

Learning OpenStack Networking

Third Edition

Build a solid foundation in virtual networking technologies for
OpenStack-based clouds



Packt>

www.packt.com

By James Denton

Learning OpenStack Networking

Third Edition

Build a solid foundation in virtual networking technologies for
OpenStack-based clouds

James Denton



BIRMINGHAM - MUMBAI

Learning OpenStack Networking

Third Edition

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author(s), nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Kartikey Pandey
Acquisition Editor: Prachi Bisht
Content Development Editor: Trusha Shriyan
Technical Editor: Cymon Pereira
Copy Editor: Safis Editing
Project Coordinator: Kinjal Bari
Proofreader: Safis Editing
Indexer: Aishwarya Gangawane
Graphics: Jisha Chirayil
Production Coordinator: Shraddha Falebhai

First published: October 2014
Second edition: November 2015
Third edition : August 2018

Production reference: 1310818

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78839-249-5

www.packtpub.com



`mapt.io`

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

James Denton is a Principal Architect at Rackspace with over 15 years of experience in systems administration and networking. He has a bachelor's degree in Business Management with a focus on Computer Information Systems from Texas State University in San Marcos, Texas. He is currently focused on OpenStack operations and support within the Rackspace Private Cloud team. James is the author of the *Learning OpenStack Networking (Neutron)*, first and second editions, as well as *OpenStack Networking Essentials*, both by Packt Publishing.

About the reviewers

Andy McCrae works as a Principal Software Engineer at Red Hat in the Multi-Architecture team. Andy began his career at Rackspace as a Linux systems administrator, after completing a master's in engineering, majoring in Computer Science at University College London (UCL). He specializes in deployment and operations automation using tools such as Ansible and Chef, as well as in distributed storage systems, specifically Swift (OpenStack Object Storage) and Ceph. Andy was the Project Technical Lead for the OpenStack-Ansible project for the Ocata and Pike cycles and has given talks at multiple international OpenStack events. Andy is currently a maintainer on the ceph-ansible project and was previously a core reviewer on the Chef-OpenStack project. Andy was also a technical reviewer on the third and fourth editions of the *OpenStack Cloud Computing Cookbook*, Packt Publishing.

Kevin Jackson is married and has three children. He has over 20 years experience working with hosted environments, and private and public clouds. He is an OpenStack specialist at Rackspace and has been working with OpenStack since the first release. Kevin has co-authored a number of OpenStack books, including the *OpenStack Cloud Computing Cookbook*.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Introduction to OpenStack Networking	8
What is OpenStack Networking?	8
Features of OpenStack Networking	9
Switching	10
Routing	10
Load balancing	11
Firewalling	11
Virtual private networks	11
Network functions virtualization	12
OpenStack Networking resources	12
Virtual network interfaces	13
Virtual network switches	14
Overlay networks	14
Virtual Extensible Local Area Network (VXLAN)	15
Generic Router Encapsulation (GRE)	16
Generic Network Virtualization Encapsulation (GENEVE)	16
Preparing the physical infrastructure	17
Configuring the physical infrastructure	19
Management network	19
API network	20
External network	20
Guest network	21
Physical server connections	21
Single interface	21
Multiple interfaces	22
Bonding	22
Separating services across nodes	24
Using a single controller node	24
Using a dedicated network node	27
Summary	28
Chapter 2: Installing OpenStack	29
System requirements	30
Operating system requirements	30
Initial network configuration	31
Example networks	31
Interface configuration	34
Initial steps	36
Permissions	36
Configuring the OpenStack repository	36

Upgrading the system	37
Setting the hostnames	37
Installing and configuring Network Time Protocol	37
Rebooting the system	38
Installing OpenStack	38
Installing and configuring the MySQL database server	39
Installing and configuring the messaging server	40
Installing and configuring memcached	40
Installing and configuring the identity service	41
Configuring the database	41
Installing Keystone	42
Configuring tokens and drivers	42
Bootstrap the Identity service	43
Configuring the Apache HTTP server	43
Setting environment variables	43
Defining services and API endpoints in Keystone	44
Defining users, projects, and roles in Keystone	45
Installing and configuring the image service	45
Configuring the database	45
Defining the Glance user, service, and endpoints	46
Installing and configuring Glance components	46
Configuring authentication settings	47
Configuring additional settings	48
Verifying the Glance image service installation	48
Installing additional images	49
Installing and configuring the Compute service	50
Configuring the database	50
Defining the Nova user, service, and endpoints	51
Installing and configuring controller node components	51
Configuring authentication settings	52
Additional controller tasks	53
Installing and configuring compute node components	54
Additional compute tasks	56
Adding the compute node(s) to the cell database	57
Installing the OpenStack Dashboard	57
Updating the host and API version configuration	57
Configuring Keystone settings	58
Modifying network configuration	58
Uninstalling default Ubuntu theme (optional)	58
Reloading Apache	59
Testing connectivity to the dashboard	59
Familiarizing yourself with the dashboard	59
Summary	61
Chapter 3: Installing Neutron	62
Basic networking elements in Neutron	62
Extending functionality with plugins	63
Modular Layer 2 plugin	64

Drivers	64
TypeDrivers	64
Mechanism drivers	65
ML2 architecture	66
Network namespaces	67
Installing and configuring Neutron services	68
Creating the Neutron database	68
Configuring the Neutron user, role, and endpoint in Keystone	69
Installing Neutron packages	69
Configuring Neutron to use Keystone	70
Configuring Neutron to use a messaging service	71
Configuring Nova to utilize Neutron networking	71
Configuring Neutron to notify Nova	72
Configuring Neutron services	72
Starting neutron-server	74
Configuring the Neutron DHCP agent	74
Restarting the Neutron DHCP agent	75
Configuring the Neutron metadata agent	76
Restarting the Neutron metadata agent	78
Interfacing with OpenStack Networking	79
Using the OpenStack command-line interface	79
Using the Neutron command-line interface	80
Using the OpenStack Python SDK	81
Using the cURL utility	81
Summary	83
Chapter 4: Virtual Network Infrastructure Using Linux Bridges	84
Using the Linux bridge driver	85
Visualizing traffic flow through Linux bridges	86
VLAN	87
Flat	90
VXLAN	94
Potential issues when using overlay networks	95
Local	96
Configuring the ML2 networking plugin	98
Configuring the bridge interface	98
Configuring the overlay interface	98
ML2 plugin configuration options	100
Type drivers	101
Mechanism drivers	101
Using the L2 population driver	101
Tenant network types	102
Flat networks	103
Network VLAN ranges	103
VNI ranges	104
Security groups	105
Configuring the Linux bridge driver and agent	105

Installing the Linux bridge agent	105
Updating the Linux bridge agent configuration file	106
Physical interface mappings	106
Enabling VXLAN	107
L2 population	107
Local IP	108
Firewall driver	108
Configuring the DHCP agent to use the Linux bridge driver	109
Restarting services	109
Verifying Linux bridge agents	109
Summary	110
Chapter 5: Building a Virtual Switching Infrastructure Using Open vSwitch	111
Using the Open vSwitch driver	111
Basic OpenvSwitch commands	114
Base commands	114
ovs-vsctl	114
ovs-ofctl	115
ovs-dpctl	115
ovs-appctl	115
Visualizing traffic flow when using Open vSwitch	116
Identifying ports on the virtual switch	118
Identifying the local VLANs associated with ports	120
Programming flow rules	122
Flow rules for VLAN networks	122
Return traffic	125
Flow rules for flat networks	126
Flow rules for overlay networks	129
Flow rules for local networks	133
Configuring the ML2 networking plugin	134
Configuring the bridge interface	134
Configuring the overlay interface	134
ML2 plugin configuration options	136
Mechanism drivers	136
Flat networks	136
Network VLAN ranges	137
Tunnel ID ranges	137
VNI Ranges	138
Security groups	139
Configuring the Open vSwitch driver and agent	139
Installing the Open vSwitch agent	140
Updating the Open vSwitch agent configuration file	140
Tunnel types	140
L2 population	141
VXLAN UDP port	141
Integration bridge	142
Tunnel bridge	142

Local IP	142
Bridge mappings	143
Configuring the bridges	144
Firewall driver	144
Configuring the DHCP agent to use the Open vSwitch driver	145
Restarting services	145
Verifying Open vSwitch agents	146
Summary	146
Chapter 6: Building Networks with Neutron	148
Network management in OpenStack	148
Provider and tenant networks	149
Managing networks in the CLI	151
Creating a flat network in the CLI	153
Creating a VLAN network in the CLI	155
Creating a local network in the CLI	157
Listing networks in the CLI	158
Showing network properties in the CLI	159
Updating network attributes in the CLI	160
Deleting networks in the CLI	161
Creating networks in the dashboard	162
Via the Project panel	162
Via the Admin panel	164
Subnet management in OpenStack	165
Working with IPv4 addresses	166
Working with IPv6 addresses	167
Creating subnets in the CLI	167
Creating a subnet in the CLI	170
Listing subnets in the CLI	172
Showing subnet properties in the CLI	172
Updating a subnet in the CLI	173
Creating subnets in the dashboard	175
Via the Project tab	175
Via the Admin tab	178
Managing subnet pools	182
Creating a subnet pool	182
Creating a subnet from a pool	184
Deleting a subnet pool	185
Assigning a default subnet pool	185
Managing network ports in OpenStack	187
Creating a port	189
Summary	190
Chapter 7: Attaching Instances to Networks	191
Attaching instances to networks	191
Attaching instances to networks at creation	192
Specifying a network	193
Specifying a port	193

Attaching multiple interfaces	193
Attaching network interfaces to running instances	194
Detaching network interfaces	195
Exploring how instances get their addresses	196
Watching the DHCP lease cycle	198
Troubleshooting DHCP	203
Exploring how instances retrieve their metadata	204
The DHCP namespace	204
Adding a manual route to 169.254.169.254	205
Using DHCP to inject the route	206
Summary	206
Chapter 8: Managing Security Groups	208
Security groups in OpenStack	208
An introduction to iptables	210
Using ipset	212
Working with security groups	213
Managing security groups in the CLI	213
Creating security groups in the CLI	214
Deleting security groups in the CLI	215
Listing security groups in the CLI	215
Showing the details of a security group in the CLI	215
Updating security groups in the CLI	216
Creating security group rules in the CLI	216
Deleting security group rules in the CLI	217
Listing security group rules in the CLI	218
Showing the details of a security group rule in the CLI	218
Applying security groups to instances and ports	218
Removing security groups from instances and ports in the CLI	219
Implementing security group rules	220
Stepping through the chains	222
Working with security groups in the dashboard	227
Creating a security group	228
Managing security group rules	229
Applying security groups to instances	230
Disabling port security	232
Configuring Neutron	233
Disabling port security for all ports on a network	234
Modifying port security on an individual port	236
Allowed address pairs	236
Summary	237
Chapter 9: Role-Based Access Control	238
Working with access control policies	239
Managing access control policies in the CLI	239
Creating access control policies in the CLI	239
Deleting access control policies in the CLI	240

Listing access control policies in the CLI	240
Showing the details of an access control policy in the CLI	241
Updating access control policies in the CLI	241
Applying RBAC policies to projects	241
Creating projects and users	242
Creating a network to share	244
Creating a policy	245
Viewing the policy in action	246
Creating policies for external networks	247
Summary	249
Chapter 10: Creating Standalone Routers with Neutron	250
Routing traffic in the cloud	250
Installing and configuring the Neutron L3 agent	252
Defining an interface driver	252
Enabling the metadata proxy	253
Setting the agent mode	253
Enabling the router service plugin	253
Enabling router management in the dashboard	254
Restarting services	254
Router management in the CLI	255
Creating routers in the CLI	256
Listing routers in the CLI	256
Displaying router attributes in the CLI	257
Updating router attributes in the CLI	257
Working with router interfaces in the CLI	257
Attaching internal interfaces to routers	258
Attaching a gateway interface to a router	258
Listing interfaces attached to routers	259
Deleting internal interfaces	259
Clearing the gateway interface	259
Deleting routers in the CLI	260
Network address translation	260
Floating IP addresses	260
Floating IP management	261
Creating floating IPs in the CLI	261
Associating floating IPs with ports in the CLI	262
Listing floating IPs in the CLI	263
Displaying floating IP attributes in the CLI	264
Disassociating floating IPs in the CLI	264
Deleting floating IPs in the CLI	264
Demonstrating traffic flow from an instance to the internet	265
Setting the foundation	265
Creating an external provider network	267
Creating a Neutron router	269
Attaching the router to an external network	269

Identifying the L3 agent and namespace	270
Testing gateway connectivity	273
Creating an internal network	273
Attaching the router to the internal network	276
Creating instances	277
Verifying instance connectivity	279
Observing default NAT behavior	283
Assigning floating IPs	284
Reassigning floating IPs	288
Router management in the dashboard	291
Creating a router in the dashboard	291
Attaching internal interfaces in the dashboard	292
Viewing the network topology in the dashboard	294
Associating floating IPs to instances in the dashboard	295
Disassociating floating IPs in the dashboard	296
Summary	297
Chapter 11: Router Redundancy Using VRRP	298
Using keepalived and VRRP to provide redundancy	298
VRRP groups	299
VRRP priority	300
VRRP working mode	300
Preemptive	301
Non-preemptive	301
VRRP timers	301
Advertisement interval timer	301
Preemption delay timer	302
Networking of highly available routers	302
Dedicated HA network	302
Limitations	303
Virtual IP	303
Determining the master router	305
Installing and configuring additional L3 agents	306
Defining an interface driver	306
Setting the agent mode	306
Restarting the Neutron L3 agent	307
Configuring Neutron	307
Working with highly available routers	308
Creating highly-available routers	309
Deleting highly-available routers	309
Decomposing a highly available router	309
Examining the keepalived configuration	314
Executing a failover	315
Summary	316
Chapter 12: Distributed Virtual Routers	317

Distributing routers across the cloud	317
Installing and configuring Neutron components	319
Installing additional L3 agents	320
Defining an interface driver	320
Enabling distributed mode	320
Setting the agent mode	320
Configuring Neutron	321
Restarting the Neutron L3 and Open vSwitch agent	322
Managing distributed virtual routers	323
Creating distributed virtual routers	323
Routing east-west traffic between instances	324
Reviewing the topology	324
Plumbing it up	327
Distributing router ports	327
Making it work	328
Demonstrating traffic between instances	329
Centralized SNAT	335
Reviewing the topology	335
Using the routing policy database	339
Tracing a packet through the SNAT namespace	340
Floating IPs through distributed virtual routers	343
Introducing the FIP namespace	343
Tracing a packet through the FIP namespace	345
Sending traffic from an instance with a floating IP	345
Returning traffic to the floating IP	349
Using proxy ARP	349
Summary	351
Chapter 13: Load Balancing Traffic to Instances	352
Fundamentals of load balancing	353
Load balancing algorithms	354
Monitoring	355
Session persistence	355
Integrating load balancers into the network	356
Network namespaces	359
Installing LBaaS v2	359
Configuring the Neutron LBaaS agent service	359
Defining an interface driver	360
Defining a device driver	360
Defining a user group	360
Configuring Neutron	361
Defining a service plugin	361
Defining a service provider	361
Updating the database schema	362
Restarting the Neutron LBaaS agent and API service	362
Load balancer management in the CLI	363

Managing load balancers in the CLI	363
Creating load balancers in the CLI	364
Deleting load balancers in the CLI	364
Listing load balancers in the CLI	365
Showing load balancer details in the CLI	365
Showing load balancer statistics in the CLI	365
Showing the load balancer's status in the CLI	365
Updating a load balancer in the CLI	366
Managing pools in the CLI	366
Creating a pool in the CLI	366
Deleting a pool in the CLI	367
Listing pools in the CLI	368
Showing pool details in the CLI	368
Updating a pool in the CLI	368
Managing pool members in the CLI	369
Creating pool members in the CLI	369
Deleting pool members	370
Listing pool members	370
Showing pool member details	370
Updating a pool member	371
Managing health monitors in the CLI	371
Creating a health monitor in the CLI	372
Deleting a health monitor in the CLI	373
Listing health monitors in the CLI	374
Showing health monitor details	374
Updating a health monitor	374
Managing listeners in the CLI	374
Creating listeners in the CLI	375
Deleting listeners in the CLI	376
Listing listeners in the CLI	376
Showing listener details in the CLI	376
Updating a listener in the CLI	376
Building a load balancer	377
Creating a load balancer	378
Creating a pool	379
Creating pool members	380
Creating a health monitor	382
Creating a listener	384
The LBaaS network namespace	385
Confirming load balancer functionality	386
Observing health monitors	387
Connecting to the virtual IP externally	388
Load balancer management in the dashboard	389
Creating a load balancer in the dashboard	391
Assigning a floating IP to the load balancer	394
Summary	396
Chapter 14: Advanced Networking Topics	397

VLAN-aware VMs	397
Configuring the trunk plugin	398
Defining the workflow	399
Managing trunks in the CLI	401
Creating trunks in the CLI	401
Deleting trunks in the CLI	402
Listing trunks in the CLI	402
Showing trunk details in the CLI	402
Updating a trunk in the CLI	403
Building a trunk	403
Creating the parent port	403
Creating a sub-port	404
Creating a trunk	405
Booting an instance with a trunk	406
Configuring the instance	407
Reviewing the network plumbing	410
BGP dynamic routing	411
Prefix advertisement requirements	413
Operations with distributed virtual routers	413
Configuring BGP dynamic routing	413
Installing the agent	414
Configuring the agent	414
Restarting services	414
Managing BGP speakers in the CLI	415
Network availability zones	416
Configuring network availability zones	416
Scheduling routers to availability zones	418
Scheduling DHCP services to availability zones	418
Summary	419
Other Books You May Enjoy	421
Index	425

Preface

OpenStack is open source software for building public and private clouds as well as privately hosted software defined infrastructure services. In the fall of 2017, the OpenStack Foundation released the 16th version of OpenStack, known as Pike, to the public. Since its introduction as an open source project in 2010 by NASA and Rackspace, OpenStack has undergone significant improvements in its features and functionality thanks to developers and operators worldwide. Their hard work has resulted in production-ready cloud software that powers workloads of all sizes throughout the world.

In 2012, the Folsom release of OpenStack introduced a standalone networking component known then as Quantum. Long since renamed Neutron, the networking component of OpenStack provides cloud operators and users with an API used to create and manage network resources in the cloud. Neutron's extensible framework allows for third-party plugins and additional network services, such as load balancers, firewalls, and virtual private networks, to be deployed and managed.

As an architect and operator of hundreds of OpenStack-based private clouds since 2012, I have seen much of what OpenStack has to offer in terms of networking capabilities. In this book, I have condensed what I feel are its most valuable and production-ready features to date. Throughout this book, we will take a look at a few common network and service architectures and lay a foundation for deploying and managing OpenStack Networking that can help you develop and sharpen your skills as an OpenStack cloud operator.

Who this book is for

This book is geared towards OpenStack cloud administrators or operators with a novice to intermediate level of experience in managing OpenStack-based clouds who are looking to build or enhance their cloud using the networking service known as Neutron. By laying down a basic installation of OpenStack based on the upstream documentation found at docs.openstack.org, the reader should be able to follow the examples laid out in the book to obtain a functional understanding of the various components of OpenStack Networking using open source reference architectures.

What this book covers

Chapter 1, Introduction to OpenStack Networking, introduces OpenStack Networking along with supported networking technologies and examples of how to architect the physical network to support an OpenStack cloud.

Chapter 2, Installing OpenStack, provides instructions to install the core components of the Pike release of OpenStack on the Ubuntu 16.04 LTS operating system, including Keystone, Glance, Nova, and Horizon.

Chapter 3, Installing Neutron, explains how to install the Neutron networking components of OpenStack. We will also cover the internal architecture of Neutron, including the use of agents and plugins to orchestrate network connectivity.

Chapter 4, Virtual Network Infrastructure Using Linux Bridges, helps you to install and configure the ML2 plugin to support the Linux bridge mechanism driver and agent, and demonstrates how Linux bridges can be used to connect instances to the network.

Chapter 5, Building a Virtual Switching Infrastructure Using Open vSwitch, helps you to install and configure the ML2 plugin to support the Open vSwitch mechanism driver and agent, and demonstrates how Open vSwitch can be used to connect instances to the network.

Chapter 6, Building Networks with Neutron, walks you through creating networks, subnets, subnet pools, and ports.

Chapter 7, Attaching Instances to Networks, demonstrates attaching instances to networks and explores the process of obtaining DHCP leases and metadata.

Chapter 8, Managing Security Groups, examines the use of iptables to secure instance traffic at the compute node and walks you through creating and managing security groups and associated rules.

Chapter 9, Role-Based Access Control, explains how access control policies can limit the use of certain network resources to groups of projects.

Chapter 10, Creating Standalone Routers with Neutron, walks you through creating standalone virtual routers and attaching them to networks, applying floating IPs to instances, and following the flow of traffic through a router to an instance.

Chapter 11, Router Redundancy Using VRRP, explores the Virtual Routing Redundancy Protocol and its use in providing highly-available virtual routers.

Chapter 12, Distributed Virtual Routers, walks you through creating and managing virtual routers that are distributed across compute nodes for better scale.

Chapter 13, Load Balancing Traffic to Instances, explores the fundamental components of a load balancer in Neutron, including listeners, pools, pool members, and monitors, and walks you through creating and integrating a virtual load balancer into the network.

Chapter 14, Advanced Networking Topics, looks at other advanced networking features, including VLAN-aware VM functionality that allows virtual machine instances to apply 802.1q VLAN tags to traffic, BGP Speaker functionality that provides dynamic routing to project routers, and network availability zone functionality that can be used to separate critical networking components such as DHCP and L3 agents into zones.

To get the most out of this book

This book assumes a moderate level of networking experience, including experience with Linux networking configurations as well as physical switch and router configurations. While this book walks the reader through a basic installation of OpenStack, little time is spent on services other than Neutron. Therefore, it is important that the reader has a basic understanding of OpenStack and its general configuration prior to configuring OpenStack networking.

In this book, the following operating system is required:

- Ubuntu 16.04 LTS

The following software is needed:

- OpenStack Pike (2017.2)

Internet connectivity is required to install OpenStack packages and to make use of the example architectures in the book. While virtualization software such as VirtualBox or VMware can be used to simulate servers and the network infrastructure, this book assumes that OpenStack is installed on physical hardware and that a physical network infrastructure is in place.

In the event that the OpenStack installation procedure documented in this book is no longer current, refer to the installation guide at docs.openstack.org for instructions on installing the latest version of OpenStack.

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Learning-OpenStack-Networking-Third-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here:

<https://www.packtpub.com/sites/default/files/downloads/LearningOpenStackNetworkingThirdEdition.pdf>.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `OPENSTACK_KEYSTONE_DEFAULT_ROLE` setting in the `/etc/openstack-dashboard/local_settings.py` file must also be modified before the dashboard can be used."

A block of code is set as follows:

```
[DEFAULT]
...
my_ip = 10.254.254.101
vncserver_proxyclient_address = 10.254.254.101
vnc_enabled = True
vncserver_listen = 0.0.0.0
novncproxy_base_url = http://controller01:6080/vnc_auto.html
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
nova boot --flavor <FLAVOR_ID> --image <IMAGE_ID> \
--nic net-id=<NETWORK_ID>--security-group <SECURITY_GROUP_ID> \
INSTANCE_NAME
```

Any command-line input or output is written as follows:

```
# systemctl restart nova-api
# systemctl restart neutron-server
```

Bold: New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy: Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

1 Introduction to OpenStack Networking

In today's data centers, networks are composed of more devices than ever before. Servers, switches, routers, storage systems, and security appliances that once consumed rows and rows of data center space now exist as virtual machines and virtual network appliances. These devices place a large strain on traditional network management systems, as they are unable to provide a scalable and automated approach to managing next-generation networks. Users now expect more control and flexibility of the infrastructure with quicker provisioning, all of which OpenStack promises to deliver.

This chapter will introduce many features that OpenStack Networking provides, as well as various network architectures supported by OpenStack. Some topics that will be covered include the following:

- Features of OpenStack Networking
- Physical infrastructure requirements
- Service separation

What is OpenStack Networking?

OpenStack Networking is a pluggable, scalable, and API-driven system to manage networks in an OpenStack-based cloud. Like other core OpenStack components, OpenStack Networking can be used by administrators and users to increase the value and maximize the utilization of existing data center resources.

Neutron, the project name for the OpenStack Networking service, complements other core OpenStack services such as Compute (Nova), Image (Glance), Identity (Keystone), Block (Cinder), Object (Swift), and Dashboard (Horizon) to provide a complete cloud solution.

OpenStack Networking exposes an application programmable interface (API) to users and passes requests to the configured network plugins for additional processing. Users are able to define network connectivity in the cloud, and cloud operators are allowed to leverage different networking technologies to enhance and power the cloud.

OpenStack Networking services can be split between multiple hosts to provide resiliency and redundancy, or they can be configured to operate on a single node. Like many other OpenStack services, Neutron requires access to a database for persistent storage of the network configuration. A simplified example of the architecture can be seen here:

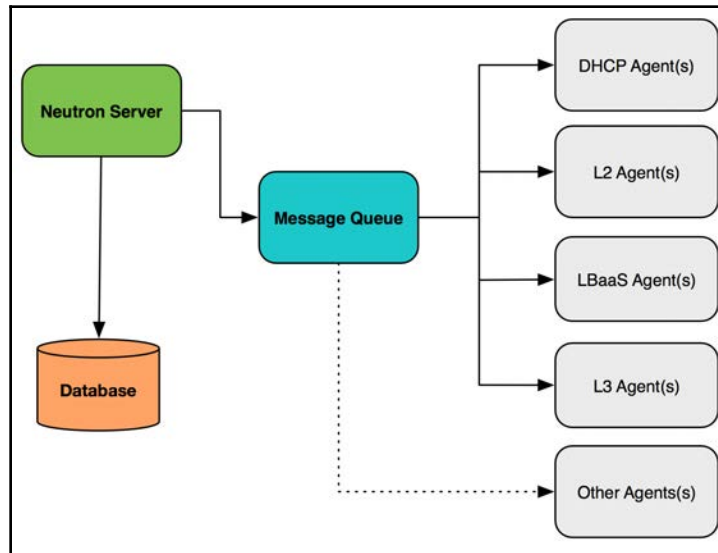


Figure 1.1

In figure 1.1, the **Neutron server** connects to a **database** where the logical network configuration persists. The Neutron server can take API requests from users and services and communicate with agents via a **message queue**. In a typical environment, network agents will be scattered across controller and compute nodes and perform duties on their respective node.

Features of OpenStack Networking

OpenStack Networking includes many technologies you would find in the data center, including switching, routing, load balancing, firewalling, and virtual private networks.

These features can be configured to leverage open source or commercial software and provide a cloud operator with all the tools necessary to build a functional and self-contained cloud networking stack. OpenStack Networking also provides a framework for third-party vendors to build on and enhance the capabilities of the cloud.

Switching

A **virtual switch** is defined as a software application or service that connects virtual machines to virtual networks at the data link layer of the OSI model, also known as layer 2. Neutron supports multiple virtual switching platforms, including Linux bridges provided by the `bridge` kernel module and Open vSwitch. Open vSwitch, also known as OVS, is an open source virtual switch that supports standard management interfaces and protocols, including NetFlow, SPAN, RSPAN, LACP, and 802.1q VLAN tagging. However, many of these features are not exposed to the user through the OpenStack API. In addition to VLAN tagging, users can build overlay networks in software using L2-in-L3 tunneling protocols, such as GRE or VXLAN. Virtual switches can be used to facilitate communication between instances and devices outside the control of OpenStack, which include hardware switches, network firewalls, storage devices, bare-metal servers, and more.

Additional information on the use of Linux bridges and Open vSwitch as switching platforms for OpenStack can be found in [Chapter 4, *Virtual Network Infrastructure Using Linux Bridges*](#), and [Chapter 5, *Building a Virtual Switching Infrastructure Using Open vSwitch*](#), respectively.

Routing

OpenStack Networking provides routing and NAT capabilities through the use of IP forwarding, iptables, and network namespaces. Each network namespace has its own routing table, interfaces, and iptables processes that provide filtering and network address translation. By leveraging network namespaces to separate networks, there is no need to worry about overlapping subnets between networks created by users. Configuring a router within Neutron enables instances to interact and communicate with outside networks or other networks in the cloud.

More information on routing within OpenStack can be found in [Chapter 10, *Creating Standalone Routers with Neutron*](#), [Chapter 11, *Router Redundancy Using VRRP*](#), and [Chapter 12, *Distributed Virtual Routers*](#).

Load balancing

First introduced in the Grizzly release of OpenStack, **Load Balancing as a Service (LBaaS v2)** provides users with the ability to distribute client requests across multiple instances or servers. Users can create monitors, set connection limits, and apply persistence profiles to traffic traversing a virtual load balancer. OpenStack Networking is equipped with a plugin for LBaaS v2 that utilizes HAProxy in the open source reference implementation, but plugins are available that manage virtual and physical load-balancing appliances from third-party network vendors.

More information on the use of load balancers within Neutron can be found in [Chapter 13, Load Balancing Traffic to Instances](#).

Firewalling

OpenStack Networking provides two API-driven methods of securing network traffic to instances: security groups and **Firewall as a Service (FWaaS)**. Security groups find their roots in nova-network, the original networking stack for OpenStack built in to the Compute service, and are based on Amazon's EC2 security groups. When using security groups in OpenStack, instances are placed into groups that share common functionality and rule sets. In a reference implementation, security group rules are implemented at the instance port level using drivers that leverage iptables or OpenFlow. Security policies built using FWaaS are also implemented at the port level, but can be applied to ports of routers as well as instances. The original FWaaS v1 API implemented firewall rules inside Neutron router namespaces, but that behavior has been removed in the v2 API.

More information on securing instance traffic can be found in [Chapter 8, Managing Security Groups](#). The use of FWaaS is outside the scope of this book.

Virtual private networks

A **virtual private network (VPN)** extends a private network across a public network such as the internet. A VPN enables a computer to send and receive data across public networks as if it were directly connected to the private network. Neutron provides a set of APIs to allow users to create IPSec-based VPN tunnels from Neutron routers to remote gateways when using the open source reference implementation. The use of VPN as a Service is outside the scope of this book.

Network functions virtualization

Network functions virtualization (NFV) is a network architecture concept that proposes virtualizing network appliances used for various network functions. These functions include intrusion detection, caching, gateways, WAN accelerators, firewalls, and more. Using SR-IOV, instances are no longer required to use para-virtualized drivers or to be connected to virtual bridges within the host. Instead, the instance is attached to a Neutron port that is associated with a **virtual function (VF)** in the NIC, allowing the instance to access the NIC hardware directly. Configuring and implementing SR-IOV with Neutron is outside the scope of this book.

OpenStack Networking resources

OpenStack gives users the ability to create and configure networks and subnets and instruct other services, such as Compute, to attach virtual devices to ports on these networks. The Identity service gives cloud operators the ability to segregate users into projects. OpenStack Networking supports project-owned resources, including each project having multiple private networks and routers. Projects can be left to choose their own IP addressing scheme, even if those addresses overlap with other project networks, or administrators can place limits on the size of subnets and addresses available for allocation.

There are two types of networks that can be expressed in OpenStack:

- **Project/tenant network:** A virtual network created by a project or administrator on behalf of a project. The physical details of the network are not exposed to the project.
- **Provider network:** A virtual network created to map to a physical network. Provider networks are typically created to enable access to physical network resources outside of the cloud, such as network gateways and other services, and usually map to VLANs. Projects can be given access to provider networks.



The terms *project* and *tenant* are used interchangeably within the OpenStack community, with the former being the newer and preferred nomenclature.

A **project network** provides connectivity to resources in a project. Users can create, modify, and delete project networks. Each project network is isolated from other project networks by a boundary such as a VLAN or other segmentation ID. A **provider network**, on the other hand, provides connectivity to networks outside of the cloud and is typically created and managed by a cloud administrator.

The primary differences between project and provider networks can be seen during the network provisioning process. Provider networks are created by administrators on behalf of projects and can be dedicated to a particular project, shared by a subset of projects, or shared by all projects. Project networks are created by projects for use by their instances and cannot be shared with all projects, though sharing with certain projects may be accomplished using role-based access control (**RBAC**) policies. When a provider network is created, the administrator can provide specific details that aren't available to ordinary users, including the network type, the physical network interface, and the network segmentation identifier, such as a VLAN ID or VXLAN VNI. Project networks have these same attributes, but users cannot specify them. Instead, they are automatically determined by Neutron.

There are other foundational network resources that will be covered in further detail later in this book, but are summarized in the following table for your convenience:

Resource	Description
Subnet	A block of IP addresses used to allocate ports created on the network.
Port	A connection point for attaching a single device, such as the virtual network interface card (vNIC) of a virtual instance, to a virtual network. Port attributes include the MAC address and the fixed IP address on the subnet.
Router	A virtual device that provides routing between self-service networks and provider networks.
Security group	A set of virtual firewall rules that control ingress and egress traffic at the port level.
DHCP	An agent that manages IP addresses for instances on provider and self-service networks.
Metadata	A service that provides data to instances during boot.

Virtual network interfaces

OpenStack deployments are most often configured to use the libvirt KVM/QEMU driver to provide platform virtualization. When an instance is booted for the first time, OpenStack creates a port for each network interface attached to the instance. A virtual network interface called a **tap interface** is created on the compute node hosting the instance. The tap interface corresponds directly to a network interface within the guest instance and has the properties of the port created in Neutron, including the MAC and IP address. Through the use of a bridge, the host can expose the guest instance to the physical network. Neutron allows users to specify alternatives to the standard tap interface, such as Macvtap and SR-IOV, by defining special attributes on ports and attaching them to instances.

Virtual network switches

OpenStack Networking supports many types of virtual and physical switches, and includes built-in support for Linux bridges and Open vSwitch virtual switches. This book will cover both technologies and their respective drivers and agents.



The terms *bridge* and *switch* are often used interchangeably in the context of OpenStack Networking, and may be used in the same way throughout this book.

Overlay networks

Neutron supports overlay networking technologies that provide network isolation at scale with little to no modification of the underlying physical infrastructure. To accomplish this, Neutron leverages L2-in-L3 overlay networking technologies such as GRE, VXLAN, and GENEVE. When configured accordingly, Neutron builds point-to-point tunnels between all network and compute nodes in the cloud using a predefined interface. These point-to-point tunnels create what is called a **mesh network**, where every host is connected to every other host. A cloud consisting of one combined controller and network node, and three compute nodes, would have a fully meshed overlay network that resembles figure 1.2:

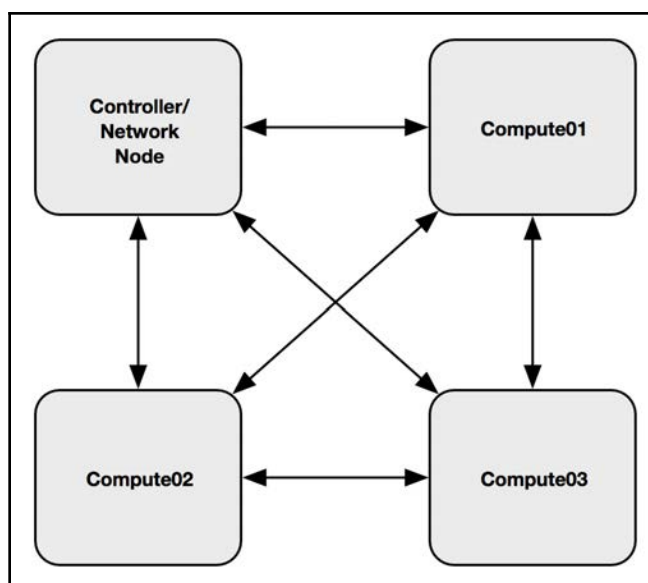


Figure 1.2

Using the overlay network pictured in figure 1.2, traffic between instances or other virtual devices on any given host will travel between layer 3 endpoints on each of the underlying hosts without regard for the layer 2 network beneath them. Due to encapsulation, Neutron routers may be needed to facilitate communication between different project networks as well as networks outside of the cloud.

Virtual Extensible Local Area Network (VXLAN)

This book focuses primarily on VXLAN, an overlay technology that helps address scalability issues with VLANs. VXLAN encapsulates layer 2 Ethernet frames inside layer 4 UDP packets that can be forwarded or routed between hosts. This means that a virtual network can be transparently extended across a large network without any changes to the end hosts. In the case of OpenStack Networking, however, a VXLAN mesh network is commonly constructed only between nodes that exist in the same cloud.

Rather than use VLAN IDs to differentiate between networks, VXLAN uses a VXLAN Network Identifier (VNI) to serve as the unique identifier on a link that potentially carries traffic for tens of thousands of networks, or more. An 802.1q VLAN header supports up to 4,096 unique IDs, whereas a VXLAN header supports approximately 16 million unique IDs. Within an OpenStack cloud, virtual machine instances are unaware that VXLAN is used to forward traffic between hosts. The VXLAN Tunnel Endpoint (VTEP) on the physical node handles the encapsulation and decapsulation of traffic without the instance ever knowing.

Because VXLAN network traffic is encapsulated, many network devices cannot participate in these networks without additional configuration, if at all. As a result, VXLAN networks are effectively isolated from other networks in the cloud and require the use of a Neutron router to provide access to connected instances. More information on creating Neutron routers begins in [Chapter 10, *Creating Standalone Routers with Neutron*](#).

While not as performant as VLAN or flat networks on some hardware, the use of VXLAN is becoming more popular in cloud network architectures where scalability and self-service are major drivers. Newer networking hardware that offers VXLAN offloading capabilities should be leveraged if you are considering implementing VXLAN-based overlay networks in your cloud.

More information on how VXLAN encapsulation works is described in RFC 7348, available at the following URL: <https://tools.ietf.org/html/rfc7348>

Generic Router Encapsulation (GRE)

A **GRE network** is similar to a VXLAN network in that traffic from one instance to another is encapsulated and sent over a layer 3 network. A unique segmentation ID is used to differentiate traffic from other GRE networks. Rather than use UDP as the transport mechanism, GRE uses IP protocol 47. For various reasons, the use of GRE for encapsulating tenant network traffic has fallen out of favor now that VXLAN is supported by both Open vSwitch and Linux Bridge network agents.

More information on how GRE encapsulation works is described in RFC 2784 available at the following URL: <https://tools.ietf.org/html/rfc2784>



As of the Pike release of OpenStack, the Open vSwitch mechanism driver is the only commonly used driver that supports GRE.

Generic Network Virtualization Encapsulation (GENEVE)

GENEVE is an emerging overlay technology that resembles VXLAN and GRE, in that packets between hosts are designed to be transmitted using standard networking equipment without having to modify the client or host applications. Like VXLAN, GENEVE encapsulates packets with a unique header and uses UDP as its transport mechanism. GENEVE leverages the benefits of multiple overlay technologies such as VXLAN, NVGRE, and STT, and may supplant those technologies over time. The Open Virtual Networking (OVN) mechanism driver relies on GENEVE as its overlay technology, which may speed up the adoption of GENEVE in later releases of OpenStack.

Preparing the physical infrastructure

Most OpenStack clouds are made up of physical infrastructure nodes that fit into one of the following four categories:

- **Controller node:** Controller nodes traditionally run the API services for all of the OpenStack components, including Glance, Nova, Keystone, Neutron, and more. In addition, controller nodes run the database and messaging servers, and are often the point of management of the cloud via the Horizon dashboard. Most OpenStack API services can be installed on multiple controller nodes and can be load balanced to scale the OpenStack control plane.
- **Network node:** Network nodes traditionally run DHCP and metadata services and can also host virtual routers when the Neutron L3 agent is installed. In smaller environments, it is not uncommon to see controller and network node services collapsed onto the same server or set of servers. As the cloud grows in size, most network services can be broken out between other servers or installed on their own server for optimal performance.
- **Compute node:** Compute nodes traditionally run a hypervisor such as KVM, Hyper-V, or Xen, or container software such as LXC or Docker. In some cases, a compute node may also host virtual routers, especially when Distributed Virtual Routing (DVR) is configured. In proof-of-concept or test environments, it is not uncommon to see controller, network, and compute node services collapsed onto the same machine. This is especially common when using DevStack, a software package designed for developing and testing OpenStack code. All-in-one installations are not recommended for production use.
- **Storage node:** Storage nodes are traditionally limited to running software related to storage such as Cinder, Ceph, or Swift. Storage nodes do not usually host any type of Neutron networking service or agent and will not be discussed in this book.

When Neutron services are broken out between many hosts, the layout of services will often resemble the following:

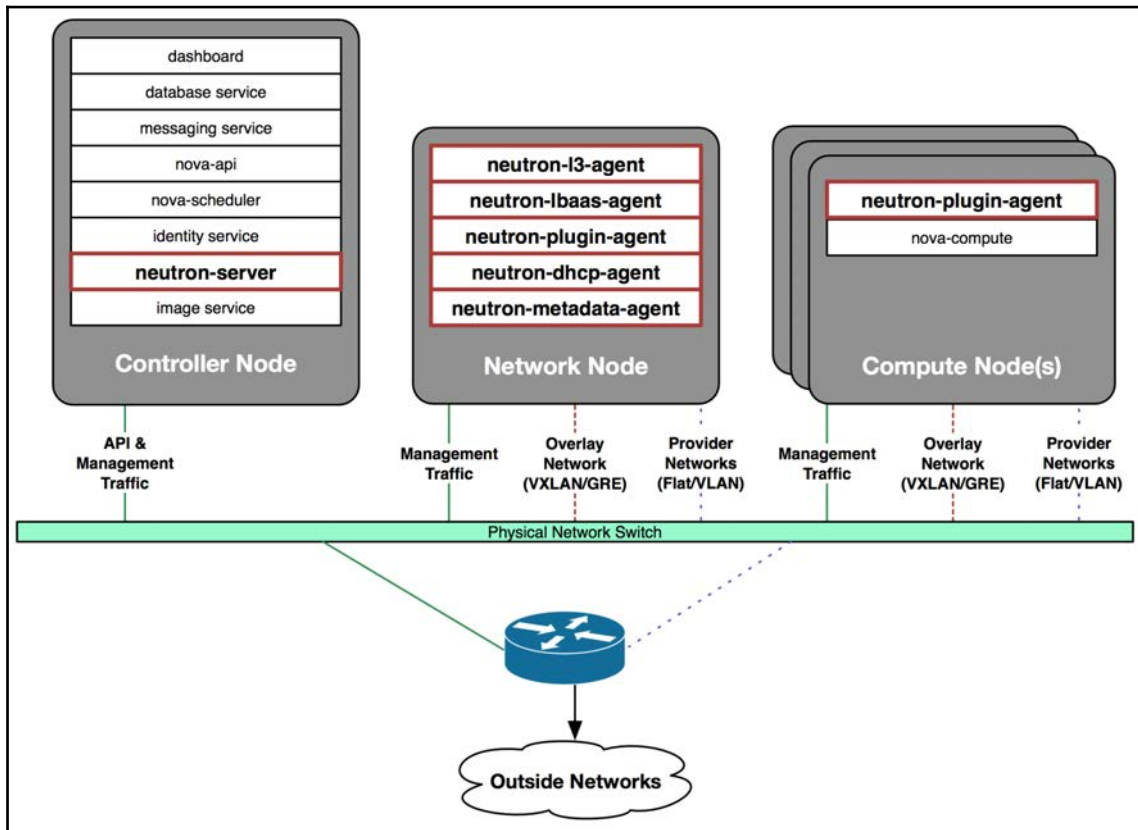


Figure 1.3

In figure 1.3, the neutron API service `neutron-server` is installed on the **Controller node**, while Neutron agents responsible for implementing certain virtual networking resources are installed on a dedicated network node. Each compute node hosts a network plugin agent responsible for implementing the network plumbing on that host. Neutron supports a highly available API service with a shared database backend, and it is recommended that the cloud operator load balances traffic to the Neutron API service when possible. Multiple DHCP, metadata, L3, and LBaaS agents should be implemented on separate network nodes whenever possible. Virtual networks, routers, and load balancers can be scheduled to one or more agents to provide a basic level of redundancy when an agent fails. Neutron even includes a built-in scheduler that can detect failure and reschedule certain resources when a failure is detected.

Configuring the physical infrastructure

Before the installation of OpenStack can begin, the physical network infrastructure must be configured to support the networks needed for an operational cloud. In a production environment, this will likely include a dedicated management VLAN used for server management and API traffic, a VLAN dedicated to overlay network traffic, and one or more VLANs that will be used for provider and VLAN-based project networks. Each of these networks can be configured on separate interfaces, or they can be collapsed onto a single interface if desired.

The reference architecture for OpenStack Networking defines at least four distinct types of traffic that will be seen on the network:

- Management
- API
- External
- Guest

These traffic types are often categorized as control plane or data plane, depending on the purpose, and are terms used in networking to describe the purpose of the traffic. In this case, **control plane** traffic is used to describe traffic related to management, API, and other non-VM related traffic. **Data plane** traffic, on the other hand, represents traffic generated by, or directed to, virtual machine instances.

