefix: 2001:db8::/64, valid_time: 86400s, preferred_time: 14400s, on_link:  
yes, autonomous_addr_conf: yes, router_addr: no
```

Notice the neighbor solicitation and neighbor advertisement are sent from the router that connects **finance-server1** with **workstation**. In the above output, **fe80::RRRR** is the link scope address of the router.

Press **Ctrl+C** to exit.

- 9.4. From **finance-server1**, find out if **workstation** is reachable using the **fe80::5054:ff:fe00:faf**e link scope IPv6 address.

```
[cloud-user@finance-server1 ~]$ ping6 -c3 fe80::5054:ff:fe00:faf%eth0  
PING fe80::5054:ff:fe00:faf%eth0(fe80::5054:ff:fe00:faf%eth0) 56 data bytes  
From fe80::NNNN%eth0 icmp_seq=1 Destination unreachable: Address unreachable  
From fe80::NNNN%eth0 icmp_seq=2 Destination unreachable: Address unreachable  
From fe80::NNNN%eth0 icmp_seq=3 Destination unreachable: Address unreachable  
  
--- fe80::5054:ff:fe00:faf%eth0 ping statistics ---  
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 1999ms
```

Observe that **workstation** is not reachable using the link scope IPv6 address as the two hosts are on different network segments.

- 9.5. Log out of the **finance-server1** instance.

```
[cloud-user@finance-server1 ~]$ logout  
Connection to 2001:db8:1:X:NNNN closed.  
[student@workstation ~]$
```

Cleanup

From **workstation**, run the **lab ipv6-addresses cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab ipv6-addresses cleanup
```

This concludes the guided exercise.

Deploying IPv6

Objectives

After completing this section, students should be able to:

- Describe prefix delegation (dibbler).
- View the IPv6 network topology in the Dashboard.
- Deploy instances using an IPv6 network.

Prefix Delegation

IPv6 *prefix delegation* provides a mechanism for assigning IPv6 prefixes using a DHCPv6 server capable of delegating IPv6 prefixes. In this mechanism, the requesting router sends a request to a delegating router to assign an IPv6 prefix. The assigned prefix has an associated valid and preferred lifetime that defines how long the prefix is allowed to be used by the requesting router.

With OpenStack networking, a prefix delegation server is deployed externally and is reachable by an external router. The prefix delegation server uses the external network bridge (br-ex) to delegate a prefix to the downstream project router. The delegated prefix is then added a subnet ID and assigned to multiple IPv6 subnets attached to the project router.

Prefix Delegation Using Dibbler

Dibbler is a DHCPv6 implementation that supports both stateful IPv6 and stateless IPv6 address assignments. It also supports prefix delegation as specified in the RFC-3633 document. **Dibbler** is not part of any Red Hat recommended solution, but has been used in this classroom to demonstrate SLAAC. The *dibbler-server* package implements the server end. The *dibbler-client* package implements the DHCPv6 client and must be installed on all OpenStack network nodes.

In OpenStack networking, prefix delegation must be enabled in the **/etc/neutron/neutron.conf** file. Use the following option to enable the prefix delegation in the file:

```
ipv6_pd_enabled = True
```

OpenStack networking uses **Dibbler** as the default prefix delegation server. In case another prefix delegation server is used, the driver path is configured in the **/etc/neutron/l3-agent.ini** OpenStack L3 agent configuration file.

```
Driver used for ipv6 prefix delegation. This needs to be an entry point
#defined in the neutron.agent.linux(pd_drivers namespace. See setup.cfg for
# entry points included with the neutron source. (string value)
#prefix_delegation_driver = dibbler
...output omitted...
# Service to handle DHCPv6 Prefix delegation. (string value)
#pd_dhcp_driver = dibbler
```

To configure OpenStack networking to use prefix delegation, **dibbler-server** must be configured and started. The **dibbler-server** is configured using the **/etc/dibbler/server.conf** file. To configure the prefix delegation server to provide prefixes, a prefix delegation pool and a client prefix length must be defined in the file.

```

① iface "br-ex" {
    ② pd-class
    {
        ③ pd-pool 2001:db8:1::/48
        ④ pd-length 64
    }
}

```

- ① Interface to listen for prefix delegation requests. In the above output, **br-ex** is the interface that listens to prefix delegation requests.
- ② Defines the prefixes that are delegated to the clients and the access control policy. The **pd-class** server option is defined for an interface that is assigned to listen to prefix delegation requests.
- ③ Defines the IPv6 address range to be used for delegating prefixes. In the above output, **2001:db8:1::/64** prefix is delegated by the prefix delegation server. The IPv6 prefix address range starts with **2001:db8:1:0:0:0:0:0** and ends with **2001:db8:1:ffff:ffff:ffff:ffff:ffff**.
- ④ Defines the length of the delegated prefix. In the above output, **64** is the prefix length of the prefix delegated by the server. The client receives **2001:db8:1:X::/64** as one of the prefixes, where **X** denotes a subnet ID.

The server configuration also provides server options. The **log-level** value defines log verbosity, **log-mode** defines how log dates and times are printed, **script** points to a script that is executed when prefixes are assigned or released.

The **dibbler-server** is started by using **dibbler-server start** or interactively using **dibbler-server run**.

```

[user@controller ~]$ sudo dibbler-server run
| Dibbler - a portable DHCPv6, version 1.0.1RC1 (SERVER, Linux port)
...output omitted...
Server Notice      My pid (942492) is stored in /var/lib/dibbler/server.pid
...output omitted...
Server Notice      Parsing /etc/dibbler/server.conf config file...
Server Debug       PD: Client will receive /64 prefixes (T1=5..3600, T2=10..5400).

① Server Debug     PD: Pool 2001:db8:1:: - 2001:db8:1:ffff:ffff:ffff:ffff:ffff, pool
length: 48, 65536 prefix(es) total.
Server Debug     PD: Up to 65536 prefixes may be assigned.
Server Debug     0 per-client configurations (exceptions) added.
Server Debug     Parsing /etc/dibbler/server.conf done.
Server Info       0 client class(es) defined.
Server Debug     1 interface(s) specified in /etc/dibbler/server.conf
Server Info       Mapping allow, deny list to PD 0

② Server Info     Interface br-ex/6 configuration has been loaded.
Server Notice     Running in stateful mode.
Server Debug      Bulk-leasequery: enabled=no, TCP port=547, max conns=10, timeout=300
Server Debug      DUID's value = 00:01:00:01:21:98:50:ec:5e:3d:c6:42:c6:2c was loaded
from server-duid file.
Server Info       My DUID is 00:01:00:01:21:98:50:ec:5e:3d:c6:42:c6:2c.
Server Info       Loading old address database (server-AddrMgr.xml), using built-in
routines.
Server Info       DB timestamp:1510649054, now()=1510649063, db is 9 second(s) old.
Server Debug      Auth: Replay detection value loaded 0
Server Debug      Loaded PD from a file: t1=3600, t2=5400, iaid=1, iface=/64

```

```
③ Server Debug      Parsed prefix 2001:db8:1:1fa1::/64, pref=86400,
valid=172800,ts=1510647935
  Server Debug      Client
00:02:00:00:22:b8:9c:dc:fa:21:4c:93:43:16:a1:c4:79:95:b9:a6:81:9c loaded from disk
successfully (0/1/0 ia/pd/ta).
  Server Debug      Loaded PD from a file: t1=3600, t2=5400, iaid=1, iface=/6
④ Server Debug      Parsed prefix 2001:db8:1:68c6::/64, pref=86400,
valid=172800,ts=1510646350
  Server Debug      Client
00:02:00:00:22:b8:b1:bb:d3:0b:3e:0d:45:52:8f:ca:d2:ae:5a:ce:40:43 loaded from disk
successfully (0/1/0 ia/pd/ta).
  Server Debug      Cache:server-cache.xml file: parsing started, expecting 2 entries.
  Server Debug      Cache: Prefix 2001:db8:1:1fa1:: added for client
(DUID=00:02:00:00:22:b8:9c:dc:fa:21:4c:93:43:16:a1:c4:79:95:b9:a6:81:9c).
  Server Debug      Cache: Prefix 2001:db8:1:68c6:: added for client
(DUID=00:02:00:00:22:b8:b1:bb:d3:0b:3e:0d:45:52:8f:ca:d2:ae:5a:ce:40:43).
```

- ① Prefix pool defined by the prefix delegation server.
- ② Interface to listen for prefix delegation requests.
- ③④ Prefixes assigned to clients.

Initiating Prefix Delegation

When an IPv6 subnet is created with an IPv6 address mode, an IPv6 RA mode defined as **slaac** or **dhcpv6-stateless**, using the **user-default-subnet-pool** option, then a subnet is created with a temporary CIDR defined by the **prefix_delegation** subnet pool. Multiple subnets can be created using the **prefix_delegation** subnet pool without any address overlap.

The following steps outline the workflow used by the prefix delegation process to assign IPv6 addresses from the prefix pool:

1. The prefix delegation process is initiated when an IPv6 subnet with temporary CIDR is added to an internal router. The prefix delegation client (**dibbler-client**) spawns within the router namespace.
2. The **radvd** daemon configuration is modified with the **::/64** temporary CIDR, and RA messages are advertised.
3. The prefix delegation client is modified to initiate a prefix delegation request for the newly added router interface.
4. The prefix delegation server assigns a prefix and sends the delegated prefix as a response to the prefix delegation client.
5. The prefix delegation client assigns an IPv6 address to the router interface from the delegated prefix.
6. The prefix delegation client's polling script detects a change in the IPv6 address of the router interface, and updates the OpenStack database with the new prefix.
7. The **radvd** daemon then assigns IPv6 addresses from the delegated prefix as new ports are created on the router using IPv6 subnets.

Viewing Prefix Delegation using Dibbler

The following steps outline the steps necessary to use IPv6 prefix delegation to provide automatic allocation of subnet CIDR. The prefix delegation allows an OpenStack administrator to use an external service, such as **Dibbler**, to manage the project network prefixes.

1. From the network node of an OpenStack deployment, enable the prefix delegation in the OpenStack networking configuration file.
2. Install a DHCPv6 server capable of delegating prefixes. If you use **Dibbler**, ensure that the *dibbler-server* package is installed.
3. Configure the prefix delegation server to define prefix delegation pool and prefix length. The name of the network interface to listen for prefix delegation messages must also be configured.

The configuration file should point to a script that adds routes when a prefix is assigned, and delete routes when a prefix is released.

4. Write a script to configure routes to enable the instance using an IPv6 global scope address to be reachable externally. Use the script name as defined in the prefix delegation server configuration file.
5. Start the prefix delegation server.
6. Create the project network and the associated IPv6 subnet. The IPv6 subnet is initially created with a temporary CIDR before the subnet is assigned a prefix by the prefix delegation server.
7. Add the internal IPv6 subnet to the router that has a gateway set to an external network. This process triggers the IPv6 prefix delegation.

The internal IPv6 subnet updates the prefix to the one delegated by the prefix delegation server.

8. Launch an instance using the IPv6 prefix of your new subnet.
The instance should be reachable externally using the associated global scope IPv6 address.
9. Optionally, removing the IPv6 internal subnet from the router results in deleting all routes added by the prefix delegation server. The subnet loses the prefix delegated by the prefix delegation server.

Viewing IPv6 Network Topology in Dashboard

Navigate to Project > Network > Network Topology. In Network Topology, view the project network with temporary CIDR **::/64**.

To add the interface from the IPv6 subnet, click **Add Interface** on the Router Details page. In the **Add Interface** dialog, select the IPv6 subnet. Click **Submit**. In the Network Topology tab, observe that the project network is assigned the new IPv6 prefix of **2001:db8:1:X:/64**.

Network Topology

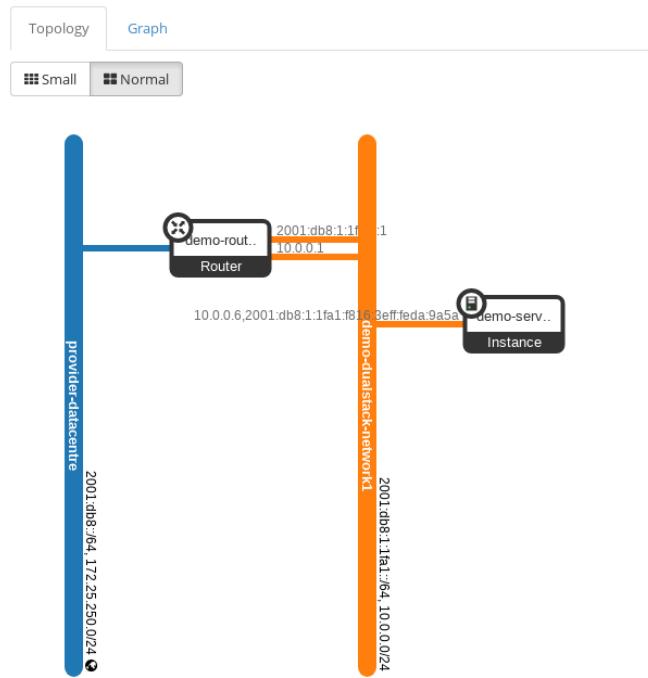


Figure 3.5: IPv6 network topology after prefix delegation

Deploying Instances Using an IPv6 Network

When an instance is created using OpenStack CLI or Dashboard, it uses the OpenStack project network with an IPv6 subnet and an IPv6 prefix delegated to that subnet. This is the default even though IPv6 prefix delegation process can be initiated after launching an instance.

In absence of the internal router and the delegated prefix, the instances are assigned only a link-local address. This enables IPv6 communication within a project using link-local address. Instances in different projects are not reachable using link-local addresses, as they are on different network segments.

In case, the prefix delegation server is configured to delegate globally routable prefixes and the server creates the required routes, this allows the instance to be externally accessible using the global scope IPv6 address. The global scope IPv6 address must be used to communicate between instances on different projects.



References

RFC 3633: IPv6 Prefix Options for DHCPv6
<https://tools.ietf.org/html/rfc3633>

Using OpenStack Networking with IPv6
<https://docs.openstack.org/liberty/networking-guide/adv-config-ipv6.html>

Further information is available in the chapter on Tenant networking with IPv6 in the *Red Hat OpenStack Platform 10 Networking Guide* at

<https://access.redhat.com/documentation/en/red-hat-openstack-platform/>

Guided Exercise: Deploying IPv6

In this exercise, you will test IPv6 network connectivity between instances within an OpenStack project, and across projects using link and global scope IPv6 addresses. You will also implement IPv6 project-level isolation using security groups.

Outcomes

You should be able to:

- Test IPv6 network connectivity between instances within a project.
- Test IPv6 network connectivity between instances across different projects.
- Implement IPv6 network project-level isolation using security groups.

Before you begin

Log in to **workstation** as **student** using **student** as the password.

Run the **lab ipv6-deploying setup** command. This script verifies that the overcloud nodes are accessible and running the correct OpenStack services. The script also creates required resources and launches three instances using a dual stack network. The instances in the **finance** project are: **finance-server1** and **finance-server2**.



Note

Please note, due to a bug in Dibbler server, if you reboot **controller0** or shutdown your environment in the middle of this exercise, you will need to run **lab ipv6-deploying setup** and then restart the exercise steps.

```
[student@workstation ~]$ lab ipv6-deploying setup
```

Steps

1. As **developer1**, create a tenant network named **research-dualstack-network1** in the **research** project.
 - 1.1. Source the **/home/student/developer1-research-rc** credential file.

```
[student@workstation ~]$ source ~/developer1-research-rc
[student@workstation ~(developer1-research)]$
```

- 1.2. Create the **research-dualstack-network1** project network.

```
[student@workstation ~(developer1-research)]$ openstack network create \
research-dualstack-network1
...output omitted...
```

2. Create two subnets for the **research-dualstack-network1** network named **research-ipv4-subnet1** and **research-ipv6-network1**. Use **SLAAC** for the IPv6 subnet so that IPv6 addresses are assigned using EUI-64, and OpenStack networking provides the routing to send RA packets.

Use the following table to create the **research-ipv4-subnet1** IPv4 subnet:

IPv4 Subnet Configuration

Option	Value
network	research-dualstack-network1
subnet-range	10.0.0.0/24
dns-nameserver	172.25.250.254

Use the following table to create the **research-ipv6-subnet1** IPv6 subnet:

IPv6 Subnet Configuration

Option	Value
network	research-dualstack-network1
ip-version	6
ipv6-ra-mode	slaac
ipv6-address-mode	slaac
use-default-subnet-pool	no value required.

- 2.1. Create an IPv4 subnet for the **research-dualstack-network1** project network named **research-ipv4-subnet1**.

```
[student@workstation ~(developer1-research)]$ openstack subnet create \
--network research-dualstack-network1 \
--subnet-range 10.0.0.0/24 \
--dns-nameserver 172.25.250.254 \
research-ipv4-subnet1
...output omitted...
```

- 2.2. Create an IPv6 subnet for the **research-dualstack-network1** project network named **research-ipv6-subnet**.

```
[student@workstation ~(developer1-research)]$ openstack subnet create \
--network research-dualstack-network1 \
--ip-version 6 \
--ipv6-ra-mode slaac \
--ipv6-address-mode slaac \
--use-default-subnet-pool \
research-ipv6-subnet1
+-----+-----+
| Field          | Value           |
+-----+-----+
| allocation_pools | ::2::ffff:ffff:ffff:ffff
| cidr           | ::/64
| created_at     | 2017-11-06T11:27:49Z
| description     |
| dns_nameservers |
| enable_dhcp     | True
| gateway_ip      | ::1
| headers         |
| host_routes     |
| id              | 704b5ecf-4cfe-4789-b2a9-b3a2800bd3b0
| ip_version       | 6
```

ipv6_address_mode	slaac
ipv6_ra_mode	slaac
name	research-ipv6-subnet1
network_id	b14d61a9-ff9e-4ce2-8a79-5a288b767ff0
project_id	01b838e1858f4a38bb2f5bcf0d86fa5f
project_id	01b838e1858f4a38bb2f5bcf0d86fa5f
revision_number	2
service_types	[]
subnetpool_id	prefix_delegation
updated_at	2017-11-06T11:27:49Z
use_default_subnetpool	True

Observe that in the output, the **cidr** property is set to `::/64` and the **subnetpool_id** property is configured to use **prefix_delegation** subnet pool.

3. Attach the **research-ipv4-subnet1** subnet to the **research-router1** router.

- 3.1. Attach the **research-ipv4-subnet1** subnet to the **research-router1** router.

```
[student@workstation ~ (developer1-research)]$ openstack router add subnet \
research-router1 research-ipv4-subnet1
```

- 3.2. Verify that an IPv4 address is assigned from the subnet pool to the port on the **research-router1** router.

```
[student@workstation ~ (developer1-research)]$ openstack port list \
--device-owner network:router_interface \
--router research-router1 \
-c 'Fixed IP Addresses' -f value \
| grep $(openstack subnet show research-ipv4-subnet1 -c id -f value)
ip_address='10.0.0.1', subnet_id='e0e6c6ac-27ab-43e6-aafb-bf4276e4f559'
```

4. Attach the **research-ipv6-subnet1** subnet to the **research-router1** router.

- 4.1. Attach the **research-ipv6-subnet1** subnet to the **research-router1** router.

```
[student@workstation ~ (developer1-research)]$ openstack router add subnet \
research-router1 research-ipv6-subnet1
```

- 4.2. Verify that an IPv6 prefix is delegated to the **research-ipv6-subnet1** subnet.

Notice that the **research-ipv6-subnet1** subnet CIDR changes from `::/64` to `2001:db8:1:X::/64`.

```
[student@workstation ~ (developer1-research)]$ openstack subnet show \
research-ipv6-subnet1
+-----+-----+
| Field | Value |
+-----+-----+
| allocation_pools | 2001:db8:1:Y::2-2001:db8:1:Y:ffff:ffff:ffff:ffff |
| cidr | 2001:db8:1:Y::/64 |
| created_at | 2017-11-06T11:27:49Z |
| description | |
| dns_nameservers | |
| enable_dhcp | True |
```

gateway_ip	2001:db8:1:Y::1
host_routes	
id	704b5ecf-4cfe-4789-b2a9-b3a2800bd3b0
ip_version	6
ipv6_address_mode	slaac
ipv6_ra_mode	slaac
name	research-ipv6-subnet1
network_id	b14d61a9-ff9e-4ce2-8a79-5a288b767ff0
project_id	01b838e1858f4a38bb2f5bcf0d86fa5f
project_id	01b838e1858f4a38bb2f5bcf0d86fa5f
revision_number	3
service_types	[]
subnetpool_id	prefix_delegation
updated_at	2017-11-06T11:43:55Z

- 4.3. Verify that an IPv6 address is assigned from the subnet pool to the port on the **research-router1** router.

```
[student@workstation ~ (developer1-research)]$ openstack port list \
--device-owner network:router_interface \
--router research-router1 \
-c 'Fixed IP Addresses' -f value \
| grep $(openstack subnet show research-ipv6-subnet1 -c id -f value)
ip_address='2001:db8:1:Y::1', subnet_id='704b5ecf-4cfe-4789-b2a9-b3a2800bd3b0'
```

5. As **developer1**, launch an instance named **research-server1** in the **research** project using the standard resources available.

Use the **research-dualstack-network1** network to associate both IPv4 and IPv6 addresses.

```
[student@workstation ~ (developer1-research)]$ openstack server create \
--image rhel7 \
--flavor default \
--key-name example-keypair \
--nic net-id=research-dualstack-network1 \
--wait research-server1
...output omitted...
```

6. Associate a floating IP address to the **research-server1** instance.

```
[student@workstation ~ (developer1-research)]$ openstack floating ip list \
-c 'Floating IP Address' -c 'Port'
+-----+-----+
| Floating IP Address | Port |
+-----+-----+
| 172.25.250.122      | None  |
| 172.25.250.123      | None  |
| 172.25.250.125      | None  |
| 172.25.250.121      | None  |
| 172.25.250.124      | None  |
+-----+-----+
[student@workstation ~ (developer1-research)]$ openstack server add floating ip \
research-server1 172.25.250.R
[student@workstation ~ (developer1-research)]$
```

7. On **workstation**, open three terminal windows. Use different terminal windows to log in to **finance-server1**, **finance-server2**, and **research-server1** instances.

- 7.1. Source the **developer1-finance-rc** credentials. Use the **openstack server list** command to retrieve the IPv6 address of **finance-server1**.

```
[student@workstation ~]$ source ~/developer1-finance-rc
[student@workstation ~(developer1-finance)]$ openstack server list \
-c Name -c Networks -f json
[
  {
    "Name": "finance-server2",
    "Networks": "finance-dualstack-network1=2001:db8:1:X:PPPP, ..."
  },
  {
    "Name": "finance-server1",
    "Networks": "finance-dualstack-network1=2001:db8:1:X:NNNN, ..."
  }
]
```

- 7.2. Log in to **finance-server1** using the **cloud-user** user and the IPv6 address retrieved previously.

```
[student@workstation ~(developer1-finance)]$ ssh cloud-user@2001:db8:1:X:NNNN
Warning: Permanently added '2001:db8:1:X:NNNN' (ECDSA) to the list of known
hosts.
[cloud-user@finance-server1 ~]$
```

- 7.3. In the second terminal window, source the **developer1-finance-rc** credentials. Use the **openstack server list** command to retrieve the IPv6 address of **finance-server2**.

```
[student@workstation ~]$ source ~/developer1-finance-rc
[student@workstation ~(developer1-finance)]$ openstack server list \
-c Name -c Networks -f json
[
  {
    "Name": "finance-server2",
    "Networks": "finance-dualstack-network1=2001:db8:1:X:PPPP, ..."
  },
  {
    "Name": "finance-server1",
    "Networks": "finance-dualstack-network1=2001:db8:1:X:NNNN, ..."
  }
]
```

- 7.4. Log in to **finance-server2** using the **cloud-user** user and the IPv6 address retrieved previously.

```
[student@workstation ~(developer1-finance)]$ ssh cloud-user@2001:db8:1:X:PPPP
Warning: Permanently added '2001:db8:1:X:PPPP' (ECDSA) to the list of known
hosts.
[cloud-user@finance-server2 ~]$
```

- 7.5. In the third terminal window, source the **developer1-research-rc** credentials. Use the **openstack server list** command to retrieve the IPv6 address of **research-server1**.

```
[student@workstation ~]$ source ~/developer1-research-rc
[student@workstation ~(developer1-research)]$ openstack server list \
-c Name -c Networks -f json
[
{
    "Name": "research-server1",
    "Networks": "research-dualstack-network1=2001:db8:1:Y:RRRR, ..."
}
```

- 7.6. Log in to **research-server1** using the **cloud-user** user and the IPv6 address retrieved previously.

```
[student@workstation ~(developer1-research)]$ ssh cloud-user@2001:db8:1:Y:RRRR
Warning: Permanently added '2001:db8:1:Y:RRRR' (ECDSA) to the list of known
hosts.
[cloud-user@research-server1 ~]$
```

8. Retrieve the link scope IPv6 address of **finance-server2**. Test the connectivity from **finance-server1** using the link scope IPv6 address of **finance-server2**.

- 8.1. From **finance-server2**, use the **ip -6 addr show** command to retrieve the link scope IPv6 address.

```
[cloud-user@finance-server2 ~]$ ip -6 addr show eth0 scope link
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 state UP qlen 1000
inet6 fe80::PPPP/64 scope link
    valid_lft forever preferred_lft forever
```

- 8.2. From **finance-server1**, use the link scope IPv6 address of **finance-server2** to test the connectivity.

Both the instances **finance-server1** and **finance-server2** should be reachable using link scope IPv6 addresses, as they are in same network segment.

```
[cloud-user@finance-server1 ~]$ ping6 -c3 fe80::PPPP%eth0
PING fe80::PPPP%eth0(fe80::PPPP%eth0) 56 data bytes
64 bytes from fe80::PPPP%eth0: icmp_seq=1 ttl=64 time=2.22 ms
64 bytes from fe80::PPPP%eth0: icmp_seq=2 ttl=64 time=0.728 ms
64 bytes from fe80::PPPP%eth0: icmp_seq=3 ttl=64 time=0.767 ms

--- fe80::PPPP%eth0 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 0.728/1.239/2.223/0.696 ms
```

9. Verify the connectivity between instances across projects using the link scope IPv6 address.

Test that **finance-server2** is reachable from **research-server1** using the link scope IPv6 address.

- 9.1. From **research-server1**, use the link scope IPv6 address of **finance-server2** to test the connectivity.

The **finance-server2** instance should not be reachable using a link scope IPv6 address, as both instances are in different network segments.

```
[cloud-user@research-server1 ~]$ ping6 -c3 fe80::PPPP%eth0
PING fe80::PPPP%eth0(fe80::PPPP%eth0) 56 data bytes
From fe80::RRRR%eth0 icmp_seq=1 Destination unreachable: Address unreachable
From fe80::RRRR%eth0 icmp_seq=2 Destination unreachable: Address unreachable
From fe80::RRRR%eth0 icmp_seq=3 Destination unreachable: Address unreachable

--- fe80::PPPP%eth0 ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 1999ms
```

10. Verify connectivity between instances across project using the global scope IPv6 address.

Test that **finance-server1** is reachable from **research-server1** using a global scope IPv6 address.

- 10.1. From **finance-server1**, use the **ip -6 addr show** command to list the global scope IPv6 address.

```
[cloud-user@finance-server1 ~]$ ip -6 addr show eth0 scope global
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 state UP qdisc mq
    inet6 2001:db8:1:X:NNNN/64 scope global mngtmpaddr dynamic
          valid_lft 86307sec preferred_lft 14307sec
```

- 10.2. From **research-server1**, use **ping** to verify connectivity using the global scope IPv6 address of **finance-server1**.

```
[cloud-user@research-server1 ~]$ ping6 -c3 2001:db8:1:X:NNNN
PING 2001:db8:1:X:NNNN(2001:db8:1::NNNN) 56 data bytes
64 bytes from 2001:db8:1:X:NNNN: icmp_seq=1 ttl=61 time=3.10 ms
64 bytes from 2001:db8:1:X:NNNN: icmp_seq=2 ttl=61 time=0.997 ms
64 bytes from 2001:db8:1:X:NNNN: icmp_seq=3 ttl=61 time=0.771 ms

--- 2001:db8:1:X:NNNN ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 0.771/1.625/3.107/1.051 ms
```

Observe that **finance-server1** is reachable from **research-server1** using its global scope IPv6 address, as global scope addresses are globally routable and reachable on the IPv6 internet.

11. Implement project-level isolation using security group rules to drop SSH IPv6 packets across projects.

Create security group rules in the **finance** project to drop SSH IPv6 packets originating from the **research** project.

- 11.1. From **research-server1**, verify that an SSH connection using the IPv6 address of **finance-server1** is successful.

Download the private key from <http://materials.example.com/keypairs/example-keypair>. Use the **cloud-user** user to log in to **finance-server1**.

```
[cloud-user@research-server1 ~]$ curl -s -o \
http://materials.example.com/keypairs/example-keypair
[cloud-user@research-server1 ~]$ chmod 400 example-keypair
[cloud-user@research-server1 ~]$ ssh \
-i example-keypair cloud-user@2001:db8:1:X:NNNN
...output omitted...
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '2001:db8:1:X:NNNN' (ECDSA) to the list of known
hosts.
[cloud-user@finance-server1 ~]$ logout
Connection to 2001:db8:1:X:NNNN closed.
```

Log out of **finance-server1** in the first terminal window.

11.2. From **workstation**, source the **developer1-finance-rc** credentials if required.

Create a security group named **restrict-research**.

```
[student@workstation ~(developer1-finance)]$ openstack security group create \
restrict-research \
--description "Drop SSH traffic from research project"
+-----+
| Field      | Value
+-----+
| created_at | 2017-11-02T10:38:12Z
| description | Drop SSH traffic from research project
| id          | 375032a4-f9d4-463e-87af-11f7664ead2a
| name        | restrict-research
| project_id  | 7d49a57a011442268db58762a69f8596
| project_id  | 7d49a57a011442268db58762a69f8596
| revision_number | 1
| rules       |
|             | created_at='2017-11-02T10:38:12Z',
|             | direction='egress', ethertype='IPv6',
|             | id='0ca49875-ea36-44d5-9510-262e010a26dc',
|             | project_id='7d49a57a011442268db58762a69f8596',
|             | revision_number='1',
|             | updated_at='2017-11-02T10:38:12Z'
|             | created_at='2017-11-02T10:38:12Z',
|             | direction='egress', ethertype='IPv4',
|             | id='16e871f5-f25c-44a7-a67c-424bd86fb8c7',
|             | project_id='7d49a57a011442268db58762a69f8596',
|             | revision_number='1',
|             | updated_at='2017-11-02T10:38:12Z'
| updated_at  | 2017-11-02T10:38:12Z
+-----+
```

11.3. Create a security group rule to allow only IPv6 SSH traffic from the IPv6 prefix **2001:db8::/64** (which includes the **workstation** virtual machine).

```
[student@workstation ~(developer1-finance)]$ openstack security group \
rule create restrict-research \
--protocol tcp \
--ethertype IPv6 \
--dst-port 22:22 \
--src-ip 2001:db8::/64
```

Field	Value
created_at	2017-11-02T11:08:02Z
description	
direction	ingress
ethertype	IPv6
headers	
id	6295501d-c86f-478e-b39e-c010567604a6
port_range_max	22
port_range_min	22
project_id	7d49a57a011442268db58762a69f8596
remote_group_id	7d49a57a011442268db58762a69f8596
protocol	tcp
remote_ip_prefix	2001:db8::/64
revision_number	1
security_group_id	375032a4-f9d4-463e-87af-11f7664ead2a
updated_at	2017-11-02T11:08:02Z

- 11.4. Remove the **default** security group from the **finance-server1** instance, as it allows IPv6 SSH traffic from all hosts.

Add the **restrict-research** security group to the **finance-server1** instance to allow IPv6 SSH traffic only from a host with a **2001:db8::/64** prefix.

```
[student@workstation ~](developer1-finance)]$ openstack server remove \
    security group finance-server1 default
[student@workstation ~](developer1-finance)]$ openstack server add \
    security group finance-server1 restrict-research
```

- 11.5. From **research-server1**, verify that SSH IPv6 packets to the **finance** project are dropped from the **research** project.

Press **Ctrl+C** to exit.

```
[cloud-user@research-server1 ~]$ ssh \
-i example-keypair cloud-user@2001:db8:1:X:NNNN
Ctrl+C
```

Log out of **finance-server2** and **research-server1**.

Cleanup

From **workstation**, run the **lab ipv6-deploying cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab ipv6-deploying cleanup
```

This concludes the guided exercise.

Lab: Deploying IPv6 Networks

In this lab, you will create a tenant network with IPv4 and IPv6 subnets and attach the network to an instance. You will also verify the connectivity to the instance using both the IPv4 and IPv6 addresses.

Outcomes

You should be able to:

- Create a dual stack tenant network with IPv4 and IPv6 subnets.
- Launch an instance using the dual stack network.
- Verify connectivity to the instance using the IPv4 and IPv6 addresses.

Before you begin

Log in to **workstation** as **student** using **student** as the password.

Run the **lab ipv6-deploying-networks setup** command. The script verifies that the **provider-datacentre** external network has an IPv6 subnet. The script also verifies that the **provider-datacentre** network is attached as a gateway to the **production-router1** router.



Note

Please note, due to a bug in Dibbler server, if you reboot **controller0** or shutdown your environment in the middle of this exercise, you will need to run **lab ipv6-deploying-networks setup** and then restart the lab steps.

```
[student@workstation ~]$ lab ipv6-deploying-networks setup
```

Steps

1. As **operator1**, create a tenant network named **production-dualstack-network2** in the **production** project.
2. Create two subnets for the **production-dualstack-network2** network named **production-ipv4-subnet2** and **production-ipv6-network2**. Use **SLAAC** for the IPv6 subnet so that IPv6 addresses are assigned using EUI-64, and OpenStack networking provides the routing to send RA packets.

Use the following table to create the **production-ipv4-subnet2** IPv4 subnet:

IPv4 Subnet Configuration

Option	Value
network	production-dualstack-network2
subnet-range	10.0.0.0/24
dns-nameserver	172.25.250.254

Use the following table to create the **production-ipv6-subnet2** IPv6 subnet:

IPv6 Subnet Configuration

Option	Value
network	production-dualstack-network2
ip-version	6
ipv6-ra-mode	slaac
ipv6-address-mode	slaac
use-default-subnet-pool	no value required.

3. Attach the **production-ipv4-subnet2** subnet to the **production-router1** router.
4. Attach the **production-ipv6-subnet2** subnet to the **production-router1** router.
5. As **operator1**, launch an instance named **production-server2** in the **production** project using the standard resources available.

Use the **production-dualstack-network2** network to associate both IPv4 and IPv6 addresses.

6. Associate a floating IP address to the **production-server2** instance.
7. Verify that the **production-server2** instance is reachable from **workstation** using the IPv4 floating IP address and that it has an internal IPv4 address assigned. Also verify that **workstation** is reachable from **production-server2** using name resolution.
8. Verify that the **production-server2** instance is reachable from **workstation** using the IPv6 address. Use SSH to log in to **production-server2** using the **cloud-user** user and the IPv6 address associated with the instance. Use **ping6** to verify that **workstation** is reachable from **production-server2** using name resolution.

Evaluation

From **workstation**, run the **lab ipv6-deploying-networks grade** command to confirm the success of this exercise. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab ipv6-deploying-networks grade
```

Cleanup

From **workstation**, run the **lab ipv6-deploying-networks cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab ipv6-deploying-networks cleanup
```

This concludes the lab.

Solution

In this lab, you will create a tenant network with IPv4 and IPv6 subnets and attach the network to an instance. You will also verify the connectivity to the instance using both the IPv4 and IPv6 addresses.

Outcomes

You should be able to:

- Create a dual stack tenant network with IPv4 and IPv6 subnets.
- Launch an instance using the dual stack network.
- Verify connectivity to the instance using the IPv4 and IPv6 addresses.

Before you begin

Log in to **workstation** as **student** using **student** as the password.

Run the **lab ipv6-deploying-networks setup** command. The script verifies that the **provider-datacentre** external network has an IPv6 subnet. The script also verifies that the **provider-datacentre** network is attached as a gateway to the **production-router1** router.



Note

Please note, due to a bug in Dibbler server, if you reboot **controller0** or shutdown your environment in the middle of this exercise, you will need to run **lab ipv6-deploying-networks setup** and then restart the lab steps.

```
[student@workstation ~]$ lab ipv6-deploying-networks setup
```

Steps

1. As **operator1**, create a tenant network named **production-dualstack-network2** in the **production** project.
 - 1.1. Source the **/home/student/operator1-production-rc** credential file.

```
[student@workstation ~]$ source ~/operator1-production-rc  
[student@workstation ~(operator1-production)]$
```

- 1.2. Create the **production-dualstack-network2** tenant network.

```
[student@workstation ~(operator1-production)]$ openstack network create \  
production-dualstack-network2  
...output omitted...
```

2. Create two subnets for the **production-dualstack-network2** network named **production-ipv4-subnet2** and **production-ipv6-network2**. Use **SLAAC** for the IPv6 subnet so that IPv6 addresses are assigned using EUI-64, and OpenStack networking provides the routing to send RA packets.

Use the following table to create the **production-ipv4-subnet2** IPv4 subnet:

IPv4 Subnet Configuration

Option	Value
network	production-dualstack-network2
subnet-range	10.0.0.0/24
dns-nameserver	172.25.250.254

Use the following table to create the **production-ipv6-subnet2** IPv6 subnet:

IPv6 Subnet Configuration

Option	Value
network	production-dualstack-network2
ip-version	6
ipv6-ra-mode	slaac
ipv6-address-mode	slaac
use-default-subnet-pool	no value required.

- 2.1. Create an IPv4 subnet for the **production-dualstack-network2** project network named **production-ipv4-subnet2**.

```
[student@workstation ~(operator1-production)]$ openstack subnet create \
--network production-dualstack-network2 \
--subnet-range 10.0.0.0/24 \
--dns-nameserver 172.25.250.254 \
production-ipv4-subnet2
...output omitted...
```

- 2.2. Create an IPv6 subnet for the **production-dualstack-network2** project network named **production-ipv6-subnet**.

```
[student@workstation ~(operator1-production)]$ openstack subnet create \
--network production-dualstack-network2 \
--ip-version 6 \
--ipv6-ra-mode slaac \
--ipv6-address-mode slaac \
--use-default-subnet-pool \
production-ipv6-subnet2
+-----+-----+
| Field      | Value
+-----+-----+
| allocation_pools | ::2-::ffff:ffff:ffff:ffff
| cidr        | ::/64
| created_at   | 2017-11-06T11:27:49Z
| description    |
| dns_nameservers |
| enable_dhcp    | True
| gateway_ip     | ::1
| headers        |
| host_routes    |
| id             | 704b5ecf-4cfe-4789-b2a9-b3a2800bd3b0
| ip_version      | 6
| ipv6_address_mode | slaac
| ipv6_ra_mode    | slaac
```

name	production-ipv6-subnet2
network_id	b14d61a9-ff9e-4ce2-8a79-5a288b767ff0
project_id	01b838e1858f4a38bb2f5bcf0d86fa5f
project_id	01b838e1858f4a38bb2f5bcf0d86fa5f
revision_number	2
service_types	[]
subnetpool_id	prefix_delegation
updated_at	2017-11-06T11:27:49Z
use_default_subnetpool	True

Observe that in the output, the **cidr** property is set to `::/64` and the **subnetpool_id** property is configured to use **prefix_delegation** subnet pool.

3. Attach the **production-ipv4-subnet2** subnet to the **production-router1** router.
- 3.1. Attach the **production-ipv4-subnet2** subnet to the **production-router1** router.

```
[student@workstation ~(operator1-production)]$ openstack router add subnet \
production-router1 production-ipv4-subnet2
```

- 3.2. Verify that an IPv4 address is assigned from the subnet pool to the port on the **production-router1** router.

```
[student@workstation ~(operator1-production)]$ openstack port list \
--device-owner network:router_interface \
--router production-router1 \
-c 'Fixed IP Addresses' -f value \
| grep $(openstack subnet show production-ipv4-subnet2 -c id -f value)
ip_address='10.0.0.1', subnet_id='e0e6c6ac-27ab-43e6-aafb-bf4276e4f559'
```

4. Attach the **production-ipv6-subnet2** subnet to the **production-router1** router.
- 4.1. Attach the **production-ipv6-subnet2** subnet to the **production-router1** router.

```
[student@workstation ~(operator1-production)]$ openstack router add subnet \
production-router1 production-ipv6-subnet2
```

- 4.2. Verify that an IPv6 prefix is delegated to the **production-ipv6-subnet2** subnet.

Notice that the **production-ipv6-subnet2** subnet CIDR changes from `::/64` to `2001:db8:1:X::/64`.

student@workstation ~(operator1-production)]\$ openstack subnet show \	production-ipv6-subnet2
+-----+-----+	+-----+
Field	Value
+-----+-----+	+-----+
allocation_pools	2001:db8:1:X::2-2001:db8:1:X:ffff:ffff:ffff:ffff
cidr	2001:db8:1:X::/64
created_at	2017-11-06T11:27:49Z
description	
dns_nameservers	
enable_dhcp	True
gateway_ip	2001:db8:1:X::1
host_routes	

id	704b5ecf-4cfe-4789-b2a9-b3a2800bd3b0
ip_version	6
ipv6_address_mode	slaac
ipv6_ra_mode	slaac
name	production-ipv6-subnet2
network_id	b14d61a9-ff9e-4ce2-8a79-5a288b767ff0
project_id	01b838e1858f4a38bb2f5bcf0d86fa5f
project_id	01b838e1858f4a38bb2f5bcf0d86fa5f
revision_number	3
service_types	[]
subnetpool_id	prefix_delegation
updated_at	2017-11-06T11:43:55Z
+	-----+-----+

- 4.3. Verify that an IPv6 address is assigned from the subnet pool to the port on the **production-router1** router.

```
[student@workstation ~ (operator1-production)]$ openstack port list \
--device-owner network:router_interface \
--router production-router1 \
-c 'Fixed IP Addresses' -f value \
| grep $(openstack subnet show production-ipv6-subnet2 -c id -f value)
ip_address='2001:db8:1:X::1', subnet_id='704b5ecf-4cfe-4789-b2a9-b3a2800bd3b0'
```

5. As **operator1**, launch an instance named **production-server2** in the **production** project using the standard resources available.

Use the **production-dualstack-network2** network to associate both IPv4 and IPv6 addresses.

```
[student@workstation ~ (operator1-production)]$ openstack server create \
--image rhel7 \
--flavor default \
--key-name example-keypair \
--nic net-id=production-dualstack-network2 \
--wait production-server2
...output omitted...
```

6. Associate a floating IP address to the **production-server2** instance.

```
[student@workstation ~ (operator1-production)]$ openstack floating ip list \
-c 'Floating IP Address' -c 'Port'
+-----+-----+
| Floating IP Address | Port |
+-----+-----+
| 172.25.250.162      | None  |
| 172.25.250.165      | None  |
| 172.25.250.161      | None  |
| 172.25.250.164      | None  |
| 172.25.250.163      | None  |
+-----+-----+
[student@workstation ~ (operator1-production)]$ openstack server add floating ip \
production-server2 172.25.250.R
[student@workstation ~ (operator1-production)]$
```

7. Verify that the **production-server2** instance is reachable from **workstation** using the IPv4 floating IP address and that it has an internal IPv4 address assigned. Also verify that **workstation** is reachable from **production-server2** using name resolution.

- 7.1. List the **production-server2** instance and retrieve the associated floating IP address and internal IPv4 address.

```
[student@workstation ~] $ openstack server list \
-c Name -c Networks -f json
[
{
    "Name": "production-server2",
    "Networks": "production-dualstack-
network2=2001:db8:1:X:NNNN, 10.0.0.P, 172.25.250.R"
}
]
```

- 7.2. Verify that the **production-server2** instance is reachable from **workstation** using the **ping** command.

```
[student@workstation ~] $ ping -c3 172.25.250.R
PING 172.25.250.R (172.25.250.R) 56(84) bytes of data.
64 bytes from 172.25.250.R: icmp_seq=1 ttl=63 time=1.57 ms
64 bytes from 172.25.250.R: icmp_seq=2 ttl=63 time=0.812 ms
64 bytes from 172.25.250.R: icmp_seq=3 ttl=63 time=0.964 ms

--- 172.25.250.R ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 0.812/1.117/1.575/0.329 ms
```

- 7.3. Use SSH to log in to **production-server2** using the **cloud-user** user and the floating IPv4 address associated with the instance.

```
[student@workstation ~] $ ssh cloud-user@172.25.250.R
Warning: Permanently added '172.25.250.R' (ECDSA) to the list of known hosts.
[cloud-user@production-server2 ~] $
```

- 7.4. Verify that an internal IPv4 address in the CIDR block of **10.0.0.0/24** is assigned to the instance.

```
[cloud-user@production-server2 ~] $ ip -4 addr show dev eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc pfifo_fast state UP
    qlen 1000
        inet 10.0.0.P/24 brd 10.0.0.255 scope global dynamic eth0
            valid_lft 86048sec preferred_lft 86048sec
```

- 7.5. Verify that **workstation** is reachable from **production-server2** using the **ping** command.

```
[cloud-user@production-server2 ~] $ ping -c3 workstation
PING workstation (172.25.250.254) 56(84) bytes of data.
64 bytes from workstation.lab.example.com (172.25.250.254): icmp_seq=1 ttl=63
time=0.435 ms
64 bytes from workstation.lab.example.com (172.25.250.254): icmp_seq=2 ttl=63
time=0.613 ms
```

```
64 bytes from workstation.lab.example.com (172.25.250.254): icmp_seq=3 ttl=63
time=0.735 ms

--- workstation ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 0.435/0.594/0.735/0.124 ms
```

7.6. Log out of **production-server2**.

```
[cloud-user@production-server2 ~]$ logout
Connection to 172.25.250.R closed.
[student@workstation ~(operator1-production)]$
```

8. Verify that the **production-server2** instance is reachable from **workstation** using the IPv6 address. Use SSH to log in to **production-server2** using the **cloud-user** user and the IPv6 address associated with the instance. Use **ping6** to verify that **workstation** is reachable from **production-server2** using name resolution.

8.1. List the **production-server2** instance and retrieve the associated IPv6 address.

```
[student@workstation ~(operator1-production)]$ openstack server list \
-c Name -c Networks -f json
[
  {
    "Name": "production-server2",
    "Networks": "production-dualstack-network2=2001:db8:1:X:NNNN, 10.0.0.P,
172.25.250.R"
  }
]
```

- 8.2. Verify that the **production-server2** instance is reachable from **workstation** using the **ping** command.

```
[student@workstation ~(operator1-production)]$ ping6 -c3 2001:db8:1:X:NNNN
PING 2001:db8:1:X:NNNN (2001:db8:1:X:NNNN) 56 data bytes.
64 bytes from 2001:db8:1:X:NNNN: icmp_seq=1 ttl=63 time=1.57 ms
64 bytes from 2001:db8:1:X:NNNN: icmp_seq=2 ttl=63 time=0.812 ms
64 bytes from 2001:db8:1:X:NNNN: icmp_seq=3 ttl=63 time=0.964 ms

--- 2001:db8:1:X:NNNN ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 0.929/1.079/1.299/0.158 ms
```

- 8.3. Use SSH to log in to **production-server2** using the **cloud-user** user and the IPv6 address associated with the instance.

```
[student@workstation ~(operator1-production)]$ ssh \
cloud-user@2001:db8:1:X:NNNN
[cloud-user@production-server2 ~]$
```

- 8.4. Verify that an IPv6 address with the **2001:db8:1:X::/64** prefix is assigned to the instance.

```
[cloud-user@production-server2 ~]$ ip -6 addr show dev eth0
```

```
2: eth0: <BROADCAST, MULTICAST, UP, LOWER_UP> mtu 1446 state UP qlen 1000
   inet6 2001:db8:1:X:NNNN/64 scope global mngtmpaddr dynamic
      valid_lft 86389sec preferred_lft 14389sec
   inet6 fe80::NNNN/64 scope link
      valid_lft forever preferred_lft forever
```

- 8.5. Verify that **workstation** is reachable from **production-server2** using the **ping6** command.

```
[cloud-user@production-server2 ~]$ ping6 -c3 workstation
PING workstation(workstation.lab.example.com (2001:db8::5054:ff:fe00:faf6)) 56
  data bytes
  64 bytes from workstation.lab.example.com (2001:db8::5054:ff:fe00:faf6):
    icmp_seq=1 ttl=62 time=0.878 ms
  64 bytes from workstation.lab.example.com (2001:db8::5054:ff:fe00:faf6):
    icmp_seq=2 ttl=62 time=0.944 ms
  64 bytes from workstation.lab.example.com (2001:db8::5054:ff:fe00:faf6):
    icmp_seq=3 ttl=62 time=0.820 ms

  --- workstation ping statistics ---
  3 packets transmitted, 3 received, 0% packet loss, time 2003ms
  rtt min/avg/max/mdev = 0.820/0.880/0.944/0.061 ms
```

- 8.6. Log out of **production-server2**.

```
[cloud-user@production-server2 ~]$ logout
Connection to 2001:db8:1:X:NNNN closed.
[student@workstation ~](operator1-production)]$
```

Evaluation

From **workstation**, run the **lab ipv6-deploying-networks grade** command to confirm the success of this exercise. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab ipv6-deploying-networks grade
```

Cleanup

From **workstation**, run the **lab ipv6-deploying-networks cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab ipv6-deploying-networks cleanup
```

This concludes the lab.

Summary

In this chapter, you learned:

- Using the EUI-64 format, a host automatically assigns itself a unique 64-bit IPv6 interface ID using the network interface MAC address.
- Every IPv6 interface automatically configures a link-local address that only works of a local link on the network. Unlike a link-local scope address, the global scope address is globally routable. Link scope IPv6 addresses use a local prefix, but global link addresses use a prefix provided by a DHCPv6 or by a prefix delegation server.
- The **Neighbor Discovery Protocol** (NDP) allows different nodes on the same link to advertise their existence to their neighbors, and to learn about the existence of their neighbors. Some of the capabilities provided by the NDP protocol are: router discovery, address resolution, duplicate address detection, neighbor unreachability detection, and router redirection.
- The ICMPv6 messages that perform functions offered by NDP are: **Router Solicitation**, **Router Advertisement**, **Neighbor Solicitation**, **Neighbor Advertisement**, and **Redirect**.
- In the router discovery process, each router on a link sends a router advertisement message when a receiving router solicitation message is received. An RA message contains information to perform various NDP functions, such as prefix discovery, parameter discovery, and stateless auto-configuration.
- In the address resolution process, a host seeking out neighbors sends an NS message using the multicast address. Any neighbor that receives the message responds with an NA message containing its own link-layer address.
- When creating a subnet in OpenStack, the **ipv6-ra-mode** option determines how RA messages are sent to instances using the subnet. The **ipv6-address-mode** option determines how instances obtain IPv6 addresses, routes, and other optional network information. Valid modes for the **ipv6-ra-mode** and the **ipv6-address-mode** option are: **dhcpv6-stateful**, **dhcpv6-stateless**, and **slaac**.



CHAPTER 4

PROVISIONING OPENSTACK NETWORKS

Overview	
Goal	Provision tenant networks and provider networks.
Objectives	<ul style="list-style-type: none">• Provision VLAN networks.• Provision VXLAN and GRE tenant networks.• Manage automatic IP address allocation for networks.
Sections	<ul style="list-style-type: none">• Provisioning VLAN Networks (and Guided Exercise)• Provisioning VXLAN and GRE Networks (and Guided Exercise)• Provisioning Provider Networks (and Guided Exercise)
Lab	Provisioning OpenStack Networks

Provisioning VLAN Tenant Networks

Objectives

After completing this section, students should be able to:

- View VLAN packets
- View VLAN tags
- Create a VLAN tenant network

OpenStack VLAN Terminology

A local area network (LAN) is composed of an Ethernet segment enabling direct Layer 2 MAC access between computer nodes. As described in a previous chapter, bridge and switch devices transmit Layer 2 segment traffic through ports managed by flow and access control tables. Communicating between computer nodes on different LANs requires router devices that forward packets between segments through ports controlled by Layer 3 routing tables.

Virtual LAN (VLAN) functionality adds further segregation and Ethernet segment partitioning by including optional packet-based VLAN identifiers (tags), as defined in the IEEE 802.1Q standard, referred to as "Dot1Q". A 32-bit field, placed in the Ethernet frame header, includes a Tag Protocol Identifier (TPID), the value Ox8100, and a VLAN Identifier field indicating the virtual segment to which this frame belongs. Excluding two reserved values, the 12-bit size of the VLAN Identifier allows a maximum of 4094 unique VLANs on a single LAN segment.

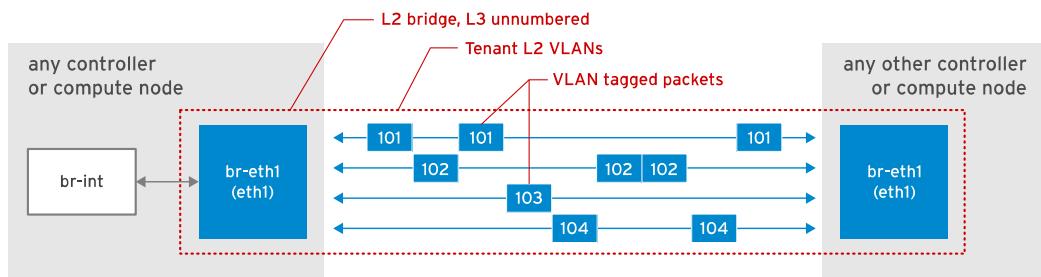


Figure 4.1: VLAN Tagging on a Linux bridge

VLAN technology is ubiquitous in enterprise computing, provided by default in almost all hardware switches and routers, allowing large network scaling without proportionate physical cable expansion. VLAN isolation is a major element of Software Defined Networking in OpenStack, implemented in Open vSwitch bridges to enable and enforce multitenancy security.

OpenStack VLAN Networks

All VLAN operations in OpenStack use 802.1Q tagging, but tagged VLAN segments are limited in range. VLAN boundaries are defined by specific Open vSwitch bridges, where VLAN functionality is configured and translated for each network type. The three network types are described in this list and the following diagram:

- local VLAN (br-int) - each OpenStack controller or compute node is a single LAN segment, subdivided into VLANs managed by that node's single, central integration bridge, br-int.

When a project places networking resources (VMs, load balancers, routers) on a node, br-int is assigned a unique VLAN tag per subnet, labeling all ports connecting to that subnet's resource and namespaces with that VLAN tag. These local VLAN tags never leave the local node.

- internal VLAN (br-vlan) - the internal tenant network connecting all OpenStack nodes to allow a project's network resources (VMs, load balancers, routers) to communicate with each other. VLAN IDs segregate each subnet's traffic. These VLANs extend from the egress bridge (br-vlan) on one node, through VLAN-capable hardware or Linux-implemented virtual tagging, to the ingress bridge on the destination node (that node's br-vlan).
- external VLAN (br-ex) - the external network (br-ex or br-provider) connected to VLAN-capable hardware, which is external to the OpenStack infrastructure. External (public) networks may be connected to tenant networks using Open vSwitch project routers, or administrators may configure and deploy project network resources directly on external networks when configured as provider networks that do not require routers.

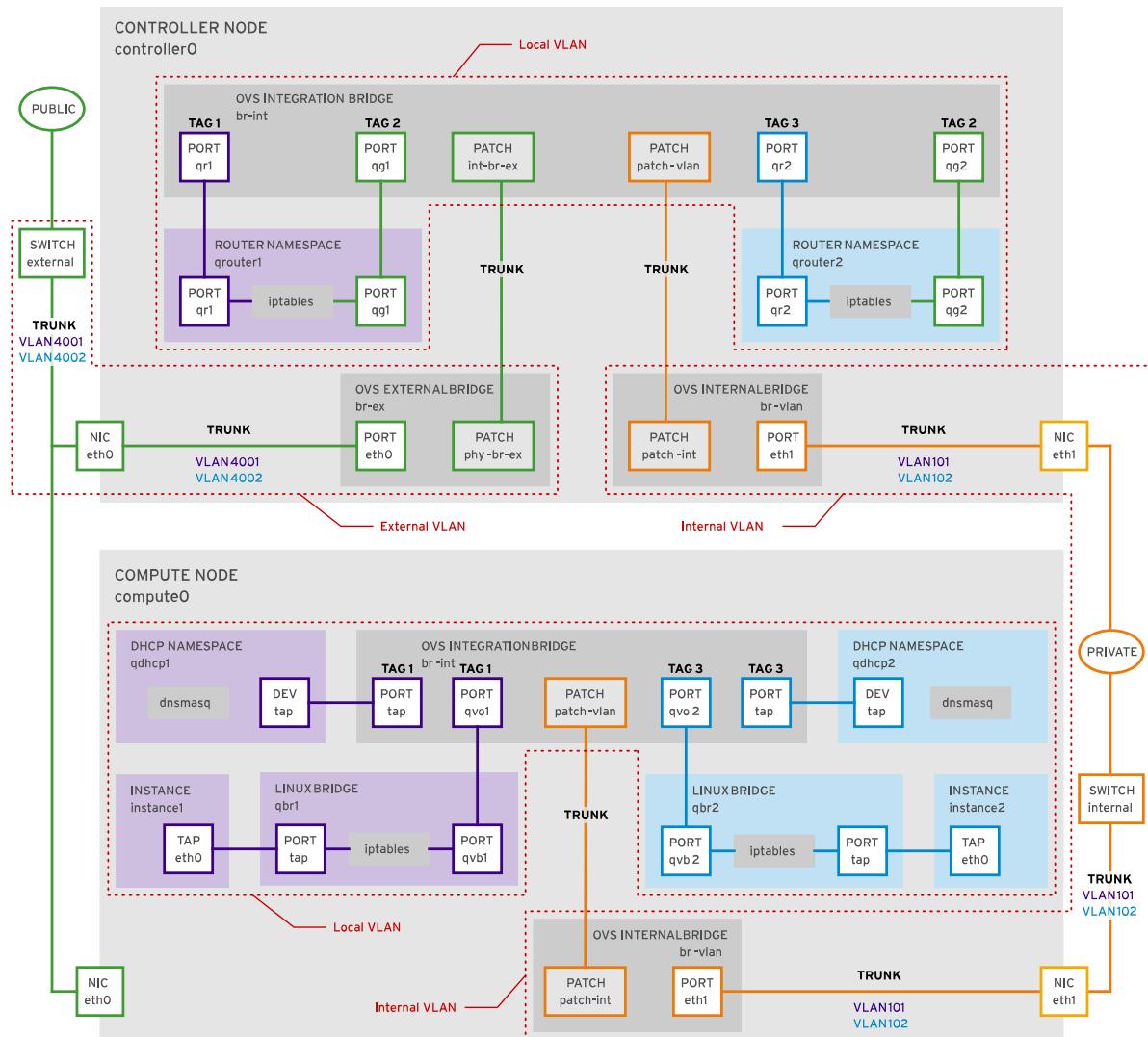


Figure 4.2: External, internal and local VLAN boundaries

Managing VLAN Networks

To configure the internal tenant network for VLAN, set parameters for the ML2 plug-in on controllers and the Open vSwitch agent on controllers and compute nodes. The example here also sets external network with the name datacenter and arbitrary VLAN IDs. This configuration matches the diagram.