Although I have taken the liberty of splitting out the network traffic onto dedicated interfaces in this book, it is not necessary to do so to create an operational OpenStack cloud. In fact, many administrators and distributions choose to collapse multiple traffic types onto single or bonded interfaces using VLAN tagging. Depending on the chosen deployment model, the administrator may spread networking services across multiple nodes or collapse them onto a single node. The security requirements of the enterprise deploying the cloud will often dictate how the cloud is built. The various network and service configurations will be discussed in the upcoming sections.

Management network

The **management network**, also referred to as the **internal network** in some distributions, is used for internal communication between hosts for services such as the messaging service and database service, and can be considered as part of the control plane.

All hosts will communicate with each other over this network. In many cases, this same interface may be used to facilitate image transfers between hosts or some other bandwidth-intensive traffic. The management network can be configured as an isolated network on a dedicated interface or combined with another network as described in the following section.

API network

The **API network** is used to expose OpenStack APIs to users of the cloud and services within the cloud and can be considered as part of the control plane. Endpoint addresses for API services such as Keystone, Neutron, Glance, and Horizon are procured from the API network.

It is common practice to utilize a single interface and IP address for API endpoints and management access to the host itself over SSH. A diagram of this configuration is provided later in this chapter.



It is recommended, though not required, that you physically separate management and API traffic from other traffic types, such as storage traffic, to avoid issues with network congestion that may affect operational stability.

External network

An **external network** is a provider network that provides Neutron routers with external network access. Once a router has been configured and attached to the external network, the network becomes the source of floating IP addresses for instances and other network resources attached to the router. IP addresses in an external network are expected to be routable and reachable by clients on a corporate network or the internet. Multiple external provider networks can be segmented using VLANs and trunked to the same physical interface. Neutron is responsible for tagging the VLAN based on the network configuration provided by the administrator. Since external networks are utilized by VMs, they can be considered as part of the data plane.

Guest network

The **guest network** is a network dedicated to instance traffic. Options for guest networks include local networks restricted to a particular node, flat, or VLAN-tagged networks, or virtual overlay networks made possible with GRE, VXLAN, or GENEVE encapsulation. For more information on guest networks, refer to [Chapter 6, *Building Networks with Neutron*](#). Since guest networks provide connectivity to VMs, they can be considered part of the data plane.

The physical interfaces used for external and guest networks can be dedicated interfaces or ones that are shared with other types of traffic. Each approach has its benefits and drawbacks, and they are described in more detail later in this chapter. In the next few chapters, I will define networks and VLANs that will be used throughout the book to demonstrate the various components of OpenStack Networking. Generic information on the configuration of switch ports, routers, or firewalls will also be provided.

Physical server connections

The number of interfaces needed per host is dependent on the purpose of the cloud, the security and performance requirements of the organization, and the cost and availability of hardware. A single interface per server that results in a combined control and data plane is all that is needed for a fully operational OpenStack cloud. Many organizations choose to deploy their cloud this way, especially when port density is at a premium, the environment is simply used for testing, or network failure at the node level is a non-impacting event. When possible, however, it is recommended that you split control and data traffic across multiple interfaces to reduce the chances of network failure.

Single interface

For hosts using a single interface, all traffic to and from instances as well as internal OpenStack, SSH management, and API traffic traverse the same physical interface. This configuration can result in severe performance penalties, as a service or guest can potentially consume all available bandwidth. A single interface is recommended only for non-production clouds.

The following table demonstrates the networks and services traversing a single interface over multiple VLANs:

Service/function	Purpose	Interface	VLAN
SSH	Host management	eth0	10
APIs	Access to OpenStack APIs	eth0	15
Overlay network	Used to tunnel overlay (VXLAN, GRE, GENEVE) traffic between hosts	eth0	20
Guest/external network(s)	Used to provide access to external cloud resources and for VLAN-based project networks	eth0	Multiple

Multiple interfaces

To reduce the likelihood of guest traffic impacting management traffic, segregation of traffic between multiple physical interfaces is recommended. At a minimum, two interfaces should be used: one that serves as a dedicated interface for management and API traffic (control plane), and another that serves as a dedicated interface for external and guest traffic (data plane). Additional interfaces can be used to further segregate traffic, such as storage.

The following table demonstrates the networks and services traversing two interfaces with multiple VLANs:

Service/function	Purpose	Interface	VLAN
SSH	Host management	eth0	10
APIs	Access to OpenStack APIs	eth0	15
Overlay network	Used to tunnel overlay (VXLAN, GRE, GENEVE) traffic between hosts	eth1	20
Guest/external network(s)	Used to provide access to external cloud resources and for VLAN-based project networks	eth1	Multiple

Bonding

The use of multiple interfaces can be expanded to utilize bonds instead of individual network interfaces. The following common bond modes are supported:

- **Mode 1 (active-backup):** Mode 1 bonding sets all interfaces in the bond to a backup state while one interface remains active. When the active interface fails, a backup interface replaces it. The same MAC address is used upon failover to avoid issues with the physical network switch. Mode 1 bonding is supported by most switching vendors, as it does not require any special configuration on the switch to implement.

- **Mode 4 (active-active):** Mode 4 bonding involves the use of **aggregation groups**, a group in which all interfaces share an identical configuration and are grouped together to form a single logical interface. The interfaces are aggregated using the IEEE 802.3ad Link Aggregation Control Protocol (LACP). Traffic is load balanced across the links using methods negotiated by the physical node and the connected switch or switches. The physical switching infrastructure *must* be capable of supporting this type of bond. While some switching platforms require that multiple links of an LACP bond be connected to the same switch, others support technology known as **Multi-Chassis Link Aggregation (MLAG)** that allows multiple physical switches to be configured as a single logical switch. This allows links of a bond to be connected to multiple switches that provide hardware redundancy while allowing users the full bandwidth of the bond under normal operating conditions, all with no additional changes to the server configuration.

Bonding can be configured within the Linux operating system using tools such as `iproute2`, `ifupdown`, and `Open vSwitch`, among others. The configuration of bonded interfaces is outside the scope of OpenStack and this book.



Bonding configurations vary greatly between Linux distributions. Refer to the respective documentation of your Linux distribution for assistance in configuring bonding.

The following table demonstrates the use of two bonds instead of two individual interfaces:

Service/function	Purpose	Interface	VLAN
SSH	Host management	bond0	10
APIs	Access to OpenStack APIs	bond0	15
Overlay network	Used to tunnel overlay (VXLAN, GRE, GENEVE) traffic between hosts	bond1	20
Guest/external network(s)	Used to provide access to external cloud resources and for VLAN-based project networks	bond1	Multiple

In this book, an environment will be built using three non-bonded interfaces: one for management and API traffic, one for VLAN-based provider or project networks, and another for overlay network traffic. The following interfaces and VLAN IDs will be used:

Service/function	Purpose	Interface	VLAN
SSH and APIs	Host management and access to OpenStack APIs	eth0 / ens160	10

Overlay network	Used to tunnel overlay (VXLAN, GRE, GENEVE) traffic between hosts	eth1 / ens192	20
Guest/external network(s)	Used to provide access to external cloud resources and for VLAN-based project networks	eth2 / ens224	30,40-43



When an environment is virtualized in VMware, interface names may differ from the standard eth0, eth1, ethX naming convention. The interface names provided in the table reflect the interface naming convention seen on controller and compute nodes that exist as virtual machines, rather than bare-metal machines.

Separating services across nodes

Like other OpenStack services, cloud operators can split OpenStack Networking services across multiple nodes. Small deployments may use a single node to host all services, including networking, compute, database, and messaging. Others might find benefit in using a dedicated controller node and a dedicated network node to handle guest traffic routed through software routers and to offload Neutron DHCP and metadata services. The following sections describe a few common service deployment models.

Using a single controller node

In an environment consisting of a single controller and one or more compute nodes, the controller will likely handle all networking services and other OpenStack services while the compute nodes strictly provide compute resources.

The following diagram demonstrates a **controller node** hosting all OpenStack management and networking services where the Neutron layer 3 agent is not utilized. Two physical interfaces are used to separate management (control plane) and instance (data plane) network traffic:

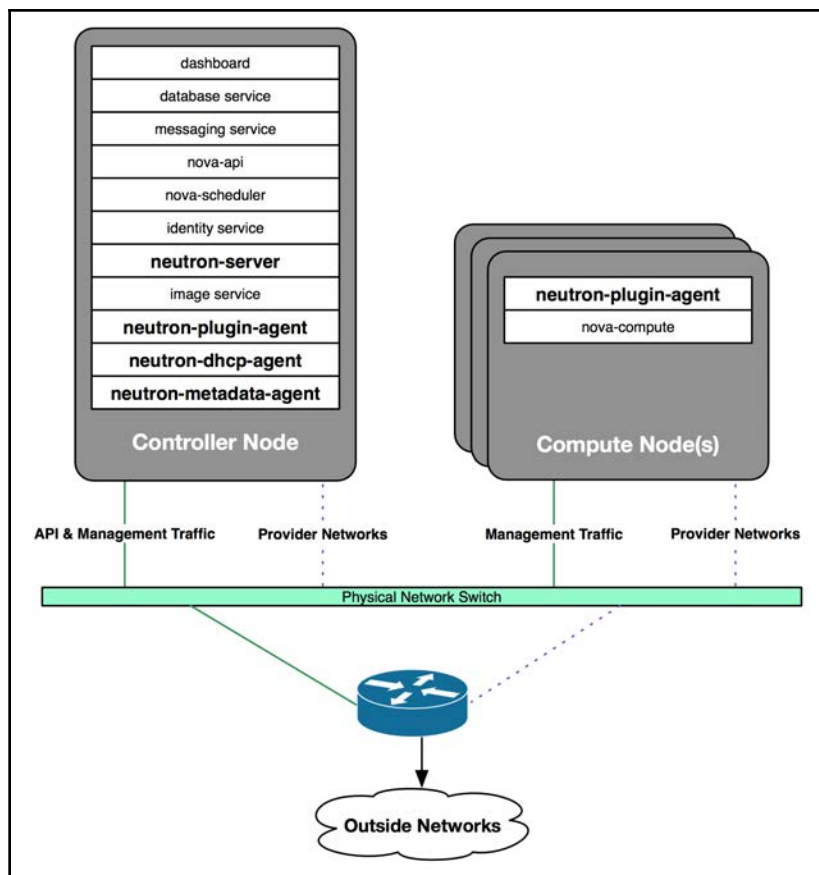


Figure 1.3

The preceding diagram reflects the use of a single combined controller/network node and one or more compute nodes, with Neutron providing only layer 2 connectivity between instances and external gateway devices. An external router is needed to handle routing between network segments.

The following diagram demonstrates a controller node hosting all OpenStack management and networking services, including the Neutron L3 agent. Three physical interfaces are used to provide separate control and data planes:

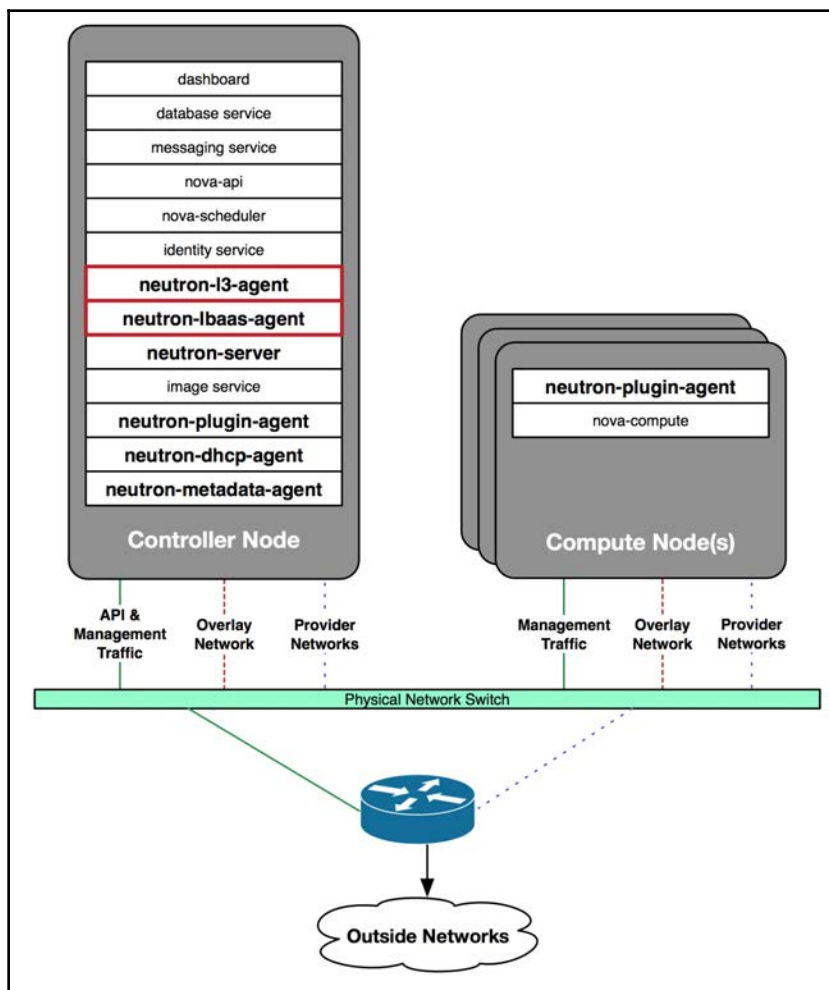


Figure 1.4

The preceding diagram reflects the use of a single combined controller/network node and one or more compute nodes in a network configuration that utilizes the Neutron L3 agent. Software routers created with Neutron reside on the controller node, and handle routing between connected project networks and external provider networks.

Using a dedicated network node

A network node is dedicated to handling most or all the OpenStack networking services, including the L3 agent, DHCP agent, metadata agent, and more. The use of a dedicated **network node** provides additional security and resilience, as the **controller node** will be at less risk of network and resource saturation. Some Neutron services, such as the L3 and DHCP agents and the Neutron API service, can be scaled out across multiple nodes for redundancy and increased performance, especially when distributed virtual routers are used.

The following diagram demonstrates a **network node** hosting all OpenStack networking services, including the Neutron L3, DHCP, metadata, and LBaaS agents. The Neutron API service, however, remains installed on the controller node. Three physical interfaces are used where necessary to provide separate control and data planes:

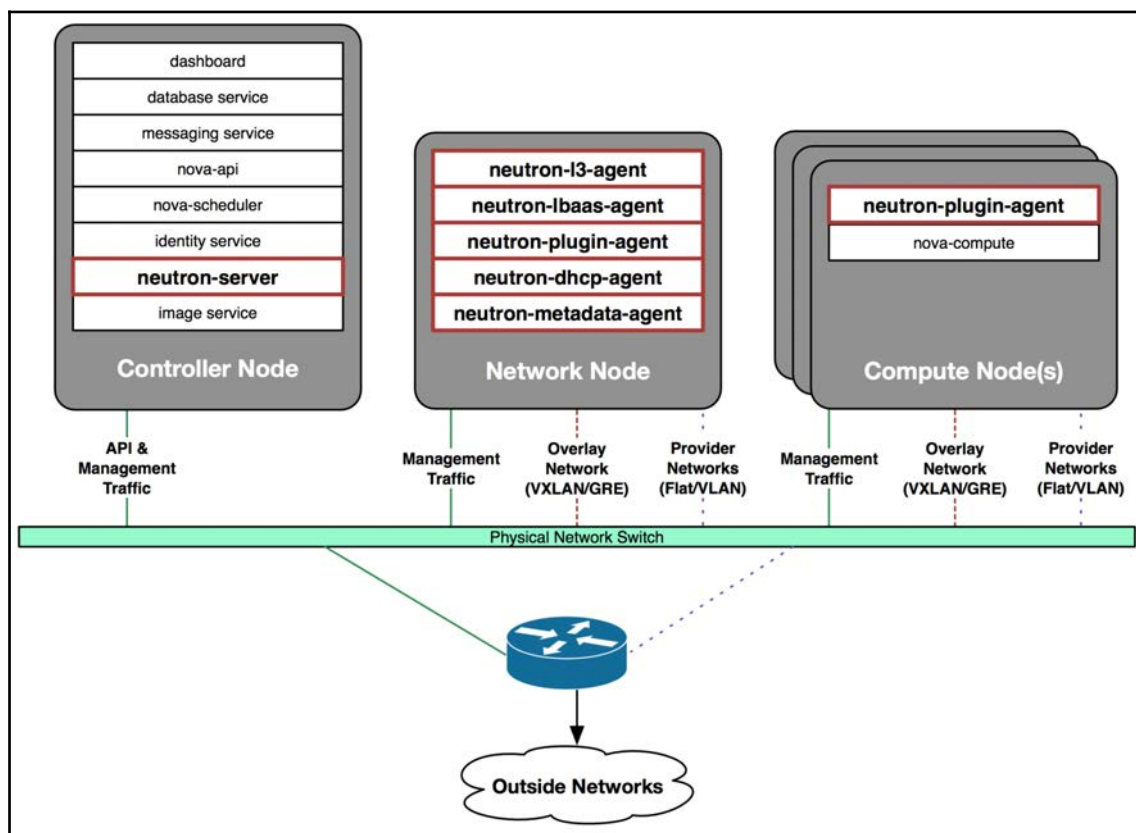


Figure 1.5

The environment built out in this book will be composed of five hosts, including the following:

- A single controller node running all OpenStack network services and the Linux bridge network agent
- A single compute node running the Nova compute service and the Linux bridge network agent
- Two compute nodes running the Nova compute service and the Open vSwitch network agent
- A single network node running the Open vSwitch network agent and the L3 agent

Not all hosts are required should you choose not to complete the exercises described in the upcoming chapters.

Summary

OpenStack Networking offers the ability to create and manage different technologies found in a data center in a virtualized and programmable manner. If the built-in features and reference implementations are not enough, the pluggable architecture of OpenStack Networking allows for additional functionality to be provided by third-party commercial and open source vendors. The security requirements of the organization building the cloud as well as the use cases of the cloud, will ultimately dictate the physical layout and separation of services across the infrastructure nodes.

To successfully deploy Neutron and harness all it has to offer, it is important to have a strong understanding of core networking concepts. In this book, we will cover some fundamental network concepts around Neutron and build a foundation for deploying instances.

In the next chapter, we will begin a package-based installation of OpenStack on the Ubuntu 16.04 LTS operating system. Topics covered include the installation, configuration, and verification of many core OpenStack projects, including Identity, Image, Dashboard, and Compute. The installation and configuration of base OpenStack Networking services, including the Neutron API, can be found in *Chapter 3, Installing Neutron*.

2 Installing OpenStack

Manually installing, configuring, and maintaining OpenStack clouds can be an arduous task. Many vendors provide downloadable cloud software based on OpenStack that provides deployment and management strategies using Chef, Puppet, Ansible, and other tools.

This chapter will take you step by step through a package-based installation of the following OpenStack components on the Ubuntu 16.04 LTS operating system:

- OpenStack Identity (Keystone)
- OpenStack Image Service (Glance)
- OpenStack Compute (Nova)
- OpenStack Dashboard (Horizon)



The installation process documented within this chapter is based on the *OpenStack Installation Guide* found at <http://docs.openstack.org/>. If you wish to install OpenStack on a different operating system, the guides available at that site provide instructions for doing so.

If you'd rather download and install a third-party cloud distribution based on OpenStack, try one of the following URLs:

- OpenStack-Ansible: <https://docs.openstack.org/openstack-ansible/>
- Red Hat RDO: <http://openstack.redhat.com/>
- Kolla: <https://docs.openstack.org/kolla-ansible/latest/>

Once installed, many of the concepts and examples used throughout this book should still apply to the above distributions, but they may require extra effort to implement.

System requirements

OpenStack components are intended to run on standard hardware, ranging from desktop machines to enterprise-grade servers. For optimal performance, the processors of the compute nodes need to support virtualization technologies such as Intel's VT-x or AMD's AMD-V.

This book assumes OpenStack will be installed on servers that meet the following minimum requirements:

Server	Hardware requirements
All	Processor: 64-bit x86 CPU count: 2-4 Memory: 4+ GB RAM Disk space: 32+ GB Network: 3x 1-Gbps network interface cards

While machines that fail to meet these minimum requirements are capable of installation based on the documentation included in this book, these minimums have been set to ensure a successful experience. Additional memory and storage is highly recommended when creating multiple virtual machine instances for demonstration purposes. Virtualization products such as VMware ESXi, VMware Fusion, or VirtualBox may be used in lieu of physical hardware, but will require additional configuration to the environment and to OpenStack that is outside the scope of this book. If virtualizing the environment, consider exposing hardware virtualization extensions for better performance.

Operating system requirements

OpenStack can be installed on the following Linux distributions: CentOS, Red Hat Enterprise Linux, openSUSE, SUSE Linux Enterprise Server, and Ubuntu. This book assumes that the Ubuntu 16.04 LTS server operating system has been installed on all hosts prior to installation of OpenStack. You can find Ubuntu Server at the following URL: <http://www.ubuntu.com/download/server>.

In order to support all of the networking features discussed in this book, the following minimum kernel version is recommended: 4.13.0-45-generic.

Initial network configuration

To understand how networking should initially be configured on each host, please refer to the following diagram:

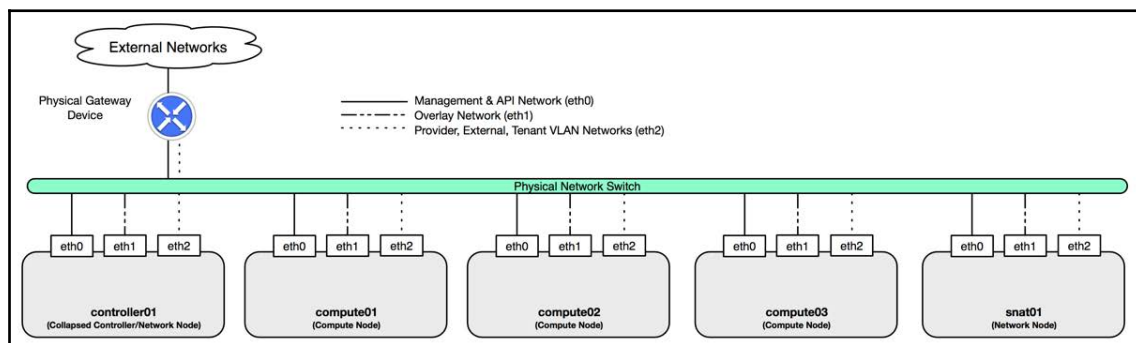


Figure 2.1

In *figure 2.1*, three interfaces are cabled to each host. The `eth0` interface will serve as the management interface for OpenStack services and API access while `eth1` will be used for overlay network traffic between hosts. On the `controller` node, `eth2` will be used for external network traffic to instances through Neutron routers. If VLAN tenant networks are used in lieu of overlay networks, then `eth2` will be configured on the `compute` nodes to support those networks.

At a minimum, the management interface should be configured with an IP address that has outbound access to the internet. Internet access is required to download OpenStack packages from the Ubuntu package repository. Inbound access to the management address of the servers from a trusted network via SSH is recommended.

Example networks

Throughout the book, there will be examples of configuring and using various OpenStack services. The following table provides the VLANs and associated networks used for those services:

VLAN name	VLAN ID	Network
MGMT_NET	10	10.10.0.0/24
OVERLAY_NET	20	10.20.0.0/24
GATEWAY_NET	30	10.30.0.0/24
PROJECT_NET40	40	TBD

PROJECT_NET41	41	TBD
PROJECT_NET42	42	TBD
PROJECT_NET43	43	TBD

The following tables provide the IP addresses and VLAN IDs recommended for each host interface should you choose to follow along with the examples:

Host Name	Interface	IP address	Switchport	VLAN ID
controller01	eth0	10.10.0.100	Access Port	VLAN 10(Untagged)
controller01	eth1	10.20.0.100	Access Port	VLAN 20(Untagged)
controller01	eth2	None	Trunk Port	VLAN 30, 40-44(Tagged)
Host Name	Interface	IP address	Switchport	VLAN ID
compute01	eth0	10.10.0.101	Access Port	VLAN 10(Untagged)
compute01	eth1	10.20.0.101	Access Port	VLAN 20(Untagged)
compute01	eth2	None	Trunk Port	VLAN 30, 40-44 (Tagged)
Host Name	Interface	IP address	Switchport	VLAN ID

compute02	eth0	10.10.0.102	Access Port	VLAN 10(Untagged)
compute02	eth1	10.20.0.102	Access Port	VLAN 20(Untagged)
compute02	eth2	None	Trunk Port	VLAN 30, 40-44 (Tagged)
Host Name	Interface	IP address	Switchport	VLAN ID
compute03	eth0	10.10.0.103	Access Port	VLAN 10(Untagged)
compute03	eth1	10.20.0.103	Access Port	VLAN 20(Untagged)
compute03	eth2	None	Trunk Port	VLAN 30, 40-44 (Tagged)
Host Name	Interface	IP address	Switchport	VLAN ID
snat01	eth0	10.10.0.104	Access Port	VLAN 10(Untagged)
snat01	eth1	10.20.0.104	Access Port	VLAN 20(Untagged)
snat01	eth2	None	Trunk Port	VLAN 30, 40-44 (Tagged)

To avoid loss of connectivity due to an interface misconfiguration or other mishap, out-of-band management access to the servers or some other mechanism is highly recommended.

Interface configuration

Ubuntu 16.04 LTS uses configuration files found in `/etc/network/interfaces.d/` or the file `/etc/network/interfaces` that describe how network interfaces should be configured.

Using a text editor, update the network interface file on each host as follows.

For controller01:

```
auto eth0
iface eth0 inet static
    address 10.10.0.100
    netmask 255.255.255.0
    gateway 10.10.0.1
    dns-nameserver 8.8.8.8
auto eth1
iface eth1 inet static
    address 10.20.0.100
    netmask 255.255.255.0
auto eth2
iface eth2 inet manual
```

For compute01:

```
auto eth0
iface eth0 inet static
    address 10.10.0.101
    netmask 255.255.255.0
    gateway 10.10.0.1
    dns-nameserver 8.8.8.8
auto eth1
iface eth1 inet static
    address 10.20.0.101
    netmask 255.255.255.0
auto eth2
iface eth2 inet manual
```

For compute02:

```
auto eth0
iface eth0 inet static
    address 10.10.0.102
```

```
        netmask 255.255.255.0
        gateway 10.10.0.1
    dns-nameserver 8.8.8.8
    auto eth1
    iface eth1 inet static
        address 10.20.0.102
        netmask 255.255.255.0
    auto eth2
    iface eth2 inet manual
```

For compute03:

```
    auto eth0
    iface eth0 inet static
        address 10.10.0.103
        netmask 255.255.255.0
        gateway 10.10.0.1
        dns-nameserver 8.8.8.8
    auto eth1
    iface eth1 inet static
        address 10.20.0.103
        netmask 255.255.255.0
    auto eth2
    iface eth2 inet manual
```

For snat01:

```
    auto eth0
    iface eth0 inet static
        address 10.10.0.104
        netmask 255.255.255.0
        gateway 10.10.0.1
        dns-nameserver 8.8.8.8
    auto eth1
    iface eth1 inet static
        address 10.20.0.104
        netmask 255.255.255.0
    auto eth2
    iface eth2 inet manual
```

The eth2 interface will be used in a network bridge described in further detail in upcoming chapters. To activate the changes, cycle the interfaces using the `ifdown` and `ifup` commands on each node:

```
# ifdown --all; ifup --all
```


For more information on configuring network interfaces, please refer to the Ubuntu man page at the following

URL: <http://manpages.ubuntu.com/manpages/xenial/man5/interfaces.5.html>

Initial steps

Before we can install OpenStack, some work must be done to prepare each system for a successful installation.

Permissions

OpenStack services can be installed either as a root user or as a user with `sudo` permissions. The latter may require the user to be added to the `sudoers` file on each host. For tips on configuring `sudoers`, please visit the following URL: <https://help.ubuntu.com/community/RootSudo>.



For this installation, all commands should be run as the root user unless specified otherwise.

Configuring the OpenStack repository

When installing versions of OpenStack that are newer than what the operating system shipped with, Ubuntu uses the Ubuntu Cloud Archive (UCA). To enable the Cloud Archive repository, update the `apt` cache and download and install the `software-properties-common` package on all hosts:

```
# apt update; apt install software-properties-common
```

Once installed, the OpenStack Pike repository should be added as an `apt` source on all hosts:

```
# add-apt-repository cloud-archive:pike
```

Upgrading the system

Before installing OpenStack, it is imperative that the kernel and other system packages on each node be upgraded to the latest version provided by Ubuntu for the 16.04 LTS release using the Cloud Archive. Issue the following `apt` commands on each node:

```
# apt update
# apt dist-upgrade
```

Setting the hostnames

Before installing OpenStack, ensure that each node in the environment has been configured with its proper hostname. Use the `hostnamectl` command on each host to set the hostname accordingly:

Host	Command
controller01	<code>hostnamectl set-hostname controller01</code>
compute01	<code>hostnamectl set-hostname compute01</code>
compute02	<code>hostnamectl set-hostname compute02</code>
compute03	<code>hostnamectl set-hostname compute03</code>
snat01	<code>hostnamectl set-hostname snat01</code>

To simplify communication between hosts, it is recommended that DNS or a local name resolver be used to resolve hostnames. Using a text editor, update the `/etc/hosts` file on each node to include the management IP address and hostname of all nodes:

```
10.10.0.100 controller01.learningneutron.com controller01
10.10.0.101 compute01.learningneutron.com compute01
10.10.0.102 compute02.learningneutron.com compute02
10.10.0.103 compute03.learningneutron.com compute03
10.10.0.104 snat01.learningneutron.com snat01
```

On each node, use the `hostname -f` command to verify that the fully qualified host name is properly reflected:

```
root@controller01:~# hostname -f
controller01.learningneutron.com
```

Installing and configuring Network Time Protocol

A time synchronization program such as Network Time Protocol (NTP) is a requirement, as OpenStack services depend on consistent and synchronized time between hosts.

For Nova Compute, having synchronized time helps to avoid problems when scheduling VM launches on `compute` nodes. Other services can experience similar issues when the time is not synchronized.

To install `chrony`, an NTP implementation, issue the following commands on all nodes in the environment:

```
# apt install chrony
```

On the `controller` node, add the following line to the `/etc/chrony/chrony.conf` file to allow other hosts in the environment to synchronize their time against the controller:

```
allow 10.10.0.0/24
```

On the other nodes, comment out any `pool` lines in the `/etc/chrony/chrony.conf` file and add the following line to direct them to synchronize their time against the controller:

```
# pool 2.debian.pool.ntp.org offline iburst
server controller01 iburst
```

On each host, restart the `chrony` service:

```
# systemctl restart chrony
```

Rebooting the system

Reboot each host before proceeding with the installation:

```
# reboot
```

Installing OpenStack

The steps in this section document the installation of OpenStack services including Keystone, Glance, Nova, and Horizon on a single controller and three `compute` nodes. Neutron, the OpenStack Networking service, will be installed in the next chapter.

Prepare for the configuration of various services by installing the OpenStack command-line client, `python-openstackclient`, on the controller node with the following command:

```
# apt install python-openstackclient
```

Installing and configuring the MySQL database server

On the controller node, use `apt` to install the MySQL database service and related Python packages:

```
# apt install mariadb-server python-pymysql
```

If prompted, set the password to `openstack`.



Insecure passwords are used throughout the book to simplify the configuration and demonstration of concepts and are not recommended for production environments. Visit <http://www.strongpasswordgenerator.org> to generate strong passwords for your environment.

Once installed, create and edit the `/etc/mysql/mariadb.conf.d/99-openstack.cnf` configuration file. Add the `[mysqld]` block and the `bind-address` definition. Doing so will allow connectivity to MySQL from other hosts in the environment. The value for `bind-address` should be the management IP of the controller node:

```
[mysqld]
bind-address = 10.10.0.100
```

In addition to adding the `bind-address` definition, add the following options to the `[mysqld]` section as well:

```
default-storage-engine = innodb
innodb_file_per_table = on
max_connections = 4096
collation-server = utf8_general_ci
character-set-server = utf8
```

Save and close the file, then restart the `mysql` service:

```
# systemctl restart mysql
```

The MySQL secure installation utility is used to build the default MySQL database and to set a password for the MySQL root user. The following command will begin the MySQL installation and configuration process:

```
# mysql_secure_installation
```

During the MySQL installation process, you will be prompted to enter a password and change various settings. The default root password may not yet be set. When prompted, set a root password of `openstack`. A more secure password suitable for your environment is highly recommended.

Answer `[Y]`es to the remaining questions to exit the configuration process. At this point, the MySQL server has been successfully installed on the `controller` node.

Installing and configuring the messaging server

Advanced Message Queue Protocol (AMQP) is the messaging technology powering most OpenStack-based clouds. Components such as Nova, Cinder, and Neutron communicate internally and between one another using a message bus. The following are instructions for installing RabbitMQ, an AMQP broker.

On the `controller` node, install the messaging server:

```
# apt install rabbitmq-server
```

Add a user to RabbitMQ named `openstack` with a password of `rabbit` as shown in the following command:

```
# rabbitmqctl add_user openstack rabbit
```

Set RabbitMQ permissions to allow configuration, read, and write access for the `openstack` user:

```
# rabbitmqctl set_permissions openstack ".*" ".*" ".*"
```

At this point, the installation and configuration of RabbitMQ is complete.

Installing and configuring memcached

The memcached service is used to cache common data and objects in RAM to reduce the numbers of times they are read from disk and it is used by various OpenStack services.

On the controller node, install memcached:

```
# apt install memcached python-memcache
```

Edit the `/etc/memcached.conf` file and replace the default listener address with the IP address of the controller node:

- From: 127.0.0.1
- To: 10.10.0.100

Restart the memcached service:

```
# systemctl restart memcached
```

Installing and configuring the identity service

Keystone is the identity service for OpenStack and is used to authenticate and authorize users and services in the OpenStack cloud. Keystone should only be installed on the controller node and will be covered in the following sections.

Configuring the database

Using the `mysql` client, create the Keystone database and associated user:

```
# mysql
```

Enter the following SQL statements at the MariaDB `[(none)] >` prompt:

```
CREATE DATABASE keystone;
GRANT ALL PRIVILEGES ON keystone.* TO 'keystone'@'localhost' IDENTIFIED BY
'keystone';
GRANT ALL PRIVILEGES ON keystone.* TO 'keystone'@'%' IDENTIFIED BY
'keystone';
quit;
```

Installing Keystone

Since the Kilo release of OpenStack, the Keystone project has used the Apache HTTP server with `mod_wsgi` to serve requests to the Identity API on ports 5000 and 35357.

Run the following command to install the Keystone packages on the controller node:

```
# apt install keystone apache2 libapache2-mod-wsgi
```

Update the `[database]` section in the `/etc/keystone/keystone.conf` file to configure Keystone to use MySQL as its database. In this installation, the username and password will be `keystone`. You will need to overwrite the existing connection string with the following value on one line:

```
[database]
...
connection = mysql+pymysql://keystone:keystone@controller01/keystone
```

Configuring tokens and drivers

Keystone supports customizable token providers that can be defined within the `[token]` section of the configuration file. Keystone provides UUID, PKI, and Fernet token providers. In this installation, the Fernet token provider will be used. Update the `[token]` section in the `/etc/keystone/keystone.conf` file accordingly:

```
[token]
...
provider = fernet
```

Populate the Keystone database using the `keystone-manage` utility:

```
# su -s /bin/sh -c "keystone-manage db_sync" keystone
```

Initialize the Fernet key repositories with the following commands:

```
# keystone-manage fernet_setup
--keystone-user keystone --keystone-group keystone
# keystone-manage credential_setup
--keystone-user keystone --keystone-group keystone
```

Bootstrap the Identity service

Using the `keystone-manage bootstrap` command, bootstrap the service catalog with Identity endpoints. In this environment, the password for the admin user will be `openstack`:

```
# keystone-manage bootstrap --bootstrap-password openstack
--bootstrap-admin-url http://controller01:35357/v3/
--bootstrap-internal-url http://controller01:5000/v3/
--bootstrap-public-url http://controller01:5000/v3/
--bootstrap-region-id RegionOne
```

Configuring the Apache HTTP server

Using `sed`, add the `ServerName` option to the Apache configuration file that references the short name of the controller node:

```
# sed -i '1s/^/ServerName controller01\n&/' /etc/apache2/apache2.conf
```

Restart the Apache web service for the changes to take effect:

```
# systemctl restart apache2
```

Setting environment variables

To avoid having to provide credentials every time you run an OpenStack command, create a file containing environment variables that can be loaded at any time. The following commands will create a file named `adminrc` containing environment variables for the admin user:

```
# cat >> ~/adminrc <<EOF
export OS_PROJECT_DOMAIN_NAME=default
export OS_USER_DOMAIN_NAME=default
export OS_PROJECT_NAME=admin
export OS_USERNAME=admin
export OS_PASSWORD=openstack
export OS_AUTH_URL=http://controller01:35357/v3
export OS_IDENTITY_API_VERSION=3
EOF
```

The following commands will create a file named `demorc` containing environment variables for the demo user:

```
# cat >> ~/demorc <<EOF
export OS_PROJECT_DOMAIN_NAME=default
export OS_USER_DOMAIN_NAME=default
```



```
export OS_PROJECT_NAME=demo
export OS_USERNAME=demo
export OS_PASSWORD=demo
export OS_AUTH_URL=http://controller01:35357/v3
export OS_IDENTITY_API_VERSION=3
EOF
```

Use the `source` command to load the environment variables from the file:

```
# source ~/adminrc
```

The demo user doesn't exist yet but will be created in the following sections.

Defining services and API endpoints in Keystone

Each OpenStack service that is installed should be registered with the Identity service so that its location on the network can be tracked. There are two commands involved in registering a service:

- `openstack service create`: Describes the service that is being created
- `openstack endpoint create`: Associates API endpoints with the service

The OpenStack Identity service and endpoint were created during the bootstrap process earlier in this chapter. You can verify the objects were created using the `openstack endpoint list` and `openstack service list` commands as shown here:

```
root@controller01:~# openstack service list
```

ID	Name	Type
cafd0b7e8f824b5c994c10cb93d489fc	keystone	identity

The IDs of resources within an OpenStack cloud are unique and will vary between environments, so don't worry if yours don't match those shown here:

```
root@controller01:~# openstack endpoint list
```

ID	Region	Service Name	Service Type	Enabled	Interface	URL
5e8cd154da40450aa0de80e1fdce59b0	RegionOne	keystone	identity	True	internal	http://controller01:5000/v3/
bfbcd2527b3b44d98b2214fc32d1cad2f	RegionOne	keystone	identity	True	public	http://controller01:5000/v3/
f99b6392c5da4d0e93f5b18daa1522bc	RegionOne	keystone	identity	True	admin	http://controller01:35357/v3/

Defining users, projects, and roles in Keystone

Once the installation of Keystone is complete, it is necessary to set up domains, users, projects, roles, and endpoints that will be used by various OpenStack services.



In this installation, the `default` domain will be used.

In Keystone, a project (or tenant) represents a logical group of users to which resources are assigned. The terms project and tenant are used interchangeably throughout various OpenStack services, but project is the preferred term. Resources are assigned to projects and not directly to users. An `admin` project, user, and role were created during the Keystone bootstrap process. Create a `demo` project for regular users and a `service` project for other OpenStack services to use:

```
# openstack project create --description "Service Project" service
# openstack project create --description "Demo Project" demo
```

Next, create a regular user called `demo`. Specify a secure password for the `demo` user:

```
# openstack user create demo --password=demo
```

Create the `user` role:

```
# openstack role create user
```

Lastly, add the `user` role to the `demo` user in the `demo` project:

```
# openstack role add --project demo --user demo user
```

Installing and configuring the image service

Glance is the image service for OpenStack. It is responsible for storing images and snapshots of instances, and for providing images to `compute` nodes when instances are created.

Configuring the database

Using the `mysql` client, create the Glance database and associated user:

```
# mysql
```

Enter the following SQL statements at the MariaDB [(none)] > prompt:

```
CREATE DATABASE glance;
GRANT ALL PRIVILEGES ON glance.* TO 'glance'@'localhost' IDENTIFIED BY
'glance';
GRANT ALL PRIVILEGES ON glance.* TO 'glance'@ '%' IDENTIFIED BY 'glance';
quit;
```

Defining the Glance user, service, and endpoints

Using the openstack client, create the Glance user:

```
# openstack user create glance --domain default --password=glance
```

Add the admin role to the glance user in the service project:

```
# openstack role add --project service --user glance admin
```

Next, create the glance service entity:

```
# openstack service create --name glance \
--description "OpenStack Image" image
```

Lastly, create the Glance endpoints:

```
# openstack endpoint create --region RegionOne \
image public http://controller01:9292
# openstack endpoint create --region RegionOne \
image internal http://controller01:9292
# openstack endpoint create --region RegionOne \
image admin http://controller01:9292
```

Installing and configuring Glance components

To install Glance, run the following command from the controller node:

```
# apt install glance
```

Update the database connection string in the glance-api configuration file found at /etc/glance/glance-api.conf to use the MySQL Glance database:

```
[database]
...
connection = mysql+pymysql://glance:glance@controller01/glance
```

Repeat the process for the `glance-registry` configuration file found at `/etc/glance/glance-registry.conf`:

```
[database]
...
connection = mysql+pymysql://glance:glance@controller01/glance
```

Configuring authentication settings

Both the `glance-api` and `glance-registry` service configuration files must be updated with the appropriate authentication settings before the services will operate.

Update the `[keystone_authtoken]` settings in the `glance-api` configuration file found at `/etc/glance/glance-api.conf`:

```
[keystone_authtoken]
...
auth_uri = http://controller01:5000
auth_url = http://controller01:35357
memcached_servers = controller01:11211
auth_type = password
user_domain_name = default
project_domain_name = default
project_name = service
username = glance
password = glance
```

Repeat the process for the `glance-registry` configuration file found at `/etc/glance/glance-registry.conf`:

```
[keystone_authtoken]
...
auth_uri = http://controller01:5000
auth_url = http://controller01:35357
memcached_servers = controller01:11211
auth_type = password
user_domain_name = default
project_domain_name = default
project_name = service
username = glance
password = glance
```

Configuring additional settings

Update the `glance-api` configuration file found at `/etc/glance/glance-api.conf` with the following additional settings:

```
[paste_deploy]
...
flavor = keystone

[glance_store]
...
stores = file,http
default_store = file
filesystem_store_datadir = /var/lib/glance/images
```

Next, update the `glance-registry` configuration file found at `/etc/glance/glance-registry.conf` with the following additional settings:

```
[paste_deploy]
...
flavor = keystone
```

When both files have been updated, populate the Glance database using the `glance-manage` utility:

```
# su -s /bin/sh -c "glance-manage db_sync" glance
```

Lastly, restart the Glance services with the following command:

```
# systemctl restart glance-registry glance-api
```

Verifying the Glance image service installation

Source the `adminrc` script to set or update the environment variables:

```
# source ~/adminrc
```

To verify that Glance was installed and configured properly, download a test image from the internet and verify it can be uploaded to the image server:

```
# mkdir /tmp/images
# wget -P /tmp/images
http://download.cirros-cloud.net/0.4.0/cirros-0.4.0-x86_64-disk.img
```

Upload the image to Glance using the following command:

```
# openstack image create "cirros-0.4.0"
--file /tmp/images/cirros-0.4.0-x86_64-disk.img
--disk-format qcow2
--container-format bare
--public
```

Verify the image exists in Glance using the `openstack image list` command shown here:

```
root@controller01:~# openstack image list
+-----+-----+-----+
| ID                               | Name           | Status |
+-----+-----+-----+
| 60346ac7-542c-47c2-aac3-7031dfc03fcd | cirros-0.4.0   | active |
+-----+-----+-----+
```

Installing additional images

The CirrOS image is limited in functionality and is recommended only for testing network connectivity and basic Compute services. Multiple vendors provide cloud-ready images for use with OpenStack, including the following:

Canonical - Ubuntu	http://cloud-images.ubuntu.com/
Red Hat - CentOS	http://cloud.centos.org/centos/

To install the Ubuntu 16.04 LTS image, download the file to `/tmp/images` and upload it to Glance:

```
# wget -P /tmp/images
http://cloud-images.ubuntu.com/xenial/current/xenial-server-cloudimg-amd64-disk1.img
```

Use the `openstack image create` command to upload the new image:

```
# openstack image create "ubuntu-xenial-16.04"
--file /tmp/images/xenial-server-cloudimg-amd64-disk1.img
--disk-format qcow2 --container-format bare
--public
```

Another look at the image list shows the new Ubuntu image is available for use:

```
root@controller01:~# openstack image list
```

ID	Name	Status
60346ac7-542c-47c2-aac3-7031dfc03fcd	cirros-0.4.0	active
ffdb76f0-6cce-4d79-99a5-ccc6e76d530a	ubuntu-xenial-16.04	active

Installing and configuring the Compute service

OpenStack Compute is a collection of services that enable cloud operators and users to launch virtual machine instances. Most services run on the `controller` node, with the exception of the `nova-compute` service, which runs on the `compute` nodes and is responsible for launching the virtual machine instances on those nodes.