```
[demo@controller0 ~]$ sudo cat /etc/neutron/plugins/ml2/ml2_conf.ini
...output omitted...
[ml2]
type_drivers = flat,vlan
tenant_network_types = vlan
mechanism_drivers = openvswitch
[ml2_type_vlan]
network_vlan_ranges = datacenter:4001:4050,physnet1:101:200

[demo@controller0 ~]$ sudo cat /etc/neutron/plugins/ml2/openvswitch_agent.ini
...output omitted...
[ovs]
integration_bridge = br-int
bridge_mappings = datacenter:br-ex,physnet1:br-vlan
```

When a new VLAN-enabled network is created, either on an internal network by a self-service user or a provider network by an administrator, a VLAN ID is assigned from the configured **network_vlan_ranges** values and set as the **provider:segmentation_id** parameter value. The **segmentation_id** field is used by all network types (VLAN, VXLAN, and GRE) to hold their protocol ID. Use the **openstack network show** command to view this parameter for a given network, regardless of the protocol displayed in the **network_type** field.

```
[demo@workstation ~]$ openstack network show 6fce40cf-7da3-4242-938a-3c7043dc2b6a
+-----+-----+
| Field | Value |
+-----+-----+
| name | research-vlan |
| provider:network_type | vlan |
| provider:physical_network | datacenter |
| provider:segmentation_id | 4001 |
...output omitted...
```

At the same time, Open vSwitch assigns local VLAN tags to the br-int ports, connecting the new project resources and deployed VMs. VLAN tags are only registered per node. The VLAN tag value for one subnet is not expected to match the VLAN tag used on another node for the same subnet. The following edited output displays local VLAN tags assigned to br-int ports:

```
[demo@controller0 ~]$ sudo ovs-vsctl show
Bridge br-int
  Port "qg-10a51d6b-fa"
    tag: 4
  Port "tap29b1c27e-6a"
    tag: 2
  Port "qr-fff7ec07-5d"
    tag: 1
  Port "sg-00ece121-4f"
    tag: 3
  Port "qg-eb834ed3-13"
    tag: 4
...output omitted...
```

VLAN use requires that associated physical network resources, such as switches, routers, and network interfaces, support VLAN-tagged traffic. The OpenStack Networking Service usually manages tagging using arbitrary VLAN IDs configured manually. Many vendors provide ML2 plugins to interface with their brand of switches and routers. When the plugins are correctly configured with the network device's management access parameters, the Networking Service obtains valid VLAN IDs directly from the hardware when new project networks are requested, and instructs the hardware device to create and manage the corresponding VLAN. When using this technique, set the **network_VLAN_ranges** parameter to **provider** without numeric values, since the values are provided by the vendor plugin.

OpenStack Isolated Network Installation

An OpenStack installation requires multiple separate networks to segregate and manage each different form of network traffic. Previous Red Hat OpenStack Platform System Administration courses have introduced these concepts. Examples of network traffic include provisioning, external, management, storage data, storage control, tenant, and internal API. To reduce the number of required physical or virtual NICs, the OpenStack TripleO Deployment provides a technique for the internal, private traffic types to share one or more NICs, which can be bonded or teamed for resilient scaling.

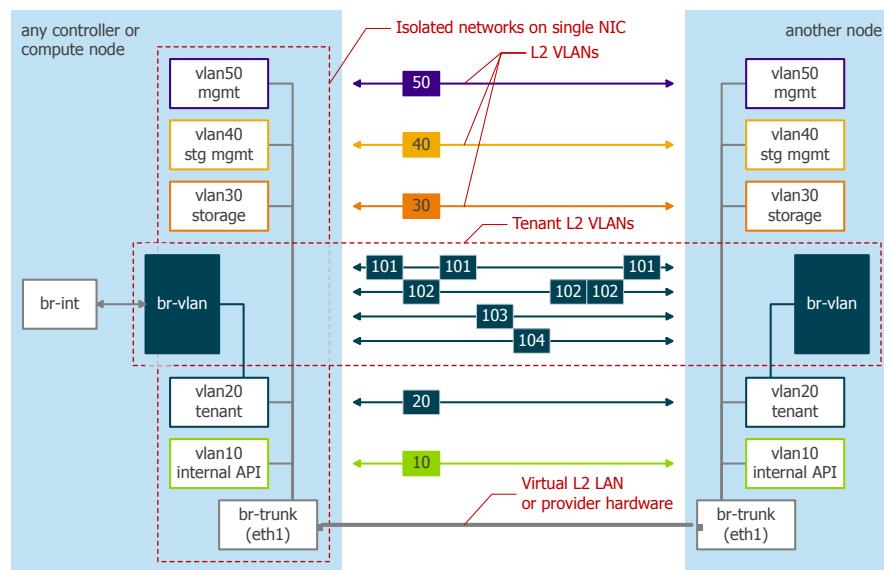


Figure 4.3: VLAN Tagging on stacked OVS bridges

This Red Hat OpenStack Platform System Administration course uses this technique, which is referred to as network isolation on a single-nic-vlans configuration. As the above diagram shows, the tenant network is a VLAN on the br-trunk bridge managing the eth1 interface. All tenant traffic traverses the eth1 interface using the VLAN tag 20.

When the tenant network is further partitioned using VLAN tags, Ethernet frames are being tagged twice. This technique is the Ethernet standard IEEE 802.1ad, informally referred to as QinQ, but also known as "provider bridging" or "stacked VLANs". When stacked, the outer tag sets the TPID to Ethertype 0x88A8 instead of 0x8100. View the Ethernet 802.1ad Double Tagged Frame diagram:

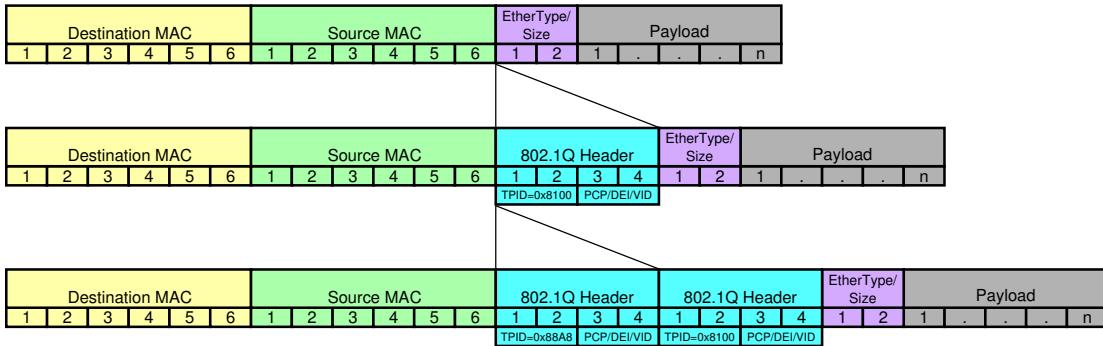


Figure 4.4: Ethernet 802.1ad showing double tagged frame fields By User: Luca Ghio [CC BY-SA 4.0]



Note

Use of the VLAN technique for segregating project tenant traffic has been increasingly replaced by use of VXLAN or GRE tunneling, which offers greater flexibility and scaling. These techniques are discussed in the next section. This classroom configuration does use network isolation to share the internal eth1 interface, but tenant traffic does not use a br-vlan bridge or stacked VLANs. Instead, the internal bridge uses br-tun and tunneling on top of the isolated VLANs. The diagram is only intended to support the QinQ lecture.

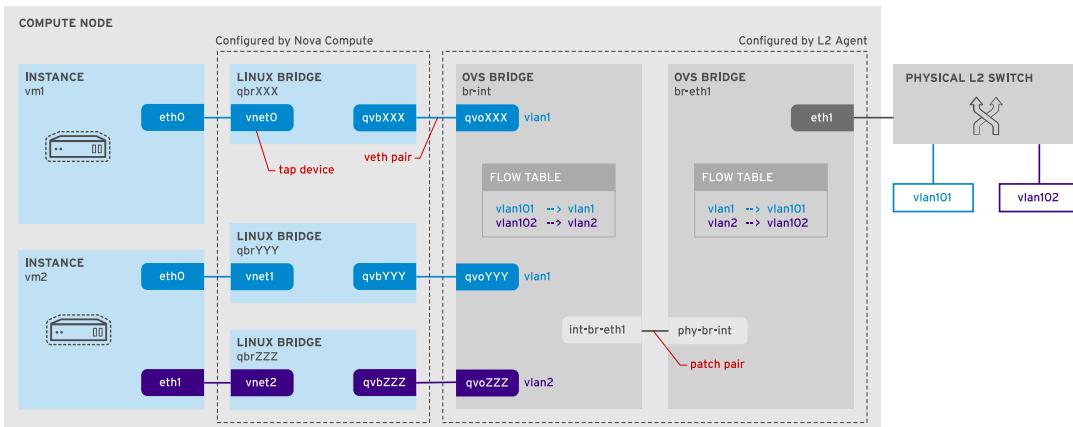


Figure 4.5: VLAN Tagging on a single bridge

The previous image shows that tenant flows are separated by locally-assigned VLAN IDs on br-int. When the traffic reaches br-ex, the locally-assigned VLAN IDs become tenant network VLAN IDs on the internal backbone. On bridge br-int, the VLAN ID is converted using OpenFlow rules, for example `dl_vlan=101 -> mod_vlan_vid:1` for incoming traffic. For outgoing traffic, bridge br-ex also translates using OpenFlow. For example, `mod_vlan_vid:1 -> dl_vlan=101`.

VLAN Tenant Networks

The following steps outline the process for creating a VLAN network and using the `ovs-ofctl` command to view OpenFlow rules.

1. Create a network of type **vlan** using the **datacenter** physical network connect to the **br-ex** bridge and using a VLAN ID within the configured range.
2. Create a subnet on the network.
3. Create a router. Set the gateway. Attach the network as an interface.
4. Use **ovs-ofctl** to view OVS rules on the bridges. Look for rules related to the segmentation ID on bridge **br-ex**.
5. Locate ports and local vlan tags referenced in the rules.
6. Use **ovs-ofctl** to view OVS rules on bridge **br-int**.
7. Locate ports and local vlan tags referenced in the rules.



References

Further information is available in the chapter on Configure bridge mappings in the *Red Hat OpenStack Platform 10 Networking Guide* at

| <https://access.redhat.com/documentation/en/red-hat-openstack-platform/>

Guided Exercise: Provisioning VLAN Tenant Networks

In this exercise, you will create a VLAN tenant network with a subnet, attach it to a router interface, and create a server instance on the new network. You will view both the external segmentation ID and the local VLAN tag for this VLAN network.

Outcomes

You should be able to:

- Create a VLAN network.
- View VLAN packets.

Before you begin

Log in to **workstation** as **student**, using **student** as the password. Run the **lab provisioning-vlan setup** command. This script verifies that the overcloud nodes are accessible, OpenStack services are running, and the correct projects, external networks, and routers are already configured.

```
[student@workstation ~]$ lab provisioning-vlan setup
```

Steps

1. Source the **architect1-finance** credentials. Create a VLAN tenant network using the **openstack network create** command with the **--provider-network-type** option. Create a subnet for this VLAN network. Attach the network to the **finance-router1** router.



Note

Tenant networks are normally created on the internal (private) network OVS bridges, segregated from external (public) network switches. To simplify this chapter's network design, the VLAN tenant network created here will be attached to the VLAN-capable physical network named **datacentre**, available through the **br-ex** external bridge, because the internal network **br-tun** bridge is currently configured for only tunneling protocols.

With a VLAN configuration, this tenant network shares the external physical network, but is isolated from other network traffic, therefore requiring a router to connect to the external 172.25.250.0/24 public network.

- 1.1. Create a VLAN tenant network called **finance-vlan-network1**.

Make note of the network's **id** and **provider:segmentation_id** fields, as you will need these values in later steps. In this example, the **segmentation_id** is displayed as the letter **S**.

```
[student@workstation ~]$ source ~/architect1-finance-rc
[student@workstation ~(architect1-finance)]$ openstack network create \
```

```
--provider-network-type vlan \
finance-vlan-network1
+-----+
| Field | Value |
+-----+
| admin_state_up | UP |
| id | 97f748ca-08db-4081-bdb5-c7cc2996d56a |
| name | finance-vlan-network1 |
...output omitted...
| provider:network_type | vlan |
| provider:physical_network | datacentre |
| provider:segmentation_id | S |
...output omitted...
```

1.2. Create a subnet for the VLAN network called **finance-vlan-subnet1**.

Make note of the subnet's **id** field, as you will need this value in later steps.

```
[student@workstation ~(architect1-finance)]$ openstack subnet create \
--network finance-vlan-network1 \
--subnet-range=192.168.3.0/24 \
--dns-nameserver=172.25.250.254 \
--dhcp finance-vlan-subnet1
+-----+
| Field | Value |
+-----+
| allocation_pools | 192.168.3.2-192.168.3.254 |
| cidr | 192.168.3.0/24 |
...output omitted...
| enable_dhcp | True |
| gateway_ip | 192.168.3.1 |
| id | a16b9cf4-0d84-4a6f-b4e2-f4730adeaeed |
| ip_version | 4 |
| name | finance-vlan-subnet1 |
| network_id | 97f748ca-08db-4081-bdb5-c7cc2996d56a |
...output omitted...
```

1.3. Make note of the **finance-router1** ID, as you will need this value in later steps.

```
[student@workstation ~(architect1-finance)]$ openstack router list \
-c ID -c Name
+-----+
| ID | Name |
+-----+
| 779631a4-1dea-4fb6-8db4-181ebd0d9248 | production-dvr-router1 |
| 9634e23f-e70f-4e32-8691-8af0167dda39 | research-router1 |
| a8ae4cbe-84c0-4127-b8de-898fd94b5379 | production-router1 |
| b07cc523-105d-418b-ac2c-cb620a2ada49 | finance-router1 |
| e8d3e5a5-b35e-4a36-a912-755ac9144284 | finance-dvr-router1 |
| e9e25f24-75b4-4439-86da-499ab2bd572a | research-dvr-router1 |
+-----+
```

1.4. Attach the VLAN network as an interface on the **finance-router1** router.

```
[student@workstation ~(architect1-finance)]$ openstack router add \
subnet finance-router1 finance-vlan-subnet1
```

- 1.5. List all port addresses on this VLAN subnet by filtering with the subnet's ID. The two addresses listed are the connected router's interface and the DHCP namespace's TAP device. You will verify these addresses in upcoming steps.

```
[student@workstation ~-(architect1-finance)]$ openstack port list \
-c "Fixed IP Addresses" | grep a16b9cf4-0d84-4a6f-b4e2-f4730adeaeed
ip_address='192.168.3.1', subnet_id='a16b9cf4-0d84-4a6f-b4e2-f4730adeaeed'
ip_address='192.168.3.2', subnet_id='a16b9cf4-0d84-4a6f-b4e2-f4730adeaeed'
```

2. Create a server instance named **finance-vlan-server1**. Make note of the compute node on which the instance is hosted, and also the fixed IP address assigned, displayed here as **192.168.3.N**.

```
[student@workstation ~-(architect1-finance)]$ openstack server create \
--flavor default \
--key-name example-keypair \
--nic net-id=finance-vlan-network1 \
--image rhel7 \
--wait finance-vlan-server1
+-----+
| Field           | Value          |
+-----+
...output omitted...
| OS-EXT-SRV-ATTR:host      | compute1.overcloud.example.com |
| OS-EXT-SRV-ATTR:hypervisor_hostname | compute1.overcloud.example.com |
| OS-EXT-SRV-ATTR:instance_name | instance-00000005 |
| addresses        | finance-vlan-network1=192.168.3.N |
| name            | finance-vlan-server1 |
...output omitted...
```

3. Open a second terminal window or tab for working on the **controller** node, leaving the first **workstation** terminal open and ready. On **controller0**, use the **ovs-ofctl** and **ovs-vsctl** commands to view OpenFlow rules for this VLAN tenant network.

- 3.1. In the second terminal, log in as **root** to the controller node. Use the **ovs-ofctl** command to view OpenFlow rules on the **br-ex** bridge.

Locate rules that modify the VLAN ID field on outgoing packets, and that contain the **segmentation_id**, which you noted earlier. In this example, when the outgoing packet arrives on this bridge's port **2**, and the incoming VLAN ID (tag) is **T**, the rule would modify the VLAN ID (tag) to be the destination physical network VLAN ID (Ethernet segmentation ID) **S**.

```
[student@workstation ~]$ ssh root@controller0
[root@controller0 ~]# ovs-ofctl dump-flows br-ex | grep mod_vlan_vid
cookie=0x94490524cb206772, duration=125792.528s, table=2,
n_packets=6298, n_bytes=578477, idle_age=52, hard_age=65534, priority=4,
in_port=2,d1_vlan=T actions=mod_vlan_vid:S,NORMAL
```

- 3.2. Use the **ovs-ofctl** command to show that **br-ex** incoming port **2** is named **phy-br-ex**, one end of the patch pair connected to bridge **br-int**, from which the outgoing packet was received.

```
[root@controller0 ~]# ovs-ofctl show br-ex
...output omitted...
1(eth2): addr:52:54:00:02:fa:01
    config:   0
    state:    0
    speed: 0 Mbps now, 0 Mbps max
2(phy-br-ex): addr:72:a0:11:a0:be:85
    config:   0
    state:    0
    speed: 0 Mbps now, 0 Mbps max
...output omitted...
```

- 3.3. Use the **ovs-ofctl** command to view the OpenFlow rules on the **br-int** bridge. Locate rules that modify the VLAN ID field on incoming packets, and that contain the **segmentation_id**, which you noted earlier. In this example, when the incoming packet arrives on this bridge's port **2**, and the incoming datalink VLAN ID (Ethernet segmentation ID) is **S**, the rule modifies the VLAN ID (tag) for use on the local VLAN **T**.

```
[root@controller0 ~]# ovs-ofctl dump-flows br-int | grep mod_vlan_vid
cookie=0x8db0e128c370450a, duration=125797.708s, table=0,
  n_packets=6391, n_bytes=604992, idle_age=57, hard_age=65534, priority=3,
  in_port=1,dl_vlan=S actions=mod_vlan_vid:T,NORMAL
...output omitted...
```

- 3.4. Use the **ovs-ofctl** command to show that **br-int** incoming port **1** is named **int-br-ex**, one end of the patch pair connected to bridge **br-ex**, from which the incoming packet was received.

```
[root@controller0 ~]# ovs-ofctl show br-int
...output omitted...
1(int-br-ex): addr:1e:35:2e:6b:d8:d8
    config:   0
    state:    0
    speed: 0 Mbps now, 0 Mbps max
2(patch-tun): addr:52:59:d7:54:0d:b3
    config:   0
    state:    0
    speed: 0 Mbps now, 0 Mbps max
...output omitted...
```

- 3.5. Use the **ovs-vsctl** command to show that local VLAN ID **T** is being used to tag local components for this VLAN tenant network. Replace **T** with the tag value (**mod_vlan_vid**) discovered in the **ovs-ofctl** command.

```
[root@controller0 ~]# ovs-vsctl show | grep "tag: T$" -B1
  Port "qr-044f6251-62"
    tag: T
  --
  Port "taped3d5fce-49"
    tag: T
```

- 3.6. Look inside this VLAN subnet's DHCP namespace to list network interfaces. The namespace name is **dhcp-** plus the network ID noted earlier. The TAP device interface

listed will match the one listed in the previous command. The TAP device IP address is one of two belonging to this VLAN network, noted earlier.

```
[root@controller0 ~]# ip -n \
qdhcp-97f748ca-08db-4081-bdb5-c7cc2996d56a addr show taped3d5fce-49
61: taped3d5fce-49: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1496 qdisc noqueue
    state UNKNOWN qlen 1000
        link/ether fa:16:3e:ad:f0:c8 brd ff:ff:ff:ff:ff:ff
        inet 192.168.3.2/24 brd 192.168.3.255 scope global taped3d5fce-49
            valid_lft forever preferred_lft forever
        inet6 fe80::f816:3eff:fead:f0c8/64 scope link
            valid_lft forever preferred_lft forever
```

- From the DHCP namespace, ping the **finance-vlan-server1** IP address. Leave this **ping** running so that it can be viewed with **tcpdump**.

```
[root@controller0 ~]# ip netns exec \
qdhcp-97f748ca-08db-4081-bdb5-c7cc2996d56a ping 192.168.3.N
PING 192.168.3.12 (192.168.3.N) 56(84) bytes of data.
64 bytes from 192.168.3.N: icmp_seq=1 ttl=64 time=1.33 ms
64 bytes from 192.168.3.N: icmp_seq=2 ttl=64 time=0.660 ms
...output omitted...
```

- Leave **ping** running in the **controller0** terminal. Return to the **workstation** window for the next steps. Find the other device (port ID **qr-044f6251-62**) listed by **ovs-vsctl**. The port ID is a short version of the full port ID prefixed with **qr-**. Use only the portion *after* the prefix to grep for the full port ID.

```
[student@workstation ~(architect1-finance)]$ openstack port list \
-c ID -c "Fixed IP Addresses" | grep 044f6251-62
| 044f6251-625b-4262-af80-ef834656bcd4 |
ip_address='192.168.3.1', subnet_id='a16b9cf4-0d84-4a6f-b4e2-f4730adeaeed' |
```

- Use the full port ID to locate the device, and use the device ID to locate the device name. In summary, this interface on local VLAN tag **T** belongs to your VLAN tenant network and is on that network's router.

```
[student@workstation ~(architect1-finance)]$ openstack port show \
044f6251-625b-4262-af80-ef834656bcd4 -c device_id -c device_owner
+-----+-----+
| Field | Value |
+-----+-----+
| device_id | b07cc523-105d-418b-ac2c-cb620a2ada49 |
| device_owner | network:router_interface |
+-----+-----+
[student@workstation ~(architect1-finance)]$ openstack router show \
b07cc523-105d-418b-ac2c-cb620a2ada49 -c name
+-----+-----+
| Field | Value |
+-----+-----+
| name | finance-router1 |
+-----+-----+
```

On **controller0**, the local VLAN tag **T** corresponds to external VLAN network segmentation ID **S**. The IDs are expected to be different for each unique installation.

4. Open a third terminal window or tab for working on the **compute** node, leaving the first **workstation** terminal open and ready. The **ping** command is still running in the second **controller0** terminal. Log in to the compute node hosting the instance. Use **tcpdump** to view the VLAN traffic.

- 4.1. In the third terminal, log in as **root** to the compute node. Use the **ovs-ofctl** command to list the OpenFlow rules in use on the **br-ex** bridge.

Locate rules that modify the VLAN ID field on outgoing packets, containing the **segmentation_id**, which you noted earlier. In this example, when the outgoing packet arrives on this bridge's port **2**, and the incoming VLAN ID (tag) is **T**, the rule modifies the VLAN ID (tag) to be the destination physical network VLAN ID (Ethernet segmentation ID) **S**.

```
[student@workstation ~]$ ssh root@compute1
[root@compute1 ~]# ovs-ofctl dump-flows br-ex | grep mod_vlan_vid
cookie=0x94490524cb206772, duration=125792.528s, table=2,
n_packets=6298, n_bytes=578477, idle_age=52, hard_age=65534, priority=4,
in_port=2,dl_vlan=T actions=mod_vlan_vid:S,NORMAL
```



Note

Note that the local VLAN tag for this VLAN tenant network on **controller0** is not the same value as the local VLAN tag on **compute1**. Local VLAN tags are unique per OpenStack node and are never shared outside that node. Only the physical network VLAN ID ("segmentation ID") is shared for tagging non-local traffic for VLAN tenant networks.

- 4.2. Use the **ovs-ofctl** command to show that **br-ex** incoming port **2** is named **phy-br-ex**, one end of the patch pair connected to bridge **br-int**, from which the outgoing packet was received.

```
[root@compute1 ~]# ovs-ofctl show br-ex
...output omitted...
1(eth2): addr:52:54:00:02:fa:0c
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
2(phy-br-ex): addr:5e:ef:e5:95:3b:10
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
...output omitted...
```

- 4.3. Use the **ovs-ofctl** command to view the OpenFlow rules on the **br-int** bridge. Locate rules that modify the VLAN ID field on incoming packets, and that contain the **segmentation_id**, which you noted earlier. In this example, when the incoming packet arrives on this bridge's port **2**, and the incoming datalink VLAN ID (Ethernet segmentation ID) is **S**, the rule modifies the VLAN ID (tag) for use on the local VLAN **T**.

```
[root@compute1 ~]# ovs-ofctl dump-flows br-int | grep mod_vlan_vid
cookie=0x8db0e128c370450a, duration=125797.708s, table=0,
```

```
n_packets=6391, n_bytes=604992, idle_age=57, hard_age=65534, priority=3,
in_port=1,dl_vlan=S actions=mod_vlan_vid:T,NORMAL
...output omitted...
```

- 4.4. Use the **ovs-ofctl** command to show that **br-int** incoming port **1** is named **int-br-ex**, one end of the patch pair connected to bridge **br-ex**, from which the incoming packet was received.

```
[root@compute1 ~]# ovs-ofctl show br-int
...output omitted...
1(int-br-ex): addr:5e:42:da:c2:f7:05
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
2(patch-tun): addr:da:1b:57:dd:4c:dd
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
...output omitted...
```

- 4.5. Use the **ovs-vsctl** command to show that local VLAN ID **T** is being used to tag local components for this VLAN tenant network. Replace **T** with the tag value (**mod_vlan_vid**) discovered in the **ovs-ofctl** command.

```
[root@compute1 ~]# ovs-vsctl show | grep "tag: T$" -B1
    Port "qvod956ea09-dd"
        tag: T
```

- 4.6. Use the **workstation** terminal for this step. Find the device referenced by the port ID ("**qvod956ea09-dd**") listed by **ovs-vsctl**. The port ID shown is a short version of the full port ID, prefixed with the veth pair endpoint designation **qvo**. Use only the portion after the prefix to grep for the full port ID. In summary, this veth pair endpoint on local VLAN tag **T** belongs to your VLAN tenant network and connects to the Linux bridge fronting the server instance.

```
[student@workstation ~(architect1-finance)]$ openstack port list \
-c ID -c "Fixed IP Addresses" | grep d956ea09-dd
| d956ea09-ddba-4d65-ae58-6f1853686b67 |
ip_address='192.168.3.N', subnet_id='a16b9cf4-0d84-4a6f-b4e2-f4730adeaeed' |
```

- 4.7. Return to the **compute1** terminal. Use **tcpdump** to view the ping packets on this VLAN tenant network. Interrupt the command as soon as it displays ICMP output. The packets are Ethernet 802.1Q VLAN tagged with the VLAN's segmentation ID. After viewing the packets, exit the compute node, then exit the extra third terminal window.

```
[root@compute1 ~]# tcpdump -ten -i eth2 | grep ICMP
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth2, link-type EN10MB (Ethernet), capture size 262144 bytes
fa:16:3e:ad:f0:c8 > fa:16:3e:ee:e3:7b, ethertype 802.1Q (0x8100), length 102:
  vlan S, p 0, ethertype IPv4, 192.168.3.2 > 192.168.3.N: ICMP echo request, id
  4304, seq 936, length 64
fa:16:3e:ee:e3:7b > fa:16:3e:ad:f0:c8, ethertype 802.1Q (0x8100), length 102:
  vlan S, p 0, ethertype IPv4, 192.168.3.N > 192.168.3.2: ICMP echo reply, id
  4304, seq 936, length 64
```

```
fa:16:3e:ad:f0:c8 > fa:16:3e:ee:e3:7b, ethertype 802.1Q (0x8100), length 102:  
vlan S, p 0, ethertype IPv4, 192.168.3.2 > 192.168.3.N: ICMP echo request, id  
4304, seq 937, length 64  
fa:16:3e:ee:e3:7b > fa:16:3e:ad:f0:c8, ethertype 802.1Q (0x8100), length 102:  
vlan S, p 0, ethertype IPv4, 192.168.3.N > 192.168.3.2: ICMP echo reply, id  
4304, seq 937, length 64  
Ctrl+C  
  
[root@compute1 ~]# exit  
logout  
Connection to compute1 closed.  
[student@workstation ~]$ exit
```

5. Return to the **controller** terminal. Interrupt the running **ping** command. Exit the compute node, then exit the extra second terminal window.

```
...output omitted...  
64 bytes from 192.168.3.N: icmp_seq=506 ttl=64 time=0.709 ms  
64 bytes from 192.168.3.N: icmp_seq=507 ttl=64 time=0.553 ms  
64 bytes from 192.168.3.N: icmp_seq=508 ttl=64 time=0.942 ms  
64 bytes from 192.168.3.N: icmp_seq=509 ttl=64 time=0.548 ms  
Ctrl+C  
[root@controller0 ~]# exit  
logout  
Connection to controller0 closed.  
[student@workstation ~]$ exit
```

Cleanup

From **workstation**, run the **lab provisioning-vlan cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab provisioning-vlan cleanup
```

This concludes the guided exercise.

Provisioning VXLAN and GRE Tenant Networks

Objectives

After completing this section, students should be able to:

- Discuss network encapsulation.
- View network encapsulation packets.
- Manage GRE tenant networks.

Expanding Beyond VLAN

The 12-bit VLAN ID field allows for 4096 unique values. This value appears sufficiently large for a single location, single organization OpenStack installation. A total of 4094 unique subnets could be created in one or more tenant projects on either self-service or provider network.

However, consider an OpenStack installation supporting multiple enterprises, such as a telco or cloud service provider, or a single OpenStack installation distributed across multiple geographic regions through a Wide Area Network (WAN). In this scenario, the 4094 VLAN limit is reached quickly. VLAN segregation is an optional tag field inserted into an Ethernet frame on a single LAN segment, so tenant networks using VLAN rely on 802.1Q-supported hardware and WAN connections to stretch single VLANs across multiple OpenStack node locations. Without such equipment, a single tenant VLAN is limited to the collision domain of a set of stackable enterprise switches.

For this reason, many OpenStack installations are moving away from using VLAN for tenant network segregation.

To expand beyond the VLAN limit, sites use an overlay protocol that transports L2 traffic across existing IP (L3) networks. Overlay protocols effectively carry the original network packet as a payload with new headers. The original packet is encapsulated at the source, and then decapsulated at the destination. This tunneling process results in an unaltered original packet at the destination, where it is treated as being on the same VLAN as at the source. Two protocols are common for OpenStack tenant networks: VXLAN and GRE. Also, a new encapsulation data format, known as GENEVE, is expected to standardize and supersede other tunneling protocols in future OpenStack versions.

Virtual Extensible LAN (VXLAN)

Virtual Extensible LAN (VXLAN) encapsulates the original L2 Ethernet frame in a new L4 UDP datagram, configured for service at IANA-assigned port 4789. The beginning and end of a VXLAN tunnel are known as VXLAN tunnel endpoints (VTEPs), which act as a service at UDP port 4789. In OpenStack, VTEP functionality is included in the Open vSwitch tunnel bridge (br-tun) controlling the internal tenant network NIC.

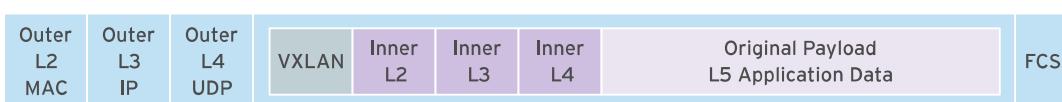


Figure 4.6: VXLAN frame details

VXLAN frame details

Header	Pertinent Fields	Comment
Outer L2	Destination MAC Address	Normal hop-to-hop L2 header.
	Source MAC Address	
	EtherType 0x0800 (IPv4) or IPv6 (0x86dd)	
Outer L3	Protocol 0x11 (UDP)	The source and destination addresses are VTEP endpoints on the OVS tunnel bridge (br-tun).
	Source IP Address	
	Destination IP Address	
Outer L4	UDP Source Port (VXLAN 4789)	The VTEP service port.
VXLAN	Flags and reserved bits (usually 0001000)	The 24-bit VNI field allows 16 million possible tunnel identifiers.
	VXLAN network identifier (VNI)	
Original Frame	L2, L3, L4 Headers	Carried as payload, unmodified during transport.
	L5 Application Data	

VXLAN uses UDP to encapsulate each packet individually, eliminating the need for TCP session setup and stream management. The 24-bit VNI field in VXLAN allows for up to 16 million possible logical networks, but most enterprise hardware routers have limits to the number of tunnels they can track concurrently. Unlike VLAN, VXLAN does not require internal tenant network hardware switches and routers to create, manage, or process VLAN tags. Open vSwitch will continue to use VLANs locally within each node to segregate networks that traverse integration bridges. The Networking Service stores the protocol-specific network ID in the provider's segmentation ID field. For VXLAN-configured networks, the segmentation ID is the VXLAN Network identifier (VNI).

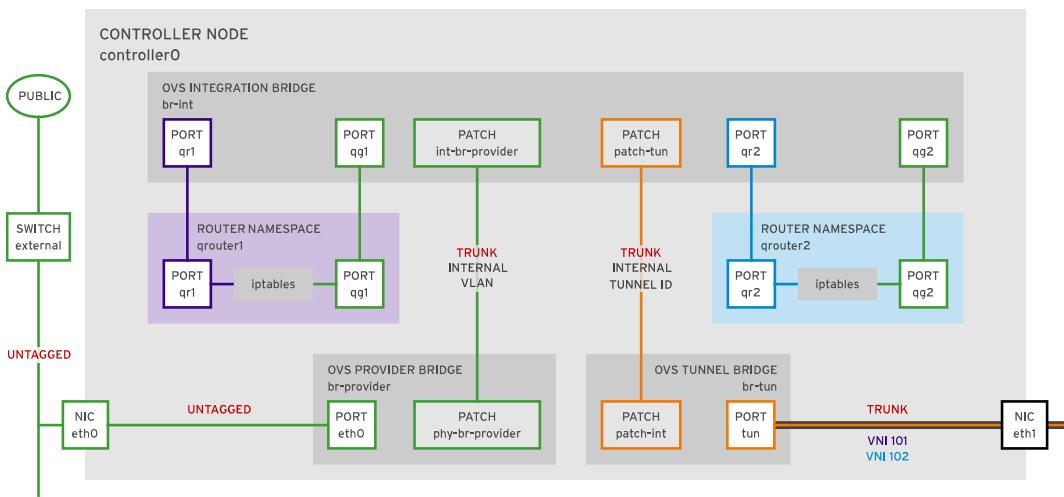


Figure 4.7: VXLAN self-service network on controller node

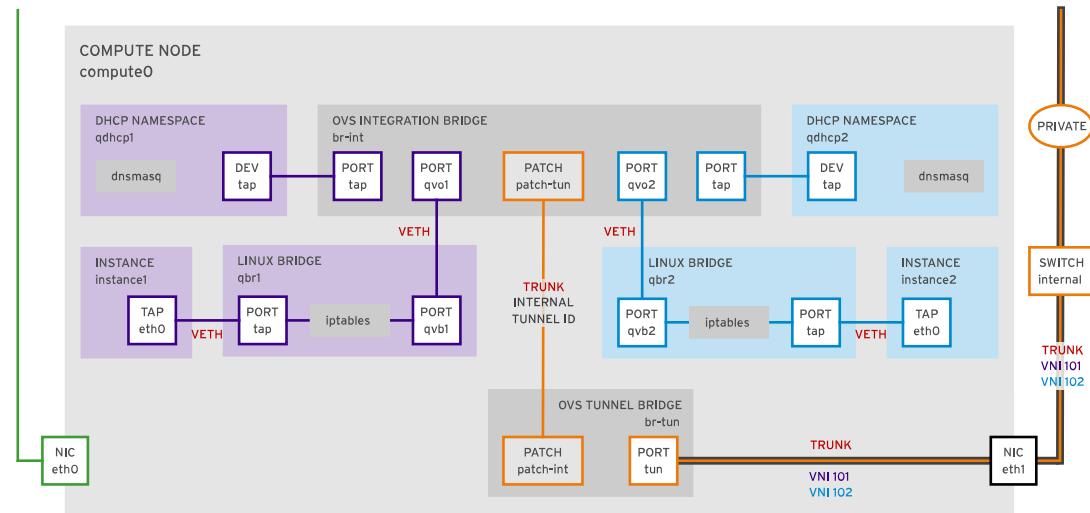


Figure 4.8: VXLAN self-service network on compute node

The diagrams show an OpenStack installation with the default VXLAN internal tenant network. The external is configured as a provider flat network. This configuration supports self-service and provider-based networks concurrently.

Generic Routing Encapsulation (GRE)

GRE also encapsulates network packets and transports them over a Layer 3 network to an endpoint, which decapsulates them to the original packet for final routing. This creates a virtual point-to-point between the source and destination switches. GRE tunnels are stateless; each tunnel endpoint is unaware about the availability of the remote endpoint. Therefore, the state of a GRE tunnel cannot reflect whether an endpoint is reachable. If packet delivery is attempted and fails, normal ICMP replies are the only indication of network failure.

Unlike VXLAN, GRE does not encapsulate the full original frame as payload in a new packet. Instead, GRE inserts a new IP header and a GRE option header in front of the original packet's IP header. Devices that recognize GRE headers then ignore remaining headers during transport. When the packet arrives at the destination endpoint, the outer IP and GRE headers are removed, resulting in the original source packet to be delivered locally. As a tunnel protocol, GRE does not require internal tenant network hardware switches and routers to create, manage, or process VLAN tags.

The GRE protocol specifies a 32-bit Key field for identifying tunnel traffic flows, split into a 24-bit Virtual Subnet ID (VSID) and an 8-bit FlowID for managing per-flow entropy. Although a 24-bit number can create over 16 million values, the number of concurrent GRE tunnels is limited, when using hardware devices for GRE endpoints, by the number of interface structures a single device can maintain. In OpenStack, the OVS br-tun switch provides the tunnel endpoint without this arbitrary limitation. The FlowID field provides context information for defining logical traffic flows. Packets belonging to a traffic flow are encapsulated using the same **Key** value. At the destination tunnel endpoint, decapsulating the packet uses the **Key** value to identify packets belonging to a traffic flow. The VSID (tunnel ID) is stored in the network's **provider:segmentation_id** parameter when the tunnel is created.

Viewing GRE packets displays protocol type 0x6558, specifying the use of the Transparent Ethernet Bridging (TEB) algorithm. Packets delivered to the br-tun switch are decapsulated and transparently forwarded to br-int for further delivery. TEB does not interpret the IP routing

header, but uses a table of learned addresses to determine the next hop port. Since the patch connection to br-int is configured as trunk ports, the decapsulated packets are transferred across the patch pair without modification to br-int, where OVS flow table rules determine the local VLAN tag assignment for the packet header.



Figure 4.9: GRE frame details

GRE

Header	Pertinent Fields	Comment
L2	Destination MAC Address	Normal hop-to-hop L2 header.
	Source MAC Address	
	EtherType 0x0800 (IPv4) or IPv6 (0x86dd)	
Carrier L3	Protocol 0x47 (GRE)	The source and destination addresses are GRE endpoints on the OVS tunnel bridge (br-tun).
	Source IP Address	
	Destination IP Address	
GRE	Protocol 0x0800 (IPv4) or IPv6 (0x86dd)	Specifies the next header to facilitate accurate endpoint processing. The 24-bit portion of the Key field allows 16 million possible tunnel identifiers.
	Key (RFC1701-compliant)	
Original Packet	L3, L4 Headers	The GRE header causes remaining headers to be ignored during transport.
	L5 Application Data	

Capturing GRE Traffic

The following steps outline the process for capturing and analyzing GRE packets.

1. List the network ID, the server ID, and the compute host for the server instance.
2. Open a second terminal window. On the compute node hosting the server instance, use tcpdump to capture GRE packets.
3. Open a third terminal window. On controller, from the DHCP namespace for this network ID, ping the server instance.
4. View the captured GRE packets.

Generic Network Virtualization Encapsulation (GENEVE)

A new network standard is being previewed for cloud network tunneling. The Generic Network Virtualization Encapsulation (GENEVE) is designed to address perceived limitations of earlier

specifications, and it supports all the capabilities of VXLAN and GRE. GENEVE could eventually replace these earlier formats entirely.

GENEVE's stated goal is to define an encapsulation data format only. Unlike other formats, it does not include information or specification for the control plane. Tunnel encapsulation in GENEVE is similar to encapsulation with VXLAN, except the header, which contains fields for GENEVE packet recognition, the tunnel identifier, and other options. The carrier packet is transmitted to the destination as a UDP packet using either IPv4 or IPv6 to UDP port 6081, where the GENEVE headers are removed. The remaining original packet is then sent to the virtual network defined by the tunnel identifier.

The GENEVE specification offers recommendations on ways to achieve efficient operation by avoiding fragmentation and taking advantage of ECMP (equal-cost multi-path) and NIC hardware offload facilities. The specification also offers options on how to support differentiated services and explicit congestion notification. One key benefit over other encapsulation methods is GENEVE's flexible option format and use of IANA-designated Option Classes for extensibility. Transition to GENEVE will not be immediate. The other encapsulation methods have been in use for some time, and multiple methods can operate within the same system. However, GENEVE is being adopted as the default tunneling protocol for Open Virtual Network, which is being promoted as an implementation of Open vSwitch in future OpenStack versions.

Comparing VXLAN and GRE

VXLAN and GRE use network overlays to support private communications between instances on the internal network. An OpenStack Networking router is required to enable traffic to traverse outside of those tenant networks. A router is also required to connect provider tenant networks with external networks, including the Internet; the router provides the ability to connect to instances directly from an external network using floating IP addresses. Currently, VXLAN is the default internal network configuration for TripleO-orchestrated deployments.

All internal and provider network protocol implementations use the **provider:segmentation_id** parameter value to store the partitioned virtual network ID. For VLAN, this field holds the VID (tag). For VXLAN, this field is the VNI value. For GRE, this field is the VSID, also referred to as the tunnel ID (TID). Use the **openstack network show** command to view this parameter for a given network, regardless of the network protocol displayed in the **network_type** field.

Both VXLAN and GRE eliminate the VLAN protocol requirement that the physical network equipment create one unique VLAN and VLAN ID for each corresponding internal OpenStack VLAN. Since the internal packet is encapsulated, the tunneling network does not have to perform VLAN interpretation or matching on the physical hardware. The performance cost is that encapsulating and decapsulating packets at the endpoints uses additional CPU overhead for packet processing, which must be considered during network design.

GRE is commonly implemented as a unicast protocol, requiring vendor-specific hardware to properly provide multicasting for tunneling over a service provider Multi-protocol Label Switching (MPLS) network. GRE is normally a point-to-point tunnel, while VXLAN is a point-to-multipoint tunnel, operating as an emulated LAN.

VXLAN uses multicasting to provide service for broadcast multicast traffic. Multicast is used for control plane traffic, allowing internal network switches supporting a given VNI to learn MAC address owners. When one VM initiates communication to another VM considered to be on the same subnet, an ARP request broadcast is generated to learn the destination VM's MAC address. The source VTEP encapsulates this packet and addresses it using an IP multicast group,

previously assigned by the management layer when the VXLAN segment was created. The destination VTEP decapsulates the ARP request and passes it to the VM, which then creates an ARP response. When the response is similarly returned to the source VTEP and VM, both VTEPs and VMs retain the accurate MAC and IP addresses required for unicast UDP communication of data plane traffic without further multicasting.

Layer 3 devices use a calculated hash (called the "5-tuple") of the source IP address, destination IP address, transport layer protocol number (TCP or UDP), source port number, and destination port number to make route-cost forwarding decisions. Because GRE is not implemented with a destination port number, most Layer 3 devices calculate the same routing decision, leading to less-than-optimal utilization. In specific use cases, VXLAN achieves better aggregate throughput across equal-cost multi-path (ECMP) routing, due to VXLAN's use of UDP-based encapsulation. VXLAN tunneling performance is further improved by introducing hardware offloading, found in current high-capability NICs. The added overhead is offloaded to hardware, resulting in improved CPU utilization and higher throughput. A VXLAN-capable hardware list is available as a reference for this section.

Encapsulation always increases the size of a packet, since new headers are being added while retaining the original headers and content. It is recommended to increase the MTU size on all network devices from endpoint to endpoint to avoid unexpected fragmentation and subsequent network performance degradation.

Managing VXLAN and GRE

To configure the internal tenant network for a tunnel protocol, set parameters for the ML2 plugin on controllers and the Open vSwitch agent on controllers and compute nodes. The example here also sets the multicast address used for VXLAN path address discovery and other broadcast traffic.

```
[demo@controller0 ~]$ sudo cat /etc/neutron/plugins/ml2/ml2_conf.ini
...output omitted...
[ml2]
type_drivers = flat,vlan,gre,vxlan
tenant_network_types = vlan,gre,vxlan
mechanism_drivers = openvswitch
[ml2_type_gre]
tunnel_id_ranges =1:4094
[ml2_type_vxlan]
vni_ranges =1:4094
vxlan_group = 224.0.0.1

[demo@controller0 ~]$ sudo cat /etc/neutron/plugins/ml2/openvswitch_agent.ini
...output omitted...
[agent]
tunnel_types =vxlan,gre
vxlan_udp_port = 4789
[ovs]
integration_bridge = br-int
tunnel_bridge = br-tun
```

The OVS switch flow tables contain rules to direct traffic between the tenant network and the GRE tunnels created for it. The rules on the tunnel bridge br-tun manage traffic to and from the tunnel endpoint. Looking up the tenant network's interfaces discovers that br-int has tagged it as local VLAN ID 8.

```
[demo@controller0 ~]$ sudo ovs-vsctl show
```

```
...output omitted...
Bridge br-int
  Port "sg-effee73a-f9"
    tag: 8
  Port "qr-f6016b8a-43"
    tag: 8
  Port "tap4ab8dc6f-d8"
    tag: 8
```

In the following edited examples, the first rule recognizes a packet from local VLAN 8 coming from that network's router (ending in c5:43:43) and modifies the source MAC address to be the address for the distributed router (ac:da:0e). This causes returning packets to be directing to the appropriate hypervisor where the source VM is located.

The second rule recognizes an incoming packet from GRE tunnel 0x38 and tags it for local VLAN 8. The third rule is one of a set that handles the egress traffic. This recognizes a packet from local VLAN 8, loading the tunnel ID 0x38 and outputting the packet through port 2. Port 2 on the br-tun bridge is the GRE tunnel interface, as viewed with **ovs-ofctl show**.

```
[demo@controller0 ~]$ sudo ovs-ofctl dump-flows br-tun
...output omitted...
cookie=0xb57abcebaea6faa, duration=318.145s, table=1, n_packets=2,
n_bytes=220, idle_age=317, priority=1,dl_vlan=8,dl_src=fa:16:3e:c5:43:43
actions=mod_dl_src:fa:16:3f:ac:da:0e,resubmit(,2)
cookie=0xb57abcebaea6faa, duration=330.454s, table=3, n_packets=63, n_bytes=5780,
idle_age=52, priority=1,tun_id=0x38 actions=mod_vlan_vid:8,resubmit(,9)
cookie=0xb57abcebaea6faa, duration=307.135s, table=22, n_packets=1,
n_bytes=42, idle_age=258, priority=1,dl_vlan=8 actions=strip_vlan,load:0x38-
>NXM_NX_TUN_ID[],output:2

[demo@controller0 ~]$ sudo ovs-ofctl show br-tun
...output omitted...
1(patch-int): addr:da:52:14:34:3e:92
2(gre-ac180202): addr:ca:8b:f6:07:0c:91
LOCAL(br-tun): addr:aa:13:50:e1:d9:49
```

The OVS flow rules on the integration bridge br-int manage traffic to and from resources on the tenant GRE network. In the two example rules below, packets from the local VLAN 8 headed for the DHCP namespace (ending in 73:d2:09) or the SNAT table (ending in b5:b6:5b) are marked as having originated from the network's router interface (ending in c5:43:43). Output 43 is the TAP interface for the DHCP namespace. Output 45 is the interface to the SNAT routing namespace.

```
[demo@controller0 ~]$ sudo ovs-ofctl dump-flows br-int
...output omitted...
cookie=0x8e945d3936120645, duration=33933.163s, table=1, n_packets=0,
n_bytes=0, idle_age=33933, priority=4,dl_vlan=8,dl_dst=fa:16:3e:73:d2:09
actions=strip_vlan,mod_dl_src:fa:16:3e:c5:43:43,output:43
cookie=0x8e945d3936120645, duration=33929.233s, table=1, n_packets=257,
n_bytes=23048, idle_age=323, priority=4,dl_vlan=8,dl_dst=fa:16:3e:b5:b6:5b
actions=strip_vlan,mod_dl_src:fa:16:3e:c5:43:43,output:45

[demo@controller0 ~]$ sudo ovs-ofctl show br-int
...output omitted...
1(int-br-ex): addr:0a:ce:d5:a4:e3:f0
2(patch-tun): addr:d2:8d:70:9f:57:dc
4(qr-2c38dead-cf): addr:00:00:00:00:20:44
25(qg-607db4e2-08): addr:ce:1a:8c:62:03:77
43(tap4ab8dc6f-d8): addr:00:00:00:00:10:c4
44(qr-f6016b8a-43): addr:00:00:00:00:30:bd
```

```
45(sg-efffef73a-f9): addr:00:00:00:00:b0:8b  
46(qg-4974195e-f1): addr:00:00:00:00:40:e0  
LOCAL(br-int): addr:1e:7f:de:c3:3b:4f
```

References

- | RFC-7348: VXLAN
| <http://www.rfc-base.org/rfc-7348.html>
- | RFC-2784: GRE
| <https://www.ietf.org/rfc/rfc2784.txt>

Guided Exercise: Provisioning GRE Tenant Networks

In this exercise, you will provision a GRE network. You will attach a subnet, attach a router, and create an instance on the GRE tenant network. You will then capture traffic using tcpdump.

Outcomes

You should be able to:

- Create a GRE tenant network.
- View GRE packets.

Before you begin

Log in to **workstation** as **student**, using **student** as the password. Run the **lab provisioning-gre setup** command. This script verifies that the overcloud nodes are accessible, OpenStack services are running, and the correct projects, external networks and routers are already configured.

```
[student@workstation ~]$ lab provisioning-gre setup
```

Steps

1. Source the **architect1-finance** credentials. Create a GRE tenant network using the **openstack network create** command with the **--provider-network-type** option. Create a subnet for this GRE network. Attach the network to the **finance-router1** router.
 - 1.1. Create a GRE tenant network called **finance-gre-network1**.

Make note of the network's **id** and **provider:segmentation_id** (GRE tunnel ID--marked as **S** in the following output and following steps) fields, as you will need these values in later steps. In this example, the **segmentation_id** is displayed as the letter **S**.

```
[student@workstation ~]$ source ~/architect1-finance-rc
[student@workstation ~(architect1-finance)]$ openstack network create \
--provider-network-type gre \
finance-gre-network1
+-----+-----+
| Field | Value |
+-----+-----+
| admin_state_up | UP |
| id | 8265f409-3fc4-4a83-bd2f-29f25021da57 |
| name | finance-gre-network1 |
...output omitted...
| provider:network_type | gre |
| provider:physical_network | None |
| provider:segmentation_id | S |
...output omitted...
```

Note that the **physical_network** field is not set. The OpenStack Networking Service is already configured to place tenant networks that use tunneling protocols onto the

OVS bridge **br-tun**, overlaying the isolated tenant **vlan20** infrastructure, with packets traversing the underlying **br-trunk**'s **eth1** interface.

1.2. Create a subnet for the GRE network called **finance-gre-subnet**.

Make note of the subnet's **id** field, as you will need this value in later steps.

```
[student@workstation ~ (architect1-finance)]$ openstack subnet create \
--network finance-gre-network1 \
--subnet-range=192.168.4.0/24 \
--dns-nameserver=172.25.250.254 \
--dhcp finance-gre-subnet1
+-----+-----+
| Field      | Value
+-----+-----+
| allocation_pools | 192.168.4.2-192.168.4.254
| cidr       | 192.168.4.0/24
| ...output omitted...
| enable_dhcp | True
| gateway_ip  | 192.168.4.1
| id          | a6afb890-d2e3-4f06-8ac4-901279ca0bda
| ip_version   | 4
| name        | finance-gre-subnet1
| network_id   | 8265f409-3fc4-4a83-bd2f-29f25021da57
| ...output omitted...
```

1.3. Make note of the **finance-router1** ID, as you will need this value in later steps.

```
[student@workstation ~ (architect1-finance)]$ openstack router list \
-c ID -c Name
+-----+-----+
| ID           | Name
+-----+-----+
| 779631a4-1dea-4fb6-8db4-181ebd0d9248 | production-dvr-router1 |
| 9634e23f-e70f-4e32-8691-8af0167dda39 | research-router1 |
| a8ae4cbe-84c0-4127-b8de-898fd94b5379 | production-router1 |
| b07cc523-105d-418b-ac2c-cb620a2ada49 | finance-router1 |
| e8d3e5a5-b35e-4a36-a912-755ac9144284 | finance-dvr-router1 |
| e9e25f24-75b4-4439-86da-499ab2bd572a | research-dvr-router1 |
+-----+-----+
```

1.4. Attach the GRE network as an interface on the **finance-router1** router.

```
[student@workstation ~ (architect1-finance)]$ openstack router add \
subnet finance-router1 finance-gre-subnet1
```

2. Create a server instance named **finance-gre-server1**. Make note of the compute node on which the instance is hosted, and also the fixed IP address assigned, displayed here as **192.168.4.N**.

```
[student@workstation ~ (architect1-finance)]$ openstack server create \
--flavor default \
--key-name example-keypair \
--nic net-id=finance-gre-network1 \
--image rhel7 \
--wait finance-gre-server1
+-----+-----+
```

Field	Value
...output omitted...	
OS-EXT-SRV-ATTR:host	compute1.overcloud.example.com
OS-EXT-SRV-ATTR:hypervisor_hostname	compute1.overcloud.example.com
OS-EXT-SRV-ATTR:instance_name	instance-00000006
addresses	finance-gre-network1=192.168.4.N
name	finance-gre-server1
...output omitted...	

3. Open a second terminal window or tab for working on the **controller** node, leaving the first **workstation** terminal open and ready. On **controller0**, use the **ovs-ofctl** and **ovs-vsctl** commands to view OpenFlow rules for this GRE tenant network.
- 3.1. In the second terminal, log in as **root** to the controller node. Use the **ovs-ofctl** command to view the OpenFlow rules on the **br-tun** bridge.

Locate rules that process packets on this GRE network by filtering for the **segmentation_id**, which you noted earlier. The embedded **printf** command transposes the decimal tunnel ID (*S*) into the required hex display format.

```
[student@workstation ~]$ ssh root@controller0
[root@controller0 ~]# ovs-ofctl dump-flows br-tun | grep $(printf "0x%04x\n" $S)
cookie=0x9abdc97a5501843d, duration=2380.646s, table=3,
  n_packets=347, n_bytes=31530, idle_age=58, priority=1,
    tun_id=0x$ actions=mod_vlan_vid:$T,resubmit(,9)
cookie=0x9abdc97a5501843d, duration=1974.371s, table=20,
  n_packets=243, n_bytes=29528, idle_age=58, priority=2,
    dl_vlan=$T,dl_dst=fa:16:3e:c1:b8:17
      actions=strip_vlan,load:0x$->NXM_NX_TUN_ID[],output:3
cookie=0x9abdc97a5501843d, duration=1948.507s, table=20,
  n_packets=0, n_bytes=0, hard_timeout=300, idle_age=1948, hard_age=58,
  priority=1,
    vlan_tci=0x000c/0xffff,dl_dst=fa:16:3e:c1:b8:17
      actions=load:0->NXM_OF_VLAN_TCI[],load:0x$->NXM_NX_TUN_ID[],output:3
cookie=0x9abdc97a5501843d, duration=1974.372s, table=22,
  n_packets=0, n_bytes=0, idle_age=1974, priority=1,
    dl_vlan=$T actions=strip_vlan,load:0x$->NXM_NX_TUN_ID[],output:3
```

In the above example, the first rule handles incoming packets from the tunnel and adds the local VLAN ID (802.1Q tag) to the Ethernet header. The other rules handle outgoing packets, either destined for a server instance MAC address or recognized as coming from the local VLAN assigned to this tunnel ID. The outgoing rules variously strip the local VLAN ID, load the tunnel ID into memory, and send the packet out **br-tun** port **3**.

Make a note of the local VLAN ID (marked in the previous output as *T*) that is added to the packet because you will need it in a later step.

- 3.2. Use the **ovs-ofctl** command to show that **br-tun** port **3** is the GRE tunnel, which connects to the internal network.

```
[root@controller0 ~]# ovs-ofctl show br-tun
...output omitted...
 1(patch-int): addr:96:99:41:db:0d:d8
    config:     0
    state:      0
    speed: 0 Mbps now, 0 Mbps max
```

```
3(gre-ac18020c): addr:7a:c5:e4:6d:1e:24
    config:      0
    state:       0
    speed: 0 Mbps now, 0 Mbps max
...output omitted...
```

- 3.3. Use the **ovs-vsctl** command to show that local VLAN ID **T** is being used to tag local components for this GRE tenant network. Replace **T** with the tag value (**mod_vlan_vid**) discovered in the **ovs-ofctl** command.

```
[root@controller0 ~]# ovs-vsctl show | grep "tag: T$" -B1
    Port "qr-8bafd704-64"
        tag: T
    --
    Port "tap2698d7d1-e9"
        tag: T
```

- 3.4. Look inside this GRE subnet's DHCP namespace to list network interfaces. The namespace name is **dhcp-** plus the network ID noted earlier. The TAP device interface listed will match that listed in the previous command. The TAP device IP address is one of two noted earlier belonging to this VLAN network.

```
[root@controller0 ~]# ip -n \
qdhcp-8265f409-3fc4-4a83-bd2f-29f25021da57 addr show tap2698d7d1-e9
63: tap2698d7d1-e9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1454 qdisc noqueue
    state UNKNOWN qlen 1000
        link/ether fa:16:3e:e6:65:f4 brd ff:ff:ff:ff:ff:ff
        inet 192.168.4.2/24 brd 192.168.4.255 scope global tap2698d7d1-e9
            valid_lft forever preferred_lft forever
        inet6 fe80::f816:3eff:fee6:65f4/64 scope link
            valid_lft forever preferred_lft forever
```

- 3.5. From the DHCP namespace, ping the **finance-gre-server1** IP address. Leave this **ping** running so that it can be viewed with **tcpdump**.

```
[root@controller0 ~]# ip netns exec \
qdhcp-8265f409-3fc4-4a83-bd2f-29f25021da57 ping 192.168.4.N
PING 192.168.3.12 (192.168.4.N) 56(84) bytes of data.
64 bytes from 192.168.4.N: icmp_seq=1 ttl=64 time=1.33 ms
64 bytes from 192.168.4.N: icmp_seq=2 ttl=64 time=0.660 ms
...output omitted...
```

4. Open a third terminal window or tab for working on the **compute** node, leaving the first **workstation** terminal open and ready. The **ping** command is still running in the second **controller0** terminal. Log in to the compute node which is hosting the instance. Use **tcpdump** to view the GRE traffic.

- 4.1. In the third terminal, log in as **root** to the compute node. Use the **ovs-ofctl** command to list the OpenFlow rules in use on the **br-tun** bridge.

Locate rules that process packets on this GRE network by filtering for the **segmentation_id**, which you noted earlier. The embedded **printf** command transposes the decimal tunnel ID (**S**) into the required hex display format.

```
[student@workstation ~]$ ssh root@compute1
[student@compute1 ~]# ovs-ofctl dump-flows br-tun | grep $(printf "%02x\n" $T)
cookie=0x884018aab0354d9e, duration=7741.346s, table=3,
  n_packets=334, n_bytes=36422, idle_age=375, priority=1,
    tun_id=0x$T actions=mod_vlan_vid:$T,resubmit(,9)
cookie=0x884018aab0354d9e, duration=7740.632s, table=20,
  n_packets=401, n_bytes=34614, idle_age=375, priority=2,
    dl_vlan=$T,dl_dst=fa:16:3e:94:0c:77 actions=strip_vlan,
    load:0x$T->NXM_NX_TUN_ID[],output:2
cookie=0x884018aab0354d9e, duration=7740.630s, table=20,
  n_packets=25, n_bytes=2394, idle_age=534, priority=2,
    dl_vlan=$T,dl_dst=fa:16:3e:e6:65:f4
    actions=strip_vlan,load:0x$T->NXM_NX_TUN_ID[],output:2
cookie=0x884018aab0354d9e, duration=7740.635s, table=22,
  n_packets=12, n_bytes=1416, idle_age=557, priority=1,
    dl_vlan=$T actions=strip_vlan,load:0x$T->NXM_NX_TUN_ID[],output:2
```



Note

Note that the local VLAN tag for this GRE tenant network on **controller0** is not necessarily the same value as the local VLAN tag on **compute1**. Local VLAN tags are unique per OpenStack node and are never shared outside that node. Only the physical network GRE tunnel ID (**segmentation ID**) is shared for non-local traffic for GRE tenant networks.

- 4.2. Use the **ovs-ofctl** command to show that **br-tun** outgoing port **2** is the GRE tunnel for this tenant network.

```
[root@compute1 ~]# ovs-ofctl show br-tun
...output omitted...
1(patch-int): addr:32:3f:f6:b2:99:92
  config: 0
  state: 0
  speed: 0 Mbps now, 0 Mbps max
2(gre-ac180201): addr:8e:d2:98:06:aa:81
  config: 0
  state: 0
  speed: 0 Mbps now, 0 Mbps max
...output omitted...
```

- 4.3. Use the **ovs-vsctl** command to show that local VLAN ID **T** is being used to tag local components for this GRE tenant network. Replace **T** with the tag value (**mod_vlan_vid**) discovered in the **ovs-ofctl** command.

```
[root@compute1 ~]# ovs-vsctl show | grep "tag: T$" -B1
  Port "qvob2eb7cac-ed"
    tag: T
```

- 4.4. Use the **workstation** terminal for this step. Find the device referenced by **ovs-vsctl** as port ID (**qvob2eb7cac-ed**). The port ID shown is a short version of the full port ID, prefixed with the **qvo** veth pair code. Use only the portion after the prefix to grep for

the full port ID. In summary, this veth pair endpoint on local VLAN tag **T** connects to the Linux bridge in front of the server instance.

```
[student@workstation ~(architect1-finance)]$ openstack port list \
-c ID -c "Fixed IP Addresses" | grep b2eb7cac-ed
| b2eb7cac-edd8-4acf-a637-e6da8e013067 |
ip_address='192.168.4.N', subnet_id='a6afb890-d2e3-4f06-8ac4-901279ca0bda' |
```

- 4.5. Return to the **compute1** terminal. Use **tcpdump** to view the **ping** packets on this GRE tenant network. Interrupt the command as soon as it displays ICMP output. The packets are GRE protocol packets using the tunnel ID (segmentation ID) as the key. After viewing the packets, exit the compute node, then exit the extra third terminal window.

```
[root@compute1 ~]# tcpdump -ten -i vlan20 | grep ICMP
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth2, link-type EN10MB (Ethernet), capture size 262144 bytes
72:aa:a4:03:d1:ca > 92:c1:d5:05:62:6a, ethertype IPv4 (0x0800), length 140:
  172.24.2.1 > 172.24.2.12: GREv0, key=0xS, proto TEB (0x6558), length 106:
    fa:16:3e:e6:65:f4 > fa:16:3e:c1:b8:17, ethertype IPv4 (0x0800), length 98:
      192.168.4.2 > 192.168.4.N: ICMP echo request, id 43519, seq 24, length 64
    92:c1:d5:05:62:6a > 72:aa:a4:03:d1:ca, ethertype IPv4 (0x0800), length 140:
      172.24.2.12 > 172.24.2.1: GREv0, key=0xS, proto TEB (0x6558), length 106:
        fa:16:3e:c1:b8:17 > fa:16:3e:e6:65:f4, ethertype IPv4 (0x0800), length 98:
          192.168.4.N > 192.168.4.2: ICMP echo reply, id 43519, seq 24, length 64
^C10 packets captured
11 packets received by filter
0 packets dropped by kernel

[root@compute1 ~]# exit
logout
Connection to compute1 closed.
[student@workstation ~]$ exit
```

5. Return to the **controller** terminal. Interrupt the running **ping** command. Exit the compute node, then exit the extra second terminal window.

```
...output omitted...
64 bytes from 192.168.4.N: icmp_seq=506 ttl=64 time=0.709 ms
64 bytes from 192.168.4.N: icmp_seq=507 ttl=64 time=0.553 ms
64 bytes from 192.168.4.N: icmp_seq=508 ttl=64 time=0.942 ms
64 bytes from 192.168.4.N: icmp_seq=509 ttl=64 time=0.548 ms
Ctrl+C
[root@controller0 ~]# exit
logout
Connection to controller0 closed.
[student@workstation ~]$ exit
```

Cleanup

From **workstation**, run the **lab provisioning-gre cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab provisioning-gre cleanup
```

This concludes the guided exercise.

Provisioning Provider Networks with Subnet Pools

Objectives

After completing this section, students should be able to:

- Describe provider networks and compare them to self-service networks.
- Automate the assigning of IP addresses when creating subnets.

Provider Networks

Before OpenStack Networking introduced Distributed Virtual Routers (DVR), all tenant project traffic was forwarded to software-defined Layer 3 devices configured on controller or network nodes, which increased traffic latency and decreased network performance. Physical network infrastructures usually have better performance than software defined networking. Provider networks are designed to move routing and other layer 3 operations to the physical network infrastructure.

Choosing a self-service tenant network or a provider network affects who has permission to create networks, and how those networks route to other external resources. Provider networks are created by an OpenStack administrator and assigned to or shared by tenant projects. Self-service tenant networks are created by tenants for use by their instances and are not shared unless configured by an OpenStack administrator.

Provider networks are typically created on physical network VLANs in the data center, but provider networks can also be built using overlay protocols, such as GRE or VXLAN, which are then routed to the rest of the physical network, similar to how existing internal tenant networks are built. Tenant networks allow address scope overlap by using routers and NAT tables to isolate the internal fixed addresses from external floating IP addresses. Provider networks eliminate routers, floating IPs, and self-service fixed IP addresses of tenant networks. Instead, provider networks use addresses taken directly from the physical network infrastructure, and rely on the physical network's routers to be a project subnet's default gateway.

If you are using provider networks, your instances will get IP addresses from the provider network. All instances will have access to other IP addresses on that provider network. The instances may not even need floating IP addresses—they may already be routed to the outside world through the provider network. If they are not already routed to the outside world, you can provide them with floating IP addresses that use an external IP address.

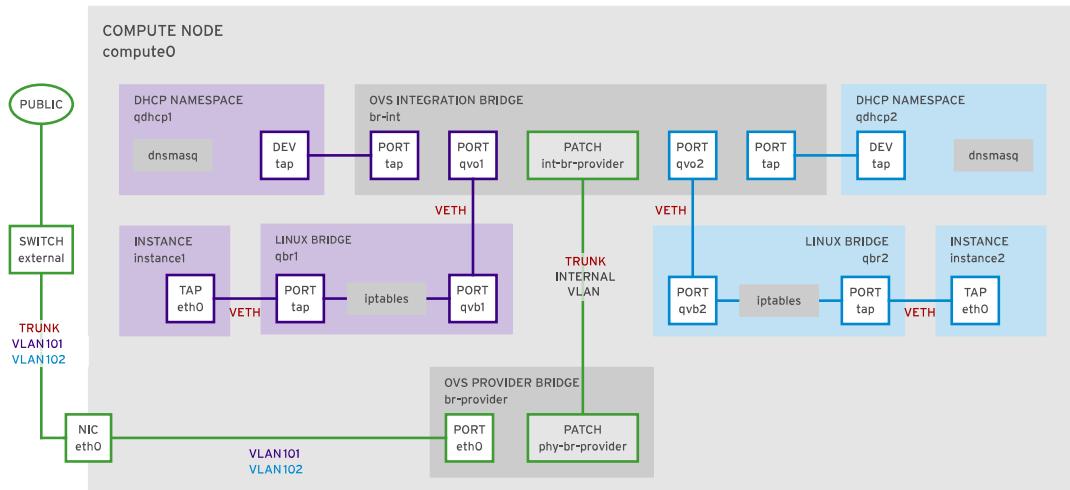


Figure 4.10: Provider network on compute node

Subnet Pools

Administrators creating new networks and subnets are required to manage IP addresses to ensure that overlapping addresses do not create conflicts across connected networks. OpenStack was designed for multitenancy by allowing projects to be privately managed, and by only allowing access to internal systems through use of NAT and a well-managed pool of floating IPs. An administrator still needs to manage the addresses within a project, because overlapping addresses in subnets could mean that conflicting subnets cannot connect to the same router, or that servers on those subnets might not be able to locate or communicate with each other.

Provider networks require even more stringent management of IP addresses. These networks are not routed to external networks through NAT; they exist directly on the external or internal networks, using IP address ranges already configured on the physical switches, routers, and systems. To allocate IP address from an existing network requires knowing what addresses are available, and tracking the ranges of addresses delegated, as new subnets are created.

OpenStack now offers subnet pools, which completely hand the management of addresses over to the Networking Service. An administrator would create a subnet pool to contain a sufficient CIDR-specified range of addresses, which are then distributed efficiently when project members create new subnets. Project members specify the pool and a CIDR mask indicating size of the address range they require, which can be safely limited by implementing subnet pool quotas.

Subnet pools track ranges already allocated, ensuring that addresses never overlap. If the subnet address pool was built with externally routable address scopes, then all projects using this subnet pool have addresses that are routable and unique. This is especially useful for IPv6, which uses unique and globally routable addresses without the floating IP technique.

Subnet pools can be shared, which allows them to be used by subnets from multiple projects. For IPv4, subnets created from the same subnet pool and connected to the same routers can communicate between those networks without using floating IP addresses or traversing external networks or routers. Project members can create subnet pools visible only to their project; shared subnet pools require admin privileges.

Class Internet Domain Routing (CIDR) Prefix Length

When subnet pools are created, a network address and corresponding CIDR mask are specified. When a subnet is created from that subnet pool, a smaller CIDR is specified to indicate a request for a smaller subset of addresses. To simplify subnet creation, a default CIDR value can be configured when the subnet pool is created, or modified later, to indicate the default size range of addresses to hand out automatically. If a project member does not include a CIDR value, or does not know how to interpret them, the default CIDR value will be used.



Note

Most experienced system administrators are familiar with subnetting calculations. The lecture and diagrams are meant as a visual point of reference for the discussion, showing a simple example of how the automated algorithms that process subnet pool requests might efficiently pull requested ranges from an available subnet pool range.

CIDR	/26	/27	/28	/29	/30
mask	255.255.255.192	255.255.255.224	255.255.255.240	255.255.255.248	255.255.255.252
subnets	4	8	16	32	64
hosts	62	30	14	6	2
.0				.0 (.1-.6)	.0 (.1-.2)
.4					.4 (.5-.6)
.8					.8 (.9-.10)
.12					.12 (.13-.14)
.16					.16 (.17-.18)
.20					.20 (.21-.22)
.24					.24 (.25-.26)
.28					.28 (.29-.30)
.32					.32 (.33-.34)
.36					.36 (.37-.38)
.40					.40 (.41-.42)
.44					.44 (.45-.46)
.48					.48 (.49-.50)
.52					.52 (.53-.54)
.56					.56 (.57-.58)
.60					.60 (.61-.62)

Figure 4.11: Sample subnet address ranges using variable-length subnet masks part 1

.64				.64 (.65-.66)
.68				.68 (.69-.70)
.72				.72 (.73-.74)
.76				.76 (.77-.78)
.80				.80 (.81-.82)
.84				.84 (.85-.86)
.88				.88 (.89-.90)
.92				.92 (.93-.94)
.96				.96 (.97-.98)
.100				.100 (.101-.102)
.104				.104 (.105-.106)
.108				.108 (.109-.110)
.112				.112 (.113-.114)
.116				.116 (.117-.118)
.120				.120 (.121-.122)
.124				.124 (.125-.126)

Figure 4.12: Sample subnet address ranges using variable-length subnet masks part 2

.128				
.132				
.136				
.140				
.144				
.148				
.152				
.156				
.160				
.164				
.168				
.172				
.176				
.180				
.184				
.188				
	.128 (.129-.190)	.128 (.129-.158)	.128 (.129-.134) .132 (.133-.134) .136 (.137-.138) .140 (.141-.142)	.128 (.129-.130) .132 (.133-.134) .136 (.137-.138) .140 (.141-.142)
			.144 (.145-.150) .148 (.149-.150) .152 (.153-.158) .156 (.157-.158)	.144 (.145-.146) .148 (.149-.150) .152 (.153-.154) .156 (.157-.158)
		.160 (.161-.190)	.160 (.161-.166) .168 (.169-.174) .176 (.177-.182)	.160 (.161-.162) .164 (.165-.166) .168 (.169-.170) .172 (.173-.174)
			.176 (.177-.190)	.176 (.177-.178) .180 (.181-.182) .184 (.185-.186) .188 (.189-.190)

Figure 4.13: Sample subnet address ranges using variable-length subnet masks part 3

.192				
.196				
.200				
.204				
.208				
.212				
.216				
.220				
.224				
.228				
.232				
.236				
.240				
.244				
.248				
.252				
	.192 (.193-.254)	.192 (.193-.222)	.192 (.193-.198) .200 (.201-.206)	.192 (.193-.194) .196 (.197-.198) .200 (.201-.202) .204 (.205-.206)
			.208 (.209-.222)	.208 (.209-.210) .212 (.213-.214) .216 (.217-.218) .220 (.221-.222)
		.224 (.225-.254)	.224 (.225-.230) .232 (.233-.238)	.224 (.225-.226) .228 (.229-.230) .232 (.233-.234) .236 (.237-.238)
			.240 (.241-.254)	.240 (.241-.242) .244 (.245-.246) .248 (.249-.250) .252 (.253-.254)

Figure 4.14: Sample subnet address ranges using variable-length subnet masks part 4

The previous set of figures represent an example of a subnet pool. The size of this pool, based on a CIDR value of 24 when the pool was created, would be described, in obsolete terminology, as a single "Class C" network containing 256 IPv4 addresses. The colored blocks illustrate how a subnet pool could be allocated in different sized groups of addresses using correct CIDR values, as seen along the top of the first figure. The table is too large to display on a single page, but is intended to be interpreted as a single table.

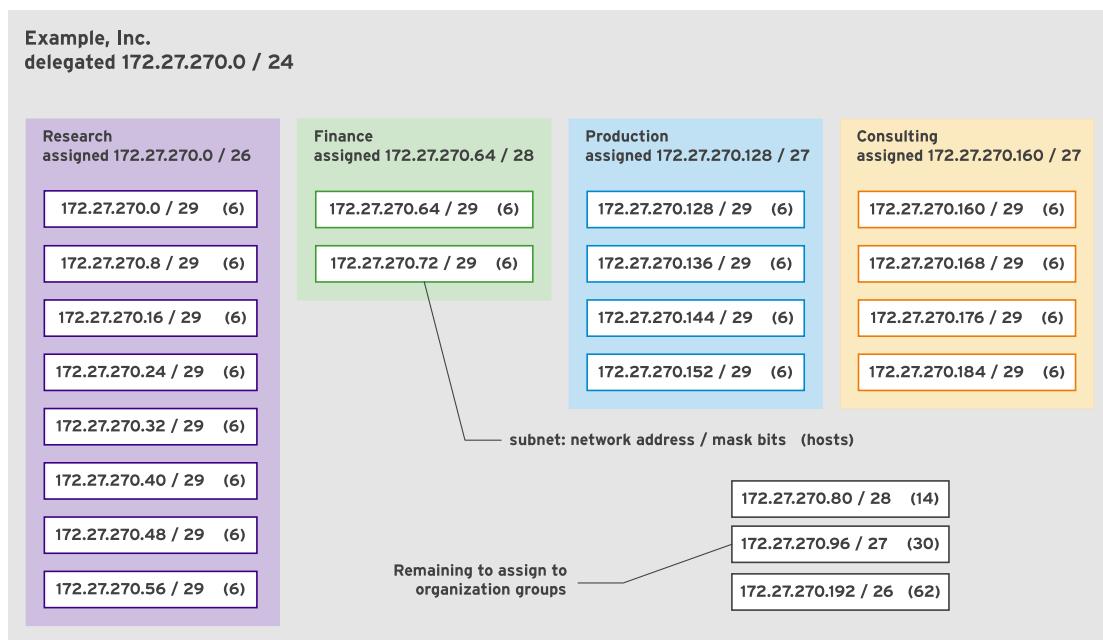


Figure 4.15: CIDR subnetting using VLSM route table entry aggregation

The above figure illustrates how project members at Example, Inc. might assign addresses to subnets, subnetting a 24 bit (old "Class C") IPv4 address range into non-overlapping subnets of 6 hosts each while using only a single route table entry per group. For this figure, the allocations shown at the top of each project box indicates the starting network address and CIDR that signifies a scope size. For example, Research was assigned the 172.27.270.0 network with a 26-bit mask, indicating one address range of 62 usable host addresses.

However, the project member is not required to delegate the whole scope to a single subnet. In this example, the assigned range was then delegated into smaller scopes for multiple subnets by specifying a smaller CIDR value. Creating subnets does not require specifying the starting network address, although projects can do so if the addresses are available to request. At a minimum, a CIDR value is specified, or the default is used, and the Networking Service determines the most efficient way to allocate the needed range size from the addresses available.

Managing Subnet Pools

The following steps outline the process for creating and using an IPv6-based subnet pool.

1. Create an IPv6 subnet pool with a 64-bit subnet mask.
2. Create an internal self-service VLAN network.
3. Create a subnet using the subnet pool and a non-default CIDR mask. Verify the IPv6 characteristics of the subnet.



References

Further information is available in the chapter on Connect an instance to the physical network in the *Red Hat OpenStack Platform 10 Networking Guide* at

| <https://access.redhat.com/documentation/en/red-hat-openstack-platform/>

Guided Exercise: Provisioning With Subnet Pools

In this exercise, you will provision self-service networks and manage a subnet pool.

Outcomes

You should be able to:

- Manage subnet pools.
- Interpret CIDR network address ranges.

Before you begin

Log in to **workstation** as **student**, using **student** as the password. Run the **lab provisioning-provider setup** command. This script verifies that the overcloud nodes are accessible and OpenStack services are running.

```
[student@workstation ~]$ lab provisioning-provider setup
```

Steps

1. Source the **architect1-finance** credentials. Using the **openstack subnet pool create** command, create a subnet pool with a 24-bit subnet mask. Use **finance-subnet-pool1** as the pool name.

```
[student@workstation ~]$ source ~/architect1-finance-rc
[student@workstation ~(architect1-finance)]$ openstack subnet pool create \
--pool-prefix 172.27.227.0/24 \
--default-prefix-length 26 \
finance-subnet-pool1
+-----+-----+
| Field | Value |
+-----+-----+
...output omitted...
| default_prefixlen | 26
| id | 66bfe44a-d44a-42a7-8d74-32f8071e79e2 |
| ip_version | 4
| is_default | False
| max_prefixlen | 32
| min_prefixlen | 8
| name | finance-subnet-pool1
| prefixes | 172.27.227.0/24
...output omitted...
```

2. Using the **openstack network create** command, create an internal self-service network. Use **finance-network3** as the network name.

```
[student@workstation ~(architect1-finance)]$ openstack network create \
finance-network3
+-----+-----+
| Field | Value |
+-----+-----+
...output omitted...
| id | 6e1a63d4-5133-4a38-9924-e90458172eb9 |
```

ipv4_address_scope	None
ipv6_address_scope	None
mtu	1446
name	finance-network3
provider:network_type	vxlan
router:external	Internal
...output omitted...	

3. Create a subnet in **finance-network3** using **finance-subnet-pool1** to specify the IP addresses. Use defaults; do not specify any subnet pool parameters other than the pool name. Use **finance-pool-subnet3** as the subnet name. View the CIDR mask and note that it controls the number of addresses in the allocation pool.

[student@workstation ~ (architect1-finance)]\$ openstack subnet create \
--subnet-pool finance-subnet-pool1 \
--network finance-network3 \
finance-pool-subnet3
+-----+-----+-----+
Field Value
+-----+-----+-----+
allocation_pools 172.27.227.2-172.27.227.62
cidr 172.27.227.0/26
created_at 2018-01-15T07:25:31Z
enable_dhcp True
gateway_ip 172.27.227.1
id 0880cedd-c555-41b9-8095-ac191215e63c
ip_version 4
name finance-pool-subnet3
...output omitted...

4. Create a server instance named **finance-server3** on this network. Do not specify the **--wait** option so you can continue while it builds.

[student@workstation ~ (architect1-finance)]\$ openstack server create \
--flavor default \
--key-name example-keypair \
--nic net-id=finance-network3 \
--image rhel7 \
finance-server3
+-----+-----+-----+
Field Value
+-----+-----+-----+
...output omitted...
name finance-server3
status BUILD
...output omitted...

5. Using the **openstack network create** command, create a second internal self-service network. Use **finance-network4** as the network name.

[student@workstation ~ (architect1-finance)]\$ openstack network create \
finance-network4
+-----+-----+-----+
Field Value
+-----+-----+-----+
...output omitted...
id 3cbdadc5-69b5-43b7-914b-69e7f21315c3
ipv4_address_scope None

ipv6_address_scope	None
mtu	1446
name	finance-network4
provider:network_type	vxlan
router:external	Internal
...output omitted...	

6. Create a subnet in **finance-network4** using **finance-subnet-pool1** to specify the IP addresses. This time, specify a prefix length of 28 to request a smaller range than is provided by the default CIDR of 26. Use **finance-pool-subnet4** as the subnet name. View the CIDR mask and note that it controls the number of addresses in the allocation pool.

[student@workstation ~ (architect1-finance)]\$ openstack subnet create \
--subnet-pool finance-subnet-pool1 \
--prefix-length 28 \
--network finance-network4 \
finance-pool-subnet4
+-----+-----+
Field Value
+-----+-----+
allocation_pools 172.27.227.66-172.27.227.78
cidr 172.27.227.64/28
enable_dhcp True
gateway_ip 172.27.227.65
id 60c548cd-d022-4b84-8169-3cceaa2f1186
ip_version 4
name finance-pool-subnet4
...output omitted...

7. Create a server instance named **finance-server4** on **finance-network4**. This time, specify **--wait**, so that you know when it is done building.

[student@workstation ~ (architect1-finance)]\$ openstack server create \
--flavor default \
--key-name example-keypair \
--nic net-id=finance-network4 \
--image rhel7 \
--wait finance-server4
+-----+-----+
Field value
+-----+-----+
...output omitted...
OS-EXT-SRV-ATTR:instance_name instance-0000000a
OS-EXT-STS:power_state Running
OS-EXT-STS:task_state None
OS-EXT-STS:vm_state active
addresses finance-network4=172.27.227.75
name finance-server4
...output omitted...

8. Use **openstack subnet show** to view **finance-pool-subnet3** allocation information. Verify that the CIDR value matches the allocation.

[student@workstation ~ (architect1-finance)]\$ openstack subnet show \
-c allocation_pools -c cidr finance-pool-subnet3
+-----+-----+
Field Value
+-----+-----+

allocation_pools 172.27.227.2-172.27.227.62
cidr 172.27.227.0/26
+-----+-----+

9. Use **openstack server show** to view **finance-server3** address information. Verify that the address is from the allocated range for **finance-pool-subnet3**.

```
[student@workstation ~(architect1-finance)]$ openstack server show \
-c addresses finance-server3
+-----+-----+
| Field | Value |
+-----+-----+
| addresses | finance-network3=172.27.227.9 |
+-----+-----+
```

10. Use **openstack subnet show** to view **finance-pool-subnet4** allocation information. Verify that the CIDR value matches the allocation. Notice the smaller range of addresses, which begins where the **finance-pool-subnet3** range ended.

```
[student@workstation ~(architect1-finance)]$ openstack subnet show \
-c allocation_pools -c cidr finance-pool-subnet4
+-----+-----+
| Field | Value |
+-----+-----+
| allocation_pools | 172.27.227.66-172.27.227.78 |
| cidr | 172.27.227.64/28 |
+-----+-----+
```

11. Use **openstack server show** to view **finance-server4** address information. Verify that the address is from the allocated range for **finance-pool-subnet4**.

```
[student@workstation ~(architect1-finance)]$ openstack server show \
-c addresses finance-server4
+-----+-----+
| Field | Value |
+-----+-----+
| addresses | finance-network4=172.27.227.75 |
+-----+-----+
```

Cleanup

From **workstation**, run the **lab provisioning-provider cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab provisioning-provider cleanup
```

This concludes the guided exercise.

Lab: Provisioning OpenStack Networks

In this lab, you will create a subnet pool and share it with two project networks. You will create the networks, subnets, a shared router, and two server instances. By pinging from one server to the other, you will verify that servers using the same subnet pool can communicate with each other without using an external network.

Outcomes

You should be able to:

- Provision a VXLAN tenant network.
- Manage subnet pools.

Before you begin

Log in to **workstation** as **student** using **student** as the password.

Run the **lab provisioning-networks setup** command. The script verifies that the **provider-datacentre** external network, correct projects, and users exist.

```
[student@workstation ~]$ lab provisioning-networks setup
```

Steps

1. As **architect1** in the **production** project, create a subnet pool using the details in the following table.

Resource	Attributes
subnet pool	<ul style="list-style-type: none"> • name production-subnet-pool1 • network range 172.27.227.0/24 • default prefix 28

2. As **architect1** in the **production** project, create a network, subnet, router, and a server instance using the resource attributes in the following table.

Resource	Attributes
network	<ul style="list-style-type: none"> • name production-network3 • network type GRE
subnet	<ul style="list-style-type: none"> • name production-pool-subnet3 • subnet pool production-subnet-pool1 • network production-network3
router	<ul style="list-style-type: none"> • name production-shared-router1 • gateway provider-datacentre • interface production-pool-subnet3
server	<ul style="list-style-type: none"> • name production-server3 • flavor default • network production-network3 • image rhel7 • key-name example-keypair

3. As **architect1** in the **research** project, create a network, a subnet, and a server instance using the resource attributes in the following table.

Resource	Attributes
network	<ul style="list-style-type: none"> name research-network3 network type VXLAN
subnet	<ul style="list-style-type: none"> name research-pool-subnet3 subnet pool production-subnet-pool1 router production-shared-router1
server	<ul style="list-style-type: none"> name research-server3 flavor default network research-network3, image rhel7 key-name example-keypair

4. As **operator1** in the **production** project, attach a floating IP to **production-server3**. Use **ssh** to connect to the **production-server3**, then ping from this instance to the internal IP address of **research-server3**.

Evaluation

From **workstation**, run the **lab provisioning-networks grade** command to confirm the success of this exercise. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab provisioning-networks grade
```

Cleanup

From **workstation**, run the **lab provisioning-networks cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab provisioning-networks cleanup
```

This concludes the lab.

Solution

In this lab, you will create a subnet pool and share it with two project networks. You will create the networks, subnets, a shared router, and two server instances. By pinging from one server to the other, you will verify that servers using the same subnet pool can communicate with each other without using an external network.

Outcomes

You should be able to:

- Provision a VXLAN tenant network.
- Manage subnet pools.

Before you begin

Log in to **workstation** as **student** using **student** as the password.

Run the **lab provisioning-networks setup** command. The script verifies that the **provider-datacentre** external network, correct projects, and users exist.

```
[student@workstation ~]$ lab provisioning-networks setup
```

Steps

1. As **architect1** in the **production** project, create a subnet pool using the details in the following table.

Resource	Attributes
subnet pool	<ul style="list-style-type: none"> • name production-subnet-pool1 • network range 172.27.227.0/24 • default prefix 28

- 1.1. Source the **architect1-production-rc** credentials. Using the **openstack subnet pool create** command, create a subnet pool with a 24-bit subnet mask and a 28-bit default prefix. Use **production-subnet-pool1** as the pool name.

```
[student@workstation ~]$ source ~/architect1-production-rc
[student@workstation ~(architect1-production)]$ openstack subnet pool create \
--pool-prefix 172.27.227.0/24 \
--default-prefix-length 28 \
production-subnet-pool1
+-----+-----+
| Field      | Value           |
+-----+-----+
...output omitted...
| default_prefixlen | 28
| id             | 93a356d5-05fb-4c18-9511-c82326a26c6f |
| ip_version     | 4
| is_default     | False
| max_prefixlen  | 32
| min_prefixlen  | 8
| name           | production-subnet-pool1
| prefixes        | 172.27.227.0/24
...output omitted...
```

2. As **architect1** in the **production** project, create a network, subnet, router, and a server instance using the resource attributes in the following table.

Resource	Attributes
network	<ul style="list-style-type: none"> name production-network3 network type GRE
subnet	<ul style="list-style-type: none"> name production-pool-subnet3 subnet pool production-subnet-pool1 network production-network3
router	<ul style="list-style-type: none"> name production-shared-router1 gateway provider-datacentre interface production-pool-subnet3
server	<ul style="list-style-type: none"> name production-server3 flavor default network production-network3 image rhel7 key-name example-keypair

- 2.1. Using the **openstack network create** command, create an internal GRE self-service network. Use **finance-network3** as the network name.

```
[student@workstation ~(architect1-production)]$ openstack network create \
--provider-network-type gre production-network3
+-----+-----+
| Field | Value |
+-----+-----+
...output omitted...
| id | 25ae7c56-5ad9-49a4-9082-0b0e277f019f |
| ipv4_address_scope | None |
| ipv6_address_scope | None |
| mtu | 1446 |
| name | production-network3 |
| provider:network_type | gre |
| router:external | Internal |
...output omitted...
```

- 2.2. Create a subnet in **production-network3** using **production-subnet-pool1** to provide the subnet range instead of specifying it on the command line. Use **production-pool-subnet3** as the subnet name.

```
[student@workstation ~(architect1-production)]$ openstack subnet create \
--subnet-pool production-subnet-pool1 \
--network production-network3 \
production-pool-subnet3
+-----+-----+
| Field | Value |
+-----+-----+
| allocation_pools | 172.27.227.2-172.27.227.14 |
| cidr | 172.27.227.0/28 |
| created_at | 2018-01-17T02:26:09Z |
| enable_dhcp | True |
| gateway_ip | 172.27.227.1 |
| id | f128e8da-ab9a-4ab1-b48c-b5204c5f09ca |
| ip_version | 4 |
```

name	production-pool-subnet3	
...output omitted...		

2.3. Create a shared router named **production-shared-router1**.

[student@workstation ~ (architect1-production)]\$ openstack router create \		
production-shared-router1		
+-----+-----+		
Field	Value	
+-----+-----+		
...output omitted...		
id	a8065722-3dd5-467b-8f44-73df534f9f7c	
name	production-shared-router1	
...output omitted...		

2.4. Connect the router to the external network **provider-datacentre**.

[student@workstation ~ (architect1-production)]\$ neutron router-gateway-set \
production-shared-router1 provider-datacentre
Set gateway for router production-shared-router1

2.5. Connect the router to the internal network **production-pool-subnet3**.

[student@workstation ~ (architect1-production)]\$ openstack router add subnet \
production-shared-router1 production-pool-subnet3

2.6. Create a server instance named **production-server3** on this network. Do not specify the **--wait** option so that you can continue working while it builds.

[student@workstation ~ (architect1-production)]\$ openstack server create \		
--flavor default \		
--key-name example-keypair \		
--nic net-id=production-network3 \		
--image rhe17 \		
production-server3		
+-----+-----+-----+		
Field	Value	
+-----+-----+-----+		
...output omitted...		
name	production-server3	
status	BUILD	
...output omitted...		

3. As **architect1** in the **research** project, create a network, a subnet, and a server instance using the resource attributes in the following table.

Resource	Attributes
network	<ul style="list-style-type: none"> name research-network3 network type VXLAN
subnet	<ul style="list-style-type: none"> name research-pool-subnet3 subnet pool production-subnet-pool1 router production-shared-router1
server	<ul style="list-style-type: none"> name research-server3

Resource	Attributes
	<ul style="list-style-type: none"> flavor default network research-network3, image rhel7 key-name example-keypair

- 3.1. Open a second terminal window so you can work in both projects simultaneously. Source the **architect1-research-rc** credentials. Using the **openstack network create** command, create an internal VXLAN self-service network. Use **research-network3** as the network name.

```
[student@workstation ~]$ source ~/architect1-research-rc
[student@workstation ~(architect1-research)]$ openstack network create \
--provider-network-type vxlan research-network3
+-----+-----+
| Field | Value |
+-----+-----+
...output omitted...
| id | 697d6569-3c41-4c51-9e6f-4222d23f3865 |
| ipv4_address_scope | None |
| ipv6_address_scope | None |
| mtu | 1446 |
| name | research-network3 |
| provider:network_type | vxlan |
| router:external | Internal |
...output omitted...
```

- 3.2. Create a subnet in **research-network3** using **production-subnet-pool1** to provide the subnet range instead of specifying it on the command line. Use **research-pool-subnet3** as the subnet name.

```
[student@workstation ~(architect1-research)]$ openstack subnet create \
--subnet-pool production-subnet-pool1 \
--network research-network3 \
research-pool-subnet3
+-----+-----+
| Field | Value |
+-----+-----+
| allocation_pools | 172.27.227.18-172.27.227.30 |
| cidr | 172.27.227.16/28 |
| created_at | 2018-01-17T02:26:09Z |
| enable_dhcp | True |
| gateway_ip | 172.27.227.17 |
| id | db999ed6-a31c-4ff6-8908-8198b9c0dacc |
| ip_version | 4 |
| name | research-pool-subnet3 |
...output omitted...
```

- 3.3. Add the **research-pool-subnet3** subnet to the **production-shared-router1** router.

```
[student@workstation ~(architect1-research)]$ openstack router add subnet \
production-shared-router1 research-pool-subnet3
```

- 3.4. Create a server instance named **research-server3** on the **research-network3** network. This time, wait for it to finish building. Note the IP address and the Project ID.

```
[student@workstation ~(architect1-research)]$ openstack server create \
--flavor default \
--key-name example-keypair \
--nic net-id=research-network3 \
--image rhel7 \
--wait research-server3
+-----+
| Field | Value |
+-----+
...output omitted...
| addresses | research-network3=172.27.227.N |
| name | research-server3 |
| project_id | f29dea47ef024f559fe6b556045312e4 |
...output omitted...
```

4. As **operator1** in the **production** project, attach a floating IP to **production-server3**. Use **ssh** to connect to the **production-server3**, then ping from this instance to the internal IP address of **research-server3**.
- 4.1. Return to the first terminal window, where you were working in the **production** project. Source the **operator1-production-rc** credentials. Using the **openstack floating ip list** command, locate an available floating IP for this project.

```
[student@workstation ~(architect1-production)]$ source ~/operator1-production-rc
[student@workstation ~(operator1-production)]$ openstack floating ip list \
-c "Floating IP Address" -c Port
+-----+
| Floating IP Address | Port |
+-----+
| 172.25.250.165 | None |
| 172.25.250.161 | None |
| 172.25.250.163 | None |
| 172.25.250.162 | None |
| 172.25.250.164 | None |
+-----+
```

- 4.2. Attach a floating IP to **production-server3**.

```
[student@workstation ~(operator1-production)]$ openstack server add \
floating ip production-server3 172.25.250.P
```

- 4.3. Ping the internal IP address of the **research-server3** instance from the **production-server3** instance.

```
[student@workstation ~(operator1-production)]$ ssh cloud-user@172.25.250.P \
ping -c 4 172.27.227.N
PING 172.27.227.N (172.27.227.N) 56(84) bytes of data.
64 bytes from 172.27.227.N: icmp_seq=1 ttl=63 time=1.45 ms
64 bytes from 172.27.227.N: icmp_seq=2 ttl=63 time=0.800 ms
64 bytes from 172.27.227.N: icmp_seq=3 ttl=63 time=0.903 ms
64 bytes from 172.27.227.N: icmp_seq=4 ttl=63 time=0.831 ms
```

```
--- 172.27.227.29 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3003ms
rtt min/avg/max/mdev = 0.800/0.997/1.455/0.267 ms
```

The **research-server3** instance is not utilizing a floating IP address, meaning that the packets are not traversing an external network. These instances from different projects are communicating across a shared router. Their IP addresses come from a single shared subnet pool, so they will not conflict.

Evaluation

From **workstation**, run the **lab provisioning-networks grade** command to confirm the success of this exercise. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab provisioning-networks grade
```

Cleanup

From **workstation**, run the **lab provisioning-networks cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab provisioning-networks cleanup
```

This concludes the lab.

Summary

In this chapter, you learned:

- Network Namespaces facilitate multi-tenancy, allowing OpenStack projects and network resources to operate in isolated network TCP/IP stacks. With namespaces, each project can utilize routing rules, IP addressing schemes, and VM configuration without creating conflicts with other projects. OpenStack has namespaces to provide isolation for many types of Networking Service resource objects: routers, DHCP servers, and NAT table processing daemons.
- VLAN networks implement the Ethernet 802.1Q tagging standard to segregate traffic shared on a single segment. OpenStack implements three distinct VLAN networks: external for public networks, internal for the backbone between OpenStack nodes, and the local VLAN managed by the **br-int** integration bridge, which is unique per OpenStack node.
- Many OpenStack infrastructures implement shared physical NICs, utilizing isolated VLANs for the internal networks. When the internal tenant network is configured to use VLAN, this results in a stacked VLAN configuration known as **QinQ**, which utilizes a double set of Ethernet 802.1Q tags.
- To expand beyond VLAN tenant network limitations, VXLAN and GRE are tunneling, or carrier, protocols designed to carry a layer 2 packet across an underlying layer 3 network. Both protocols are variations on packet encapsulation.
- Provider networks increase network performance by eliminating the need for OpenStack Networking Service routers and placing project deployments directly on networks and subnets built on existing external or internal networking infrastructure.
- A subnet pool is a feature commonly used with provider networks. Pools manage a large range of IP addresses, tracking small allocations to newly-created subnets such that no address conflicts or overlaps are created, while performing address assignments automatically.



CHAPTER 5

IMPLEMENTING DISTRIBUTED VIRTUAL ROUTING

Overview	
Goal	Implement Distributed Virtual Routing (DVR) to provide scaling and performance.
Objectives	<ul style="list-style-type: none">• Describe Distributed Virtual Routing (DVR) architecture, benefits, and use cases.• Manage Distributed Virtual Routing (DVR) routers.
Sections	<ul style="list-style-type: none">• Describing DVR Architecture (and Quiz)• Managing DVR Routers (and Guided Exercise)
Quiz	Implementing Distributed Virtual Routing

Describing DVR Architecture

Objectives

After completing this section, students should be able to:

- Describe the overcloud configuration for DVR.
- Describe DVR architecture.
- Compare east-west traffic to north-south traffic.

Distributed Virtual Routing (DVR)

Neutron Distributed Virtual Routing is a routing model that places L3 routers directly onto compute nodes, enabling both inter-project instance traffic and external traffic to be directly routed without traversing Networking Service (controller) nodes. DVR implements a floating IP namespace on each compute node where VMs are running, providing source network address translation (SNAT) behavior for private VMs. Introduced as a technology preview feature in Red Hat OpenStack Platform 6, DVR is fully supported with Red Hat OpenStack Platform 10. DVR configuration is an optional feature; typical installations default to legacy, centralized routing.

Routing Traffic Direction Flows

Routing services in OpenStack can be categorized into distinctions of distributed (DVR) or centralized (legacy), and either traffic between VMs within an installation (east-west) or traffic between a VM and systems external to the OpenStack installation (north-south).

- East-west non-DVR (legacy) traffic is routed between different subnets, IPv4 or IPv6, in either the same project or between subnets of different projects, requiring legacy routers. Such traffic remains within OpenStack nodes and does not traverse external networks.
- East-west DVR traffic is routed directly from a compute node in a distributed design, which bypasses controller nodes.
- North-south non-DVR (legacy) traffic using floating IPs is a one-to-one association between the floating IP and an instance's fixed address, implemented by IPv4 NAT tables on a legacy Networking Service router. Instances communicate with external resources using floating IPs reserved from the provider network. Instances configured with IPv6 use routable Global Unicast Addresses (GUAs), precluding the need for address overlap management, and are routed without needing NAT.
- North-south DVR traffic with floating IPs is distributed and routed directly from the compute nodes, requiring external provider network connectivity on every compute node.
- North-south non-DVR (legacy) traffic without floating IPs is handled by a port address translation (PAT) service, enabling instances to initiate bidirectional traffic to external systems, but to disallow external sites to initiate connections directly to internal instances. Externally initiated traffic must traverse Networking Service (controller) nodes for proper firewall filtering and route addressing to locate instances on their compute host. SNAT applies to IPv4 traffic only.
- North-south DVR traffic without floating IPs (DVR) is not distributed, requiring a dedicated Neutron Service (controller) node.

Minimum DVR Requirements

DVR works in a typical, minimum OpenStack environment. The controller node must contain an interface used for network management. The controller and compute nodes must contain management, instance tunnels, and external connectivity interfaces along with an Open vSwitch bridge port on the external interface.

Controller Node Components

- Open vSwitch Agent

Manages virtual switches, connectivity among them, and interaction over virtual ports with other networking components such as namespaces, Linux bridges, and physical interfaces.

- DHCP Agent

Manages DHCP network namespaces, which provide IP allocations for project instances. The DHCP agent starts dnsmasq processes to manage IP address allocation.

- L3 Agent

Manages the qrouter and SNAT namespaces.

The qrouter namespaces route north-south and east-west network traffic, performs destination network address translation (DNAT) and source network address translation (SNAT), and routes metadata traffic between instances and the metadata agent.

SNAT namespaces perform SNAT for north-south network traffic for instances with both a fixed IP address and project networks on distributed routers.

- Metadata Agent

Processes metadata operations for instances using project networks on legacy routers.

Compute Node Components

- Open vSwitch Agent

Manages virtual switches, connectivity among them, and interaction over virtual ports with other network components such as namespaces, Linux bridges, and physical interfaces.

- DHCP Agent

Manages DHCP network namespaces, which provide IP allocations for project instances. The DHCP agent starts dnsmasq processes to manage IP address allocation. The default is to run the DHCP agent on the controller node, but you can make a simple configuration change to run DHCP agents on the compute nodes.

- L3 Agent

The FIP namespaces route north-south network traffic and perform DNAT/SNAT for instances that have a floating IP and that use project networks on distributed routers.

The qrouter namespaces route east-west traffic for instances that have either a fixed or a floating IP address, and that use project networks on distributed routers.

- Meta Data Agent

Processes metadata operations for instances that use project networks on distributed routers.

- Linux Bridge

The Neutron service uses a Linux bridge to manage security groups for instances.

Observing Packet Flows in DVR Scenarios

This section describes OpenStack network packet flow when using Distributed Virtual Routers. Most traffic no longer requires forwarding by legacy routers on a controller or networking node to communicate with external systems or other project VMs. However, some traffic still traverses a controller node to obtain NAT processing. Four scenarios are described:

- East-west flow between VMs in a single subnet on different hosts.
- East-west flow between VMs in different subnets and on different hosts.
- North-south flow for a VM when a Floating IP is necessary.
- North-south flow for a VM when a Floating IP is not necessary.

East-west Packet Flow in One Subnet

Compute nodes route east-west network traffic within project networks on the same router, bypassing controller node processing, using the instances' fixed IP addresses. Floating IP addresses are not needed, even if assigned, because packets will not reach an external network. The narrative is in two parts:

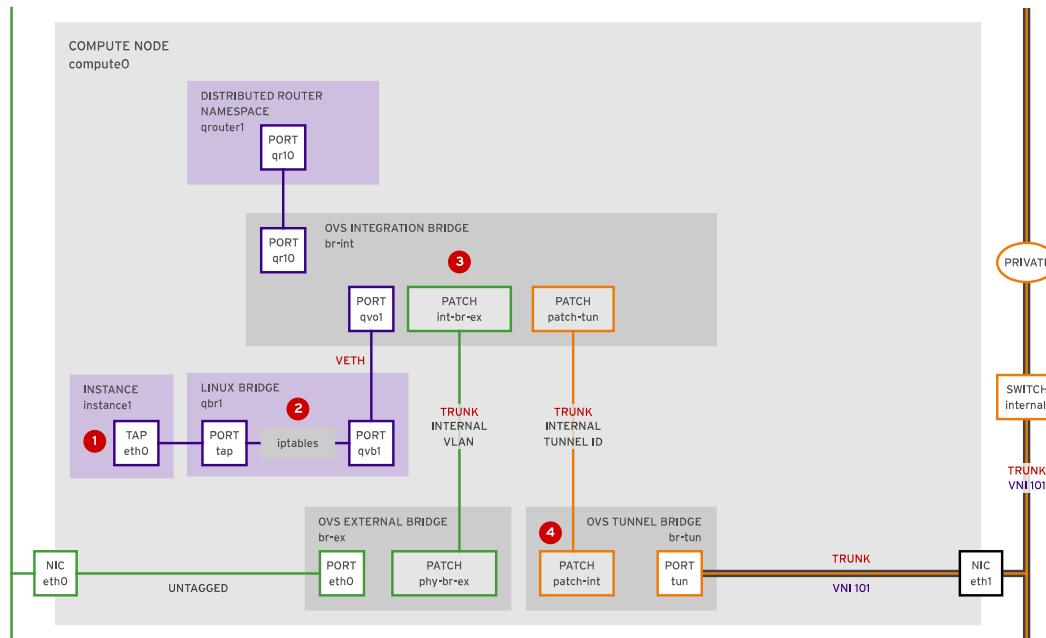
- Outbound from the source VM on any compute host.
- Inbound to the destination VM on another compute host.

Resource Objects Referenced In This Narrative

Resource	Description
network0	Project tenant network with subnet 192.168.0.0/24 and gateway 192.168.0.1
compute0	Compute Service node hosting server instance1 with IP 192.168.0.4
compute1	Compute Service node hosting server instance2 with IP 192.168.0.5
instance1	Server instance hosted on compute0 , using project network network0
instance2	Server instance hosted on compute1 , using project network network0

Outbound From The Source VM On Any Compute Host

This first narrative describes packets originating from an instance residing on one compute host. Packets will travel to a destination instance in the same project and subnet, but on another compute host. This initial narrative follows the packet starting at the instance until it leaves this compute host. Use the following figure to reference the numbered callouts.



East-west flow in one subnet across two hosts - outbound

- ➊ The instance1 TAP interface `tapf8bf6552-04` forwards the packet containing the destination MAC address `fa:16:3e:17:bf:c8`, to the Linux bridge `qbrf8bf6552-04`

```
[heat-admin@compute0 ~]$ brctl show
bridge name     bridge id      STP enabled interfaces
qbrf8bf6552-04  8000.7a0086c25d19    no  qvbf8bf6552-04
                                         tapf8bf6552-04
```

- ➋ Security group rules on the Linux bridge `qbrf8bf6552-04` process state tracking for the packet. The security group rules can be viewed with the `iptables` command and the `grep` option using the first three digits of the TAP interface. The letter `o` stands for *option* and is required before the first three digits of the TAP interface when using `grep`.

```
[heat-admin@compute0 ~]$ sudo iptables -S | grep of8
-N neutron-openvswi-o1bc37093-9
-A neutron-openvswi-INPUT -m physdev --physdev-in tap1bc37093-9b --physdev-is-bridged -m comment --comment "Direct incoming traffic from VM to the security group chain." -j neutron-openvswi-o1bc37093-9
-A neutron-openvswi-o1bc37093-9 -s 0.0.0.0/32 -d 255.255.255.255/32 -p udp -m udp --sport 68 --dport 67 -m comment --comment "Allow DHCP client traffic." -j RETURN
-A neutron-openvswi-o1bc37093-9 -j neutron-openvswi-s1bc37093-9
-A neutron-openvswi-o1bc37093-9 -p udp -m udp --sport 68 --dport 67 -m comment --comment "Allow DHCP client traffic." -j RETURN
-A neutron-openvswi-o1bc37093-9 -p udp -m udp --sport 67 -m udp --dport 68 -m comment --comment "Prevent DHCP Spoofing by VM." -j DROP
-A neutron-openvswi-o1bc37093-9 -m state --state RELATED,ESTABLISHED -m comment --comment "Direct packets associated with a known session to the RETURN chain." -j RETURN
-A neutron-openvswi-o1bc37093-9 -j RETURN
-A neutron-openvswi-o1bc37093-9 -m state --state INVALID -m comment --comment "Drop packets that appear related to an existing connection (e.g. TCP ACK/FIN) but do not have an entry in conntrack." -j DROP
-A neutron-openvswi-o1bc37093-9 -m comment --comment "Send unmatched traffic to the fallback chain." -j neutron-openvswi-sg-fallback
```

```
-A neutron-openvswi-sg-chain -m physdev --physdev-in tap1bc37093-9b --physdev-is-bridged -m comment --comment "Jump to the VM specific chain." -j neutron-openvswi-01bc37093-9
```

The Linux bridge `qbrf8bf6552-04` forwards the packet to the Open vSwitch integration bridge **br-int**. The forwarding table on **compute0** contains the MAC for **compute0fa:16:3e:6a:43:83** which is located on OpenFlow port 2 and the `ovs-ofctl show br-int` command displays port **patch-tun** with a link to br-tun.

```
[heat-admin@compute0 ~]$ sudo ovs-appctl fdb/show br-int
port  VLAN  MAC          Age
      3     1  fa:16:3e:92:16:1f  48
      2     1  fa:16:3e:6a:43:83  47
      4     1  fa:16:3e:8d:cd:b0  47
[heat-admin@compute0 ~]$ sudo ovs-ofctl show br-int
...output omitted...
1(patch-tun): addr:96:e9:e0:16:9f:a3
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
[heat-admin@compute0 ~]$ sudo ovs-vsctl show
...output omitted...
    Port patch-tun
        Interface patch-tun
            type: patch
            options: {peer=patch-int}
...output omitted...
```

- ③ The Open vSwitch integration bridge **br-int** forwards the packet to the Open vSwitch tunnel bridge **br-tun**.
- ④ The Open vSwitch tunnel bridge **br-tun** forwards the packet to **compute1** over the tunnel interface. **Patch-int** maps to OpenFlow **port 1**. **Port 2** and **port 5** map to the VXLAN tunnels.

```
[heat-admin@compute0 ~]$ sudo ovs-ofctl show br-tun
OFPT_FEATURES_REPLY (xid=0x2): dpid:000092652337b04f
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src mod_dl_dst
mod_nw_src mod_nw_dst mod_nw_tos mod_tp_src mod_tp_dst
1(patch-int): addr:1e:d4:d1:b0:b4:17
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
2(vxlan-ac180201): addr:5a:00:f5:33:24:1d
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
5(vxlan-ac18020c): addr:46:fc:51:d4:cb:5c
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
LOCAL(br-tun): addr:92:65:23:37:b0:4f
    config: PORT_DOWN
    state: LINK_DOWN
    speed: 0 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0
```



Note

The first few flows just redirect to other tables and do not impact this traffic. We show these flows so you can follow the full path through each table.

The first table in the OpenFlow entry bridge states that traffic incoming on **port 1** is resubmitted to flow **table 1**. Therefore inbound traffic to **br-int** goes to **flow table 1**.

```
[heat-admin@compute0 ~]$ sudo ovs-ofctl dump-flows br-tun table=0
NXST_FLOW reply (xid=0x4):
cookie=0xae9558283badfdbe, duration=4374.365s, table=0, n_packets=2493,
n_bytes=243554, idle_age=0, priority=1, in_port=1 actions=resubmit(,1)
```

In flow **table 1**, only the last rule gets applied to DVR traffic, resubmitting the traffic to **table 2**.

```
[heat-admin@compute0 ~]$ sudo ovs-ofctl dump-flows br-tun table=1
...output omitted...
cookie=0xae9558283badfdbe, duration=5357.271s, table=1, n_packets=46, n_bytes=4160,
idle_age=4013, priority=0 actions=resubmit(,2)
```

In OpenFlow **table 2**, there are three entries. The first entry applies to ARP traffic. The second entry applies to unicast traffic, which has a 0 on the last bit of the first octet in the destination MAC address. ICMP traffic is unicast and this entry resubmits to **table 20**. The third entry, which resubmits to **table 22**, applies to broadcast and multicast traffic, which has a 1 in the last bit of the first octet of the destination MAC address.

```
[heat-admin@compute0 ~]$ sudo ovs-ofctl dump-flows br-tun table=2
NXST_FLOW reply (xid=0x4):
cookie=0xae9558283badfdbe, duration=5964.618s, table=2, n_packets=12, n_bytes=504,
idle_age=4659, priority=1,arp,d1_dst=ff:ff:ff:ff:ff:ff actions=resubmit(,21)
cookie=0xae9558283badfdbe, duration=5964.617s,
table=2, n_packets=4021, n_bytes=394002, idle_age=0,
priority=0,d1_dst=00:00:00:00:00:00/01:00:00:00:00:00 actions=resubmit(,20)
cookie=0xae9558283badfdbe, duration=5964.615s,
table=2, n_packets=39, n_bytes=4294, idle_age=4621,
priority=0,d1_dst=01:00:00:00:00:00/01:00:00:00:00:00 actions=resubmit(,22)
```

Flow **table 20** is populated by the L2 population mechanism driver and contains entries for remote MAC **d1_dst=fa:16:3e:c1:d7:95**, tunnel ID **0x1a**, VLAN ID **d1_vlan=3**, and VXLAN tunnel port for OVS **output:5**. If the traffic matches the VLAN ID and destination MAC, the VLAN tag is stripped and the tunnel ID tag **0x1a** is set. Then, the traffic is sent outbound through **port 5**, used for the VXLAN tunnel.

```
[heat-admin@compute0 ~]$ sudo ovs-ofctl dump-flows br-tun table=20
cookie=0xae9558283badfdbe, duration=5333.330s,
table=20, n_packets=4880, n_bytes=478240, idle_age=0,
priority=2,d1_vlan=3,d1_dst=fa:16:3e:c1:d7:95 actions=strip_vlan,load:0x1a->NXM_NX_TUN_ID[],output:5
[heat-admin@compute0 ~]$ sudo ovs-ofctl show br-tun
    5(vxlan-ac18020c): addr:46:fc:51:d4:cb:5c
        config:      0
        state:       0
```

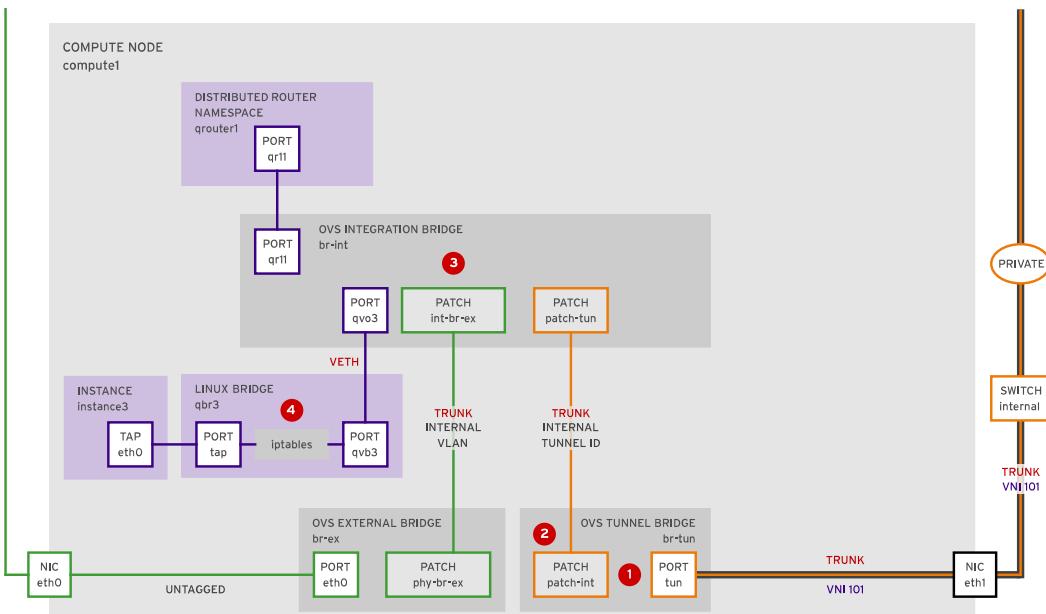
```
speed: 0 Mbps now, 0 Mbps max
```

The OVS port of the tunneling bridge is a VXLAN port with a source of 172.24.2.2 and a destination of 172.24.2.12.

```
[heat-admin@compute0 ~]$ sudo ovs-vsctl show
Port patch-int
Interface patch-int
type: patch
options: {peer=patch-tun}
Port "vxlan-ac18020c"
Interface "vxlan-ac18020c"
type: vxlan
options: {df_default="true", in_key=flow, local_ip="172.24.2.2", out_key=flow,
remote_ip="172.24.2.12"}
```

Inbound To The Destination VM On Another Compute Host

This second narrative describes packets being received by an instance residing on a compute host. Packets were originated by another VM in the same project and subnet, but residing on another compute host. This continuing narrative follows the packet starting at the incoming interface until it reaches the destination. Use the following figure to reference the numbered callouts.