Configuring the database

Using the `mysql` client on the `controller` node, create the Nova database(s) and associated user(s):

```
# mysql
```

Enter the following SQL statements at the MariaDB `[(none)] >` prompt:

```
CREATE DATABASE nova;
CREATE DATABASE nova_api;
CREATE DATABASE nova_cell0;
GRANT ALL PRIVILEGES ON nova.* TO 'nova'@'localhost' IDENTIFIED BY 'nova';
GRANT ALL PRIVILEGES ON nova.* TO 'nova'@'%' IDENTIFIED BY 'nova';
GRANT ALL PRIVILEGES ON nova_api.* TO 'nova'@'localhost' IDENTIFIED BY
'nova';
GRANT ALL PRIVILEGES ON nova_api.* TO 'nova'@'%' IDENTIFIED BY 'nova';
GRANT ALL PRIVILEGES ON nova_cell0.* TO 'nova'@'localhost' IDENTIFIED BY
'nova';
GRANT ALL PRIVILEGES ON nova_cell0.* TO 'nova'@'%' IDENTIFIED BY 'nova';
quit;
```

Defining the Nova user, service, and endpoints

Source the `adminrc` credentials as shown here:

```
# source ~/adminrc
```

Using the `openstack` client, create both the `nova` and `placement` users:

```
# openstack user create nova --domain default --password=nova
# openstack user create placement
  --domain default --password=placement
```

Add the `admin` role to the `nova` and `placement` users in the `service` project:

```
# openstack role add --project service --user nova admin
# openstack role add --project service --user placement admin
```

Next, create the `compute` and `placement` service entities:

```
# openstack service create --name nova
  --description "OpenStack Compute" compute
# openstack service create --name placement
  --description "Placement API" placement
```

Lastly, create the `compute` and `placement` endpoints:

```
# openstack endpoint create --region RegionOne
  compute public http://controller01:8774/v2.1
# openstack endpoint create --region RegionOne
  compute internal http://controller01:8774/v2.1
# openstack endpoint create --region RegionOne
  compute admin http://controller01:8774/v2.1
# openstack endpoint create --region RegionOne
  placement public http://controller01:8778
# openstack endpoint create --region RegionOne
  placement internal http://controller01:8778
# openstack endpoint create --region RegionOne
  placement admin http://controller01:8778
```

Installing and configuring controller node components

Execute the following command on the `controller` node to install the various `Compute` services used by the controller:

```
# apt install nova-api nova-conductor nova-consoleauth
  nova-novncproxy nova-scheduler nova-placement-api
```


Update the `[database]` and `[api_database]` sections of the Nova configuration file found at `/etc/nova/nova.conf` with the following database connection strings:

```
[database]
...
connection = mysql+pymysql://nova:nova@controller01/nova

[api_database]
...
connection = mysql+pymysql://nova:nova@controller01/nova_api
```

Update the `[DEFAULT]` section of the Nova configuration file to configure Nova to use the RabbitMQ message broker:

```
[DEFAULT]
...
transport_url = rabbit://openstack:rabbit@controller01
```

The VNC Proxy is an OpenStack component that allows users to access their instances through VNC clients. VNC stands for **Virtual Network Computing**, and is a graphical desktop sharing system that uses the Remote Frame Buffer protocol to control another computer over a network. The controller must be able to communicate with `compute` nodes for VNC services to work properly through the Horizon dashboard or other VNC clients.

Update the `[vnc]` section of the Nova configuration file to configure the appropriate VNC settings for the `controller` node:

```
[vnc]
...
enabled = true
vncserver_listen = 10.10.0.100
vncserver_proxyclient_address = 10.10.0.100
```

Configuring authentication settings

Update the Nova configuration file at `/etc/nova/nova.conf` with the following Keystone-related attribute:

```
[api]
...
auth_strategy= keystone

[keystone_authtoken]
...
auth_uri = http://controller01:5000
auth_url = http://controller01:35357
```

```
memcached_servers = controller01:11211
auth_type = password
project_domain_name = Default
user_domain_name = Default
project_name = service
username = nova
password = nova
```

Additional controller tasks

Update the Nova configuration file at `/etc/nova/nova.conf` to specify the location of the Glance API:

```
[glance]
...
api_servers = http://controller01:9292
```

Update the Nova configuration file to set the lock file path for Nova services and specify the IP address of the controller node:

```
[oslo_concurrency]
...
lock_path = /var/lib/nova/tmp

[DEFAULT]
...
my_ip = 10.10.0.100
```

Configure the `[placement]` section within `/etc/nova/nova.conf`. Be sure to comment out any existing `os_region_name` configuration:

```
[placement]
...
os_region_name = RegionOne
auth_url = http://controller01:35357/v3
auth_type = password
project_domain_name = Default
user_domain_name = Default
project_name = service
username = placement
password = placement
```

Populate the nova-api database using the nova-manage utility:

```
# su -s /bin/sh -c "nova-manage api_db sync" nova
```

Register the cell10 database using the nova-manage utility:

```
# su -s /bin/sh -c "nova-manage cell_v2 map_cell10" nova
```

Create the cell11 cell:

```
# su -s /bin/sh -c "nova-manage cell_v2 create_cell  
--name=cell11 --verbose" nova
```

Populate the nova database using the nova-manage utility:

```
# su -s /bin/sh -c "nova-manage db sync" nova
```

Lastly, restart the controller-based Compute services for the changes to take effect:

```
# systemctl restart nova-api nova-consoleauth nova-scheduler nova-  
conductor nova-novncproxy
```

Installing and configuring compute node components

Once the Compute services have been configured on the controller node, at least one other host must be configured as a compute node. The compute node receives requests from the controller node to host virtual machine instances.

On the compute nodes, install the nova-compute package and related packages. These packages provide virtualization support services to the compute node:

```
# apt install nova-compute
```

Update the Nova configuration file at `/etc/nova/nova.conf` on the compute nodes with the following Keystone-related settings:

```
[api]  
...  
auth_strategy= keystone  
  
[keystone_authtoken]  
...  
auth_uri = http://controller01:5000  
auth_url = http://controller01:35357  
memcached_servers = controller01:11211  
auth_type = password
```

```
project_domain_name = Default
user_domain_name = Default
project_name = service
username = nova
password = nova
```

Next, update the [DEFAULT] section of the Nova configuration file to configure Nova to use the RabbitMQ message broker:

```
[DEFAULT]
...
transport_url = rabbit://openstack:rabbit@controller01
```

Then, update the Nova configuration file to provide remote console access to instances through a proxy on the controller node. The remote console is accessible through the Horizon dashboard. The IP configured as `my_ip` should be the respective management IP of each compute node.

Compute01:

```
[DEFAULT]
...
my_ip = 10.10.0.101

[vnc]
...
vncserver_proxyclient_address = 10.10.0.101
enabled = True
vncserver_listen = 0.0.0.0
novncproxy_base_url = http://controller01:6080/vnc_auto.html
```

Compute02:

```
[DEFAULT]
...
my_ip = 10.10.0.102

[vnc]
...
vncserver_proxyclient_address = 10.10.0.102
enabled = True
vncserver_listen = 0.0.0.0
novncproxy_base_url = http://controller01:6080/vnc_auto.html
```

Compute03:

```
[DEFAULT]
...
```

```
my_ip = 10.10.0.103

[vnc]
...
vncserver_proxyclient_address = 10.10.0.103
enabled = True
vncserver_listen = 0.0.0.0
novncproxy_base_url = http://controller01:6080/vnc_auto.html
```

Additional compute tasks

Update the Nova configuration file at `/etc/nova/nova.conf` to specify the location of the Glance API:

```
[glance]
...
api_servers = http://controller01:9292
```

Update the Nova configuration file to set the lock file path for Nova services:

```
[oslo_concurrency]
...
lock_path = /var/lib/nova/tmp
```

Configure the `[placement]` section within `/etc/nova/nova.conf`. Be sure to comment out any existing `os_region_name` configuration:

```
[placement]
...
os_region_name = RegionOne
auth_url = http://controller01:35357/v3
auth_type = password
project_domain_name = Default
user_domain_name = Default
project_name = service
username = placement
password = placement
```

Restart the `nova-compute` service on all compute nodes:

```
# systemctl restart nova-compute
```

Adding the compute node(s) to the cell database

When compute services are started for the first time on a compute node, the node is registered via the Nova Placement API. To verify the compute node(s) are registered, use the `openstack compute service list` command as shown here:

```
root@controller01:~# openstack compute service list
```

ID	Binary	Host	Zone	Status	State	Updated At
1	nova-consoleauth	controller01	internal	enabled	up	2018-06-21T01:47:33.000000
2	nova-scheduler	controller01	internal	enabled	up	2018-06-21T01:47:34.000000
3	nova-conductor	controller01	internal	enabled	up	2018-06-21T01:47:34.000000
8	nova-compute	compute01	nova	enabled	up	2018-06-21T01:47:26.000000
9	nova-compute	compute02	nova	enabled	up	2018-06-21T01:47:35.000000
10	nova-compute	compute03	nova	enabled	up	2018-06-21T01:47:32.000000

Installing the OpenStack Dashboard

The OpenStack Dashboard, also known as Horizon, provides a web-based user interface to OpenStack services including Compute, Networking, Storage, and Identity, among others.

To add Horizon to the environment, install the following package on the `controller` node:

```
# apt install openstack-dashboard
```

Updating the host and API version configuration

Edit the `/etc/openstack-dashboard/local_settings.py` file and change the `OPENSTACK_HOST` value from its default to the following:

```
OPENSTACK_HOST = "controller01"
```

Set the API versions with the following dictionary:

```
OPENSTACK_API_VERSIONS = {
    "identity": 3,
    "image": 2,
    "volume": 2,
}
```

Configuring Keystone settings

Edit the `/etc/openstack-dashboard/local_settings.py` file and replace the following Keystone-related configuration options with these values:

```
OPENSTACK_KEYSTONE_URL = "http://%s:5000/v3" % OPENSTACK_HOST
OPENSTACK_KEYSTONE_MULTIDOMAIN_SUPPORT = True
OPENSTACK_KEYSTONE_DEFAULT_DOMAIN = "Default"
OPENSTACK_KEYSTONE_DEFAULT_ROLE = "user"
```

Modifying network configuration

The stock Horizon configuration enables certain network functionality that has not yet been implemented. As a result, you will experience errors in the dashboard that may impact usability. For now, disable all network functions in the dashboard by setting the following configuration:

```
OPENSTACK_NEUTRON_NETWORK = {
    'enable_router': False,
    'enable_quotas': False,
    'enable_ipv6': False,
    'enable_distributed_router': False,
    'enable_ha_router': False,
    'enable_lb': False,
    'enable_firewall': False,
    'enable_vpn': False,
    'enable_fip_topology_check': False,
}
```

Uninstalling default Ubuntu theme (optional)

By default, installations of the OpenStack Dashboard on Ubuntu include a theme that has been customized by Canonical. To remove the theme, update the `/etc/openstack-dashboard/local_settings.py` file and replace the `DEFAULT_THEME` value with `default` instead of `ubuntu`:

```
DEFAULT_THEME = 'default'
```

The examples in this book assume the default theme is in place.

Reloading Apache

Once the above changes have been made, reload the Apache web server configuration using the following command:

```
# systemctl reload apache2
```

Testing connectivity to the dashboard

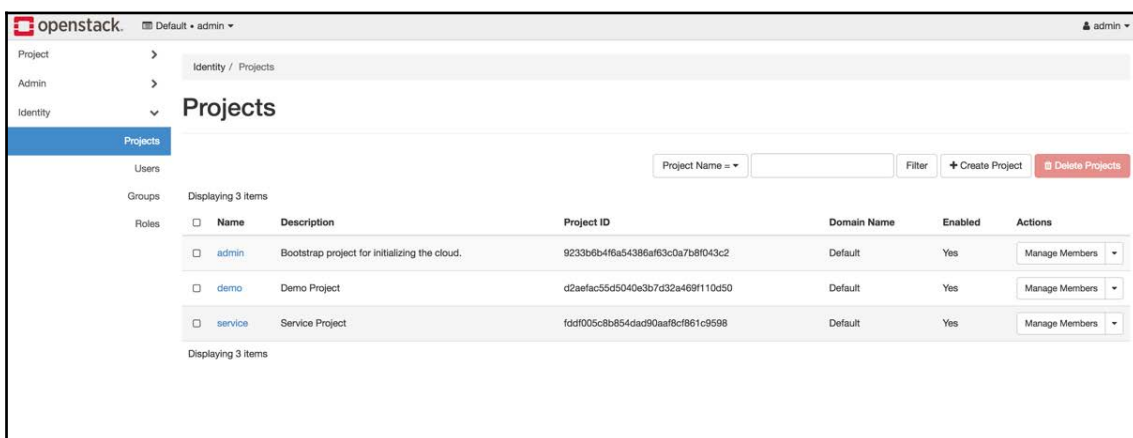
From a machine that has access to the management network of the `controller01` node, open the following URL in a web browser: `http://controller01/horizon/`.

The API network is reachable from my workstation and the `/etc/hosts` file on my client workstation has been updated to include the same hostname-to-IP mappings configured earlier in this chapter.

The following screenshot demonstrates a successful connection to the dashboard. The username and password were created in the *Defining users, projects, and roles in Keystone* section earlier in this chapter. In this installation, the domain is `Default`, the username is `admin`, and the password is `openstack`.

Familiarizing yourself with the dashboard

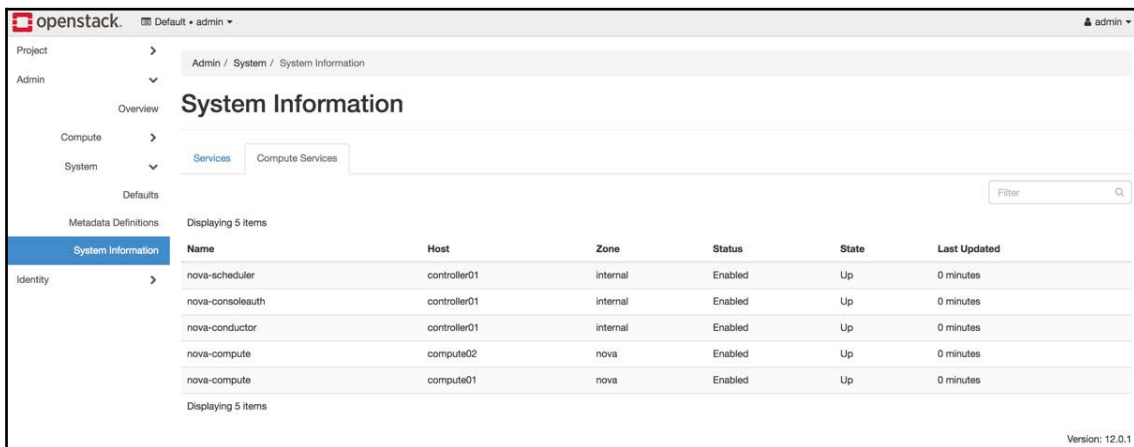
Once you have successfully logged in, the dashboard defaults to the **Identity** panel. Project-related tasks can be completed in the **Project** panel, while users with the `admin` role will find an **Admin** panel that allows for administrative tasks to be completed:



The screenshot shows the OpenStack Horizon dashboard interface. The top navigation bar includes the OpenStack logo, a breadcrumb trail "Default > admin", and a user profile icon labeled "admin". The left sidebar contains a menu with "Project", "Admin", "Identity", and "Projects" (which is highlighted in blue). The main content area is titled "Projects" and displays a table of existing projects. Above the table, there are controls for "Project Name" (a dropdown), a "Filter" button, a "+ Create Project" button, and a "Delete Projects" button. The table lists three projects: "admin", "demo", and "service". Each row includes checkboxes, the project name, description, ID, domain name, enabled status, and an "Actions" menu with a "Manage Members" option.

	Name	Description	Project ID	Domain Name	Enabled	Actions
☐	admin	Bootstrap project for initializing the cloud.	9233b6b4f6a54386af63c0a7b8f043c2	Default	Yes	Manage Members
☐	demo	Demo Project	d2aefac55d5040e3b7d32a469f110d50	Default	Yes	Manage Members
☐	service	Service Project	fdcf005c8b854dad90aaf8cf861c9598	Default	Yes	Manage Members

By navigating to the **System Information** page within the **Admin / System** panel, we can see services listed that correspond to those installed earlier in this chapter:



The screenshot shows the OpenStack Admin interface. The breadcrumb trail is 'Admin / System / System Information'. The 'System Information' page has two tabs: 'Services' (selected) and 'Compute Services'. A filter box is present on the right. Below the tabs, it says 'Displaying 5 items'. A table lists the services with columns: Name, Host, Zone, Status, State, and Last Updated.

Name	Host	Zone	Status	State	Last Updated
nova-scheduler	controller01	internal	Enabled	Up	0 minutes
nova-consoleauth	controller01	internal	Enabled	Up	0 minutes
nova-conductor	controller01	internal	Enabled	Up	0 minutes
nova-compute	compute02	nova	Enabled	Up	0 minutes
nova-compute	compute01	nova	Enabled	Up	0 minutes

At the bottom right of the screenshot, the version '12.0.1' is displayed.

The OpenStack Dashboard allows users to make changes within any project for which they have permissions. The menu located to the right of the OpenStack logo is a drop-down menu that allows users to select the **project** they'd like to work with:



Only projects to which the user is assigned will be listed. If the user has the `admin` role, additional cloud-wide changes can be made within the **Admin** panel. Using the Horizon dashboard to perform networking functions will be covered in later chapters.

Summary

At this point in the installation, the OpenStack Identity, Image, Dashboard, and Compute services have been successfully deployed across the nodes of the cloud. The environment is not ready to host instances just yet, as OpenStack Networking services have not been installed or configured. If issues arise during the installation of services described in this chapter, be sure to check the log messages found in `/var/log/nova/`, `/var/log/glance/`, `/var/log/apache2/`, and `/var/log/keystone/` for assistance in troubleshooting. For additional information regarding the installation process, feel free to check out the documentation available the OpenStack website at the following URL: <http://docs.openstack.org>.

In the next chapter, we will begin the installation of Neutron networking services and discover additional information about the internal architecture of OpenStack Networking.

3

Installing Neutron

OpenStack Networking, also known as Neutron, provides a network infrastructure-as-a-service platform to users of the cloud. In the last chapter, we installed some of the base services of OpenStack, including the Identity, Image, and Compute services. In this chapter, I will guide you through the installation of Neutron networking services on top of the OpenStack environment that we installed in the previous chapter

The components to be installed include the following:

- Neutron API server
- Modular Layer 2 (ML2) plugin
- DHCP agent
- Metadata agent

By the end of this chapter, you will have a basic understanding of the function and operation of various Neutron plugins and agents, as well as a foundation on top of which a virtual switching infrastructure can be built.

Basic networking elements in Neutron

Neutron constructs the virtual network using elements that are familiar to most system and network administrators, including networks, subnets, ports, routers, load balancers, and more.

Using version 2.0 of the core Neutron API, users can build a network foundation composed of the following entities:

- **Network:** A network is an isolated Layer 2 broadcast domain. Typically, networks are reserved for the projects that created them, but they can be shared among projects if configured accordingly. The network is the core entity of the Neutron API. Subnets and ports must always be associated with a network.

- **Subnet:** A subnet is an IPv4 or IPv6 address block from which IP addresses can be assigned to virtual machine instances. Each subnet must have a CIDR and must be associated with a network. Multiple subnets can be associated with a single network and can be non-contiguous. A DHCP allocation range can be set for a subnet that limits the addresses provided to instances.
- **Port:** A port in Neutron is a logical representation of a virtual switch port. Virtual machine interfaces are mapped to Neutron ports, and these ports define both the MAC address and the IP address that is to be assigned to the interfaces plugged into them. Neutron port definitions are stored in the Neutron database, which is then used by the respective plugin agent to build and connect the virtual switching infrastructure.

Cloud operators and users alike can configure network topologies by creating and configuring networks and subnets, and then instruct services like Nova to attach virtual devices to ports on these networks. Users can create multiple networks, subnets, and ports, but are limited to thresholds defined by per-project quotas set by the cloud administrator.

Extending functionality with plugins

The OpenStack Networking project provides reference plugins and drivers that are developed and supported by the OpenStack community, and also supports third-party plugins and drivers that extend network functionality and implementation of the Neutron API. Plugins and drivers can be created that use a variety of software and hardware-based technologies to implement the network built by operators and users.

There are two major plugin types within the Neutron architecture:

- Core plugin
- Service plugin

A **core plugin** implements the core Neutron API, and is responsible for adapting the logical network described by networks, ports, and subnets into something that can be implemented by the L2 agent and IP address management system running on the host.

A **service plugin** provides additional network services such as routing, load balancing, firewalling, and more.

In this book, the following core plugin will be discussed:

- Modular Layer 2 Plugin

The following service plugins will be covered in later chapters:

- Router
- Load balancer
- Trunk



The Neutron API provides a consistent experience to the user despite the chosen networking plugin. For more information on interacting with the Neutron API, please visit the following URL: <https://developer.openstack.org/api-ref/network/v2/index.html>.

Modular Layer 2 plugin

Prior to the inclusion of the **Modular Layer 2 (ML2)** plugin in the Havana release of OpenStack, Neutron was limited to using a single core plugin. This design resulted in homogenous network architectures that were not extensible. Operators were forced to make long-term decisions about the network stack that could not easily be changed in the future. The ML2 plugin, on the other hand, is extensible by design and supports heterogeneous network architectures that can leverage multiple technologies simultaneously. The ML2 plugin replaced two monolithic plugins in its reference implementation: the Linux bridge core plugin and the Open vSwitch core plugin.

Drivers

The ML2 plugin introduced the concept of TypeDrivers and Mechanism drivers to separate the types of networks being *implemented* and the mechanisms for *implementing* networks of those types.

TypeDrivers

An ML2 **TypeDriver** maintains a type-specific network state, validates provider network attributes, and describes network segments using provider attributes. Provider attributes include network interface labels, segmentation IDs, and network types. Supported network types include `local`, `flat`, `vlan`, `gre`, `vxlan`, and `geneve`. The following table describes the differences between those network types:

Type	Description
Local	A local network is one that is isolated from other networks and nodes. Instances connected to a local network may communicate with other instances in the same network on the same <code>compute</code> node, but are unable to communicate with instances on another host. Because of this design limitation, local networks are recommended for testing purposes only.

Flat	In a flat network , no 802.1q VLAN tagging or other network segregation takes place. In many environments, a flat network corresponds to an <i>access</i> VLAN or <i>native</i> VLAN on a trunk.
VLAN	VLAN networks are networks that utilize 802.1q tagging to segregate network traffic. Instances in the same VLAN are considered part of the same network and are in the same Layer 2 broadcast domain. Inter-VLAN routing, or routing between VLANs, is only possible through the use of a physical or virtual router.
GRE	GRE networks use the generic routing encapsulation tunneling protocol (IP protocol 47) to encapsulate packets and send them over point-to-point networks between nodes. The <code>KEY</code> field in the GRE header is used to segregate networks.
VXLAN	A VXLAN network uses a unique segmentation ID, called a VXLAN Network Identifier (VNI), to differentiate traffic from other VXLAN networks. Traffic from one instance to another is encapsulated by the host using the VNI and sent over an existing Layer 3 network using UDP, where it is decapsulated and forwarded to the instance. The use of VXLAN to encapsulate packets over an existing network is meant to solve limitations of VLANs and physical switching infrastructure.
GENEVE	A GENEVE network resembles a VXLAN network, in that it uses a unique segmentation ID, called a virtual network interface (VNI), to differentiate traffic from other GENEVE networks. Packets are encapsulated with a unique header and UDP is used as the transport mechanism. GENEVE leverages the benefits of multiple overlay technologies such as VXLAN, NVGRE, and STT and is primarily used by OVN at this time.

Mechanism drivers

An ML2 **Mechanism driver** is responsible for taking information established by the type driver and ensuring that it is properly implemented. Multiple Mechanism drivers can be configured to operate simultaneously, and can be described using three types of models:

- **Agent-based:** Includes Linux bridge, Open vSwitch, SR-IOV, and others
- **Controller-based:** Includes Juniper Contrail, Tungsten Fabric, OVN, Cisco ACI, VMWare NSX, and others
- **Top-of-Rack:** Includes Cisco Nexus, Arista, Mellanox, and others

Mechanism drivers to be discussed in this book include the following:

- Linux bridge
- Open vSwitch
- L2 population

The Linux bridge and Open vSwitch ML2 Mechanism drivers are used to configure their respective virtual switching technologies within nodes that host instances and network services. The Linux bridge driver supports `local`, `flat`, `vlan`, and `vxlan` network types, while the Open vSwitch driver supports all of those as well as the `gre` network type. Support for other type drivers, such as `geneve`, will vary based on the implemented Mechanism driver.

The L2 population driver is used to limit the amount of broadcast traffic that is forwarded across the overlay network fabric when VXLAN networks are used. Under normal circumstances, unknown unicast, multicast, and broadcast traffic may be flooded out from all tunnels to other `compute` nodes. This behavior can have a negative impact on the overlay network fabric, especially as the number of hosts in the cloud scales out.

As an authority on what instances and other network resources exist in the cloud, Neutron can pre-populate forwarding databases on all hosts to avoid a costly learning operation. ARP proxy, a feature of the L2 population driver, enables Neutron to pre-populate the ARP table on all hosts in a similar manner to avoid ARP traffic from being broadcast across the overlay fabric.

ML2 architecture

The following diagram demonstrates how the Neutron API service interacts with the various plugins and agents responsible for constructing the virtual and physical network at a high level:

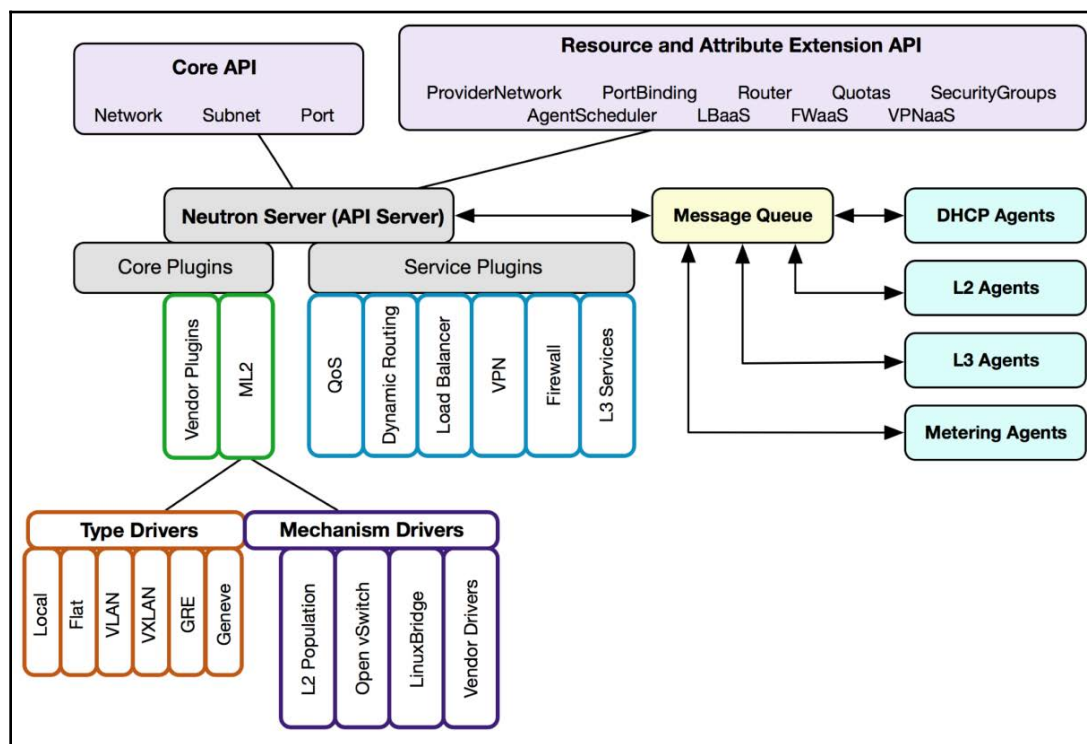


Figure 3.1

The preceding diagram demonstrates the interaction between the Neutron API, Neutron plugins and drivers, and services such as the L2 and L3 agents. For more information on the Neutron ML2 plugin architecture, please refer to the following URL: <https://docs.openstack.org/neutron/pike/admin/config-ml2.html>

Network namespaces

OpenStack was designed with multi-tenancy in mind, and provides users with the ability to create and manage their own compute and network resources. Neutron supports each tenant having multiple private networks, routers, firewalls, load balancers, and other networking resources, and is able to isolate many of these objects through the use of network namespaces.

A **network namespace** is defined as a logical copy of the network stack with its own routes, firewall rules, and network interfaces. When using the open source reference plugins and drivers, every DHCP server, router, and load balancer that is created by a user is implemented in a network namespace. By using network namespaces, Neutron is able to provide isolated DHCP and routing services to each network, allowing users to create overlapping networks with other users in other projects and even other networks in the same project.

The following naming convention for network namespaces should be observed:

- **DHCP Namespace:** `qdhcp-<network UUID>`
- **Router Namespace:** `qrouter-<router UUID>`
- **Load Balancer Namespace:** `qlbaas-<load balancer UUID>`

A `qdhcp` namespace contains a DHCP service that provides IP addresses to instances using the DHCP protocol. In a reference implementation, `dnsmasq` is the process that services DHCP requests. The `qdhcp` namespace has an interface plugged into the virtual switch and is able to communicate with instances and other devices in the same network. A `qdhcp` namespace is created for every network where the associated subnet(s) have DHCP enabled.

A `qrouter` namespace represents a virtual router, and is responsible for routing traffic to and from instances in subnets it is connected to. Like the `qdhcp` namespace, the `qrouter` namespace is connected to one or more virtual switches depending on the configuration. In some cases, multiple namespaces may be used to plumb the virtual router infrastructure. These additional namespaces, known as `fip` and `snat`, are used for distributed virtual routers (DVR) and will be discussed later in this book.

A `qlbaas` namespace represents a virtual load balancer, and contains a service such as HAProxy that load balances traffic to instances. The `qlbaas` namespace is connected to a virtual switch and can communicate with instances and other devices in the same network.



Fun fact: The leading `q` in the name of the network namespaces stands for Quantum, the original name for the OpenStack Networking service.

Network namespaces of the aforementioned types will only be seen on nodes running the Neutron DHCP, L3, or LBaaS agents, respectively. These services are typically only configured on controllers or dedicated network nodes. When distributed virtual routers are configured, you may find router-related namespaces on `compute` nodes as well. The `ip netns list` command can be used to list available namespaces, and commands can be executed within the namespace using the following syntax:

```
ip netns exec NAMESPACE_NAME <command>
```

Commands that can be executed in the namespace include `ip`, `route`, `iptables`, and more. The output of these commands corresponds to data that's specific to the namespace they are executed in. Tools such as `tcpdump` can also be executed in a network namespace to assist in troubleshooting the virtual network infrastructure.

For more information on network namespaces, see the man page for `ip netns` at the following URL: <http://man7.org/linux/man-pages/man8/ip-netns.8.html>.

Installing and configuring Neutron services

In this installation, the various services that make up OpenStack Networking will be installed on the `controller` node rather than a dedicated networking node. The `compute` nodes will run L2 agents that interface with the `controller` node and provide virtual switch connections to instances.



Remember, the configuration settings recommended here and online at docs.openstack.org may not be appropriate for production environments.

Creating the Neutron database

Using the `mysql` client on the `controller` node, create the Neutron database and associated user:

```
# mysql
```

Enter the following SQL statements at the MariaDB [(none)] > prompt:

```
CREATE DATABASE neutron;
GRANT ALL PRIVILEGES ON neutron.* TO 'neutron'@'localhost' IDENTIFIED BY
'neutron';
GRANT ALL PRIVILEGES ON neutron.* TO 'neutron'@ '%' IDENTIFIED BY 'neutron';
quit;
```

Configuring the Neutron user, role, and endpoint in Keystone

To function properly, Neutron requires that a user, role, and endpoint be created in Keystone. When executed from the `controller` node, the following commands will create a user called `neutron` in Keystone, associate the `admin` role with the `neutron` user, and add the `neutron` user to the `service` project:

```
# source ~/adminrc
# openstack user create --domain Default --password=neutron neutron
# openstack role add --project service --user neutron admin
```

Create a service in Keystone that describes the OpenStack Networking service by executing the following command on the `controller` node:

```
# openstack service create --name neutron
--description "OpenStack Networking" network
```

To create the endpoints, use the following `openstack endpoint create` commands:

```
# openstack endpoint create --region RegionOne
network public http://controller01:9696
# openstack endpoint create --region RegionOne
network internal http://controller01:9696
# openstack endpoint create --region RegionOne
network admin http://controller01:9696
```

Installing Neutron packages

To install the Neutron API server, the DHCP and metadata agents, and the ML2 plugin on the controller, issue the following command:

```
# apt install neutron-server neutron-dhcp-agent
neutron-metadata-agent neutron-plugin-ml2
python-neutronclient
```



The Neutron DHCP and metadata agents may not be required by all Mechanism drivers but are used when implementing the `openvswitch` and `linuxbridge` drivers.

On all other hosts, only the ML2 plugin is required at this time:

```
# apt install neutron-plugin-ml2
```

On all nodes, update the `[database]` section of the Neutron configuration file at `/etc/neutron/neutron.conf` to use the proper MySQL database connection string based on the preceding values rather than the default value:

```
[database]
...
connection = mysql+pymysql://neutron:neutron@controller01/neutron
```

Configuring Neutron to use Keystone

The Neutron configuration file found at `/etc/neutron/neutron.conf` has dozens of settings that can be modified to meet the needs of the OpenStack cloud administrator. A handful of these settings must be changed from their defaults as part of this installation.

To specify Keystone as the authentication method for Neutron, update the `[DEFAULT]` section of the Neutron configuration file on all hosts with the following setting:

```
[DEFAULT]
...
auth_strategy = keystone
```

Neutron must also be configured with the appropriate Keystone authentication settings. The username and password for the `neutron` user in Keystone were set earlier in this chapter. Update the `[keystone_authtoken]` section of the Neutron configuration file on all hosts with the following settings:

```
[keystone_authtoken]
...
auth_uri = http://controller01:5000
auth_url = http://controller01:35357
memcached_servers = controller01:11211
auth_type = password
project_domain_name = default
```

```
user_domain_name = default
project_name = service
username = neutron
password = neutron
```

Configuring Neutron to use a messaging service

Neutron communicates with various OpenStack services on the AMQP messaging bus. Update the [DEFAULT] section of the Neutron configuration file on all hosts to specify RabbitMQ as the messaging broker:

```
[DEFAULT]
...
transport_url = rabbit://openstack:rabbit@controller01
```

Configuring Nova to utilize Neutron networking

Before Neutron can be utilized as the network manager for OpenStack Compute services, the appropriate configuration options must be set in the Nova configuration file located at `/etc/nova/nova.conf` on certain hosts.

On the controller and compute nodes, update the [neutron] section with the following:

```
[neutron]
...
url= http://controller01:9696
auth_url = http://controller01:35357
auth_type = password
project_domain_name = default
user_domain_name = default
region_name = RegionOne
project_name = service
username = neutron
password = neutron
```

Nova may require additional configuration once a Mechanism driver has been determined. The Linux bridge and Open vSwitch Mechanism drivers and their respective agents and Nova configuration changes will be discussed in further detail in upcoming chapters.

Configuring Neutron to notify Nova

Neutron must be configured to notify Nova of network topology changes. On the `controller` node, update the `[nova]` section of the Neutron configuration file located at `/etc/neutron/neutron.conf` with the following settings:

```
[nova]
...
auth_url = http://controller01:35357
auth_type = password
project_domain_name = default
user_domain_name = default
region_name = RegionOne
project_name = service
username = nova
password = nova
```

Configuring Neutron services

The `neutron-server` service exposes the Neutron API to users and passes all calls to the configured Neutron plugins for processing. By default, Neutron is configured to listen for API calls on all configured addresses, as seen by the default `bind_hosts` option in the Neutron configuration file:

```
bind_host = 0.0.0.0
```

As an additional security measure, it is possible to expose the API on the management or API network. To change the default value, update the `bind_host` value in the `[DEFAULT]` section of the Neutron configuration located at `/etc/neutron/neutron.conf` with the management address of the `controller` node. The deployment explained in this book will retain the default value.

Other configuration options that may require tweaking include the following:

- `core_plugin`
- `service_plugins`
- `dhcp_lease_duration`
- `dns_domain`

Some of these settings apply to all nodes, while others only apply to the `network` or `controller` node. The `core_plugin` configuration option instructs Neutron to use the specified networking plugin. Beginning with the Icehouse release, the ML2 plugin supersedes both the Linux bridge and Open vSwitch monolithic plugins.

On all nodes, update the `core_plugin` value in the `[DEFAULT]` section of the Neutron configuration file located at `/etc/neutron/neutron.conf` and specify the ML2 plugin:

```
[DEFAULT]
...
core_plugin = ml2
```

The `service_plugins` configuration option is used to define plugins that are loaded by Neutron for additional functionality. Examples of plugins include `router`, `firewall`, `lbaas`, `vpnaas` and `metering`. This option should only be configured on the `controller` node or any other node running the `neutron-server` service. Service plugins will be defined in later chapters.

The `dhcp_lease_duration` configuration option specifies the duration of an IP address lease by an instance. The default value is 86,400 seconds, or 24 hours. If the value is set too low, the network may be flooded with traffic due to short leases and frequent renewal attempts. The DHCP client on the instance itself is responsible for renewing the lease, and the frequency of this operation varies between operating systems. It is not uncommon for instances to attempt to renew their lease well before exceeding the lease duration. The value set for `dhcp_lease_duration` does not dictate how long an IP address stays associated with an instance, however. Once an IP address has been allocated to a port by Neutron, it remains associated with the port until the port or related instance is deleted.

The `dns_domain` configuration option specifies the DNS search domain that is provided to instances via DHCP when they obtain a lease. The default value is `openstacklocal`. This can be changed to whatever fits your organization. For the purpose of this installation, change the value from `openstacklocal` to `learningneutron.com`. On the `controller` node, update the `dns_domain` option in the Neutron configuration file located at `/etc/neutron/neutron.conf` to `learningneutron.com`:

```
[DEFAULT]
...
dns_domain = learningneutron.com
```

When instances obtain their address from the DHCP server, the domain is appended to the hostname, resulting in a fully-qualified domain name. Neutron does not support multiple domain names by default, instead relying on the project known as Designate to extend support for this functionality. More information on Designate can be found at the following URL: <https://docs.openstack.org/designate/latest/>.

Starting neutron-server

Before the `neutron-server` service can be started, the Neutron database must be updated based on the options we configured earlier in this chapter. Use the `neutron-db-manage` command on the `controller` node to update the database accordingly:

```
# su -s /bin/sh -c "neutron-db-manage
--config-file /etc/neutron/neutron.conf
--config-file /etc/neutron/plugins/ml2/ml2_conf.ini
upgrade head" neutron
```

Restart the Nova compute services on the `controller` node:

```
# systemctl restart nova-api nova-scheduler nova-conductor
```

Restart the Nova compute service on the `compute` nodes:

```
# systemctl restart nova-compute
```

Lastly, restart the `neutron-server` service on the `controller` node:

```
# systemctl restart neutron-server
```

Configuring the Neutron DHCP agent

Neutron utilizes `dnsmasq`, a free and lightweight DNS forwarder and DHCP server, to provide DHCP services to networks. The `neutron-dhcp-agent` service is responsible for spawning and configuring `dnsmasq` and metadata processes for each network that leverages DHCP.

The DHCP driver is specified in the `/etc/neutron/dhcp_agent.ini` configuration file. The DHCP agent can be configured to use other drivers, but `dnsmasq` support is built-in and requires no additional setup. The default `dhcp_driver` value is `neutron.agent.linux.dhcp.Dnsmasq` and can be left unmodified.

Other notable configuration options found in the `dhcp_agent.ini` configuration file include the following:

- `interface_driver`
- `enable_isolated_metadata`

The `interface_driver` configuration option should be configured appropriately based on the Layer 2 agent chosen for your environment:

- **Linux bridge:** `neutron.agent.linux.interface.BridgeInterfaceDriver`
- **Open vSwitch:** `neutron.agent.linux.interface.OVSInterfaceDriver`

Both the Linux bridge and Open vSwitch drivers will be discussed in further detail in upcoming chapters. For now, the default value of `<none>` will suffice.



Only one interface driver can be configured at a time per agent.

The `enable_isolated_metadata` configuration option is useful in cases where a physical network device such as a firewall or router serves as the default gateway for instances, but Neutron is still required to provide metadata services to those instances. When the L3 agent is used, an instance reaches the metadata service through the Neutron router that serves as its default gateway. An isolated network is assumed to be one in which a Neutron router is not serving as the gateway, but Neutron still handles DHCP requests for the instances. This is often the case when instances are leveraging flat or VLAN networks with physical gateway devices. The default value for `enable_isolated_metadata` is `False`. When set to `True`, Neutron can provide instances with a static route to the metadata service via DHCP in certain cases. More information on the use of metadata and this configuration can be found in Chapter 7, *Attaching Instances to Networks*. On the controller node, update the `enable_isolated_metadata` option in the DHCP agent configuration file located at `/etc/neutron/dhcp_agent.ini` to `True`:

```
[DEFAULT]
...
enable_isolated_metadata = True
```

Configuration options not mentioned here have sufficient default values and should not be changed unless your environment requires it.