East-west flow in one subnet across two hosts - inbound

- ① The tunnel interface forwards a packet to the Open vSwitch tunnel bridge **br-tun** through **port 2** and decapsulates the traffic. Port vxlan-ac180202 is the other end of the tunnel on **compute1**. The **ovs-vsctl show** command displays **port 2** for the tunnel.

```
[heat-admin@compute1 ~]$ sudo ovs-vsctl show
Port "vxlan-ac180202"
Interface "vxlan-ac180202"
type: vxlan
```

```

        options: {df_default="true", in_key=flow, local_ip="172.24.2.12",
out_key=flow, remote_ip="172.24.2.2"}
    Port br-tun
        Interface br-tun
            type: internal
    Port patch-int
        Interface patch-int
            type: patch
            options: {peer=patch-tun}
Port "vxlan-ac180201"
        Interface "vxlan-ac180201"
            type: vxlan
            options: {df_default="true", in_key=flow, local_ip="172.24.2.12",
out_key=flow, remote_ip="172.24.2.1"}
[heat-admin@compute0 ~]$ sudo ovs-ofctl show br-tun
...output omitted...
2(vxlan-ac180202): addr:86:da:fe:68:12:49
config: 0
state: 0
speed: 0 Mbps now, 0 Mbps max

```

The second entry in flow **table 0** states that traffic inbound on **port 2** is resubmitted to flow **table 4**. Therefore, inbound traffic from the tunnel goes to **table 4**.

```

[heat-admin@compute1 ~]$ sudo ovs-ofctl dump-flows br-tun table=0
NXST_FLOW reply (xid=0x4):Hosted
cookie=0xae9558283badfdbe, duration=4374.365s, table=0, n_packets=2493,
n_bytes=243554, idle_age=0, priority=1,in_port=2 actions=resubmit(,1)
NXST_FLOW reply (xid=0x4):
cookie=0xa8dd6f4eb7e8d344, duration=8304.479s, table=0, n_packets=7837,
n_bytes=768026, idle_age=0, priority=1,in_port=2 actions=resubmit(,4)
...output omitted...

```

The applicable entry in flow **table 4** is identified by the VNI tunnel ID *0x1a*. The action sets 4 as the VLAN tag and resubmits the traffic to **table 9**.

```

[heat-admin@compute1 ~]$ sudo ovs-ofctl dump-flows br-tun table=4
NXST_NXST_FLOW reply (xid=0x4):
...output omitted...
cookie=0xa8dd6f4eb7e8d344, duration=9109.064s, table=4,
n_packets=8642, n_bytes=847474, idle_age=0, priority=1,tun_id=0x1a
actions=mod_vlan_vid:4,resubmit(,9)
cookie=0xa8dd6f4eb7e8d344, duration=10594.342s, table=4, n_packets=0, n_bytes=0,
idle_age=10594, priority=0 actions=drop

```

The last entry in flow **table 9** by default, passes the traffic to flow **table 10**.

```

[heat-admin@compute1 ~]$ sudo ovs-ofctl dump-flows br-tun table=9
NXST_NXST_FLOW reply (xid=0x4):
...output omitted...
cookie=0xa8dd6f4eb7e8d344, duration=11302.969s, table=9, n_packets=3, n_bytes=852,
idle_age=9782,
priority=0 actions=resubmit(,10)

```

Flow **table 10** contains a single entry used to learn the mapping between tunnels and the VM's IP address. When the entry is applied to tunnel traffic, the tunneling bridge creates

an entry in flow **table 20**, which specifies information for return traffic back to the VM, including the appropriate tunnel ID and port.

```
[heat-admin@compute1 ~]$ sudo ovs-ofctl dump-flows br-tun table=10
NXST_FLOW reply (xid=0x4):
cookie=0xa8dd6f4eb7e8d344, duration=11717.864s, table=10, n_packets=3, n_bytes=852,
idle_age=10197,
priority=1 actions=learn(table=20,hard_timeout=300,priority=1,
cookie=0xa8dd6f4eb7e8d344,NXM_OF_VLAN_TCI[0..11],NXM_OF_ETH_DST[]>NXM_OF_ETH_SRC[],
load:0->NXM_OF_VLAN_TCI[],load:NXM_NX_TUN_ID[]->NXM_NX_TUN_ID[],output:0XM_OF_IN_PORT[],output:1
```

Flow **table 20** contains the entry used for return traffic, which can be identified by tunnel ID *0x1a*. The traffic exits outbound on OpenFlow **port 3**.

```
[heat-admin@compute1 ~]$ sudo ovs-ofctl dump-flows br-tun table=20
NXST_NXST_FLOW reply (xid=0x4):
...output omitted...
cookie=0xa8dd6f4eb7e8d344, duration=10685.271s, table=20, n_packets=1, n_bytes=42,
idle_age=10651, priority=2,dl_vlan=4,dl_dst=fa:16:3e:20:74:be
actions=strip_vlan,load:0x1a->NXM_NX_TUN_ID[],output:3
```

- 2 The Open vSwitch tunnel bridge **br-tun** forwards the packet to the Open vSwitch integration bridge **br-int**. **Port 1** is the patch to **br-int**.

```
[heat-admin@compute1 ~]$ sudo ovs-ofctl show br-tun
...output omitted...
1(patch-int): addr:96:e9:e0:16:9f:a3
    config:      0
    state:       0
    speed: 0 Mbps now, 0 Mbps max
```

- 3 The Open vSwitch integration bridge **br-int** forwards the packet to the Linux bridge *qbr8bf6552-04*.

```
[heat-admin@compute0 ~]$ brctl show
bridge name      bridge id      STP enabled interfaces
qbr8bf6552-04  8000.7a0086c25d19    no qvb8bf6552-04
                           tapf8bf6552-04
```

- 4 Security group rules on the Linux bridge *qbr-f8bf6552-04* processes egress firewall rules and state tracking for the packet.

The Linux bridge *qbr-f8bf6552-04* forwards the packet to the *instance1* tap interface.

East-west Packet Flow Between Different Subnets

Compute nodes route east-west network traffic between project networks on an outbound router, bypassing controller node processing, using the instances' fixed and floating IP addresses. The narrative is in two parts:

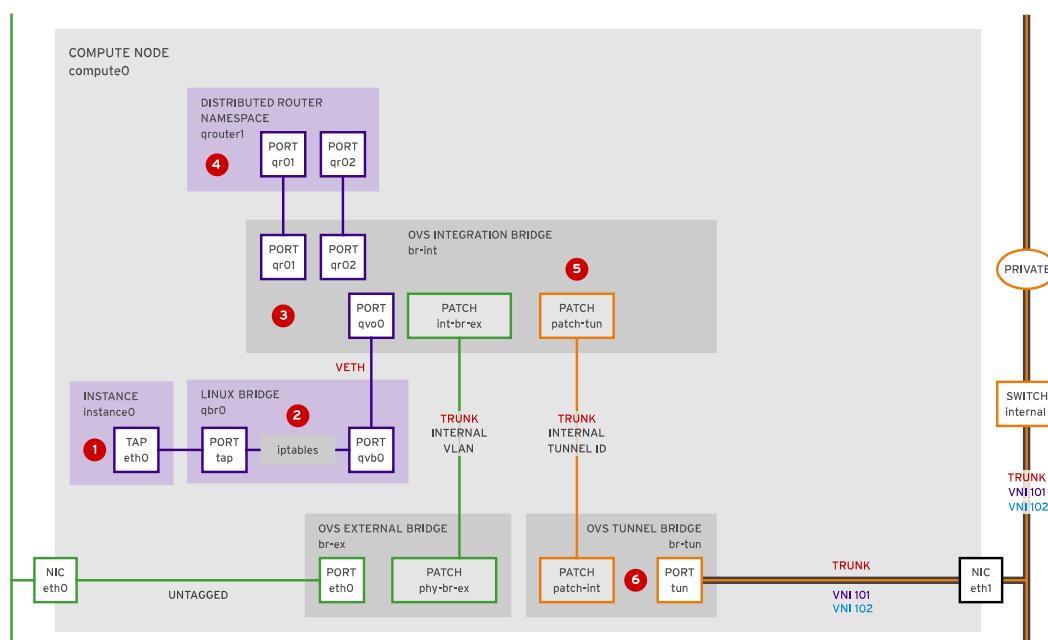
- Outbound from the source VM in one project on any compute host.
- Inbound to the destination VM in another project and on another compute host.

Resource Objects Referenced In This Narrative

Resource Objects	Description
network0	Project tenant network with subnet 192.168.0.0/24 and gateway 192.168.0.1
network1	Project tenant network with subnet 192.168.1.0/24 and gateway 192.168.1.1
compute0	Compute Service node hosting server instance0 with IP 192.168.0.4
compute1	Compute Service node hosting server instance1 with IP 192.168.1.8
instance0	Server instance hosted on compute0 , using project network network0
instance1	Server instance hosted on compute1 , using project network network1

Outbound From The Source VM In One Project On Any Compute Host

This first narrative describes outbound packets from a instance being processed for routing and then being transferred to another compute node using the internal network using a VXLAN tunnel. Use the following figure to reference the numbered callouts.



East-west flow in two subnets across two hosts - outbound

- ① The instance0 TAP interface `tapf8bf6112-02` forwards packet to Linux bridge `qbrf8bf6112-02`.

```
[heat-admin@compute1 ~]$ brctl show
bridge name      bridge id      STP enabled interfaces
qbrf8bf6112-02  8000.7a0086c25d19    no qvbf8bf6112-02
                                tapf8bf6112-02
```

- ② Security group rules on the Linux bridge `qbrf8bf6112-02` processes state tracking for the packet. The security group rules can be viewed with the `iptables` command and the `grep` option, using the first three digits of the TAP interface. The letter `o` stands for *option* and is required before the first three digits of the `tap` interface when using `grep`.

```
[heat-admin@compute0 ~]$ sudo iptables -S | grep of8b
-N neutron-openvswi-o1bc37093-9
-A neutron-openvswi-INPUT -m physdev --physdev-in tap1bc37093-9b --physdev-is-
bridged -m comment --comment "Direct incoming traffic from VM to the security group
chain." -j neutron-openvswi-o1bc37093-9
-A neutron-openvswi-o1bc37093-9 -s 0.0.0.0/32 -d 255.255.255.255/32 -p udp -m udp --sport 68 --dport 67 -m comment --comment "Allow DHCP client traffic." -j RETURN
-A neutron-openvswi-o1bc37093-9 -j neutron-openvswi-s1bc37093-9
-A neutron-openvswi-o1bc37093-9 -p udp -m udp --sport 68 --dport 67 -m comment --
comment "Allow DHCP client traffic." -j RETURN
-A neutron-openvswi-o1bc37093-9 -p udp -m udp --sport 67 -m udp --dport 68 -m
comment --comment "Prevent DHCP Spoofing by VM." -j DROP
-A neutron-openvswi-o1bc37093-9 -m state --state RELATED,ESTABLISHED -m comment --
comment "Direct packets associated with a known session to the RETURN chain." -j
RETURN
-A neutron-openvswi-o1bc37093-9 -j RETURN
-A neutron-openvswi-o1bc37093-9 -m state --state INVALID -m comment --comment "Drop
packets that appear related to an existing connection (e.g. TCP ACK/FIN) but do not
have an entry in conntrack." -j DROP
-A neutron-openvswi-o1bc37093-9 -m comment --comment "Send unmatched traffic to the
fallback chain." -j neutron-openvswi-sg-fallback
-A neutron-openvswi-sg-chain -m physdev --physdev-in tap1bc37093-9b --physdev-is-
bridged -m comment --comment "Jump to the VM specific chain." -j neutron-openvswi-
o1bc37093-9
```

The Linux bridge `qbrf8bf6552-04` forwards the packet to the Open vSwitch integration bridge `br-int`. The forwarding table on `compute0` contains the MAC for `compute1` `fa:16:3e:6a:43:83` which is located on OpenFlow **port 2**.

```
[heat-admin@compute0 ~]$ sudo ovs-appctl fdb/show br-int
port  VLAN  MAC          Age
      3     1  fa:16:3e:92:16:1f  48
      2     1  fa:16:3e:6a:43:83  47
      4     1  fa:16:3e:8d:cd:b0  47
```

- ③ The Open vSwitch integration bridge `br-int` forwards the packet to the project network `network0` interface in the distributed router namespace `qrouter`.

```
[heat-admin@compute0 ~]$ sudo ip netns
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284
[heat-admin@compute0 ~]$ sudo ip -n \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 addr show
...output omitted...
21:qr-f22a5dfa-1b: BROADCAST,MULTICAST,UP,LOWER_UP mtu 1446 qdisc noqueue state
UNKNOWN qlen 1000
    link/ether fa:16:3e:17:bf:c8 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.1/24 brd 192.168.1.255 scope global qr-f22a5dfa-1b
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe17:bfc8/64 scope link
        valid_lft forever preferred_lft forever
```

- ④ The project network `network0` interface in the `qrouter` namespace forwards the packet to the Open vSwitch integration bridge `br-int`.

```
[heat-admin@compute0 ~]$ sudo ip -n \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 addr show
...output omitted...
```

```

20: qr-b21be091-13: BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue state UNKNOWN
    qlen 1000
    link/ether fa:16:3e:08:14:ac brd ff:ff:ff:ff:ff:ff
    inet 192.168.2.1/24 brd 192.168.2.255 scope global qr-b21be091-13
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe08:14ac/64 scope link
        valid_lft forever preferred_lft forever
[heat-admin@compute0 ~]$ sudo ip netns exec \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 tcptdump -e -n -i qr-b21be091-13
tcptdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on qr-b21be091-13, link-type EN10MB (Ethernet), capture size 262144 bytes
23:46:51.494240 fa:16:3e:08:14:ac > fa:16:3e:0a:c0:63, ethertype IPv4 (0x0800),
  length 98: 192.168.1.5 > 192.168.2.8: ICMP echo request, id 10616, seq 8, length 64
...output omitted...

```

Flow **table 1** contains an entry that matches traffic for **vlan4** and the source MAC of the router **fa:16:3e:08:14:ac**. The *action* substitute the router source MAC address *fa:16:3e:08:14:ac* with the DVR MAC *fa:16:3f:ae:b2:47*. The DVR MAC is used so that it does not appear as though there is more than one host with the same MAC address. When the traffic reaches **compute1**, the MAC is restored to the original one for the router. At the end of the OpenFlow entry, the traffic is resubmitted to **table 2**.

```

[heat-admin@compute1 ~]$ sudo ovs-ofctl dump-flows br-tun table=1
...output omitted...
cookie=0xac9953b8f33033a7, duration=5275.844s, table=1, n_packets=261,
  n_bytes=25578,
idle_age=857, priority=1,dl_vlan=4,dl_src=fa:16:3e:08:14:ac
actions=mod_dl_src:fa:16:3f:ae:b2:47,resubmit(,2)

```



Note

The first few flows just redirect to other tables and do not impact this traffic. We show these flows so you can follow the full path through each table.

In OpenFlow **table 2**, there are three entries. The first entries applies to ARP traffic. The second entry, which resubmits to **table 20**, applies to unicast traffic, which has a 0 on the last bit of the first octet in the destination MAC address. The third entry, which redirects to **table 22**, applies to unicast and multicast traffic, which has a 1 in the last bit of the first octet of the destination MAC address.

```

[heat-admin@compute0 ~]$ sudo ovs-ofctl dump-flows br-tun table=1
NXST_FLOW reply (xid=0x4):
cookie=0xae9558283badfdbe, duration=5964.618s, table=2, n_packets=12, n_bytes=504,
idle_age=4659, priority=1,arp,d1_dst=ff:ff:ff:ff:ff:ff actions=resubmit(,21)
cookie=0xae9558283badfdbe, duration=5964.617s, table=2,
n_packets=4021, n_bytes=394002, idle_age=0,
priority=0,dl_dst=00:00:00:00:00:00/01:00:00:00:00:00 actions=resubmit(,20)
cookie=0xae9558283badfdbe, duration=5964.615s, table=2,
n_packets=39, n_bytes=4294, idle_age=4621,
priority=0,dl_dst=01:00:00:00:00:00/01:00:00:00:00:00 actions=resubmit(,22)

```

Flow **table 20** is populated by the L2 population mechanism driver and contains entries for remote MAC *dl_dst=d1_dst=fa:16:3e:c1:d7:95*, tunnel ID *0x1a*, VLAN ID *dl_vlan=3*, and VXLAN tunnel port for OVS *output:5*. If the traffic matches the VLAN ID and destination

MAC, the VLAN tag is stripped and the tunnel ID tag is set to *0x1a*. Then, the traffic is sent outbound through **port 5** used for the VXLAN tunnel.

```
[heat-admin@compute0 ~]$ sudo ovs-ofctl dump-flows br-tun table=20
cookie=0xae9558283badfdbe, duration=5333.330s,
  table=20, n_packets=4880, n_bytes=478240, idle_age=0,
  priority=2,dl_vlan=3,dl_dst=fa:16:3e:c1:d7:95 actions=strip_vlan,load:0x1a-
>NXM_NX_TUN_ID[],output:5
[heat-admin@compute0 ~]$ sudo ovs-ofctl show br-tun
  5(vxlan-ac18020c): addr:46:fc:51:d4:cb:5c
    config:      0
    state:       0
    speed: 0 Mbps now, 0 Mbps max
```

The Open vSwitch integration bridge **br-int** modifies the packet to contain the internal tag for project network **network0**. The interface *qvo8bf6552-04* is the other end of *qvb8bf6552-04*, and carries traffic to and from the firewall bridge. The label, **tag: 3**, is an access port attached to VLAN 3.

```
[heat-admin@compute0 ~]$ sudo ovs-vsctl show
...output omitted...
  Port "qvo8bf6552-04"
    tag: 3
    Interface "qvo8bf6552-04"
  Port int-br-ex
    Interface int-br-ex
      type: patch
      options: {peer=phy-br-ex}
  Port "qr-b21be091-13"
    tag: 4
    Interface "qr-b21be091-13"
      type: internal
  ovs_version: "2.6.1"
```

- 5 The Open vSwitch integration bridge **br-int** forwards the packet to the Open vSwitch tunnel bridge **br-tun**.

```
[heat-admin@compute0 ~]$ sudo ovs-vsctl show
Bridge br-tun
  network "tcp:127.0.0.1:6633"
    is_connected: true
  fail_mode: secure
  Port br-tun
    Interface br-tun
      type: internal
  Port patch-int
    Interface patch-int
      type: patch
      options: {peer=patch-tun}
  Port "vxlan-ac18020c"
    Interface "vxlan-ac18020c"
      type: vxlan
      options: {df_default="true", in_key=flow, local_ip="172.24.2.2",
out_key=flow, remote_ip="172.24.2.12"}
    Port "vxlan-ac180201"
      Interface "vxlan-ac180201"
        type: vxlan
```

```
options: {df_default="true", in_key=flow, local_ip="172.24.2.2",
out_key=flow, remote_ip="172.24.2.1"}
```

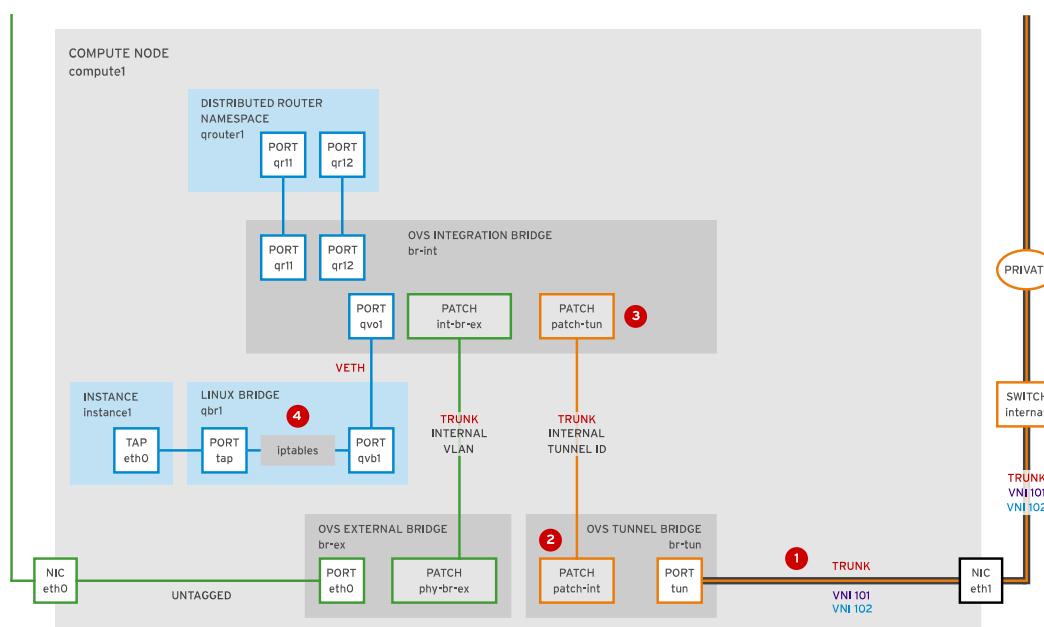
The Open vSwitch tunnel bridge **br-tun** replaces the packet source MAC address of project **network0**'s gateway with the DVR's MAC address.

The Open vSwitch tunnel bridge **br-tun** wraps the packet in a VXLAN tunnel that contains a VNI that is associated with the project network0.

- ⑥ The Open vSwitch tunnel bridge **br-tun** forwards the packet to **compute1** via the tunnel interface.

Inbound To The Destination VM In Another Project And On Another Compute Host

This second narrative describes inbound packets received on the internal tunneled interface, and directed to the destination instance by the integration bridge. Use the following figure to reference the numbered callouts.



East-west flow on two subnets across two hosts - inbound

- ① The tunnel interface forwards the packet to the Open vSwitch tunnel bridge **br-tun**.

Flow **table 0** specifies that traffic inbound on **port 2** is resubmitted to flow **table 4**. Therefore, inbound traffic from the tunnel goes to flow **table 4**.

```
[heat-admin@compute1 ~]$ sudo ovs-ofctl dump-flows br-tun table=0
NXST_FLOW reply (xid=0x4):
cookie=0xae9558283badfdb, duration=4374.365s, table=0, n_packets=2493,
n_bytes=243554, idle_age=0, priority=1,in_port=1 actions=resubmit(,1)
NXST_FLOW reply (xid=0x4):
cookie=0xa8dd6f4eb7e8d344, duration=8304.479s, table=0, n_packets=7837,
n_bytes=768026, idle_age=0, priority=1,in_port=2 actions=resubmit(,4)
...output omitted...
```

The applicable entry in flow **table 4** is identified by the VNI tunnel ID *0x49*. The action tags the traffic with VLAN 4 tag and resubmits the traffic to **table 9**

```
[heat-admin@compute1 ~]$ sudo ovs-ofctl dump-flows br-tun table=4
NXST_NXST_FLOW reply (xid=0x4):
...output omitted...
cookie=0xa8dd6f4eb7e8d344, duration=9109.064s,
table=4, n_packets=8642, n_bytes=847474, idle_age=0,
priority=1,tun_id=0x49 actions=mod_vlan_vid:4,resubmit(,9)
cookie=0xa8dd6f4eb7e8d344, duration=10594.342s, table=4, n_packets=0, n_bytes=0,
idle_age=10594, priority=0 actions=drop
```

Flow **table 9** contains an entry matching traffic from the source DVR MAC address on **compute0** and exits **port 1**, which is a patch port connected to **br-int**.

```
[heat-admin@compute1 ~]$ sudo ovs-ofctl dump-flows br-tun table=20
NXST_NXST_FLOW reply (xid=0x4):
...output omitted...
cookie=0xa8dd6f4eb7e8d344, duration=10685.271s, table=9, n_packets=1, n_bytes=42,
idle_age=10651, priority=2,dl_vlan=4,dl_dst=fa:16:3e:20:74:be
actions=strip_vlan,load:0x49->NXM_NX_TUN_ID[],output:1
```

There is not an action to resubmit the traffic, so it goes to flow **table 0**. In flow **table 0**, there is an entry to match the DVR MAC address of **compute1**. The traffic is then resubmitted to flow **table 1**.

Flow **table 1** contains an entry to match traffic that contains the destination MAC address of instance2. The action removes the VLAN tag and restores the source MAC address of the interface of the DVR router on network1. The traffic is then resubmitted to **port 4**, which is the port for instance1.

```
[heat-admin@compute0 ~]$ sudo watch -n.5 "ovs-ofctl dump-flows br-tun table=1"
cookie=0xae9558283badfdb, duration=5333.330s, table=1, n_packets=4880,
n_bytes=478240, idle_age=0,
priority=2,dl_vlan=3,dl_dst=fa:16:3e:c1:d7:95
actions=strip_vlan,mod_dl_src:xx:xx:xx:xx,output:4
```

- ② The Open vSwitch integration bridge **br-int** forwards the packet to the Linux bridge **qbr**.
- ③ Security group rules on the Linux bridge **qbr** process a set of ip filter rules and state tracking for the packet.
- ④ The Linux bridge **qbr** forwards the packet to the instance1 **tap** interface.

North-south Packet Flow With Floating IP Requirement

Compute nodes route north-south traffic between project and external networks directly on the compute node's DVR router, avoiding any use of the controller node. When IPv4 instances are deployed on self-service tenant networks, floating IP addresses are required to communicate with external systems. Using the FIP namespace, the compute node performs the NAT processing that formerly required a legacy router on a controller node, but now is only an extra step during DVR routing. The narrative is in two parts:

- Outbound from any VM on any compute host with FIP processing.

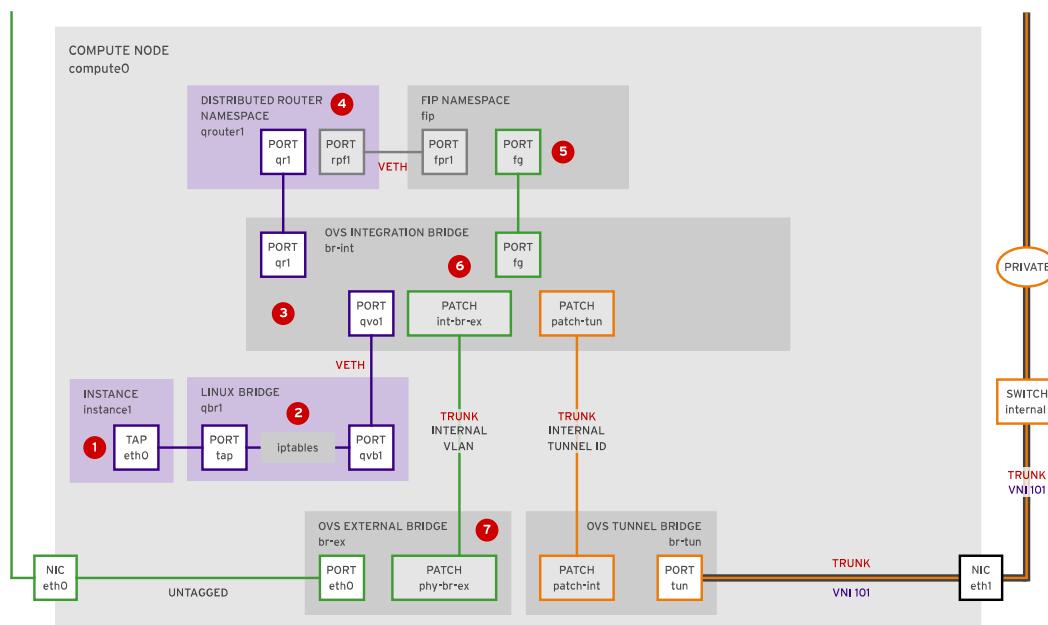
- Inbound to any VM on any compute host with FIP processing.

Resource Objects Referenced In This Narrative

Resource	Description
external	External network with subnet 172.25.250.0/24 and gateway 172.25.250.1
floating IP	Floating IP address 172.25.250.113 reserved from external network, assigned to instance1
network1	Project tenant network with subnet 192.168.1.0/24 and gateway 192.168.1.1
compute0	Compute Service node hosting server instance1 with IP 192.168.1.8
instance1	Server instance hosted on compute0 , using project network network1

Outbound From Any VM On Any Compute Host With FIP Processing

This first narrative describes outbound packets from an instance being routed and processed for network address translation, then exiting the external interface. Use the following figure to reference the numbered callouts.



North-south flow on one subnet with FIP handling - outbound

- ① The instance1 TAP interface tapf8bf6112-02 forwards a packet containing the destination MAC address, which resides in another network, to the Linux bridge qbrf8bf6112-02.

```
[heat-admin@compute1 ~]$ brctl show
bridge name      bridge id      STP enabled interfaces
qbrf8bf6112-02  8000.7a0086c25d19    no qvbf8bf6112-02
                                tapf8bf6112-02
```

- ② Security group rules on the Linux bridge qbrf8bf6112-02 process state tracking for the packet. The security group rules can be viewed with the **iptables** command and the **grep** option using the first three digits of the **tap** interface. The letter **o** stands for *option* and is required before the first three digits of the TAP interface when using **grep**.

```
[heat-admin@compute0 ~]$ sudo iptables -S | grep of8b
-N neutron-openvswi-o1bc37093-9
-A neutron-openvswi-INPUT -m physdev --physdev-in tap1bc37093-9b --physdev-is-
bridged -m comment --comment "Direct incoming traffic from VM to the security group
chain." -j neutron-openvswi-o1bc37093-9
-A neutron-openvswi-o1bc37093-9 -s 0.0.0.0/32 -d 255.255.255.255/32 -p udp -m udp --sport 68 --dport 67 -m comment --comment "Allow DHCP client traffic." -j RETURN
-A neutron-openvswi-o1bc37093-9 -j neutron-openvswi-s1bc37093-9
-A neutron-openvswi-o1bc37093-9 -p udp -m udp --sport 68 --dport 67 -m comment --
comment "Allow DHCP client traffic." -j RETURN
-A neutron-openvswi-o1bc37093-9 -p udp -m udp --sport 67 -m udp --dport 68 -m
comment --comment "Prevent DHCP Spoofing by VM." -j DROP
-A neutron-openvswi-o1bc37093-9 -m state --state RELATED,ESTABLISHED -m comment --
comment "Direct packets associated with a known session to the RETURN chain." -j
RETURN
-A neutron-openvswi-o1bc37093-9 -j RETURN
-A neutron-openvswi-o1bc37093-9 -m state --state INVALID -m comment --comment "Drop
packets that appear related to an existing connection (e.g. TCP ACK/FIN) but do not
have an entry in conntrack." -j DROP
-A neutron-openvswi-o1bc37093-9 -m comment --comment "Send unmatched traffic to the
fallback chain." -j neutron-openvswi-sg-fallback
-A neutron-openvswi-sg-chain -m physdev --physdev-in tap1bc37093-9b --physdev-is-
bridged -m comment --comment "Jump to the VM specific chain." -j neutron-openvswi-
o1bc37093-9
```

The Linux bridge qbrf8bf6552-04 forwards the packet to the Open vSwitch integration bridge **br-int**. The forwarding table on **compute0** contains the MAC for **compute1** fa:16:3e:6a:43:83 is located on OpenFlow **port 2**.

```
[heat-admin@compute0 ~]$ sudo ovs-appctl fdb/show br-int
port  VLAN  MAC          Age
      3     1  fa:16:3e:92:16:1f  48
      2     1  fa:16:3e:6a:43:83  47
      4     1  fa:16:3e:8d:cd:b0  47
```

- ③ The Open vSwitch integration bridge **br-int** forwards the packet to the qr-f131be64-1b interface in the distributed router namespace qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284. The qr-f131be64-1b interface contains the project network gateway IP address 192.168.1.1. There is not a route for the external network in the routing table. The routing table labeled **main** does not match the traffic, however there is a policy route **16** that matches traffic from 192.168.1.8 and performs a lookup for the routing table.

```
[heat-admin@compute1 ~]$ sudo ip -n \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 route
192.168.1.0/24 dev qr-f131be64-1b proto kernel scope link src 192.168.1.1
192.168.2.0/24 dev qr-b21be091-13 proto kernel scope link src 192.168.2.1
[heat-admin@compute1 ~]$ sudo ip -n \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 rule show
0: from all lookup local
32766: from all lookup main
32767: from all lookup default
57481: from 192.168.1.8 lookup 16
3232235777: from 192.168.1.1/24 lookup 3232235777
3232236033: from 192.168.2.1/24 lookup 3232236033
```

Routing table **16** contains a default route for traffic to use interface **rfp-e8d3e5a5-b** via 192.168.1.7. The **rfp-e8d3e5a5-b** interface is the link to the floating IP namespace.

```
[heat-admin@compute0 ~]$ sudo ip -n \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 route table show 16 addr show
default via 169.254.106.115 dev rfp-e8d3e5a5-b
```

The iptables service performs SNAT on the packet using the **rfp** interface **rfp-e8d3e5a5-b** as the source IP address. The **rfp** interface contains the instance's floating IP address 172.25.250.113. An iptables rule modifies the source IP address of 192.168.1.8 for **compute1** to the floating IP 172.25.250.113.

```
[heat-admin@compute0 ~]$ sudo ip netns exec \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 iptables -s -t nat
-A neutron-13-agent-float-snat -s 192.168.1.8/32 -j SNAT --to-source 172.25.250.113
[heat-admin@compute0 ~]$ sudo ip -n \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 addr show
...output omitted...
28: rfp-e8d3e5a5-b:BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue state
UNKNOWN qlen 1000
    link/ether fa:16:3e:17:bfc8 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.1/24 brd 192.168.1.255 scope global qr-f22a5dfa-1b
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe17:bfc8/64 scope link
        valid_lft forever preferred_lft forever
```

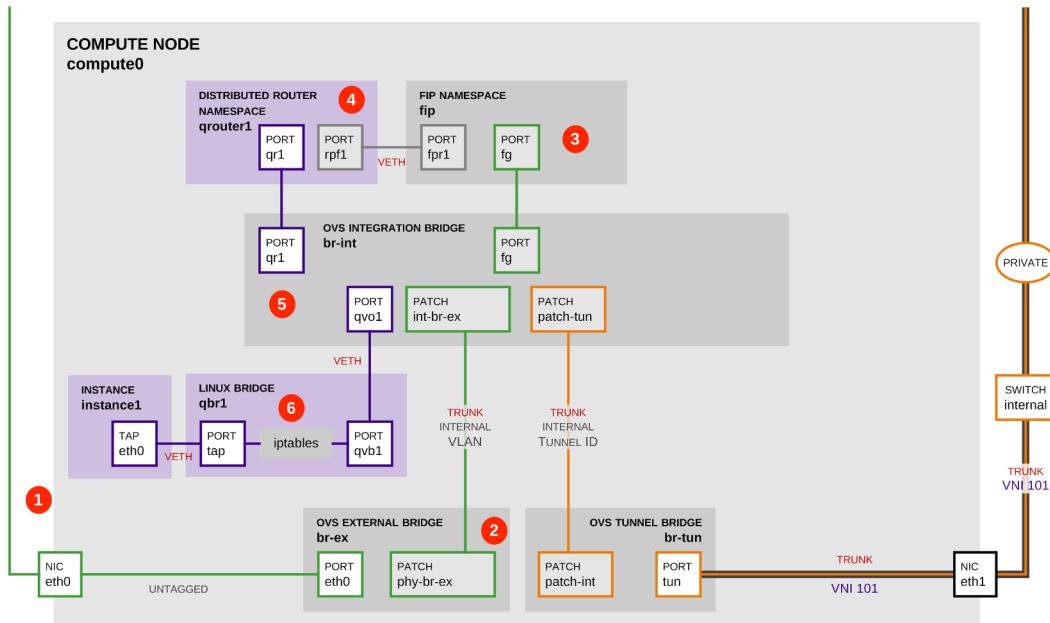
- ④ The distributed router namespace **qrouter** routes the packet to the floating IP namespace FIP using DVR internal IP addresses. The **fpr** interface resides in the FIP namespace and the **rfp** interface resides in the **qrouter** namespace.

```
[heat-admin@compute0 ~]$ sudo ip netns
fip-b4d8228c-2415-4222-80bc-500fda5fa696
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284
[heat-admin@compute1 ~]$ sudo ip -n \
fip-b4d8228c-2415-4222-80bc-500fda5fa696 route
169.254.106.114/31 dev fpr-e8d3e5a5-b proto kernel scope link src 169.254.106.115
172.25.250.0/24 dev fg-75d40aeb-e2 proto kernel scope link src 172.25.250.101
172.25.250.113 via 169.254.106.114 dev fpr-e8d3e5a5-b
```

- ⑤ The **fg** interface, in the floating IP namespace FIP, forwards the packet to the Open vSwitch external bridge **br-ex**. The **fg** interface contains the project router's external IP address.
- ⑥ The Open vSwitch external bridge **br-ex** forwards the packet to the external network over the external interface.

Inbound To Any VM On Any Compute Host With FIP Processing

This second narrative describes inbound packets being received on an external interface, then routed and processed for network address translation and finally being passed to the destination instance by the integration bridge. Use the following figure to reference the numbered callouts.



North-south flow on one subnet with FIP handling - inbound

- 1 The external interface forwards the packet to the Open vSwitch external bridge **br-ex**. The packet contains the destination IP address.
- 2 The Open vSwitch external bridge **br-ex** forwards the packet to the **fg** interface in the floating IP namespace FIP. The **fg-75d40aeb-e2** interface responds to any ARP requests for the instance floating IP address 172.25.250.113.

```
[heat-admin@compute1 ~]$ sudo ip -n \fip-b4d8228c-2415-4222-80bc-500fd45fa696 link
...output omitted...
20: fg-75d40aeb-e2: BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1496 qdisc noqueue state UNKNOWN mode DEFAULT qlen 1000
    link/ether fa:16:3e:fa:fe:9b brd ff:ff:ff:ff:ff:ff
```

- 3 The floating IP namespace FIP routes the packet to the distributed router namespace qrouter using DVR internal IP address. The **fpr** interface resides in the FIP namespace and the **rpf** interface resides in the qrouter namespace.

The iptables service in the distributed router namespace qrouter performs a DNAT on the packet using the destination IP address. The **qr** interface contains the project network gateway IP address.

```
[heat-admin@compute1 ~]$ sudo ip netns exec \qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 iptables -S -t nat
-A neutron-13-agent-PREROUTING -d 172.25.250.113/32 -i rfp-e8d3e5a5-b
-j DNAT --to-destination 192.168.1.8
```

- 4 The qrouter forwards the packet to the Open vSwitch integration bridge **br-int**.
- 5 The packet is forwarded from the Open vSwitch br-int integration bridge to the **qbr** Linux bridge.

The packet is processed by a set of ip filter and state tracking security group rules on the Linux bridge **qbr**.

- ➏ Finally, the packet is forwarded from the **qbr** Linux bridge to the **TAP** interface of the destination instance.

North-south Packet Flow Without Floating IP Requirement

For instances with a fixed IP address using project networks on distributed routers, the controller node routes north-south network traffic between project and external networks. The narrative is in two parts:

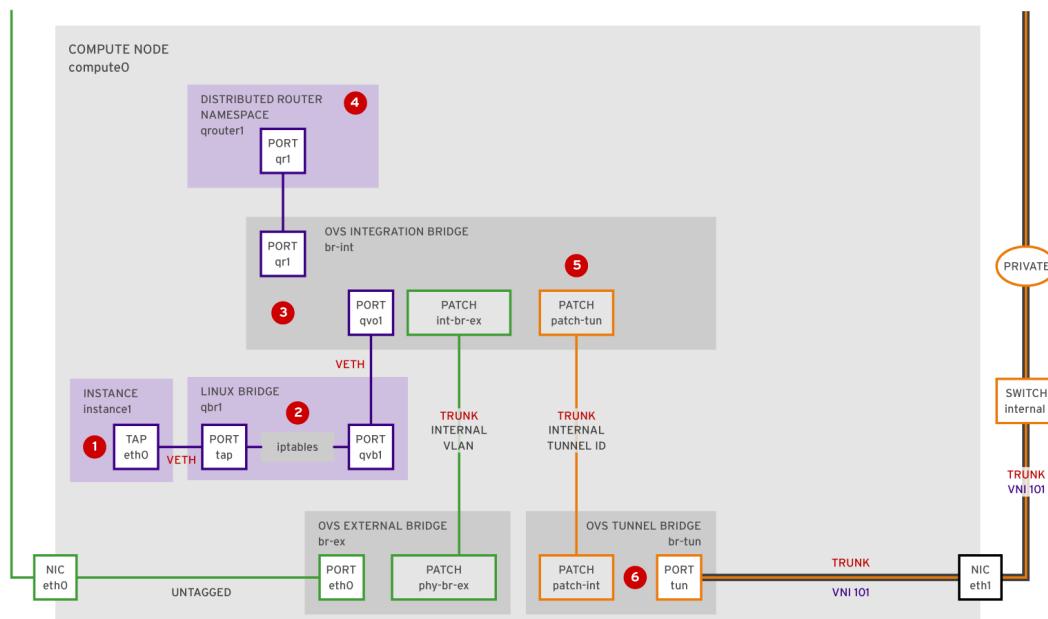
- Outbound from any VM on any compute host without FIP processing.
- Continuing outbound on any controller node to external network.

Resource Objects Referenced In This Narrative

Resource	Description
external	External network with subnet 172.25.250.0/24 and gateway 172.25.250.1
floating IP	A floating IP address reserved from the external network 172.25.250.113, assigned to instance 1
network1	Project tenant network with subnet 192.168.1.0/24 and gateway 192.168.1.1
compute1	Compute Service node hosting server instance1 with IP 192.168.1.8
instance1	Server instance hosted on compute1, using project network network1

Outbound From Any VM On Any Compute Host Without FIP processing

This initial narrative describes outbound packets from an instance being touted to the internal interface and VXLAN tunnel to be sent to the controller node. For instances with a fixed IP address using project networks on distributed routers, the controller node routes north-south network traffic between project and external networks. Use the following figure to reference the numbered callouts.



North-south flow on one subnet without FIP handling - compute node

- ① A packet is forwarded from the **tap** interface of instance1 to the **qbr** Linux bridge. The packet contains the destination MAC address because the destination resides on another network.

- ② The packet is processed by a set of ip filter and state tracking security group rules on the Linux bridge **qbr**.

The packet is forwarded from the **qbr** Linux bridge to the **br-int** Open vSwitch integration bridge.

- ③ The packet is forwarded from the **br-int** Open vSwitch integration bridge to the **qr-f22a5dfa-1b** interface within the **qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284** distributed router namespace. The **qr-f22a5dfa-1b** interface contains the project network gateway IP address 192.168.1.1. There is no route for the external network in the routing table. The routing table labeled **main** does not match the traffic, however the policy route **3232235777** matches traffic from 192.168.1.1/24 and performs a lookup for the routing table.

```
[heat-admin@compute1 ~]$ sudo ip -n \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 route
192.168.1.0/24 dev qr-f22a5dfa-1b proto kernel scope link src 192.168.1.1
...output omitted...

[heat-admin@compute1 ~]$ sudo ip -n \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 rule show
0: from all lookup local
32766: from all lookup main
32767: from all lookup default
3232235777: from 192.168.1.1/24 lookup 3232235777
```

Routing table **3232235777** contains a single route that redirects traffic to 192.168.1.7 using the **qr-f22a5dfa-1b**. The **qr-f22a5dfa-1b** interface is the link to the SNAT namespace.

```
[heat-admin@compute1 ~]$ sudo ip -n \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 route table show 3232235777 addr show
default via 192.168.1.7 dev qr-f22a5dfa-1b
```

The Open vSwitch integration bridge **br-int** adds the internal tag for project network1 to the packet.

- ④ The packet is forwarded from **br-int** to **qr**, the project network1 gateway interface, in the **qrouter** namespace.

The **qrouter** namespace resolves the MAC address of **sg**, the project network1 SNAT interface, in the SNAT namespace **snat-e8d3e5a5-b35e-4a36-a912-755ac9144284**. The packet is forwarded from **sg** to **br-int**.

```
[heat-admin@compute1 ~]$ sudo ip netns
snat-e8d3e5a5-b35e-4a36-a912-755ac9144284
[heat-admin@compute1 ~]$ sudo ip -n snat-e8d3e5a5-b35e-4a36-a912-755ac9144284 addr
show
...output omitted...
```

```

17: sg-9e8e4633-1e: BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue state UNKNOWN qlen 1000
    link/ether fa:16:3e:c2:3a:eb brd ff:ff:ff:ff:ff:ff
    inet 192.168.2.9/24 brd 192.168.2.255 scope global sg-9e8e4633-1e
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe:3aeb/64 scope link
        valid_lft forever preferred_lft forever
19: sg-ed9249fa-cb: BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue state UNKNOWN qlen 1000
    link/ether fa:16:3e:fb:59:fa brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.7/24 brd 192.168.1.255 scope global sg-ed9249fa-cb
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fb:59fa/64 scope link
        valid_lft forever preferred_lft forever

```

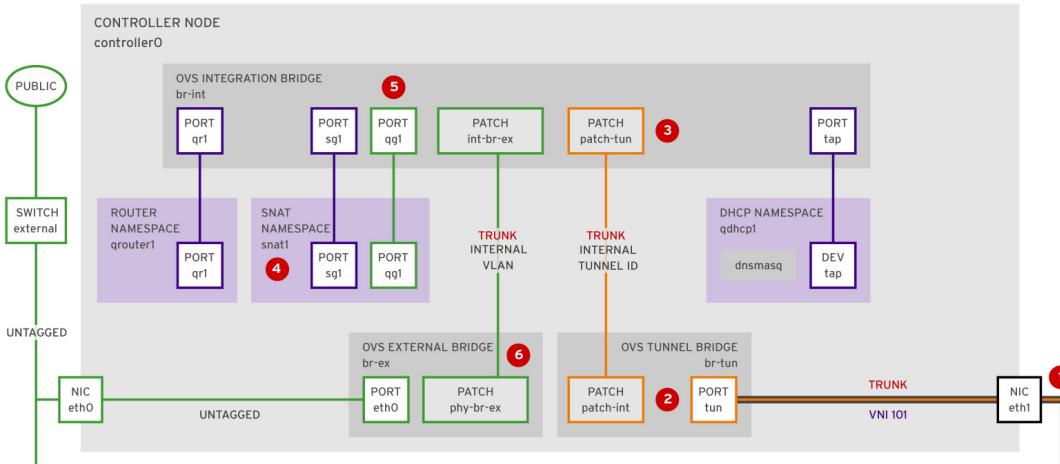
- ⑤ The packet is forwarded from **br-int** to **br-tun**, the Open vSwitch tunnel bridge.

The Open vSwitch tunnel bridge, **br-tun**, replaces the packet's source MAC address, currently set to instance1, with the DVR's MAC address.

- ⑥ Finally, the packet is forwarded from **br-tun** to the network node over the tunnel interface.

Continuing Outbound On Any Controller Node To External Network

This continuing narrative describes outbound packets on the controller node. The packet from the previous narrative is received on the internal tunneled interface, then network address translation process and sent out the external interface. Use the following figure to reference the numbered callouts.



North-south flow on one subnet without FIP handling - controller node

- ① A packet is forwarded from a tunnel interface to **br-tun**, the Open vSwitch tunnel bridge.

The Open vSwitch tunnel bridge **br-tun** unwraps the packet.

- ② The packet is forwarded from **br-tun** to **br-int**, the Open vSwitch integration bridge.

The Open vSwitch integration bridge **br-int** replaces the DVR's source MAC address with project network1's gateway MAC address and adds the internal tag for project network 1..

- ③ The packet is forwarded from **br-int** to **sg**, an interface in the SNAT namespace.

The iptables service performs an SNAT on the packet using the project network1 router interface's IP address on the **qg** interface.

- ④ The **qg** interface forwards the packet to the Open vSwitch external bridge **br-ex**.
- ⑤ The packet is processed by a set of ip filter and state tracking security group rules on the Linux bridge **qbr**.
- ⑥ Finally, the packet is forwarded from the **br-ex** Open vSwitch external bridge to the external network over the external interface.

Configuring Distributed Virtual Routing

DVR can be configured using an environmental YAML file. The **neutron-ovs-dvr.yaml** file is used to configure the required parameters for DVR. The following list provides specific requirements for configuring DVR:

- Both compute and controller nodes must have an interface connected to the physical network for external network traffic.
- A bridge is required on compute and controller nodes, with an interface for external network traffic.
- The configured bridge must be allowed in the Neutron configuration for the bridge to be used.
- There are two files which contain a value that must match. The **environments/neutron-ovs-dvr.yaml** file contains a value for **OS::TripleO::Compute::Net::SoftwareConfig**, which must match the value for the same variable contained in the environmental file used to deploy the overcloud. Without this configuration, the appropriate external network bridge for the compute node's L3 agent will *not* be created.
- Configure a neutron port for the compute node on the external network by modifying **OS::TripleO::Compute::Ports::ExternalPort** to include the path to **external.yaml**. **OS::TripleO::Compute::Ports::ExternalPort: ./network/ports/external.yaml**.
- Include **neutron-ovs-dvr.yaml** as an environment file when deploying the overcloud.
- Verify that L3 HA is disabled.

References

Further information is available in the chapter on Configuring Distributed Virtual Routing in the *Red Hat OpenStack Platform 10 Networking Guide* at
https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/

Quiz: DVR Architecture

Choose the correct answer(s) to the following questions:

1. Assuming north-south traffic flow without a FIP, which interface processes a set of ip filter and state tracking security group rules?
 - a. **qbr**
 - b. **sg**
 - c. **br-tun**
 - d. **frp**
2. Assuming an instance has the TAP interface **tapf1af5332-02**, what command would you use to view the outbound iptables rules referencing that TAP interface?
 - a. sudo iptables -S | grep f1a
 - b. sudo iptables show tap
 - c. sudo iptables -S | grep of1a
 - d. sudo iptables -S --tap
3. On which node is the router found when using an IPv4 floating IP address in DVR?
 - a. Controller node
 - b. Undercloud node
 - c. Compute node
 - d. Storage node
 - e. Power node
4. When deploying the Overcloud with DVR, what is the name of the default environment file that you need to include?
 - a. **ovs-vswitch-dvr.yaml**
 - b. **ovs-dvr.yaml**
 - c. **neutron-ovs-dvr.yaml**
 - d. **architecture-ovs-dvr.yaml**
 - e. **overcloud-dvr.yaml**

Solution

Choose the correct answer(s) to the following questions:

1. Assuming north-south traffic flow without a FIP, which interface processes a set of ip filter and state tracking security group rules?
 - a. **qbr**
 - b. **sg**
 - c. **br-tun**
 - d. **frp**
2. Assuming an instance has the TAP interface **tapf1af5332-02**, what command would you use to view the outbound iptables rules referencing that TAP interface?
 - a. **sudo iptables -S | grep f1a**
 - b. **sudo iptables show tap**
 - c. **sudo iptables -S | grep of1a**
 - d. **sudo iptables -S --tap**
3. On which node is the router found when using an IPv4 floating IP address in DVR?
 - a. Controller node
 - b. Undercloud node
 - c. **Compute node**
 - d. Storage node
 - e. Power node
4. When deploying the Overcloud with DVR, what is the name of the default environment file that you need to include?
 - a. **ovs-vswitch-dvr.yaml**
 - b. **ovs-dvr.yaml**
 - c. **neutron-ovs-dvr.yaml**
 - d. **architecture-ovs-dvr.yaml**
 - e. **overcloud-dvr.yaml**

Managing DVR Routers

Objectives

After completing this section, students should be able to:

- View a DVR namespace
- Compare and contrast a DVR router with a standard router
- View the FIP network namespace in a DVR router

Centralized SNAT Service

OpenStack uses both the original centralized and the newer distributed routing models. In a centralized routing model, project virtual routers, managed by the neutron L3 agent, are deployed to dedicated or clustered Networking Service node. Routed traffic is required to traverse through a Networking Service node. By comparison, the distributed virtual routing model optimizes network traffic by deploying L3 agents and virtual routers on each compute node, wherein traffic is routed between compute nodes without traverse Networking Service nodes. This applies to east-west and north-south traffic but not north-south traffic without FIPs, often referred to as SNAT north-south traffic. SNAT north-south traffic is not distributed to the compute nodes, but is centralized on the controller node. If the SNAT north-south traffic used the distributed routing model, every node that provides the SNAT service would require an address from the external network. To avoid this overhead, the SNAT service remains centralized, similar to legacy routers.

DVR MAC Resolution

When a DVR OVS agent initializes, an RPC message is sent to the Neutron Service requesting the agent's MAC address. If the MAC address is previously recorded in the database, then it is returned. If it does not exist, then a new MAC address is generated and returned to the OVS agent. Compute node outbound traffic will have the agent's source MAC address replaced with the MAC address of the host. For compute node inbound traffic, the remote system's source MAC address is replaced with the gateway interface's MAC address of the destination VM's router.

DVR Issues

The following is a list of known DVR issues to be considered prior to deploying DVR.

- The only backends supported for DVR are the ML2 core plug-in and the Open vSwitch mechanism driver.
- IPv6 traffic is not distributed. Customers that use IPv6 extensively are advised to avoid DVR at this time. All IPv6 routed traffic traverses the centralized controller node.
- DVR is not supported in conjunction with L3 HA. Routers are still scheduled from the controller nodes, but in the event of an L3 agent failure, all routers hosted by that agent also fail. Use of the **allow_automatic_l3agent_failover** feature (controller node **neutron.conf** is recommended so that routers are rescheduled to a different node should a network node fail.

View a DVR Router

The following lists the steps to view a DVR router.

1. From the compute node, the command **ip netns list** displays the network namespace, which is a logical copy of the network stack, with its own routes, firewall rules, and network devices. The qrouter namespace is the distributed router namespace.

```
[demo-admin@compute1 ~]$ sudo ip netns \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284
```

2. The **ip** command lists the interfaces in the qrouter. The qrouter ID and the **ip addr show** command are passed as arguments to the command.

```
[demo-admin@compute1 ~]$ sudo ip -n \
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 addr show
1: lo: LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
15: qr-e2eb2c44-29: BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue
    state UNKNOWN qlen 1000
    link/ether fa:16:3e:86:38:44 brd ff:ff:ff:ff:ff:ff
    inet 192.168.2.1/24 brd 192.168.2.255 scope global qr-e2ea1c33-39
        valid_lft forever preferred_lft forever
        inet6 fe80::f816:3eff:fe86:3844/64 scope link
            valid_lft forever preferred_lft forever
```

3. The **ovs-vsctl list-ports br-int** displays the port on the integration bridge **br-int**, which is also the interface on the qrouter.

```
[demo-admin@compute1 ~]$ sudo ovs-vsctl show br-int
qr-e2eb2c44-29
```

From the two previous commands, it can be determined that br-int forwards the packet to the project network interface qr-e2ea1c33-39 in the qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284 distributed router namespace.

4. The **ip** command is used to list the routing tables that exist in the qrouter namespace. The qrouter ID and the **rule list** option are passed as arguments to the command. If the **main** routing table does not contain the route, then the policy route is applied. The numbers to the left in the output signify the priority processing order. The routing tables are processed in lexical order with the lowest number indicating the highest priority.

```
[demo-admin@compute1 ~]$ sudo ip -n qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284
rule list
0: from all lookup local
32766: from all lookup main
32767: from all lookup default
3232235777: from 192.168.1.1/24 lookup 3232235777
```

5. The **ip** command is used to view the entries of a specific routing tables that exists in the qrouter namespace. For example, to view the **main** routing table, the qrouter ID, the command **route show table**, and the table ID itself are passed as arguments to the command. If the **main** routing table does not contain a required route, then the policy route is applied. The numbers to the left in the output signify the priority processing order. The routing tables are processed in lexical order with the lowest number indicating the highest priority.

```
[demo-admin@compute1 ~]$ sudo ip -n qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284
route table show main
192.168.1.0/24 dev qr-e2eb2c44-29 proto kernel scope link src 192.168.1.1
```

In the output above, the **main** table has a route for packets transiting to the destination network 192.168.1.0/24. When the routing entry is applied, packets from that network are routed to 192.168.1.1, which is the gateway for the project network.

6. The previous command used for viewing the main table is also used to view the policy table. In the previous output, the policy table had an ID of **3232235777**. Similar to the previous command, the qrouter ID, the command **route show table**, and the table ID itself, **3232235777**, are passed as arguments.

```
[demo-admin@compute1 ~]$ sudo ip -n qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284
route table show 3232235777
default via 192.168.1.8 dev qr-e2eb2c44-29
```

The route in the output specifies that packets are forwarded to 192.168.1.8, which is an interface IP address in the SNAT namespace.