Restarting the Neutron DHCP agent

Use the following commands to restart the `neutron-dhcp-agent` service on the controller node and check its status:

```
# systemctl restart neutron-dhcp-agent
# systemctl status neutron-dhcp-agent
```

The output should resemble the following:

```
root@controller01:~# systemctl status neutron-dhcp-agent
● neutron-dhcp-agent.service - OpenStack Neutron DHCP agent
   Loaded: loaded (/lib/systemd/system/neutron-dhcp-agent.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2018-02-08 14:14:34 UTC; 1min 29s ago
     Process: 968 ExecStartPre=/bin/chown neutron:adm /var/log/neutron (code=exited, status=0/SUCCESS)
     Process: 960 ExecStartPre=/bin/chown neutron:neutron /var/lock/neutron /var/lib/neutron (code=exited, status=0/SUCCESS)
     Process: 953 ExecStartPre=/bin/mkdir -p /var/lock/neutron /var/log/neutron /var/lib/neutron (code=exited, status=0/SUCCESS)
  Main PID: 972 (neutron-dhcp-ag)
    Tasks: 1
   Memory: 108.2M
      CPU: 5.067s
   CGroup: /system.slice/neutron-dhcp-agent.service
           └─972 /usr/bin/python /usr/bin/neutron-dhcp-agent --config-file=/etc/neutron/neutron.conf
              --config-file=/etc/neutron/dhcp_agent.ini --log-file=/var/log/neutron/neutron-dhcp-agent.log
```

The agent should be in an active (running) status. Use the `openstack network agent list` command to verify that the service has checked in:

```
root@controller01:~# openstack network agent list --agent-type=dhcp
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
75c095d1-df2a-491e-bd87-9b8b9ba4f9d4	DHCP agent	controller01	nova	:)	UP	neutron-dhcp-agent

A smiley face under the `Alive` column means that the agent is properly communicating with the `neutron-server` service.

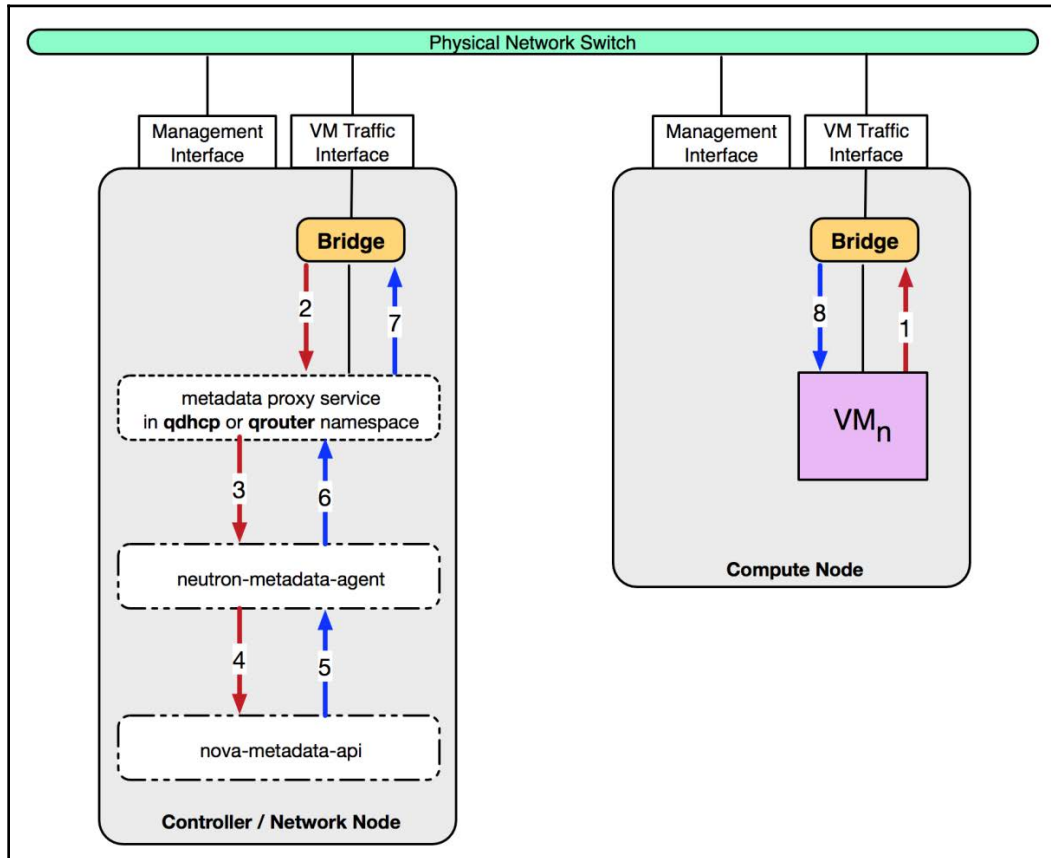
Configuring the Neutron metadata agent

OpenStack Compute provides a metadata service that enables users to retrieve information about their instances that can be used to configure or manage the running instance.

Metadata includes information such as the hostname, fixed and floating IPs, public keys, and more. In addition to metadata, users can access **userdata** such as scripts and other bootstrapping configurations that can be executed during the boot process or once the instance is active. OpenStack Networking implements a proxy that forwards metadata requests from instances to the metadata service provided by OpenStack Compute.

Instances typically access the metadata service over HTTP at `http://169.254.169.254` during the boot process. This mechanism is provided by `cloud-init`, a utility found on most cloud-ready images and available at the following URL: `https://launchpad.net/cloud-init`.

The following diagram provides a high-level overview of the retrieval of metadata from an instance when the `controller` node hosts networking services:



In the preceding diagram, the following actions take place when an instance makes a request to the metadata service:

- An instance sends a request for metadata to `169.254.169.254` via HTTP
- The metadata request hits either the router or DHCP namespace depending on the route in the instance

- The metadata proxy service in the namespace sends the request to the Neutron metadata agent service via a Unix socket
- The Neutron metadata agent service forwards the request to the Nova metadata API service
- The Nova metadata API service responds to the request and forwards the response to the Neutron metadata agent service
- The Neutron metadata agent service sends the response back to the metadata proxy service in the namespace
- The metadata proxy service forwards the HTTP response to the instance
- The instance receives the metadata and/or the user data and continues the boot process

For proper operation of metadata services, both Neutron and Nova must be configured to communicate together with a shared secret. Neutron uses this secret to sign the `Instance-ID` header of the metadata request to prevent spoofing. On the `controller` node, update the following metadata options in the `[neutron]` section of the Nova configuration file located at `/etc/nova/nova.conf`:

```
[neutron]
...
service_metadata_proxy = true
metadata_proxy_shared_secret = MetadataSecret123
```

Next, update the `[DEFAULT]` section of the metadata agent configuration file located at `/etc/neutron/metadata_agent.ini` with the Neutron authentication details and the metadata proxy shared secret:

```
[DEFAULT]
...
nova_metadata_host = controller01
metadata_proxy_shared_secret = MetadataSecret123
```

Configuration options not mentioned here have sufficient default values and should not be changed unless your environment requires it.

Restarting the Neutron metadata agent

Use the following command to restart the `neutron-metadata-agent` and `nova-api` services on the `controller` node and to check the services' status:

```
# systemctl restart nova-api neutron-metadata-agent
# systemctl status neutron-metadata-agent
```

The output should resemble the following:

```
root@controller01:~# systemctl status neutron-metadata-agent
● neutron-metadata-agent.service - OpenStack Neutron Metadata Agent
   Loaded: loaded (/lib/systemd/system/neutron-metadata-agent.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2018-02-08 14:30:52 UTC; 1min 58s ago
     Process: 1269 ExecStartPre=/bin/chown neutron:adm /var/log/neutron (code=exited, status=0/SUCCESS)
     Process: 1264 ExecStartPre=/bin/chown neutron /var/lock/neutron /var/lib/neutron (code=exited, status=0/SUCCESS)
     Process: 1258 ExecStartPre=/bin/mkdir -p /var/lock/neutron /var/log/neutron /var/lib/neutron (code=exited, status=0/SUCCESS)
   Main PID: 1278 (neutron-metadat)
      Tasks: 2
     Memory: 108.9M
        CPU: 4.153s
    CGroup: /system.slice/neutron-metadata-agent.service
            └─1278 /usr/bin/python /usr/bin/neutron-metadata-agent --config-file=/etc/neutron/neutron.conf
              --config-file=/etc/neutron/metadata_agent.ini --log-file=/var/log/neutron/neutron-metadata-agent.log
            └─1301 /usr/bin/python /usr/bin/neutron-metadata-agent --config-file=/etc/neutron/neutron.conf
              --config-file=/etc/neutron/metadata_agent.ini --log-file=/var/log/neutron/neutron-metadata-agent.log
```

The agent should be in an active (running) status. Use the `openstack network agent list` command to verify that the service has checked in:

```
root@controller01:~# openstack network agent list --agent-type=metadata
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
bf8b2e8c-4633-4b8d-939a-06663aa88708	Metadata agent	controller01	None	: -)	UP	neutron-metadata-agent

A smiley face under the `Alive` column means that the agent is properly communicating with the `neutron-server` service.

If the services do not appear or have `xxx` under the `Alive` column, check the respective log files located at `/var/log/neutron` for assistance in troubleshooting. More information on the use of metadata can be found in [Chapter 7, Attaching Instances to Networks](#), and later chapters.

Interfacing with OpenStack Networking

The OpenStack Networking APIs can be accessed in a variety of ways, including via the Horizon dashboard, the `openstack` and `neutron` clients, the Python SDK, HTTP, and other methods. The following few sections will highlight the most common ways of interfacing with OpenStack Networking.

Using the OpenStack command-line interface

Prior to the `openstack` command-line client coming on the scene, each project was responsible for maintaining its own client. Each client often used its own syntax for managing objects and the lack of consistency between clients made life for users and operators difficult. The `openstack` client provides a consistent naming structure for commands and arguments, along with a consistent output format with optional parsable formats such as `csv`, `json`, and others. Not all APIs and services are supported by the `openstack` client, however, which may mean that a project-specific client is required for certain actions.

To invoke the `openstack` client, issue the `openstack` command at the Linux command line:

```
root@controller01:~# openstack  
(openstack)
```

The `openstack` shell provides commands that can be used to create, read, update, and delete the networking configuration within the OpenStack cloud. By typing a question mark or `help` within the `openstack` shell, a list of commands can be found. Additionally, running `openstack help` from the Linux command line provides a brief description of each command's functionality.

Using the Neutron command-line interface

Neutron provides a command-line client for interfacing with its API. Neutron commands can be run directly from the Linux command line, or the Neutron shell can be invoked by issuing the `neutron` command:

```
root@controller01:~# neutron  
neutron CLI is deprecated and will be removed in the future. Use openstack CLI instead.  
(neutron)
```



You must source the credentials file prior to invoking the `openstack` and `neutron` clients or an error will occur.

The `neutron` shell provides commands that can be used to create, read, update, and delete the networking configuration within the OpenStack cloud. By typing a question mark or `help` within the Neutron shell, a list of commands can be found. Additionally, running `neutron help` from the Linux command line provides a brief description of each command's functionality.

The `neutron` client has been deprecated in favor of the `openstack` command-line client. However, certain functions, including LBaaS-related commands, are not yet available within the `openstack` client and must be managed using the `neutron` client. In future releases of OpenStack, the `neutron` client will no longer be available.

Many of the commands listed within the client's `help` listing will be covered in subsequent chapters of this book.

Using the OpenStack Python SDK

The OpenStack Python SDK is available on PyPI and can be installed with the following command:

```
$ pip install openstacksdk
```

Documentation for the SDK is available at the following

URL: <https://developer.openstack.org/sdks/python/openstacksdk/users/index.html>.

Using the cURL utility

The OpenStack Networking API is REST-based and can be manipulated directly using HTTP. To make API calls using HTTP, you will need a token. Source the OpenStack credentials file and use the `openstack token issue` command shown here to retrieve a token:

```
root@controller01:~# openstack token issue --fit-width
```

Field	Value
expires	2018-02-08T14:34:52+0000
id	gAAAAABafFH8TfwDaM8NoFFXuS3sEBm3NnU6a5f1kRG5oyAxxk9v1Aj-SEEUT9su8rwQvJ7jLu5LcX_8rdf0iuPo_WhcjL5lQd-hVo4GGLTC2snqASwxf-h7HDHkgmLUJsnI02PQtPrDH9tCL6fdCxnDzGRoq4sGeY4FJqhYD55P0cDGBPMjHQU
project_id	9233b6b4f6a54386af63c0a7b8f043c2
user_id	8a679d8b7b574e54a5cfe02c422bbe68



The `--fit-width` argument is not necessary in normal operations, but helps make the token ID manageable for demonstration purposes.

To get a list of networks, the command should resemble the following:

```
$ curl -v -X GET -H 'X-Auth-Token: <token id>'
http://controller01:9696/v2.0/networks
```

The output will resemble the following:

```
root@controller01:~# curl -v -X GET \
> -H 'X-Auth-Token: gAAAAABafFF48g13m5lKB-<snip>_nGVdRhZrir8' \
> http://controller01:9696/v2.0/networks
* Trying 10.10.0.100...
* Connected to controller01 (10.10.0.100) port 9696 (#0)
> GET /v2.0/networks HTTP/1.1
> Host: controller01:9696
> User-Agent: curl/7.47.0
> Accept: */*
> X-Auth-Token: gAAAAABafFF48g13m5lKB-<snip>_nGVdRhZrir8
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< Content-Length: 15
< X-Openstack-Request-Id: req-2437f4ff-866f-4054-a49d-58ebacf336f0
< Date: Thu, 08 Feb 2018 13:39:48 GMT
<
* Connection #0 to host controller01 left intact
{"networks": []}
```

In this example, the Neutron API returned a 200 OK response in json format. No networks currently exist, so an empty list was returned. Neutron returns HTTP status codes that can be used to determine if the command was successful.

The OpenStack Networking API is documented at the following URL: <https://developer.openstack.org/api-ref/network/v2/>.

Summary

OpenStack Networking provides an extensible plugin architecture that makes implementing new network features possible. Neutron maintains the logical network architecture in its database, and network plugins and agents on each node are responsible for configuring virtual and physical network devices accordingly. Using the Modular Layer 2 (ML2) plugin, developers can spend less time implementing core Neutron API functionality and more time developing value-added features.

Now that OpenStack Networking services have been installed across all nodes in the environment, configuration of the Mechanism driver is all that remains before instances can be created. In the following two chapters, you will be guided through the configuration of the ML2 plugin and both the Linux bridge and Open vSwitch drivers and agents. We will also explore the differences between Linux bridge and Open vSwitch agents in terms of how they function and provide connectivity to instances.

4

Virtual Network Infrastructure Using Linux Bridges

One of the core functions of OpenStack Networking is to provide end-to-end network connectivity to instances running in the cloud. In [chapter 3 *Installing Neutron*](#), we installed the Neutron API service and the ML2 plugin across all nodes in the cloud. Beginning with this chapter, you will be introduced to networking concepts and architectures that Neutron relies on to provide connectivity to instances and other virtual devices.

The ML2 plugin for Neutron allows an OpenStack cloud to leverage multiple Layer 2 technologies simultaneously through the use of Mechanism drivers. In the next few chapters, we will look at multiple Mechanism drivers that extend the functionality of the ML2 network plugin, including the Linux bridge and Open vSwitch drivers.

In this chapter, you will do the following:

- Discover how Linux bridges are used to build a virtual network infrastructure
- Visualize traffic flow through virtual bridges
- Deploy the Linux bridge Mechanism driver and agent on hosts

Using the Linux bridge driver

The Linux bridge Mechanism driver supports a range of traditional and overlay networking technologies, and has support for the following types of drivers:

- Local
- Flat
- VLAN
- VXLAN

When a host is configured to use the ML2 plugin and the Linux bridge Mechanism driver, the Neutron agent on the host relies on the `bridge`, `8021q`, and `vxlan` kernel modules to properly connect instances and other network resources to virtual switches. These connections allow instances to communicate with other network resources in and out of the cloud. The Linux bridge Mechanism driver is popular for its dependability and ease of troubleshooting but lacks support for some advanced Neutron features such as distributed virtual routers.

In a Linux bridge-based network implementation, there are five types of interfaces managed by OpenStack Networking:

- Tap interfaces
- Physical interfaces
- VLAN interfaces
- VXLAN interfaces
- Linux bridges

A **tap interface** is created and used by a hypervisor such as QEMU/KVM to connect the guest operating system in a virtual machine instance to the underlying host. These virtual interfaces on the host correspond to a network interface inside the guest instance. An Ethernet frame sent to the tap device on the host is received by the guest operating system, and frames received from the guest operating system are injected into the host network stack.

A **physical interface** represents an interface on the host that is plugged into physical network hardware. Physical interfaces are often labeled `eth0`, `eth1`, `em0`, `em1`, and so on, and may vary depending on the host operating system.

Linux supports 802.1q VLAN tagging through the use of virtual VLAN interfaces. A VLAN interface can be created using `iproute2` commands or the traditional `vlan` utility and `8021q` kernel module. A VLAN interface is often labeled `ethX.<vlan>` and is associated with its respective physical interface, `ethX`.

A **VXLAN interface** is a virtual interface that is used to encapsulate and forward traffic based on parameters configured during interface creation, including a VXLAN Network Identifier (VNI) and VXLAN Tunnel End Point (VTEP). The function of a VTEP is to encapsulate virtual machine instance traffic within an IP header across an IP network. Traffic on the same VTEP is segregated from other VXLAN traffic using an ID provided by the VNI. The instances themselves are unaware of the outer network topology providing connectivity between VTEPs.

A **Linux bridge** is a virtual interface that connects multiple network interfaces. In Neutron, a bridge will usually include a physical interface and one or more virtual or tap interfaces. Linux bridges are a form of virtual switches.

Visualizing traffic flow through Linux bridges

For an Ethernet frame to travel from a virtual machine instance to a device on the physical network, it will pass through three or four devices inside the host:

Network type	Interface type	Interface name
all	tap	tapN
all	bridge	brqXXXX
vxlan	vxlan	vxlan-z (where Z is the VNI)
vlan	vlan	ethX.Y (where X is the physical interface and Y is the VLAN ID)
flat, vlan	physical	ethX (where X is the interface)

To help conceptualize how Neutron uses Linux bridges, a few examples of Linux bridge architectures are provided in the following sections.

VLAN

Imagine an OpenStack cloud that consists of a single `vlan` provider network with the segmentation ID `100`. Three instances have been connected to the network. As a result, the network architecture within the `compute` node resembles the following:

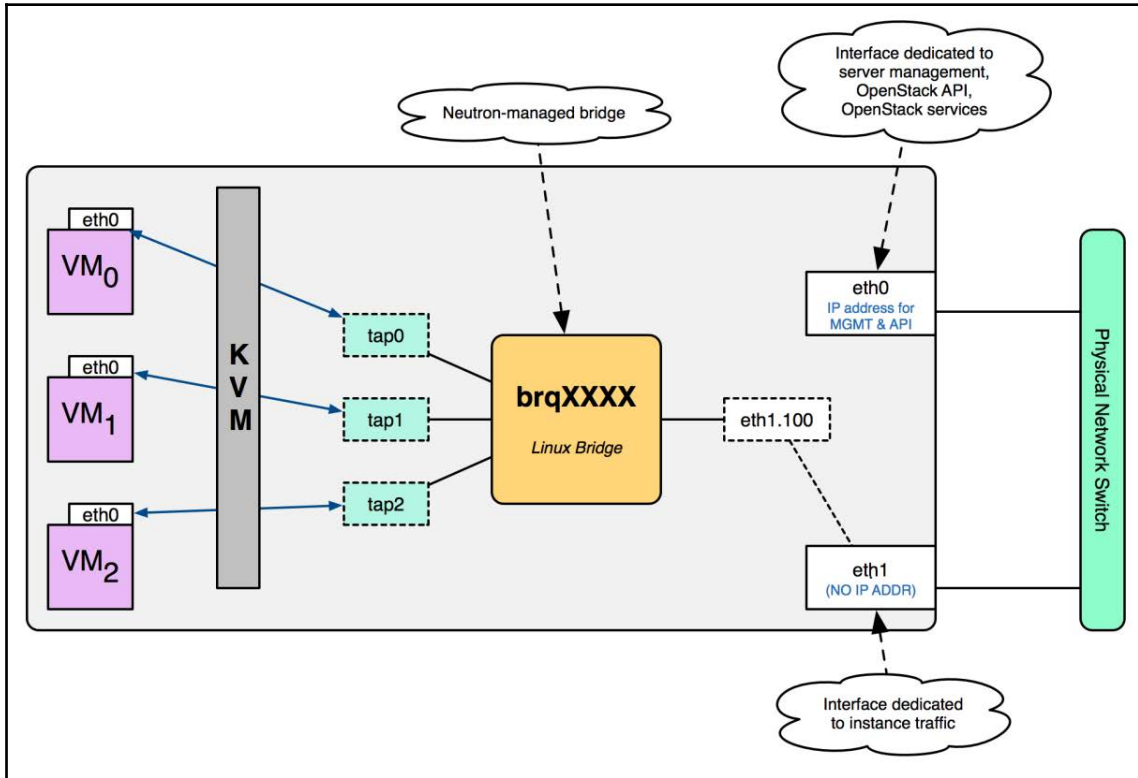


Figure 4.1

In Figure 4.1, three virtual machine instances are connected to a Linux bridge named `brqXXXX` via their respective tap interfaces. When the first instance was launched and connected to the network, Neutron created the bridge and a virtual interface named `eth1.100` and automatically connected the interface to the bridge. The `eth1.100` interface is bound to physical interface `eth1`. As traffic from instances traverses the Linux bridge and out toward the physical interface, interface `eth1.100` tags that traffic as VLAN 100 and drops it on `eth1`. Likewise, ingress traffic toward the instances through `eth1` is inversely untagged by `eth1.100` and sent to the appropriate instance connected to the bridge.

Using the `brctl show` command, the preceding diagram can be realized in the Linux CLI as the following:

```
# brctl show

bridge name      bridge id      STP enabled    interfaces
brqXXXX          <based on NIC> no             eth1.100
                                     tap0
                                     tap1
                                     tap2
```

The `bridge id` in the output is dynamically generated based on the parent NIC of the virtual VLAN interface. In this bridge, the parent interface is `eth1`.

The `bridge name`, beginning with the `brq` prefix, is generated based on the ID of the corresponding Neutron network it is associated with. In a Linux bridge architecture, every network uses its own bridge. Bridge names should be consistent across nodes for the same network.

On the physical switch, the necessary configuration to facilitate the networking described here will resemble the following:

```
vlan 100
  name VLAN_100

interface Ethernet1/3
  description Provider_Interface_eth1
  switchport
  switchport mode trunk
  switchport trunk allowed vlan add 100
  no shutdown
```

When configured as a trunk port, the provider interface can support multiple VLAN networks. If more than one VLAN network is needed, another Linux bridge will be created automatically that contains a separate VLAN interface. The new virtual interface, `eth1.101`, is connected to a new bridge, `brqYYYY`, as seen in Figure 4.2:

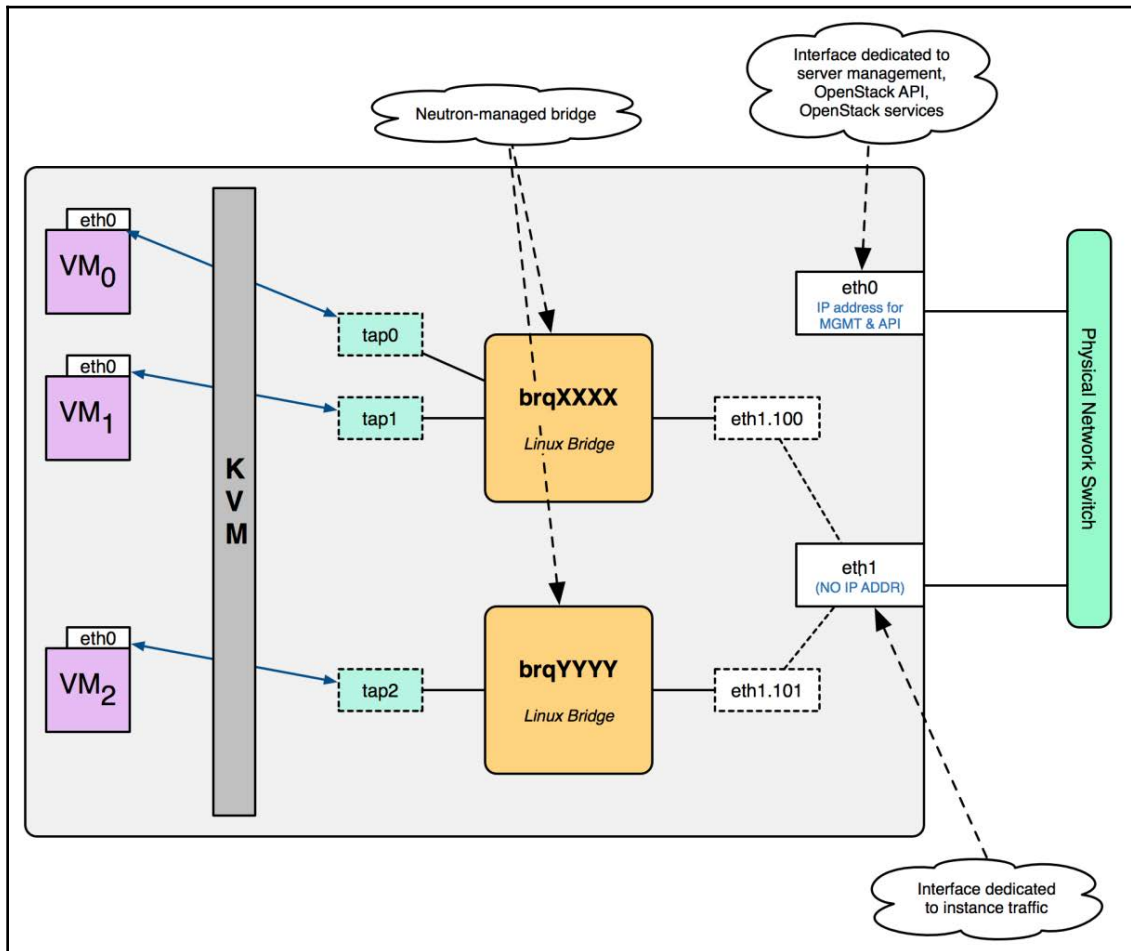


Figure 4.2

On the `compute` node, the preceding diagram can be realized as the following `brctl show` output:

# brctl show			
bridge name	bridge id	STP enabled	interfaces
brqXXXX	<based on NIC>	no	eth1.100
			tap0
			tap1
bridge name	bridge id	STP enabled	interfaces
brqYYYY	<based on NIC>	no	eth1.101
			tap2

On the physical switch, the necessary configuration to facilitate the networking described here will resemble the following:

```
vlan 100
  name VLAN_100
vlan 101
  name VLAN_101

interface Ethernet1/3
  description Provider_Interface_eth1
  switchport
  switchport mode trunk
  switchport trunk allowed vlan add 100-101
  no shutdown
```

Flat

A flat network in Neutron describes a network in which *no* VLAN tagging takes place. Unlike VLAN networks, flat networks require that the physical interface of the host associated with the network be connected directly to the bridge. This means that only a *single* flat network can exist per physical interface.

Figure 4.3 demonstrates a physical interface connected directly to a Neutron-managed bridge in a flat network scenario:

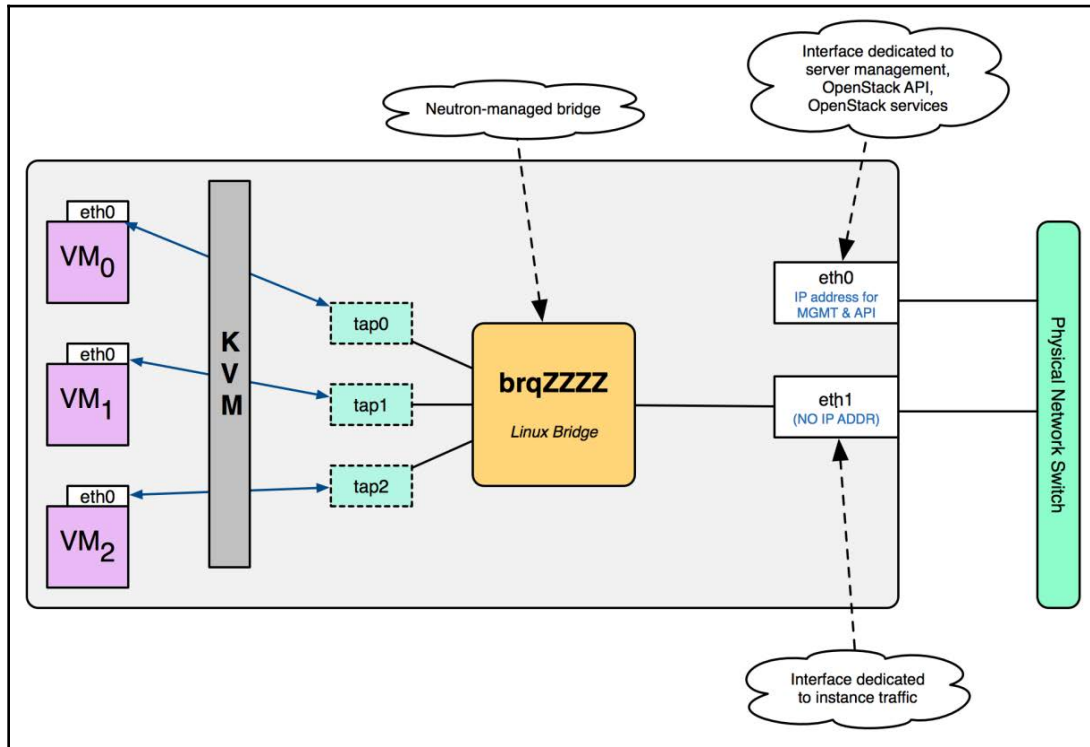


Figure 4.3

In Figure 4.3, `eth1` is connected to the bridge named `brqZZZZ` along with three tap interfaces that correspond to guest instances. No VLAN tagging for instance traffic takes place in this scenario.

On the compute node, the preceding diagram can be realized as the following `brctl show` output:

# brctl show			
bridge name	bridge id	STP enabled	interfaces
brqZZZ	<based on NIC>	no	eth1
			tap0
			tap1
			tap2

On the physical switch, the necessary configuration to facilitate the networking described here will resemble the following:

```
vlan 200
  name VLAN_200

interface Ethernet1/3
  description Provider_Interface_eth1
  switchport
  switchport mode trunk
  switchport trunk native vlan 200
  switchport trunk allowed vlan add 200
  no shutdown
```

Alternatively, the interface can also be configured as an access port:

```
interface Ethernet1/3
  description Provider_Interface_eth1
  switchport
  switchport mode access
  switchport access vlan 200
  no shutdown
```

Only one flat network is supported per provider interface. When configured as a trunk port with a native VLAN, the provider interface can support a single flat network and multiple VLAN networks. When configured as an access port, the interface can only support a single flat network and any attempt to tag traffic will fail.

When multiple flat networks are created, a separate physical interface must be associated with each flat network. Figure 4.4 demonstrates the use of a second physical interface required for the second flat network:

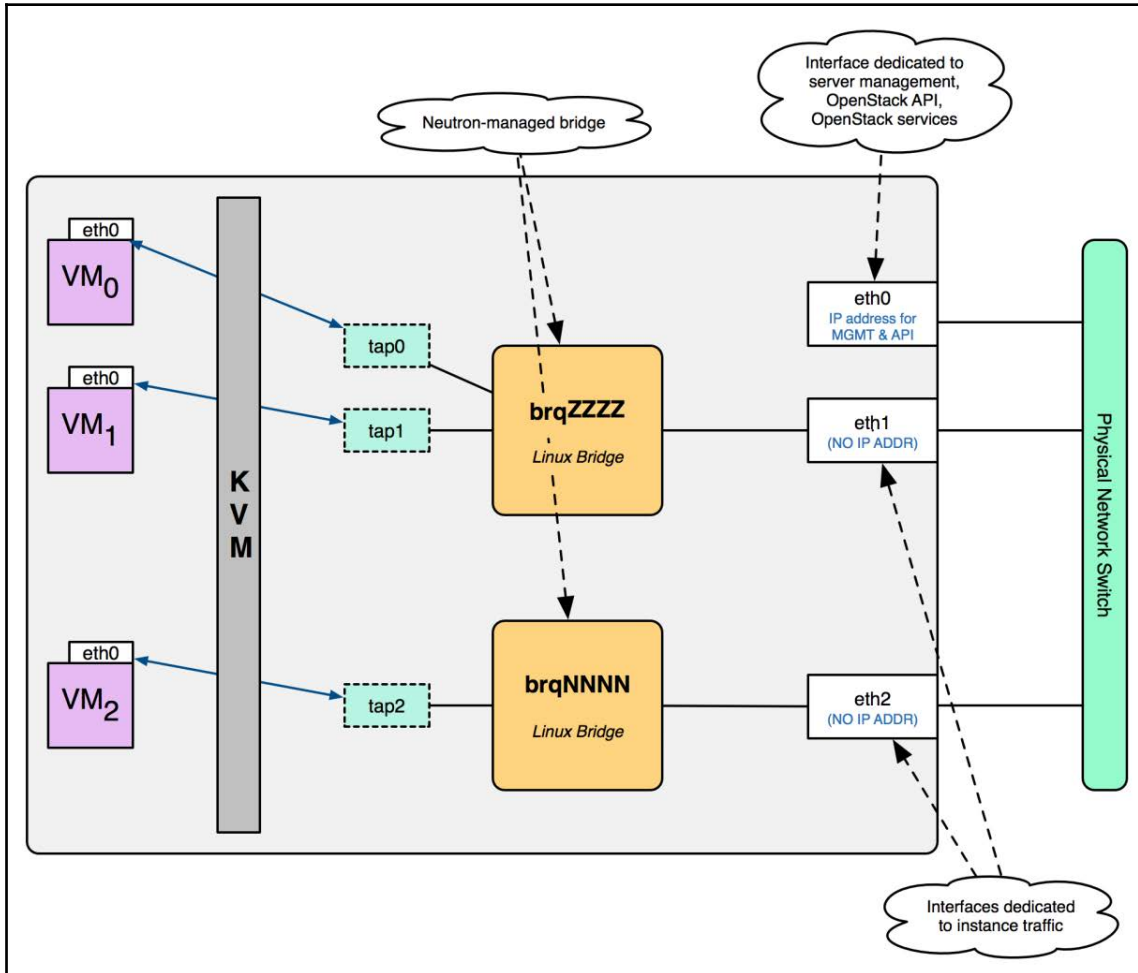


Figure 4.4

On the `compute` node, the use of two physical interfaces for separate flat networks can be realized as the following `brctl show` output:

```
# brctl show
```

bridge name	bridge id	STP enabled	interfaces
brqZZZZ	<based on NIC>	no	eth1 tap0 tap1
brqNNNN	<based on NIC>	no	eth2 tap2

On the physical switch, the necessary configuration to facilitate the networking described here will resemble the following:

```
vlan 200
    name VLAN_200
vlan 201
    name VLAN_201

interface Ethernet1/3
    description Flat_Provider_Interface_eth1
    switchport
    switchport mode trunk
    switchport trunk native vlan 200
    switchport trunk allowed vlan add 200
    no shutdown

interface Ethernet1/4
    description Flat_Provider_Interface_eth2
    switchport
    switchport mode trunk
    switchport trunk native vlan 201
    switchport trunk allowed vlan add 201
    no shutdown
```

With the two flat networks, the host does not perform any VLAN tagging on traffic traversing those bridges. Instances connected to the two bridges require a router to communicate with one another. Given the requirement for unique interfaces per flat network, flat networks do not scale well and are not common in production environments.

VXLAN

When VXLAN networks are created, the Neutron Linux bridge agent creates a corresponding VXLAN interface using `iproute2` user-space utilities and connects it to a Linux bridge. The VXLAN interface is programmed with information such as the VNI and local VTEP address.

When the L2 population driver is configured, Neutron prepopulates the forwarding database with static entries consisting of the MAC addresses of instances and their respective host VTEP addresses. As a packet from an instance traverses the bridge, the host determines how to forward the packet by consulting the forwarding table. If an entry is found, Neutron will forward the packet out of the corresponding local interface and encapsulate the traffic accordingly. To view the forwarding database table on each host, use the `bridge fdb show` command.

Potential issues when using overlay networks

One thing to be aware of when using overlay networking technologies is that the additional headers added to the encapsulated packets may cause them to exceed the **maximum transmission unit (MTU)** of the switchport or interface. The MTU is the largest size of packet or frame that can be sent over the network. Encapsulating a packet with VXLAN headers may cause the packet size to exceed the default maximum 1500-byte MTU. Connection issues caused by exceeding the MTU manifest themselves in strange ways, including partial failures in connecting to instances over SSH or a failure to transfer large payloads between instances, and more. To avoid this, consider lowering the MTU of interfaces within virtual machine instances from 1500 bytes to 1450 bytes to account for the overhead of VXLAN encapsulation to avoid connectivity issues.

An alternative to dropping the MTU is to increase the MTU of the interfaces used for the VTEPs. It is common to set a jumbo MTU of 9000 on VTEP interfaces and corresponding switchports to avoid having to drop the MTU inside instances. Increasing the MTU of the VTEP interfaces has also been shown to provide increases in network throughput when using overlay networks.

The DHCP agent can be configured to push a non-standard MTU to instances within the DHCP lease offer by modifying DHCP option 26. To configure a lower MTU, complete the following steps:

1. On the controller node, modify the DHCP configuration file at `/etc/neutron/dhcp_agent.ini` and specify a custom `dnsmasq` configuration file:

```
[DEFAULT]
dnsmasq_config_file = /etc/neutron/dnsmasq-neutron.conf
```

2. Next, create the custom `dnsmasq` configuration file at `/etc/neutron/dnsmasq-neutron.conf` and add the following contents:

```
dhcp-option-force=26,1450
```

3. Save and close the file. Restart the Neutron DHCP agent with the following command:

```
# systemctl restart neutron-dhcp-agent
```

Inside an instance running Linux, the MTU can be observed within the instance using the `ip link show <interface>` command.



A change to the `dnsmasq` configuration affects all networks, even instances on VLAN networks. Neutron ports can be modified individually to avoid this effect.

Local

When creating a local network in Neutron, it is not possible to specify a VLAN ID or even a physical interface. The Neutron Linux bridge agent will create a bridge and connect only the tap interface of the instance to the bridge. Instances in the same local network on the same node will be connected to the same bridge and are free to communicate with one another. Because the host does not have a physical or virtual VLAN interface connected to the bridge, traffic between instances is limited to the host on which the instances reside. Traffic between instances in the same local network that reside on different hosts will be unable to communicate with one another.

Figure 4.5 demonstrates the lack of physical or virtual VLAN interfaces connected to the bridge:

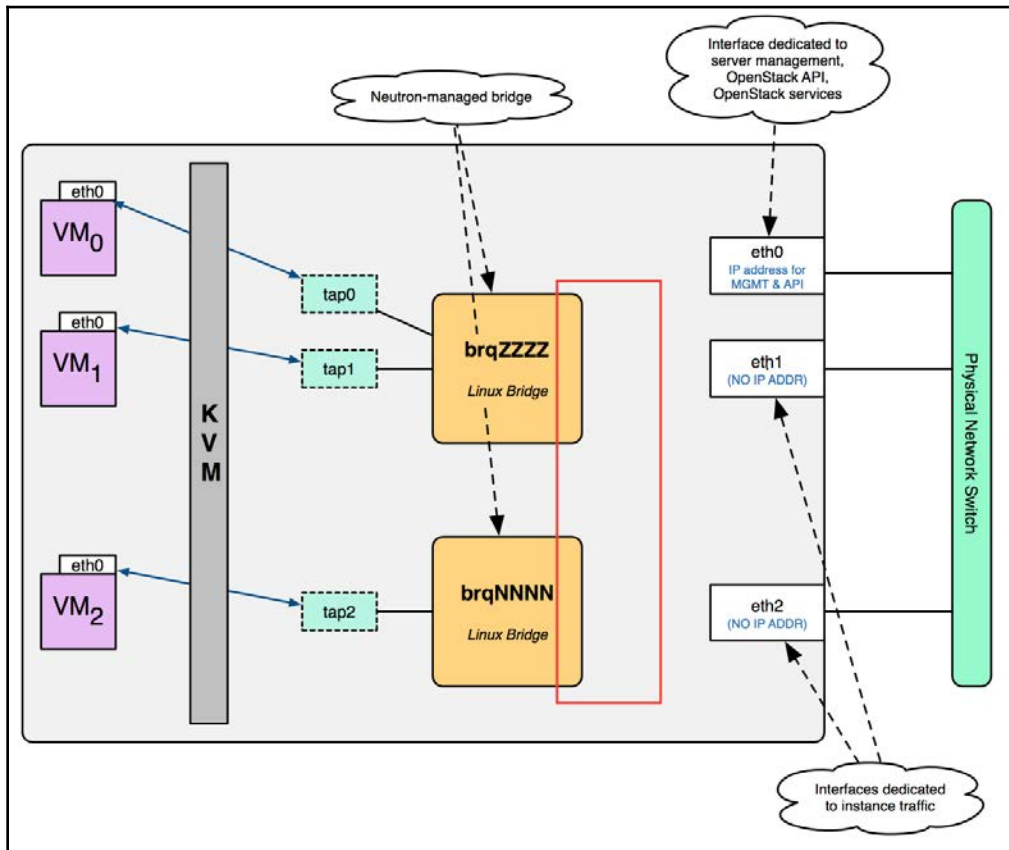


Figure 4.5

In Figure 4.5, two local networks have been created along with their respective bridges, `brqZZZZ` and `brqNNNN`. Instances connected to the same bridge can communicate with one another, but nothing else outside of the bridge. There is no mechanism to permit traffic between instances on different bridges or hosts when using local networks.

Some application architectures may require multiple instances be deployed on the same host without the need for cross-host communication. A local network might make sense in this scenario and can be used to avoid the consumption of precious VLAN IDs or VXLAN overhead.

Configuring the ML2 networking plugin

Before you can build network resources in an OpenStack cloud, a network plugin must be defined and configured. The ML2 plugin provides a common framework that allows multiple drivers to interoperate with one another. In this section, we will look at how to configure the Linux bridge ML2 driver and agent on the `controller01` and `compute01` hosts.



Configuring the Linux bridge and Open vSwitch drivers for simultaneous operation will be discussed in this book but may not be appropriate for a production environment. To make things simple, I recommend deploying the Linux bridge driver if distributed virtual routers are not required. The configuration and architecture of distributed virtual routers are outlined in Chapter 12, *Distributed Virtual Routers*.

Configuring the bridge interface

In this installation, physical network interface `eth2` will be utilized as the **provider interface** for VLAN and flat networks. Neutron will be responsible for configuring VLAN interfaces off `eth2` once the initial network configuration has been completed.

On the `controller01` and `compute01` nodes, configure the `eth2` interface within the `/etc/network/interfaces` file as follows:

```
auto eth2
iface eth2 inet manual
```

Close and save the file, and bring the interface up with the following command:

```
# ip link set dev eth2 up
```

Confirm the interface is in an UP state using the `ip link show dev eth2` command. If the interface is up, it is ready for use in bridges that Neutron will create and manage.



Because the interface will be used in a bridge, an IP address cannot be applied directly to the interface. If there is an IP address applied to `eth2`, it will become inaccessible once the interface is placed in a bridge. If an IP is required, consider moving it to an interface not required for Neutron networking.

Configuring the overlay interface

In this installation, physical network interface `eth1` will be utilized as the **overlay interface** for overlay networks using VXLAN. Neutron will be responsible for configuring VXLAN interfaces once the initial network configuration has been completed.

On the `controller01` and `compute01` nodes, configure the `eth1` interface within the `/etc/network/interfaces` file as follows:

```
auto eth1
iface eth1 inet static
    address 10.20.0.X/24
```

Use the following table for the appropriate address, and substitute for `x` where appropriate:

Host	Address
<code>controller01</code>	<code>10.20.0.100</code>
<code>compute01</code>	<code>10.20.0.101</code>

Close and save the file, and bring the interface up with the following command:

```
# ip link set dev eth1 up
```

Confirm the interface is in an UP state and that the address has been set using the `ip addr show dev eth1` command. Ensure both hosts can communicate over the newly configured interface by pinging `compute01` from the `controller01` node:

```
root@controller01:~# ping -c 5 10.20.0.101
PING 10.20.0.101 (10.20.0.101) 56(84) bytes of data.
64 bytes from 10.20.0.101: icmp_seq=1 ttl=64 time=0.448 ms
64 bytes from 10.20.0.101: icmp_seq=2 ttl=64 time=0.522 ms
64 bytes from 10.20.0.101: icmp_seq=3 ttl=64 time=0.624 ms
64 bytes from 10.20.0.101: icmp_seq=4 ttl=64 time=0.524 ms
64 bytes from 10.20.0.101: icmp_seq=5 ttl=64 time=0.545 ms

--- 10.20.0.101 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4083ms
rtt min/avg/max/mdev = 0.448/0.532/0.624/0.061 ms
```



If you experience any issues communicating across this interface, you *will* experience issues with VXLAN networks created with OpenStack Networking. Any issues should be corrected before continuing.

ML2 plugin configuration options

The ML2 plugin was installed in the previous chapter and its configuration file located at `/etc/neutron/plugins/ml2/ml2_conf.ini` must be configured before OpenStack Networking services can be used. The ML2 plugin configuration file is referenced by the `neutron-server` service may be referenced by multiple agents, including Linux bridge and Open vSwitch agents. Agent-specific changes will be made in their respective configuration files on each host.

The `ml2_conf.ini` file is broken into configuration blocks and contains the following commonly used options:

```
[ml2]
...
type drivers = ...
mechanism drivers = ...
tenant_network_types = ...
extension_drivers = ...

[ml2_type_flat]
...
flat_networks = ...

[ml2_type_vlan]
...
network_vlan_ranges = ...

[ml2_type_vxlan]
...
vni_ranges = ...
vxlan_group = ...

[securitygroup]
...
enable_security_group = ...
enable_ipset = ...
```



Configuration options must remain in the appropriate block, otherwise Neutron services may not start or operate properly.

Type drivers

Type drivers describe the type of networks that can be created and implemented by Mechanism drivers. Type drivers included with the ML2 plugin include `local`, `flat`, `vlan`, `gre`, `vxlan`, and `geneve`. Not all Mechanism drivers can implement all types of networks, however. The Linux bridge driver lacks support for GENEVE and GRE networks.

Update the ML2 configuration file on the `controller01` node and add the following `type_drivers`:

```
[ml2]
...
type_drivers = local,flat,vlan,vxlan
```

Mechanism drivers

Mechanism drivers are responsible for implementing networks described by the type driver. Mechanism drivers included with the ML2 plugin include `linuxbridge`, `openvswitch`, and `l2population`.

Update the ML2 configuration file on the `controller01` node and add the following `mechanism_drivers`:

```
[ml2]
...
mechanism_drivers = linuxbridge,l2population
```



The Neutron Linux bridge agent requires specific configuration options that will be discussed later in this chapter.

Using the L2 population driver

The L2 population driver was introduced in the Havana release of OpenStack alongside the ML2 plugin. It enables broadcast, multicast, and unicast traffic to scale on large overlay networks constructed by OpenStack.

The goal of the L2 population driver is to inhibit costly switch learning behaviors by pre-populating bridge forwarding and IP neighbor (ARP) tables on all hosts. Because Neutron is seen as a source of truth for the logical layout of networks and instances created by users, it can easily pre-populate forwarding tables consisting of MAC addresses and destination VTEPs with that information. The L2 population driver also implements an ARP proxy on each host, eliminating the need to broadcast ARP requests across the overlay network. Each compute or network node is able to intercept an ARP request from an instance or router and proxy the response to the requestor. However, the L2 population driver does have limitations that will be discussed later in this chapter.

An alternative to using the L2 population driver is to rely on the use of multicast to propagate forwarding database information between hosts. Each host should be configured to subscribe to a multicast group configured outside of OpenStack. If not properly configured, broadcast messages may be used in lieu of multicast and may cause unnecessary chatter on the network. The configuration of multicast is outside the scope of this book.

Tenant network types

The `tenant_network_types` configuration option describes the type of networks that a tenant or project can create. When using the Linux bridge Mechanism driver, the supported tenant network types are `flat`, `vlan`, `local`, `vxlan`, and `none`.

The configuration option takes values in an ordered list, such as `vlan, vxlan`. In this example, when a user creates a network, Neutron will automatically provision a VLAN network and ID without any user interaction. When all available VLAN IDs have been allocated, Neutron will allocate a network of the next type in the list. In this case, a VXLAN network and VNI would be allocated. When all segmentation IDs of any listed network type have been allocated, users will no longer be able to create networks and an error will be presented to the user.



Users with the `admin` role can override the behavior of `tenant_network_types` by specifying provider attributes during the network creation process.

Update the ML2 configuration file on the `controller` node and add the following `tenant_network_types` configuration to the `[ml2]` section:

```
[ml2]
...
tenant_network_types= vlan,vxlan
```

If at any time you wish to change the value of `tenant_network_types`, edit the plugin configuration file accordingly on all nodes and restart the `neutron-server` service.

Flat networks

The `flat_networks` configuration option defines interfaces that support the use of untagged networks, commonly referred to as a native or access VLAN. This option requires that a provider label be specified. A **provider label** is an arbitrary label or name that is mapped to a physical interface or bridge on the host. These mappings will be discussed in further detail later in this chapter.

In the following example, the `physnet1` interface has been configured to support a flat network:

```
flat_networks = physnet1
```

Multiple interfaces can be defined using a comma-separated list:

```
flat_networks = physnet1,physnet2
```



Due to the lack of an identifier to segregate untagged traffic on the same interface, an interface can only support a single flat network.

In this environment, the `flat_networks` option can remain *unconfigured*.

Network VLAN ranges

The `network_vlan_ranges` configuration option defines a range of VLANs that project networks will be associated with upon their creation when `tenant_network_types` is `vlan`. When the number of available VLANs reaches zero, tenants will no longer be able to create VLAN networks.

In the following example, VLAN IDs 40 through 43 are available for tenant network allocation:

```
network_vlan_ranges = physnet1:40:43
```

Non-contiguous VLANs can be allocated by using a comma-separated list:

```
network_vlan_ranges = physnet1:40:43,physnet1:51:55
```

In this specific deployment, the provider label `physnet1` will be used with VLANs 40 through 43. Those VLANs will be automatically assigned to `vlan` networks upon creation unless overridden by a user with the `admin` role.

Update the ML2 configuration file on the `controller01` node and add the following `network_vlan_ranges` to the `[ml2_type_vlan]` section:

```
[ml2_type_vlan]
...
network_vlan_ranges = physnet1:40:43
```

VNI ranges

When VXLAN networks are created, each network is assigned a unique segmentation ID that is used to encapsulate traffic. When the Linux bridge Mechanism driver is used, the segmentation ID is used when creating the respective VXLAN interface on each host.

The `vni_ranges` configuration option is a comma-separated list of ID ranges that are available for project network allocation when `tunnel_type` is set to `vxlan`.

In the following example, segmentation IDs 1 through 1000 are reserved for allocation to tenant networks upon creation:

```
vni_ranges = 1:1000
```

The `vni_ranges` option supports non-contiguous IDs using a comma-separated list as follows:

```
vni_ranges = 1:1000,2000:2500
```

Update the ML2 configuration file on the `controller01` node and add the following `vni_ranges` to the `[ml2_type_vxlan]` section:

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



The 24-bit VNI field in the VXLAN header supports up to approximately 16 million unique identifiers.

Security groups

The `enable_security_group` configuration option instructs Neutron to enable or disable security group-related API functions. The option is set to `true` by default.

The `enable_ipset` configuration option instructs Neutron to enable or disable the `ipset` extension for iptables when the iptables firewall driver is used. The use of ipsets allows for the creation of firewall rules that match entire sets of addresses at once rather than having individual lines per address, making lookups very efficient compared to traditional linear lookups. The option is set to `true` by default.



If at any time the ML2 configuration file is updated, you must restart the `neutron-server` service and respective Neutron agent for the changes to take effect.

Configuring the Linux bridge driver and agent

The Linux bridge Mechanism driver is included with the ML2 plugin, and was installed in Chapter 3, *Installing Neutron*. The following sections will walk you through the configuration of OpenStack Networking to utilize the Linux bridge driver and agent.



While the Linux bridge and Open vSwitch agents and drivers can coexist in the same environment, they should not be installed and configured simultaneously on the same host.

Installing the Linux bridge agent

To install the Neutron Linux bridge agent, issue the following command on `controller01` and `compute01`:

```
# apt install neutron-plugin-linuxbridge-agent
```



If prompted to overwrite existing configuration files, type `N` at the `[default=N]` prompt.

Updating the Linux bridge agent configuration file

The Linux bridge agent uses a configuration file located at `/etc/neutron/plugins/ml2/linuxbridge_agent.ini`. The most common options are as follows:

```
[linux_bridge]
...
physical_interface_mappings = ...

[vxlan]
...
enable_vxlan = ...
vxlan_group = ...
l2_population = ...
local_ip = ...
arp_responder = ...

[securitygroup]
...
firewall_driver = ...
```

Physical interface mappings

The `physical_interface_mappings` configuration option describes the mapping of an artificial label to a physical interface in the server. When networks are created, they are associated with an interface label, such as `physnet1`. The label `physnet1` is then mapped to a physical interface, such as `eth2`, by the `physical_interface_mappings` option. This mapping can be observed in the following example:

```
physical_interface_mappings = physnet1:eth2
```

The chosen label(s) must be consistent between all nodes in the environment that are expected to handle traffic for a given network created with Neutron. However, the physical interface mapped to the label may be different. A difference in mappings is often observed when one node maps `physnet1` to a gigabit interface while another maps `physnet1` to a 10-gigabit interface.

Multiple interface mappings are allowed, and can be added to the list using a comma-separated list:

```
physical_interface_mappings = physnet1:eth2,physnet2:bond2
```

In this installation, the `eth2` interface will be utilized as the physical network interface, which means that traffic for any networks associated with `physnet1` will traverse `eth2`. The physical switch port connected to `eth2` must support 802.1q VLAN tagging if VLAN networks are to be created by tenants.

Configure the Linux bridge agent to use `physnet1` as the physical interface label and `eth2` as the physical network interface by updating the ML2 configuration file accordingly on `controller01` and `compute01`:

```
[linux_bridge]
...
physical_interface_mappings = physnet1:eth2
```

Enabling VXLAN

To enable support for VXLAN networks, the `enable_vxlan` configuration option must be set to `true`. Update the `enable_vxlan` configuration option in the `[vxlan]` section of the ML2 configuration file accordingly on `Controller01` and `compute01`:

```
[vxlan]
...
enable_vxlan = true
```

L2 population

To enable support for the L2 population driver, the `l2_population` configuration option must be set to `true`. Update the `l2_population` configuration option in the `[vxlan]` section of the ML2 configuration file accordingly on `controller01` and `compute01`:

```
[vxlan]
...
l2_population = true
```

A useful feature of the L2 population driver is its ARP responder functionality that helps avoid the broadcasting of ARP requests across the overlay network. Each `compute` node can proxy ARP requests from virtual machines and provide them with replies, all without that traffic leaving the host.

To enable the ARP responder, update the following configuration option:

```
[vxlan]
...
arp_responder = true
```


The ARP responder has known incompatibilities with the `allowed-address-pairs` extension on systems using the Linux bridge agent, however. The `vxlan` kernel module utilized by the Linux bridge agent does not support dynamic learning when ARP responder functionality is enabled. As a result, when an IP address moves between virtual machines, the forwarding database may not be updated with the MAC address and respective VTEP of the destination host as Neutron is not notified of this change. If `allowed-address-pairs` functionality is required, my recommendation is to disable the ARP responder until this behavior is changed.

Local IP

The `local_ip` configuration option specifies the local IP address on the node that will be used to build the overlay network between hosts. Refer to [Chapter 1, Introduction to OpenStack Networking](#), for ideas on how the overlay network should be architected. In this installation, all guest traffic through overlay networks will traverse a dedicated network over the `eth1` interface configured earlier in this chapter.

Update the `local_ip` configuration option in the `[vxlan]` section of the ML2 configuration file accordingly on the `controller01` and `compute01` hosts:

```
[vxlan]
...
local_ip = 10.20.0.X
```

The following table provides the interfaces and addresses to be configured on each host. Substitute for `X` where appropriate:

Hostname	Interface	IP address
controller01	eth1	10.20.0.100
compute01	eth1	10.20.0.101

Firewall driver

The `firewall_driver` configuration option instructs Neutron to use a particular firewall driver for security group functionality. There may be different firewall drivers configured based on the Mechanism driver in use.

Update the ML2 configuration file on `controller01` and `compute01` and define the appropriate `firewall_driver` in the `[securitygroup]` section on a single line:

```
[securitygroup]
...
firewall_driver = iptables
```

If you do not want to use a firewall, and want to disable the application of security group rules, set `firewall_driver` to `noop`.

Configuring the DHCP agent to use the Linux bridge driver

For Neutron to properly connect DHCP namespace interfaces to the appropriate network bridge, the DHCP agent on the `controller` node must be configured to use the Linux bridge interface driver.

On the `controller` node, update the `interface_driver` configuration option in the Neutron DHCP agent configuration file at `/etc/neutron/dhcp_agent.ini` to use the Linux bridge interface driver:

```
[DEFAULT]
...
interface_driver = linuxbridge
```



The interface driver will vary based on the plugin agent in use on the node hosting the DHCP agent.

Restarting services

Some services must be restarted for the changes made to take effect. The following services should be restarted on `controller01` and `compute01`:

```
# systemctl restart neutron-linuxbridge-agent
```

The following services should be restarted on the `controller01` node:

```
# systemctl restart neutron-server neutron-dhcp-agent
```

Verifying Linux bridge agents

To verify the Linux bridge network agents have properly checked in, issue the `openstack network agent list` command on the `controller` node:

```
root@controller01:~# openstack network agent list
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
75c095d1-df2a-491e-bd87-9b8b9ba4f9d4	DHCP agent	controller01	nova	:~)	UP	neutron-dhcp-agent
b0a0156d-1199-4324-8bff-c6c5f7eea1f2	Linux bridge agent	compute01	None	:~)	UP	neutron-linuxbridge-agent
bf8b2e8c-4633-4b8d-939a-06663aa88708	Metadata agent	controller01	None	:~)	UP	neutron-metadata-agent
dc2ba98c-7304-411f-9dc7-15956177f7c3	Linux bridge agent	controller01	None	:~)	UP	neutron-linuxbridge-agent

The Neutron Linux bridge agents on the `controller01` and `compute01` nodes should be visible in the output with a state of `UP`. If an agent is not present, or the state is `DOWN`, you will need to troubleshoot agent connectivity issues by observing log messages found in `/var/log/neutron/neutron-linuxbridge-agent.log` on the respective host.

Summary

In this chapter, we discovered how Neutron leverages Linux bridges and virtual interfaces to provide network connectivity to instances. The Linux bridge driver supports many different network types, including tagged, untagged, and overlay networks, and I will demonstrate in later chapters how these differ when we launch instances on those networks.

In the next chapter, you will learn the difference between a Linux bridge and Open vSwitch implementation and will be guided through the process of installing the Open vSwitch driver and agent on two additional `compute` nodes and a network node dedicated to distributed virtual routing functions.

5

Building a Virtual Switching Infrastructure Using Open vSwitch

In [Chapter 4](#), *Virtual Network Infrastructure Using Linux Bridges*, we looked at how the Linux bridge mechanism driver and agent build a virtual network infrastructure using different types of interfaces and Linux bridges. In this chapter, you will be introduced to the Open vSwitch mechanism driver and its respective agent, which utilizes Open vSwitch as the virtual switching technology that connect instances and hosts to the physical network.

In this chapter, you will do the following:

- Discover how Open vSwitch is used to build a virtual network infrastructure
- Visualize traffic flow through virtual switches
- Deploy the Open vSwitch mechanism driver and agent on hosts

Using the Open vSwitch driver

The Open vSwitch mechanism driver supports a range of traditional and overlay networking technologies, and has support for the following types of drivers:

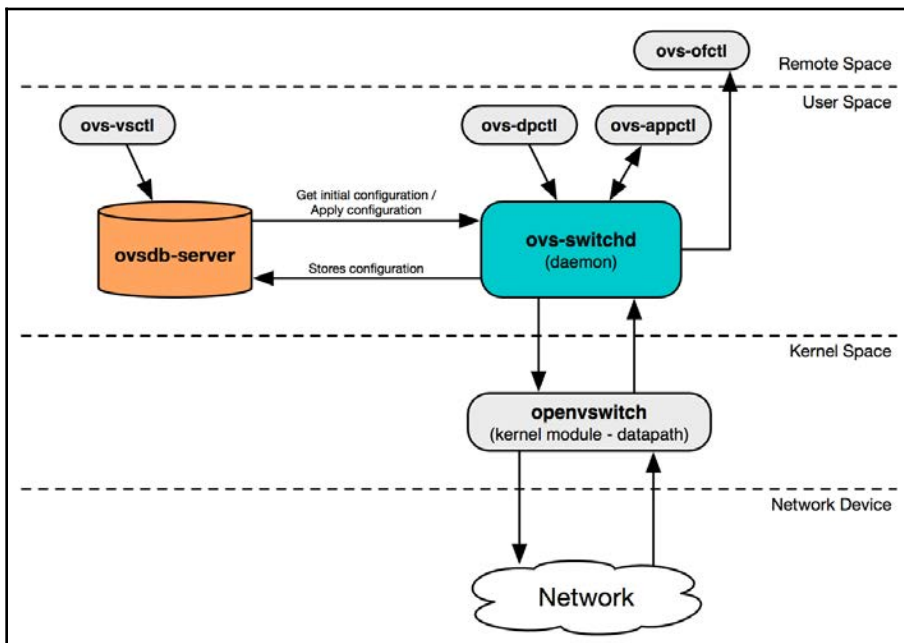
- Local
- Flat
- VLAN
- VXLAN
- GRE

Within OpenStack Networking, Open vSwitch operates as a software switch that uses virtual network bridges and flow rules to forward packets between hosts. Although it is capable of supporting many technologies and protocols, only a subset of Open vSwitch features are leveraged by OpenStack Networking.

The following are three main components of Open vSwitch:

- **Kernel module:** The `openvswitch` kernel module is the equivalent of ASICs on a hardware switch. It is the data plane of the switch where all packet processing takes place.
- **vSwitch daemon:** The `ovs-vswitchd` daemon is a Linux process that runs in user space on every physical host and dictates how the kernel module will be programmed.
- **Database server:** An OpenStack/Open vSwitch implementation uses a local database on every physical host called the **Open vSwitch Database Server (OVSDB)**, which maintains the configuration of the virtual switches.

A high-level architecture diagram of the preceding components can be seen here:



The Neutron Open vSwitch agent, `neutron-openvswitch-agent`, is a service that's configured on hosts using the Open vSwitch mechanism driver and is responsible for managing the implementation of networks and related interfaces. The agent connects tap interfaces to Open vSwitch or Linux bridges, depending on the firewall configuration, and programs flows using utilities such as `ovs-vsctl` and `ovs-ofctl` based on data provided by the `neutron-server` service.

In an Open vSwitch-based network implementation, there are five distinct types of virtual networking devices, as follows:

- Tap devices
- Linux bridges
- Virtual ethernet cables
- OVS bridges
- OVS patch ports

Tap devices and Linux bridges were described briefly in the previous section, and their use in an Open vSwitch-based network remains the same. Virtual Ethernet (**veth**) cables are virtual interfaces that mimic network patch cables. An Ethernet frame sent to one end of a veth cable is received by the other end, just like a real network patch cable. Neutron makes use of veth cables when making connections between network namespaces and Linux bridges, as well as when connecting Linux bridges to Open vSwitch switches.

Neutron connects interfaces used by DHCP or router namespaces and instances to OVS bridge ports. The ports themselves can be configured much like a physical switch port. Open vSwitch maintains information about connected devices, including MAC addresses and interface statistics.

Open vSwitch has a built-in port type that mimics the behavior of a Linux veth cable, but is optimized for use with OVS bridges. When connecting two Open vSwitch bridges, a port on each switch is reserved as a **patch port**. Patch ports are configured with a peer name that corresponds to the patch port on the other switch. Graphically, it looks something like this:

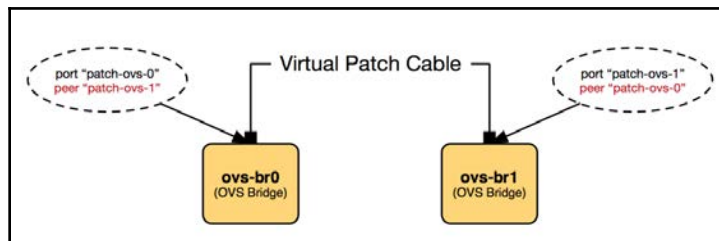


Figure 5.1

In the preceding diagram, two OVS bridges are cross-connected via a patch port on each switch. Open vSwitch patch ports are used to connect Open vSwitch bridges to each other, while Linux veth interfaces are used to connect Open vSwitch bridges to Linux bridges, or Linux bridges to other Linux bridges.

Basic OpenvSwitch commands

Open vSwitch includes utilities that can be used to manage virtual switches created by users, including those created by the OpenStack Networking agent. These commands are useful when troubleshooting issues that inevitably occur on the network.

Base commands

The majority of Open vSwitch configuration and troubleshooting can be accomplished with the following commands:

- `ovs-vsctl`: A tool used to configure the `ovs-vswitchd` database
- `ovs-ofctl`: A tool used for monitoring and administering OpenFlow switches
- `ovs-dpctl`: A tool used to administer Open vSwitch data paths
- `ovs-appctl`: A tool used to query and manage Open vSwitch daemons

ovs-vsctl

The `ovs-vsctl` tool is used to configure and view OVS bridge/switch operations. With this tool, users can configure ports on a switch, create and delete virtual switches, create bonds, and manage VLAN tagging on ports.

Useful commands include the following:

- `ovs-vsctl show`: Prints a brief overview of the switch database configuration, including ports, VLANs, and so on
- `ovs-vsctl list-br`: Prints a list of configured bridges
- `ovs-vsctl list-ports <bridge>`: Prints a list of ports on the specified bridge
- `ovs-vsctl list interface`: Prints a list of interfaces along with statistics and other data

ovs-ofctl

The `ovs-ofctl` tool is used to monitor and administer OpenFlow switches. The Neutron Open vSwitch agent uses `ovs-ofctl` to program flows on the virtual switches that are used to dictate traffic flow, perform VLAN tagging, perform NAT, and more.

Useful commands include the following:

- `ovs-ofctl show <bridge>`: Shows OpenFlow features, actions, and port descriptions for the specified bridge.
- `ovs-ofctl dump-flows <bridge> <flow>`: Prints the flow entries for the specified bridge. If the flow is specified, only that flow is shown.
- `ovs-ofctl dump-ports-desc <bridge>`: Prints port statistics for the specified bridge, including the state, peer, and speed of the interface.

ovs-dpctl

The `ovs-dpctl` tool is used to administer and query Open vSwitch data paths. Unlike `ovs-ofctl`, `ovs-dpctl` reflects flows for packets that have been matched by actual traffic traversing the system.

Useful commands include the following:

- `ovs-dpctl dump-flows`: Shows the flow table data for all flows traversing the system

ovs-appctl

The `ovs-appctl` tool is used to query and manage Open vSwitch daemons, including `ovs-vswitchd`, `ovs-controller`, and others.

Useful commands include the following:

- `ovs-appctl bridge/dump-flows <bridge>`: Dumps OpenFlow flows on the specified bridge
- `ovs-appctl dpif/dump-flows <bridge>`: Dumps data path flows on the specified bridge
- `ovs-appctl ofproto/trace <bridge> <flow>`: Shows the entire flow field of a given flow, including the matched rule and the action taken



Many of these commands are used by the Neutron Open vSwitch agent to program virtual switches and can often be used by operators to troubleshoot network connectivity issues along the way. Familiarizing yourself with these commands and their output is highly recommended.

Visualizing traffic flow when using Open vSwitch

When using the Open vSwitch driver, for an Ethernet frame to travel from the virtual machine instance to the physical network, it will pass through many different interfaces, including the following:

Network Type	Interface Type	Interface Name
all	tap	tapN
all	bridge	qbrXXXX (only used with the iptables firewall driver)
all	veth	qvbXXXX, qvoXXXX (only used with the iptables firewall driver)
all	vSwitch	br-int
flat, vlan	vSwitch	br-ex (user-configurable)
vxlan, gre	vSwitch	br-tun
flat, vlan	patch	int-br-ethX, phy-br-ethX
vxlan, gre	patch	patch-tun, patch-int
flat, vlan	physical	ethX (where X is the interface)

The Open vSwitch bridge `br-int` is known as the **integration bridge**. The integration bridge is the central virtual switch that most virtual devices are connected to, including instances, DHCP servers, routers, and more. When Neutron security groups are enabled and the iptables firewall driver is used, instances are not directly connected to the integration bridge. Instead, instances are connected to individual Linux bridges that are cross-connected to the integration bridge using a veth cable.



The `openvswitch` firewall driver is an alternative driver that implements security group rules using OpenFlow rules, but this is outside the scope of this book.

The Open vSwitch bridge `br-ethX` is known as the **provider bridge**. The provider bridge provides connectivity to the physical network via a connected physical interface. The provider bridge is also connected to the integration bridge by a virtual patch cable which is provided by patch ports `int-br-ethX` and `phy-br-ethX`.

A visual representation of the architecture described here can be seen in the following diagram:

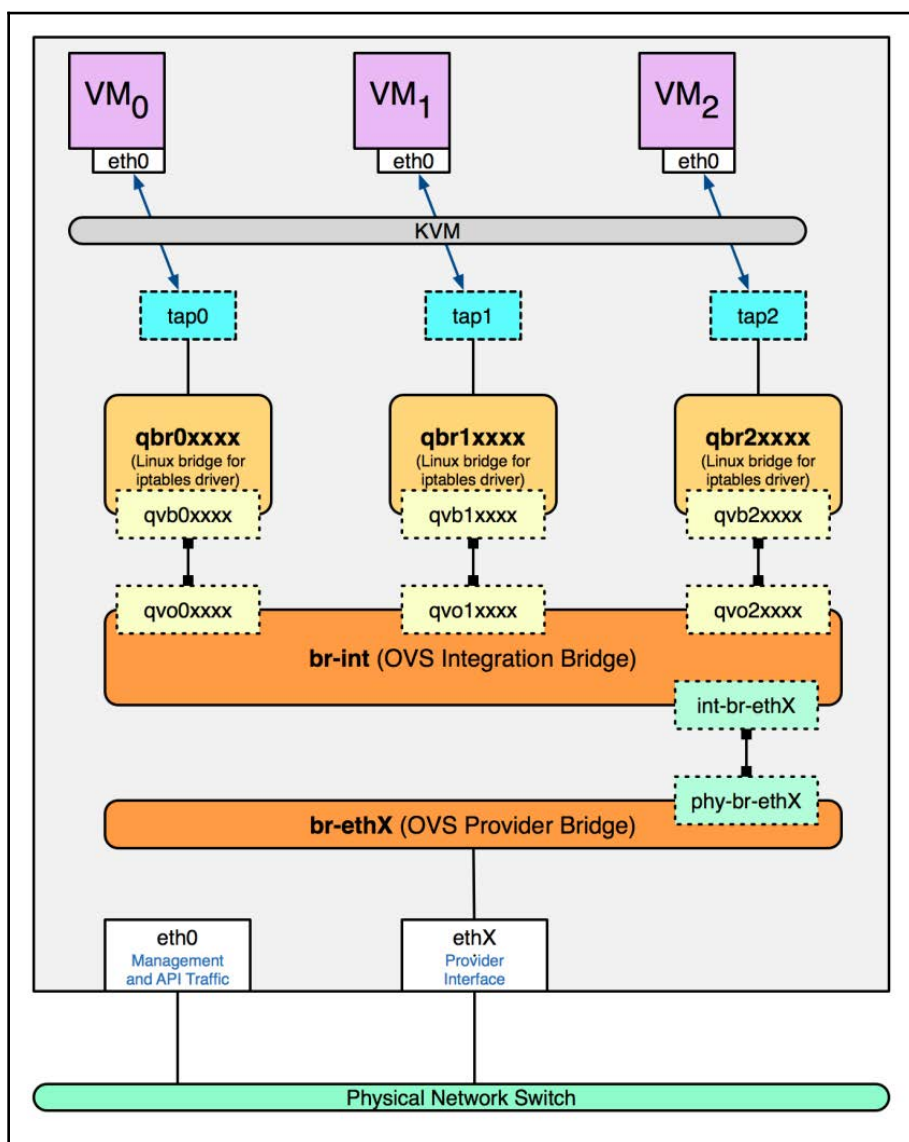


Figure 5.2

In the preceding diagram, instances are connected to an individual Linux bridge via their respective tap interface. The Linux bridges are connected to the OVS integration bridge using a **veth** interface. OpenFlow rules on the integration bridge dictate how traffic is forwarded through the virtual switch. The integration bridge is connected to the provider bridge using an OVS patch cable. Lastly, the provider bridge is connected to the physical network interface, which allows traffic to enter and exit the host onto the physical network infrastructure.

When using the Open vSwitch driver, each controller, network, or compute node in the environment has its own integration bridge and provider bridge. The virtual switches across nodes are effectively cross-connected to one another through the physical network. More than one provider bridge can be configured on a host, but often requires the use of a dedicated physical interface per provider bridge.

Identifying ports on the virtual switch

Using the `ovs-ofctl show br-int` command, we can see a logical representation of the integration bridge. The following screenshot demonstrates the use of this command to show the switch ports of the integration bridge on `compute02`:

```

root@compute02:~# ovs-ofctl show br-int
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000da053b9e154b
n_tables:254, n_buffers:0
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
1(int-br-eth2): addr:0a:21:8b:2b:7d:dd
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
2(patch-tun): addr:9e:71:de:c8:c9:cf
    config: 0
    state: 0
    speed: 0 Mbps now, 0 Mbps max
3(qvo3de035cc-79): addr:4a:8c:e8:f8:69:fd
    config: 0
    state: 0
    current: 10GB-FD COPPER
    speed: 10000 Mbps now, 0 Mbps max
4(qvoce30da31-3a): addr:ee:c8:17:d9:12:97
    config: 0
    state: 0
    current: 10GB-FD COPPER
    speed: 10000 Mbps now, 0 Mbps max
5(qvoa943af89-8e): addr:6e:55:54:3a:ad:8b
    config: 0
    state: 0
    current: 10GB-FD COPPER
    speed: 10000 Mbps now, 0 Mbps max
LOCAL(br-int): addr:da:05:3b:9e:15:4b
    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

```

The following are the components demonstrated in the preceding screenshot:

- Port number 1 is named `int-br-eth2` and is one end of an OVS patch cable. The other end connects to the provider bridge, `br-eth2` (not shown).
- Port number 2 is named `patch-tun` and is one end of an OVS patch cable. The other end connects to the tunnel bridge, `br-tun` (not pictured).
- Port number 3 is named `qvo3de035cc-79` and corresponds to Neutron port `3de035cc-79a9-4172-bb25-d4a7ea96325e`.
- Port number 4 is named `qvoce30da31-3aa` and corresponds to Neutron port `ce30da31-3a71-4c60-a350-ac0453b24d7d`.
- Port number 5 is named `qvoa943af89-8e` and corresponds to Neutron port `a943af89-8e21-4b1d-877f-abe946f6e565`.
- The LOCAL port named `br-int` is used internally by Open vSwitch and can be ignored.

The following screenshot demonstrates the switch configuration in a graphical manner:

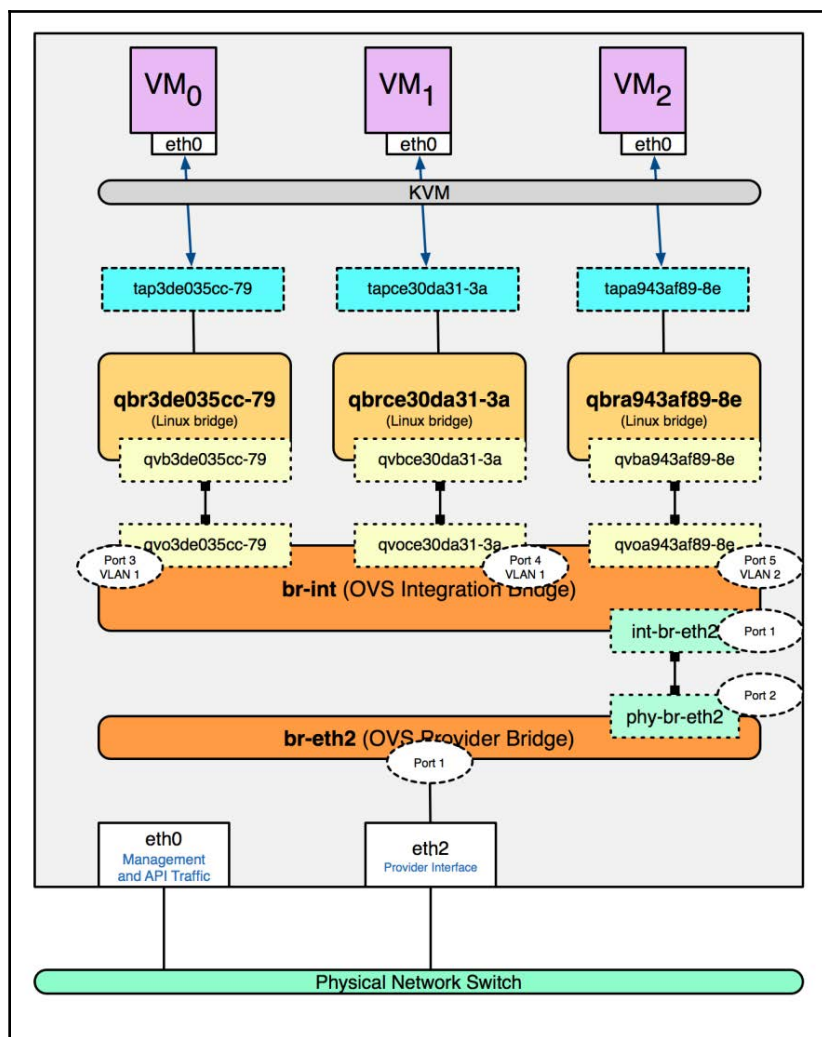


Figure 5.3

Identifying the local VLANs associated with ports

Every port on the integration bridge connected to an instance or other network resource is placed in a VLAN that is local to that virtual switch.

The Open vSwitch database on each host is independent of all other hosts, and the local VLAN database is not directly related to the physical network infrastructure. Instances in the same Neutron network on a particular host are placed in the same VLAN on the local integration bridge, but there is no VLAN ID consistency expected between hosts. That said, flow rules will be implemented on each host that maps the local VLAN ID to the ID associated with the respective Neutron network, allowing for traffic between hosts across the common VLAN. This behavior will be discussed in further detail later in this chapter.

Using the `ovs-vsctl show` command, you can identify the local VLAN tag of all ports on all virtual switches on the host. The following screenshot demonstrates this command in action on `compute02`:

```
root@compute02:~# ovs-vsctl show
5a68f2c6-318e-4d46-b549-4696f6f53cf5
  Manager "ptcp:6640:127.0.0.1"
    is_connected: true
  Bridge br-tun
    Controller "tcp:127.0.0.1:6633"
      is_connected: true
    fail_mode: secure
    Port patch-int
      Interface patch-int
        type: patch
        options: {peer=patch-tun}
    Port br-tun
      Interface br-tun
        type: internal
  Bridge "br-eth2"
    Controller "tcp:127.0.0.1:6633"
      is_connected: true
    fail_mode: secure
    Port "phy-br-eth2"
      Interface "phy-br-eth2"
        type: patch
        options: {peer="int-br-eth2"}
    Port "ens224"
      Interface "ens224"
    Port "br-eth2"
      Interface "br-eth2"
        type: internal
  Bridge br-int
    Controller "tcp:127.0.0.1:6633"
      is_connected: true
    fail_mode: secure
    Port patch-tun
      Interface patch-tun
        type: patch
        options: {peer=patch-int}
    Port br-int
      Interface br-int
        type: internal
    Port "qvoce30da31-3a"
      tag: 1
      Interface "qvoce30da31-3a"
    Port "qvoa943af89-8e"
      tag: 2
      Interface "qvoa943af89-8e"
    Port "int-br-eth2"
      Interface "int-br-eth2"
        type: patch
        options: {peer="phy-br-eth2"}
    Port "qvo3de035cc-79"
      tag: 1
      Interface "qvo3de035cc-79"
  ovs_version: "2.8.0"
```

Connected to the integration bridge are three interfaces named `qvoce30da31-3a`, `qvoa943af89-8e`, and `qvo3de035cc-79`. Two of the interfaces are in the same network and reside in the same local VLAN. The other interface, `qvoa943af89-8e`, is in a different network and thus is a different VLAN.



The local VLAN IDs are arbitrarily assigned by the local Open vSwitch process and may change upon restart of the `openvswitch-switch` service or after a reboot.

Programming flow rules

Unlike the Linux bridge architecture, the Open vSwitch driver does not use VLAN interfaces on the host to tag traffic. Instead, the Open vSwitch agent programs flow rules on the virtual switches that dictate how traffic traversing the switch should be manipulated before forwarding. When traffic traverses a virtual switch, flow rules on the switch can transform, add, or strip the VLAN tags before forwarding the traffic. In addition to this, flow rules can be added that drop traffic if it matches certain characteristics. Open vSwitch is capable of performing other types of actions on traffic, but those actions are outside the scope of this book.

Using the `ovs-ofctl dump-flows <bridge>` command, we can observe the flows that are currently programmed on the specified bridge. The Open vSwitch plugin agent is responsible for converting information about the network in the Neutron database to Open vSwitch flows, and constantly maintains the flows as changes are being made to the network.

Flow rules for VLAN networks

In the following example, VLANs 40 and 42 represent two networks in the data center. Both VLANs have been trunked down to the `controller` and `compute` nodes, and Neutron networks have been configured that utilize those VLAN IDs.

On the physical switch, the necessary configuration to facilitate the networking described here will resemble the following:

```
vlan 40
    name VLAN_40
vlan 42
    name VLAN_42
```

```
interface Ethernet1/4
  description Provider_Interface_eth2
  switchport
  switchport mode trunk
  switchport trunk allowed vlan add 40,42
  no shutdown
```

When configured as a trunk port, the provider interface can support multiple VLAN networks. Traffic that enters physical interface `eth2` is processed by the flow rules on the `br-eth2` bridge it is connected to. Flow rules are processed in order of priority from highest to lowest. By default, `ovs-ofctl` returns flow entries in the same order that the virtual switch sends them. Using `--rsort`, it is possible to return the results in order of priority, from highest to lowest, to match the order in which packets are processed:

```
root@compute02:~# ovs-ofctl dump-flows br-eth2 --rsort
table=0, n_packets=0, n_bytes=0, priority=4,in_port="phy-br-eth2",dl_vlan=1 actions=mod_vlan_vid:42,NORMAL
table=0, n_packets=0, n_bytes=0, priority=4,in_port="phy-br-eth2",dl_vlan=2 actions=mod_vlan_vid:40,NORMAL
table=0, n_packets=12, n_bytes=1016, priority=2,in_port="phy-br-eth2" actions=drop
table=0, n_packets=256769, n_bytes=41302038, priority=0 actions=NORMAL
```



For readability, both the `duration` and `cookie` fields have been removed.

The first three rules specify a particular inbound port:

```
in_port="phy-br-eth2"
```

According to the diagram in Figure 5.3, traffic entering the bridge `br-eth2` from physical interface `eth2` does so through port 1, not the port named `phy-br-eth2`, so the first three rules do not apply. As a result, traffic is forwarded to the integration bridge via the fourth rule, where no particular port is specified:

```
root@compute02:~# ovs-ofctl dump-flows br-eth2 --rsort
table=0, n_packets=0, n_bytes=0, priority=4,in_port="phy-br-eth2",dl_vlan=1 actions=mod_vlan_vid:42,NORMAL
table=0, n_packets=0, n_bytes=0, priority=4,in_port="phy-br-eth2",dl_vlan=2 actions=mod_vlan_vid:40,NORMAL
table=0, n_packets=12, n_bytes=1016, priority=2,in_port="phy-br-eth2" actions=drop
table=0, n_packets=256769, n_bytes=41302038, priority=0 actions=NORMAL
```

Flows with an action of `NORMAL` instructs Open vSwitch to act as a learning switch, which means traffic will be forwarded out of all of the ports, other than the one where traffic was received, until the switch learns and updates its forwarding database. Traffic is forwarded out of the port that's connected to the integration bridge.



The forwarding database, or FDB table, is the equivalent of a CAM or MAC address table on a physical switch. This learning behavior is similar to that of a hardware switch that floods traffic out of all ports until it learns the proper path.

As traffic exits the provider bridge `br-eth2` and enters port 1 of the integration bridge `br-int`, it is evaluated by the flow rules on `br-int`, as shown here:

```
root@compute02:~# ovs-ofctl dump-flows br-int --rsort
table=0, n_packets=0, n_bytes=0, priority=10,icmp6,in_port="qvo3de035cc-79",icmp_type=136 actions=resubmit(,24)
table=0, n_packets=0, n_bytes=0, priority=10,icmp6,in_port="qvoce30da31-3a",icmp_type=136 actions=resubmit(,24)
table=0, n_packets=0, n_bytes=0, priority=10,icmp6,in_port="qvoa943af89-8e",icmp_type=136 actions=resubmit(,24)
table=0, n_packets=0, n_bytes=0, priority=10,arp,in_port="qvo3de035cc-79" actions=resubmit(,24)
table=0, n_packets=0, n_bytes=0, priority=10,arp,in_port="qvoce30da31-3a" actions=resubmit(,24)
table=0, n_packets=0, n_bytes=0, priority=10,arp,in_port="qvoa943af89-8e" actions=resubmit(,24)
table=0, n_packets=8, n_bytes=560, priority=9,in_port="qvo3de035cc-79" actions=resubmit(,25)
table=0, n_packets=9, n_bytes=630, priority=9,in_port="qvoce30da31-3a" actions=resubmit(,25)
table=0, n_packets=8, n_bytes=560, priority=9,in_port="qvoa943af89-8e" actions=resubmit(,25)
table=0, n_packets=0, n_bytes=0, priority=3,in_port="int-br-eth2",dl_vlan=42 actions=mod_vlan_vid:1,resubmit(,60)
table=0, n_packets=0, n_bytes=0, priority=3,in_port="int-br-eth2",dl_vlan=40 actions=mod_vlan_vid:2,resubmit(,60)
table=60, n_packets=12, n_bytes=1016, priority=3 actions=NORMAL
table=0, n_packets=12650, n_bytes=1136948, priority=2,in_port="int-br-eth2" actions=drop
table=24, n_packets=0, n_bytes=0, priority=2,icmp6,in_port="qvo3de035cc-79",icmp_type=136,nd_target=fe80::f816:3eff:fe77:60c5 actions=resubmit(,60)
table=24, n_packets=0, n_bytes=0, priority=2,icmp6,in_port="qvoce30da31-3a",icmp_type=136,nd_target=fe80::f816:3eff:fed8:230b actions=resubmit(,60)
table=24, n_packets=0, n_bytes=0, priority=2,icmp6,in_port="qvoa943af89-8e",icmp_type=136,nd_target=fe80::f816:3eff:fede:9edc actions=resubmit(,60)
table=24, n_packets=0, n_bytes=0, priority=2,arp,in_port="qvo3de035cc-79",arp_spa=192.168.0.6 actions=resubmit(,25)
table=24, n_packets=0, n_bytes=0, priority=2,arp,in_port="qvoce30da31-3a",arp_spa=192.168.0.10 actions=resubmit(,25)
table=24, n_packets=0, n_bytes=0, priority=2,arp,in_port="qvoa943af89-8e",arp_spa=192.168.1.3 actions=resubmit(,25)
table=25, n_packets=0, n_bytes=0, priority=2,arp,in_port="qvo3de035cc-79",dl_src=fa:16:3e:77:60:c5 actions=resubmit(,60)
table=25, n_packets=0, n_bytes=0, priority=2,in_port="qvoce30da31-3a",dl_src=fa:16:3e:d8:23:0b actions=resubmit(,60)
table=25, n_packets=0, n_bytes=0, priority=2,in_port="qvoa943af89-8e",dl_src=fa:16:3e:d8:23:0b actions=resubmit(,60)
table=0, n_packets=12, n_bytes=1016, priority=0 actions=resubmit(,60)
table=23, n_packets=0, n_bytes=0, priority=0 actions=drop
table=24, n_packets=0, n_bytes=0, priority=0 actions=drop
```

Of immediate importance are the flow rules inspecting traffic sourced from the `int-br-eth2` interface, as that is where traffic enters the integration bridge from the provider bridge. The first rule shown here performs the action of modifying the VLAN ID of a packet from its original VLAN to a VLAN that is local to the integration bridge on the `compute` node when the original VLAN ID, as identified by the `dl_vlan` value, is 42:

```
root@compute02:~# ovs-ofctl dump-flows br-int --rsort
...
table=0, n_packets=0, n_bytes=0, priority=3,in_port="int-br-eth2",dl_vlan=42 actions=mod_vlan_vid:1,resubmit(,60)
table=0, n_packets=0, n_bytes=0, priority=3,in_port="int-br-eth2",dl_vlan=40 actions=mod_vlan_vid:2,resubmit(,60)
table=60, n_packets=12, n_bytes=1016, priority=3 actions=NORMAL
table=0, n_packets=12650, n_bytes=1136948, priority=2,in_port="int-br-eth2" actions=drop
...
table=0, n_packets=12, n_bytes=1016, priority=0 actions=resubmit(,60)
...
```

When traffic tagged as VLAN 42 on the physical network is sent to an instance and forwarded through the provider bridge to the integration bridge, the VLAN tag is modified from 42 to local VLAN 1. The frame is then forwarded to Table 60 for additional processing, where the default action is NORMAL. As a result, the frame is forwarded to a port on `br-int`, which is connected to the instance that matches the destination MAC address.

The next rule performs a similar action when the data link VLAN is 40 by replacing it with local VLAN 2. If traffic matches the `drop` rule, it means that no other rules of a higher priority entering `int-br-eth2` and traffic will be dropped:

```
root@compute02:~# ovs-ofctl dump-flows br-int --rsort
...
table=0, n_packets=0, n_bytes=0, priority=3,in_port="int-br-eth2",dl_vlan=42 actions=mod_vlan_vid:1,resubmit(,60)
table=0, n_packets=0, n_bytes=0, priority=3,in_port="int-br-eth2",dl_vlan=40 actions=mod_vlan_vid:2,resubmit(,60)
table=60, n_packets=12, n_bytes=1016, priority=3 actions=NORMAL
table=0, n_packets=12650, n_bytes=1136948, priority=2,in_port="int-br-eth2" actions=drop
...
table=0, n_packets=12, n_bytes=1016, priority=0 actions=resubmit(,60)
...
```

Return traffic

Return traffic from the instances through the integration bridge `br-int` may be processed by various flow rules that are used to inhibit ARP and MAC spoofing from instances. If the traffic is allowed, it is forwarded to Table 60 for additional processing and out to the provider bridge:

```
root@compute02:~# ovs-ofctl dump-flows br-int --rsort
table=0, n_packets=0, n_bytes=0, priority=10,icmp6,in_port="qvo3de035cc-79",icmp_type=136 actions=resubmit(,24)
...
table=0, n_packets=0, n_bytes=0, priority=10,arp,in_port="qvo3de035cc-79" actions=resubmit(,24)
...
table=0, n_packets=8, n_bytes=560, priority=9,in_port="qvo3de035cc-79" actions=resubmit(,25)
...
table=60, n_packets=12, n_bytes=1016, priority=3 actions=NORMAL
...
table=24, n_packets=0, n_bytes=0, priority=2,icmp6,in_port="qvo3de035cc-79",icmp_type=136,
    nd_target=fe80::f816:3eff:fe77:60c5 actions=resubmit(,60)
...
table=24, n_packets=0, n_bytes=0, priority=2,arp,in_port="qvo3de035cc-79",arp_spa=192.168.0.6 actions=resubmit(,25)
...
table=25, n_packets=0, n_bytes=0, priority=2,in_port="qvo3de035cc-79",dl_src=fa:16:3e:77:60:c5 actions=resubmit(,60)
...
table=0, n_packets=12, n_bytes=1016, priority=0 actions=resubmit(,60)
table=23, n_packets=0, n_bytes=0, priority=0 actions=drop
table=24, n_packets=0, n_bytes=0, priority=0 actions=drop
```

Once traffic hits the provider bridge `br-eth2`, it is processed by the flow rules as follows:

```
root@compute02:~# ovs-ofctl dump-flows br-eth2 --rsort
table=0, n_packets=0, n_bytes=0, priority=4,in_port="phy-br-eth2",dl_vlan=1 actions=mod_vlan_vid:42,NORMAL
table=0, n_packets=0, n_bytes=0, priority=4,in_port="phy-br-eth2",dl_vlan=2 actions=mod_vlan_vid:40,NORMAL
table=0, n_packets=12, n_bytes=1016, priority=2,in_port="phy-br-eth2" actions=drop
table=0, n_packets=2704987, n_bytes=673083387, priority=0 actions=NORMAL
```

If these rules look familiar, it's because they are the same flow rules on the provider bridge that we showed you earlier. This time, however, traffic from the integration bridge connected to port `phy-br-eth2` is processed by these rules.

The first flow rule on the provider bridge checks the VLAN ID in the Ethernet header, and if it is 1, modifies it to 42 before forwarding the traffic to the physical interface. The second rule modifies the VLAN tag of the frame from 2 to 40 before it exits the bridge. All other traffic from the integration bridge not tagged as VLAN 1 or 2 is dropped.



Flow rules for a particular network will not exist on a bridge if there are no instances or resources in that network scheduled to that node. The Neutron Open vSwitch agent on each node is responsible for creating the appropriate flow rules for virtual switches on the respective node.

Flow rules for flat networks

Flat networks in Neutron are untagged networks, meaning there is no 802.1q VLAN tag associated with the network when it is created. Internally, however, Open vSwitch treats flat networks similarly to VLAN networks when programming the virtual switches. Flat networks are assigned a local VLAN ID in the Open vSwitch database just like a VLAN network, and instances in the same flat network connected to the same integration bridge are placed in the same local VLAN. However, there is a difference between VLAN and flat networks that can be observed in the flow rules that are created on the integration and provider bridges. Instead of mapping the local VLAN ID to a physical VLAN ID, and vice versa, as traffic traverses the bridges, the local VLAN ID is added to or stripped from the Ethernet header by flow rules.

On the physical switch, the necessary configuration to facilitate the networking described here will resemble the following:

```
vlan 200
  name VLAN_200

interface Ethernet1/4
  description Provider_Interface_eth2
  switchport
  switchport mode trunk
  switchport trunk native vlan 200
  switchport trunk allowed vlan add 200
  no shutdown
```

Alternatively, the interface can also be configured as an access port:

```
interface Ethernet1/4
  description Provider_Interface_eth2
  switchport
  switchport mode access
  switchport access vlan 200
  no shutdown
```

Only one flat network is supported per provider interface. When configured as a trunk port with a native VLAN, the provider interface can support a single flat network and multiple VLAN networks. When configured as an access port, the interface can only support a single flat network and any attempt to tag traffic will fail.

In this example, a flat network has been added in Neutron that has no VLAN tag:

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-02-27T21:53:13Z
description	
dns_domain	None
id	97aa85e6-1d88-414e-bed2-bb610fd5c03c
ipv4_address_scope	None
ipv6_address_scope	None
is_default	None
is_vlan_transparent	None
mtu	1500
name	MyFlatNetwork
port_security_enabled	False
project_id	9233b6b4f6a54386af63c0a7b8f043c2
provider:network_type	flat
provider:physical_network	physnet1
provider:segmentation_id	None
qos_policy_id	None
revision_number	1
router:external	Internal
segments	None
shared	False
status	ACTIVE
subnets	
tags	
updated_at	2018-02-27T21:53:13Z

On the physical switch, this network will correspond to the native (untagged) VLAN on the switch port connected to eth2 of compute02. In this case, the native VLAN is 200. An instance has been spun up on the network MyFlatNetwork, which results in the following virtual switch configuration:

```
Bridge br-int
  Controller "tcp:127.0.0.1:6633"
    is_connected: true
  fail_mode: secure
  Port patch-tun
    Interface patch-tun
      type: patch
      options: {peer=patch-int}
  Port br-int
    Interface br-int
      type: internal
  Port "qvo4655b19f-85"
    tag: 3
    Interface "qvo4655b19f-85"
  Port "qvoce30da31-3a"
    tag: 1
    Interface "qvoce30da31-3a"
  Port "qvoa943af89-8e"
    tag: 2
    Interface "qvoa943af89-8e"
  Port "int-br-eth2"
    Interface "int-br-eth2"
      type: patch
      options: {peer="phy-br-eth2"}
  Port "qvo3de035cc-79"
    tag: 1
    Interface "qvo3de035cc-79"
  ovs_version: "2.8.0"
```

Notice that the port associated with the instance has been assigned a local VLAN ID of 3, as identified by the `tag` value, even though it is a flat network. On the integration bridge, there now exists a flow rule that modifies the VLAN header of an incoming Ethernet frame when it has no VLAN ID set:

```
root@compute02:~# ovs-ofctl dump-flows br-int --rsort
...
table=0, n_packets=0, n_bytes=0, priority=3, in_port="int-br-eth2", dl_vlan=42 actions=mod_vlan_vid:1, resubmit(,60)
table=0, n_packets=0, n_bytes=0, priority=3, in_port="int-br-eth2", dl_vlan=40 actions=mod_vlan_vid:2, resubmit(,60)
table=0, n_packets=26, n_bytes=1580, priority=3, in_port="int-br-eth2", vlan_tci=0x0000/0x1fff actions=mod_vlan_vid:3, resubmit(,60)
table=60, n_packets=43, n_bytes=3022, priority=3 actions=NORMAL
...
```



TCI stands for **Tag Control Information**, and is a 2-byte field of the 802.1q header. For packets with an 802.1q header, this field contains VLAN information including the VLAN ID. For packets without an 802.1q header, also known as untagged, the `vlan_tci` value is set to zero (0x0000).

The result is that incoming traffic on the flat network is tagged as VLAN 3 and forwarded to instances connected to the integration bridge that reside in VLAN 3.

As return traffic from the instance is processed by flow rules on the provider bridge, the local VLAN ID is stripped and the traffic becomes untagged:

```
root@compute02:~# ovs-ofctl dump-flows br-eth2 --rsort
table=0, n_packets=0, n_bytes=0, priority=4,in_port="phy-br-eth2",dl_vlan=1 actions=mod_vlan_vid:42,NORMAL
table=0, n_packets=0, n_bytes=0, priority=4,in_port="phy-br-eth2",dl_vlan=2 actions=mod_vlan_vid:40,NORMAL
table=0, n_packets=0, n_bytes=0, priority=4,in_port="phy-br-eth2",dl_vlan=3 actions=strip_vlan,NORMAL
table=0, n_packets=17, n_bytes=1442, priority=2,in_port="phy-br-eth2" actions=drop
table=0, n_packets=3104979, n_bytes=729425517, priority=0 actions=NORMAL
```

The untagged traffic is then forwarded out to the physical interface `eth2` and processed by the physical switch.

Flow rules for overlay networks

Overlay networks in a reference implementation of Neutron are ones that use either VXLAN or GRE to encapsulate virtual instance traffic between hosts. Instances connected to an overlay network are attached to the integration bridge and use a local VLAN mapped to that network, just like the other network types we have discussed so far. All instances on the same host are connected to the same local VLAN.

In this example, an overlay network has been created with Neutron auto-assigning a segmentation ID of 39.

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-03-02T13:20:57Z
description	
dns_domain	None
id	06952b48-1cf1-48b9-be60-78ad4e97fec4
ipv4_address_scope	None
ipv6_address_scope	None
is_default	None
is_vlan_transparent	None
mtu	1450
name	MyOverlayNetwork
port_security_enabled	False
project_id	9233b6b4f6a54386af63c0a7b8f043c2
provider:network_type	vxlان
provider:physical_network	None
provider:segmentation_id	39
qos_policy_id	None
revision_number	1
router:external	Internal
segments	None
shared	False
status	ACTIVE
subnets	
tags	
updated_at	2018-03-02T13:20:57Z

No changes are needed on the physical switching infrastructure to support this network, as the traffic will be encapsulated and forwarded through the overlay network interface, `eth1`.

An instance has been spun up on the network `MyOverlayNetwork`, which results in the following virtual switch configuration:

```
Bridge br-int
  Controller "tcp:127.0.0.1:6633"
    is_connected: true
  fail_mode: secure
  Port patch-tun
    Interface patch-tun
      type: patch
      options: {peer=patch-int}
  Port "qvo2bc7251b-07"
    tag: 4
    Interface "qvo2bc7251b-07"
  Port br-int
    Interface br-int
      type: internal
  Port "qvo4655b19f-85"
    tag: 3
    Interface "qvo4655b19f-85"
  Port "qvoce30da31-3a"
    tag: 1
    Interface "qvoce30da31-3a"
  Port "qvoa943af89-8e"
    tag: 2
    Interface "qvoa943af89-8e"
  Port "int-br-eth2"
    Interface "int-br-eth2"
      type: patch
      options: {peer="phy-br-eth2"}
  Port "qvo3de035cc-79"
    tag: 1
    Interface "qvo3de035cc-79"
```


Notice that the port associated with the instance has been assigned a local VLAN ID of 4, even though it is an overlay network. When an instance sends traffic to another instance or device in the same network, the integration bridge forwards the traffic out toward the tunnel bridge, `br-tun`, where the following flow rules are consulted:

```
root@compute02:/home/jdenton# ovs-ofctl dump-flows br-tun --rstart
table=20, n_packets=0, n_bytes=0, priority=2,d_l_vlan=4,d_l_dst=fa:16:3e:f1:b0:49 actions=strip_vlan,
    load:0x27->NXM_NX_TUN_ID[],output:"vxlan-0a140064"
table=0, n_packets=24415, n_bytes=1476920, priority=1,in_port="patch-int" actions=resubmit(,2)
table=0, n_packets=0, n_bytes=0, priority=1,in_port="vxlan-0a140064" actions=resubmit(,4)
table=2, n_packets=23247, n_bytes=1394820, priority=1,arp,d_l_dst=ff:ff:ff:ff:ff:ff actions=resubmit(,21)
table=4, n_packets=0, n_bytes=0, priority=1,tun_id=0x27 actions=mod_vlan_vid:4,resubmit(,10)
table=10, n_packets=0, n_bytes=0, priority=1
    actions=learn(table=20,hard_timeout=300,priority=1,cookie=0x4959e8e689bdd36,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:OXM_OF_IN_PORT[]),output:"patch-int"
table=21, n_packets=0, n_bytes=0, priority=1,arp,d_l_vlan=4,arp_tpa=172.18.90.2
    actions=load:0x2->NXM_OF_ARP_OP[],move:NXM_NX_ARP_SHA[]->NXM_NX_ARP_THA[],move:NXM_OF_ARP_SPA[]->NXM_OF_ARP_TPA[],
    load:0xfa163ef1b049->NXM_NX_ARP_SHA[],load:0xac125a02->NXM_OF_ARP_SPA[],move:NXM_OF_ETH_SRC[]->NXM_OF_ETH_DST[],
    mod_d_l_src:fa:16:3e:f1:b0:49,IN_PORT
table=22, n_packets=0, n_bytes=0, priority=1,d_l_vlan=4 actions=strip_vlan,load:0x27->NXM_NX_TUN_ID[],output:"vxlan-0a140064"
table=0, n_packets=0, n_bytes=0, priority=0 actions=drop
table=2, n_packets=0, n_bytes=0, priority=0,d_l_dst=00:00:00:00:00:00/01:00:00:00:00:00 actions=resubmit(,20)
table=2, n_packets=1168, n_bytes=82100, priority=0,d_l_dst=01:00:00:00:00:00/01:00:00:00:00:00 actions=resubmit(,22)
table=3, n_packets=0, n_bytes=0, priority=0 actions=drop
table=4, n_packets=0, n_bytes=0, priority=0 actions=drop
table=6, n_packets=0, n_bytes=0, priority=0 actions=drop
table=20, n_packets=0, n_bytes=0, priority=0 actions=resubmit(,22)
table=21, n_packets=23247, n_bytes=1394820, priority=0 actions=resubmit(,22)
table=22, n_packets=24415, n_bytes=1476920, priority=0 actions=drop
```

The flows rules implemented on the tunnel bridge are unique, in that they specify a **virtual tunnel endpoint**, or VTEP, for every destination MAC address, including other instances and routers that are connected to the network. This behavior ensures that traffic is forwarded directly to the `compute` or `network` node where the destination resides and is not forwarded out on all ports of the bridge. Traffic that does not match is dropped.

In this example, traffic to destination MAC address `fa:16:3e:f1:b0:49` is forwarded out to port `vxlان0a140064`, which, as we can see here, is mapped to a tunnel endpoint:

```
Bridge br-tun
  Controller "tcp:127.0.0.1:6633"
    is_connected: true
    fail_mode: secure
  Port patch-int
    Interface patch-int
      type: patch
      options: {peer=patch-tun}
  Port br-tun
    Interface br-tun
      type: internal
  Port "vxlan-0a140064"
    Interface "vxlan-0a140064"
      type: vxlan
      options: {df_default="true", in_key=flow, local_ip="10.20.0.102", out_key=flow, remote_ip="10.20.0.100"}
```

The address `10.20.0.100` is the VXLAN tunnel endpoint for `controller01`, and the MAC address `fa:16:3e:f1:b0:49` belongs to the DHCP server in the `MyOverlayNetwork` network.

Return traffic to the instance is first processed by flow rules on the tunnel bridge and then forwarded to the integration bridge, where it is then forwarded to the instance.

Flow rules for local networks

Local networks in an Open vSwitch implementation behave similar to that of a Linux bridge implementation. Instances in local networks are connected to the integration bridge and can communicate with other instances in the same network and local VLAN. There are no flow rules created for local networks, however.

Traffic between instances in the same network remains local to the virtual switch, and by definition, local to the `compute` node on which they reside. This means that connectivity to services hosted on other nodes, such as DHCP and metadata, will be unavailable to any instance not on the same host as those services.

Configuring the ML2 networking plugin

The remainder of this chapter is dedicated to providing instructions on installing and configuring the Neutron Open vSwitch agent and the ML2 plugin for use with the Open vSwitch mechanism driver. In this book, `compute02`, `compute03`, and `snat01` will be the only nodes configured for use with Open vSwitch.

Configuring the bridge interface

In this installation, physical network interface `eth2` will be utilized as the **provider interface** for bridging purposes.

On `compute02`, `compute03`, and `snat01`, configure the `eth2` interface within the `/etc/network/interfaces` file as follows:

```
auto eth2
iface eth2 inet manual
```

Close and save the file, and bring the interface up with the following command:

```
# ip link set dev eth2 up
```



Because the interface will be used in a bridge, an IP address cannot be applied directly to the interface. If there is an IP address applied to `eth2`, it will become inaccessible once the interface is placed in a bridge. The bridge will be created later on in this chapter.

Configuring the overlay interface

In this installation, physical network interface `eth1` will be utilized as the **overlay interface** for overlay networks using VXLAN. For VXLAN networking, this is the equivalent of the VXLAN tunnel endpoint, or VTEP. Neutron will be responsible for configuring some aspects of Open vSwitch once the initial network configuration has been completed.

On all hosts, configure the `eth1` interface within the `/etc/network/interfaces` file, if it has not already been done:

```
auto eth1
iface eth1 inet static
    address 10.20.0.X/24
```

Use the following table for the appropriate address. Substitute the address with `x` where appropriate:

Host	Address
compute02	10.20.0.102
compute03	10.20.0.103
snat01	10.20.0.104

Close and save the file, and bring the interface up with the following command:

```
# ip link set dev eth1 up
```

Confirm that the interface is in an UP state and that the address has been set using the `ip addr show dev eth1` command. Ensure the `compute02` can communicate over the newly configured interface by pinging `controller01`:

```
root@compute02:~# ping 10.20.0.100 -c5
PING 10.20.0.100 (10.20.0.100) 56(84) bytes of data.
64 bytes from 10.20.0.100: icmp_seq=1 ttl=64 time=0.314 ms
64 bytes from 10.20.0.100: icmp_seq=2 ttl=64 time=0.473 ms
64 bytes from 10.20.0.100: icmp_seq=3 ttl=64 time=0.424 ms
64 bytes from 10.20.0.100: icmp_seq=4 ttl=64 time=0.323 ms
64 bytes from 10.20.0.100: icmp_seq=5 ttl=64 time=0.434 ms

--- 10.20.0.100 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4091ms
rtt min/avg/max/mdev = 0.314/0.393/0.473/0.067 ms
```

Repeat this process for all of the nodes.



If you experience any issues communicating across this interface, you *will* experience issues with VXLAN networks that have been created with OpenStack Networking. Any issues should be corrected before continuing.

ML2 plugin configuration options

The ML2 plugin was initially installed in Chapter 3, *Installing Neutron*, and was configured to support the Linux bridge mechanism driver in the previous chapter. It must be modified to support the Open vSwitch mechanism driver.

Mechanism drivers

Mechanism drivers are responsible for implementing networks described by the type driver. Mechanism drivers shipped with the ML2 plugin include `linuxbridge`, `openvswitch`, and `l2population`.

Update the ML2 configuration file on `controller01` and append `openvswitch` to the list of mechanism drivers:

```
[ml2]
...
mechanism_drivers = linuxbridge,l2population,openvswitch
```



The Neutron Open vSwitch agent requires specific configuration options, which will be discussed later on in this chapter.

Flat networks

The `flat_networks` configuration option defines interfaces that support the use of untagged networks, commonly referred to as native or access VLANs. This option requires that a provider label is specified. A **provider label** is an arbitrary label or name that is mapped to a physical interface or bridge on the host. These mappings will be discussed in further detail later on in this chapter.

In the following example, the `physnet1` interface has been configured to support a flat network:

```
flat_networks = physnet1
```

Multiple interfaces can be defined using a comma-separated list:

```
flat_networks = physnet1,physnet2
```



Due to the lack of an identifier to segregate untagged traffic on the same interface, an interface can only support a single flat network.

In this environment, the `flat_networks` option can remain *unconfigured*.

Network VLAN ranges

The `network_vlan_ranges` configuration option defines a range of VLANs that project networks will be associated with upon their creation when `tenant_network_types` is `vlan`. When the number of available VLANs reaches zero, tenants will no longer be able to create VLAN networks.

In the following example, VLAN IDs 40 through 43 are available for tenant network allocation:

```
network_vlan_ranges = physnet1:40:43
```

Non-contiguous VLANs can be allocated by using a comma-separated list:

```
network_vlan_ranges = physnet1:40:43,physnet1:51:55
```

In this installation, the provider label `physnet1` will be used with VLANs 40 through 43. Those VLANs will be automatically assigned to `vlan` networks upon creation, unless overridden by a user with the `admin` role.

Update the ML2 configuration file on the `controller` node and add the following `network_vlan_ranges` to the `[ml2_type_vlan]` section if it doesn't already exist:

```
[ml2_type_vlan]
...
network_vlan_ranges = physnet1:40:43
```

Tunnel ID ranges

When GRE networks are created, each network is assigned a unique segmentation ID that is used to encapsulate traffic. As traffic traverses the Open vSwitch tunnel bridge, the segmentation ID is used to populate a field in the encapsulation header of the packet. For GRE packets, the `KEY` header field is used.

The `tunnel_id_ranges` configuration option found under `[ml2_type_gre]` is a comma-separated list of ID ranges that are available for tenant network allocation when `tunnel_type` is set to `gre`.

In the following example, segmentation IDs 1 through 1,000 are reserved for allocation to tenant networks upon creation:

```
tunnel_id_ranges = 1:1000
```

The `tunnel_id_ranges` option supports non-contiguous IDs using a comma-separated list as follows:

```
tunnel_id_ranges = 1:1000,2000:2500
```

GRE networks will not be configured as part of the exercises in this book, so `tunnel_id_ranges` can remain *unconfigured*.

VNI Ranges

When VXLAN networks are created, each network is assigned a unique segmentation ID that is used to encapsulate traffic.

The `vni_ranges` configuration option is a comma-separated list of ID ranges that are available for project network allocation when `tunnel_type` is set to `vxlan`.

In the following example, segmentation IDs 1 through 1,000 are reserved for allocation to tenant networks upon creation:

```
vni_ranges = 1:1000
```

The `vni_ranges` option supports non-contiguous IDs using a comma-separated list as follows:

```
vni_ranges = 1:1000,2000:2500
```

Update the ML2 configuration file on the `controller` node and add the following `vni_ranges` to the `[ml2_type_vxlan]` section if it doesn't already exist:

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



The 24-bit VNI field in the VXLAN header supports up to approximately 16 million unique identifiers.

Security groups

The `enable_security_group` configuration option instructs Neutron to enable or disable security group-related API functions. This option is set to `true` by default.

The `enable_ipset` configuration option instructs Neutron to enable or disable the `ipset` extension for `iptables` when the `iptables_hybrid` firewall driver is used. The use of `ipsets` allows for the creation of firewall rules that match entire sets of addresses at once rather than having individual lines per address, making lookups very efficient compared to traditional linear lookups. This option is set to `true` by default.



If at any time the ML2 configuration file is updated, you must restart the `neutron-server` service and respective Neutron agent for the changes to take effect.

Configuring the Open vSwitch driver and agent

The Open vSwitch mechanism driver is included with the ML2 plugin, and was installed in Chapter 3, *Installing Neutron*. The following sections will walk you through the configuration of OpenStack Networking so that you can utilize the Open vSwitch driver and agent.



While the Linux bridge and Open vSwitch agents and drivers can coexist in the same environment, they should not be installed and configured simultaneously on the same host.

Installing the Open vSwitch agent

To install the Open vSwitch agent, issue the following command on `compute02`, `compute03`, and `snat01`:

```
# apt install neutron-plugin-openvswitch-agent
```

Dependencies, such as Open vSwitch components `openvswitch-common` and `openvswitch-switch`, will be installed. If prompted to overwrite existing configuration files, type `N` at the `[default=N]` prompt.

Updating the Open vSwitch agent configuration file

The Open vSwitch agent uses a configuration file located at `/etc/neutron/plugins/ml2/openvswitch_agent.ini`. The most common options can be seen as follows:

```
[agent]
...
tunnel_types = ...
l2_population = ...
arp_responder = ...
enable_distributed_routing = ...

[ovs]
...
integration_bridge = ...
tunnel_bridge = ...
local_ip = ...
bridge_mappings = ...

[securitygroup]
...
firewall_driver = ...
```

Tunnel types

The `tunnel_types` configuration option specifies the types of tunnels supported by the agent. The two available options are `gre` and/or `vxlan`. The default value is `None`, which disables tunneling.

Update the `tunnel_types` configuration option in the `[agent]` section of the Open vSwitch agent configuration file accordingly on `compute02`, `compute03`, and `snat01`:

```
[agent]
...
tunnel_types = vxlan
```

L2 population

To enable support for the L2 population driver, the `l2_population` configuration option must be set to `true`. Update the `l2_population` configuration option in the `[vxlan]` section of the Open vSwitch agent configuration file accordingly on `compute02`, `compute03`, and `snat01`:

```
[agent]
...
l2_population = true
```

An important feature of the L2 population driver is its ARP responder functionality, which avoids the broadcasting of ARP requests across the overlay network. Each `compute` node can proxy ARP requests from virtual machines and provide them with replies, all without traffic leaving the host.

To enable the ARP responder, update the following configuration option:

```
[agent]
...
arp_responder = true
```

The default `arp_responder` configuration is `false` and can remain *unchanged* for this environment.

VXLAN UDP port

The default port for UDP traffic between VXLAN tunnel endpoints varies depending on the system. The Internet Assigned Numbers Authority, or IANA, has assigned UDP port 4789 for the purposes of VXLAN and that is the default port used by Open vSwitch. The Linux kernel, on the other hand, uses UDP port 8472 for VXLAN. To maintain compatibility with the hosts using the Linux bridge mechanism driver and `vxlan` kernel module, the port must be changed from its default.

To change the port number, update the following configuration option from 4789 to 8472:

```
[agent]  
...  
vxlan_udp_port = 8472
```

This change is typically unnecessary in a pure Open vSwitch-based environment, but is required for the environment described in this book.

Integration bridge

The `integration_bridge` configuration option specifies the name of the integration bridge used on each node. There is a single integration bridge per node that acts as the virtual switch where all virtual machine VIFs, otherwise known as **virtual network interfaces**, are connected. The default name of the integration bridge is `br-int` and should not be modified.



Since the Icehouse release of OpenStack, the Open vSwitch agent automatically creates the integration bridge the first time the agent service is started. You do not need to add an interface to the integration bridge, as Neutron is responsible for connecting network devices to this virtual switch.

Tunnel bridge

The tunnel bridge is a virtual switch, similar to the integration and provider bridges, and is used to connect GRE and VXLAN tunnel endpoints. Flow rules exist on this bridge that are responsible for properly encapsulating and decapsulating tenant traffic as it traverses the bridge.

The `tunnel_bridge` configuration option specifies the name of the tunnel bridge. The default value is `br-tun` and should not be modified. It is not necessary to create this bridge manually since Neutron does this automatically.

Local IP

The `local_ip` configuration option specifies the local IP address on the node, which will be used to build the overlay network between hosts. Refer to *Chapter 1, Introduction to OpenStack Networking*, for ideas on how the overlay network should be architected. In this installation, all guest traffic through the overlay networks will traverse a dedicated network over the `eth1` interface that we configured earlier in this chapter.

Update the `local_ip` configuration option in the `[vxlan]` section of the Open vSwitch agent configuration file accordingly on `compute02`, `compute03`, and `snat01`:

```
[vxlan]
...
local_ip = 10.20.0.X
```

The following table provides the interfaces and addresses to be configured on each host. Substitute these for `X` where appropriate:

Hostname	Interface	IP Address
compute02	eth1	10.20.0.102
compute03	eth1	10.20.0.103
snat01	eth1	10.20.0.104

Bridge mappings

The `bridge_mappings` configuration option describes the mapping of an artificial label to a virtual switch created with Open vSwitch. Unlike the Linux bridge driver that configures a separate bridge for every network, each with its own interface, the Open vSwitch driver uses a single virtual switch containing a single physical interface and uses flow rules to tag traffic if necessary.

When networks are created, they are associated with an interface label, such as `physnet1`. The label `physnet1` is then mapped to a bridge, such as `br-eth1`, which contains the physical interface `eth1`. The mapping of the label to the bridge interface is handled by the `bridge_mappings` option. This mapping can be observed as follows:

```
bridge_mappings = physnet1:br-eth1
```

The chosen label(s) must be consistent between all nodes in the environment that are expected to handle traffic for a given network created with Neutron. However, the physical interface mapped to the label may be different. A difference in mappings is often observed when one node maps `physnet1` to a one gigabit-capable bridge, while another maps `physnet1` to a ten gigabit-capable bridge.

Multiple bridge mappings are allowed and can be added using a comma-separated list:

```
bridge_mappings = physnet1:br-eth1,physnet2:br-eth2
```

In this installation process, `physnet1` will be used as the interface label and will map to the bridge `br-eth2`. Update the Open vSwitch agent configuration file accordingly on `compute02`, `compute03`, and `snat01`:

```
[ovs]
...
bridge_mappings = physnet1:br-eth2
```

Configuring the bridges

To configure a bridge with Open vSwitch, use the Open vSwitch utility `ovs-vsctl`. Create the bridge `br-eth2` on `compute02`, `compute03`, and `snat01`, as follows:

```
# ovs-vsctl add-br br-eth2
```

Use the `ovs-vsctl add-port` command to add physical interface `eth2` to the bridge like so:

```
# ovs-vsctl add-port br-eth2 eth2
```

The configuration of the bridge should persist reboots. However, the bridge interface can also be configured in `/etc/network/interfaces` if necessary using the following syntax:

```
auto br-eth2
allow-ovs br-eth2
iface br-eth2 inet manual
    ovs_type OVSBridge
    ovs_port seth2

allow-br-eth2 eth2
iface eth2 inet manual
    ovs_bridge br-eth2
    ovs_type OVSPort
```



Please note that the physical switch port connected to `eth2` must support 802.1q VLAN tagging if VLAN networks of any type are to be created. On many switches, the switch port can be configured as a trunk port.

Firewall driver

The `firewall_driver` configuration option instructs Neutron to use a particular firewall driver for security group functionality. Different firewall drivers may be configured based on the mechanism driver in use.

Update the ML2 configuration file on `compute02` and `compute03` and define the appropriate `firewall_driver` in the `[securitygroup]` section on a single line:

```
[securitygroup]
...
firewall_driver = iptables_hybrid
```

The `iptables_hybrid` firewall driver implements firewall rules using `iptables` and relies on the use of Linux bridges in-between the instance's tap interface and the integration bridge. The `openvswitch` firewall driver, on the other hand, implements firewall rules using `OpenFlow` and does not rely on Linux bridges or `iptables`. As of the Pike release of `OpenStack`, the `openvswitch` firewall driver is not production-ready and is not recommended.

If you do not want to use a firewall and want to disable the application of security group rules, set `firewall_driver` to `noop`.

Configuring the DHCP agent to use the Open vSwitch driver

For Neutron to properly connect DHCP namespace interfaces to the appropriate network bridge, the DHCP agent on the node hosting the agent must be configured to use the Open vSwitch interface driver, as shown here:

```
[DEFAULT]
...
interface_driver = openvswitch
```

In this environment, the DHCP agent is running on the `controller01` node utilizing the Linux bridge driver and agent, and the interface driver was configured to work with Linux bridges. No change is necessary at this time. For environments running only Open vSwitch, be sure to set the interface driver accordingly.

Restarting services

Now that the appropriate `OpenStack` configuration files have been modified to use Open vSwitch as the networking driver, certain services must be started or restarted for the changes to take effect.

The Open vSwitch network agent should be restarted on `compute02`, `compute03`, and `snat01`:

```
# systemctl restart neutron-openvswitch-agent
```

The following services should be restarted on the controller node:

```
# systemctl restart neutron-server
```

Verifying Open vSwitch agents

To verify that the Open vSwitch network agents have been properly checked in, issue the `openstack network agent list` command on the controller node:

```
root@controller01:~# openstack network agent list
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
12bd7389-3b23-478d-9467-ebb6be4920b0	Linux bridge agent	controller01	None	:~)	UP	neutron-linuxbridge-agent
4c2f803c-a7ed-43e4-9625-2d2409f8571a	DHCP agent	controller01	nova	:~)	UP	neutron-dhcp-agent
54dd16aa-f0d4-41a7-b2ca-e1aae6959657	Open vSwitch agent	snat01	None	:~)	UP	neutron-openvswitch-agent
69eeed18-4079-45ba-9517-cc7785bc8599	Linux bridge agent	compute01	None	:~)	UP	neutron-linuxbridge-agent
a19d0599-4306-4f53-bd44-aa549b6d028d	Metadata agent	controller01	None	:~)	UP	neutron-metadata-agent
ace32522-1261-4814-b635-622329a019a9	Open vSwitch agent	compute03	None	:~)	UP	neutron-openvswitch-agent
fcbad2eb-f964-4434-9125-67162e1a1be7	Open vSwitch agent	compute02	None	:~)	UP	neutron-openvswitch-agent

The Open vSwitch agent on `compute02`, `compute03`, and `snat01` should now be visible in the output with a state of `UP`. If an agent is not present, or the state is `DOWN`, you will need to troubleshoot agent connectivity issues by observing log messages found in `/var/log/neutron/neutron-openvswitch-agent.log` on the respective host.

Summary

This chapter saw us installing and configuring the Neutron Open vSwitch mechanism driver and agent on two `compute` nodes and a dedicated `network` node, which will be used for distributed virtual routing functions at a later time. Instances scheduled to `compute02` and `compute03` will leverage Open vSwitch virtual network components, while `compute01` and network services on `controller01` will leverage Linux bridges.

Both the Linux bridge and Open vSwitch drivers and agents for Neutron provide unique solutions to the same problem of connecting virtual machine instances to the network. The use of Open vSwitch relies on flow rules to determine how traffic in and out of the environment should be processed and requires both user space utilities and kernel modules to perform such actions. On the other hand, the use of Linux bridges requires the `veth` and `bridge` kernel modules and relies on the use of VLAN and VXLAN interfaces on the host to bridge instances to the physical network. For simple environments, I recommend using the ML2 plugin and Linux bridge mechanism driver and agent, unless integration with OpenFlow controllers or the use of a third-party solution or plugin is required. Other Neutron technologies, such as distributed virtual routers, are only available when using the Open vSwitch driver and agent.

In the next chapter, you will be guided through the process of creating different types of networks to provide connectivity to instances. The process of creating networks is the same for both Linux bridge and Open vSwitch-based environments, but the underlying network implementation will vary based on the driver and agent in use.

6

Building Networks with Neutron

In the [Chapter 4](#), *Virtual Network Infrastructure Using Linux Bridges*, and [Chapter 5](#), *Building a Virtual Switching Infrastructure Using Open vSwitch*, we laid down a virtual switching infrastructure that would support the OpenStack Neutron networking features that we have discussed in this book. In this chapter, we will build network resources on top of that foundation. These will be able to be consumed by instances.

In this chapter, I will guide you through the following tasks:

- Managing networks using the CLI and dashboard
- Managing IPv4 subnets using the CLI and dashboard
- Managing subnet pools
- Creating ports

Networks, subnets, and ports are the core resources of the Neutron API, which was introduced in [Chapter 3](#), *Installing Neutron*. The relationship between these core resources and instances and other virtual network devices can be observed in the following sections.

Network management in OpenStack

OpenStack can be managed in a variety of ways, including through the Horizon dashboard, the Neutron API, and the OpenStack CLI. A command-line client, provided by the `python-openstackclient` package, gives users the ability to execute commands from a shell that interfaces with the Neutron API. To enter the shell, type `openstack` in a terminal connected to the controller node, like so:

```
root@controller01:~# openstack
(openstack)
```

The `openstack` shell features tab completion and a `help` command that lists all of the available commands within the shell. Openstack-related commands can also be executed straight from the Linux command line by using the `openstack` client like so:

```
# openstack network list
# openstack server create
```

The client provides a number of commands that assist with the creation, modification, and deletion of networks, subnets, and ports.



The `openstack` client is preferred over the `neutron` client moving forward, but may not have complete command parity. The use of the `neutron` client should be limited.

All `openstack` client commands can be determined by using the `--help` flag. The primary commands associated with network management that will be discussed in this chapter are listed in the following table:

Network Commands	Description
<code>network create</code>	Creates a new network
<code>network delete</code>	Deletes a network(s)
<code>network show</code>	Shows network details
<code>network list</code>	Lists networks
<code>network set</code>	Sets network properties
<code>network unset</code>	Unsets network properties

Whether you've chosen a Linux bridge or Open vSwitch-based virtual networking infrastructure, the process to create, modify, and delete networks and subnets is the same. Behind the scenes, however, the process of connecting instances and other resources to the network differs greatly.

Provider and tenant networks

There are two categories of networks that can provide connectivity to instances and other network resources, including virtual routers:

- Provider networks
- Project or tenant networks, also known as self-service networks

Every network created in Neutron, whether created by a regular user or a user with an admin role, has provider attributes that describe that network. Attributes that describe a network include the network's type, such as flat, VLAN, GRE, VXLAN, or local, the physical network interface that the traffic will traverse, and the segmentation ID of the network. The difference between a provider and project or tenant network is in who or what sets those attributes and how they are managed within OpenStack.

Provider networks can only be created and managed by an OpenStack administrator, since they require knowledge and configuration of the physical network infrastructure.



An OpenStack administrator refers to a user associated with the `admin` role in Keystone.

When a provider network is created, the administrator must manually specify the provider attributes for the network in question. The administrator is expected to have some understanding of the physical network infrastructure and may be required to configure switch ports for proper operation. Provider networks allow for either virtual machine instances or virtual routers created by users to be connected to them. When a provider network is configured to act as an external network for Neutron routers, the provider network is known as an **external provider network**. Provider networks are often configured as flat or vlan networks, and utilize an external routing device to properly route traffic in and out of the cloud.

Self-service networks, unlike provider networks, are created by users and are usually isolated from other networks in the cloud. The inability to configure the physical infrastructure means that tenants will likely connect their networks to Neutron routers when external connectivity is required. Tenants are unable to specify provider attributes manually and are restricted to creating networks whose attributes have been pre-defined by the administrator in the Neutron configuration files. More information on the configuration and use of Neutron routers begins in [Chapter 10, Creating Standalone Routers with Neutron](#).

Managing networks in the CLI

To create networks using the OpenStack client, use the `network create` command that's shown here:

```
(openstack) network create -h
usage: network create [-h] [-f {json,shell,table,value,yaml}] [-c COLUMN]
                    [--max-width <integer>] [--fit-width] [--print-empty]
                    [--noindent] [--prefix PREFIX] [--share | --no-share]
                    [--enable | --disable] [--project <project>]
                    [--description <description>]
                    [--project-domain <project-domain>]
                    [--availability-zone-hint <availability-zone>]
                    [--enable-port-security | --disable-port-security]
                    [--external | --internal] [--default | --no-default]
                    [--qos-policy <qos-policy>]
                    [--transparent-vlan | --no-transparent-vlan]
                    [--provider-network-type <provider-network-type>]
                    [--provider-physical-network <provider-physical-network>]
                    [--provider-segment <provider-segment>]
                    [--tag <tag> | --no-tag]
                    <name>
```

The `--share` and `--no-share` arguments are used to share the network with all other projects, or limit the network to the owning project, respectively. By default, networks are not shared and can only be used by the owning project. Neutron's RBAC functionality can be used to share a network between a subset of projects and will be discussed in *Chapter 9, Role-Based Access Control*.

The `--enable` and `--disable` arguments are used to enable or disable the administrative state of the network.

The `--availability-zone-hint` argument is used to define the availability zone in which the network should be created. By default, all networks are placed in a single zone and all hosts within the environment are expected to have the capability of servicing traffic for the network. Network availability zones are an advanced networking topic that will be touched on in *Chapter 14, Advanced Networking Topics*.

The `--enable-port-security` and `--disable-port-security` arguments are used to enable or disable port security on any port that's created from a given network. Port security refers to the use and support of security groups and MAC/ARP filtering on Neutron ports, and will be discussed further in *Chapter 8, Managing Security Groups*.

The `--qos-policy` argument is used to set a QoS policy on any port created in the network. The configuration and use of the Quality of Service extension is an advanced topic that is outside the scope of this book.

The `--external` and `--internal` arguments are used to specify whether the network is considered an external provider network that's eligible for use as a gateway network or floating IP pool, or if it is only to be used internally within the cloud. The default value for the `router:external` attribute of a network is `false`, or `internal`. For more information on Neutron routers, refer to *Chapter 10, Creating Standalone Routers with Neutron*.

The `--default` and `--no-default` arguments are used to specify whether or not a network should act as the default external network for the cloud and are often used for the network auto-allocation feature that was introduced in Mitaka.



The network auto-allocation features of Neutron require the `auto-allocated-topology`, `router`, `subnet_allocation`, and `external-net` extensions. Some of these are not enabled by default. For more information on these features, please refer to the upstream documentation that's available at <https://docs.openstack.org/neutron/pike/admin/config-auto-allocation.html>.

The `--provider-network-type` argument defines the type of network being created. Available options include `flat`, `VLAN`, `local`, `GRE`, `geneve`, and `VXLAN`. For a network type to be functional, the corresponding type driver must be enabled in the ML2 configuration file and supported by the enabled mechanism driver.

The `--provider-physical-network` argument is used to specify the network interface that will be used to forward traffic through the host. The value specified here corresponds to the provider label defined by the `bridge_mappings` or `physical_interface_mappings` options that are set in the Neutron agent configuration file.

The `--provider-segment` argument is used to specify a unique ID for the network that corresponds to the respective network type. If you are creating a VLAN-type network, the value used will correspond to the 802.1q VLAN ID which is mapped to the network and should be trunked to the host. If you are creating a GRE or VXLAN network, the value should be arbitrary, but unique; the integer is not used by any other network of the same type. This ID is used to provide network isolation via the GRE key or VXLAN.

VNI header fields for GRE and VXLAN networks, respectively. When the provider-segment parameter is not specified, one is automatically allocated from the tenant range that's specified in the plugin configuration file. Users have no visibility or option to specify a segment ID when creating networks. When all available IDs in the range that are available to projects are exhausted, users will no longer be able to create networks of that type.

The `--tag` and `--no-tag` arguments are used to apply or remove a tag from a network. Tags are label that are applied to a network resource which can be used by a client for filtering purposes.



By default, provider attributes can only be set by users with the admin role in Keystone. Users without the admin role are beholden to values provided by Neutron based on configurations set in the ML2 and agent configuration files.

Creating a flat network in the CLI

If you recall from previous chapters, a flat network is a network in which no 802.1q tagging takes place.

The syntax to create a flat network can be seen here:

```
openstack network create
--provider-network-type flat
--provider-physical-network <provider-physical-network>
<name>
```

The following is an example of using the OpenStack client to create a flat network by the name of `MyFlatNetwork`. The network will utilize an interface or bridge represented by the label `physnet1` and will be shared among all projects:

```

root@controller01:~# openstack network create --provider-network-type
> --provider-physical-network physnet1 \
> --share \
> MyFlatNetwork

```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-01-08T19:04:53Z
description	
dns_domain	None
id	d51943ef-8061-4bdb-b684-8a7d2b7ce73b
ipv4_address_scope	None
ipv6_address_scope	None
is_default	None
is_vlan_transparent	None
mtu	1500
name	MyFlatNetwork
port_security_enabled	False
project_id	877f949397ad428cbeaac4e5123bad83
provider:network_type	flat
provider:physical_network	physnet1
provider:segmentation_id	None
qos_policy_id	None
revision_number	1
router:external	Internal
segments	None
shared	True
status	ACTIVE
subnets	
tags	
updated_at	2018-01-08T19:04:53Z

In the preceding output, the project ID corresponds to the admin project where the user who executed the `network create` command was scoped. Since the network is shared, all projects can create instances and network resources that utilize the `MyFlatNetwork` network.

Attempting to create an additional flat network using the same `provider-physical-network` name of `physnet1` will result in an error, as shown in the following screenshot:

```
root@controller01:~# openstack network create --provider-network-type flat \  
> --provider-physical-network physnet1 \  
> MyFlatNetwork2  
Error while executing command: Conflict (HTTP 409) (Request-ID: req-47d76ebf-fd4c-48e4-90c1-21498199d761)
```

Because there is only one untagged or native VLAN available on the interface, Neutron cannot create a second flat network and returns a `Conflict` error. Given this limitation, flat networks are rarely utilized in favor of VLAN networks.

Creating a VLAN network in the CLI

A VLAN network is one in which Neutron will tag traffic based on an 802.1q segmentation ID. The syntax used to create a VLAN network can be seen here:

```
openstack network create  
--provider-network-type vlan  
--provider-physical-network <provider-physical-network>  
--provider-segment <provider-segment>  
<name>
```



By default, only users with the admin role are allowed to specify provider attributes.

Up to 4,096 VLANs can be defined on a single provider interface, though some of those may be reserved by the physical switching platform for internal use. The following is an example of using the OpenStack client to create a VLAN network by the name of `MyVLANNetwork`.