- The qrouter namespace resolves the MAC address of **sg**, the project network1 SNAT interface, in the SNAT namespace. The packet is forwarded from **sg** to **br-int**. The **ip netns** command is used to list the SNAT namespace on the controller node.

```
[demo-admin@compute1 ~]$ sudo ip netns
snat-e8d3e5a5-b35e-4a36-a912-755ac9144284
```

The interfaces in the SNAT namespace can be viewed similarly to the qrouter namespace. The **ip** command can be used to list the interfaces in the SNAT namespace. The SNAT ID and the **ip addr show** command are passed as arguments to the command.

```
[demo-admin@compute1 ~]$ sudo ip -n snat-e8d3e5a5-b35e-4a36-a912-755ac9144284 addr
show
...output omitted...
17: sg-9e8e4633-1e: BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue state
UNKNOWN qlen 1000
link/ether fa:16:3e:c2:3a:eb brd ff:ff:ff:ff:ff:ff
inet 192.168.2.9/24 brd 192.168.2.255 scope global sg-9e8e4633-1e
    valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fec2:3aeb/64 scope link
        valid_lft forever preferred_lft forever
19: sg-ed9249fa-cb: BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue state
UNKNOWN qlen 1000
link/ether fa:16:3e:fb:59:fa brd ff:ff:ff:ff:ff:ff
inet 192.168.1.7/24 brd 192.168.1.255 scope global sg-ed9249fa-cb
    valid_lft forever preferred_lft forever
```

```
inet6 fe80::f816:3eff:febf:59fa/64 scope link  
  valid_lft forever preferred_lft forever
```

- The SNAT namespace contains iptables rules which replace the source IP of the instance with the IP address assigned to the external gateway.

```
[demo-admin@compute1 ~]$ sudo ip -n snat-e8d3e5a5-b35e-4a36-a912-755ac9144284  
  iptables
```

View the FIP Namespace

- The distributed router namespace qrouter routes the packet to the floating IP namespace FIP using DVR internal IP addresses. The **fpr** interface resides in the FIP namespace. The **rfp** interface resides in the qrouter namespace.

```
[demo-admin@compute1 ~]$ sudo ip netns  
fip-b4d8228c-2415-4222-80bc-500fda5fa696  
qrouter-e8d3e5a5-b35e-4a36-a912-755ac9144284  
[demo-admin@compute1 ~]$ sudo ip -n \  
fip-b4d8228c-2415-4222-80bc-500fda5fa696 route  
169.254.106.114/31 dev fpr-e8d3e5a5-b proto kernel scope link src 169.254.106.115  
172.25.250.0/24 dev fg-75d40aeb-e2 proto kernel scope link src 172.25.250.101  
172.25.250.113 via 169.254.106.114 dev fpr-e8d3e5a5-b
```

View the FIP Namespace

Watch this video as the instructor demonstrates how to view the FIP namespace and north-south traffic flow with a FIP.

- View the **demo-dvr-router1** distributed router to verify that the **distributed** attribute is set to **True**. The **distributed** attribute is only visible to users with **admin** role because of keystone policy.
- Ensure the instances **demo-server1** and **demo-server2** are running.
- View the ports on the **demo-dvr-router1** router.
- Using the **openstack port list** command with the **device-owner** option, list the ID for **demo-dvr-router1**.
- Using the **openstack port show** command, list the details of the port.
- View port details for the port that is the gateway for project instances, and take note of its IP address.
- Using the **openstack port show** command, list the details of the first port in the list.
- Using the **openstack port show** command, list the details of the second port in the list.
- View the namespaces created on **controller0**.
- Ensure that the floating IP address is attached to the **demo-server1** instance and is available to ICMP.
- List the network namespaces on **compute1**.

12. List the interfaces in the qrouter namespace on **compute1**.
13. List the ports on **br-int**.
14. On **compute0**, list the route tables present in the qrouter namespace.
15. List the **main** route table.
16. List the subnet route tables.
17. List the interfaces in the fip namespace.



References

Further information is available in the chapter on Configuring Distributed Virtual Routing in the *Red Hat OpenStack Platform 10 Networking Guide* at

https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/

Guided Exercise: Managing DVR Routers

In this exercise, you will view a DVR router namespace and SNAT network namespace. You will also verify an instance with no floating IP address using a DVR router to forward packets externally using SNAT.

Outcomes

You should be able to:

- View a DVR router namespace.
- View the SNAT network namespace in a DVR router.
- View the firewall rules in a SNAT network namespace.
- Verify an instance with no floating IP address using DVR router forward packets externally using SNAT.

Before you begin

Log in to **workstation** as **student** using **student** as the password.

Run the **lab dvr-routers setup** command. This script verifies that the overcloud nodes are accessible and running the correct OpenStack services. The script also creates required resources and launches an instance named **finance-server1** on **compute0** node.

```
[student@workstation ~]$ lab dvr-routers setup
```

Steps

1. View the **finance-dvr-router1** distributed router and ensure that the **distributed** attribute is set to **true**.

- 1.1. Source the **/home/student/architect1-finance-rc** credential file.

```
[student@workstation ~]$ source ~/architect1-finance-rc
[student@workstation ~(architect1-finance)]$
```

- 1.2. View the **finance-dvr-router1** distributed router to verify that the **distributed** attribute is set to **True**. The **distributed** attribute is only visible to users with **admin** role because of keystone policy.

```
[student@workstation ~(architect1-finance)]$ openstack router show \
finance-dvr-router1
...output omitted...
| distributed | True |
...output omitted...
| id | c7ab6809-1fee-40e9-8891-eb3816bdeb6f |
...output omitted...
```

2. View the external gateway IP address assigned to **finance-dvr-router1**. View the ports attached to the **finance-dvr-router1** distributed router.

- 2.1. View the ports on the **finance-dvr-router1** router.

```
[student@workstation ~(architect1-finance)]$ openstack port list \
--router finance-dvr-router1 \
-c 'ID' -c 'Fixed IP Addresses' -f json
[
  {
    "Fixed IP Addresses": "ip_address='172.25.250.190', subnet_id='a879632b-b2e4-4824-99fc-cae727451ba8'
      ip_address='2001:db8::190', subnet_id='f5982da9-b7dd-495b-8465-6d486e1cb717',
      "ID": "0db330bc-5eed-4788-a6cf-3fa87073274a"
    },
    {
      "Fixed IP Addresses": "ip_address='192.168.2.9',
      subnet_id='e729f403-2cc2-4842-a176-45c22d708b15'",
      "ID": "75e6b190-3532-471e-be3e-2620be7263ca"
    },
    {
      "Fixed IP Addresses": "ip_address='192.168.2.1',
      subnet_id='e729f403-2cc2-4842-a176-45c22d708b15",
      "ID": "e2ea1c33-398f-4528-878f-d3d5fbe3c6e2"
    }
]
```

2.2. View port details and take note of the external gateway IP address on **finance-dvr-router1**.

```
[student@workstation ~(architect1-finance)]$ openstack port list \
-c ID --router finance-dvr-router1 \
--device-owner network:router_gateway
+-----+
| ID |
+-----+
| 0db330bc-5eed-4788-a6cf-3fa87073274a |
+-----+
[student@workstation ~(architect1-finance)]$ openstack port show \
0db330bc-5eed-4788-a6cf-3fa87073274a
+-----+-----+
| Field | Value |
+-----+-----+
...output omitted...
| device_owner | network:router_gateway |
| extra_dhcp_opts | |
| fixed_ips | ip_address='172.25.250.190', subnet_id='a879632b-b2e4-4824-99fc-cae727451ba8'
| | ip_address='2001:db8::190', subnet_id='f5982da9-b7dd-495b-8465-6d486e1cb717'
| id | 0db330bc-5eed-4788-a6cf-3fa87073274a |
| mac_address | fa:16:3e:0d:7d:a9
...output omitted...
+-----+-----+
```

2.3. View port details for the port that is the gateway for project instances and take note of its IP address.

```
[student@workstation ~(architect1-finance)]$ openstack port list \
-c ID --router finance-dvr-router1 \
```

```
--device-owner network:router_interface_distributed
+-----+
| ID
+-----+
| e2ea1c33-398f-4528-878f-d3d5fbe3c6e2 |
+-----+
[student@workstation ~-(architect1-finance)]$ openstack port show \
e2ea1c33-398f-4528-878f-d3d5fbe3c6e2
+-----+
| Field | Value
+-----+
...output omitted...
| device_owner | network:router_interface_distributed
| extra_dhcp_opts |
| fixed_ips | ip_address='192.168.2.1', subnet_id='e' |
| | 729f403-2cc2-4842-a176-45c22d708b15' |
| id | e2ea1c33-398f-4528-878f-d3d5fbe3c6e2
| mac_address | fa:16:3e:86:38:44
...output omitted...
+-----+
```

2.4. View port details for the SNAT port and take note of its IP address.

```
[student@workstation ~-(architect1-finance)]$ openstack port list \
-c ID --router finance-dvr-router1 \
--device-owner network:router_centralized_snat
+-----+
| ID
+-----+
| 75e6b190-3532-471e-be3e-2620be7263ca |
+-----+
[student@workstation ~-(architect1-finance)]$ openstack port show \
75e6b190-3532-471e-be3e-2620be7263ca
+-----+
| Field | Value
+-----+
...output omitted...
| device_owner | network:router_centralized_snat
| extra_dhcp_opts |
| fixed_ips | ip_address='192.168.2.9', subnet_id='e' |
| | 729f403-2cc2-4842-a176-45c22d708b15' |
| id | 75e6b190-3532-471e-be3e-2620be7263ca
| mac_address | fa:16:3e:21:b5:21
...output omitted...
+-----+
```

- View the **finance-server1** launched by the lab script. The instance is launched using a fixed IP address. No floating IP address is attached to the instance.

```
[student@workstation ~-(architect1-finance)]$ openstack server list \
-c Name -c Status -c Networks
+-----+-----+-----+
| Name | Status | Networks |
+-----+-----+-----+
| finance-server1 | ACTIVE | finance-network2=192.168.2.N |
+-----+-----+-----+
```

- On **compute0**, list network namespaces and the interfaces in the namespace. List the ports on the OVS integration bridge.

- 4.1. From **workstation**, connect using SSH to **compute0**.

```
[student@workstation ~](architect1-finance)]$ ssh root@compute0
[root@compute0 ~]#
```

- 4.2. List the network namespaces that exist on **compute0**.

```
[root@compute0 ~]# ip netns list
qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f
```

The **qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f** namespace is the distributed router namespace. The ID associated with the **qrouter** namespace name matches the project DVR router ID, **finance-dvr-router1**.

- 4.3. List the interfaces in the **qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f** namespace.

```
[root@compute0 ~]# ip -n \
qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
15: qr-e2ea1c33-39: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue
    state UNKNOWN qlen 1000
    link/ether fa:16:3e:86:38:44 brd ff:ff:ff:ff:ff:ff
    inet 192.168.2.1/24 brd 192.168.2.255 scope global qr-e2ea1c33-39
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe86:3844/64 scope link
        valid_lft forever preferred_lft forever
```

The **qr-e2ea1c33-39** interface has **192.168.2.1** as its IP address, and is the gateway for the project network.

- 4.4. List the ports on the **br-int** integration bridge.

```
[root@compute0 ~]# ovs-vsctl list-ports br-int
int-br-ex
patch-tun
qr-e2ea1c33-39
qvocb7f8c98-1c
```

The **br-int** OVS integration bridge forwards packets to the project network interface **qr-e2ea1c33-39** in the **qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f** distributed router namespace.

5. On **compute0** node, list the route tables that exist in the **qrouter** namespace. View the route table rules that allow the packets with an external destination to be forwarded to the SNAT namespace on the **controller0** node.

- 5.1. From **compute0**, list the route tables present in the **qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f** namespace.

```
[root@compute0 ~]# ip -n \
qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f rule list
0: from all lookup local
32766: from all lookup main
32767: from all lookup default
3232236033: from 192.168.2.1/24 lookup 3232236033
```

By default, there are three tables **local**, **main**, and **default**. The table that is added for forwarding packets to the SNAT namespace is **3232236033**. The rules are executed based on the priority list in the first column. The rule with lowest priority is processed first. These rules apply to all packets processed by the router.

- 5.2. View the **main** route table that forwards the packets for east-west communication between instances. The **main** route table's output is the same as that of **ip route show**. The **default** route table is empty.

```
[root@compute0 ~]# ip -n \
qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f route show table main
192.168.2.0/24 dev qr-e2ea1c33-39 proto kernel scope link src 192.168.2.1
```

The packets with a destination address in the **192.168.2.0/24** subnet are routed to **192.168.2.1**, which is the gateway for the project network. The **main** table forwards the packets to the appropriate **qr** interface.

- 5.3. View the **3232236033** route table, which forwards the packets to the SNAT namespace on **controller0**. The **3232236033** is the route table name listed in the previous step. The SNAT interface IP address retrieved from the previous step is **192.168.1.9**.

```
[root@compute0 ~]# ip -n \
qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f route show table 3232236033
default via 192.168.2.9 dev qr-e2ea1c33-39
[root@compute0 ~]# logout
[student@workstation ~(architect1-finance)]$
```

All packets that do not match the routing table rules in the **main** table are processed by this routing table. The packets are forwarded to **192.168.2.9**, which is an interface IP address in the SNAT namespace. The SNAT namespace exists on the **controller0** node.

6. On **controller0** node, list the SNAT network namespace and the interfaces in the namespace. List the firewall rules that allow packets from the project instances to be routed externally.

- 6.1. From **workstation**, connect using SSH to **controller0**.

```
[student@workstation ~(architect1-finance)]$ ssh root@controller0
[root@controller0 ~]#
```

- 6.2. List the network namespaces that exist on **controller0**.

```
[root@controller0 ~]# ip netns list
...output omitted...
```

```
snat-c7ab6809-1fee-40e9-8891-eb3816bdeb6f
...output omitted...
```

The **snat-c7ab6809-1fee-40e9-8891-eb3816bdeb6f** namespace is the distributed router SNAT namespace. The ID associated with the SNAT namespace name is equivalent to the project DVR router ID, **finance-dvr-router1**.

- 6.3. List the interfaces in the **snat-c7ab6809-1fee-40e9-8891-eb3816bdeb6f** namespace.

```
[root@controller0 ~]# ip -n \
snat-c7ab6809-1fee-40e9-8891-eb3816bdeb6f address
...output omitted...
44: sg-75e6b190-35: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 qdisc noqueue
    state UNKNOWN qlen 1000
        link/ether fa:16:3e:21:b5:21 brd ff:ff:ff:ff:ff:ff
        inet 192.168.2.9/24 brd 192.168.2.255 scope global sg-75e6b190-35
            valid_lft forever preferred_lft forever
            inet6 fe80::f816:3eff:fe21:b521/64 scope link
                valid_lft forever preferred_lft forever
47: qg-0db330bc-5e: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1496 qdisc noqueue
    state UNKNOWN qlen 1000
        link/ether fa:16:3e:0d:7d:a9 brd ff:ff:ff:ff:ff:ff
        inet 172.25.250.190/24 brd 172.25.250.255 scope global qg-0db330bc-5e
            valid_lft forever preferred_lft forever
            inet6 2001:db8::190/64 scope global
                valid_lft forever preferred_lft forever
            inet6 fe80::f816:3eff:fe0d:7da9/64 scope link
                valid_lft forever preferred_lft forever
```

The **sg-e2ea1c33-39** interface has **192.168.2.9** as its IP address, and is the SNAT interface that receives the packets forwarded by the **qrouter** namespace from **compute0**. The **qg-0db330bc-5e** interface has **172.25.250.190** as its IP address. The interface is the external gateway and forwards the packets externally after the SNAT firewall rule translates the source IP address from the project instance to the external gateway IP address.

- 6.4. View the IP tables rule that translates the source IP address of the instance to the IP address assigned to the external gateway interface.

```
[root@controller0 ~]# ip netns exec \
snat-c7ab6809-1fee-40e9-8891-eb3816bdeb6f \
iptables -t nat --line-numbers -nL neutron-l3-agent-snat
Chain neutron-l3-agent-snat (1 references)
num target      prot opt source      destination
1  SNAT          all  --  0.0.0.0/0  0.0.0.0/0   to:172.25.250.190
2  SNAT          all  --  0.0.0.0/0  0.0.0.0/0   mark match ! 0x2/0xffff ctstate
DNAT to:172.25.250.190
[root@controller0 ~]# logout
[student@workstation ~(architect1-finance)]$
```

The first firewall rule changes the source IP address of an instance to **172.25.250.190** before forwarding it to the external network using the external bridge using the **qg** interface.

7. Verify that **finance-server1** can access the external network.

Use the **architect1** credentials to get the console URL of the **finance-server1** instance. Verify that the external network is accessible from **finance-server1** by pinging **workstation**.

- 7.1. From **workstation**, obtain the console URL of **finance-server1**.

```
[student@workstation ~](architect1-finance)]$ openstack console url show \
-c url -f value finance-server1
http://172.25.250.50:6080/vnc_auto.html?token=44fd7af5-47a1-4408-
be6e-91126251e3a0
```

- 7.2. Open the console URL using a browser, then log in as user **root** with the password **redhat**.
- 7.3. Verify that **workstation** is reachable using the **ping** command.

```
[root@finance-server1 ~]# ping -c3 workstation
PING workstation (172.25.250.254) 56(84) bytes of data.
64 bytes from workstation.lab.example.com (172.25.250.254): icmp_seq=1 ttl=64
time=0.034 ms
64 bytes from workstation.lab.example.com (172.25.250.254): icmp_seq=2 ttl=64
time=0.046 ms
64 bytes from workstation.lab.example.com (172.25.250.254): icmp_seq=3 ttl=64
time=0.045 ms

--- workstation ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 0.475/0.651/0.740/0.126 ms
```

Cleanup

From **workstation**, run the **lab dvr-routers cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab dvr-routers cleanup
```

This concludes the guided exercise.

Quiz: Implementing Distributed Virtual Routing

Choose the correct answer(s) to the following questions:

1. Assuming a qrouter ID is qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f, which command displays the interfaces for the qrouter namespace?
 - a. `ip -n qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f list`
 - b. `ip -n qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f show int`
 - c. `ip -n qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f list interfaces`
 - d. `ip -n qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f addr show`
2. Which command can be used to verify that an interface in the qrouter namespace is part of the integration bridge?
 - a. `ip -n qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f ip addr show`
 - b. `sudo ovs-vsctl list-ports br-tun`
 - c. `ip -n qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f show int`
 - d. `sudo ovs-vsctl list-ports br-int`
3. When determining traffic flow for the tunneling bridge, which command would you use to view the OpenFlow table 0?
 - a. `sudo ovs-vsctl dump-flows br-tun table=0`
 - b. `sudo ovs-ofctl dump-flows br-tun table0`
 - c. `sudo ovs-ofctl dump-flows br-tun table=0`
 - d. `sudo ovs-vsctl --flows br-tun table=0`
 - e. `sudo ovs-ofctl --flows br-tun table=0`
4. On which node is the router found when using an internal IPv4 IP address in DVR to access externally?
 - a. Compute node
 - b. Undercloud node
 - c. Controller node
 - d. Storage node
 - e. Power node
5. On which node is the router found when using an IPv6 IP address in DVR?
 - a. Compute node
 - b. Undercloud node
 - c. Controller node
 - d. Storage node
 - e. Power node

Solution

Choose the correct answer(s) to the following questions:

1. Assuming a qrouter ID is qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f, which command displays the interfaces for the qrouter namespace?
 - a. `ip -n qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f list`
 - b. `ip -n qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f show int`
 - c. `ip -n qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f list interfaces`
 - d. **`ip -n qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f addr show`**
2. Which command can be used to verify that an interface in the qrouter namespace is part of the integration bridge?
 - a. `ip -n qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f ip addr show`
 - b. `sudo ovs-vsctl list-ports br-tun`
 - c. `ip -n qrouter-c7ab6809-1fee-40e9-8891-eb3816bdeb6f show int`
 - d. **`sudo ovs-vsctl list-ports br-int`**
3. When determining traffic flow for the tunneling bridge, which command would you use to view the OpenFlow table 0?
 - a. `sudo ovs-vsctl dump-flows br-tun table=0`
 - b. `sudo ovs-ofctl dump-flows br-tun table0`
 - c. **`sudo ovs-ofctl dump-flows br-tun table=0`**
 - d. `sudo ovs-vsctl --flows br-tun table=0`
 - e. `sudo ovs-ofctl --flows br-tun table=0`
4. On which node is the router found when using an internal IPv4 IP address in DVR to access externally?
 - a. Compute node
 - b. Undercloud node
 - c. **Controller node**
 - d. Storage node
 - e. Power node
5. On which node is the router found when using an IPv6 IP address in DVR?
 - a. Compute node
 - b. Undercloud node
 - c. **Controller node**
 - d. Storage node
 - e. Power node

Summary

In this chapter, you learned:

- East-west DVR traffic bypasses the network node completely and travels between compute nodes hosting the VMs.
- North-south DVR traffic with FIPs bypasses the network nodes and is routed by the compute nodes hosting the VMs.
- North-south DVR traffic without FIPs are routed through the network node.
- SNAT north-south traffic is not distributed to the compute nodes, but is centralized on the controller node.
- IPv6 traffic is not distributed and customers that use IPv6 extensively are advised to not use DVR at this time. All IPv6 routed traffic traverses the centralized controller node.



CHAPTER 6

TUNING NFV PERFORMANCE

Overview	
Goal	Tune OpenStack Networking performance.
Objectives	<ul style="list-style-type: none">• Describe the network functions virtualization (NFV) architecture and terms.• Tuning cloud applications for performance.
Sections	<ul style="list-style-type: none">• Describing the NFV Architecture (and Quiz)• Tuning Cloud Applications (and Guided Exercise)
Lab	Tuning NFV Performance

Describing the NFV Architecture

Objectives

After completing this section, students should be able to:

- Understand terms used in Network Function Virtualization.
- Understand some of the components in an NFV architecture.
- See the benefits of using OpenStack as part of the solution.

Network Function Virtualization

Network Function Virtualization (NFV) remains a fast evolving set of solutions. In general, dedicated physical network hardware is replaced with virtual network appliances running on resilient, scalable commodity hardware. Current proposals and standards are found on the European Telecommunication Standards Institute (ETSI) website, which is the application layer running on OpenStack, and is beyond the scope of this course. Several NFV drivers and technologies are discussed in this chapter. Here are definitions for commonly used NFV terms:

NFV Terms

Term	Description
NFV	Network Function Virtualization: The move from custom hardware performing network functions to a software solution.
VNF	Virtual Network Function: Software, typically running on a virtual machine, that performs a specific network function, such as routing.
NFVI	Network Function Virtualization Infrastructure: The hardware and software hosting the NFV components. Typically, the NFVI is Red Hat OpenStack Platform.
VIM	Virtual Infrastructure Manager: This component controls the NFVI. When RHOSP is the NFVI, it is also the VIM.
SDN	Software Defined Networking: This technology creates and manages virtual network components through an API, and is complementary to NFV. SDN separates control and data plane components, permitting dynamic changes needed for virtualization environments.
NFV-MANO	NFV Management and Orchestration: Defines the management of an NFV platform, including an NFV Orchestrator, a VNF Manager, and a VIM.
EMS	Element Management System: Responsible for function management of one or more VNFs using the VNF Manager.

NFV Architecture

ETSI defines the NFV architectural framework. The figure below shows the components of the NFV architecture, including NFV Infrastructure (NFVI), Virtual Network Functions (VNF), and Management and Orchestration (MANO). Each of these components is described in further detail in this section.

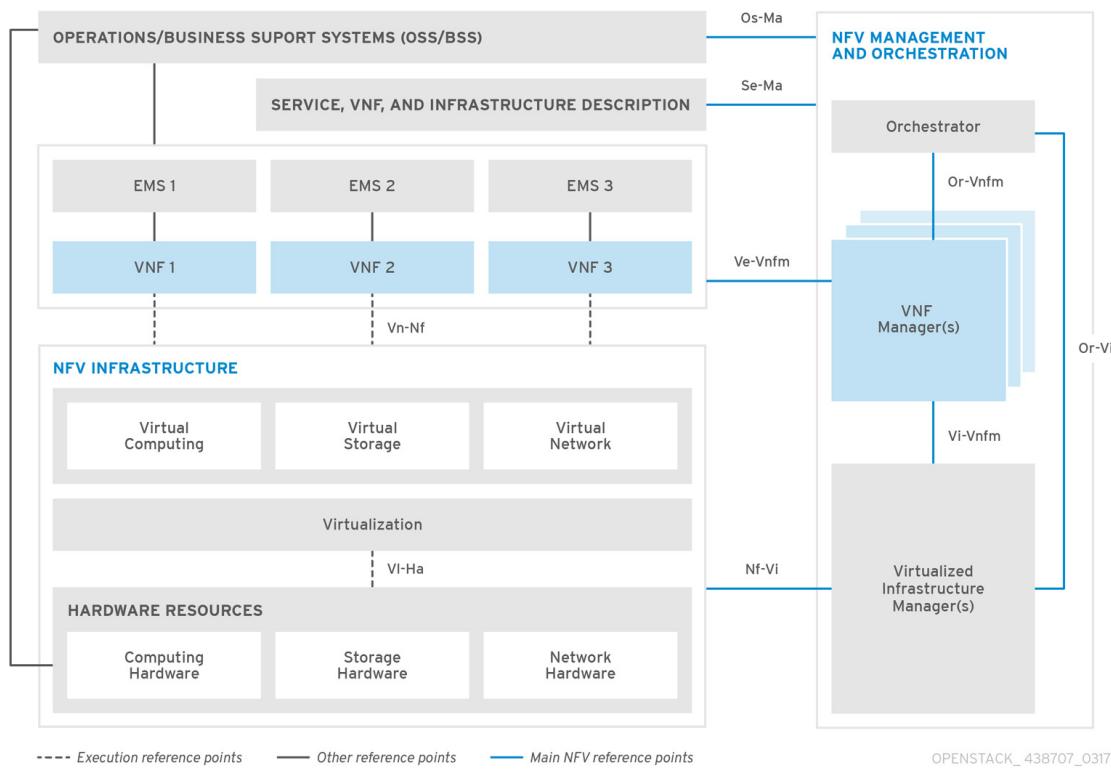


Figure 6.1: NFV architecture

Virtual Network Function (VNF)

A *Virtual Network Function* (VNF) is the fundamental building block in NFV architecture. A VNF represents virtual network elements implemented on Commercial Off The Shelf (COTS) equipment available from vendors such as DELL, HP, IBM, and Cisco. VNF examples include virtualized routers, switches, firewalls, load-balancers, content delivery and optimizers. VNFs help telecommunication providers use low cost generic hardware and scale these network elements on demand. A VNF can map one-to-one to a particular network function that was provided by a legacy network appliance. Multiple VNFs can be chained together to provide an end-to-end *Service* or *Service Chain*. The **Evolved Packet Core** (EPC) is an example of a service used by telcos.

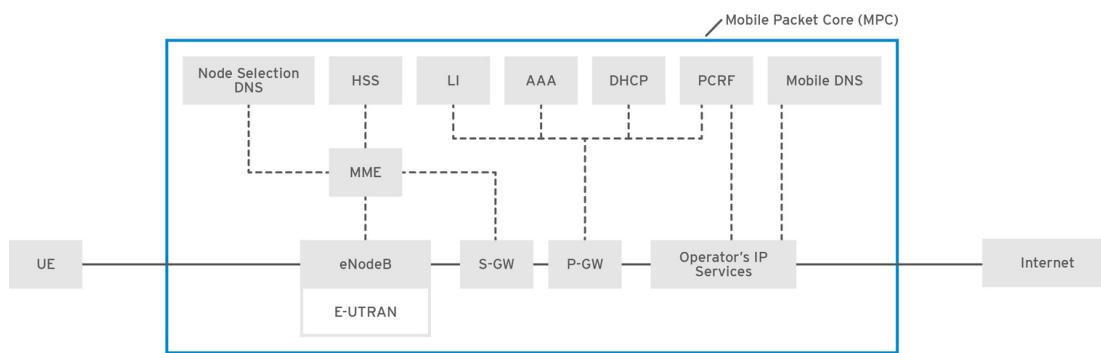


Figure 6.2: Evolved Packet Core (EPC) components

Network Function Infrastructure (NFVI)

The NFVI functional block has three domains: compute, storage, and network. The primary goal of NFV is as an abstraction; to decouple VNFs from the hardware used in these three domains. These domains must offer optimal performance as required by the telecommunication provider using the NFV architecture.

NFVI Components

Virtual Computing, Storage, and Network

This domain includes virtual computing, storage, and network resources providing processing, storage, and networking resources to VNFs using the virtualization layer.

Virtualization

This domain ensures VNFs are decoupled from hardware, and enables deployment of VNFs on different hardware resources based on resource requirements. Virtual machines running in this domain host various network functions or VNFs. These hosting VMs can have direct access to hardware resources for better performance. The VNFs communicate with the **Vn-Nf** interface, or **VNF-NFV** interface drivers provided by Red Hat OpenStack Platform using KVM virtualization.

Physical Hardware

This domain contains hardware resources used by the virtualization layer. The domain exposes all physical hardware resources to virtualization domain using the **Virtualization Layer-Hardware Resources (Vi-Ha)** interface.

Management and Orchestration (MANO)

In NFV architecture, *Management and Orchestration* (MANO) is responsible for managing NFVI infrastructure and deploying VNFs. This functional block consists of the **Virtualized Infrastructure Manager** (VIM), **VNF Manager** (VNFM), and **NFV Orchestrator** (NFVO).

NFV MANO Components

Virtualized Infrastructure Manager (VIM)

The *Virtualized Infrastructure Manager* (VIM) is responsible for controlling and managing the NFVI resources. VIM orchestrates the NFVI resources and maintains the association of the virtualized resources with the physical resources. VIM also evaluates placement policies defined in the VNF descriptor and chooses an appropriate resource pool for VNF placement. VIM exposes the **Nf-Vi** interface, or **NFVI-VIM** interface, to orchestrate NFVI resources. VIM uses the **Or-Vi** interface, or **Orchestrator-VIM** interface of the **Orchestrator**, to provision the NFVI resources.

The VIM uses the **Vi-VNFM** interface, or the **VIM-VNFManager** interface, to interact with **VNF Manager**.

VNF Manager (VNFM)

The *VNF Manager* manages VNF life cycle and sends change notifications. VNFM manages virtual resources associated to the VNF using the **Vi-VNFM** interface presented by VIM, and the **Or-VFNM** interface exposed by the NFV Orchestrator. There are two types of VNF Managers: **Generic VNFM** (GVNFM) and **Specialized VNFM** (SVNFM). The SVNFM are purposely built and are tightly coupled with the VNFs which they manage. The GVNFM are designed to handle some of the basic functionality required by any VNFs which includes load-balancing, failover, monitoring, among others.

NFV Orchestrator (NFVO)

The *NFV Orchestrator* (NFVO) orchestrates resources across multiple VIMs and manage VNF life cycle using VNFM. NFVO offers a logical placement for global resource management, such as compute, storage, and networking resources over multiple VIMs. The ETSI standard divides NFVO into a Network Service Orchestrator (NSO) and a Resource Orchestrator (RO). If multiple VNFs implement a service, the NFVO orchestrates an end to end service using resource management and policy enforcement. The NFVO interacts with VNFM using the **Or-Vnfm** interface to manage VNFs. NFVO also uses the **Or-Vi** interface to orchestrate required virtualized infrastructure resources using VIM.

Operation Support System/Business Support Systems (OSS/BSS)

The OSS/BSS is a propriety software used by an operator and provides support for end-to-end telecommunication services. The *Operation Support System* (OSS) is a software component that handles operations such as network management, service delivery, fault tolerance, configuration management, fulfillment, and assurance.

The *Business Support System* (BSS) is a software component that handles business operations such as customer management, product management, order management, and similar services that focus on customer relationship management. The OSS/BSS functional block interacts with the NFVO using the **Os-Ma** interface.

Service, VNF, and Infrastructure Description

Service, VNF and Infrastructure Description schemas provide information regarding the VNF deployment template. A *VNF Descriptor* (VNFD) is a deployment template which contains information that describes deployment and operational requirements. The VNFD network connectivity information that is used by MANO functional block to establish connectivity between a VNF instance other network functions. The **Ve-Vnfm** interface is used to perform configuration and postprocessing tasks on VNFs by VNFM. The **Se-Ma** interface is used to retrieve information regarding the VNF deployment template.

Element Management System (EMS)

The *Element Management System* (EMS) provides functional management of VNFs. The functional management commonly referred as **FCAPS** includes fault, configuration, accounting, performance, and security management of VNFs. The EMS collaborates with VNF Manager for VNF fault management in particular. One EMS can manage one VNF or multiple VNFs. An EMS is often implemented as another VNF.

Before Network Function Virtualization

Before NFV, telco infrastructure included many thousands of rack-mounted hardware appliances. Dedicated, specialized services, such as firewall, video optimization, or access policies, were implemented in each appliance's firmware and internal storage. Each hardware piece had to be cabled in sequence and configured with unique, proprietary commands.

By replacing each appliance with an identical software-only network function in an OpenStack virtual machine or container, and all physical cabling and configuration with a network service chain definition that sequences each function to the next as a requestable service, the telco industry swapped their largest CapEx and OpEx outlays for instantaneously scalable and replaceable software.

Example Service Function Chain

The following diagram shows Service Function Chain (SFC) classifiers, and Connection Points (CP), which represent the interfaces of the VNFs, connected by Virtual Links.

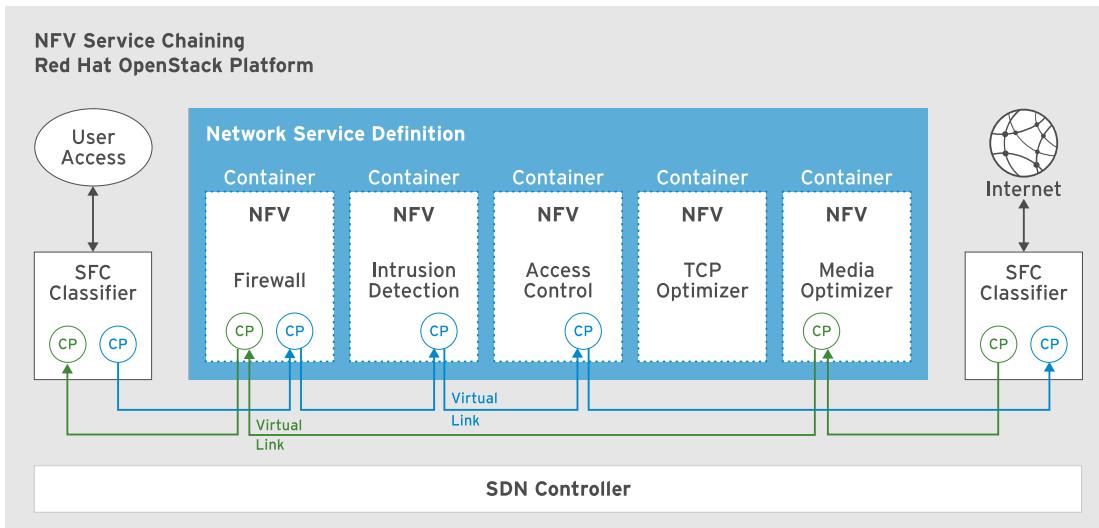


Figure 6.3: NFV service chain

Drivers for Network Function Virtualization

The installation of new network hardware often takes weeks. This time includes delivery of the hardware to site, obtaining data center access, electrical and network cabling under the floor or above the racks, installing the equipment, and so on. The drive to virtualize network devices is due to organizations wanting to reduce time-to-market, improve ROI, and to have the ability to scale network functions easily. Additionally, the life cycle for proprietary network hardware is shrinking due to the pace of development, which is preventing organizations from getting a high return on their investment.

Benefits of Network Function Virtualization

Virtualizing network functions allows for a reduced footprint in the data center, saving on power and cooling. It also allows for multitenancy and multiple versions on a single platform. Trialing new network technologies can be performed safely and cheaply on a single development platform. Adhering to open standards aids integration, and avoids vendor lock-in.

Challenges for Network Function Virtualization

One of the biggest challenges for NFV is achieving hardware equivalent performance for the VNFs. As a physical system, a network appliance may have had hardware tuned to perform a specific role, or even ASICs (Application Specific Integrated Circuits) for specific tasks. VNFs run on commodity servers, and the virtualization layer of the hypervisor and emulated devices reduces performance. The goal is to minimize this performance loss, and get comparative performance to the physical network equipment.

A smooth migration from physical network equipment to virtualized is desirable, but may be difficult when dealing with custom hardware. Vendors migrating their traditional hardware offering to a virtual offering, may be tempted to simply transplant their software into a compatible operating system without taking the destination platform into account.

For a customer, the migration from physical to virtual network equipment may require changes to security procedures. Securing a physical console cable is relatively easy compared to securing access to a virtual appliance. The hypervisor adds another avenue of attack that may not be managed by the network operations team, and often is not.

Red Hat OpenStack Platform in a Network Function Virtualization Architecture

OpenStack provides many of the features that are desirable in an NFV architecture. It is a virtualization platform driven by APIs, providing Software Defined Networking, as well as storage and compute resources. OpenStack fits the role of Virtual Infrastructure Manager, as well as providing the NFV infrastructure. Performance improvements for VNFs are available in the form of dedicated CPU cores, and hugepages for memory. One of the features built in to OpenStack is Enhanced Platform Awareness, which allows workloads with specific hardware requirements to be placed in a location that meets those requirements.



References

ETSI NFV standards and drafts

<http://www.etsi.org/technologies-clusters/technologies/nfv>

Further information is available in the chapter on Understanding Red Hat Network Functions Virtualization (NFV) in the *Red Hat OpenStack Platform 10 Network Functions Virtualization Product Guide* at

https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/

Quiz: Describing the NFV Architecture

Choose the correct answer(s) to the following questions:

1. Which one of the following includes the physical servers in an NFV architecture?
 - a. VNF
 - b. VIM
 - c. NFV-MANO
 - d. NFVI

2. Which three of the following are benefits of NFV? (Choose three.)
 - a. Ease of scaling
 - b. Lower ROI
 - c. Longer time-to-market
 - d. Reduced data center footprint
 - e. ASICs
 - f. Shorter time-to-market

3. Which two of the following roles does OpenStack fill in an NFV architecture? (Choose two.)
 - a. VNF
 - b. VIM
 - c. NFVI
 - d. VNF Manager
 - e. NFV Orchestrator

Solution

Choose the correct answer(s) to the following questions:

1. Which one of the following includes the physical servers in an NFV architecture?
 - a. VNF
 - b. VIM
 - c. NFV-MANO
 - d. **NFVI**
2. Which three of the following are benefits of NFV? (Choose three.)
 - a. **Ease of scaling**
 - b. Lower ROI
 - c. Longer time-to-market
 - d. **Reduced data center footprint**
 - e. ASICs
 - f. **Shorter time-to-market**
3. Which two of the following roles does OpenStack fill in an NFV architecture? (Choose two.)
 - a. VNF
 - b. VIM
 - c. **NFVI**
 - d. VNF Manager
 - e. NFV Orchestrator

Tuning Cloud Applications

Objectives

After completing this section, students should be able to:

- Discuss NUMA architecture.
- Configure CPU pinning on compute hosts.
- Configure hugepages on compute hosts.
- Manage compute scheduler filters.

Improving VNF Performance

A primary goal for virtualized network devices is achieving the same throughput performance as with native physical devices. One common performance tuning task is to use jumbo frames throughout the physical and virtual network. However, MTU must be consistently applied to all physical and virtual network interfaces, requiring extensive coordination between the enterprise data center network and OpenStack engineers. Mismatched MTU become bottlenecks in the networking fabric, causing retransmissions, rejected packets or fragmentation. Although MTU tuning is not very feasible in a completely virtualized classroom, each student's overcloud was properly configured when originally designed for matching MTUs at the hypervisor, Open vSwitch and at the SDN level with DHCP for configuring instance interfaces.

There are two other common tuning configuration items in Red Hat OpenStack Platform that can be practiced in this course: CPU pinning and hugepages.

NUMA Architecture

On or smaller i386 and x86-64 systems, all memory is equally accessible and shared by all CPUs. Access times are the same to any memory address, for any CPU performing a memory operation. CPUs are connected by a shared front-side bus (FSB) to main memory, to coordinate access requests. This memory architecture is called *Uniform Memory Access (UMA)*.

On larger systems, system memory is configured in banks connected directly to particular CPUs or sockets. The local CPU is responsible for controlling that bank, in addition to having exclusive access to that memory. In a *Non-Uniform Memory Access (NUMA)* system, memory access is faster for the local CPU. Memory addresses not found in the local bank generates a memory miss, requiring the data to be copied from the remote bank, which is slower.

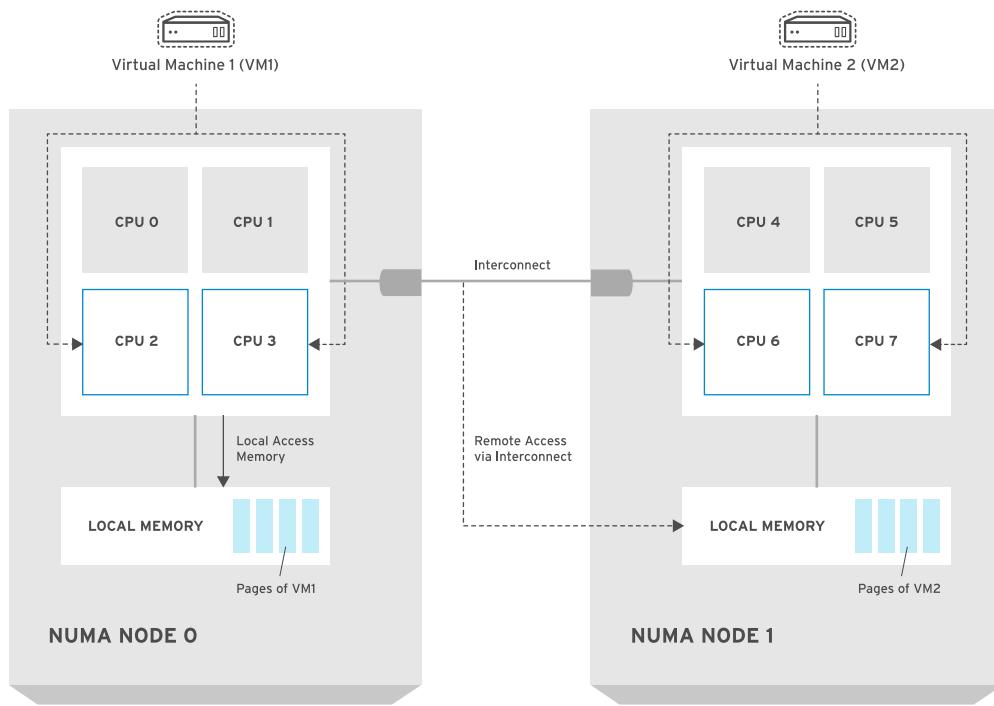


Figure 6.4: Two node NUMA host

Use the **numactl --hardware** command from the *numactl* package to view NUMA memory configuration and the how it is organized into nodes. NUMA nodes can be configured to include any number of CPUs and their corresponding memory banks. The command shows which nodes have which processors, the available memory, and the in-use memory per node. Monitor NUMA misses using the **numastat** command.

```
[root@compute0 ~]# numastat
node0
numa_hit          149501030
numa_miss          0
numa_foreign        0
interleave_hit     35684
local_node         149501030
other_node          0
```

To run a process in a specific NUMA node, use **numactl** to start the process, passing it the allowed CPU nodes or CPUs, memory zones, and the memory allocation policy. The following example starts a process on NUMA node0 with all memory allocated from NUMA nodes 0 and 1.

```
[root@demo ~]# numactl --cpunodebind=0 --membind=0,1 process
```

CPU Pinning

CPU pinning refers to reserving physical cores for specific virtual guest instances. The concept is also referred to as *CPU isolation* or *processor affinity*. The configuration is in two parts; ensuring that virtual guests can only run on dedicated cores, and ensuring that common host processes

do not run on those cores. The term *pinning* refers to the 1-to-1 mapping of a physical core to a guest vCPU.

Configuring Compute Node for CPU Pinning

In OpenStack, CPU pinning establishes a mapping between a virtual CPU and a physical core. With CPU pinning, the instance's virtual CPU processes always run on the same physical core and NUMA node. This improves performance by ensuring that memory access to memory is always local in the NUMA topology.

To enable CPU pinning, the following steps are required on every compute host where relevant processes can be scheduled:

1. View the NUMA topology for the compute node.
2. On compute node, set **vcpu_pin_set** in the **/etc/nova/nova.conf** to a range of physical CPU cores to be reserved for virtual machine processes. The OpenStack Compute service will ensure that virtual machine instances are pinned to these physical CPU cores. In the following example, the compute host reserves two cores in each NUMA node; the 2nd and 3rd core from NUMA node0 and the 6th and 7th core from NUMA node1.

```
vcpu_pin_set=2-3,6-7
```

3. On the compute node, set **reserved_host_memory_mb** to reserve RAM for common host processes and not eligible for the Compute scheduler. In the following example, the value is set to **512 MB**.

```
reserved_host_memory_mb=512
```

4. After editing **/etc/nova/nova.conf**, restart the **openstack-nova-compute** service.

```
[root@compute ~]# systemctl restart openstack-nova-compute
```

5. Host processes should not run on the CPU cores reserved for virtual machine processes. Ensure CPU isolation by adding the **isolcpus** argument to kernel boot arguments. Use the same range of CPU cores reserved for virtual machine processes in **/etc/nova/nova.conf** as the **isolcpus** parameter.

Edit **/etc/default/grub** to add the **isolcpus** parameter. Rebuild the **grub.cfg** file using **grub2-mkconfig**. Reboot the compute host to use the updated configuration.

```
[root@compute ~]# cat /etc/default/grub
GRUB_TIMEOUT=1
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="...isolcpus=2-3,6-7"
GRUB_DISABLE_RECOVERY="true"
[root@compute ~]# grub2-mkconfig > /boot/grub2/grub.cfg
...output omitted...
[root@compute ~]# reboot
```

Configuring CPU Pinning

The following steps outline the process for configuring CPU pinning on a compute node.

1. View the NUMA architecture for the node, then decide how to allocate the cores.
2. Configure the **vcpu_pin_set** option for Nova.
3. Configure the **isolcpus** kernel parameter.
4. Reboot the node.
5. Verify that no host processes are running on the isolated cores.

Host Aggregate

Host aggregates are a method to group hypervisor hosts based on configurable metadata. For example, hosts may be grouped based on hardware features, capabilities or performance characteristics such as CPU pinning. Host aggregates are not visible by users, but are used automatically for instance scheduling. A compute node can be included in multiple host aggregates. To specify the required features requested for an instance deployment, an administrator builds a flavor with the extra specifications to match to available host's aggregate metadata. At deployment, the compute scheduler matches the request declared in the flavor by scheduling the provisioning on a compute host in a matching host aggregate.

To create a host aggregate use the **openstack create aggregate** command.

```
[user@demo ~ (admin-admin)]# openstack create aggregate demo-perf
+-----+-----+
| Field      | Value   |
+-----+-----+
| availability_zone | None |
| created_at | 2018-02-22T10:22:25.878358 |
| deleted    | False  |
| deleted_at | None   |
| id         | 1      |
| name       | demo-perf |
| updated_at | None   |
+-----+-----+
```

To set the metadata for the **demo-perf** aggregate to use CPU pinning. The metadata property will be used to match the flavor's extra specification for provisioning an instance.

```
[user@demo ~ (admin-admin)]# openstack create aggregate demo-perf
```

Set the **aggregate_instance_extra_specs:pinned** flavor extra specification to **true**. All instances created using the **demo-flavor** flavor will be sent to hosts in host aggregates with **pinned=true** in their aggregate metadata.

```
[user@demo ~ (admin-admin)]# openstack flavor set \
--property aggregate_instance_extra_specs:pinned=true demo-perf-flavor
```

Managing Instance CPU Pinning Policies

Host aggregates are useful to ensure that pinned instances run on different hosts than the remaining unpinned instances. Normally, instance vCPU processes are able to be scheduled to any host CPUs as determined by current load and CPU availability. This dynamic allocation is

flexible and makes best use of available CPU resources, even CPU overcommit. However, this also causes scheduling contention which may diminish system performance for specific instances but provide best performance for the system as a whole. However, pinning instances eliminates scheduling contention for resources. Therefore, in addition to creating a flavor specification for CPU pinning, a CPU policy and a thread policy are required to improve performance and scheduling on NUMA.

To configure a flavor to use CPU pinning policy use the following command:

```
[user@demo ~(admin-admin)]# openstack flavor set demo-flavor \
--property hw:cpu_policy=dedicated \
--property hw:cpu_thread=prefer
```

Set **hw:cpu_policy** to **dedicated** to force instance vCPUs to be pinned to a set of host physical CPUs. The default for **hw:cpu_policy** is **shared** which vCPUs to float to any host physical CPUs. Hyper-Threaded processor core consider each thread as a logical vCPU. Setting **hw:cpu_thread** to **prefer** will have thread siblings pinned together. The **hw:cpu_thread_policy** option is only valid if **hw:cpu_policy** is set to **dedicated**.

The following steps outline the process to create host aggregate and to configure flavor extra specification to use CPU pinning defined on compute host:

1. Create the **demo-perf** host aggregate for hosts that will receive pinning requests.

```
[user@demo ~(admin-admin)]# openstack aggregate create demo-perf
+-----+-----+
| Field      | Value   |
+-----+-----+
| availability_zone | None |
| created_at    | 2018-02-22T10:22:25.878358 |
| deleted      | False  |
| deleted_at   | None   |
| id           | 1      |
| name         | demo-perf |
| updated_at   | None   |
+-----+-----+
```

2. Set the properties on the **demo-perf** aggregate, this will be used to match the extra specification of a flavor.

```
[user@demo ~(admin-admin)]# openstack aggregate set \
--property pinned=true demo-perf
```

3. Create the **demo-normal** aggregate for all other hosts that do not use CPU pinning.

```
[user@demo ~(admin-admin)]# openstack aggregate create demo-normal
+-----+-----+
| Field      | Value   |
+-----+-----+
| availability_zone | None |
| created_at    | 2018-02-22T10:30:25.878358 |
| deleted      | False  |
| deleted_at   | None   |
| id           | 1      |
| name         | demo-normal |
| updated_at   | None   |
+-----+-----+
```

```
+-----+-----+
|
```

4. Set the properties on the **demo-normal** aggregate to do not use CPU pinning.

```
[user@demo ~ (admin-admin)]# openstack aggregate set \
--property pinned=false demo-normal
```

5. Create a new flavor for instances requiring NUMA topology and CPU pinning.

```
[user@demo ~ (admin-admin)]# openstack flavor create \
--ram 2048 --disk 10 --vcpus 2 demo-perf-flavor
+-----+-----+
| Field | Value |
+-----+-----+
| OS-FLV-DISABLED:disabled | False |
| OS-FLV-EXT-DATA:ephemeral | 0 |
| disk | 10 |
| id | e2af23bc-286b-4382-8b76-0894e3a070d8 |
| name | demo-perf-flavor |
| os-flavor-access:is_public | True |
| properties | |
| ram | 2048 |
| rxtx_factor | 1.0 |
| swap | |
| vcpus | 2 |
+-----+-----+
```

6. Set the flavor to use the following extra specifications and values:

Flavor extra specification	Values
aggregate_instance_extra_specs:pinned	true
hw:cpu_policy	dedicated
hw:cpu_thread_policy	prefer

```
[user@demo ~ (admin-admin)]# openstack flavor set \
--property aggregate_instance_extra_specs:pinned=true \
--property hw:cpu_policy=dedicated \
--property hw:cpu_thread_policy=prefer \
demo-perf-flavor
```

7. Add compute host to the **demo-perf** and **demo-normal** aggregate.

```
[user@demo ~ (admin-admin)]# openstack aggregate add host \
demo-perf compute0.overcloud.example.com
+-----+-----+
| Field | Value |
+-----+-----+
| availability_zone | None |
| created_at | 2018-02-22T00:18:48.000000 |
| deleted | False |
| deleted_at | None |
| hosts | [u'compute0.overcloud.example.com'] |
| id | 3 |
| metadata | {u'pinned': u'true'} |
| name | base-aggregate |
```

```
[user@demo ~(admin-admin)]# openstack aggregate add host \
demo-normal compute1.overcloud.example.com
+-----+
| Field      | Value   |
+-----+
| availability_zone | None |
| created_at    | 2018-02-22T00:18:48.000000 |
| deleted       | False  |
| deleted_at    | None   |
| hosts         | [u'compute1.overcloud.example.com'] |
| id            | 3      |
| metadata      | {u'pinned': u'false'} |
| name          | base-aggregate |
| updated_at    | None   |
+-----+
```

Hugepages

Modern computer systems use paging to securely and flexibly manage system memory. Memory on a computer system is organized into fixed-size chunks called pages. The size of a page depends on the processor architecture; for i386 and x86_64, the normal page size is 4 KiB. Processes do not address physical memory directly. Instead, each process has a virtual address space. In a Linux system, physical memory is mapped to virtual addresses for process use. The process of looking up a page mapping on a hierarchical page table can be expensive. Therefore, when a mapping from a virtual address to a physical address is looked up in the page table, it is cached in dedicated hardware called the *Translation Look-aside Buffer*, or TLB.

The number of translation lookaside buffer (TLB) entries for a processor is fixed, but with a larger page size, the referenced TLB space for a processor becomes correspondingly larger. Having fewer TLB entries that point to more memory means that a TLB hit is more likely to occur. To assist performance for processes accessing a large but contiguous amount of memory frequently, like databases, the processor architectures support a feature called *hugepages*. This allows memory allocation with a larger fixed page size (2 MiB, and on some systems, 1 GiB) to more efficiently use the TLB. The disadvantage of hugepages is that they force large blocks of contiguous memory to be reserved and allocated to a process as a single unbreakable chunk.

Look in the **/proc/meminfo** file to determine the size of a huge page on a particular system. The file will contain a line similar to:

```
[root@demo ~]# cat /proc/meminfo
...
Hugepagesize: 2048 kB
...
```

To verify the amount of huge memory pages allocated, display the contents of **/proc/meminfo**.

```
[root@demo ~]# grep Huge /proc/meminfo
HugePages_Total: 20
HugePages_Free: 20
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
```

Configuring Hugepages

The following steps outline the process for configuring hugepages on a compute node.

1. Configure the **`transparent_hugepage`**, **`default_hugepagesz`**, **`hugepagesz`**, and **`hugepages`** kernel arguments with appropriate values.
2. Reboot the node.
3. Verify the hugepage configuration in **`/proc/meminfo`**.

Allocating HugePages to an Instance

OpenStack instances do not use hugepages by default, only when such memory is explicitly requested. Hugepages can be a large performance improvement for large critical use cases such as NFV applications. Hugepages are requested explicitly through flavor extra specifications or metadata attached to a Glance image. To use hugepages set the **`hw:mem_page`** flavor extra specification:

```
[user@demo ~(user-demo)]$ openstack flavor set default --property hw:mem_page_size=2048
```

Use the **`hw:mem_page_size`** property to requested a supported hugepage size. In the previous example **2048**, with no unit suffix, defaults to Kilobytes. Setting the **`hw:mem_page_size`** property to **large** requests to only use larger page sizes, either an architecture-dependent 2MB or 1GB, for instance RAM allocation. Similarly, the default of **small** specifies to only use the small page sizes of 4Kb. However, setting **`hw:mem_page_size`** to **any** allows the compute driver implementation to decide what hugepage size to use.

Compute Scheduler Filters

The controller node's **`nova-scheduler`** service determines instance scheduling and provisioning. The **`scheduler_driver`** parameter in the controller node's **`/etc/nova/nova.conf`** sets **`nova.scheduler.filter_scheduler.FilterScheduler`** as the default scheduler.

The **`scheduler_available_filters`** configuration option in the compute node's **`/etc/nova/nova.conf`** provides the Compute service with the filters used by the scheduler. The default setting specifies to use all of filters included in the Compute service:

```
scheduler_available_filters = nova.scheduler.filters.all_filters
```

Custom filters can also be added apart from the default filters.

The **`scheduler_default_filters`** configuration option in **`nova.conf`** defines the list of filters that are used by the **`nova-scheduler`** service.



Note

Although the list below appears to be have multiple lines in this coursebook, remove all hard carriage returns when adding this setting to **`/etc/nova/nova.conf`**.

```
scheduler_default_filters = RetryFilter, AvailabilityZoneFilter, RamFilter,
ComputeFilter, ComputeCapabilitiesFilter, ImagePropertiesFilter,
```

ServerGroupAntiAffinityFilter, ServerGroupAffinityFilter

NUMA Topology Compute Filters

The following is the list of filters used by the compute service to filter hosts based on NUMA topology and extra specifications defined for an instance.

AggregateInstanceExtraSpecsFilter

Matches properties defined in extra specification for an instance type against administrator defined properties on a host aggregate. Works with specifications that are scoped with **aggregate_instance_extra_specs**.

NUMATopologyFilter

Filters hosts based on the NUMA topology that was specified for the instance through the use of flavor extra specifications in combination with the image properties.

References

Further information is available in the chapter on Configure CPU pinning with NUMA in the *Red Hat OpenStack Platform 10 Instances and Images Guide* at

https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/

Guided Exercise: Tuning Cloud Applications

In this exercise, you will:

- Configure CPU isolation and reservation on a compute host.
- Configure static hugepages on a compute host.
- Configure the Nova Scheduler with additional filters.
- Configure aggregates for high performance workloads.
- Create a high performance flavor.

Outcomes

You should be able to customize Red Hat OpenStack Platform to support high performance workloads.

Before you begin

Log in to workstation as **student** using **student** as the password. On **workstation**, run the **lab nfv-tuning setup** command. This script verifies that the overcloud nodes are accessible and that the OpenStack services are running, and sets up the prerequisites for this lab.

```
[student@workstation ~]$ lab nfv-tuning setup
```

Steps

- View the NUMA configuration of **compute0**.

- Log in to **compute0** as **root**.

```
[student@workstation ~]$ ssh root@compute0
[root@compute0 ~]#
```

- View the NUMA configuration of **compute0**.

```
[root@compute0 ~]# numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3
node 0 size: 6143 MB
node 0 free: 4821 MB
node distances:
node 0
 0: 10
```

- On **compute0**, add the **isolcpus** kernel argument to prevent host processes from running on the reserved cores (1-3). Edit the **/etc/nova/nova.conf** file to reserve cores 2 and 3 out of the reserved cores 1, 2, and 3 for virtual guests.

- Isolate the reserved cores 1, 2, and 3 on **compute0**.

Edit the **/etc/default/grub** file to update **GRUB_CMDLINE_LINUX** to add **isolcpus=1-3** as one of the kernel arguments.

```
[root@compute0 ~]# cat /etc/default/grub
GRUB_TIMEOUT=1
GRUB_DISTRIBUTOR=$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="... rhgb quiet isolcpus=1-3"
GRUB_DISABLE_RECOVERY="true"
```

- 2.2. Edit **/etc/nova/nova.conf**, and in the **DEFAULT** section add the **vcpu_pin_set** option with the desired CPU cores, separated by a comma.

```
vcpu_pin_set=2,3
```

3. Configure static hugepages on **compute0** using the arguments and values in the following table. Verify hugepage configuration after rebooting.

Hugepages Kernel Arguments

Argument	Value
default_hugepagesz	2M
hugepagesz	2M
hugepages	2048

- 3.1. Edit the **/etc/default/grub** file to update **GRUB_CMDLINE_LINUX** to set hugepages kernel arguments. Update the **grub.cfg** file using **grub2-mkconfig**.

Reboot **compute0**.

```
[root@compute0 ~]# cat /etc/default/grub
GRUB_TIMEOUT=1
GRUB_DISTRIBUTOR=$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="... default_hugepagesz=2M hugepagesz=2M hugepages=2048"
GRUB_DISABLE_RECOVERY="true"
[root@compute0 ~]# grub2-mkconfig > /boot/grub2/grub.cfg
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-3.10.0-693.1.1.el7.x86_64
Found initrd image: /boot/initramfs-3.10.0-693.1.1.el7.x86_64.img
Found linux image: /boot/vmlinuz-0-rescue-ece3291e67d24665afb67e7774511eb2
Found initrd image: /boot/initramfs-0-
rescue-ece3291e67d24665afb67e7774511eb2.img
done
[root@compute0 ~]# reboot
Connection to compute0 closed by remote host.
Connection to compute0 closed.
[student@workstation ~]$
```

- 3.2. Log back in to **compute0**. Check for the **hugetlbfs** mount, then check for hugepage information in **/proc/meminfo**.

```
[student@workstation ~]$ ssh root@compute0
```

```
[root@compute0 ~]# mount | grep hugetlbfs
hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime,seclabel)
[root@compute0 ~]# grep Huge /proc/meminfo
AnonHugePages:      55296 kB
HugePages_Total:    2048
HugePages_Free:     2048
HugePages_Rsvd:      0
HugePages_Surp:      0
Hugepagesize:       2048 kB
```

- On **compute0**, add the **reserved_host_memory_mb** option to the **DEFAULT** section of **/etc/nova/nova.conf**. Use a value of 512 MB. The 512 MB memory is reserved for the host so that it is always available to host processes and is not used by the compute scheduler. Restart the **openstack-nova-compute** service.

```
reserved_host_memory_mb=512
```

```
[root@compute0 ~]# systemctl restart openstack-nova-compute
[root@compute0 ~]# logout
Connection to compute0 closed.
[student@workstation ~]$
```

- On **controller0**, add the to the following filters in **/etc/nova/nova.conf**:

- NUMATopologyFilter**
- AggregateInstanceExtraSpecsFilter**

Restart the **openstack-nova-scheduler** service.

- Edit **/etc/nova/nova.conf** on **controller0** and add the **NUMATopologyFilter** and **AggregateInstanceExtraSpecsFilter** filters.



Note

Although the list below appears on multiple lines, it must not contain carriage returns when added to **/etc/nova/nova.conf**.

```
[student@workstation ~]$ ssh root@controller0
[root@controller0 ~]# cat /etc/nova/nova.conf
[DEFAULT]
...output omitted...
scheduler_default_filters=...NUMATopologyFilter,AggregateInstanceExtraSpecsFilter
...output omitted...
```

- Restart the **openstack-nova-scheduler** service.

```
[root@controller0 ~]# systemctl restart openstack-nova-scheduler
[root@controller0 ~]# logout
Connection to controller0 closed.
[student@workstation ~]$
```

6. On **workstation** as the **architect1** user in the **finance** project, create an aggregate named **perf-aggregate**. Set **pinned** and **hpgs** properties to **true**.

- 6.1. Source the **/home/student/architect1-finance-rc** credential file.

```
[student@workstation ~]$ source ~/architect1-finance-rc
[student@workstation ~(architect1-finance)]$
```

- 6.2. Create the **perf-aggregate** aggregate.

```
[student@workstation ~(architect1-finance)]$ openstack aggregate create \
perf-aggregate
+-----+-----+
| Field | Value |
+-----+-----+
| availability_zone | None |
| created_at | 2017-11-17T10:22:25.878358 |
| deleted | False |
| deleted_at | None |
| id | 1 |
| name | perf-aggregate |
| updated_at | None |
+-----+-----+
```

- 6.3. Set the **pinned** property to **true**.

```
[student@workstation ~(architect1-finance)]$ openstack aggregate set \
--property pinned=true perf-aggregate
[student@workstation ~(architect1-finance)]$
```

- 6.4. Set the **hpgs** property to **true** for the **perf-aggregate** aggregate.

```
[student@workstation ~(architect1-finance)]$ openstack aggregate set \
--property hpgs=true perf-aggregate
[student@workstation ~(architect1-finance)]$
```

7. Add **compute0** as host to the **perf-aggregate** aggregate.

```
[student@workstation ~(architect1-finance)]$ openstack aggregate add host \
perf-aggregate compute0.overcloud.example.com
+-----+-----+
| Field | Value |
+-----+-----+
| availability_zone | None |
| created_at | 2017-11-17T10:22:25.000000 |
| deleted | False |
| deleted_at | None |
| hosts | [u'compute0.overcloud.example.com'] |
| id | 1 |
| metadata | {u'pinned': u'true', u'hpgs': u'true'} |
| name | perf-aggregate |
| updated_at | None |
+-----+-----+
```

8. Create an aggregate named **base-aggregate**. Set **pinned** and **hpgs** properties to **false**.

8.1. Create the **base-aggregate** aggregate.

```
[student@workstation ~(architect1-finance)]$ openstack aggregate create \
base-aggregate
+-----+-----+
| Field | Value |
+-----+-----+
| availability_zone | None |
| created_at | 2017-11-17T10:23:43.375990 |
| deleted | False |
| deleted_at | None |
| id | 2 |
| name | base-aggregate |
| updated_at | None |
+-----+-----+
```

8.2. Set the **pinned** property to **false**.

```
[student@workstation ~(architect1-finance)]$ openstack aggregate set \
--property pinned=false base-aggregate
[student@workstation ~(architect1-finance)]$
```

8.3. Set the **hpgs** property to **false** for the **base-aggregate** aggregate.

```
[student@workstation ~(architect1-finance)]$ openstack aggregate set \
--property hpgs=false base-aggregate
[student@workstation ~(architect1-finance)]$
```

9. Add **compute1** as host to the **base-aggregate** aggregate.

```
[student@workstation ~(architect1-finance)]$ openstack aggregate add host \
base-aggregate compute1.overcloud.example.com
+-----+-----+
| Field | Value |
+-----+-----+
| availability_zone | None |
| created_at | 2017-11-21T00:18:48.000000 |
| deleted | False |
| deleted_at | None |
| hosts | [u'compute1.overcloud.example.com'] |
| id | 3 |
| metadata | {u'pinned': u'false', u'hpgs': u'false'} |
| name | base-aggregate |
| updated_at | None |
+-----+-----+
```

10. For each existing flavor, set the **aggregate_instance_extra_specs:pinned** property to **false**.

```
[student@workstation ~(architect1-finance)]$ for id in \
$(openstack flavor list -c ID -f value)
do
    openstack flavor set --property aggregate_instance_extra_specs:pinned=false ${id}
done
```

11. For each existing flavor, set the **aggregate_instance_extra_specs:hpgs** property to **false**.

```
[student@workstation ~]$(architect1-finance)]$ for id in \
$(openstack flavor list -c ID -f value)
do
    openstack flavor set --property aggregate_instance_extra_specs:hpgs=false ${id}
done
```

12. Create a flavor named **high-perf** with a 10 GiB disk, 2 GiB of RAM, and 2 vCPUs. Set the following properties and values:

Flavor Properties

Property	Value
hw:cpu_policy	dedicated
hw:cpu_thread_policy	prefer
aggregate_instance_extra_specs:pinned	true

- 12.1. Create the **high-perf** flavor.

```
[student@workstation ~]$(architect1-finance)]$ openstack flavor create \
--ram 2048 \
--disk 10 \
--vcpus 2 high-perf
+-----+-----+
| Field          | Value
+-----+-----+
| OS-FLV-DISABLED:disabled | False
| OS-FLV-EXT-DATA:ephemeral | 0
| disk            | 10
| id              | e2af23bc-286b-4382-8b76-0894e3a060d3
| name            | high-perf
| os-flavor-access:is_public | True
| properties      |
| ram             | 2048
| rxtx_factor     | 1.0
| swap            |
| vcpus           | 2
+-----+-----+
```

- 12.2. Set the **high-perf** flavor properties.

```
[student@workstation ~]$(architect1-finance)]$ openstack flavor set \
--property hw:cpu_policy=dedicated high-perf
[student@workstation ~]$(architect1-finance)]$ openstack flavor set \
--property hw:cpu_thread_policy=prefer high-perf
[student@workstation ~]$(architect1-finance)]$ openstack flavor set \
--property aggregate_instance_extra_specs:pinned=true high-perf
```

13. Set the following properties for the **high-perf** flavor.

Flavor Properties for Hugepages

Property	Value
<code>hw:mem_page_size</code>	2048
<code>aggregate_instance_extra_specs:hpgs</code>	true

```
[student@workstation ~(architect1-finance)]$ openstack flavor set \
--property hw:mem_page_size=2048 high-perf
[student@workstation ~(architect1-finance)]$ openstack flavor set \
--property aggregate_instance_extra_specs:hpgs=true high-perf
[student@workstation ~(architect1-finance)]$
```

14. Create an instance named **finance-server1** in the **finance** project using the following settings:

Instance Properties

Property	Value
<code>image</code>	rhel7
<code>flavor</code>	high-perf
<code>key-name</code>	example-keypair
<code>net-id</code>	finance-network1

```
[student@workstation ~(architect1-finance)]$ openstack server create \
--image rhel7 \
--flavor high-perf \
--key-name example-keypair \
--nic net-id=finance-network1 \
--wait finance-server1
+-----+
| Field | Value |
+-----+
...output omitted...
| OS-EXT-SRV-ATTR:hypervisor_hostname | compute0.overcloud.example.com |
| OS-EXT-SRV-ATTR:instance_name | instance-00000002 |
...output omitted...
```

Note the `instance_name` attribute of **finance-server1**. The instance should be running on **compute0**, as this is the only host matching the filter requirements.

15. Verify that the **finance-server1** instance is using a pinned CPU set and hugepages.

- 15.1. Log in to **compute0** as root, then dump the instance configuration in XML format.

```
[student@workstation ~(architect1-finance)]$ ssh root@compute0
[root@compute0 ~]# virsh dumpxml instance-00000002
...output omitted...
<memory unit='KiB'>2097152<memory>
<currentMemory unit='KiB'>2097152<currentMemory>
<memoryBacking>
<hugepages>
<page size='2048' unit='KiB' nodeset='0'/>
<hugepages>
<memoryBacking>
```

```
<vcpu placement='static'>2<vcpu>
  <cputune>
    <shares>2048<shares>
      <vcpu pin vcpu='0' cpuset='2' />
      <vcpu pin vcpu='1' cpuset='3' />
      <emulatorpin cpuset='2-3' />
    <cputune>
    ...output omitted...
[root@compute0 ~]#
```

Note the reference to hugepages and the page size of 2048 KiB. Also make note of the cpu pinning of vcpu0 to core 2 in the cpuset, and vcpu1 to core 3 in the cpuset.

15.2.Determine the current hugepage availability in **/proc/meminfo**.

```
[root@compute0 ~]# grep Huge /proc/meminfo
AnonHugePages:      55296 kB
HugePages_Total:    2048
HugePages_Free:     1024
HugePages_Rsvd:     0
HugePages_Surp:     0
Hugepagesize:       2048 kB
[root@compute0 ~]# logout
Connection to compute0 closed.
[student@workstation ~ (architect1-finance)]$
```

You have now confirmed that the VM with 2 GiB of RAM using the **high-perf** flavor has used 2 of the 3 free hugepages.

Cleanup

From **workstation**, run the **lab nfv-tuning cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab nfv-tuning cleanup
```

This concludes the guided exercise.

Lab: Tuning NFV Performance

In this lab, you will:

- Configure CPU isolation and reservation on a compute host.
- Configure static hugepages on a compute host.
- Configure the Nova Scheduler with additional filters.
- Configure aggregates for high performance workloads.
- Create a high performance flavor.

Outcomes

You should be able to customize Red Hat OpenStack Platform to support high performance workloads.

Before you begin

Log in to workstation as **student** using **student** as the password. On **workstation**, run the **lab nfv-performance-tuning setup** command. This script verifies that the overcloud nodes are accessible and that the OpenStack services are running, and sets up the prerequisites for this lab.

```
[student@workstation ~]$ lab nfv-performance-tuning setup
```

Steps

- View the NUMA configuration of **compute0**.
- On **compute0**, edit the **/etc/nova/nova.conf** file to reserve cores 2 and 3 for virtual guests. Add the **isolcpus** kernel argument to prevent host processes from running on the reserved cores.
- Configure static hugepages on **compute0** using the arguments and values in the following table. Verify hugepage configuration after rebooting.

Hugepages Kernel Arguments

Argument	Value
default_hugepagesz	2M
hugepagesz	2M
hugepages	2048

- On **compute0**, add the **reserved_host_memory_mb** option to the **DEFAULT** section of **/etc/nova/nova.conf**. Use a value of 512. Restart the **openstack-nova-scheduler** service.
- On **controller0**, add the **scheduler_default_filters** option to the **DEFAULT** section of **/etc/nova/nova.conf**. Use the following list of filters:
 - NUMATopologyFilter

- AggregateInstanceExtraSpecsFilter
6. On **workstation**, as the **admin** user create an aggregate named **perf-aggregate**. Set the **pinned** and **hpgs** properties to **true**.
Add **compute0** as a host in **perf-aggregate**.
7. Create an aggregate named **base-aggregate**. Set the **pinned** and **hpgs** properties to **false**.
Add **compute1** as a host in **base-aggregate**.
8. For each existing flavor, set the **aggregate_instance_extra_specs:pinned** property and **aggregate_instance_extra_specs:hpgs** to **false**.
9. Create a flavor named **high-perf** with a 10 GiB disk, 2 GiB of RAM, and 2 vCPUs. Set the following properties and values:

Flavor Properties

Property	Value
hw:cpu_policy	dedicated
hw:cpu_thread_policy	prefer
aggregate_instance_extra_specs:pinned	true

10. Set the following properties for the **high-perf** flavor:

Flavor Properties for Hugepages

Property	Value
hw:mem_page_size	2048
aggregate_instance_extra_specs:hpgs	true

11. As **operator1**, create an instance named **production-server1** in the **production** project using the following settings:

Instance Properties

Property	Value
image	rhel7
flavor	high-perf
key-name	example-keypair
net-id	production-network1

12. Verify that the **production-server1** instance is using a pinned CPU set and hugepages.

Evaluation

From **workstation**, run the **lab nfv-performance-tuning grade** command to confirm the success of this exercise. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab nfv-performance-tuning grade
```

Cleanup

From **workstation**, run the **lab nfv-performance-tuning cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab nfv-performance-tuning cleanup
```

This concludes the lab.

Solution

In this lab, you will:

- Configure CPU isolation and reservation on a compute host.
- Configure static hugepages on a compute host.
- Configure the Nova Scheduler with additional filters.
- Configure aggregates for high performance workloads.
- Create a high performance flavor.

Outcomes

You should be able to customize Red Hat OpenStack Platform to support high performance workloads.

Before you begin

Log in to workstation as **student** using **student** as the password. On **workstation**, run the **lab nfv-performance-tuning setup** command. This script verifies that the overcloud nodes are accessible and that the OpenStack services are running, and sets up the prerequisites for this lab.

```
[student@workstation ~]$ lab nfv-performance-tuning setup
```

Steps

- View the NUMA configuration of **compute0**.

- Log in to **compute0** as **root**.

```
[student@workstation ~]$ ssh root@compute0
[root@compute0 ~]#
```

- View the NUMA configuration of **compute0**.

```
[root@compute0 ~]# numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3
node 0 size: 6143 MB
node 0 free: 4821 MB
node distances:
node 0
 0: 10
```

- On **compute0**, edit the **/etc/nova/nova.conf** file to reserve cores 2 and 3 for virtual guests. Add the **isolcpus** kernel argument to prevent host processes from running on the reserved cores.
 - Edit **/etc/nova/nova.conf**, and in the **DEFAULT** section, add the **vcpu_pin_set** option with the desired CPU cores, separated by a comma.

```
vcpu_pin_set=2,3
```

- 2.2. Add the **isolcpus** kernel argument, then update the bootloader. Isolate all cores except 0, even though Nova is only configured to use 2 and 3. You will reboot **compute0** later to activate this change.

```
[root@compute0 ~]# cat /etc/default/grub
GRUB_TIMEOUT=1
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="... rhgb quiet isolcpus=1-3"
GRUB_DISABLE_RECOVERY="true"
```

3. Configure static hugepages on **compute0** using the arguments and values in the following table. Verify hugepage configuration after rebooting.

Hugepages Kernel Arguments

Argument	Value
default_hugepagesz	2M
hugepagesz	2M
hugepages	2048

- 3.1. Edit the **/etc/default/grub** file to update **GRUB_CMDLINE_LINUX** to set hugepages kernel arguments. Update the **grub.cfg** file using **grub2-mkconfig**.

Reboot **compute0**.

```
[student@workstation ~(architect1-finance)]$ cat /etc/default/grub
GRUB_TIMEOUT=1
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="...default_hugepagesz=2M hugepagesz=2M hugepages=2048"
GRUB_DISABLE_RECOVERY="true"
[root@compute0 ~]# grub2-mkconfig > /boot/grub2/grub.cfg
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-3.10.0-693.1.1.el7.x86_64
Found initrd image: /boot/initramfs-3.10.0-693.1.1.el7.x86_64.img
Found linux image: /boot/vmlinuz-0-rescue-ece3291e67d24665afb67e7774511eb2
Found initrd image: /boot/initramfs-0-
rescue-ece3291e67d24665afb67e7774511eb2.img
done
[root@compute0 ~]# reboot
Connection to compute0 closed by remote host.
Connection to compute0 closed.
[student@workstation ~]$
```

- 3.2. Log back in to **compute0**. Check for the **hugetlbfs** mount, then check for hugepage information in **/proc/meminfo**.

```
[student@workstation ~]$ ssh root@compute0
[root@compute0 ~]# mount | grep hugetlbfs
hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime,seclabel)
```

```
[root@compute0 ~]# grep Huge /proc/meminfo
AnonHugePages:      55296 kB
HugePages_Total:    2048
HugePages_Free:     2048
HugePages_Rsvd:      0
HugePages_Surp:      0
Hugepagesize:       2048 kB
[root@compute0 ~]# logout
Connection to compute0 closed.
[student@workstation ~]$
```

4. On **compute0**, add the **reserved_host_memory_mb** option to the **DEFAULT** section of **/etc/nova/nova.conf**. Use a value of 512. Restart the **openstack-nova-scheduler** service.

```
reserved_host_memory_mb=512
```

```
[root@compute0 ~]# systemctl restart openstack-nova-compute
[root@compute0 ~]# logout
Connection to compute0 closed.
[student@workstation ~]$
```

5. On **controller0**, add the **scheduler_default_filters** option to the **DEFAULT** section of **/etc/nova/nova.conf**. Use the following list of filters:

- NUMATopologyFilter
- AggregateInstanceExtraSpecsFilter

Restart the **openstack-nova-scheduler** service.

```
[student@workstation ~]$ ssh root@controller0
[root@controller0 ~]# cat /etc/nova/nova.conf
[DEFAULT]
...output omitted...
scheduler_default_filters=...NUMATopologyFilter,AggregateInstanceExtraSpecsFilter
...output omitted...
[root@controller0 ~]# systemctl restart openstack-nova-scheduler
[root@controller0 ~]# logout
Connection to controller0 closed.
[student@workstation ~]$
```

6. On **workstation**, as the **admin** user create an aggregate named **perf-aggregate**. Set the **pinned** and **hpgs** properties to **true**.

Add **compute0** as a host in **perf-aggregate**.

- 6.1. Source the **/home/student/admin-rc** credential file.

```
[student@workstation ~]$ source ~/admin-rc
[student@workstation ~(admin-admin)]$
```

- 6.2. Create the **perf-aggregate** aggregate.

```
[student@workstation ~(admin-admin)]$ openstack aggregate create \
```

perf-aggregate	
Field	Value
availability_zone	None
created_at	2017-11-17T10:22:25.878358
deleted	False
deleted_at	None
id	1
name	perf-aggregate
updated_at	None

- 6.3. Set the **pinned** property to **true**.

```
[student@workstation ~(admin-admin)]$ openstack aggregate set \
--property pinned=true perf-aggregate
```

- 6.4. Set the **hpgs** property to **true** for the **perf-aggregate** aggregate.

```
[student@workstation ~(admin-admin)]$ openstack aggregate set \
--property hpgs=true perf-aggregate
```

- 6.5. Add **compute0** to the **perf-aggregate** aggregate.

[student@workstation ~(admin-admin)]\$ openstack aggregate add host \ perf-aggregate compute0.overcloud.example.com	
+-----+-----+	
Field	Value
+-----+-----+	
availability_zone	None
created_at	2017-11-17T10:22:25.000000
deleted	False
deleted_at	None
hosts	[u'compute0.overcloud.example.com']
id	1
metadata	{u'pinned': u'true', u'hpgs': u'true'}
name	perf-aggregate
updated_at	None
+-----+-----+	

7. Create an aggregate named **base-aggregate**. Set the **pinned** and **hpgs** properties to **false**.

Add **compute1** as a host in **base-aggregate**.

- 7.1. Create the **base-aggregate** aggregate.

```
[student@workstation ~(admin-admin)]$ openstack aggregate create \
base-aggregate
+-----+-----+
| Field      | Value   |
+-----+-----+
| availability_zone | None |
| created_at    | 2017-11-17T10:23:43.375990 |
| deleted      | False  |
| deleted_at   | None   |
```

id	2	
name	base-aggregate	
updated_at	None	

- 7.2. Set the **pinned** property to **false**.

```
[student@workstation ~(admin-admin)]$ openstack aggregate set \
--property pinned=false base-aggregate
```

- 7.3. Set the **hpgs** property to **false** for the **base-aggregate** aggregate.

```
[student@workstation ~(admin-admin)]$ openstack aggregate set \
--property hpgs=false base-aggregate
```

- 7.4. Add **compute1** to the **base-aggregate** aggregate.

```
[student@workstation ~(admin-admin)]$ openstack aggregate add host \
base-aggregate compute1.overcloud.example.com
+-----+-----+
| Field | Value |
+-----+-----+
| availability_zone | None |
| created_at | 2017-11-21T00:18:48.000000 |
| deleted | False |
| deleted_at | None |
| hosts | [u'compute1.overcloud.example.com'] |
| id | 3 |
| metadata | {u'pinned': u'false', u'hpgs': u'false'} |
| name | base-aggregate |
| updated_at | None |
+-----+-----+
```

8. For each existing flavor, set the **aggregate_instance_extra_specs:pinned** property and **aggregate_instance_extra_specs:hpgs** to **false**.

```
[student@workstation ~(admin-admin)]$ for id in \
$(openstack flavor list -c ID -f value)
do
    openstack flavor set --property aggregate_instance_extra_specs:pinned=false ${id}
    openstack flavor set --property aggregate_instance_extra_specs:hpgs=false ${id}
done
```

9. Create a flavor named **high-perf** with a 10 GiB disk, 2 GiB of RAM, and 2 vCPUs. Set the following properties and values:

Flavor Properties

Property	Value
hw:cpu_policy	dedicated
hw:cpu_thread_policy	prefer
aggregate_instance_extra_specs:pinned	true

- 9.1. Create the **high-perf** flavor.

```
[student@workstation ~ (admin-admin)]$ openstack flavor create \
--ram 2048 \
--disk 10 \
--vcpus 2 high-perf
+-----+-----+
| Field | Value |
+-----+-----+
| OS-FLV-DISABLED:disabled | False |
| OS-FLV-EXT-DATA:ephemeral | 0 |
| disk | 10 |
| id | e2af23bc-286b-4382-8b76-0894e3a060d3 |
| name | high-perf |
| os-flavor-access:is_public | True |
| properties | |
| ram | 2048 |
| rxtx_factor | 1.0 |
| swap | |
| vcpus | 2 |
+-----+-----+
```

9.2. Set the **high-perf** flavor properties.

```
[student@workstation ~ (admin-admin)]$ openstack flavor set \
--property hw:cpu_policy=dedicated high-perf
[student@workstation ~ (admin-admin)]$ openstack flavor set \
--property hw:cpu_thread_policy=prefer high-perf
[student@workstation ~ (admin-admin)]$ openstack flavor set \
--property aggregate_instance_extra_specs:pinned=true high-perf
```

10. Set the following properties for the **high-perf** flavor:

Flavor Properties for Hugepages

Property	Value
hw:mem_page_size	2048
aggregate_instance_extra_specs:hpgs	true

```
[student@workstation ~ (admin-admin)]$ openstack flavor set \
--property hw:mem_page_size=2048 high-perf
[student@workstation ~ (admin-admin)]$ openstack flavor set \
--property aggregate_instance_extra_specs:hpgs=true high-perf
```

11. As **operator1**, create an instance named **production-server1** in the **production** project using the following settings:

Instance Properties

Property	Value
image	rhel7
flavor	high-perf
key-name	example-keypair
net-id	production-network1

- 11.1. Source the **/home/student/operator1-production-rc** credential file.

```
[student@workstation ~ (admin-admin)]$ source ~/operator1-production-rc  
[student@workstation ~ (operator1-production)]$
```

- 11.2. Create the **production-server1** instance.

```
[student@workstation ~ (operator1-production)]$ openstack server create \  
--image rhel7 \  
--flavor high-perf \  
--key-name example-keypair \  
--nic net-id=production-network1 \  
--wait production-server1  
...output omitted...
```

12. Verify that the **production-server1** instance is using a pinned CPU set and hugepages.

- 12.1. As the **architect** user, determine the `instance_name` attribute of **production-server1**. The instance should be running on **compute0**, as this is the only host matching the filter requirements.

```
[student@workstation ~ (operator1-production)]$ source ~/architect1-production-rc  
[student@workstation ~ (architect1-production)]$ openstack server show \  
production-server1  
...output omitted...  
| OS-EXT-SRV-ATTR:hypervisor_hostname | compute0.overcloud.example.com |  
| OS-EXT-SRV-ATTR:instance_name | instance-00000002 |  
...output omitted...
```

- 12.2. Log in to **compute0** as root, then dump the instance configuration in XML format.

```
[student@workstation ~ (architect1-production)]$ ssh root@compute0  
[root@compute0 ~]# virsh list  
Id   Name           State  
----  
 1   instance-00000002  running  
[root@compute0 ~]# virsh dumpxml instance-00000002  
...output omitted...  
<memory unit='KiB'>2097152<memory>  
<currentMemory unit='KiB'>2097152<currentMemory>  
<memoryBacking>  
<hugepages>  
<page size='2048' unit='KiB' nodeset='0' />  
<hugepages>  
<memoryBacking>  
<vcpu placement='static'>2</vcpu>  
<cpuctune>  
<shares>2048</shares>  
<vcpu pin vcpu='0' cpuset='2' />  
<vcpu pin vcpu='1' cpuset='3' />  
<emulatorpin cpuset='2-3' />  
<cpuctune>  
...output omitted...
```

- 12.3. Determine the current hugepage availability in **/proc/meminfo**:

```
[root@compute0 ~]# grep Huge /proc/meminfo
AnonHugePages:      55296 kB
HugePages_Total:    2048
HugePages_Free:    1024
HugePages_Rsvd:     0
HugePages_Surp:     0
Hugepagesize:       2048 kB
[root@compute0 ~]# logout
Connection to compute0 closed.
[student@workstation ~(architect1-production)]$
```

You have now confirmed that the VM with 2 GiB of RAM using the **high-perf** flavor has used 2 of the 3 free hugepages.

Evaluation

From **workstation**, run the **lab nfv-performance-tuning grade** command to confirm the success of this exercise. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab nfv-performance-tuning grade
```

Cleanup

From **workstation**, run the **lab nfv-performance-tuning cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab nfv-performance-tuning cleanup
```

This concludes the lab.

Summary

In this chapter, you learned:

- Migration from physical network infrastructure to NFV has many considerations, including performance and security.
- Red Hat OpenStack Platform can provide the NFVI and VIM roles in an NFV architecture.
- Tuning an OpenStack environment for VNF performance may include MTU tuning, CPU pinning, and hugepages configuration.



CHAPTER 7

IMPLEMENTING NFV DATAPATHS

Overview	
Goal	Implement network functions virtualization (NFV) datapaths.
Objectives	<ul style="list-style-type: none">Describe single-root I/O virtualization configuration.Configure Quality of Service (QoS) to adjust network performance.Deploy Open vSwitch Data Plane Development Kit (OVS-DPDK) to improve network performance between instances.
Sections	<ul style="list-style-type: none">Describing SR-IOV (and Quiz)Configuring QoS (and Guided Exercise)Deploying OVS-DPDK (and Guided Exercise)
Quiz	Implementing NFV Datapaths

Describing SR-IOV

Objectives

After completing this section, students should be able to:

- Describe the Overcloud configuration for SR-IOV.
- Explain SR-IOV architecture.
- Explain the benefits of SR-IOV.
- List the hardware requirements for SR-IOV.

What is SR-IOV?

Single-Root I/O Virtualization (SR-IOV) is a feature available in some PCIe network interface cards (NIC), that allows the physical NIC to be presented as several virtual NICs. This means that a single physical NIC can be shared between multiple virtual machines, removing the usual 1:1 mapping limitation when passing PCI devices through from the hypervisor to the virtual guests. In SR-IOV terms, a physical NIC is called a Physical Function (PF), and a virtual NIC is called a Virtual Function (VF).

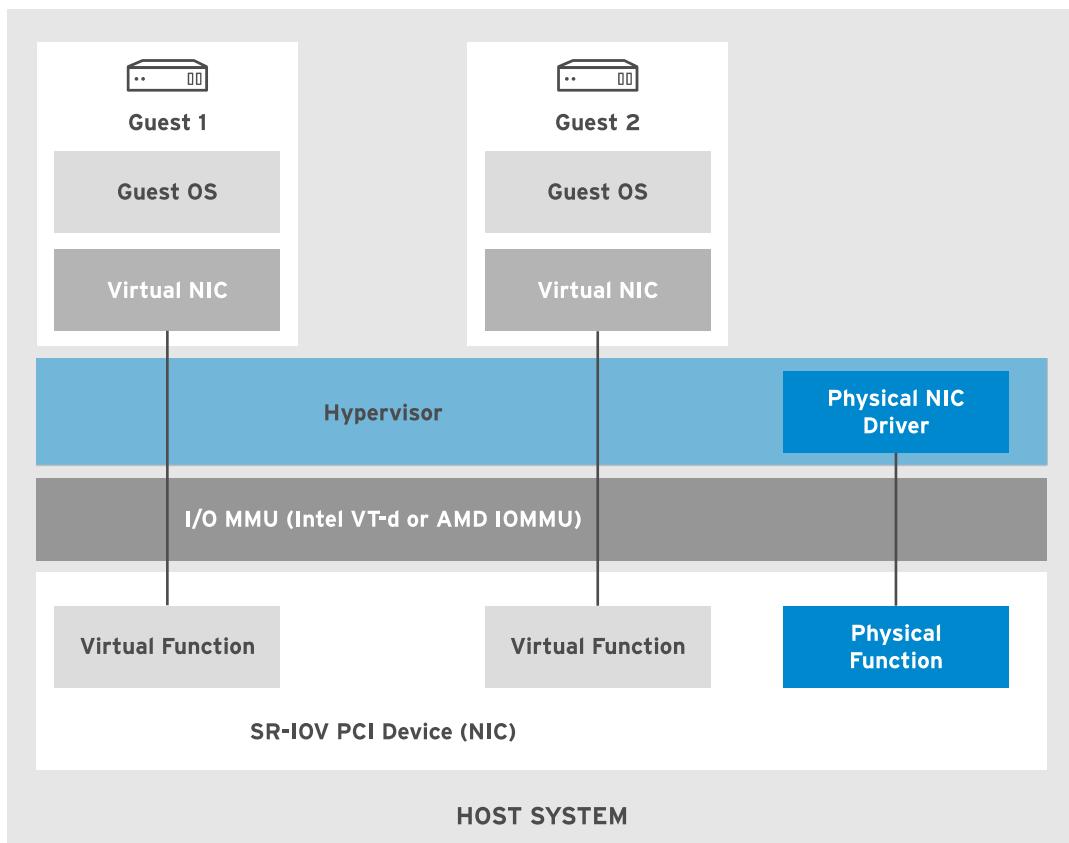


Figure 7.1: SR-IOV architecture

PCI Passthrough

Prior to SR-IOV, it was possible to provide a guest with direct access to a PCI device using PCI passthrough. This provided better performance but meant that the device was dedicated to the instance, and could not be utilized if the instance was idle. With SR-IOV, this limitation was overcome, allowing direct access to a NIC that is shared between instances. You can still dedicate the whole NIC to a specific instance, by passing through the Physical Function instead of a Virtual Function. You can also use PCI passthrough to provide direct access to other types of PCI devices, or if you do not want to enable SR-IOV on your NIC.

SR-IOV Use Cases

Described simply, virtual NICs are processes emulating a physical NIC, so they consume CPU and memory resources on the hypervisor. Using SR-IOV removes the need for these resources, offloading packet processing to the NIC hardware, and freeing up hypervisor resources for other tasks.

SR-IOV provides direct access to the NIC, avoiding the layer of virtualization in the hypervisor, as well as the virtual switch. This increases performance compared to instances with virtual NICs. High performance networking is often required in an NFV implementation, where the VNF is replacing dedicated, optimized network hardware.

Supported Hardware

Red Hat has tested several original Intel network cards with SR-IOV support, they included the following chip sets:

- 82598/82599
- X520/X540/X550
- X710/XL710/X722

Red Hat has also tested 10 Gb SR-IOV cards from Mellanox and Qlogic. If your NIC is not listed here, please check the Red Hat Ecosystem website for the latest list of supported hardware.

SR-IOV Configuration

To configure SR-IOV during overcloud deployment, add composable roles through **roles-data.yaml** to the Heat templates on the Undercloud node. You also need to add a flavor with appropriate settings, and configure kernel arguments, in **network-environment.yaml**. The **neutron-sriov.yaml** file contains the SR-IOV agent, drivers, parameters, and devices needed to configure SR-IOV.

If your OpenStack environment does not use Heat for deployment, you can configure SR-IOV manually. Manual configuration is documented but is not a simple process. Using Heat templates is the method recommended by Red Hat.



References

Further information is available in the chapter on Configure SR-IOV Support For Virtual Networking in the *Red Hat OpenStack Platform 7 Network Functions Virtualization Configuration Guide* at

 | https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/

Further information about hardware compatibility is available in the *Red Hat Ecosystem* site at

 | <https://access.redhat.com/ecosystem/>

Quiz: Describing SR-IOV

Choose the correct answer(s) to the following questions:

1. Which Red Hat website should you visit to check whether a NIC is supported?
 - a. Red Hat Vendor Support
 - b. Red Hat NIC NACs
 - c. Red Hat Hardware Central
 - d. Red Hat Ecosystem

2. You can pass through a virtual or physical function to an instance. (true or false)
 - a. true
 - b. false

3. Which two TripleO heat template files (using their default file names) are updated to include parameters for kernel arguments, SR-IOV driver, and PCI passthrough devices during overcloud deployment? (Choose two.)
 - a. netenv.yaml
 - b. neutron-ovs-sriov.yaml
 - c. controller.yaml
 - d. neutron-sriov.yaml
 - e. network-environment.yaml

Solution

Choose the correct answer(s) to the following questions:

1. Which Red Hat website should you visit to check whether a NIC is supported?
 - a. Red Hat Vendor Support
 - b. Red Hat NIC NACs
 - c. Red Hat Hardware Central
 - d. **Red Hat Ecosystem**
2. You can pass through a virtual or physical function to an instance. (true or false)
 - a. **true**
 - b. **false**
3. Which two TripleO heat template files (using their default file names) are updated to include parameters for kernel arguments, SR-IOV driver, and PCI passthrough devices during overcloud deployment? (Choose two.)
 - a. **netenv.yaml**
 - b. **neutron-ovs-sriov.yaml**
 - c. **controller.yaml**
 - d. **neutron-sriov.yaml**
 - e. **network-environment.yaml**

Configuring QoS and DSCP

Objectives

After completing this section, students should be able to:

- Describe QoS.
- Explain the benefits and limitations of QoS.
- Configure QoS limits.
- Describe DSCP.
- Configure DSCP.

Introduction to Quality of Service

Cloud environments with self-service catalogs suffer from over-consummation of resources such as CPU, networking, or storage bandwidth. Support for network *Quality of Service*, or **QoS**, policies were introduced in Red Hat OpenStack 10. These policies apply rate limits to outgoing traffic for instances and networks. Any traffic exceeding the limit is dropped. The QoS service is designed as a service plugin and is therefore decoupled from Neutron. It extends resources by using the ML2 extension driver.

The ML2 plugin relies on an ML2 QoS extension driver which notifies the agent about a change in the configuration of a specific port. The ML2 agent then passes the configuration onto all enabled extensions, including QoS. The QoS extension then asks the port for its policy and rules. The QoS extension applies the rules using the QoS driver.

When a QoS policy is updated, including a rule update, the server pushes the new policy. That update is then sent to any agent listening for QoS updates. If the agent manages a port that is using that policy, then it updates accordingly. If the agent does not know the policy, and therefore is not managing any ports using that policy, then the update is ignored.

Any ports with no specific policy attached inherit the policy of the tenant. If a port is assigned a policy, then that policy supersedes the tenant policy. It is possible to associate a QoS policy when creating a new network. A network QoS policy can contain many different rules. When a rule is applied to an entire network, policy rules may or may not apply to all resources on that network. Resources include routers, load balancers, and dhcp. By default, the policy is applied to all network resources, but that can be overridden.

A project can have only *one* QoS policy. However, assigning a QoS policy to a project is not mandatory. If a QoS policy is assigned to a project, by default all new networks within that project are assigned the same QoS policy. All network resources within that network are also, by default, assigned the same QoS policy. However, if during network creation a different QoS policy is defined, then the new network uses that QoS policy.

QoS can be used in conjunction with Open vSwitch, Linux Bridge ML2 drivers, and SR-IOV.

Configuring QoS

The **neutron.conf** configuration file on the controller node requires an additional parameter when configuring the QoS plugin:

```
[DEFAULT]
...output omitted...
service_plugins = neutron.services.qos.qos_plugin.QoSPlugin,
neutron.services.l3_router.l3_router_plugin.L3RouterPlugin,
neutron.services.metering.metering_plugin.MeteringPlugin
...output omitted...
```

Optionally, the **notification_drivers** attribute is set to the drivers that is used to send notification messages. The **message_queue** driver is the defualt driver used for sending notification messages.

```
[qos]
#notification_drivers = message_queue
```

The ML2 plugin and L2 agent configuration file, **/etc/neutron/plugins/ml2/ml2_conf.ini** on the controller node, must contain the following configuration.

```
[ml2]
...output omitted...
extension_drivers = port_security,qos
```

If the Open vSwitch agent is being used, in the **/etc/neutron/plugins/ml2/openvswitch_agent.ini** configuration file on the controller node, set **extensions** to **qos** in the **[agent]** section.

```
[agent]
...output omitted...
extensions = qos
```

The **/etc/neutron/plugins/ml2/openvswitch_agent.ini** file on each compute node requires the **extensions = qos** parameter in the **[agent]** section.

```
[agent]
...output omitted...
extensions = qos
```

If the tenants are trusted to configure their own QoS polices, the **policy.json** file in **/etc/neutron/** must be modified to allow it.

Add these entries:

```
"get_policy": "rule:regular_user",
"create_policy": "rule:regular_user",
"update_policy": "rule:regular_user",
"delete_policy": "rule:regular_user",
```

To enable the bandwidth limit rule, include the following in the same configuration file:

```
"get_policy_bandwidth_limit_rule": "rule:regular_user",
"create_policy_bandwidth_limit_rule": "rule:admin_only",
"delete_policy_bandwidth_limit_rule": "rule:admin_only",
"update_policy_bandwidth_limit_rule": "rule:admin_only",
"get_rule_type": "rule:regular_user",
```

To enable DSCP marking rules, include the following configuration in the same file:

```
"get_policy_dscp_marking_rule": "rule:regular_user",
"create_dscp_marking_rule": "rule:admin_only",
"delete_dscp_marking_rule": "rule:admin_only",
"update_dscp_marking_rule": "rule:admin_only",
"get_rule_type": "rule:regular_user",
```

Database Schema

The following objects are defined in the database schema. The database schema is configured by default when QoS policies are created. All database models are defined under one model:

neutron.db.qos.models

- `QosPolicy`: directly maps to the conceptual policy resource.
- `QosNetworkPolicyBinding`, `QosPortPolicyBinding`: defines attachment between a Neutron resource and a QoS policy.
- `QosPolicyDefault`: defines a default QoS policy per project.
- `QosBandwidthLimitRule`: defines a rule to limit the maximum egress bandwidth.
- `QosDscpMarkingRule`: defines a rule marking the Differentiated Service bits for egress traffic.
- `QosMinimumBandwidthRule`: defines a rule creating a minimum bandwidth constraint.

Introduction to Differentiated Services Code Point

Differentiated Services Code Point, or **DSCP**, is a protocol specifying and controlling network traffic by class. For example, voice traffic, which requires uninterrupted data flow, might get precedence over other kinds of traffic. DSCP is a 6-bit field in the IP header. The DS, or DiffServ, field has 8 bits. The DSCP contains the high 6-bits of the DS. The DS field can be referred to as the ToS, or Type of Service. In IPv6, the DS or ToS field, is renamed to the Traffic Class field and is the second field in the IP header. In IPv4, the DS or ToS field, is the third field in the IP header.

Before implementing DSCP, it is important to understand how the *PHB* settings work. PHB, or per-hop behavior, describes how traffic is handled depending on the DSCP value. Traffic with a lower DSCP mark, or no DSCP mark, is dropped before traffic with a higher DSCP mark. There are four kinds of PHB.

- None, which is the default setting.
- Class Selectors 1-7.
- Three forwarding subclasses for Class Selectors 1-4.
- Expedited forwarding.

For troubleshooting purposes, it is important to understand how DSCP bits are translated in ToS values. The ToS or DS values, the second or third fields in the IP header hold a hexadecimal value. For example, a DSCP mark of 22 translates to ToS hex 0x58. When tracing traffic using Wireshark or **tcpdump**, it is necessary to translate the DSCP mark into a ToS or DS value first.

DSCP Use Cases

Preferential Treatment During Congestion

Networks typically treat all traffic equally and operate on a best-effort delivery basis. This means that when a network is congested, all traffic has an equal chance of being dropped. DSCP allows administrators to prioritize specific network traffic. DSCP provides congestion-avoidance techniques using preferential treatment for certain traffic types.

Security

DSCP marks can be used as criteria in firewall rules and network device ACLs. ACLs then permit traffic depending on the DSCP mark, depending on their function and the programmed convention.

Implementing DSCP in Neutron QoS

To implement DSCP, you must first create a Qos policy using the **neutron qos-policy-create** command. The next step is to create a DSCP rule using the **neutron qos-dscp-marking-rule-create** command. The **--dscp-mark** option is used to set the DSCP value. Finally, assign the QoS policy to a port using the **neutron port-update** command.

DSCP marking takes place on the OVS integration bridge, **br-int**. The OpenFlow switch finds the matching flow entry with the highest priority base on the ingress port, packet headers and metadata. It executes instructions found in the action list, and then updates the action set. Possible actions include modifying packet headers, updating metadata, and sending packets to another table.

Once DSCP has been configured, the OVS flow table will include an action tag **mod_nw_tos:88**. As mentioned, the tcpdump output includes a **tos** parameter with the hexadecimal value. The ToS value of 88 translates to hex **0x58**. In Wireshark, search for the **Differentiated Services Field**, which holds the ToS HEX value.

Limiting Bandwidth

Limiting bandwidth for an instance can be done with the **neutron** command. The unified CLI (**openstack**) does not support the OpenStack Networking QoS commands in Red Hat OpenStack Platform 10. The bandwidth limits apply to the egress path from the instance (upload) and not the networking ingress to the instance (download). The limits are set using units of Kibibits per second (Kib/s), thus a limit of **16384** is 2 Mebibytes per second (MiB/s) (divide by 8 to convert from bits to bytes, and divide by 1024 to convert from Kibi to Mebi).

```
[user@demo ~]$ neutron qos-bandwidth-limit-rule-create qos-demo1 --max-kbps 16384
Created a new bandwidth_limit_rule:
+-----+-----+
| Field      | Value           |
+-----+-----+
| id          | bace596a-b488-4f93-bb7e-a50dd2d95248 |
| max_burst_kbps | 0               |
| max_kbps    | 16384          |
+-----+-----+
```

The bandwidth burst value is the amount of data (in Kibibits) that can be sent before the bandwidth limit applies. This option (**--max-burst-kbps**) value is not given in Kbps as the name might suggest, but a value of Kibibits. This value is actually required (even though the command does not require the option). If the value is not specified when you create the rule the default is to create it as 80% of the **--max-kbps** value.

Notice in the rule created previously with no **--max-burst-kbps** option specified, the default burst value is still applied. First, find a port to which to apply the QoS rule.

```
[user@demo ~]$ openstack port list \
-c ID -c "Fixed IP Addresses" | grep 192.168.1.8
| 4d887968-48d7-439c-a0d1-1a555070f871 | ip_address='192.168.1.8', subnet_id='a4d...b76'
```

Add the policy to the port.

```
[user@demo ~]$ neutron port-update 4d887968-48d7-439c-a0d1-1a555070f871 \
--qos-policy qos-demo1
Updated port: 4d887968-48d7-439c-a0d1-1a555070f871
```

On the compute node, view the interface of the instance.

```
[user@compute ~]$ sudo ovs-vsctl list Interface qvo4d887968-48 | grep ingress
ingress_policing_burst: 13107
ingress_policing_rate: 16384
```

Notice that even though the burst value wasn't given initially, a default value of **13107** was set, which is 80% of **16384**

Configuring QoS and DSCP

The following steps outline the process for configuring QoS and DSCP.

1. Create a new QoS policy.
2. Create a DSCP rule and apply it to the QoS policy using a DSCP mark of 18.
3. Change the DSCP mark value of the QoS policy.
4. Delete the DSCP and QoS policy.
5. Ensure that the policy has been deleted.



References

RFC 2474

<https://www.ietf.org/rfc/rfc2474.txt>

Further information is available in the chapter on Configure Quality-Of-Service (QOS) in the *Red Hat OpenStack Platform 10 Networking Guide* at

https://access.redhat.com/documentation/en-US/red_hat_openstack_platform/

Guided Exercise: Configuring QoS

In this exercise, you will configure QoS limits and create DSCP policies.

Outcomes

You should be able to:

- Configure QoS limits.
- Verify QoS limits.
- Confirm that QoS is implemented.

Before you begin

Log in to workstation as **student** using **student** as the password. On **workstation**, run the **lab datapaths-qos setup** command. This script verifies that the overcloud nodes are accessible and that the OpenStack services are running, and sets up the prerequisites for this lab.

```
[student@workstation ~]$ lab datapaths-qos setup
```

Steps

- Create a **QoS** policy and a bandwidth limit rule.
 - Source the **/home/student/architect1-finance-rc** credentials file.

```
[student@workstation ~]$ source ~/architect1-finance-rc
[student@workstation ~(architect1-finance)]$
```

- Create a QoS policy named **qos-policy1**.

```
[student@workstation ~(architect1-finance)]$ neutron qos-policy-create \
qos-policy1
+-----+-----+
| Field | Value |
+-----+-----+
| description |
| id | b1ef563c-a5c9-4d53-90a0-1559e46ba353 |
| name | qos-policy1 |
| rules |
| shared | False |
| tenant_id | cb3cae441b404b078b5bb693be35a63 |
+-----+-----+
```

- Assign bandwidth limits to the QoS rule. Set the maximum burst value to **6553**, and the maximum kbps to **8192**.

```
[student@workstation ~(architect1-finance)]$ neutron \
qos-bandwidth-limit-rule-create qos-policy1 \
--max-kbps 8192 --max-burst-kbps 6553
Created a new bandwidth_limit_rule:
+-----+-----+
| Field | Value |
+-----+-----+
```

id	b1ef563c-a5c9-4d53-90a0-1559e46ba353
max_burst_kbps	6553
max_kbps	8192

2. Associate the QoS policy to the network port of an instance.
 - 2.1. List the available instances. Make a note of both the internal IP address and the external IP address because they will be used in later steps.

```
[student@workstation ~ (architect1-finance)]$ openstack server list \
-c Name -c Networks
+-----+
| Name           | Networks          |
+-----+
| finance-qos-server1 | finance-network1=192.168.1.N, 172.25.250.P |
+-----+
```

- 2.2. Find the port ID for the **finance-qos-server1** instance using the internal IP address found in the previous step.

```
[student@workstation ~ (architect1-finance)]$ openstack port list \
-c ID -c "Fixed IP Addresses" | grep 192.168.1.N
| 6085...a079 | ip_address='192.168.1.N', subnet_id='0187...b3a4'|
```

- 2.3. Attach the QoS policy to the port of **finance-qos-server1** using the **Port ID** found in the previous step.

```
[student@workstation ~ (architect1-finance)]$ neutron port-update \
60852d77-8404-4c67-b527-69e51a20a079 --qos-policy qos-policy1
Updated port: 60852d77-8404-4c67-b527-69e51a20a079
```

3. Verify that bandwidth rate limiting is applied to the instances interface.
 - 3.1. Determine which compute node **finance-qos-server1** is running on, and the value of `instance_name`.

```
[student@workstation ~ (architect1-finance)]$ openstack server show \
finance-qos-server1
...output omitted...
| OS-EXT-SRV-ATTR:hypervisor_hostname | compute1.overcloud.example.com |
| OS-EXT-SRV-ATTR:instance_name       | instance-00000002 |
...output omitted...
```

- 3.2. Log on to the compute node using SSH. Find the ID of the instance's TAP interface.

```
[student@workstation ~ (architect1-finance)]$ ssh root@compute1
[root@compute1 ~]# virsh domiflist instance-00000002
Interface  Type      Source    Model      MAC
-----
tap9188faac-f1 bridge   qbr9188faac-f1 virtio     fa:16:3e:f1:fa:64
```

3.3. The ID used for the TAP interface and Linux bridge, will also match the **qvb/qvo** veth pair connecting the Linux bridge to OVS. Display the OVS configuration for the instances qvo interface. Note the **ingress_policing_burst** and **ingress_policing_rate** attributes.

```
[root@compute1 ~]# ovs-vsctl list Interface qvo9188faac-f1
...output omitted...
ifindex          : 16
ingress_policing_burst: 6553
ingress_policing_rate: 8192
lacp_current     : []
link_resets      : 0
link_speed       : 100000000000
link_state       : up
...output omitted...
[root@compute1 ~]# logout
[student@workstation ~(architect1-finance)]$
```

4. Test bandwidth restrictions.

4.1. Log on to **finance-qos-server1** using SSH, then create a 1GB file.

```
[student@workstation ~(architect1-finance)]$ ssh cloud-user@172.25.250.P
[cloud-user@finance-qos-server1 ~]$ dd if=/dev/zero of=1G.dat bs=1M count=1000
1000+0 records in
1000+0 records out
1048576000 bytes (1.0 GB) copied, 3.74791 s, 280 MB/s
[cloud-user@finance-qos-server1 ~]$ logout
[student@workstation ~(architect1-finance)]$
```

4.2. Open a new terminal on workstation, then use **rsync** to copy the file from **finance-qos-server1** to **workstation**.

```
[student@workstation ~]$ rsync --progress cloud-user@172.25.250.P:1G.dat ./
1G.dat
6619136  0%  993.30kB/s    0:17:24
```

Notice that the transfer rate is about 1 MB/s.

4.3. Back in the original terminal, remove the QoS policy from the port of the instance.

```
[student@workstation ~(architect1-finance)]$ neutron port-update \
60852d77-8404-4c67-b527-69e51a20a079 --no-qos-policy
Updated port: 60852d77-8404-4c67-b527-69e51a20a079
```

4.4. In the second terminal, observe the transfer rate increase.

```
[student@workstation ~]$ rsync --progress cloud-user@172.25.250.P:1G.dat ./
1G.dat
592084992  56%  84.84MB/s    0:00:05
```

You have now verified that bandwidth restrictions are working.