The network will utilize the bridge or interface represented by `physnet1`, and the traffic will be tagged with a VLAN identifier of 300. The resulting output is as follows:

```

root@controller01:~# openstack network create \
> --provider-network-type vlan \
> --provider-physical-network physnet1 \
> --provider-segment 300 \
> MyVLANNetwork
+-----+
| Field                                | Value                                |
+-----+
| admin_state_up                       | UP                                  |
| availability_zone_hints               |                                     |
| availability_zones                   |                                     |
| created_at                           | 2018-01-09T13:32:28Z              |
| description                           |                                     |
| dns_domain                           | None                                |
| id                                    | b2ca3d41-c79c-416a-8e86-0bb01060ddec |
| ipv4_address_scope                   | None                                |
| ipv6_address_scope                   | None                                |
| is_default                           | None                                |
| is_vlan_transparent                  | None                                |
| mtu                                   | 1500                                |
| name                                  | MyVLANNetwork                      |
| port_security_enabled                 | False                               |
| project_id                            | 877f949397ad428cbeaac4e5123bad83 |
| provider:network_type                 | vlan                                |
| provider:physical_network             | physnet1                            |
| provider:segmentation_id              | 300                                  |
| qos_policy_id                         | None                                |
| revision_number                       | 1                                    |
| router:external                       | Internal                             |
| segments                              | None                                |
| shared                                | False                               |
| status                                 | ACTIVE                              |
| subnets                              |                                     |
| tags                                  |                                     |
| updated_at                            | 2018-01-09T13:32:28Z              |
+-----+

```

To create an additional VLAN network that utilizes the provider interface, simply specify a different segmentation ID. In the following example, VLAN 301 is used for the new network, **MyVLANNetwork2**. The resulting output is as follows:

```

root@controller01:~# openstack network create \
> --provider-network-type vlan \
> --provider-physical-network physnet1 \
> --provider-segment 301 \
> MyVLANNetwork2

```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-01-09T13:34:39Z
description	
dns_domain	None
id	9cd264e9-d3b7-415e-bf49-5efe3463c18d
ipv4_address_scope	None
ipv6_address_scope	None
is_default	None
is_vlan_transparent	None
mtu	1500
name	MyVLANNetwork2
port_security_enabled	False
project_id	877f949397ad428cbeaac4e5123bad83
provider:network_type	vlan
provider:physical_network	physnet1
provider:segmentation_id	301
qos_policy_id	None
revision_number	1
router:external	Internal
segments	None
shared	False
status	ACTIVE
subnets	
tags	
updated_at	2018-01-09T13:34:40Z



When using VLAN networks, don't forget to configure the physical switch port interface as a trunk. This configuration will vary between platforms and is outside the scope of this book.

Creating a local network in the CLI

When an instance sends traffic on a local network, the traffic remains isolated to the instance and other interfaces connected to the same bridge and/or segment. Services such as DHCP and metadata might not be available to instances on local networks, especially if they are located on different nodes.

To create a local network, use the following syntax:

```
openstack network create
--provider-network-type local
<name>
```

When using the Linux bridge driver, a bridge is created for the local network but no physical or tagged interface is added. Traffic is limited to any port connected to the bridge and will not leave the host. When using the Open vSwitch driver, instances are attached to the integration bridge and can only communicate with other instances in the same local VLAN.

Listing networks in the CLI

To list networks known to Neutron, use the `openstack network list` command, as shown in the following screenshot:

```
root@controller01:~# openstack network list
```

ID	Name	Subnets
9cd264e9-d3b7-415e-bf49-5efe3463c18d	MyVLANNetwork2	
b2ca3d41-c79c-416a-8e86-0bb01060ddec	MyVLANNetwork	
d51943ef-8061-4bdb-b684-8a7d2b7ce73b	MyFlatNetwork	

The list output provides the network ID, network name, and any associated subnets. An OpenStack user with the admin role can see all networks, while regular users can see shared networks and networks within their respective projects.

If tags are utilized, the results may be filtered using the following parameters:

- `--tags <tag>`: Lists networks which have all given tag(s)
- `--any-tags <tag>`: Lists networks which have any given tag(s)
- `--not-tags <tag>`: Excludes networks which have all given tag(s)
- `--not-any-tags <tag>`: Excludes networks which have any given tag(s)

Showing network properties in the CLI

To show the properties of a network, use the `openstack network show` command, as shown here:

```
openstack network show <network>
```

The output of the preceding command can be observed in the following screenshot:

```
root@controller01:~# openstack network show MyVLANNetwork
```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-01-09T13:32:28Z
description	
dns_domain	None
id	b2ca3d41-c79c-416a-8e86-0bb01060ddec
ipv4_address_scope	None
ipv6_address_scope	None
is_default	None
is_vlan_transparent	None
mtu	1500
name	MyVLANNetwork
port_security_enabled	False
project_id	877f949397ad428cbeaac4e5123bad83
provider:network_type	vlan
provider:physical_network	physnet1
provider:segmentation_id	300
qos_policy_id	None
revision_number	1
router:external	Internal
segments	None
shared	False
status	ACTIVE
subnets	
tags	
updated_at	2018-01-09T13:32:28Z

Various properties of the network can be seen in the output, including the administrative state, default MTU, sharing status, and more. Network provider attributes are hidden from ordinary users and can only be seen by users with the admin role.

Updating network attributes in the CLI

At times, it may be necessary to update the attributes of a network after it has been created. To update a network, use the `openstack network set` and `openstack network unset` commands as follows:

```
openstack network set
[--name <name>]
[--enable | --disable]
[--share | --no-share]
[--description <description>]
[--enable-port-security | --disable-port-security]
[--external | --internal]
[--default | --no-default]
[--qos-policy <qos-policy> | --no-qos-policy]
[--tag <tag>] [--no-tag]
<network>

openstack network unset
[--tag <tag> | --all-tag]
<network>
```

The `--name` arguments can be used to change the name of the network. The ID will remain the same.

The `--enable` and `--disable` arguments are used to enable or disable the administrative state of the network.

The `--share` and `--no-share` arguments are used to share the network with all other projects, or limit the network to the owning project, respectively. Once other projects have created ports in a shared network, it is not possible to revoke the shared state of the network until those ports have been deleted.

The `--enable-port-security` and `--disable-port-security` arguments are used to enable or disable port security on any port created from the given network. Port security refers to the use and support of security groups and MAC/ARP filtering on Neutron ports, and will be discussed further in *Chapter 8, Managing Security Groups*.

The `--external` and `--internal` arguments are used to specify whether the network is considered an external provider network that's eligible for use as a gateway network and floating IP pool, or if it is only to be used internally within the cloud. The default value for the `router:external` attribute of a network is `false`, or `internal`.

The `--default` and `--no-default` arguments are used to specify whether or not a network should act as the default external network for the cloud.

The `--qos-policy` argument is used to set a QoS policy on any port created in the network.

The `--tag` argument, when used with the `set` command, will add the specified tag to the network. When used with the `unset` command, the specified tag will be removed from the network. Using `--all-tag` with the `unset` command will remove all tags from the network.

Provider attributes are among those that cannot be changed once a network has been created. If a provider attribute such as `segmentation_id` must be changed after the network has been created, you must delete and recreate the network.

Deleting networks in the CLI

To delete a network, use the `openstack network delete` command and specify the ID or name of the network:

```
openstack network delete <network> [<network> ...]
```

To delete a network named `MySampleNetwork`, you can enter the following command:

```
openstack network delete MySampleNetwork
```

Alternatively, you can use the network's ID:

```
openstack network delete 3a342db3-f918-4760-972a-cb700d5b6d2c
```

Multiple networks can also be deleted simultaneously, like so:

```
openstack network delete MySampleNetwork MyOtherSampleNetwork
```

Neutron will successfully delete the network as long as ports utilized by instances, routers, floating IPs, load balancer virtual IPs, and other user-created ports have been deleted. Any Neutron-created port, like those used for DHCP namespaces, will be automatically deleted when the network is deleted.

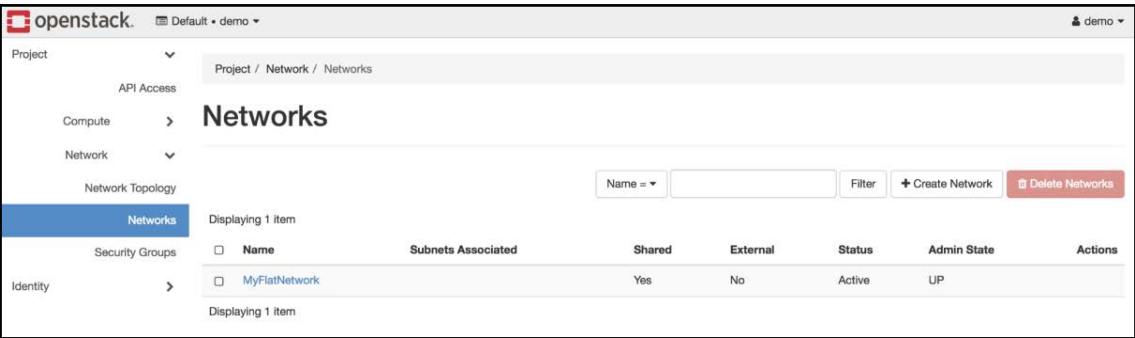
Creating networks in the dashboard

Networks can be created in the Horizon dashboard, but the method of doing so may vary based on the role of the user. Users with the admin role have the ability to create networks on behalf of other projects and specify provider attributes, while users without the admin role are limited to creating networks in their respective projects and have the same capabilities that are available via the OpenStack client. Both methods are described in the following sections.

Via the Project panel

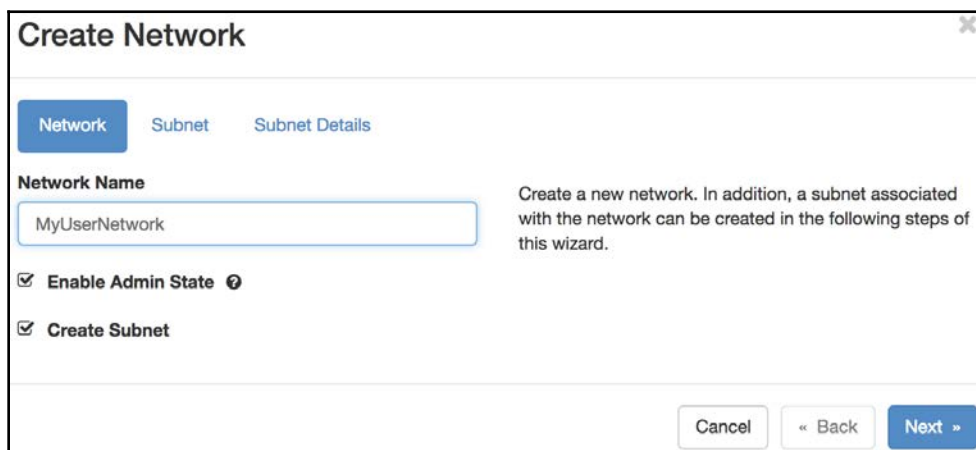
Users can create networks using a wizard located within the Project tab in the dashboard. To create a network, login as the user in the demo project and perform the following steps:

1. Navigate to **Project | Network | Networks**:



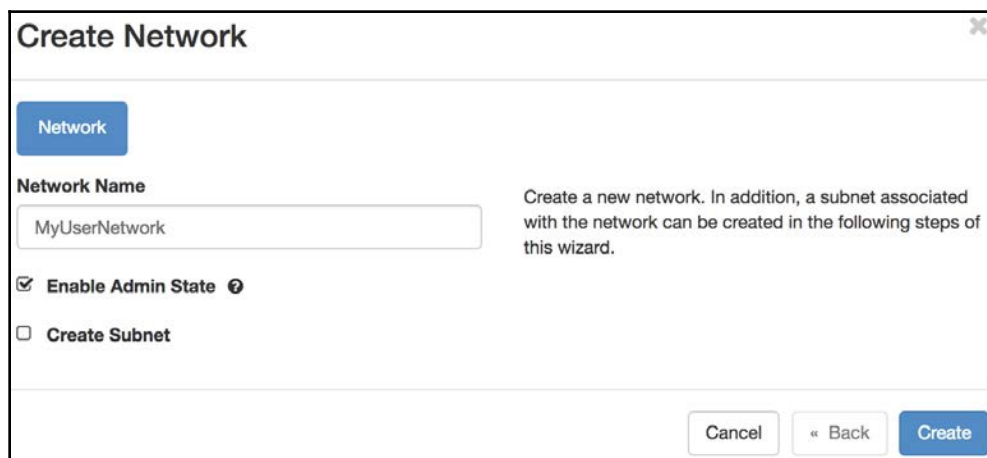
In the preceding screenshot, notice that there are no actions available next to the networks that are currently defined. Even though the networks are shared, they are not modifiable by users and can only be modified by an administrator.

2. Click the **Create Network** button in the upper right-hand corner of the screen. A window will appear that will allow you to specify network properties:



The screenshot shows a 'Create Network' dialog box with a close button (X) in the top right corner. It has three tabs: 'Network' (selected), 'Subnet', and 'Subnet Details'. Under the 'Network' tab, there is a 'Network Name' field containing 'MyUserNetwork'. To the right of the field is a text box that says: 'Create a new network. In addition, a subnet associated with the network can be created in the following steps of this wizard.' Below the field are two checked checkboxes: 'Enable Admin State' and 'Create Subnet'. At the bottom right are three buttons: 'Cancel', '< Back', and 'Next >'.

3. From the **Network** tab, you can define the Network Name and Admin State (on or off). Users creating networks within the dashboard are not required to create a subnet at the time the network is created. By unchecking the **Create Subnet** checkbox in the **Subnet** tab, the network creation process can be completed:



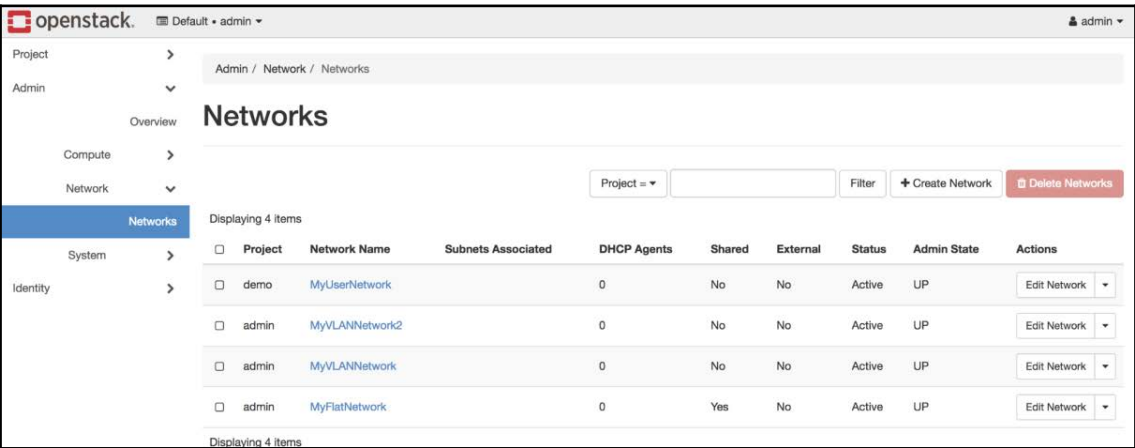
The screenshot shows the same 'Create Network' dialog box, but now the 'Subnet' tab is selected. The 'Network Name' field still contains 'MyUserNetwork'. The 'Enable Admin State' checkbox is still checked. The 'Create Subnet' checkbox is now unchecked. The text box on the right remains the same. At the bottom right are three buttons: 'Cancel', '< Back', and 'Create'.

The process to create subnets within the dashboard will be explained later on in this chapter.

Via the Admin panel

Users with the admin role will notice additional functionality available within the dashboard. An Admin panel can be seen on the left-hand side of the dashboard, which allows users with the admin role to manipulate networks outside of their respective project. To create a network in the dashboard as a cloud administrator, login as the admin user and perform the following steps:

1. Navigate to **Admin | Network | Networks**:



2. Click on **Create Network** in the upper right-hand corner of the screen. A wizard will appear that will allow you to specify network properties:

Create Network

Network * Subnet Subnet Details

Name

Project *

Select a project

Provider Network Type * ?

Local

☒ Enable Admin State

☐ Shared

☐ External Network

☒ Create Subnet

Create a new network. In addition, a subnet associated with the network can be created in the following steps of this wizard.

Cancel < Back Next >

Various network attributes can be set from the wizard, including the network type, interface, and segmentation ID, if applicable. Other options include associating the network with a project, setting the administrative state, enabling sharing, creating a subnet, and enabling the network to be used as an external network for Neutron routers. When complete, click the **Create Network** button to create the network.

Subnet management in OpenStack

A subnet in Neutron is a Layer 3 object and can be an IPv4 or IPv6 address block defined using classless inter-domain routing (CIDR) notation. CIDR is a method of allocating IP addresses using variable-length subnet masking, or VLSM. Subnets have a direct relationship to networks and cannot exist without them.



More information on CIDR and VLSM can be found on Wikipedia at http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing.

The primary commands associated with subnet management that will be discussed in this chapter are listed in the following table:

Subnet Commands	Description
subnet create	Creates a subnet
subnet delete	Deletes a subnet(s)
subnet show	Displays subnet details
subnet list	Lists subnets
subnet set	Sets subnet properties
subnet unset	Unsets subnet properties
subnet pool create	Creates a subnet pool
subnet pool delete	Deletes a subnet pool(s)
subnet pool list	Lists subnet pools
subnet pool set	Sets subnet pool properties
subnet pool show	Displays subnet pool details
subnet pool unset	Unsets subnet pool properties

Working with IPv4 addresses

A few examples of IPv4 addresses and subnets described using CIDR notation are as follows:

- 192.168.100.50/24 represents the IP address 192.168.100.50, its associated routing prefix 192.168.100.0, and the subnet mask 255.255.255.0 (that is, 24 "1" bits). There are 256 total addresses in a /24, with 254 addresses available for use.
- 172.16.1.200/23 represents the IP address 172.16.0.200, its associated routing prefix 172.16.0.0, and the subnet mask 255.255.254.0 (that is, 23 "1" bits). There are 512 total addresses in a /23, with 510 addresses available for use.
- 10.0.10.4/22 represents the IP address 10.0.10.4, its associated routing prefix 10.0.8.0, and the subnet mask 255.255.252.0 (that is, 22 "1" bits). There are 1,024 total addresses in a /22, with 1,022 addresses available for use.

Not every address in a subnet might be usable, as the first and last addresses are usually reserved as the network and broadcast addresses, respectively. As a result, Neutron will not assign the first or last address of a subnet to network resources, including instances. Use the following formula to determine the total number of usable addresses in a subnet when sizing your network. The x variable represents the number of host bits available in the subnet mask:

$$2^x - 2 = \text{number of useable addresses in a subnet}$$

Keep in mind that when creating a subnet, it is important to plan ahead as the subnet mask, or CIDR, is currently not an updatable attribute. When instances and other resources consume all of the available IP addresses in a subnet, devices can no longer be added to the network. To resolve this, a new subnet will need to be created and added to the existing network, or an entirely new network and subnet will need to be created. Depending on your network infrastructure, this might not be an easy change to implement.

VLSM and CIDR, to an extent, are important when designing networks for use within an OpenStack cloud and will play an even more important role when we discuss the use of subnet pools later on in this chapter.

Working with IPv6 addresses

IPv6 is a first-class citizen in OpenStack Networking, but is an advanced topic that will not be discussed in this chapter. All of the subnet-related commands described here will behave similarly when defining IPv6 subnets, but additional parameters may be required.



For more information on IPv6 in Pike, please refer to the upstream documentation available at <https://docs.openstack.org/neutron/pike/admin/config-ipv6.html>.

Creating subnets in the CLI

To create a subnet using the OpenStack client, use the `subnet create` command, as shown here:

```
(openstack) subnet create -h
usage: subnet create [-h] [-f {json,shell,table,value,yaml}] [-c COLUMN]
                    [--max-width <integer>] [--fit-width] [--print-empty]
                    [--noindent] [--prefix PREFIX] [--project <project>]
                    [--project-domain <project-domain>]
                    [--subnet-pool <subnet-pool> | --use-default-subnet-pool]
                    [--prefix-length <prefix-length>]
                    [--subnet-range <subnet-range>] [--dhcp | --no-dhcp]
                    [--gateway <gateway>] [--ip-version {4,6}]
                    [--ipv6-ra-mode {dhcpv6-stateful,dhcpv6-stateless,slaac}]
                    [--ipv6-address-mode {dhcpv6-stateful,dhcpv6-stateless,slaac}]
                    [--network-segment <network-segment>] --network <network>
                    [--description <description>]
                    [--allocation-pool start=<ip-address>,end=<ip-address>]
                    [--dns-nameserver <dns-nameserver>]
                    [--host-route destination=<subnet>,gateway=<ip-address>]
                    [--service-type <service-type>] [--tag <tag> | --no-tag]
                    name
```

The `--project` argument specifies the ID of the project that the subnet should be associated with and is only available to users with the admin role. This should be the same project associated with the parent network.

The `--project-domain` argument specifies the ID of the project domain that the subnet should be associated with and is only available to users with the admin role (this is not commonly used).

The `--subnet-pool` argument specifies the pool from which the subnet will obtain a CIDR.

The `--use-default-subnet-pool` argument instructs Neutron to choose the default subnet pool to obtain a CIDR. Using this parameter first requires the creation of at least one subnet pool.

The `--prefix-length` argument specifies the prefix length for subnet allocation from the subnet pool.

The `--subnet-range` argument specifies the CIDR notation of the subnet being created. This option is required unless the subnet is associated with a subnet pool.

The `--dhcp` and `--no-dhcp` argument enable or disable DHCP services for the subnet, respectively. DHCP is enabled by default.



DHCP is not required for network operation. Disabling DHCP simply means that instances attached to the subnet will not utilize DHCP for dynamic interface configuration. Instead, interfaces will need to be configured manually or by using scripts or other methods within the instance itself. Instances are still allocated IP addresses within the allocation pool range whether DHCP is disabled or enabled.

The `--gateway` argument defines the gateway address for the subnet. The three possible options include `auto`, `none`, or an IP address of the user's choosing. When the subnet is attached to the instance side of a Neutron router, the router's interface will be configured with the address specified here. The address is then used as the default gateway for instances in the subnet. If the subnet is attached to the external side of a Neutron router, the address is used as the default gateway for the router itself. To see this behavior in action, refer to *Chapter 10, Creating Standalone Routers with Neutron*. If a gateway is not specified, Neutron defaults to `auto` and uses the first available address in the subnet.

The `--ip-version` argument specifies the version of the IP protocol represented by the subnet. Possible options are 4 for IPv4 and 6 for IPv6. The default is 4.

The `--ipv6-ra-mode` argument defines the router advertisement mode for the subnet when IPv6 is used. Possible options include `dhcpv6-stateful`, `dhcpv6-stateless`, and `slaac`.

The `--ipv6-address-mode` argument defines the address mode for the subnet when IPv6 is used. Possible options include `dhcpv6-stateful`, `dhcpv6-stateless`, and `slaac`.



Not all combinations of the `ipv6-ra-mode` and `ipv6-address-mode` arguments are valid. To review both valid and invalid use cases, please refer to the API guide at <https://docs.openstack.org/neutron/pike/admin/config-ipv6.htm>
1. More information on IPv6 can be found in the appendix.

The `--network-segment` argument specifies the network segment to associate with the subnet.

The `--network` argument specifies the network the subnet should be associated with. Multiple subnets can be associated with a single network as long as the subnet range does not overlap with another subnet in the same network.

The `--allocation-pool` argument specifies the range of IP addresses within the subnet that can be assigned to ports. IP addresses cannot be excluded from a single range. However, multiple allocation pools can be defined that exclude addresses. For example, to exclude 192.168.1.50–55 from 192.168.1.0/24, the following syntax would be needed:

```
openstack subnet create MyFlatNetwork
--subnet-range 192.168.1.0/24
--allocation-pool start=192.168.1.2,end=192.168.1.49
--allocation-pool start=192.168.1.56,end=192.168.1.254
```



Depending on the type of network in use, it is possible for devices outside of OpenStack to coexist with instances in the same network and subnet. The allocation pool(s) should be defined so that addresses allocated to instances do not overlap with devices outside of the OpenStack cloud.

The `--dns-nameserver` argument specifies a DNS name server for the subnet. This option can be repeated to set multiple name servers. However, the default maximum number of name servers is five per subnet and can be modified by updating the `max_dns_nameservers` configuration option in the `/etc/neutron/neutron.conf` file.

The `--host-route` argument specifies one or more static routes defined as destinations and next hop pairs to be injected into an instance's routing table via DHCP. This option can be used multiple times to specify multiple routes. The default maximum number of routes per subnet is 20 and can be modified by updating the `max_subnet_host_routes` configuration option in the `/etc/neutron/neutron.conf` file.

The `--tag` and `--no-tag` argument are used to apply or remove a tag from a subnet. Tags are labels that are applied to a network resource, which can be used by a client for filtering purposes.

The `name` argument specifies the name of the subnet. While you can create multiple subnets with the same name, it is recommended that subnet names remain unique for easy identification.

Creating a subnet in the CLI

To demonstrate this command in action, create a subnet within the `MyFlatNetwork` network with the following characteristics:

- Name: `MyFlatSubnet`
- Internet Protocol: `IPv4`

- Subnet: 192.168.100.0/24
- Subnet mask: 255.255.255.0
- External gateway: 192.168.100.1
- DNS servers: 8.8.8.8, 8.8.4.4

To create the subnet and associate it with `MyFlatNetwork`, refer to the following screenshot:

```

root@controller01:~# openstack subnet create \
> --subnet-range 192.168.100.0/24 \
> --gateway 192.168.100.1 \
> --ip-version 4 \
> --network MyFlatNetwork \
> --dns-nameserver 8.8.8.8 \
> --dns-nameserver 8.8.4.4 \
> MyFlatSubnet

```

Field	Value
allocation_pools	192.168.100.2-192.168.100.254
cidr	192.168.100.0/24
created_at	2018-01-15T01:22:50Z
description	
dns_nameservers	8.8.4.4, 8.8.8.8
enable_dhcp	True
gateway_ip	192.168.100.1
host_routes	
id	63eaa79b-8129-4fc4-a783-cc7ba917d462
ip_version	4
ipv6_address_mode	None
ipv6_ra_mode	None
name	MyFlatSubnet
network_id	d51943ef-8061-4bdb-b684-8a7d2b7ce73b
project_id	877f949397ad428cbeaac4e5123bad83
revision_number	0
segment_id	None
service_types	
subnetpool_id	None
tags	
updated_at	2018-01-15T01:22:50Z
use_default_subnet_pool	None

Listing subnets in the CLI

To list existing subnets, use the `openstack subnet list` command, as shown in the following screenshot:

```
root@controller01:~# openstack subnet list
```

ID	Name	Network	Subnet
63eaa79b-8129-4fc4-a783-cc7ba917d462	MyFlatSubnet	d51943ef-8061-4bdb-b684-8a7d2b7ce73b	192.168.100.0/24

By default, the command output provides the ID, name, CIDR notation, and associated networks of each subnet that are available to the user. Users with the admin role may see all subnets, while ordinary users are restricted to subnets within their project or subnets associated with shared networks. The `openstack subnet list` command also accepts filters that narrow down returned results.

If tags are utilized, the results may be filtered using the following parameters:

- `--tags <tag>`: Lists subnets which have all given tag(s)
- `--any-tags <tag>`: Lists subnets which have any given tag(s)
- `--not-tags <tag>`: Excludes subnets which have all given tag(s)
- `--not-any-tags <tag>`: Excludes subnets which have any given tag(s)

Additional details can be found by using the `-h` or `--help` options.

Showing subnet properties in the CLI

To show the properties of a subnet, use the `openstack subnet show` command, as shown here:

```
openstack subnet show <subnet>
```

The output of the preceding command can be observed in the following screenshot:

```
root@controller01:~# openstack subnet show MyFlatSubnet
```

Field	Value
allocation_pools	192.168.100.2-192.168.100.254
cidr	192.168.100.0/24
created_at	2018-01-15T01:22:50Z
description	
dns_nameservers	8.8.4.4, 8.8.8.8
enable_dhcp	True
gateway_ip	192.168.100.1
host_routes	
id	63eaa79b-8129-4fc4-a783-cc7ba917d462
ip_version	4
ipv6_address_mode	None
ipv6_ra_mode	None
name	MyFlatSubnet
network_id	d51943ef-8061-4bdb-b684-8a7d2b7ce73b
project_id	877f949397ad428cbeaac4e5123bad83
revision_number	0
segment_id	None
service_types	
subnetpool_id	None
tags	
updated_at	2018-01-15T01:22:50Z
use_default_subnet_pool	None

Updating a subnet in the CLI

To update a subnet in the CLI, use the `openstack subnet set` and `openstack subnet unset` commands as follows:

```
openstack subnet set
[--name <name>] [--dhcp | --no-dhcp]
[--gateway <gateway>]
[--description <description>] [--tag <tag>]
[--no-tag]
[--allocation-pool start=<ip-address>,end=<ip-address>]
[--no-allocation-pool]
[--dns-nameserver <dns-nameserver>]
[--no-dns-nameservers]
[--host-route destination=<subnet>,gateway=<ip-address>]
[--no-host-route] [--service-type <service-type>]
<subnet>
```

```
openstack subnet unset  
[--allocation-pool start=<ip-address>,end=<ip-address>]  
[--dns-nameserver <dns-nameserver>]  
[--host-route destination=<subnet>,gateway=<ip-address>]  
[--service-type <service-type>]  
[--tag <tag> | --all-tag]  
<subnet>
```

The `--name` argument specifies the updated name of the subnet.

The `--dhcp` and `--no-dhcp` arguments enable or disable DHCP services for the subnet, respectively.



Instances that rely on DHCP to procure or renew an IP address lease might lose network connectivity over time if DHCP is disabled.

The `--gateway` argument defines the gateway address for the subnet. The two possible options when updating a subnet includes none or an IP address of the user's choosing.

The `--tag` argument, when used with the `set` command, will add the specified tag to the subnet. When used with the `unset` command, the specified tag will be removed from the subnet. Using the `--all-tag` with the `unset` command will remove all tags from the subnet.

The `--allocation-pool` argument, when used with the `set` command, adds the specified pool to the subnet. When used with the `unset` command, the specified pool will be removed from the subnet.

The `--dns-nameserver` argument, when used with the `set` command, adds the specified DNS name server to the subnet. When used with the `unset` command, the specified DNS name server is removed from the subnet. Using `--no-name-servers` will remove all DNS name servers from the subnet.

The `--host-route` argument, when used with the `set` command, adds the specified static route defined using destination and next hop pairs. When used with the `unset` command, the specified route is removed. Using `--no-host-route` with the `set` command will remove all host routes from the subnet.

The `subnet` argument specifies the name of the subnet being modified.

Creating subnets in the dashboard

Subnets can be created in the Horizon dashboard, but the method of doing so may vary depending on the role of the user. Users with the admin role have the ability to create subnets on behalf of other projects and specify other attributes that may not be available to ordinary users. Users without the admin role are limited to creating subnets in their respective projects and have the same capabilities that are available via the OpenStack client. Both methods are described in the following sections.

Via the Project tab

Users can create subnets at the time of network creation using a wizard located within the Project tab in the dashboard. They can also add new subnets to existing networks. To create a network and subnet, login as the user in the demo project and perform the following steps:

1. Navigate to **Project | Network | Networks** and click on the **Create Network** button:

Name	Subnets Associated	Shared	External	Status	Admin State	Actions
MyUserNetwork		No	No	Active	UP	Edit Network
MyFlatNetwork	MyFlatSubnet 192.168.100.0/24	Yes	No	Active	UP	

2. Clicking on **Create Network** will open a window where you can specify the network and subnet details:

Create Network

Network

Subnet

Subnet Details

Network Name

Create a new network. In addition, a subnet associated with the network can be created in the following steps of this wizard.

☒ Enable Admin State

☒ Create Subnet

Cancel

« Back

Next »

3. Clicking on the **Subnet** tab or the **Next** button will navigate you to a panel where subnet details are defined, including the name, network address, and gateway information:

Create Network

Network

Subnet

Subnet Details

Subnet Name

Network Address

Gateway IP

☐ Disable Gateway

Creates a subnet associated with the network. You need to enter a valid "Network Address" and "Gateway IP". If you did not enter the "Gateway IP", the first value of a network will be assigned by default. If you do not want gateway please check the "Disable Gateway" checkbox. Advanced configuration is available by clicking on the "Subnet Details" tab.

Cancel

« Back

Next »

4. Finally, clicking on **Subnet Details** or **Next** will navigate you to a panel where additional subnet details are defined, including **Allocation pools**, **DNS Name Servers**, and **Host Routes**. Enter the details shown here and click on the blue **Create** button to complete the creation of the **network** and **subnet**:

Create Network

[Network](#) [Subnet](#) [Subnet Details](#)

☒ **Enable DHCP**

Specify additional attributes for the subnet.

Allocation Pools ?

192.168.204.50,192.168.204.99
192.168.204.200,192.168.204.253

DNS Name Servers ?

8.8.8.8
8.8.4.4

Host Routes ?

Cancel« BackCreate

The ability to add additional subnets or delete a network entirely is provided within the menu located under the **Actions** column, as pictured in the following screenshot:

The screenshot shows the 'Networks' page in the OpenStack dashboard. At the top, there is a breadcrumb 'Project / Network / Networks' and a title 'Networks'. Below the title, there are filters for 'Name' and a 'Filter' button, along with '+ Create Network' and 'Delete Networks' buttons. The table displays three networks: 'MyOtherUserNetwork', 'MyUserNetwork', and 'MyFlatNetwork'. Each row has a checkbox, a name, associated subnets, and status columns. The 'Actions' column for 'MyFlatNetwork' is expanded, showing 'Create Subnet' and 'Delete Network' options.

☐	Name	Subnets Associated	Shared	External	Status	Admin State	Actions
☐	MyOtherUserNetwork	MyOtherUserSubnet 192.168.204.0/24	No	No	Active	UP	Edit Network
☐	MyUserNetwork		No	No	Active	UP	Edit Network
☐	MyFlatNetwork	MyFlatSubnet 192.168.100.0/24	Yes	No	Active	UP	Create Subnet Delete Network

Via the Admin tab

Users with the admin role will notice additional functionality available within the dashboard. An Admin panel can be seen on the left-hand side of the dashboard, which allows users with the admin role to manipulate subnets outside of their respective project.

To create a subnet in the dashboard as a cloud administrator, login as the admin user and perform the following steps:

1. Navigate to **Admin | Network | Networks**, and click on the name of the network you wish to add a subnet to:

openstack. Default • admin

Project > Admin / Network / Networks

Admin Overview

Compute > Network > Networks

Project = Filter + Create Network Delete Networks

Displaying 5 items

Project	Network Name	Subnets Associated	DHCP Agents	Shared	External	Status	Admin State	Actions
demo	MyOtherUserNetwork	MyOtherUserSubnet 192.168.204.0/24	1	No	No	Active	UP	Edit Network
demo	MyUserNetwork		0	No	No	Active	UP	Edit Network
admin	MyVLANNetwork2		0	No	No	Active	UP	Edit Network
admin	MyVLANNetwork		0	No	No	Active	UP	Edit Network
admin	MyFlatNetwork	MyFlatSubnet 192.168.100.0/24	1	Yes	No	Active	UP	Edit Network

Displaying 5 items

2. Clicking on **MyVLANNetwork** provides a list of details of the network, including the associated subnets and ports in their respective panels:

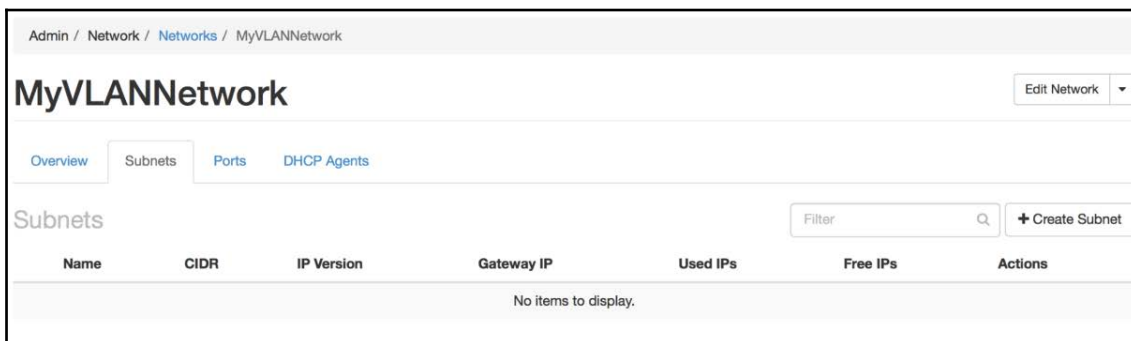
Admin / Network / Networks / MyVLANNetwork

MyVLANNetwork Edit Network

Overview Subnets Ports DHCP Agents

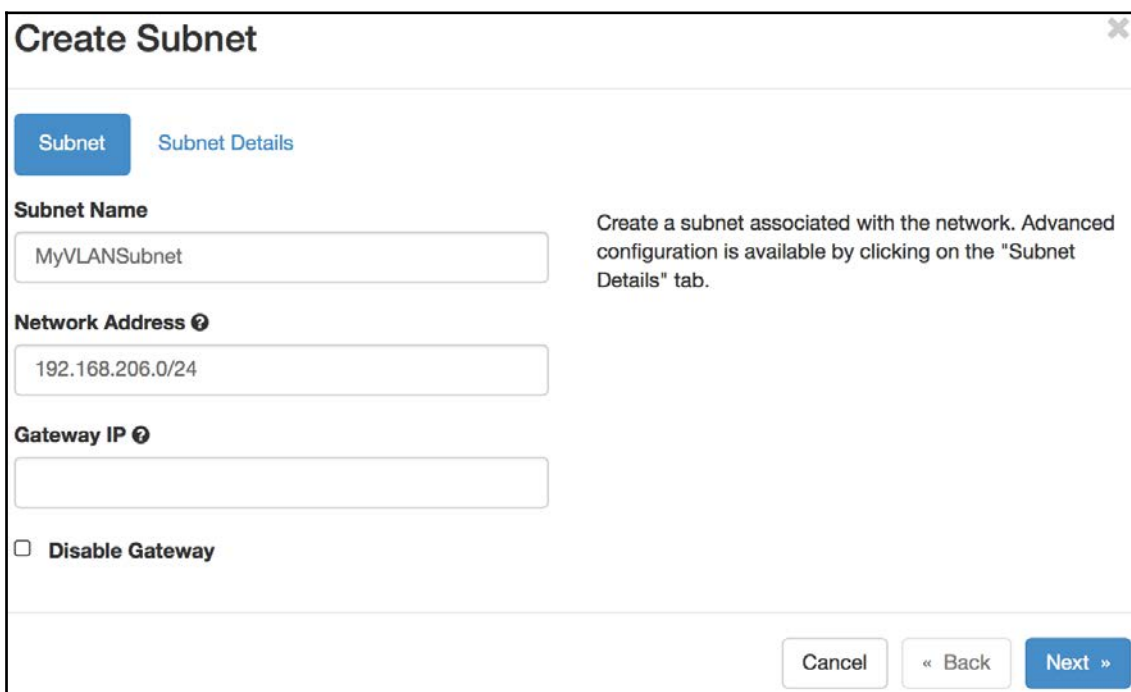
Name	MyVLANNetwork
ID	b2ca3d41-c79c-416a-8e86-0bb01060ddec
Project ID	877f949397ad428cbeaac4e5123bad83
Status	Active
Admin State	UP
Shared	No
External Network	No
MTU	1500
Provider Network	Network Type: vlan Physical Network: physnet1 Segmentation ID: 300

3. To add a subnet to the network, click on the **Subnet** tab and click on the **Create Subnet** button on the right-hand side:



The screenshot shows the 'MyVLANNetwork' page in a web interface. The breadcrumb trail is 'Admin / Network / Networks / MyVLANNetwork'. The page title is 'MyVLANNetwork' with an 'Edit Network' button. Below the title are tabs for 'Overview', 'Subnets', 'Ports', and 'DHCP Agents'. The 'Subnets' tab is active. Below the tabs is a 'Subnets' section with a 'Filter' input, a search icon, and a '+ Create Subnet' button. A table with columns 'Name', 'CIDR', 'IP Version', 'Gateway IP', 'Used IPs', 'Free IPs', and 'Actions' is shown, but it contains 'No items to display.'

4. A wizard will appear that allows you to define the properties of the new subnet:



The screenshot shows the 'Create Subnet' wizard. It has a close button (X) in the top right corner. Below the title are two tabs: 'Subnet' (active) and 'Subnet Details'. The 'Subnet' tab contains the following fields and options:

- Subnet Name**: A text input field containing 'MyVLANSubnet'.
- Network Address ?**: A text input field containing '192.168.206.0/24'.
- Gateway IP ?**: An empty text input field.
- ☐ **Disable Gateway**

To the right of these fields is a text box: 'Create a subnet associated with the network. Advanced configuration is available by clicking on the "Subnet Details" tab.'

At the bottom right are three buttons: 'Cancel', '« Back', and 'Next »'.

5. Clicking **Next** reveals additional configuration options:

Create Subnet

[Subnet](#)[Subnet Details](#)

☒ **Enable DHCP**

Specify additional attributes for the subnet.

Allocation Pools

DNS Name Servers

8.8.8.8
8.8.4.4

Host Routes

Cancel« BackCreate

- Click on the blue **Create** button to complete the creation of the subnet. The **Subnets** panel will be updated accordingly:

Admin / Network / Networks / MyVLANNetwork

MyVLANNetwork

Edit Network ▼

Overview Subnets Ports DHCP Agents

Subnets

Filter + Create Subnet Delete Subnets

Displaying 1 item

☐	Name	CIDR	IP Version	Gateway IP	Used IPs	Free IPs	Actions
☐	MyVLANSubnet	192.168.206.0/24	IPv4	192.168.206.1	1	252	Edit Subnet ▼

Displaying 1 item

Managing subnet pools

Subnet pools were introduced in the Kilo release of OpenStack and allow Neutron to control the assignment of IP address space upon subnet creation. Users can create subnet pools for use within their respective project only, or an administrator can create a subnet pool that is shared between projects. The use of subnet pools helps ensure that there is no IP address overlap between subnets and makes the creation of subnets easier for users. Subnet pools work best for self-service networks that are expected to be connected to Neutron routers and are ideal for cases where address scopes and Neutron's BGP Speaker functionality has been implemented.



BGP stands for Border Gateway Protocol and is a dynamic routing protocol used to route traffic over the internet or within autonomous systems. For more information on BGP Speaker functionality in Neutron, please refer to the upstream documentation available at <https://docs.openstack.org/neutron/pike/admin/config-bgp-dynamic-routing.html>.

Creating a subnet pool

To create a subnet pool, you will need the following information at a minimum:

- Subnet pool prefix in CIDR notation
- Subnet pool name



Additional information can be provided based on your environment and can be determined using the `openstack subnet pool create -h` command.

To demonstrate this command in action, create a subnet pool with the following characteristics:

- Name: `MySubnetPool`
- Subnet Pool CIDR: `172.31.0.0/16`

To create the subnet pool, refer to the following screenshot:

```
root@controller01:~# openstack subnet pool create \
> --pool-prefix 172.31.0.0/16 MySubnetPool
```

Field	Value
address_scope_id	None
created_at	2018-03-07T13:18:22Z
default_prefixlen	8
default_quota	None
description	
id	e49703d8-27f4-4a16-9bf4-91a6cf00fff3
ip_version	4
is_default	False
max_prefixlen	32
min_prefixlen	8
name	MySubnetPool
prefixes	172.31.0.0/16
project_id	9233b6b4f6a54386af63c0a7b8f043c2
revision_number	0
shared	False
tags	
updated_at	2018-03-07T13:18:22Z

The subnet pool `MySubnetPool` is now available for use, but only by the project that created it, as `shared` is `False`.



The default prefix length is 8, which is not ideal and will cause issues if users do not specify a prefix length when creating a subnet. Setting a default prefix length when creating the subnet pool is highly recommended.

Creating a subnet from a pool

To create a subnet from the subnet pool `MySubnetPool`, use the `openstack subnet create` command with the `--subnet-pool` argument. To demonstrate this command in action, create a subnet with the following characteristics:

- Name: `MySubnetFromPool`
- Network: `MyVLANNetwork2`
- Prefix Length: 28

To create the subnet, refer to the following screenshot:

```

root@controller01:~# openstack subnet create \
> --subnet-pool MySubnetPool \
> --prefix-length 28 \
> --network MyVLANNetwork2 \
> MySubnetFromPool

```

Field	Value
allocation_pools	172.31.0.50-172.31.0.62
cidr	172.31.0.48/28
created_at	2018-03-07T15:20:03Z
description	
dns_nameservers	
enable_dhcp	True
gateway_ip	172.31.0.49
host_routes	
id	da20b588-f663-4c8a-a32a-07fd8f336b5a
ip_version	4
ipv6_address_mode	None
ipv6_ra_mode	None
name	MySubnetFromPool
network_id	e01ca743-607c-4a94-9176-b572a46fba84
prefixlen	28
project_id	9233b6b4f6a54386af63c0a7b8f043c2
revision_number	0
segment_id	None
service_types	
subnetpool_id	e49703d8-27f4-4a16-9bf4-91a6cf00fff3
tags	
updated_at	2018-03-07T15:20:03Z
use_default_subnet_pool	None

As shown in the preceding screenshot, the subnet's CIDR was carved from the provided subnet pool, and all other attributes were automatically determined by Neutron.



At the time of writing this book, the following bug may restrict the use of the OpenStack client when providing a prefix length when creating subnets: <https://bugs.launchpad.net/python-openstacksdk/+bug/1754062>. If an error occurs, try the Neutron client instead.