Cleanup

From **workstation**, run the **lab datapaths-qos cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab datapaths-qos cleanup
```

This concludes the guided exercise.

Deploying OVS-DPDK

Objectives

After completing this section, students should be able to:

- Compare native OVS to OVS-DPDK.
- Explain OVS-DPDK architecture.
- Describe the Overcloud configuration for OVS-DPDK.
- Use OVS-DPDK for inter-VM NFV applications.



Important

Save any work you want to keep from machines in the current classroom, and delete the current classroom. Use **Provision The Lab** to switch to the **DPDK** classroom.

Switching the classroom requires approximately 15 minutes. Start the switch before you begin reading the narrative of this section so that the machines are ready for the next guided exercise.

Open vSwitch Datapath

Open vSwitch datapath is the forwarding plane of Open vSwitch bridge and is implemented using a kernel module. The Open vSwitch kernel module allows users to have flexible control over flow-level packet processing on selected network devices. Each datapath has an associated flow tables the users populate these table with flows. These flows have a set of actions taken on packets that match criteria defined in the flow table. In OVS, there are two types of datapaths to choose from: *kernel datapath* and *userspace datapath*. In Red Hat Enterprise Linux, the kernel datapath is the default datapath type and is implemented using the **openvswitch.ko** kernel module.

Kernel Datapath

Native Open vSwitch forwards packets over the kernel datapath. When a packet arrives on a port on OVS, the **openvswitch.ko** kernel module, in case of the kernel datapath, passes the packet to the flow table residing in the kernel space. In the event of a matching flow, it executes the associated action. These flow tables are known as *kernel fastpath tables*. Packets are sent for userspace processing if that do not match any existing entries in the kernel fastpath table. As part of this processing, the userspace sets up a flow to handle these packets and updates the kernel fastpath table. This approach eliminates the need for context-switching between kernel and userspace for most of the packets received. Only the first packet requires processing by userspace. However when using the kernel datapath, the Linux network stack limits the network bandwidth, which reduces the overall network throughput.

Userspace Datapath

Open vSwitch can also operate entirely in userspace without assistance from a kernel module. The main forwarding plane runs in userspace as separate threads of an **ovs-vswitchd** process. This approach skips the Linux network stack and delivers packet directly to a userspace application. The network throughput increases when using the userspace datapath because there is no need for context-switching between userspace and kernel space.

Refer to this article to understand how userspace networking improves NFV performance: *User Space Networking Fuels NFV Performance* [<https://software.intel.com/en-us/blogs/2015/06/12/user-space-networking-fuels-nfv-performance>]

OVS-DPDK (Data Plane Development Kit)

The mission-critical applications used by Communication Service Providers (CSPs) and Telcos require the least amount of service disruption. Without being able to achieve very high throughput that is comparable to what is achievable with purpose-built hardware solutions, this business model breaks down. Deploying network function virtualization for these providers requires very high network throughput that is comparable to purpose-built hardware solutions.

The *OVS-DPDK (Data Plane Development Kit)* developed by Intel is a collection of libraries that reduce virtualization overheads by allowing a userspace application to poll a physical NIC for receiving or sending data. Using **Environmental Abstraction Layer (EAL)** software architecture, DPDK provides a generic interface to userspace applications to interact with different physical NICs.

OVS-DPDK uses Linux hugepages to allocate large regions of memory, and then allow userspace applications to access the data directly from these memory pages. *Poll Mode Drivers (PMDs)* allow applications to access a physical NIC directly without involving the kernel space processing.

OVS-DPDK High Level Architecture

OVS-DPDK high-level architecture consists of these following modules and libraries:

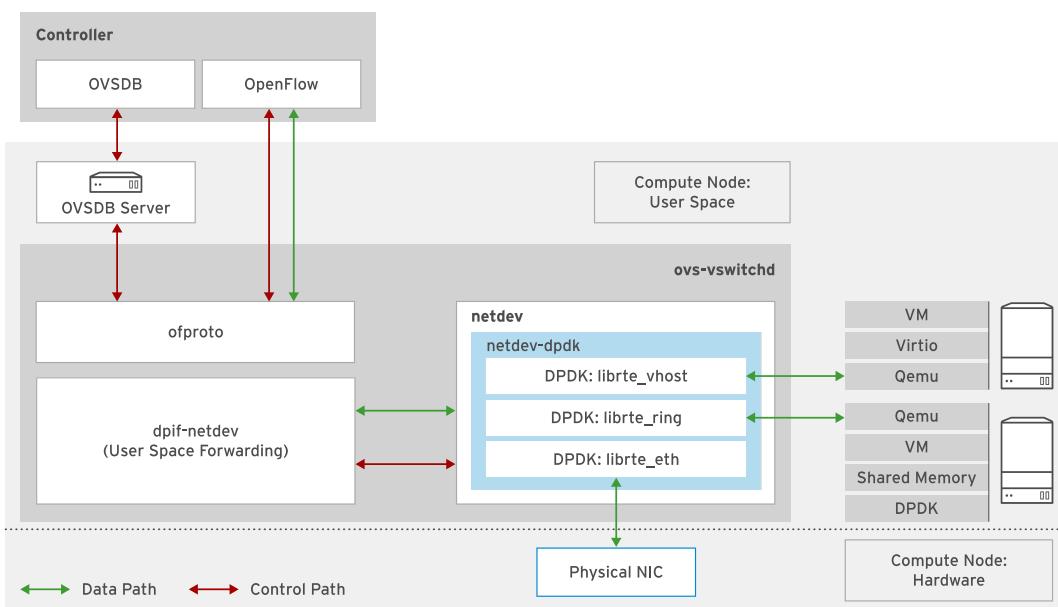


Figure 7.2: OVS-DPDK high level architecture

ovs-vswitchd

The **ovs-vswitchd** daemon runs in userspace and reads the OVS configuration from the **ovsdb-server** database server. The other components of OVS receive configuration settings from the OVS database server.

ofproto

The **ofproto** OVS library implements the OpenFlow switch. The library configures OpenFlow rules created by users using the OpenFlow controller. The **oproto** provider is used to manage these forwarding plane rules and actions to be taken on a packet. The providers for **oproto** are **dpif-netlink** and **dpif-netdev**.

netdev

The **netdev** OVS library provides an abstraction layer for interacting with a physical NIC. Every port on OVS has a corresponding **netdev** provider associated with it that is attached to a hardware or virtual NIC used inside the virtual machine. The **netdev** provider implements an interface to interact with network devices. Two such providers for **netdev** are **netdev-linux** and **netdev-dpdk**.

With OVS-DPDK, the **netdev-dpdk** library of the **netdev** module provides access to a physical DPDK-supported NIC. Three software components offer three interfaces: **librte_eth**, **librte_ring**, and **librte_vhost**. The **librte_eth** library interacts with a physical NIC using DPDK **Poll Mode Drivers (PMDs)**. The **librte_vhost** library communicates with the **vhost** port with a virtual machine's **virtio** interface. The **librte_ring** library implements the ring structure that maintains the pointer to the packet buffers.

One of the critical difference between using native OVS and OVS-DPDK is the use of **Poll Mode Drivers (PMDs)**. PMDs use a polling approach to continuously scan a physical NIC to see whether packets have arrived or not.

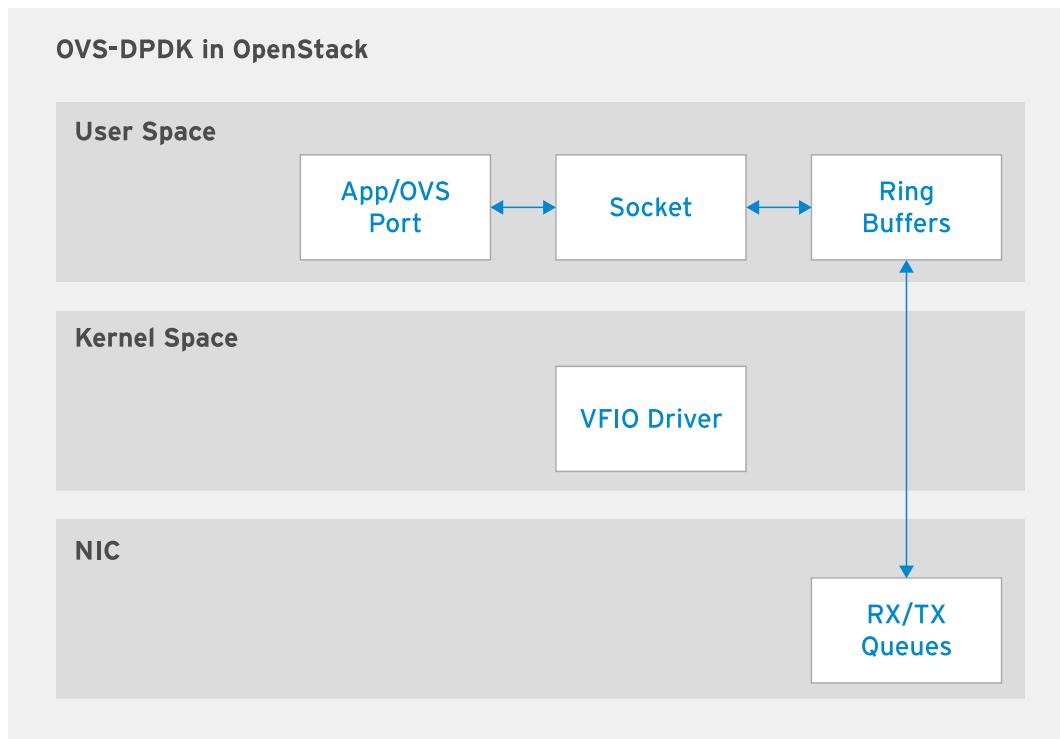


Figure 7.3: OVS-DPDK packet processing

The following steps explain packet processing within OVS-DPDK:

- When packets are received by a physical NIC, it is sent to a ring buffer, which acts as a receiving queue in userspace. The ring buffer maintains a table with pointers to packet buffers. The userspace application accesses the packet buffers in the DPDK memory pool area. These memory pools are created using hugepages provided by the compute host.
- When no packets are available, the userspace application polls the physical NIC using the PMDs. The userspace application then accesses the ring buffer to obtain the packets received.

Configuring OVS-DPDK Using Red Hat OpenStack Platform

OVS-DPDK configured with Red Hat OpenStack Platform compute nodes helps boost the performance of OVS running on compute nodes by increasing the network throughput between instances or VNFs. Applications running within the VNFs do not necessarily need to have DPDK enabled. Administrators can introduce another layer of optimization by using DPDK-enabled applications within the VNFs. Additionally, it is also possible to run DPDK-enabled VNFs without using OVS or OVS-DPDK on the compute node. DPDK-enabled VNFs require SR-IOV implemented on the compute host to access the physical NICs using VFs. The configuration requires the use of **VF Poll Mode Driver (PMD)** running within a VNF.

In Red Hat OpenStack Platform TripleO heat templates are used to deploy OVS-DPDK on compute nodes. Based on the memory, CPU, and NUMA nodes of the compute node, parameters are passed, through the network environment YAML file, to configure OVS-DPDK for optimal performance. Using OVS-DPDK with default parameters is suitable for basic deployment, but compute host tuning is often desired, and in fact required, to take advantage of multiple NUMA nodes.

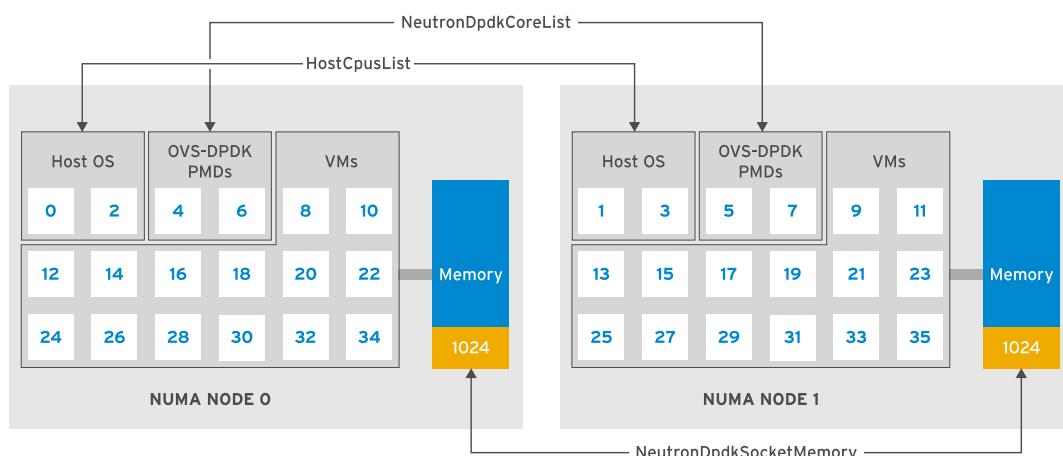


Figure 7.4: OVS-DPDK configured on compute host with two NUMA nodes

OVS-DPDK CPU Parameters

When using OVS-DPDK, the associated components should be pinned across multiple NUMA nodes to maximize resource usage. With OVS-DPDK, the host CPU is partitioned between host processes, virtual machines, and for running OVS-DPDK PMD threads. The PMDs require dedicated cores from the host CPU to run polling to receive packets from the physical network cards. In case PMD threads need to be scaled, additional cores can be added, even when traffic is already flowing.

Chapter 7. Implementing NFV Datapaths

OVS-DPDK uses the following CPU partitioning parameters in Red Hat OpenStack Platform heat templates:

NeutronDpdkCoreList

Sets CPU cores dedicated for use by OVS-DPDK PMDs located on each NUMA node. Administrators should avoid allocating the first few physical CPU cores from all NUMA nodes because the host processes uses those. CPU cores assigned to PMD threads must be excluded from compute service scheduling using **vcpu_pin_set** parameter in the **/etc/nova/nova.conf** file. For example, consider a machine with two NUMA nodes and with the total number of CPUs per NUMA node of 18. The **NeutronDpdkCoreList** parameter in network environment YAML should have cores 4,6 from NUMA0 and 5,7 from NUMA1:

This parameter sets the **pmd-cpu-mask** option of OVS-DPDK.

The parameter uses the following constraint for the values provided to this parameter:

```
NeutronDpdkCoreList:  
    description: List of cores to be used for DPDK Poll Mode Driver  
    type: string  
    constraints:  
        - allowed_pattern: "[0-9,-]+"
```

For multiple cores the values are comma-separated list inside a single quote.

```
NeutronDpdkCoreList: "4,6,5,7"
```

NovaVcpuPinSet

Sets CPU cores for nova vCPU pinning. The compute nodes use these cores for VNFs or VMs.

```
NovaCpuPinSet: "8-35"
```

HostCpusList

Sets CPU cores that must be excluded from the **cpu-partitioning** tuned profile and are not going to be used for OVS-DPDK.

This parameter sets the **dplk-lcore-mask** option of OVS-DPDK.

```
HostCpusList: "0,1,2,3"
```

HostIsolatedCoreList

Sets CPU cores that must be included with the **cpu-partitioning** tuned profile. The **HostIsolatedCoreList** and **HostCpusList** are mutually exclusive. The value for the **HostIsolatedCoreList** parameter includes the CPU cores passed as value for the **NeutronDpdkCoreList** and the **NovaVcpuPinSet** parameters.

```
HostIsolatedCoreList: "4-35"
```

OVS-DPDK Memory Parameters

OVS-DPDK uses the memory pool to store packet buffers. This is accessed by userspace applications. These memory pool areas use the hugepages allocated across all NUMA nodes.

Thus it becomes essential to allocate hugepage memory to all NUMA nodes that will have DPDK interfaces associated with them.

OVS-DPDK uses the following memory parameters in Red Hat OpenStack Platform tripleo heat templates:

NeutronDpdkSocketMemory

Sets the amount of memory in megabytes to be allocated to the memory pool per NUMA node. This value depends upon various factors such as the number of DPDK physical NIC per NUMA node and MTU, among others. The attribute is a comma-separated value for each NUMA node. The default value is **1024 MB** for the first NUMA node.

This parameter sets the **dplk-socket-mem** option of OVS-DPDK.

```
NeutronDpdkSocketMemory: "1024,1024"
```

NovaReservedHostMemory

Sets the reserve memory in megabytes for VMs on the compute host. This value sets the `reserved_host_memory_mb` attribute in **nova.conf** file.

```
NovaReservedHostMemory: "512"
```

NeutronDpdkMemoryChannels

Sets the number of memory channels attached to a NUMA node. Use **dmidecode -t memory** to determine the number of memory channels available on a compute host. Divide the number of memory channels available by the number of NUMA nodes to set the value of this parameter.

```
NeutronDpdkMemoryChannels: "4"
```

OVS-DPDK Kernel Parameters

Multiple configurations are required with kernel arguments on the compute node:

ComputeKernelArgs

Hugepage configuration: Number of hugepages to be allocated.

```
default_hugepagesz=1G hugepagesz=1G hugepages=3
```

IOMMU configuration: When the physical NIC first receives a packet, it sends it to a receive queue of the OVS-DPDK port. From there, the packet gets copied to the main memory using the Direct Memory Access (DMA) mechanism. An **Input-Output Memory Management Unit (IOMMU)** is required for safely driving DMA-capable hardware from userspace. Set the kernel argument to either **intel_iommu=on** (for Intel systems) or **amd_iommu=on** (for AMD systems). Additionally, it is also recommended to use the **iommu=pt** option to improve input-output performance for these devices.

```
intel_iommu=on iommu=pt
```

Isolated CPUs configuration: Sets the isolated CPUs to be used by the host processes from all NUMA nodes.

```
isolcpus=0,1,2,3
```



Note

Isolated CPUs configuration is not required if the deployment uses **2.8.x** version of the *tuned* package.

OVS-DPDK Other Parameters

Other parameters required for configuring OVS-DPDK are:

NeutronDpdkDriverType

Sets the driver type used by the DPDK network interface cards. During installation, the Red Hat OpenStack director assigns the **vfio-pci** drivers to the OVS-DPDK ports on the host.

NeutronDpdkType

Sets the OVS datapath type. The default value is **netdev**, which uses the userspace datapath.

NeutronVhostuserSocketDir

Sets the vhost-user socket directory.

The screen below shows an example network environment file for OVS-DPDK deployment using a compute host with two NUMA nodes:

```
parameter_defaults:
    NeutronDatapathType: "netdev"
    NeutronVhostuserSocketDir: "/var/run/openvswitch"
    NeutronDpdkDriverType: "vfio-pci"
    NovaSchedulerDefaultFilters: "...,NUMATopologyFilter,..."
    NeutronDpdkMemoryChannels: "4"
    NeutronDpdkSocketMemory: "'1024,1024'"
    NovaReservedHostMemory: 4096
    ComputeKernelArgs: "default_hugepagesz=1GB hugepagesz=1G hugepages=64 intel_iommu=on"
    NeutronDpdkCoreList: "'4,5,6,7'"
    HostCpusList: "'0,1,2,3'"
    NovaVcpuPinSet: "8-35"
    HostIsolatedCoreList: "4-35"
```

Interacting with VNFs

The VNFs send and receive packets to and from the VM's **virtio** network device using DPDK-backed **vhost-user** ports. The OVS-DPDK provides two types of **vhost-user** user ports: **dpidvhostuser** and **dpidvhostuserclient**. In order to support **vhost-user** interfaces, hugepages must be configured on the compute host.

With **dpidvhostuser** port, OVS configured with DPDK acts as the server and the QEMU as the client. This introduces limitation, whereby if DPDK based application (OVS-DPDK) crashes or is restarted, VMs hosting VNFs can not reestablish network connectivity and must be restarted. These VMs must be rebooted to recreate network sockets that were lost as a result of restart or crash as these are maintained by OVS-DPDK.

With **dpdkvhostuserclient** port, OVS-DPDK application acts as the client and QEMU acts as the server. In this configuration mode, when the OVS-DPDK is restarted or crashed the network sockets are not lost as they are maintained by QEMU. This allows OVS-DPDK to reestablish and restore network connectivity for VNFs without restart of VMs.

In the Red Hat OpenStack Platform 10, the OVS mechanism driver uses **dpdkvhostuser** as the default port. The **dpdkvhostuserclient** vhost-user port has been enabled from Red Hat OpenStack Platform 11 onwards as the default port.

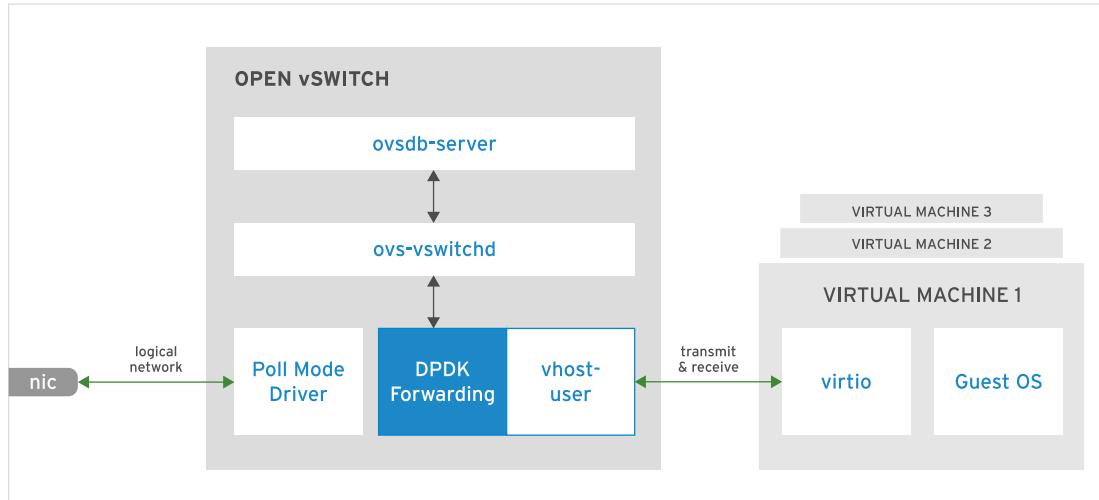


Figure 7.5: OVS-DPDK vHost port

OVS-DPDK Bridges on Compute Host

Compute nodes configured with OVS-DPDK must have the **datapath_type** option set to **netdev** for all bridges on the compute host. The OVS-DPDK physical NIC device name is represented as **dpgkX**, where X represents the port number and starts with **0**.

In a typical Red Hat OpenStack Platform deployment, the **dpgk0** OVS-DPDK physical port is linked to the **br-link** OVS-DPDK bridge.

```

Bridge "br-link"
  Port "dpgk0"
    Interface "dpgk0"
    type: dpdk

```

The **br-int** and **br-link** bridges are connected using a OVS patch peer.

```

Bridge "br-link"
  Port "phy-br-link"
    Interface "phy-br-link"
    type: patch
    options: {peer="int-br-link"}
Bridge br-int
  Port "int-br-link"
    Interface "int-br-link"
    type: patch
    options: {peer="phy-br-link"}

```

Provisioning an instance creates vhost-user interface that is associated with the virtual interface of the instance. A **vhost-user** port use hugepages configured on compute host to create the

network transmission packet buffer. A flavor is used to associate hugepages to an instance. To request an instance and use hugepage, set flavor's extra specs to use **large** pages.

```
[user@demo ~]$ openstack flavor set default --property hw:mem_page_size=large
```

The instance port is represented as **vhuX** vhost-user port and is linked to the **br-int** bridge.

```
Bridge br-int
Port "vhud502c1ea-cf"
tag: 2
Interface "vhud502c1ea-cf"
type: dpdkvhostuser
```

The screen below shows an extract from the **virsh dumpxml** command of an instance created with a **vhost-user** port.

```
...output omitted...
<interface type='vhostuser'>
<mac address='fa:16:3e:c9:74:8b' />
<source type='unix' path='/var/run/openvswitch/vhud502c1ea-cf' mode='client' />
<model type='virtio' />
<alias name='net0' />
<address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0' />
</interface>
...output omitted...
```

Gathering OVS-DPDK Port Information and Statistics

Getting the OVS-DPDK port information and statistics are essential to troubleshooting, as well as recognizing performance bottlenecks. The **ovs-appctl** command provides options to query this information and statistics. The **ovs-ofctl** command can also provide a method to administer and monitor OpenFlow rules on the OVS-DPDK bridge.

To view the information about port speeds and other states, use **ovs-ofctl show**:

```
[root@demo ~]# ovs-ofctl show br-int
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000ea7500288c4f
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src mod_dl_dst
mod_nw_src mod_nw_dst mod_nw_tos mod_tp_src mod_tp_dst
1(int-br-link): addr:f6:9b:4b:ec:cd:b9
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
2(patch-tun): addr:72:87:5b:da:6a:97
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
3(int-br-ex): addr:62:58:1d:30:aa:e8
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
...output omitted...
7(vhud502c1ea-cf): addr:00:00:00:00:00:00
    config: 0
    state: 0
```

```

speed: 0 Mbps now, 0 Mbps max
LOCAL(br-int): addr:ea:75:00:28:8c:4f
config:      PORT_DOWN
state:       LINK_DOWN
current:    10MB-FD COPPER
speed: 10 Mbps now, 0 Mbps max
OFPT_GET_CONFIG_REPLY (xid=0x4): frags=normal miss_send_len=0

```

To view the OVS-DPDK port statistics, including the transmitted and received packets, use **ovs-ofctl dump-ports**:

```

[root@demo ~]# ovs-ofctl dump-ports br-int
OFPST_PORT reply (xid=0x2): 8 ports
  port LOCAL: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0
            tx pkts=100, bytes=8004, drop=0, errs=0, coll=0
  port 5: rx pkts=?, bytes=?, drop=?, errs=?, frame=?, over=?, crc=?
            tx pkts=?, bytes=?, drop=?, errs=?, coll=?
  port 1: rx pkts=25, bytes=1600, drop=?, errs=?, frame=?, over=?, crc=?
            tx pkts=78, bytes=6248, drop=?, errs=?, coll=?
...output omitted...
  port 7: rx pkts=414, bytes=39692, drop=0, errs=0, frame=?, over=?, crc=?
            tx pkts=321, bytes=36963, drop=15, errs=?, coll=?
  port 2: rx pkts=0, bytes=0, drop=?, errs=?, frame=?, over=?, crc=?
            tx pkts=100, bytes=7668, drop=?, errs=?, coll=?
  port 3: rx pkts=271, bytes=26464, drop=?, errs=?, frame=?, over=?, crc=?
            tx pkts=329, bytes=27454, drop=?, errs=?, coll=?

```

Poll Mode Driver (PMD) statistics are primarily useful for performance tuning. These statistics are viewed using the **ovs-appctl dpif-netdev/pmd-stats-show** command.

```

[root@demo ~]# ovs-appctl dpif-netdev/pmd-stats-show
main thread:
  emc hits:9698
  megaflow hits:2165
  avg. subtable lookups per hit:1.39
  miss:933
  lost:0
  polling cycles:1165838839 (30.61%)
  processing cycles:2643134241 (69.39%)
  avg cycles per packet: 240860.82 (3808973080/15814)
  avg processing cycles per packet: 167138.88 (2643134241/15814)
pmd thread numa_id 0 core_id 1:
  emc hits:11830
  megaflow hits:2428
  avg. subtable lookups per hit:1.59
  miss:1199
  lost:0
  polling cycles:12085975255459 (99.99%)
  processing cycles:992819614 (0.01%)
  avg cycles per packet: 625975870.06 (12086968075073/19309)
  avg processing cycles per packet: 51417.45 (992819614/19309)

```

To view different ports on the existing OVS bridges, use the **ovs-appctl dpif/show**. Use the **ovs-appctl dpctl/show -s** command to view statistics for all ports.

```

[root@demo ~]# ovs-appctl dpif/show
netdev@ovs-netdev: hit:3184849 missed:565
br-int:
  br-int 65534/1: (tap)
  int-br-link 2/none: (patch: peer=phy-br-link)

```

```
patch-tun 1/none: (patch: peer=patch-int)
vhuaef323b70-00 6/6: (dpdkvhostuser: configured_rx_queues=1,..., requested_tx_queues=1)
br-link:
br-link 65534/5: (tap)
dpdk0 1/4: (dpdk: configured_rx_queues=1,...,requested_tx_queues=5)
phy-br-link 2/none: (patch: peer=int-br-link)
br-tun:
br-tun 65534/2: (tap)
patch-int 1/none: (patch: peer=patch-tun)
vxlan-ac18fa01 2/3: (vxlan: key=flow, local_ip=172.24.250.2, remote_ip=172.24.250.1)
```

Verifying OVS-DPDK Deployment on Compute Host

The following steps outline the process for verifying that a compute host is configured with OVS-DPDK.

1. View the NUMA configuration on a compute host.

```
[root@compute ~]# numactl -H
available: 1 nodes (0)
node 0 cpus: 0 1 2 3
node 0 size: 6143 MB
node 0 free: 4821 MB
node distances:
node 0
 0: 10
```

2. Ensure hugepages are available on the compute host where OVS-DPDK physical NICs are attached.

```
[root@compute ~]# grep HugePages_ /proc/meminfo
HugePages_Total:    4096
HugePages_Free:    2560
HugePages_Rsvd:     0
HugePages_Surp:     0
```

3. Use the **dpdk-devbind** command to check the status of the DPDK port binding.

```
[root@compute ~]# dpdk-devbind --status
Network devices using DPDK-compatible driver
=====
0000:00:05.0 '82574L Gigabit Network Connection' drv=uio_pci_generic unused=e1000e
0000:00:06.0 '82574L Gigabit Network Connection' drv=uio_pci_generic unused=e1000e

Network devices using kernel driver
=====
0000:00:03.0 'Virtio network device' if=ens3 drv=virtio-pci
  unused=virtio_pci,uio_pci_generic *Active*
0000:00:04.0 'Virtio network device' if=ens4 drv=virtio-pci
  unused=virtio_pci,uio_pci_generic *Active*
...output omitted...
```

4. Verify OVS-DPDK is initialized and started. Also verify that the OVS-DPDK options **dpdk-socket-mem**, **pmd-cpu-mask**, and **dpdk-lcore-mask** are set in the Red Hat OpenStack deployment.

```
[root@compute ~]# ovs-vsctl --no-wait get Open_vSwitch . other_config
```

```
{dpdk-init="true", dpdk-lcore-mask="0x1", dpdk-socket-mem="1024", pmd-cpu-
mask="0x2"}
```

5. List the interface types supported by the current OVS deployment. Ensure that DPDK interfaces are listed.

```
[root@compute ~]# ovs-vsctl --no-wait get Open_vSwitch . iface
[dpdk, dpdkr, dpdkvhostuser, dpdkvhostuserclient, geneve, gre, internal, ipsec_gre,
lisp, patch, stt, system, tap, vxlan]
```

6. List the **netdev** OVS bridges that exist on the compute host and associated ports.

```
[root@compute ~]# ovs-appctl dpif/show
netdev@ovs-netdev: hit:3184497 missed:554
  br-int:
    br-int 65534/1: (tap)
    int-br-link 2/none: (patch: peer=phy-br-link)
    patch-tun 1/none: (patch: peer=patch-int)
  br-link:
    br-link 65534/5: (tap)
    dpdk0 1/4: (dpdk: configured_rx_queues=1, ..., requested_tx_queues=5)
    phy-br-link 2/none: (patch: peer=int-br-link)
  br-tun:
    br-tun 65534/2: (tap)
    patch-int 1/none: (patch: peer=patch-tun)
    vxlan-ac18fa01 2/3: (vxlan: key=flow, local_ip=172.24.250.2,
    remote_ip=172.24.250.1)
```

7. Verify that PMD threads are running.

```
[root@compute ~]# ovs-appctl dpif-netdev/pmd-stats-show
main thread:
  emc hits:0
  megaflow hits:0
  avg. subtable lookups per hit:0.00
  miss:0
  lost:0
  polling cycles:395502426 (100.00%)
  processing cycles:0 (0.00%)
  pmd thread numa_id 0 core_id 1:
    emc hits:3183190
    megaflow hits:1446
    avg. subtable lookups per hit:1.11
    miss:558
    lost:0
    polling cycles:9895772619669 (99.94%)
    processing cycles:6053860254 (0.06%)
    avg cycles per packet: 3107135.77 (9901826479923/3186802)
    avg processing cycles per packet: 1899.67 (6053860254/3186802)
```

8. Launch an instance using the flavor with hugepages and the OVS-DPDK provider network.
9. Verify that the **vhost-user** port is created for the instance launched.

```
[root@compute ~]# virsh domiflist instance-00000001
Interface  Type      Source      Model      MAC
-----
```

-	vhostuser	-	virtio	fa:16:3e:c9:74:8b
---	-----------	---	--------	-------------------

Advantages of OVS-DPDK over SR-IOV

SR-IOV, as discussed previously, provides high network throughput by accessing the physical NIC directly from the VNFs. The VNFs using SR-IOV need to have VF drivers that support the specific physical NIC, resulting in a lack of software abstraction on the hypervisor side. In OVS-DPDK, the VNFs do not need to have any drivers that are aware of the physical NICs being used, thus providing the required abstraction. SR-IOV is limited to the number of VFs that are supported by a PF. OVS-DPDK version 2.6 has the following enhancements useful for NFV in general:

- Provides support for jumbo frames.
- Provides support for the multiqueue **vhost-user** port. The **vhost-user** multiqueue allows multiple transmit and receive queues with a vNIC in the VM.
- Supports live migration.
- Supports security groups.

Analyzing OVS-DPDK Performance

OVS-DPDK provides an application named **testpmd** that can be used to check performance and features of different DPDK based network devices. The application is distributed with the *dpdk* package. The main purpose of **testpmd** is to forward the packets between Ethernet ports on a DPDK based network device. This allows the user to test network throughput and features under different network workloads.



Important

Red Hat has tested the following original Intel NICs for OVS-DPDK: 82598, 82599, X520, X540, X550, X710, XL710, X722.

Comparing Network Throughput Between DPDK and Non-DPDK VMs

The following steps outline the process to test network throughput between two DPDK instances. The **iperf3** application is used to check the network throughput by running an **iperf3** server and **iperf3** client.



Note

TCP Segmentation Offload (TSO) is not available in the classroom deployment of OVS-DPDK because the virtual **e1000e** driver does not support TSO in OVS-DPDK. However, TSO is enabled in native OVS, therefore the network throughput for DPDK instances will actually be lower than non-DPDK instances for TCP traffic. If you want to fairly compare DPDK and non-DPDK in the classroom, disable TSO on the non-DPDK instances. This is not an issue when using physical hardware for DPDK.

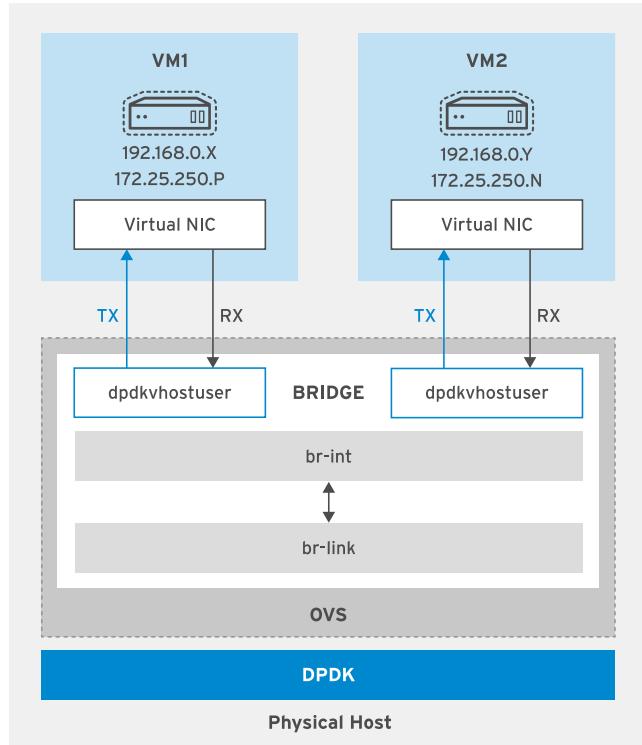


Figure 7.6: Inter-VM communication using OVS-DPDK

1. Launch two instances using OVS-DPDK tenant network.
2. Install the **iperf3** application on both instances. Set the MTU to 9000 on both instances.
3. Run the **iperf3** server on one of the instances.

```
[cloud-user@dpdk-server2 ~]$ iperf3 -s
-----
Server listening on 5201
-----
```

4. Run the **iperf3** client from the second instance.

Use the private IP address to send the traffic.

```
[cloud-user@dpdk-server1 ~]$ iperf3 -c 192.168.1.Y -u -b 0
Connecting to host 192.168.1.Y, port 5201
[  4] local 192.168.1.X port 44064 connected to 192.168.1.Y port 5201
[ ID] Interval           Transfer     Bandwidth      Total Datagrams
[  4]  0.00-1.00   sec   1.85 GBytes   15.9 Gbits/sec    221990
[  4]  1.00-2.00   sec   1.90 GBytes   16.3 Gbits/sec    227590
...output omitted...
```

5. Compare the network bandwidth reported with that of non-DPDK based instances.

The OVS-DPDK instances should report higher network bandwidth in comparison to non-DPDK based instance.

```

Connecting to host 192.168.1.X, port 5201
[ 4] local 192.168.1.Y port 44064 connected to 192.168.1.X port 5201
[ ID] Interval           Transfer     Bandwidth      Total Datagrams
[ 4]  0.00-1.00   sec    142 MBytes   1.19 Gbits/sec  16630
[ 4]  1.00-2.00   sec    157 MBytes   1.32 Gbits/sec  18400
[ 4]  2.00-3.00   sec    168 MBytes   1.41 Gbits/sec  19720
[ 4]  3.00-4.00   sec    170 MBytes   1.42 Gbits/sec  19880
[ 4]  4.00-5.00   sec    169 MBytes   1.41 Gbits/sec  19750
[ 4]  5.00-6.00   sec    169 MBytes   1.42 Gbits/sec  19860
[ 4]  6.00-7.00   sec    169 MBytes   1.42 Gbits/sec  19820
[ 4]  7.00-8.00   sec    171 MBytes   1.43 Gbits/sec  20030
[ 4]  8.00-9.00   sec    171 MBytes   1.43 Gbits/sec  20000
[ 4]  9.00-10.00  sec    168 MBytes   1.41 Gbits/sec  19650
[ 4] Sent 0 datagrams

[ ID] Interval           Transfer     Bandwidth      Jitter     Lost/Total Datagrams
[ 4]  0.00-10.00  sec  1.61 GBytes   1.39 Gbits/sec  0.000 ms  0/0 (0%)
[ 4] Sent 0 datagrams

iperf Done.

```

References



- VFIO - Virtual Function I/O**
<https://www.kernel.org/doc/Documentation/vfio.txt>
- DPDK Linux Drivers**
http://dpdk.org/doc/guides/linux_gsg/linux_drivers.html
- Open vSwitch with DPDK Overview**
<https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview>
- Boosting the NFV datapath with RHEL OpenStack Platform**
<https://redhatstackblog.redhat.com/2016/02/10/boosting-the-nfv-datapath-with-rhel-openstack-platform/>
- OVS-DPDK Parameters: Dealing with multi-NUMA**
<https://developers.redhat.com/blog/2017/06/28/ovs-dpdk-parameters-dealing-with-multi-numa/>
- Further information is available in the Configure DPDK Accelerated Open vSwitch (OVS) for Networking chapter of the *Network Functions Virtualization Configuration Guide* for Red Hat OpenStack Platform 10; at
https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/

Guided Exercise: Deploying OVS-DPDK

In this exercise, you will verify OVS-DPDK deployment on a compute host and compare the inter-VM network performance with a native OVS bridge.

Outcomes

You should be able to:

- Verify OVS-DPDK deployment on a compute host.
- Launch an instance using an OVS-DPDK bridge.
- Compare network performance between OVS-DPDK and a native OVS bridge.

Before you begin

Switch to **dpdk** classroom. Refer to the Introduction section of the course if you need help switching the classroom in your environment. The **dpdk** classroom contains **controller0**, **compute0**, and **compute1** virtual machines. These machines are started for you. OVS-DPDK is already configured on **compute0**, with the **dpdk0** interface attached to the **br-link** OVS-DPDK bridge. The **compute1** virtual machine uses native OVS bridges.

Log in to **workstation** as **student** using **student** as the password. On **workstation**, run the **lab datapaths-ovs-dpdk setup** command. This script verifies that the nodes are accessible. The script also launches the **dpdk-server2**, **nondpdk-server1**, and **nondpdk-instance2** instances for comparing network performance.

```
[student@workstation ~]$ lab datapaths-ovs-dpdk setup
```

Steps

1. Verify the NUMA configuration on the **compute0** node.

- 1.1. Log in to **compute0** as **root**.

```
[student@workstation ~]$ ssh root@compute0
[root@compute0 ~]#
```

- 1.2. View the NUMA configuration of **compute0**.

```
[root@compute0 ~]# numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3
node 0 size: 9999 MB
node 0 free: 732 MB
node distances:
node 0
0: 10
```

2. Ensure that the compute node with OVS-DPDK ports is properly tuned.
- 2.1. Verify the kernel parameters passed for **compute0** node. Ensure that the **isolcpus** and **hugepages** parameters are defined correctly.

```
[root@compute0 ~]# cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-3.10.0-693.el7.x86_64
root=UUID=716e713d-4e91-4186-81fd-c6cfa1b0974d ro console=tty0
console=ttyS0,115200n8 no_timer_check net.ifnames=1 crashkernel=auto
LANG=en_US.UTF-8 default_hugepagesz=2M hugepagesz=2M hugepages=4096
isolcpus=1-3
```

- 2.2. Check for **CPUAffinity** to ensure that the CPU cores specified in **HostCpusList** are set as desired.

```
[root@compute0 ~]# tuned-adm active
Current active profile: cpu-partitioning
[root@compute0 ~]# cat /etc/tuned/cpu-partitioning-variables.conf
...
isolated_cores=1-3
```

3. Determine the current hugepage availability on **compute0**.

```
[root@compute0 ~]# grep Huge /proc/meminfo
AnonHugePages:      157696 kB
HugePages_Total:    4096
HugePages_Free:     3072
HugePages_Rsvd:     0
HugePages_Surp:     0
Hugepagesize:       2048 kB
```

4. On **compute0**, use the **dpdk-devbind** command to check the status of the DPDK port binding.

```
[root@compute0 ~]# dpdk-devbind --status
Network devices using DPDK-compatible driver
=====
0000:00:05.0 '82574L Gigabit Network Connection' drv=uio_pci_generic unused=e1000e
0000:00:06.0 '82574L Gigabit Network Connection' drv=uio_pci_generic unused=e1000e

Network devices using kernel driver
=====
0000:00:03.0 'Virtio network device' if=ens3 drv=virtio-pci
  unused=virtio_pci,uio_pci_generic *Active*
0000:00:04.0 'Virtio network device' if=ens4 drv=virtio-pci
  unused=virtio_pci,uio_pci_generic *Active*
...output omitted...
```

5. Check that OVS-DPDK is initialized and properties are set. Also verify that the DPDK interface types are supported by the current OVS-DPDK deployment.

- 5.1. Ensure that OVS-DPDK is initialized and its other properties are using the **ovs-vsctl** command.

```
[root@compute0 ~]# ovs-vsctl get Open_vSwitch . other_config
{dpdk-init="true", dpdk-lcore-mask="1", dpdk-socket-mem="1024", pmd-cpu-
mask="2"}
```

-
- 5.2. Verify that the DPDK interface types are supported by the current OVS-DPDK deployment.

```
[root@compute0 ~]# ovs-vsctl get Open_vSwitch . iface_types  
[dpdk, dpdkr, dpdkvhostuser, dpdkvhostuserclient, geneve, gre, internal,  
 ipsec_gre, lisp, patch, stt, system, tap, vxlan]
```

6. Verify that an OVS-DPDK bridge is created using the DPDK interface.

```
[root@compute0 ~]# ovs-vsctl show  
...output omitted...  
    Bridge br-link  
        Controller "tcp:127.0.0.1:6633"  
            is_connected: true  
            fail_mode: secure  
        Port phy-br-link  
            Interface phy-br-link  
                type: patch  
                options: {peer=int-br-link}  
        Port "dpdk0"  
            Interface "dpdk0"  
                type: dpdk  
            Port br-link  
                Interface br-link  
                    type: internal  
...output omitted...
```

7. View the statistics and the CPU core for the running PMD threads.

```
[root@compute0 ~]# ovs-appctl dpif-netdev/pmd-stats-show  
...output omitted...  
pmd thread numa_id 0 core_id 1:  
    emc hits:2287  
    megaflow hits:24  
    avg. subtable lookups per hit:1.29  
    miss:86  
    lost:0  
    polling cycles:8130701542374 (100.00%)  
    processing cycles:77563767 (0.00%)  
    avg cycles per packet: 3348755809.78 (8130779106141/2428)  
    avg processing cycles per packet: 31945.54 (77563767/2428)
```

Log out of **compute0**.

```
[root@compute0 ~]# logout  
Connection to compute0 closed.  
[student@workstation ~]$
```

8. On **controller0**, verify that the required scheduler filters are added to the **scheduler_default_filters** option in **/etc/nova/nova.conf**.

- 8.1. Log in to **controller0** as **root**.

```
[student@workstation ~]$ ssh root@controller0
```

```
[root@controller0 ~]#
```

- 8.2. Use **crudini** to check that the **NUMATopologyFilter** filter is added to the **scheduler_default_filters** option in **/etc/nova/nova.conf**.

```
[root@controller0 ~]# crudini --get /etc/nova/nova.conf \
DEFAULT scheduler_default_filters
RetryFilter,AvailabilityZoneFilter,RamFilter,DiskFilter,
ComputeFilter,ComputeCapabilitiesFilter,ImagePropertiesFilter,
CoreFilter,NUMATopologyFilter,AggregateInstanceExtraSpecsFilter,
ServerGroupAntiAffinityFilter,ServerGroupAffinityFilter
```

Log out of **controller0**.

```
[root@controller0 ~]# logout
Connection to controller0 closed.
[student@workstation ~]$
```

9. From **workstation**, using **admin** credentials, verify the DPDK flavor with hugepages.

```
[student@workstation ~]$ source ~/admin-rc
[student@workstation ~(admin-admin)]$ openstack flavor show default
+-----+-----+
| Field | Value |
+-----+-----+
| OS-FLV-DISABLED:disabled | False |
| OS-FLV-EXT-DATA:ephemeral | 0 |
| access_project_ids | None |
| disk | 10 |
| id | bc27a0fb-572a-4842-8d81-3b884afb0a87 |
| name | default |
| os-flavor-access:is_public | True |
| properties | hw:mem_page_size='large' |
| ram | 1024 |
| rxtx_factor | 1.0 |
| swap | |
| vcpus | 2 |
+-----+-----+
```

10. Ensure that an internal network is created with the **dpdk** physical network and that a port is attached as an interface to the **router1** router.

```
[student@workstation ~(admin-admin)]$ openstack network show dpdk-net1
+-----+-----+
| Field | Value |
+-----+-----+
| admin_state_up | UP |
| availability_zone_hints | |
| availability_zones | nova |
| created_at | 2017-12-21T19:10:25Z |
| description | |
| id | ac0f5e6d-6a69-47f8-8c74-21fa391958cd |
| ipv4_address_scope | None |
| ipv6_address_scope | None |
| mtu | 1500 |
| name | dpdk-net1 |
| project_id | 72344a85ee34443f8d1bd75c62d1a971 |
+-----+-----+
```

project_id	72344a85ee34443f8d1bd75c62d1a971
provider:network_type	flat
provider:physical_network	dpdk
provider:segmentation_id	None
revision_number	4
router:external	Internal
shared	True
status	ACTIVE
subnets	9344c2b7-1b71-4906-a4ac-ed9cc29773c2
tags	[]
updated_at	2017-12-21T19:10:26Z

[student@workstation ~ (admin-admin)]\$ openstack port list \
--router router1 \
--device-owner network:router_interface \
-c 'Fixed IP Addresses'
+-----+ Fixed IP Addresses +-----+
ip_address='192.168.1.1', subnet_id='9344c2b7-1b71-4906-a4ac-ed9cc29773c2' ... +-----+

11. Launch an instance named **dpdk-server1** using the **default** flavor and **dpdk-net1** network. Note the value of the **OS-EXT-SRV-ATTR:instance_name** and **OS-EXT-SRV-ATTR:hypervisor_hostname** fields. In the below output, the **instance-00000008** instance is launched on the **compute0.overcloud.example.com** hypervisor.

Associate the **172.25.250.101** floating IP address to **dpdk-server1**.

[student@workstation ~ (admin-admin)]\$ openstack server create \
--image rhel7 \
--flavor default \
--key-name example-keypair \
--nic net-id=dpdk-net1,v4-fixed-ip=192.168.1.101 \
--wait dpdk-server1
+-----+ Field Value +-----+
OS-DCF:diskConfig MANUAL
OS-EXT-AZ:availability_zone nova
OS-EXT-SRV-ATTR:host compute0.overcloud.example.com
OS-EXT-SRV-ATTR:hypervisor_hostname compute0.overcloud.example.com
OS-EXT-SRV-ATTR:instance_name instance-00000008
OS-EXT-STS:power_state Running
...output omitted...
[student@workstation ~ (admin-admin)]\$ openstack server add floating ip \
dpdk-server1 172.25.250.101

12. On **compute0**, verify that the **vhostuser** port is created for the instance launched.

12.1. Log in to **compute0** as **root**.

[student@workstation ~ (admin-admin)]\$ ssh root@compute0
[root@compute0 ~]#

12.2. View the interface attached to the **dpdk-server1** instance.

```
[root@compute0 ~]# virsh domiflist instance-00000008
interface  Type      Source    Model      MAC
-----
-         vhostuser -         virtio   fa:16:3e:11:76:9a
```

Log out of **compute0**.

```
[root@compute0 ~]# logout
Connection to compute0 closed.
[student@workstation ~(admin-admin)]$
```

13. Compare the network bandwidth reported with that of non-DPDK based instances. OVS-DPDK based instances should report higher network bandwidth in comparison to non-DPDK based instances.



Note

Your actual bandwidth may differ from what is shown in the following steps.

The **dplk-server1** and **dplk-server2** instances are running on **compute0** node, which is configured with OVS-DPDK. Use the **iperf3** command to check the network bandwidth between these instances using their private addresses.

The **nondplk-server1** and **nondplk-server2** instances are running on **compute1** node, which is configured with native OVS. Use the **iperf3** command to check the network bandwidth between these instances using their private addresses.

- 13.1. On **workstation**, use **openstack server list** to list the IP addresses of all instances. Make a note of the internal IP addresses of **dplk-server1** and **nondplk-server1** because you will use them in a later step.

```
[student@workstation ~(admin-admin)]$ openstack server list \
-c Name -c Status -c Networks
+-----+-----+-----+
| Name      | Status | Networks          |
+-----+-----+-----+
| dplk-server1 | ACTIVE | dplk-net1=192.168.1.101, 172.25.250.101 |
| dplk-server2 | ACTIVE | dplk-net1=192.168.1.R, 172.25.250.102 |
| nondplk-server1 | ACTIVE | nondplk-net2=192.168.2.X, 172.25.250.103 |
| nondplk-server2 | ACTIVE | nondplk-net2=192.168.2.Y, 172.25.250.104 |
+-----+-----+-----+
```

- 13.2. On **workstation**, open four extra command terminals. Log in to all the instances on different terminals using **cloud-user** and the floating IP address associated with instances. From the first terminal, install the **iperf3** package on all instances. Set the MTU to 9000. Using jumbo frames will allow higher network bandwidth.

```
[student@workstation ~(admin-admin)]$ for ip in 101 102 103 104 \
do \
ssh cloud-user@172.25.250.${ip} sudo yum install -y iperf3 \
ssh cloud-user@172.25.250.${ip} sudo ip link set mtu 9000 dev eth0 \
done
```

```
...output omitted...
```

- 13.3. On the terminal that has an open SSH connection to **dpdk-server1**, run the **iperf3** server.

```
[cloud-user@dpdk-server1 ~]$ iperf3 -s
-----
Server listening on 5201
-----
```

On the terminal that has an open SSH connection to **dpdk-server2**, run the **iperf3** client to connect the server on **dpdk-server1**, using its internal IP address. The **-u** option forces UDP transfers, and **-b 0** sets the bandwidth limit to 0 (zero), which mean unlimited.

```
[cloud-user@dpdk-server2 ~]$ iperf3 -c 192.168.1.101 -u -b 0
Connecting to host 192.168.1.101, port 5201
[ 4] local 192.168.1.R port 51796 connected to 192.168.1.101 port 5201
[ ID] Interval      Transfer     Bandwidth      Total Datagrams
[ 4]  0.00-1.00    sec   1.85 GBytes   15.9 Gbits/sec  221990
[ 4]  1.00-2.00    sec   1.90 GBytes   16.3 Gbits/sec  227590
...output omitted...
```

- 13.4. In the terminal that has an open SSH connection to **nondpdk-server1**. Run the **iperf3** server.

```
[cloud-user@nondpdk-server1 ~]$ iperf3 -s
-----
Server listening on 5201
-----
```

In the terminal that has an open SSH connection to **nondpdk-server2**. Run the **iperf3** client to connect the server running on **nondpdk-server1** using its internal IP address. Again, use UDP and force the bandwidth limit to infinite.

```
[cloud-user@nondpdk-server2 ~]$ iperf3 -c 192.168.2.X -u -b 0
Connecting to host 192.168.2.X, port 5201
[ 4] local 192.168.2.Y port 50744 connected to 192.168.2.X port 5201
[ 4] local 192.168.2.10 port 39559 connected to 192.168.2.6 port 5201
[ ID] Interval      Transfer     Bandwidth      Total Datagrams
[ 4]  0.00-1.00    sec   142 MBytes   1.19 Gbits/sec  16630
[ 4]  1.00-2.00    sec   157 MBytes   1.32 Gbits/sec  18400
...output omitted...
```

- 13.5. Compare the network bandwidth reported by OVS-DPDK based instances running on **compute0** with that of non-DPDK based instances running on **compute1**. The OVS-DPDK based instances should report higher network bandwidth in comparison to non-DPDK based instance.

The screen below shows the reported network bandwidth by instances using OVS-DPDK bridge.

```
[cloud-user@dpdk-server2 ~]$ iperf3 -c 192.168.1.101 -u -b 0
```

```
...output omitted...
[ ID] Interval          Transfer       Bandwidth      Jitter      Lost/Total
Datagrams
[  4]  0.00-10.00  sec   20.4 GBytes   17.5 Gbits/sec  0.000 ms  0/0 (0%)
[  4] Sent 0 datagrams

iperf Done.
```

Your actual bandwidth may be different to that shown in the screen below. The screen below shows the reported network bandwidth by instances using native OVS bridge.

```
[cloud-user@nondpdk-server2 ~]$ iperf3 -c 192.168.2.X -u -b 0
...output omitted...
[ ID] Interval          Transfer       Bandwidth      Jitter      Lost/Total
Datagrams
[  4]  0.00-10.00  sec   1.61 GBytes   1.39 Gbits/sec  0.000 ms  0/0 (0%)
[  4] Sent 0 datagrams

iperf Done.
```

The above result shows that the OVS-DPDK bridge attains performance gain in network bandwidth in comparison to native OVS bridge for inter-VM communication.

Log out of all instances.

Cleanup

From **workstation**, run the **lab datapaths-ovs-dpdk cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab datapaths-ovs-dpdk cleanup
```

This concludes the guided exercise.

Quiz: Implementing NFV Datapaths

Choose the correct answer(s) to the following questions:

1. Which two items can be passed through to an instance with SR-IOV? (Choose two.)
 - a. Virtual function (VF)
 - b. Root function (RF)
 - c. Physical function (PF)
 - d. Functional port (FP)
 - e. A TAP device

2. Which command would you use to ensure **hugepages** are available on the compute host?
 - a. `grep hugepages /proc/modules`
 - b. `grep HugePages_ /proc/modules`
 - c. `grep hugepages_ /proc/meminfo`
 - d. `grep HugePages_ /proc/meminfo`

3. Which command would you use to check the status of the **DPDK port binding**?
 - a. `dpdk-devbind --list`
 - b. `dpdk-devbind --info`
 - c. `dpdk-devbind --status`
 - d. `dpdk-devbind --show`

4. Which command would you use to attach a QoS policy to a port? The port ID is **60852d77-8404-4c67-b527-69e51a20a079**. The policy name is **bw-limiter**.
 - a. `neutron port-attach 60852d77-8404-4c67-b527-69e51a20a079 --qos-policy bw-limiter`
 - b. `neutron policy-update --qos-policy bw-limiter 60852d77-8404-4c67-b527-69e51a20a079`
 - c. `neutron port-update 60852d77-8404-4c67-b527-69e51a20a079 --qos-policy bw-limiter`
 - d. `neutron policy-port-attach 60852d77-8404-4c67-b527-69e51a20a079 --qos-policy bw-limiter`

Solution

Choose the correct answer(s) to the following questions:

1. Which two items can be passed through to an instance with SR-IOV? (Choose two.)
 - a. Virtual function (VF)
 - b. Root function (RF)
 - c. Physical function (PF)
 - d. Functional port (FP)
 - e. A TAP device

2. Which command would you use to ensure **hugepages** are available on the compute host?
 - a. grep hugepages /proc/modules
 - b. grep HugePages_ /proc/modules
 - c. grep hugepages_ /proc/meminfo
 - d. grep HugePages_ /proc/meminfo

3. Which command would you use to check the status of the **DPDK port binding**?
 - a. dpdk-devbind --list
 - b. dpdk-devbind --info
 - c. **dpdk-devbind --status**
 - d. dpdk-devbind --show

4. Which command would you use to attach a QoS policy to a port? The port ID is **60852d77-8404-4c67-b527-69e51a20a079**. The policy name is **bw-limiter**.
 - a. neutron port-attach 60852d77-8404-4c67-b527-69e51a20a079 --qos-policy bw-limiter
 - b. neutron policy-update --qos-policy bw-limiter 60852d77-8404-4c67-b527-69e51a20a079
 - c. **neutron port-update 60852d77-8404-4c67-b527-69e51a20a079 --qos-policy bw-limiter**
 - d. neutron policy-port-attach 60852d77-8404-4c67-b527-69e51a20a079 --qos-policy bw-limiter

Summary

In this chapter, you learned:

- SR-IOV provides direct access to a NIC, avoiding the layer of virtualization in the hypervisor, as well as the virtual switch. This increases performance compared to instances with virtual NICs.
- VNFs using SR-IOV need to have VF drivers that support the specific physical NIC.
- QoS policies apply rate limits to outgoing traffic for instances and networks.
- Once a QoS policy is assigned to a project, all new networks within that project will be assigned the same QoS policy by default.
- OVS-DPDK configured on compute nodes helps boost the performance of an OVS running on compute nodes by increasing the network throughput between instances or VNFs.
- When using OVS-DPDK, the associated components should be pinned across multiple NUMA nodes to maximize resource usage.
- OVS-DPDK uses a memory pool to store packet buffers and is accessed by user space applications. These memory pool areas use hugepages allocated across all NUMA nodes.
- The OVS-DPDK provides two types of user ports **dppkvhostuser** and **dppkvhostuserclient**. With **dppkvhostuserclient** ports, OVS acts as the client with QEMU as the server.



CHAPTER 8

BUILDING SOFTWARE-DEFINED NETWORKS WITH OPENDAYLIGHT

Overview	
Goal	Build software-defined networks with OpenDaylight (ODL).
Objectives	<ul style="list-style-type: none">• Describe OpenDaylight architecture and use cases.• Implement OpenDaylight for automated software-defined networks.
Sections	<ul style="list-style-type: none">• Describing OpenDaylight Architecture (and Quiz)• Implementing OpenDaylight (and Guided Exercise)
Quiz	Building SDN with OpenDaylight

Describing OpenDaylight Architecture

Objectives

After completing this section, students should be able to:

- Describe the OpenDaylight Architecture.
- Describe the Overcloud Configuration for OpenDaylight.
- Discuss the relationship between OpenDaylight, Open vSwitch, and OpenFlow.

OpenDaylight Architecture

What is OpenDaylight?

The OpenDaylight project intends to unite the networking industry around a common SDN platform. Implemented into OpenStack through an ML2 Neutron mechanism driver, this SDN controller is inter-operable with a wide variety of hardware and software vendor networking solutions, providing basic functions for device and capability discovery and inventory management. The OpenDaylight controller can be enhanced with plug-ins to provide advanced features, such as performing analytics and using orchestration to dynamically propagate flow rules in response to load conditions and application requirements.

OpenDaylight is an open source software project under the Linux Foundation, with the goal of furthering the adoption and innovation of SDN. OpenDaylight is capable of handling diverse use cases and implementations. It offers centralized control in an SDN network by providing standardized interfaces to applications and business processes. It is said to be above the SDN controller through 'northbound' APIs. In switches and routers, it is said to be below the SDN controller through 'southbound' APIs and drivers. The result is a programmable network service with programmable network applications.

OpenDaylight use cases include traffic sharing, automated setup of software profiles, integration of both legacy and SDN equipment, network configuration and management, network analytics and policy control, and alarming and notification.

OpenDaylight, Open vSwitch, and OpenFlow

Open vSwitch is the virtual switch handling bridging between instances and other network resources. It uses OpenFlow to provide flow information to make packet forwarding rules. OpenFlow is an SDN protocol standard, and is able to manage switches from a centralized control plane. OpenDaylight is an SDN controller that programs and manages OpenFlow-capable switches and uses the OpenFlow protocol to communicate with hypervisors.

With OpenDaylight installed and configured as the network service provider, OpenDaylight replaces **neutron-openvswitch-agent** to perform tenant communication configuration. Use **ovs-vsctl set-manager** to add an extra manager on both the controller and compute nodes. When a new network is created and a VM is attached to it, the Neutron Network Service informs the OpenDaylight controller about the configuration by invoking OpenDaylight's REST APIs. The OpenDaylight controller then creates the port interface on **br-int** and configures the flow rules for routing.

Currently, OpenFlow offers the widest range of standardized protocol support for multi-vendor switch management. For virtual switches, there are alternatives to Open vSwitch, such as Infoblox or LINC, but Open vSwitch remains the most popular implementation.

For the purpose of OpenStack integration, OpenDaylight exposes a single common northbound service, which is implemented by the Neutron Northbound component. The exposed API matches exactly the REST API of the Neutron Networking Service. This common service allows multiple Neutron providers to exist in OpenDaylight. The Red Hat OpenDaylight solution is based on NetVirt as a Neutron provider for OpenStack. NetVirt consumes the Neutron API rather than replace it.

OpenStack and OpenDaylight NetVirt

OpenStack Neutron has two drivers to handle OpenDaylight and NetVirt: networking-odl and the OpenDaylight ML2 Driver. OpenDaylight utilizes Neutron for northbound traffic using NetVirt and MD-SAL. The southbound protocols are OVSDB and OpenFlow. The OpenDaylight OpenFlow rules can be viewed by using the **ovs-ofctl -O OpenFlow13 dump-flows** command. Add the **br_tun=** option to view the rules for a specific network.

```
[heat-admin@controller0 ~]$ sudo ovs-ofctl -O OpenFlow13 dump-flows br-int | grep tun_id
cookie=0x8800008, duration=17267.842s, table=55, n_packets=0, n_bytes=0,
priority=10, tun_id=0x8, metadata=0x800000000000/0x1fffff0000000000 actions=drop
cookie=0x8800004, duration=17267.636s, table=55, n_packets=0, n_bytes=0,
priority=10, tun_id=0x4, metadata=0x400000000000/0x1fffff0000000000 actions=drop
cookie=0x8800006, duration=17267.603s, table=55, n_packets=0, n_bytes=0,
priority=10, tun_id=0x6, metadata=0x600000000000/0x1fffff0000000000 actions=drop
cookie=0x8800007, duration=17266.951s, table=55, n_packets=0, n_bytes=0,
priority=10, tun_id=0x7, metadata=0x700000000000/0x1fffff0000000000 actions=drop
cookie=0x8800009, duration=17265.823s, table=55, n_packets=0, n_bytes=0,
priority=10, tun_id=0x9, metadata=0x900000000000/0x1fffff0000000000 actions=drop
```

Features of NetVirt delivered in Boron:

- Automatic bridge creation
- Automatic tunnel creation
- Floating IPs
- Support for multiple internal and external VLAN and flat networks
- Stateful and Stateless security groups

NetVirt Internals

NetVirt's NeutronVPNManager listens on the data control networks for Neutron data stores. From this, it obtains information about the network, routers, subnets, and port creation, and then updates events to trigger L2 and L3 configuration. The neutron VPN service listens on northbound neutron components. When a new instance is created, a tag is created for the instance. This tag is a data plane identifier and is persistent across a reboot.

Neutron networks map to Layer2 instances and determine corresponding broadcast domains. NetVirt supports VLAN trunk ports, VLAN sub-ports, and VLAN transparent networks. Forwarding is handled by switches using the VXLAN overlay. The L2-Agent automatically performs flow rewrites on source and destination switches.

L2-Agent Connectivity

OpenDaylight provides L2 connectivity between VM instances residing on the same tenant network. When a new tenant network is created, and a VM is attached to that network, OpenDaylight automatically creates Open vSwitch rules on the compute nodes to ensure tenant and VM communication.

VXLAN is currently the recommended format for tenant networks, although VLAN and GRE are both supported options. For VXLAN, OpenDaylight creates and manages tunnel endpoints to ensure communication on tenant networks.

IP Address Management

Instances created on tenant networks created with DHCP enabled automatically receive an IPv4 address. Tenants are isolated so the same subnet can be used on different tenant networks.

OpenDaylight can operate as the DHCP server. However, since the default DHCP agent also offers High Availability and cloud-init support, it is recommended over the OpenDaylight DHCP agent.

OpenDaylight currently only supports IPv4 on RHOSP10.

L3 Service Components

The NeutronVPNManager receives neutron router creation and deletion requests from OpenStack from Neutron Northbound. It also communicates with the FIB, Forwarding Information Base, Manager to create L3/VRF entries in the data plane.

The FIB service receives forwarding information from the VPN Manager. It identifies the output port for the next hop on the route. It installs the L3 forwarding rules and programs the tunnel table for remote destinations.

Routing between OpenStack Networks

When a virtual router device is created, OpenDaylight provides the L3 routing between OpenStack networks. East-west routing is supported between different networks within the same tenant. The forwarding actually takes place on the compute nodes. Both VLAN and flat external floating IPs are supported.

Security Groups

Security Groups in OpenStack filter packets based on user-created policies configured. The current implementation in OpenStack uses iptables to enforce security group rules.

OpenDaylight also supports tenant security groups. Groups can be assigned to a single instance or an entire network. OpenDaylight does not change the default behavior of OpenStack. All ingress traffic is blocked by default, but ARP and DHCP traffic are exceptions. Anti-spoofing rules are implemented to block instances from sending or receiving packets that are unknown to Neutron.

OpenDaylight does not use iptables to enforce rules. Instead, it uses OpenFlow rules, eliminating many layers of bridges and ports in the iptables implementation.

The security group is implemented in two modes: stateful and stateless. The stateful implementation used the conntrack capabilities of Open vSwitch to track an existing connection. Open vSwitch tracks the connection using the layer3 protocol, source address, destination address, layer4 protocol, and a layer4 key. The connection state is independent of the upper level state of connection-oriented protocols, such as TCP. Even connectionless protocols, such as UDP, have a pseudo state in connection tracking.

The stateless mode is for older versions of OVS, where connection tracking is not supported. Pseudo-connection tracking is achieved using the TCP SYN flag. All protocol packets other than TCP are allowed by default. For the TCP protocol, the SYN packets are dropped by default unless there is a specific rule allowing TCP SYN packets to access a particular port.

OpenDaylight and Karaf

Apache Karaf is a small OSGi-based runtime providing a lightweight container that can be used to deploy applications.

Karaf offers the following features:

- Hot deployment of OSGi bundles. As JAR files are copied to the deploy directory, they are automatically installed to the runtime.
- Dynamic Configuration. Services configured through the OSGi service are monitored and the configuration is automatically updated.
- A centralized logging back end.
- Bound to a native operating system.
- A test-based console to manage services, install, and manage applications and libraries.
- Remote access using SSH.

Karaf offers a minimal OpenDaylight container consisting of:

- karaf
- branding fragment
- osgi framework extensions
- configuration files
- installation of features using one-line commands

Karaf allows the end-user to choose specific high-level features. End-users can choose between stable and unstable features. Production environments can use **stable** versions of a feature. Development environments can be installed using a feature considered **unstable** or **experimental**.

When an end-user chooses to install OpenDaylight, the precise minimum set of features are installed.

Installed OpenDaylight features can only be uninstalled by stopping the OpenDaylight services, removing the features from Karaf, and restarting the OpenDaylight services.

It is possible to create a local repository so that environments with no Internet access can still install features. Collisions can happen between features. They should be installed one by one so that collisions can be managed.

OpenDaylight Use Cases



Note

OpenDaylight is Technology Preview in Red Hat OpenStack Platform 10. Functionality is in rapid development. Some OpenDaylight plug-ins are written for specific hardware, so they are visible within OpenStack but unavailable to you unless you have the hardware they require.

OpenDaylight controllers can be used to define the architecture for deploying Multi-Protocol Label Switching (MPLS) data carrying over high throughput telecommunications networks, handling many forms of traffic including IP, ATM, SONET and Ethernet. Using segment routing, OpenDaylight acts as a core network to enable traffic sharing between switched and routed domains. With OpenFlow control, OpenDaylight provides rapid failover and provisioning to the data plane. OpenFlow control also provides integration with Topology and Orchestration Specification for Cloud Applications (TOSCA) compliant NFV service definitions for network resource orchestration.

Both hardware and software network switches and devices are able to announce their presence upon installation and request a software task profile. OpenDaylight creates an initial configuration determined by device location, feature presentation, high-level network infrastructure architecture, and implemented policies. This OpenDaylight feature enables completely automated ("zero-touch"), vendor independent, network scaling.

Through the use of TOSCA-based orchestration definitions, OpenDaylight can define and manage network function service application definitions, such as network access control, firewalls, routers, and load balancers, and then implement the logic necessary to provision NFVs into services chains.

OpenDaylight leverages the OpenFlow protocol to manage both new software-defined networks and existing legacy hardware, using vendor specific plug-ins to provide common and consistent provisioning and operation functions. Data-mapping frameworks normalize the legacy interfaces to YANG data structures within the OpenDaylight controller. By allowing integration of existing legacy hardware, OpenDaylight eases the transition to software-defined networking at a controllable pace, while also facilitating rapid scaling without overloading existing hardware.

Through orchestration, OpenDaylight provides network device configuration management using a single, consistent management API. High level architecture and service definitions implement device configuration parameters and actions, plus the ability to track, analyze, and revert to historical configurations based on analytical evaluation of network performance and behavior. Similarly, actions can be implemented for local and networked policies by evaluating traffic patterns and event heuristics. Switches and devices can be reprogrammed by resulting OpenFlow rule- or trigger-based changes. Operators can also be notified of events and status through OpenStack polling and telemetry agents.

References

OpenDaylight home page
<https://www.opendaylight.org/>

Further information is available in the chapter on What are the components of OpenDaylight? in the *Red Hat OpenStack Platform 10 Red Hat OpenDaylight Product Guide* at

https://access.redhat.com/documentation/en-US/red_hat_openstack_platform/

Further information is available in the chapter on How does OpenDaylight cooperate with OpenStack? in the *Red Hat OpenStack Platform 10 Red Hat OpenDaylight Product Guide* at

https://access.redhat.com/documentation/en-US/red_hat_openstack_platform/

Quiz: ODL Architecture

Choose the correct answer(s) to the following questions:

1. Which two of the following are OpenDaylight use cases? (Choose two.)
 - a. Automated service delivery
 - b. Decentralized network administration
 - c. Network resource optimization
 - d. Network isolation
2. Which process does ODL replace to perform tenant communication?
 - a. neutron-openflow-agent
 - b. nova-l2-agent
 - c. neutron-openvswitch-agent
 - d. neutron-l2-agent
3. Which two of the following are features of NetVirt? (Choose two.)
 - a. Automatic floating IP assignment
 - b. Automatic security group assignment
 - c. Automatic bridge creation
 - d. Automatic tunnel creation
 - e. Automatic subnet creation
4. Which of the following could be considered a Southbound protocol?
 - a. Open vSwitch
 - b. OpenFlow
 - c. NFV
 - d. GFlow

Solution

Choose the correct answer(s) to the following questions:

1. Which two of the following are OpenDaylight use cases? (Choose two.)
 - a. **Automated service delivery**
 - b. Decentralized network administration
 - c. **Network resource optimization**
 - d. Network isolation
2. Which process does ODL replace to perform tenant communication?
 - a. neutron-openflow-agent
 - b. nova-l2-agent
 - c. **neutron-openvswitch-agent**
 - d. neutron-l2-agent
3. Which two of the following are features of NetVirt? (Choose two.)
 - a. Automatic floating IP assignment
 - b. Automatic security group assignment
 - c. **Automatic bridge creation**
 - d. **Automatic tunnel creation**
 - e. Automatic subnet creation
4. Which of the following could be considered a Southbound protocol?
 - a. Open vSwitch
 - b. **OpenFlow**
 - c. NFV
 - d. GFlow

Implementing OpenDaylight

Objectives

After completing this section, students should be able to:

- Describe the Overcloud configuration for OpenDaylight.
- Define the features and functions of YANG and NETCONF.

OpenDaylight Overcloud Configuration

OpenStack can use OpenDaylight through the ML2 (Modular Layer) plug-in. It manages network flows by utilizing Open vSwitch and OpenFlow southbound plug-ins.

To install OpenDaylight, you must first ensure that the image you will use to deploy the overcloud includes the repositories needed for OpenStack. You should also include the RHEL repositories to provide any updates that are available. The Overcloud image can be found on the Undercloud in the **/usr/share/rhosp-director-images/overcloud-full.tar** tar file.

Launch a virtual machine using the image and subscribe to the proper repositories. In the classroom environment, we added the repository files using **guestfish** as seen in the following command.

```
[stack@director ~]$ export LIBGUESTFS_BACKEND=direct \
guestfish -i -a overcloud-full-uncompressed.qcow2 \
-f guestfish-add-repos.txt
```

Install the **opendaylight** package using the **virt-customize** command. Make sure you relabel the SELinux contexts on the files.

```
[stack@director ~]$ virt-customize -a \
overcloud-full-uncompressed.qcow2 \
--install opendaylight --selinux-relabel
```

Sparsify and compress the image before uploading it into OpenStack.

```
[stack@director ~]$ virt-sparsify --compress \
overcloud-full-uncompressed.qcow2 overcloud-full.qcow2
```

Then update the image in OpenStack.

```
[stack@director ~]$ source /home/stack/stackrc
[stack@director ~]$ openstack overcloud image upload \
--update-existing --image-path /home/stack/images
```

The next step is to update the heat templates. The ODL template file can be found in the **/usr/share/openstack-tripleo-heat-templates/environments/** directory on the Undercloud node. Copy the **neutron-opendaylight-13.yaml** file into your environment templates directory. Do not use the **neutron-opendaylight.yaml** file as it is deprecated. In the classroom, we copied the **neutron-opendaylight-13.yaml** to **/home/stack/**

templates/cl310-environment/38-neutron-odl.yaml. The file numbering provides a means to order the environment files.

Configure the ODL provider network. This maps the name of the external provider network to the network interface it should use.

```
[stack@director ~]$ echo "OpenDaylightProviderMappings: \
'datacentre:br-ex'" >> \
/home/stack/templates/cl310-environment/32-network-environment.yaml
```

YANG and NETCONF

NETCONF, Network Configuration Protocol, is an IETF configuration management protocol, and YANG is its data modeling language. YANG Tools is an infrastructure project developing tooling and libraries to support NETCONF. Model Driven SAL for Controller uses YANG as its modeling language. MD-SAL is a set of infrastructure services which provides common and generic support for developers.

NETCONF technology arose from shortcomings in SNMP and SMI. Those shortcomings include configuration, backup and restore. NETCONF is designed to support the management of configuration. It has multiple configuration data stores and manages validations and transactions. It is able to distinguish between configuration and state data.

YANG was designed to write data models for the NETCONF protocol. It is human-readable and easy to learn. YANG uses structured and reusable types and groupings and supports RPCs and has well-defined versioning rules. The design also provides data modularity through modules and sub-modules.

Basic NETCONF operations include:

- Retrieve all or part of a configuration from one of the named data stores.
- Retrieve device state information.
- Load specific configuration to a specified target.
- Configure an entire data store with the contents of another data store.
- Delete a configuration data store.

Basic YANG features include:

- YANG definitions map to the XML content of NETCONF.
- YANG uses a highly readable, compact, syntax similar to C or Java.
- Can be translated to other languages, as described in RFC 6110.
- YANG is organized into containers, lists, grouping, and choices.
- YANG is structured into modules and sub-modules.
- YANG has different data types and reusable groupings.

YANG Example:

```
module redhat-system {
    namespace "http://redhat.example.com/system";
    prefix "acme";

    organization "REDHAT Inc.";
    contact "bob@redhat.com";

    description
```

```

"The modules for RedHat entities.";
revision 2017-12-12 {
    description "Updated version.";
}

container system {
    leaf host-name {
        type string;
        description "System Hostname";
    }
    ...
}

```

NETCONF is usually transported over the SSH protocol. Device configuration and the NETCONF protocol itself are encoded using XML. NETCONF devices must allow data to be locked, edited, saved, and unlocked. Data modifications are stored in nonvolatile storage and are persistent over a reboot.

YANG modules define the data module. Each module is uniquely identified by a URI. Modules are built from containers, lists of data nodes, and leafs of the data tree.

Common YANG statements include:

Statement	Description
augment	Extend existing data hierarchies
container	Defines the layer of the data hierarchy
grouping	Groups data definitions into sets
leaf	Defines a leaf node within the data hierarchy
leaf-list	A single leaf node can appear multiple times
rpc	Parameters for an RPC operation

Constraints

YANG allows for constraints to be added to the data model. These constraints prevent illogical data from being added to the data model. Constraints apply to configurations, RPC, and notification data. The **type** statement is the principal constraint. It limits the contents of a **leaf node** to that of a named type.

Other constraints include:

Statement	Description
length	Limits the length of a string
must	XPath expression must be true
pattern	The regular expression must be correct
unique	The value must be unique
min/max-elements	Used to limit the number of instances

The "leaf" Statement

The **leaf** statement has one title and no children.

```

leaf host-name {
    type string;
}

```

```
mandatory true;
config true;
description "host for db1"
}
```

NETCONF XML Encoding:

```
<host-name>my.host-name.com</host-name>
```

The "leaf-list" Statement

A **leaf-list** statement has one value, no children, and multiple instances

```
leaf-list domain-search {
    type string;
    ordered-by user;
    description "List of domain names to search";
}
```

NETCONF XML Encoding:

```
<domain search>domain1.example.com</domain search>
<domain search>domain2.example.com</domain search>
<domain search>domain3.example.com</domain search>
```

The "container" Statement

A **container** statement has no value, holds related children and only one instance

```
container system {
    container services {
        container ssh {
            presence "Enables SSH";
            description "SSH Service specific configuration";
            // more leafs, containers and stuff here....
        }
    }
}
```

NETCONF XML Encoding:

```
<system>
    <services>
        <ssh>
    </services>
</system>
```

The "augment" Statement

The **augment** extends the data model. It also inserts nodes into an existing hierarchy.

```
augment system/login/user {
    leaf expire {
        type yang:date-and-time;
    }
}
```

NETCONF XML Encoding:

```
<user>
  <name>bob</name>
  <class>senior</class>
  <other:expire>2017-12-31</other:expire>
</user>
```

YANG in OpenDaylight

An SDN Controller is a platform for deploying SDN applications. It needs to be flexible enough to accommodate diverse applications, but use a common framework and programming model. It should provide consistent APIs. It is used to scale the development process, but should be independent of controller applications. It adapts to data models. In OpenDaylight, the application development requirements, which include enforcing standards, generating code for repetitive tasks, and producing APIs, are achieved by YANG. The MD-SAL, or Model-Driven SAL, on the controller node uses YANG, which in turn invokes NETCONF for southbound traffic.

The YANG model uses YANG tools to generate an API definition. Using plug-in source code and the API definition, the build tools create either an **API** or **Plugin** bundle which is deployed to the controller node.

YANG has its own User Interface, or UI, which can be used to build these APIs and plug-ins.

OpenDaylight, Open vSwitch, and OpenFlow

Watch this video as the instructor shows the connection between OpenDaylight, Open vSwitch, and OpenFlow.

1. Ensure that OVS is connecting to ODL as a manager.
2. Ensure that OVS has OpenDaylight connected as a manager.
3. List the OVS ports and interfaces.
4. Show the OVS configuration on the controller node.
5. Show the OVS configuration on the compute node.
6. Show instance traffic on the compute node.



References

NETCONF Configuration Protocol
<https://tools.ietf.org/html/rfc7803>

An Architecture for Network Management using NETCONF and YANG
<https://tools.ietf.org/html/rfc6244>

Guided Exercise: Implementing OpenDaylight

In this exercise, you will ensure that the OpenDaylight installation is working correctly.

Outcomes

You should be able to fully test the ODL installation.

Before you begin

Switch to the **odl** classroom. Refer to the Introduction section of the course if you need help switching the classroom in your environment. Log in to **workstation** as **student** using **student** as the password. On **workstation**, run the **lab overcloud start** command to start the overcloud nodes **controller0**, **compute0**, **compute1**, and **ceph0**.

```
[student@workstation ~]$ lab overcloud start
```

On **workstation**, also run the **lab odl-building-sdn setup** command. This script verifies that the overcloud nodes are accessible and are running the correct OpenStack services.

```
[student@workstation ~]$ lab odl-building-sdn setup
```

Steps

1. Log in to the **controller node**. Use **systemctl status** to check the status of **opendaylight**. Verify that HAProxy is properly configured to listen on port **8081**.

- 1.1. Using the **systemctl status** command to check the status of **opendaylight**.

```
[student@workstation ~]$ ssh controller0
[heat-admin@controller0 ~]$ systemctl status opendaylight
● opendaylight.service - OpenDaylight SDN Controller
  Loaded: loaded (/usr/lib/systemd/system/opendaylight.service; enabled; vendor
  preset: disabled)
  Active: active (running) since Mon 2017-11-20 12:10:30 UTC; 5h 10min ago
    Docs: https://wiki.opendaylight.org/view/Main_Page
          http://www.opendaylight.org/
  Main PID: 2169 (java)
  CGroup: /system.slice/opendaylight.service
          └─2169 /usr/bin/java -Djava.security.properties=/opt/opendaylight/etc/odl.java.security -server -Xms128M -Xmx2048M -XX:+Unlo...
```

Nov 20 12:10:27 controller0 systemd[1]: Starting OpenDaylight SDN Controller...
Nov 20 12:10:30 controller0 systemd[1]: Started OpenDaylight SDN Controller.

- 1.2. Verify the HAProxy is configured to listen on port **8081**.

```
[heat-admin@controller0 ~]$ grep -A7 opendaylight \
/etc/haproxy/haproxy.cfg
listen opendaylight
  bind 172.24.1.50:8081 transparent
  bind 172.25.249.50:8081 transparent
  mode http
  balance source
  server controller0.internalapi.localdomain 172.24.1.1:8081 check fall 5 inter
2000 rise 2
```

-
- listen redis
- Using the **ssh** command, connect to the karaf account, using **karaf** as the password. List the installed features. Verify that the API is up and running, using the **web:list** command. Using the **vxlan:show** command, verify that the inter-nodes' VXLAN tunnels are up.

- Use **ssh** to log in to karaf.

```
[heat-admin@controller0 ~]$ ssh -p 8101 karaf@localhost
Password authentication
Password: karaf
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.

opendaylight-user@root>
```

- Use the **feature:list** command to list all installed features of karaf. Use the **grep** command to view only the **odl-netvirt-openstack** feature.

```
opendaylight-user@root> feature:list | grep odl-netvirt-openstack
odl-netvirt-openstack | 0.3.2.Boron-SR2-redhat-7 | x | odl-netvirt-0.3.2.Boron-
SR2-redhat-7 | OpenDaylight :: NetVirt :: OpenStack
```

- Use the **web:list** command to ensure that the API is up and running.

```
opendaylight-user@root> web:list | grep neutron
284 | Active | Deployed | 80 | /controller/nb/v2/neutron |
org.opendaylight.neutron.northbound-api (0.7.2.Boron-SR2-redhat-7)
```

- Use the **vxlan:show** command to verify that the inter-node VXLAN tunnels are up. The number of tunnels will vary as they are created on demand when instances are deployed. Log out of the karaf application using **CTRL+D**.

```
opendaylight-user@root> vxlan:show
Name                                     Description
Local IP       Remote IP      Gateway IP     AdmState
OpState        Parent          Tag
-----
tunb1666553128                               VXLAN Trunk Interface
172.24.2.12      172.24.2.1      0.0.0.0      ENABLED
UP            198508338267430/tunb1666553128 14

tun2b4b9f42af0                               VXLAN Trunk Interface
172.24.2.1      172.24.2.12     0.0.0.0      ENABLED
UP            158480259760686/tun2b4b9f42af0 13
opendaylight-user@root> Ctrl+D
[heat-admin@controller0 ~]$
```

- Using the **curl** command, ensure that the REST API is responding correctly. Then list the REST streams. Verify that **NetVirt** is ready and running.

- Use the **curl** command to ensure that the REST API is running. Then list the REST streams. Using the **curl** command, verify that NetVirt is ready and running.

```
[heat-admin@controller0 ~]$ curl -s -u "admin:admin" \
http://172.24.1.1:8181/restconf/modules | python -m json.tool
{
    "modules": [
        "module": [
            {
                "name": "dhcpservice-config",
                "namespace": "urn:opendaylight:params:xml:ns:yang:dhcpservice:config",
                "revision": "2015-07-10"
            },
            {
                "name": "openflowjava-nx-config",
                "namespace": "urn:opendaylight:params:xml:ns:yang:openflowplugin:ofjava:nx:config",
                "revision": "2014-07-11"
            },
            ...
        ]
    ]
}
```

3.2. List the REST streams.

```
[heat-admin@controller0 ~]$ curl -s -u "admin:admin" \
http://172.24.1.1:8181/restconfstreams | python -m json.tool
{
    "streams": {}
}
```

3.3. Verify that NetVirt is up and running.

```
[heat-admin@controller0 ~]$ curl -s -u "admin:admin" \
http://172.24.1.1:8181/restconf\
/operational/network-topology:network-topology\
/topology/netvirt:1 | python -m json.tool
{
    "topology": [
        {
            "topology-id": "netvirt:1"
        }
    ]
}
```

- Verify the Neutron configuration. Make sure that the file `/etc/neutron/neutron.conf` contains the `service_plugins=odl-router_v2`. Ensure that the `/etc/neutron/plugin.ini` contains the ODL configuration.

```
[heat-admin@controller0 ~]$ sudo grep 'service_plugins=' /etc/neutron/neutron.conf
service_plugins=odl-router_v2
[heat-admin@controller0 ~]$ sudo less /etc/neutron/plugin.ini
...
[m12]
...
[ml2]
...
mechanism_drivers =opendaylight_v2
...
[ml2_odl]
password=admin
username=admin
```

```
url=http://172.24.1.50:8081/controller/nb/v2/neutron
```

- Source the **overcloudrc** credentials. Verify that the Metadata and DHCP agents are up and running. Note that there is no Open vSwitch or L3-Agent, as both are now managed by OpenDaylight. Log out of **controller0** when complete.

```
[heat-admin@controller0 ~]$ source ~/overcloudrc
[heat-admin@controller0 ~]$ openstack network agent list -c "Agent Type" -c State
+-----+-----+
| Agent Type | State |
+-----+-----+
| DHCP agent | UP   |
| Metadata agent | UP   |
| Metadata agent | UP   |
| Metadata agent | UP   |
+-----+-----+
[heat-admin@controller0 ~]$ logout
[student@workstation ~]$
```

- Validate that **Open vSwitch** is properly configured and connected to OpenDaylight. Connect to one of the compute nodes and use the **ovs-vsctl show** command to list the ovs bridges.

```
[student@workstation ~]$ ssh compute0
[heat-admin@compute0 ~]$ sudo ovs-vsctl show
d6d1a689-298b-4d9d-9a23-7404805bce8f
    Manager "ptcp:6639:127.0.0.1"
    Manager "tcp:172.24.1.1:6640"
        is_connected: true
...output omitted...
```

Notice that there are multiple managers. Verify that the **tcp** manager points to the controller IP address (**172.24.1.1**). The **is_connected:** parameter should be set to **true**.

- Using the **ovs-vsctl** with the **list open_vswitch** option, list the Open vSwitch configuration settings. Verify that the **other_config** option has the correct **local_ip** value, and that the **provider_mappings** option matches the value of the provider network. Exit from the compute node.

```
[heat-admin@compute0 ~]$ sudo ovs-vsctl list open_vswitch
_uuid          : d6d1...ce8f
bridges        : [3c56...1c3b, 9784...a73a]
cur_cfg        : 54
datapath_types : [netdev, system]
db_version     : "7.14.0"
external_ids   : {hostname="compute0.overcloud.example.com", system-
id="b334...7cff"}
iface_types    : [geneve, gre, internal, ipsec_gre, lisp, patch, stt, system,
    tap, vxlan]
manager_options : [705e...fef8, a84c...f8af]
next_cfg       : 54
other_config   : {local_ip="172.24.2.2", provider_mappings="datacentre:br-ex"}
ovs_version    : "2.6.1"
ssl            : []
statistics     : {}
system_type    : rhel
```

```
system_version      : "7.4"  
[heat-admin@compute0 ~]$ exit  
[student@workstation ~]$
```

Cleanup

From **workstation**, run the **lab odl-building-sdn cleanup** script to clean up the resources created in this exercise.

```
[student@workstation ~]$ lab odl-building-sdn cleanup
```

This concludes the guided exercise.

Quiz: Building Software-defined Networks with OpenDaylight

Choose the correct answer(s) to the following questions:

1. Which of the following commands would you use to verify the Open vSwitch configuration?
 - a. sudo ovs-vsctl list open_vswitch
 - b. sudo ovs-ofctl list open_vswitch
 - c. sudo ovs-vsctl show open_vswitch
 - d. sudo ovs-vsctl open_vswitch
2. Which command would you use to log in to karaf?
 - a. ssh -p 8101 karaf@compute0
 - b. ssh -p 8181 karaf@localhost
 - c. ssh -p 8101 karaf@localhost
 - d. ssh -p 8181 karaf@controller0
3. What is NETCONF?
 - a. A data modeling protocol
 - b. An East/West networking protocol
 - c. An IETF configuration management protocol
 - d. An add-on to the SNMP protocol
4. Which of the following are basic NETCONF operations? (Choose two.)
 - a. Invoke groupings and lists
 - b. Retrieve device state information
 - c. Delete a configuration store
 - d. Translate data models into containers and lists
5. Which of the following are YANG statements? (Choose three.)
 - a. modules
 - b. grouping
 - c. leaf-list
 - d. rpc
 - e. users

Solution

Choose the correct answer(s) to the following questions:

1. Which of the following commands would you use to verify the Open vSwitch configuration?
 - a. **sudo ovs-vsctl list open_vswitch**
 - b. sudo ovs-ofctl list open_vswitch
 - c. sudo ovs-vsctl show open_vswitch
 - d. sudo ovs-vsctl open_vswitch
2. Which command would you use to log in to karaf?
 - a. ssh -p 8101 karaf@compute0
 - b. ssh -p 8181 karaf@localhost
 - c. **ssh -p 8101 karaf@localhost**
 - d. ssh -p 8181 karaf@controller0
3. What is NETCONF?
 - a. A data modeling protocol
 - b. An East/West networking protocol
 - c. **An IETF configuration management protocol**
 - d. An add-on to the SNMP protocol
4. Which of the following are basic NETCONF operations? (Choose two.)
 - a. Invoke groupings and lists
 - b. **Retrieve device state information**
 - c. **Delete a configuration store**
 - d. Translate data models into containers and lists
5. Which of the following are YANG statements? (Choose three.)
 - a. modules
 - b. **grouping**
 - c. **leaf-list**
 - d. **rpc**
 - e. users

Summary

In this chapter, you learned:

- OpenDaylight is an open source software project under the Linux Foundation, with the goal of furthering the adoption and innovation of SDN.
- Open vSwitch is the virtual switch handling bridging between instances. Open vSwitch uses OpenFlow to make packet forwarding rules. OpenFlow is an SDN protocol controller that programs and manages OpenFlow capable switches.
- OpenDaylight provides L2 connectivity between VM instances. OpenDaylight automatically creates Open vSwitch rules to ensure tenant communication. OpenDaylight can operate as a DHCP server, but it is recommended to use OpenStack DHCP.
- OpenDaylight does not use iptables to enforce rules, it uses OpenFlow rules. Open vSwitch tracks connections using the layer 3 protocol, source address, destination address, layer 4 protocol, and a layer 4 key.
- NETCONF operations include retrieving all or part of a configuration, loading specific configuration to a specified target, and deleting configuration data stores.
- Basic YANG features include definition mapping to the XML content of NETCONF, translation to other languages, organizing containers, lists, grouping and choices, and structuring modules.



CHAPTER 9

LAB: COMPREHENSIVE REVIEW OF RED HAT OPENSTACK ADMINISTRATION III

Overview	
Goal	Configure advanced networking on Red Hat OpenStack Platform.
Objectives	<ul style="list-style-type: none">• Tune OpenStack Networking for optimized performance.• Deploy VLAN tenant networks using IPv6.
Lab	<ul style="list-style-type: none">• Tuning OpenStack Networking Performance• Deploying Tenant Networks

Lab: Tuning OpenStack Networking Performance

In this lab, you will tune OpenStack Networking for optimal performance by using OVS-DPDK.

Outcomes

You should be able to:

- Create a flavor that uses hugepages and CPU Affinity.
- Configure an aggregate for high performance workloads.
- Launch an instance with a **vhost-user** network interface.

Before you begin

Switch to **dpdk** classroom. Refer to the Introduction section of the course if you need help switching the classroom in your environment. The **dpdk** classroom contains **controller0**, **compute0**, and **compute1** virtual machines. These machines are started for you. OVS-DPDK is already configured on **compute0**, with the **dpdk0** interface attached to the **br-link** OVS-DPDK bridge. The **compute1** virtual machine uses native OVS bridges.

Log in to **workstation** as **student** using **student** as the password. On **workstation**, run the **lab cr-networking-performance setup** command. This script verifies that the nodes are accessible, and sets up the prerequisites for this lab.

```
[student@workstation ~]$ lab cr-networking-performance setup
```

Steps

1. On **compute0**, view the NUMA configuration and reserve cores for virtual guests. Determine the current hugepages available on **compute0**. Ensure that OVS-DPDK PMD threads are running.
2. On **controller0**, ensure that the required **nova-scheduler** default filters are added.
3. On **workstation**, as the **admin** user, create an aggregate named **perf-aggregate**. Set **pinned** and **hpgs** properties to **true**.
Add **compute0** to the **perf-aggregate** aggregate.
4. Create a flavor named **ovs-dpdk** with a **10** GB disk, **1**GB of RAM, and **2** vCPUs. Use these property values:

Flavor Properties

Property	Value
hw:cpu_policy	dedicated
hw:cpu_thread_policy	prefer
hw:mem_page_size	large
aggregate_instance_extra_specs:pinned	true

Property	Value
aggregate_instance_extra_specs:hpgs	true

5. As **architect1** of the **production** project, create a network named **production-dpdk-network1** using the following properties and values:

Network Properties

Property	Value
provider-network-type	flat
provider-physical-network	dpdk

6. Create a subnet named **production-dpdk-subnet1** using these values:

Subnet Properties

Property	Value
subnet-range	192.168.1.0/24
dhcp	enable
dns-nameserver	172.25.250.254
network	production-dpdk-network1

7. Attach the **production-dpdk-subnet1** subnet as an interface to **production-router1**.
8. As **architect1**, launch an instance named **production-dpdk-server1** in the **production** project using these values:

Instance Properties

Property	Value
image	rhel7
flavor	ovs-dpdk
key-name	example-keypair
net-id	production-dpdk-network1

9. Verify that **production-dpdk-server1** uses a pinned CPU set and hugepages.

10. Verify that the **vhost-user** port is created for the instance launched.

Log out of **compute0**.

Evaluation

From **workstation**, run the **lab cr-networking-performance grade** command to confirm exercise success. Correct reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab cr-networking-performance grade
```

Cleanup

On **workstation**, run the **lab cr-networking-performance cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab cr-networking-performance cleanup
```

This concludes the lab.

Solution

In this lab, you will tune OpenStack Networking for optimal performance by using OVS-DPDK.

Outcomes

You should be able to:

- Create a flavor that uses hugepages and CPU Affinity.
- Configure an aggregate for high performance workloads.
- Launch an instance with a **vhost-user** network interface.

Before you begin

Switch to **dpdk** classroom. Refer to the Introduction section of the course if you need help switching the classroom in your environment. The **dpdk** classroom contains **controller0**, **compute0**, and **compute1** virtual machines. These machines are started for you. OVS-DPDK is already configured on **compute0**, with the **dpdk0** interface attached to the **br-link** OVS-DPDK bridge. The **compute1** virtual machine uses native OVS bridges.

Log in to **workstation** as **student** using **student** as the password. On **workstation**, run the **lab cr-networking-performance setup** command. This script verifies that the nodes are accessible, and sets up the prerequisites for this lab.

```
[student@workstation ~]$ lab cr-networking-performance setup
```

Steps

1. On **compute0**, view the NUMA configuration and reserve cores for virtual guests. Determine the current hugepages available on **compute0**. Ensure that OVS-DPDK PMD threads are running.

- 1.1. Log in to **compute0** as **root**.

```
[student@workstation ~]$ ssh root@compute0
[root@compute0 ~]#
```

- 1.2. View the NUMA configuration of **compute0**. This compute node consists of one NUMA node with four CPUs.

```
[root@compute0 ~]# numactl -H
available: 1 nodes (0)
node 0 cpus: 0 1 2 3
node 0 size: 9999 MB
node 0 free: 726 MB
node distances:
node 0
0: 10
```

- 1.3. View the reserve cores for virtual guests.

```
[root@compute0 ~]# crudini --get /etc/nova/nova.conf DEFAULT vcpu_pin_set
2,3
```

- 1.4. Determine the current hugepage availability.

```
[root@compute0 ~]# grep Huge /proc/meminfo
AnonHugePages: 155648 kB
HugePages_Total: 4096
HugePages_Free: 3584
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
```

- 1.5. View the statistics and the CPU cores used by the running PMD threads.

```
[root@compute0 ~]# ovs-appctl dpif-netdev/pmd-stats-show
...output omitted...
pmd thread numa_id 0 core_id 1:
  emc hits:671
  megaflow hits:30
  avg. subtable lookups per hit:1.30
  miss:61
  lost:0
  polling cycles:1978502513898 (100.00%)
  processing cycles:37258197 (0.00%)
  avg cycles per packet: 2470087106.24 (1978539772095/801)
  avg processing cycles per packet: 46514.60 (37258197/801)
```

Log out of **compute0**.

2. On **controller0**, ensure that the required **nova-scheduler** default filters are added.

- 2.1. Log in to **controller0** as **root**.

```
[student@workstation ~]$ ssh root@controller0
[root@controller0 ~]#
```

- 2.2. Use **crudini** to check that the following scheduler default filters are added to the **scheduler_default_filters** option in **/etc/nova/nova.conf**:

- NUMATopologyFilter
- AggregateInstanceExtraSpecsFilter

```
[root@controller0 ~]# crudini --get /etc/nova/nova.conf DEFAULT \
scheduler_default_filters
RetryFilter,AvailabilityZoneFilter,RamFilter,DiskFilter,ComputeFilter,
ComputeCapabilitiesFilter,ImagePropertiesFilter,NUMATopologyFilter,
AggregateInstanceExtraSpecsFilter,ServerGroupAntiAffinityFilter,
ServerGroupAffinityFilter,CoreFilter
```

Log out of **controller0**.

3. On **workstation**, as the **admin** user, create an aggregate named **perf-aggregate**. Set **pinned** and **hpgs** properties to **true**.

Add **compute0** to the **perf-aggregate** aggregate.

-
- 3.1. Source the `~/admin-rc` credential file.

```
[student@workstation ~]$ source ~/admin-rc
[student@workstation ~(admin-admin)]$
```

- 3.2. Create the **perf-aggregate** aggregate.

```
[student@workstation ~(admin-admin)]$ openstack aggregate create \
perf-aggregate
+-----+-----+
| Field | Value |
+-----+-----+
| availability_zone | None |
| created_at | 2018-01-09T10:06:48.414455 |
| deleted | False |
| deleted_at | None |
| id | 1 |
| name | perf-aggregate |
| updated_at | None |
+-----+-----+
```

- 3.3. Set **pinned** and **hpgs** properties to **true**.

```
[student@workstation ~(admin-admin)]$ openstack aggregate set \
--property pinned=true \
--property hpgs=true \
perf-aggregate
[student@workstation ~(admin-admin)]$
```

- 3.4. Add **compute0** to the **perf-aggregate** aggregate.

```
[student@workstation ~(admin-admin)]$ openstack aggregate add host \
perf-aggregate compute0.overcloud.example.com
+-----+-----+
| Field | Value |
+-----+-----+
| availability_zone | None |
| created_at | 2018-01-09T10:06:48.000000 |
| deleted | False |
| deleted_at | None |
| hosts | [u'compute0.overcloud.example.com'] |
| id | 1 |
| metadata | {u'pinned': u'true', u'hpgs': u'true'} |
| name | perf-aggregate |
| updated_at | None |
+-----+-----+
```

4. Create a flavor named **ovs-dpdk** with a **10** GB disk, **1** GB of RAM, and **2** vCPUs. Use these property values:

Flavor Properties

Property	Value
hw:cpu_policy	dedicated
hw:cpu_thread_policy	prefer

Property	Value
hw:mem_page_size	large
aggregate_instance_extra_specs:pinned	true
aggregate_instance_extra_specs:hpgs	true

4.1. Create the **ovs-dpdk** flavor.

```
[student@workstation ~(admin-admin)]$ openstack flavor create \
--ram 1024 --disk 10 --vcpus 2 \
ovs-dpdk
+-----+-----+
| Field | Value |
+-----+-----+
| OS-FLV-DISABLED:disabled | False
| OS-FLV-EXT-DATA:ephemeral | 0
| disk | 10
| id | e659db3d-617c-4937-a763-94fdf7410930
| name | ovs-dpdk
| os-flavor-access:is_public | True
| properties |
| ram | 1024
| rxtx_factor | 1.0
| swap |
| vcpus | 2
+-----+-----+
```

4.2. Set the **ovs-dpdk** flavor properties.

```
[student@workstation ~(admin-admin)]$ openstack flavor set \
--property hw:cpu_policy=dedicated \
--property hw:cpu_thread_policy=prefer \
--property hw:mem_page_size=large \
--property aggregate_instance_extra_specs:pinned=true \
--property aggregate_instance_extra_specs:hpgs=true \
ovs-dpdk
```

5. As **architect1** of the **production** project, create a network named **production-dpdk-network1** using the following properties and values:

Network Properties

Property	Value
provider-network-type	flat
provider-physical-network	dpdk

Source the **~/architect1-production-rc** credential file.

```
[student@workstation ~(admin-admin)]$ source ~/architect1-production-rc
[student@workstation ~(architect1-production)]$ openstack network create \
--provider-network-type flat \
--provider-physical-network dpdk \
production-dpdk-network1
+-----+-----+
| Field | Value |
+-----+-----+
```

admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-01-09T11:10:31Z
description	
headers	
id	9d04cc82-3a7d-449c-b334-a8cb30c61c73
ipv4_address_scope	None
ipv6_address_scope	None
mtu	1500
name	production-dpdk-network1
project_id	c474620b6ff84c2abe9b8d54c697e851
provider_id	c474620b6ff84c2abe9b8d54c697e851
provider:network_type	flat
provider:physical_network	dpdk
provider:segmentation_id	None
revision_number	2
router:external	Internal
shared	False
status	ACTIVE
subnets	
tags	[]
updated_at	2018-01-09T11:10:31Z

6. Create a subnet named **production-dpdk-subnet1** using these values:

Subnet Properties

Property	Value
subnet-range	192.168.1.0/24
dhcp	enable
dns-nameserver	172.25.250.254
network	production-dpdk-network1

```
[student@workstation ~(architect1-production)]$ openstack subnet create \
--subnet-range 192.168.1.0/24 \
--dhcp \
--dns-nameserver 172.25.250.254 \
--network production-dpdk-network1 \
production-dpdk-subnet1
+-----+-----+
| Field      | Value
+-----+-----+
| allocation_pools | 192.168.1.2-192.168.1.254
| cidr        | 192.168.1.0/24
| created_at   | 2018-01-09T11:11:52Z
| description    |
| dns_nameservers | 172.25.250.254
| enable_dhcp   | True
| gateway_ip    | 192.168.1.1
| headers       |
| host_routes    |
| id            | 3e4fcb25-fbfe-48ca-9e85-a11d8f84e042
| ip_version    | 4
| ipv6_address_mode | None
| ipv6_ra_mode   | None
| name          | production-dpdk-subnet1
| network_id    | 9d04cc82-3a7d-449c-b334-a8cb30c61c73
| project_id    | c474620b6ff84c2abe9b8d54c697e851
```

project_id	c474620b6ff84c2abe9b8d54c697e851
revision_number	2
service_types	[]
subnetpool_id	None
updated_at	2018-01-09T11:11:52Z

7. Attach the **production-dpdk-subnet1** subnet as an interface to **production-router1**.

```
[student@workstation ~(architect1-production)]$ openstack router add subnet \
production-router1 \
production-dpdk-subnet1
```

8. As **architect1**, launch an instance named **production-dpdk-server1** in the **production** project using these values:

Instance Properties

Property	Value
image	rhel7
flavor	ovs-dpdk
key-name	example-keypair
net-id	production-dpdk-network1

9. Verify that **production-dpdk-server1** uses a pinned CPU set and hugepages.

- 9.1. Determine the **instance_name** attribute of **production-dpdk-server1**. The instance should be on **compute0**, as only this host matched the filter requirements.

```
[student@workstation ~(architect1-production)]$ openstack server show \
production-dpdk-server1
+-----+-----+
| Field | Value |
+-----+-----+
| OS-DCF:diskConfig | MANUAL |
| OS-EXT-AZ:availability_zone | nova |
| OS-EXT-SRV-ATTR:host | compute0.overcloud.example.com |
| OS-EXT-SRV-ATTR:hypervisor_hostname | compute0.overcloud.example.com |
| OS-EXT-SRV-ATTR:instance_name | instance-00000005 |
| OS-EXT-STS:power_state | Running |
...output omitted...
```

- 9.2. Log in to **compute0** using **root**.

```
[student@workstation ~(architect1-production)]$ ssh root@compute0
```

```
[root@compute0 ~]#
```

- 9.3. Verify that hugepages are allocated to **production-dpdk-server1**.

```
[root@compute0 ~]# virsh dumpxml instance-00000005 | grep -C 1 hugepages
<memoryBacking>
  <hugepages>
    <page size='2048' unit='KiB' nodeset='0' />
  </hugepages>
</memoryBacking>
```

- 9.4. Verify that pinned CPU sets are used by **production-dpdk-server1**.

```
[root@compute0 ~]# virsh dumpxml instance-00000005 | grep vcpupin
<vcpupin vcpu='0' cpuset='2' />
<vcpupin vcpu='1' cpuset='3' />
```

10. Verify that the **vhost-user** port is created for the instance launched.

```
[root@compute0 ~]# virsh dumpxml instance-00000005 | grep -A 2 'interface type'
<interface type='vhostuser'>
  <mac address='fa:16:3e:da:96:d7' />
    <source type='unix' path='/var/run/openvswitch/vhu36689105-e1'
      mode='client' />
```

Log out of **compute0**.

Evaluation

From **workstation**, run the **lab cr-networking-performance grade** command to confirm exercise success. Correct reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab cr-networking-performance grade
```

Cleanup

On **workstation**, run the **lab cr-networking-performance cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab cr-networking-performance cleanup
```

This concludes the lab.

Lab: Deploying Tenant Networks

In this lab, you will create a centralized router. You will also create a VLAN tenant dual stack network.

Outcomes

You should be able to:

- Create a centralized router.
- Create a VLAN tenant network using both IPv6 and IPv4.
- Launch an instance using a dual stack network.

Before you begin

Switch to the **standard** classroom. Refer to the Introduction section of the course if you need help switching the classroom in your environment. Log in to **workstation** as **student** using **student** as the password. On **workstation**, run the **lab overcloud start** command to start the overcloud nodes **controller0**, **compute0**, **compute1**, and **ceph0**.

```
[student@workstation ~]$ lab overcloud start
```

On **workstation**, also run the **lab cr-tenant-networks setup** command. This script verifies that the nodes are accessible, and sets up the prerequisites for this lab.



Note

Please note, due to a bug in the classroom's Dibbler server, if you reboot **controller0** or shutdown your environment in the middle of this exercise, you will need to run **lab cr-tenant-networks setup** and then restart the exercise.

```
[student@workstation ~]$ lab cr-tenant-networks setup
```

Steps

1. As **architect1**, create the **production-router2** centralized router in the **production** project.
2. Attach an external gateway to **production-router2**, using the following properties and values:

Router Properties

Property	Value
IPv6 fixed IP address	2001:db8::185
IPv4 fixed IP address	172.25.250.185
external network	provider-datacentre

3. As **architect1**, create a tenant VLAN network named **production-dualstack-network2** in the **production** project.

-
4. As **operator1**, create two subnets for the **production-dualstack-network2** network, named **production-ipv4-subnet2** and **production-ipv6-subnet2**.

Use the following properties and values to create **production-ipv4-subnet2**:

IPv4 Subnet Properties

Property	Value
network	production-dualstack-network2
subnet-range	10.0.0.0/24
dns-nameserver	172.25.250.254

Use the following properties to create **production-ipv6-subnet2**:

IPv6 Subnet Properties

Property	Value
network	production-dualstack-network2
ip-version	6
ipv6-ra-mode	slaac
ipv6-address-mode	slaac
use-default-subnet-pool	no value required.

5. Attach the **production-ipv4-subnet2** subnet and **production-ipv6-subnet2** to the **production-router2** router.
6. As **operator1**, launch an instance named **production-server2** in the **production** project using the following settings:

Instance Properties

Property	Value
image	rhel7
flavor	default
key-name	example-keypair
net-id	production-dualstack-network2

7. Assign a floating IP address to the **production-server2** instance.
8. Verify that the **production-server2** instance uses both an IPv6 global IP address and an IPv4 floating IP address.

Use the IPv6 address to verify network connectivity to the **production-server2** instance from **workstation**. Also verify that **workstation** is reachable from the instance.

Evaluation

From **workstation**, run the **lab cr-tenant-networks grade** command to confirm the success of this exercise. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab cr-tenant-networks grade
```

Cleanup

From **workstation**, run the **lab cr-tenant-networks cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab cr-tenant-networks cleanup
```

This concludes the lab.

Solution

In this lab, you will create a centralized router. You will also create a VLAN tenant dual stack network.

Outcomes

You should be able to:

- Create a centralized router.
- Create a VLAN tenant network using both IPv6 and IPv4.
- Launch an instance using a dual stack network.

Before you begin

Switch to the **standard** classroom. Refer to the Introduction section of the course if you need help switching the classroom in your environment. Log in to **workstation** as **student** using **student** as the password. On **workstation**, run the **lab overcloud start** command to start the overcloud nodes **controller0**, **compute0**, **compute1**, and **ceph0**.

```
[student@workstation ~]$ lab overcloud start
```

On **workstation**, also run the **lab cr-tenant-networks setup** command. This script verifies that the nodes are accessible, and sets up the prerequisites for this lab.



Note

Please note, due to a bug in the classroom's Dibbler server, if you reboot **controller0** or shutdown your environment in the middle of this exercise, you will need to run **lab cr-tenant-networks setup** and then restart the exercise.

```
[student@workstation ~]$ lab cr-tenant-networks setup
```

Steps

1. As **architect1**, create the **production-router2** centralized router in the **production** project.

- 1.1. Source the **/home/student/architect1-production-rc** credential file.

```
[student@workstation ~]$ source ~/architect1-production-rc
[student@workstation ~(architect1-production)]$
```

- 1.2. Create the **production-dvr-router2** DVR router.

```
[student@workstation ~(architect1-production)]$ neutron router-create \
--distributed False production-router2
+-----+-----+
| Field | Value |
+-----+-----+
| admin_state_up | True |
| availability_zone_hints | |
| availability_zones | |
```

created_at	2018-01-12T05:34:18Z
description	False
distributed	False
external_gateway_info	
flavor_id	
ha	False
id	1ab2c30a-c61c-4fd1-b373-efab347fd96a
name	production-router2
project_id	11a3d996aebc44dd84805eb889e435fa
revision_number	3
routes	
status	ACTIVE
tenant_id	11a3d996aebc44dd84805eb889e435fa
updated_at	2018-01-12T05:34:18Z
+-----+-----+-----+	

2. Attach an external gateway to **production-router2**, using the following properties and values:

Router Properties

Property	Value
IPv6 fixed IP address	2001:db8::185
IPv4 fixed IP address	172.25.250.185
external network	provider-datacentre

```
[student@workstation ~(architect1-production)]$ neutron router-gateway-set \
--fixed-ip ip_address=2001:db8::185 \
--fixed-ip ip_address=172.25.250.185 \
production-router2 provider-datacentre
Set gateway for router production-router2
```

3. As **architect1**, create a tenant VLAN network named **production-dualstack-network2** in the **production** project.

```
[student@workstation ~(architect1-production)]$ openstack network create \
--provider-network-type vlan \
production-dualstack-network2
+-----+-----+
| Field | Value |
+-----+-----+
| admin_state_up | UP
| availability_zone_hints | 
| availability_zones | 
| created_at | 2018-01-12T05:35:25Z
| description | 
| headers | 
| id | 4623167d-aa3a-4fe5-b2e7-338682ada4c6
| ipv4_address_scope | None
| ipv6_address_scope | None
| mtu | 1496
| name | production-dualstack-network2
| port_security_enabled | True
| project_id | 11a3d996aebc44dd84805eb889e435fa
| project_id | 11a3d996aebc44dd84805eb889e435fa
| provider:network_type | vlan
| provider:physical_network | datacentre
| provider:segmentation_id | 62
```

qos_policy_id	None
revision_number	3
router:external	Internal
shared	False
status	ACTIVE
subnets	
tags	[]
updated_at	2018-01-12T05:35:25Z
+	-----+-----+

4. As **operator1**, create two subnets for the **production-dualstack-network2** network, named **production-ipv4-subnet2** and **production-ipv6-subnet2**.

Use the following properties and values to create **production-ipv4-subnet2**:

IPv4 Subnet Properties

Property	Value
network	production-dualstack-network2
subnet-range	10.0.0.0/24
dns-nameserver	172.25.250.254

Use the following properties to create **production-ipv6-subnet2**:

IPv6 Subnet Properties

Property	Value
network	production-dualstack-network2
ip-version	6
ipv6-ra-mode	slaac
ipv6-address-mode	slaac
use-default-subnet-pool	no value required.

- 4.1. Source the **/home/student/operator1-production-rc** credential file.

```
[student@workstation ~ (architect1-production)]$ source ~/operator1-production-rc
[student@workstation ~ (operator1-production)]$
```

- 4.2. Create an IPv4 subnet named **production-ipv4-subnet2** for the **production-dualstack-network2** project network.

```
[student@workstation ~ (operator1-production)]$ openstack subnet create \
--network production-dualstack-network2 \
--subnet-range 10.0.0.0/24 \
--dns-nameserver 172.25.250.254 \
production-ipv4-subnet2
...output omitted...
```

- 4.3. Create an IPv6 subnet named **production-ipv6-subnet** for the **production-dualstack-network2** project network.

```
[student@workstation ~ (operator1-production)]$ openstack subnet create \
```

```
--network production-dualstack-network2 \
--ip-version 6 \
--ipv6-ra-mode slaac \
--ipv6-address-mode slaac \
--use-default-subnet-pool \
production-ipv6-subnet2
+-----+-----+
| Field | Value |
+-----+-----+
| allocation_pools | ::2-::ffff:ffff:ffff:ffff |
| cidr | ::/64 |
| created_at | 2018-01-12T05:37:13Z |
| description | |
| dns_nameservers | |
| enable_dhcp | True |
| gateway_ip | ::1 |
| headers | |
| host_routes | |
| id | 9c4e70dd-15ce-495e-8bbf-46cde8c55327 |
| ip_version | 6 |
| ipv6_address_mode | slaac |
| ipv6_ra_mode | slaac |
| name | production-ipv6-subnet2 |
| network_id | 4623167d-aa3a-4fe5-b2e7-338682ada4c6 |
| project_id | 11a3d996aebc44dd84805eb889e435fa |
| project_id | 11a3d996aebc44dd84805eb889e435fa |
| revision_number | 2 |
| service_types | [] |
| subnetpool_id | prefix_delegation |
| updated_at | 2018-01-12T05:37:13Z |
| use_default_subnetpool | True |
+-----+-----+
```

5. Attach the **production-ipv4-subnet2** subnet and **production-ipv6-subnet2** to the **production-router2** router.

- 5.1. Attach the **production-ipv4-subnet2** subnet to the **production-router2** router.

```
[student@workstation ~(operator1-production)]$ openstack router add subnet \
production-router2 production-ipv4-subnet2
```

- 5.2. Attach the **production-ipv6-subnet2** subnet to the **production-router2** router.

```
[student@workstation ~(operator1-production)]$ openstack router add subnet \
production-router2 production-ipv6-subnet2
```

6. As **operator1**, launch an instance named **production-server2** in the **production** project using the following settings:

Instance Properties

Property	Value
image	rhel7
flavor	default
key-name	example-keypair
net-id	production-dualstack-network2

```
[student@workstation ~(operator1-production)]$ openstack server create \
--image rhel7 \
--flavor default \
--key-name example-keypair \
--nic net-id=production-dualstack-network2 \
--wait production-server2
...output omitted...
```

7. Assign a floating IP address to the **production-server2** instance.

```
[student@workstation ~(operator1-production)]$ openstack floating ip list \
-c 'Floating IP Address' -c 'Port'
+-----+-----+
| Floating IP Address | Port |
+-----+-----+
| 172.25.250.161      | None |
| 172.25.250.164      | None |
| 172.25.250.162      | None |
| 172.25.250.163      | None |
| 172.25.250.165      | None |
+-----+-----+
[student@workstation ~(operator1-production)]$ openstack server add floating ip \
production-server2 172.25.250.161
[student@workstation ~(operator1-production)]$
```

8. Verify that the **production-server2** instance uses both an IPv6 global IP address and an IPv4 floating IP address.

Use the IPv6 address to verify network connectivity to the **production-server2** instance from **workstation**. Also verify that **workstation** is reachable from the instance.

- 8.1. Get a list of server instances. Retrieve the associated floating IP address, and the internal IPv4 address, for **production-server2**.

```
[student@workstation ~(operator1-production)]$ openstack server list \
-c Name -c Networks -f json
[
  {
    "Name": "production-server2",
    "Networks": "production-dualstack-
network2=2001:db8:1:X:NNNN, 10.0.0.P, 172.25.250.161"
  }
]
```

- 8.2. Use SSH to log in to **production-server2** using the **cloud-user** user and the IPv6 address associated with the instance.

```
[student@workstation ~(operator1-production)]$ ssh cloud-user@2001:db8:1:X:NNNN
[cloud-user@production-server2 ~]$
```

- 8.3. Verify that an IPv6 address with the **2001:db8:1:X::/64** prefix is assigned to the instance.

```
[cloud-user@production-server2 ~]$ ip -6 addr show dev eth0
```

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1446 state UP qlen 1000
inet6 2001:db8:1:X:NNNN/64 scope global mngtmpaddr dynamic
    valid_lft 86389sec preferred_lft 14389sec
inet6 fe80::NNNN/64 scope link
    valid_lft forever preferred_lft forever
```

- 8.4. Verify that **workstation** is reachable from **production-server2** using the **ping6** command.

```
[cloud-user@production-server2 ~]$ ping6 -c3 workstation
PING workstation(workstation.lab.example.com (2001:db8::5054:ff:fe00:faf
e)) 56
data bytes
64 bytes from workstation.lab.example.com (2001:db8::5054:ff:fe00:faf
e):
icmp_seq=1 ttl=62 time=0.878 ms
64 bytes from workstation.lab.example.com (2001:db8::5054:ff:fe00:faf
e):
icmp_seq=2 ttl=62 time=0.944 ms
64 bytes from workstation.lab.example.com (2001:db8::5054:ff:fe00:faf
e):
icmp_seq=3 ttl=62 time=0.820 ms

--- workstation ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 0.820/0.880/0.944/0.061 ms
```

Log out of **production-server2**.

Evaluation

From **workstation**, run the **lab cr-tenant-networks grade** command to confirm the success of this exercise. Correct any reported failures and rerun the command until successful.

```
[student@workstation ~]$ lab cr-tenant-networks grade
```

Cleanup

From **workstation**, run the **lab cr-tenant-networks cleanup** command to clean up this exercise.

```
[student@workstation ~]$ lab cr-tenant-networks cleanup
```

This concludes the lab.