Deleting a subnet pool

To delete a subnet pool, use the `openstack subnet pool delete` command, as shown here:

```
openstack subnet pool delete <subnet-pool> [<subnet-pool> ...]
```

Multiple subnet pools can be deleted simultaneously. Existing subnets that reference a subnet pool must be deleted before the subnet pool can be deleted.

Assigning a default subnet pool

A default subnet pool can be defined that allows users to create subnets without specifying a prefix or a subnet pool. At subnet creation, use both the `--default-prefix-length` and `--default` arguments to make the subnet pool the default pool for a particular network. A subnet pool can also be updated to become the default subnet pool by using the same argument.

The following command demonstrates setting `MySubnetPool` as the default subnet pool:

```
# openstack subnet pool set --default MySubnetPool
```

No output is returned upon successful completion of the preceding command.

The following screenshot demonstrates the creation of a new subnet using the default subnet pool:

```

root@controller01:~# openstack subnet create \
> --use-default-subnet-pool \
> --network MyVLANNetwork2 \
> MySubnetFromDefaultPool

```

Field	Value
allocation_pools	172.31.0.66-172.31.0.78
cidr	172.31.0.64/28
created_at	2018-03-07T15:46:41Z
description	
dns_nameservers	
enable_dhcp	True
gateway_ip	172.31.0.65
host_routes	
id	918d11df-aad8-4234-9f7f-1e927a6b9238
ip_version	4
ipv6_address_mode	None
ipv6_ra_mode	None
name	MySubnetFromDefaultPool
network_id	e01ca743-607c-4a94-9176-b572a46fba84
prefixlen	None
project_id	9233b6b4f6a54386af63c0a7b8f043c2
revision_number	0
segment_id	None
service_types	
subnetpool_id	e49703d8-27f4-4a16-9bf4-91a6cf00fff3
tags	
updated_at	2018-03-07T15:46:41Z
use_default_subnet_pool	True

As you can see, Neutron automatically carved out a /28 subnet from the default subnet pool and set basic attributes without user interaction. The subnet can now be attached to a Neutron router for use by instances.

Managing network ports in OpenStack

A port in Neutron is a logical connection of a virtual network interface to a subnet and network. Ports can be associated with virtual machine instances, DHCP servers, routers, firewalls, load balancers, and more. Ports can even be created simply to reserve IP addresses from a subnet. Neutron stores port relationships in the Neutron database and uses that information to build switching connections at the physical or virtual switch layer through the networking plugin and agent.

The primary commands associated with port management are listed in the following table:

Port Commands	Description
<code>port create</code>	Creates a new port
<code>port delete</code>	Deletes a port(s)
<code>port list</code>	Lists ports
<code>port set</code>	Sets port properties
<code>port show</code>	Displays port details
<code>port unset</code>	Unsets port properties

When a port is created in OpenStack and associated with an instance or other virtual network device, it is bound to a Neutron agent on the respective node hosting the instance or device. Using details provided by the port, OpenStack services may construct a virtual machine interface (vif) or virtual ethernet interface (veth) on the host for use with a virtual machine, network namespace, or more depending on the application.

To retrieve a list of all Neutron ports, use the `openstack port list` command, as shown in the following screenshot:

```
root@controller01:~# openstack port list
```

ID	Name	MAC Address	Fixed IP Addresses	Status
1f48a8f5-<snip>-91fd82cce7c1		fa:16:3e:d0:95:55	ip_address='192.168.100.2', subnet_id='63eaa79b-8129-4fc4-a783-cc7ba917d462'	ACTIVE
5a962785-<snip>-377415cbfdcf		fa:16:3e:85:64:17	ip_address='192.168.206.2', subnet_id='9cb3e07c-cc91-44c4-b8be-082043720db1'	ACTIVE
603d3937-<snip>-ca2af1dea42e		fa:16:3e:be:12:f4	ip_address='192.168.204.200', subnet_id='6cffffc42-0366-447f-89ef-5bad385322ff'	ACTIVE

Users with the admin role will see all ports known to Neutron, while ordinary users will only see ports associated with their respective project.

If tags are utilized, the results may be filtered using the following parameters:

- `--tags <tag>`: Lists ports which have all given tag(s)
- `--any-tags <tag>`: Lists ports which have any given tag(s)

- `--not-tags <tag>`: Excludes ports which have all given tag(s)
- `--not-any-tags <tag>`: Excludes ports which have any given tag(s)

Use the `openstack port show` command to see the details of a particular port:

```
root@controller01:~# openstack port show 1f48a8f5-10d5-4985-9ff2-91fd82cce7c1
```

Field	Value
admin_state_up	UP
allowed_address_pairs	
binding_host_id	book-controller01
binding_profile	
binding_vif_details	port_filter='True'
binding_vif_type	bridge
binding_vnic_type	normal
created_at	2018-01-15T01:22:51Z
data_plane_status	None
description	
device_id	dhcpa6f16a08-9242-5c9a-85e6-61f4c10cded7-d51943ef-8061-4bdb-b684-8a7d2b7ce73b
device_owner	network:dhcp
dns_assignment	None
dns_name	None
extra_dhcp_opts	
fixed_ips	ip_address='192.168.100.2', subnet_id='63eaa79b-8129-4fc4-a783-cc7ba917d462'
id	1f48a8f5-10d5-4985-9ff2-91fd82cce7c1
ip_address	None
mac_address	fa:16:3e:d0:95:55
name	
network_id	d51943ef-8061-4bdb-b684-8a7d2b7ce73b
option_name	None
option_value	None
port_security_enabled	False
project_id	877f949397ad428cbeaac4e5123bad83
qos_policy_id	None
revision_number	5
security_group_ids	
status	ACTIVE
subnet_id	None
tags	
trunk_details	None
updated_at	2018-01-15T01:23:00Z

The port pictured in the preceding screenshot is owned by an interface that's used within a DHCP namespace, as represented by a `device_owner` of `network:dhcp`. The `network_id` field reveals the network to be `d51943ef-8061-4bdb-b684-8a7d2b7ce73b`, which is the `MyFlatNetwork` network that we created earlier on in this chapter.

We can use the `ip netns exec` command to execute commands within the DHCP namespace. If you recall, the DHCP namespace can be identified with the prefix `qdhcp-` and a suffix of the respective network ID. On the `controller01` node, run the `ip addr` command within the namespace to list its interfaces and their details:

```
root@controller01:~# ip netns exec qdhcp-d51943ef-8061-4bdb-b684-8a7d2b7ce73b ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
2: ns-1f48a8f5-10@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:d0:95:55 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.100.2/24 brd 192.168.100.255 scope global ns-1f48a8f5-10
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fed0:9555/64 scope link
        valid_lft forever preferred_lft forever
```

In the DHCP namespace, the interface's MAC address corresponds to the port's `mac_address` field, while the name of the interface corresponds to the first 10 characters of the Neutron port UUID:

```
root@controller01:~# openstack port show 1f48a8f5-10d5-4985-9ff2-91fd82cce7c1 -c id -c mac_address
```

Field	Value
id	1f48a8f5-10d5-4985-9ff2-91fd82cce7c1
mac_address	fa:16:3e:d0:95:55

Creating a port

By creating a Neutron port manually, users have the ability to specify a particular fixed IP address, MAC address, security group, and more.

To create a port, use the `openstack port create` command, as shown here:

```
openstack port create
--network <network> [--description <description>]
[--device <device-id>]
[--mac-address <mac-address>]
[--device-owner <device-owner>]
[--vnic-type <vnic-type>] [--host <host-id>]
[--dns-name dns-name]
[--fixed-ip subnet=<subnet>,ip-address=<ip-address>]
```

```
[--binding-profile <binding-profile>]
[--enable | --disable] [--project <project>]
[--project-domain <project-domain>]
[--security-group <security-group> | --no-security-group]
[--qos-policy <qos-policy>]
[--enable-port-security | --disable-port-security]
[--allowed-address ip-address=<ip>[,mac-address=<mac>]]
[--tag <tag> | --no-tag]
<name>
```

Once created, the port can then be associated with a virtual machine instance or other virtual network device. It can also be used to reserve an IP address in a subnet. In the next chapter, we will look at creating ports and associating them with instances in multiple ways.

Summary

This chapter laid the foundation for creating networks and subnets that can be utilized by instances and other virtual and physical devices. Both the Horizon dashboard and the OpenStack command-line client can be used to manage networks, subnets, and ports, though the latter is recommended for most administrative tasks.

For more information on network, subnet, and port attributes, as well as for guidance on how to use the Neutron API, refer to the OpenStack wiki at <https://developer.openstack.org/api-ref/network/v2/index.html>.

In the next chapter, we will learn the basics of creating instances and attaching them to networks. Some of the networks built in this chapter will be used to demonstrate the creation of ports that can be attached to instances with the end goal of end-to-end connectivity.

7

Attaching Instances to Networks

In `chapter 6`, *Building Networks with Neutron*, we created multiple networks that can be utilized by projects in the cloud. In this chapter, we will create instances that reside in those networks.

I will guide you through the following tasks:

- Creating and attaching instances to networks
- Adding additional interfaces to instances
- Demonstrating DHCP and metadata services

Attaching instances to networks

Using the OpenStack command-line client, instances can be attached to networks in a couple of ways. When an instance is first created, it can be attached to one or more networks using the `openstack server create` command. Running instances can be attached to additional networks using the `openstack server add port` command. Both methods are explained in the following sections.



If you're following along, the networks we created in the previous chapters have all been destroyed and recreated with similar names and segmentation IDs, but with new ID numbers.

Attaching instances to networks at creation

Instances are created using the `openstack server create` command, as you can see here:

```
openstack server create
(--image <image> | --volume <volume>) --flavor <flavor>
[--security-group <security-group>]
[--key-name <key-name>]
[--property <key=value>]
[--file <dest-filename=source-filename>]
[--user-data <user-data>]
[--availability-zone <zone-name>]
[--block-device-mapping <dev-name=mapping>]
[--nic <net-id=net-uuid,v4-fixed-ip=ip-addr,v6-fixed-ip=ip-addr,port-
id=port-uuid,auto,none>]
[--network <network>] [--port <port>]
[--hint <key=value>]
[--config-drive <config-drive-volume> | True]
[--min <count>] [--max <count>] [--wait]
<server-name>
```

Nova attaches instances to virtual bridges and switches on the `compute` node via their virtual interfaces, or VIFs. Each VIF has a corresponding Neutron port in the database.

When using the Open vSwitch mechanism driver and Open vSwitch firewall driver, each VIF plugs into the integration bridge on the respective `compute` node hosting the instance. The virtual switch port is configured with a local VLAN ID that corresponds to the network associated with the Neutron port and VIF. When the `iptables_hybrid` firewall driver is used, the VIF is connected to a Linux bridge where `iptables` rules are applied.

When using the Linux bridge mechanism driver, each VIF connects to a Linux bridge that corresponds to the associated network. Every network has a corresponding bridge that is used to segregate traffic at Layer 2.

For a refresher on these concepts, refer to *Chapter 4, Virtual Switching Infrastructure Using Linux Bridges*, and *Chapter 5, Building a Virtual Switching Infrastructure Using Open vSwitch*.

Specifying a network

The `openstack server create` command provides a `--nic` argument that specifies the network or port to be attached to the instance.

Users can specify a network identified by the network's ID by using the `net-id` key:

```
--nic net-id=<Network ID>
```

In the preceding example, Nova interfaces with the Neutron API to create a port using the network ID provided by the user. Neutron then returns details of the port back to Nova for use by the instance. Users can request a specific unused IPv4 or IPv6 address using the `v4-fixed-ip` and `v6-fixed-ip` keys, respectively, as shown here:

```
--nic net-id=<Network ID>,v4-fixed-ip=<ipv4 address>  
--nic net-id=<Network ID>,v6-fixed-ip=<ipv6 address>
```

Specifying a port

Alternatively, users can specify a port that's been identified by the port's ID using the `port-id` key, as shown here:

```
--nic port-id=<Port ID>
```

In this example, Neutron associates the existing port with the instance and sets the port's `device_id` attribute accordingly. A port can later be detached from an instance and associated with a new instance using this method. Possible options include `auto`, `none`, or the ID of an existing port. The default is `auto`.

Attaching multiple interfaces

By passing the `--nic` argument multiple times, it is possible to attach multiple interfaces to an instance. The interfaces within the instance may then be enumerated as `eth0`, `eth1`, `eth2`, and so on, depending on the operating system.

Attaching multiple network interfaces to an instance is referred to as **multihoming**. When an instance is multihomed, neither Neutron nor the instance itself is aware of which network takes precedence over another. When attached networks and subnets have their own respective gateway addresses set, an instance's routing table can be populated with multiple default routes. This scenario can wreak havoc on the connectivity and routing behavior of an instance. This configuration is useful when connecting instances to multiple networks directly, however, care should be taken to avoid network issues in this type of design.



Para-virtualized devices, including network and storage devices that use the virtio drivers, are PCI devices. Virtual machine instances under KVM are currently limited to 32 total PCI devices. Some PCI devices are critical for operation, including the host bridge, the ISA/USB bridge, the graphics card, and the memory balloon device, leaving up to 28 PCI slots available for use. Every para-virtualized network or block device uses one slot. This means that users may have issues attempting to connect upwards of 20-25 networks to an instance depending on the characteristics of that instance.

The following `openstack server create` command demonstrates the basic procedure of connecting an instance to multiple networks when creating the instance:

```
openstack server create --flavor FLAVOR --image IMAGE \  
--nic net-id=NETWORK1 \  
--nic net-id=NETWORK2 \  
--nic net-id=NETWORK3 \  
<SERVER-NAME>
```

Inside the instance, the first attached NIC corresponds to `NETWORK1`, the second NIC corresponds to `NETWORK2`, and so on. For many cloud-ready images, a single interface within the instance is brought online automatically using DHCP. Modification of the network interface file(s) or use of the `dhclient` command within the instance may be required to activate and configure additional network interfaces once the instance is active.

Attaching network interfaces to running instances

Using the `openstack server add port` or `openstack server add fixed ip` commands, you can attach an existing or new port to running instances.

The `openstack server add port` command can be used as follows:

```
openstack server add port <server> <port>
```

The port argument specifies the port to be attached to the given server. The port must be one that is not currently associated with any other instance or resource. Otherwise, the operation will fail.

The `openstack server add fixed ip` command can be used as follows:

```
openstack server add fixed ip
[--fixed-ip-address <ip-address>]
<server> <network>
```

The network argument specifies the network to be attached to the given server. A new port that has a unique MAC address and an IP from the specified network will be created automatically.

The `--fixed-ip-address` argument can be used to specify a particular IP address in the given network rather than relying on an automatic assignment from Neutron.



While additional network interfaces may be added to running instances using hot-plug technology, the interfaces themselves may need to be configured within the operating system before they can be used. You may use the `dhclient` command to configure the newly-connected interface using DHCP or configure the interface file manually.

Detaching network interfaces

To detach an interface from an instance, use the `openstack server remove port` or `openstack server remove fixed ip` commands, as shown here:

```
openstack server remove port <server> <port>
openstack server remove fixed ip <server> <ip-address>
```

Interfaces detached from instances are removed completely from the Neutron port database.



Take caution when removing interfaces from running instances, as it may cause unexpected behavior within the instance.

Exploring how instances get their addresses

When a network is created and DHCP is enabled on a subnet within the network, the network is scheduled to one or more DHCP agents in the environment. In most environments, DHCP agents are configured on `controllers` or dedicated `network` nodes. In more advanced environments, such as those utilizing network segments and leaf/spine topologies, DHCP agents may be needed on `compute` nodes.

A DHCP agent is responsible for creating a local network namespace that corresponds to each network that has been scheduled to that agent. An IP address is then configured on a virtual interface inside the namespace, along with a `dnsmasq` process that listens for DHCP requests on the network. If a `dnsmasq` process already exists for the network and a new subnet is added, the existing process is updated to support the additional subnet.



When DHCP is not enabled on a subnet, a `dnsmasq` process is not spawned. An IP address is still associated with the Neutron port that corresponds to the interface within the instance, however. Without DHCP services, it is up to the user to manually configure the IP address on the interface within the guest operating system through a console connection.

Most instances rely on DHCP to obtain their associated IP address. DHCP follows the following stages:

- A DHCP client sends a `DHCPDISCOVER` broadcast packet that requests IP information from a DHCP server.
- A DHCP server responds to the request with a `DHCPOFFER` packet. The packet contains the MAC address of the instance that makes the request, the IP address, the subnet mask, lease duration, and the IP address of the DHCP server. A Neutron network can be scheduled to multiple DHCP agents simultaneously, and each DHCP server may respond with a `DHCPOFFER` packet. However, the client will only accept the first one.
- In response to the offer, the DHCP client sends a `DHCPREQUEST` packet back to the DHCP server, requesting the offered address.
- In response to the request, the DHCP server will issue a `DHCPACK` packet or acknowledgement packet to the instance. At this point, the IP configuration is complete. The DHCP server sends other DHCP options such as name servers, routes, and so on to the instance.

Network namespaces associated with DHCP servers are prefixed with `qdhcp`, followed by the entire network ID. DHCP namespaces will only reside on hosts running the `neutron-dhcp-agent` service. Even then, the network must be scheduled to the DHCP agent for the namespace to be created on that host. In this example, the DHCP agent runs on the `controller01` node.

To view a list of namespaces on the `controller01` node, use the `ip netns list` command that's shown here:

```
root@controller01:~# ip netns list
qdhcp-03327707-c369-4bd7-bd71-a42d9bcf49b8 (id: 1)
qdhcp-7745a4a9-68a4-444e-a5ff-f9439e3aac6c (id: 0)
```

The two namespaces listed in the output directly correspond to two networks for which a subnet exists and DHCP is enabled:

```
root@controller01:~# openstack network list
```

ID	Name	Subnets
03327707-c369-4bd7-bd71-a42d9bcf49b8	MyVLANNetwork	02013907-1a5f-493f-8ece-ee1ee9e44afb
7745a4a9-68a4-444e-a5ff-f9439e3aac6c	MyFlatNetwork	c43179e1-8bbb-48d1-a486-63d048e78367

An interface exists within the `qdhcp` namespace for the network **MyVLANNetwork**, which is used to connect the namespace to the virtual network infrastructure:

```
root@controller01:~# ip netns exec qdhcp-03327707-c369-4bd7-bd71-a42d9bcf49b8 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
2: ns-6c15d7b8-87@if74: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:ad:59:b9 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.206.2/24 brd 192.168.206.255 scope global ns-6c15d7b8-87
        valid_lft forever preferred_lft forever
    inet 169.254.169.254/16 brd 169.254.255.255 scope global ns-6c15d7b8-87
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fead:59b9/64 scope link
        valid_lft forever preferred_lft forever
```

The interface `ns-6c15d7b8-87` within the namespace is one end of a `veth` interface. The IP address assigned to the `ns-6c15d7b8-87` interface, `192.168.206.2/24`, has been automatically assigned by Neutron and was procured from the subnet's allocation pool.

When using the Linux bridge driver, the other end of the interface, known as the peer, is connected to a bridge that corresponds to the network and is represented by the `tap6c15d7b8-87` interface, as shown in the following screenshot:

```
root@controller01:~# brctl show
```

bridge name	bridge id	STP enabled	interfaces
brq03327707-c3	8000.000c291124cb	no	eth2.300 tap6c15d7b8-87
brq7745a4a9-68	8000.000c291124cb	no	eth2 tapd1848f67-2e

In the preceding screenshot, the bridge labeled `brq7745a4a9-68` corresponds to the network `MyFlatNetwork`, as evidenced by the untagged interface `eth2` connected to the bridge. The interface `tapd1848f67-2e` is the other end of the veth interface which is connected to the DHCP namespace for the network `MyFlatNetwork`.

Watching the DHCP lease cycle

In this example, an instance will be created with the following characteristics:

- Name: `TestInstance1`
- Flavor: `tiny`
- Image: `cirros-0.4.0`
- Network: `MyVLANNetwork`
- Compute Node: `compute01`

The command to create the instance is as follows:

```
openstack server create \
--image cirros-0.4.0 \
--flavor tiny \
--nic net-id=MyVLANNetwork \
--availability-zone nova:compute01 \
TestInstance1
```



The tiny flavor may not exist in your environment, but can be created and defined with 1 vCPU, 1 MB RAM, and 1 GB disk.

To observe the instance requesting a DHCP lease, start a packet capture within the DHCP network namespace that corresponds to the instance's network using the following command:

```
ip netns exec <namespace> tcpdump -i any -ne port 67 or port 68
```

As an instance starts up, it will send broadcast packets that will be answered by the DHCP server within the namespace:

```
root@controller01:~# ip netns exec qdhcp-03327707-c369-4bd7-bd71-a42d9bcf49b8 tcpdump -i any -ne port 67 or port 68
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 262144 bytes
12:04:40.466828 B fa:16:3e:27:c5:d2 ethertype IPv4 (0x0800), length 344: 0.0.0.0.68 > 255.255.255.255.67: BOOTP/DHCP, Request from fa:16:3e:27:c5:d2, length 300
12:04:40.467624 Out fa:16:3e:ad:59:b9 ethertype IPv4 (0x0800), length 377: 192.168.206.2.67 > 192.168.206.12.68: BOOTP/DHCP, Reply, length 333
12:04:40.473332 B fa:16:3e:27:c5:d2 ethertype IPv4 (0x0800), length 346: 0.0.0.0.68 > 255.255.255.255.67: BOOTP/DHCP, Request from fa:16:3e:27:c5:d2, length 302
12:04:40.558296 Out fa:16:3e:ad:59:b9 ethertype IPv4 (0x0800), length 398: 192.168.206.2.67 > 192.168.206.12.68: BOOTP/DHCP, Reply, length 354
```

In the preceding screenshot, all four stages of the DHCP lease cycle can be observed. A similar output can be observed by performing the capture on the tap interface of the instance on the compute node:

```
root@compute01:~# tcpdump -i tap0e3c29d0-75 -ne port 67 or port 68
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on tap0e3c29d0-75, link-type EN10MB (Ethernet), capture size 262144 bytes
12:04:40.466317 fa:16:3e:27:c5:d2 > ff:ff:ff:ff:ff:ff, ethertype IPv4 (0x0800), length 342: 0.0.0.0.68 > 255.255.255.255.67: BOOTP/DHCP, Request from fa:16:3e:27:c5:d2, length 300
12:04:40.468062 fa:16:3e:ad:59:b9 > fa:16:3e:27:c5:d2, ethertype IPv4 (0x0800), length 375: 192.168.206.2.67 > 192.168.206.12.68: BOOTP/DHCP, Reply, length 333
12:04:40.473176 fa:16:3e:27:c5:d2 > ff:ff:ff:ff:ff:ff, ethertype IPv4 (0x0800), length 344: 0.0.0.0.68 > 255.255.255.255.67: BOOTP/DHCP, Request from fa:16:3e:27:c5:d2, length 302
12:04:40.558776 fa:16:3e:ad:59:b9 > fa:16:3e:27:c5:d2, ethertype IPv4 (0x0800), length 396: 192.168.206.2.67 > 192.168.206.12.68: BOOTP/DHCP, Reply, length 354
```

Using the `dhcpcd` utility, we can observe more verbose output of the DHCP lease cycle. To install `dhcpcd`, issue the following command on all nodes:

```
# apt install dhcpcd
```

Within the network namespace hosting the DHCP server for the `MyVLANNetwork` network, run `dhcpcd`, as shown here:

```
root@controller01:~# ip netns exec qdhcp-03327707-c369-4bd7-bd71-a42d9bcf49b8 dhcpcd -i ns-6c15d7b8-87
```

As the client makes its initial DHCP request, you will see a `DHCPDISCOVER` broadcast packet:

```
TIME: 2018-03-16 12:08:28.250
  IP: 0.0.0.0 (fa:16:3e:27:c5:d2) > 255.255.255.255 (ff:ff:ff:ff:ff:ff)
  OP: 1 (BOOTPREQUEST)
HTYPE: 1 (Ethernet)
HLEN: 6
HOPS: 0
XID: 1af16110
SECS: 0
FLAGS: 0
CIADDR: 0.0.0.0
YIADDR: 0.0.0.0
SIADDR: 0.0.0.0
GIADDR: 0.0.0.0
CHADDR: fa:16:3e:27:c5:d2:00:00:00:00:00:00:00:00:00:00
SNAME: .
FNAME: .
OPTION: 53 ( 1) DHCP message type      1 (DHCPDISCOVER)
OPTION: 61 ( 7) Client-identifier      01:fa:16:3e:27:c5:d2
OPTION: 57 ( 2) Maximum DHCP message size 576
OPTION: 55 ( 9) Parameter Request List  1 (Subnet mask)
          3 (Routers)
          6 (DNS server)
          12 (Host name)
          15 (Domainname)
          26 (Interface MTU)
          28 (Broadcast address)
          42 (NTP servers)
          121 (Classless Static Route)

OPTION: 60 (12) Vendor class identifier  udhcp 1.23.2
OPTION: 12 (13) Host name                testinstance1
-----
```

Then, the DHCP server will send a DHCP`OFFER` unicast packet directly to the instance:

```

TIME: 2018-03-16 12:08:28.251
IP: 192.168.206.2 (fa:16:3e:ad:59:b9) > 192.168.206.12 (fa:16:3e:27:c5:d2)
OP: 2 (BOOTPREPLY)
HTYPE: 1 (Ethernet)
HLEN: 6
HOPS: 0
XID: 1af16110
SECS: 0
FLAGS: 0
CIADDR: 0.0.0.0
YIADDR: 192.168.206.12
SIADDR: 192.168.206.2
GIADDR: 0.0.0.0
CHADDR: fa:16:3e:27:c5:d2:00:00:00:00:00:00:00:00:00:00
SNAME: .
FNAME: .
OPTION: 53 ( 1) DHCP message type      2 (DHCPOFFER)
OPTION: 54 ( 4) Server identifier      192.168.206.2
OPTION: 51 ( 4) IP address leasetime   86400 (24h)
OPTION: 58 ( 4) T1                     43200 (12h)
OPTION: 59 ( 4) T2                     75600 (21h)
OPTION:  1 ( 4) Subnet mask             255.255.255.0
OPTION: 28 ( 4) Broadcast address      192.168.206.255
OPTION:  6 ( 4) DNS server              192.168.206.2
OPTION: 15 (19) Domainname              learningneutron.com
OPTION:  3 ( 4) Routers                 192.168.206.1
OPTION: 121 (14) Classless Static Route 20a9fea9fec0a8ce .....
                                0200c0a8ce01 .....
OPTION: 26 ( 2) Interface MTU          1500

```

Next, the client will send a `DHCPREQUEST` broadcast packet:

```

TIME: 2018-03-16 12:08:28.251
  IP: 0.0.0.0 (fa:16:3e:27:c5:d2) > 255.255.255.255 (ff:ff:ff:ff:ff:ff)
  OP: 1 (BOOTPREQUEST)
HTYPE: 1 (Ethernet)
HLEN: 6
HOPS: 0
  XID: 1af16110
SECS: 0
FLAGS: 0
CIADDR: 0.0.0.0
YIADDR: 0.0.0.0
SIADDR: 0.0.0.0
GIADDR: 0.0.0.0
CHADDR: fa:16:3e:27:c5:d2:00:00:00:00:00:00:00:00:00:00:00
SNAME: .
FNAME: .
OPTION: 53 ( 1) DHCP message type      3 (DHCPREQUEST)
OPTION: 61 ( 7) Client-identifier      01:fa:16:3e:27:c5:d2
OPTION: 50 ( 4) Request IP address     192.168.206.12
OPTION: 54 ( 4) Server identifier      192.168.206.2
OPTION: 57 ( 2) Maximum DHCP message size 576
OPTION: 55 ( 9) Parameter Request List  1 (Subnet mask)
                                           3 (Routers)
                                           6 (DNS server)
                                           12 (Host name)
                                           15 (Domainname)
                                           26 (Interface MTU)
                                           28 (Broadcast address)
                                           42 (NTP servers)
                                           121 (Classless Static Route)

OPTION: 60 (12) Vendor class identifier  udhcp 1.23.2
OPTION: 12 (13) Host name               testinstance1
-----

```


Lastly, the DHCP server will acknowledge the request with a DHCPACK unicast packet:

```

TIME: 2018-03-16 12:08:28.251
IP: 192.168.206.2 (fa:16:3e:ad:59:b9) > 192.168.206.12 (fa:16:3e:27:c5:d2)
OP: 2 (BOOTPREPLY)
HTYPE: 1 (Ethernet)
HLEN: 6
HOPS: 0
XID: 1af16110
SECS: 0
FLAGS: 0
CIADDR: 0.0.0.0
YIADDR: 192.168.206.12
SIADDR: 192.168.206.2
GIADDR: 0.0.0.0
CHADDR: fa:16:3e:27:c5:d2:00:00:00:00:00:00:00:00:00:00
SNAME: .
FNAME: .
OPTION: 53 ( 1) DHCP message type      5 (DHCPACK)
OPTION: 54 ( 4) Server identifier       192.168.206.2
OPTION: 51 ( 4) IP address leasetime    86400 (24h)
OPTION: 58 ( 4) T1                      43200 (12h)
OPTION: 59 ( 4) T2                      75600 (21h)
OPTION: 1 ( 4) Subnet mask              255.255.255.0
OPTION: 28 ( 4) Broadcast address       192.168.206.255
OPTION: 6 ( 4) DNS server               192.168.206.2
OPTION: 15 (19) Domainname              learningneutron.com
OPTION: 12 (19) Host name                host-192-168-206-12
OPTION: 3 ( 4) Routers                  192.168.206.1
OPTION: 121 (14) Classless Static Route 20a9fea9fec0a8ce .....
                                0200c0a8ce01 .....
OPTION: 26 ( 2) Interface MTU           1500

```

Troubleshooting DHCP

If an instance is unable to procure its address from DHCP, it may be helpful to run a packet capture from various points in the network to see where the request or reply is failing.

Using `tcpdump` or `dhcpcdump`, one can capture DHCP request and response packets on UDP ports 67 and 68 from the following locations:

- Within the DHCP namespace
- On the physical interface of the `network` and/or `compute` nodes

- On the tap interface of the instance
- Within the guest operating system via the console

Further investigation may be required once the node or interface responsible for dropping traffic has been identified.

Exploring how instances retrieve their metadata

In *Chapter 3, Installing Neutron*, we briefly covered the process of instances accessing metadata over the network: either through a proxy in the router namespace or the DHCP namespace. The latter is described in the following section.

The DHCP namespace

Instances access metadata at `http://169.254.169.254`, followed by a URI that corresponds to the version of metadata, which is usually `/latest`. When an instance is connected to a network that does not utilize a Neutron router as the gateway, the instance must learn how to reach the metadata service. This can be accomplished in a few different ways, including the following:

- Setting a route manually on the instance
- Allowing DHCP to provide a route

When `enable_isolated_metadata` is set to `True` in the DHCP configuration file at `/etc/neutron/dhcp_agent.ini`, each DHCP namespace provides a proxy to the metadata service running on the controller node(s). The proxy service listens directly on port 80, as shown in the following screenshot:

```
root@controller01:~# ip netns exec qdhcp-03327707-c369-4bd7-bd71-a42d9bcf49b8 netstat -ntlp
Active Internet connections (only servers)

```

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State	PID/Program name
tcp	0	0	0.0.0.0:80	0.0.0.0:*	LISTEN	3555/haproxy
tcp	0	0	192.168.206.2:53	0.0.0.0:*	LISTEN	3793/dnsmasq
tcp	0	0	169.254.169.254:53	0.0.0.0:*	LISTEN	3793/dnsmasq
tcp6	0	0	fe80::f816:3eff:fead:53	:::*	LISTEN	3793/dnsmasq

Using the `ps` command within the namespace, you can see the process associated with this listener is the Neutron metadata proxy:

```
root@controller01:~# ip netns exec qdhcp-03327707-c369-4bd7-bd71-a42d9bcf49b8 ps 3555
PID TTY      STAT   TIME COMMAND
3555 ?        Ss     0:01 haproxy -f /var/lib/neutron/ns-metadata-proxy/03327707-c369-4bd7-bd71-a42d9bcf49b8.conf
```

Adding a manual route to 169.254.169.254

Before an instance can reach the metadata service in the DHCP namespace at 169.254.169.254, a route must be configured to use the DHCP namespace interface as the next hop rather than at the default gateway of the instance.

Observe the IP addresses within the following DHCP namespace:

```
root@controller01:~# ip netns exec qdhcp-03327707-c369-4bd7-bd71-a42d9bcf49b8 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
2: ns-6c15d7b8-87@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:ad:59:b9 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.206.2/24 brd 192.168.206.255 scope global ns-6c15d7b8-87
        valid_lft forever preferred_lft forever
    inet 169.254.169.254/16 brd 169.254.255.255 scope global ns-6c15d7b8-87
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fead:59b9/64 scope link
        valid_lft forever preferred_lft forever
```

169.254.169.254/16 has been automatically configured as a secondary address on the interface inside the namespace. To reach 169.254.169.254 from an instance in the 192.168.206.0/24 network, the following `ip route` command can be used within the guest instance that sets 192.168.206.2 as the next hop:

```
# ip route add 169.254.169.254/32 via 192.168.206.2
```

While this method works, the process of adding a route to each instance does not scale well, especially when multiple DHCP agents exist in the environment. A single network can be scheduled to multiple agents that, in turn, have their own respective namespaces and IP addresses in the same subnet. Users will need prior knowledge of the IP to use in their route statement, and the address is subject to change. Allowing DHCP to inject the route automatically is the recommended method that will be discussed in the next section.

Using DHCP to inject the route

When `enable_isolated_metadata` is set to `True` and the gateway for a subnet is not set or is not a Neutron router, the DHCP service is capable of injecting a route to the metadata service via the classless-static-route DHCP option, also known as option 121.



Possible DHCP options, including those leveraged by Neutron for various tasks, can be found on the IANA website at the following URL: <https://www.iana.org/assignments/bootp-dhcp-parameters/bootp-dhcp-parameters.xhtml>.

Once an instance connected to a subnet with the mentioned characteristics has been created, observe the following routes passed to the instance via DHCP:

```
$ ip r
169.254.169.254 via 192.168.206.2 dev eth0
192.168.206.0/24 dev eth0 src 192.168.206.12
```

The next hop address for the metadata route is the IP address of the DHCP server that responded to the initial DHCP request from the client. If there were multiple DHCP agents in the environment and the same network was scheduled to all of them, it is possible that the next hop address would vary between instances, as any of the DHCP servers could have responded to the request.

Summary

In this chapter, I demonstrated how to attach instances to networks and mapped out the process of an instance obtaining its IP address from an OpenStack-managed DHCP server. I also showed you how an instance reaches out to the metadata service when connected to a VLAN provider network. The same examples in this chapter can be applied to any recent OpenStack cloud and many different network topologies.

For additional details on deployment scenarios based on provider networks, refer to the following upstream documentation for the Pike release of OpenStack at the following locations:

Open vSwitch:

<https://docs.openstack.org/neutron/pike/admin/deploy-ovs-provider.html>

Linux bridge:

<https://docs.openstack.org/neutron/pike/admin/deploy-lb-provider.html>

In the next chapter, we will learn how to leverage Neutron security group functionality to provide network-level security to instances.

8

Managing Security Groups

OpenStack Networking provides two different APIs that can be used to implement network traffic filters. The first API, known as the Security Group API, provides basic traffic filtering at an instance port level. Security group rules are implemented within iptables or as Open vSwitch flow rules on a compute node and filter traffic entering or leaving Neutron ports attached to instances. The second API, known as the Firewall as a Service API (FWaaS), also implements filtering rules at the port level, but extends filtering capabilities to router ports and other ports besides those belonging to traditional instances.

In this chapter, we will focus on security groups and cover some fundamental security features of Neutron, including the following:

- A brief introduction to iptables
- Creating and managing security groups
- Demonstrating traffic flow through iptables
- Configuring port security
- Managing allowed address pairs



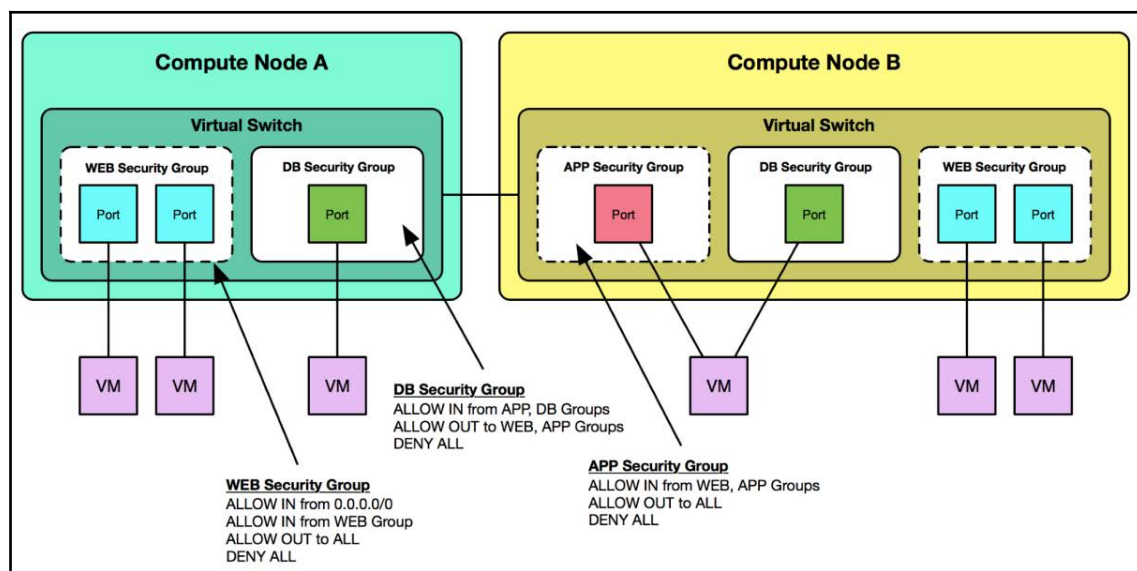
Due to its experimental status as of the Pike and Queens releases of OpenStack, the `openvswitch` firewall driver will not be discussed in this book.

Security groups in OpenStack

A **security group** is a collection of network access rules known as **security group rules** that limit the types of traffic an instance can send or receive. When an iptables-based driver is used, security group rules are converted to iptables rules that are applied on the `compute` nodes hosting the instances. Each OpenStack project is provided with a default security group that can be modified by users within the project. OpenStack Networking provides an API for creating, modifying, applying, and deleting security groups and rules.

There are multiple ways to apply security groups to instances. For example, one or more instances, usually of similar functionality or role, can be placed in a security group. Security group rules can reference IPv4 and IPv6 hosts and networks as well as other security groups. Referencing a particular security group in a rule, rather than a particular host or network, frees the user from having to specify individual network addresses. Neutron will construct the filtering rules applied to the host automatically based on information in the database.

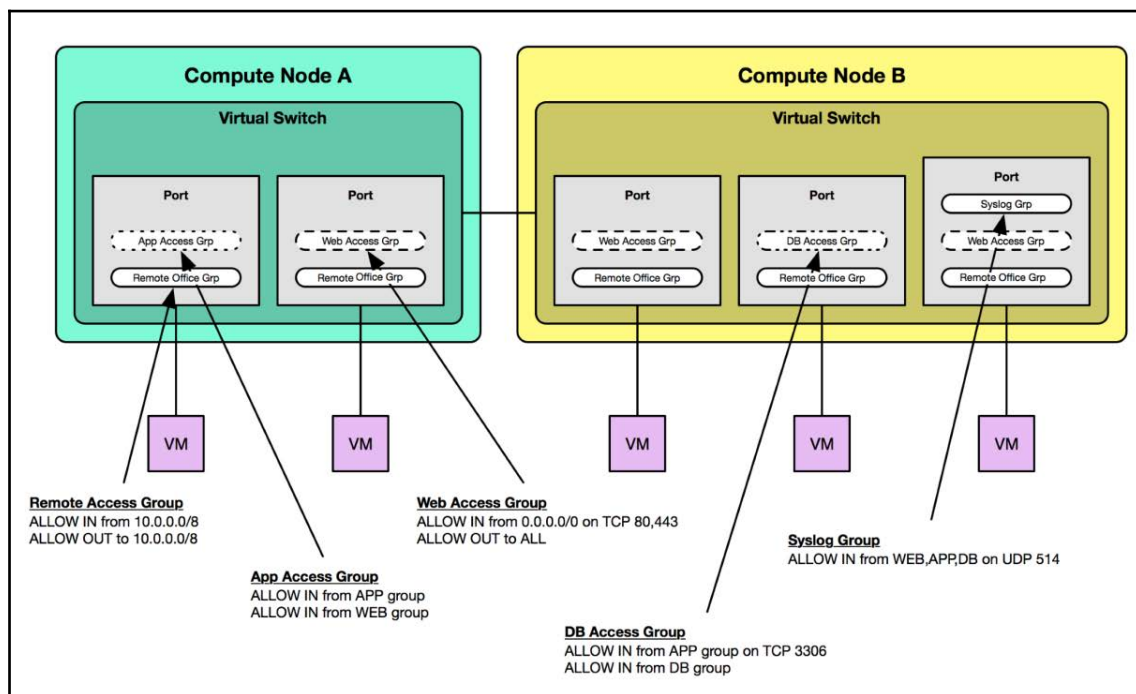
Rules within a security group are applied at a port level on the compute node, as demonstrated in the following diagram:



In the preceding diagram, ports within the virtual switch belong to one of three security groups: WEB, DB, or APP. When a change is made to a security group, such as adding or removing rules to the group, corresponding iptables rule changes are made automatically on the compute nodes.

Some users may use security groups to describe certain types of rules that should be applied to a particular instance port. For example, a security group can be used to categorize multiple hosts or subnets that are allowed access to a port. Multiple security groups can be applied to a port, and all rules defined in those groups are applied to the port. Because all traffic is implicitly denied and security group rules define only allowed traffic through a port, there is no chance of a rule in one security group applied to a port counteracting a rule in another security group applied to the same port.

The following example demonstrates the use of security groups to categorize traffic that is allowed access through a port:



When a port is created in Neutron, it is associated with a default security group unless a specific security group is specified. The default security group drops all ingress traffic and allows all egress traffic from the associated port. Rules can be added to, or removed from, the default security group to change its behavior. In addition, baseline port security rules are applied to every port that prohibit IP, DHCP, and MAC address spoofing. This behavior can be changed and will be discussed later on in this chapter.

An introduction to iptables

Iptables is a firewall built into Linux that allows a system administrator to define tables containing chains of rules that determine how network packets should be treated. Packets are processed by sequentially traversing rules in chains within the following tables:

- **Raw:** This is a default table that filters packets before any other table. It is mainly used for rules related to connection tracking.

- **Filter:** This is a default table for filtering packets.
- **NAT:** This is a default table used for network address translation.
- **Mangle:** This is a default table used for specialized packet alteration and is not used by the Security Group API.

A rule in a chain can cause a jump to another chain, which, in turn, can jump to another chain, and so on. This behavior can be repeated to whatever level of nesting is desired. If the traffic does not match the rules of a subchain, the system recalls the point at which the jump occurred and returns to that point for further processing. When iptables is enabled, every network packet arriving at or leaving an interface traverses at least one chain.

There are five default chains, and the origin of the packet determines which chain will be initially traversed. The five default chains include the following:

- **PREROUTING:** Packets will enter this chain before a routing decision is made. The PREROUTING chain is used by the raw, mangle, and NAT tables.
- **INPUT:** This is used when a packet is going to be locally delivered to the host machine. The INPUT chain is used by the mangle and filter tables.
- **FORWARD:** All packets that have been routed and were not for local delivery will traverse this chain. The FORWARD chain is used by the mangle and filter tables.
- **OUTPUT:** Packets sent from the host machine itself will traverse this chain. The OUTPUT chain is used by the raw, mangle, NAT, and filter tables.
- **POSTROUTING:** Packets will enter this chain when a routing decision has been made. The POSTROUTING chain is used by the mangle and NAT tables.

Each rule in a chain contains criteria that packets can be matched against. The rule may also contain a target, such as another chain, or a verdict, such as DROP or ACCEPT. As a packet traverses a chain, each rule is examined. If a rule does not match the packet, the packet is passed to the next rule. If a rule does match the packet, the rule takes the action indicated by the target or verdict.

Possible verdicts include the following:

- **ACCEPT:** The packet is accepted and sent to the application for processing
- **DROP:** The packet is dropped silently
- **REJECT:** The packet is dropped and an error message is sent to the sender
- **LOG:** The packet details are logged
- **DNAT:** This rewrites the destination IP of the packet
- **SNAT:** This rewrites the source IP of the packet
- **RETURN:** Processing returns to the calling chain

The ACCEPT, DROP, and REJECT verdicts are often used by the filter table. Common rule criteria include the following:

- `-p <protocol>`: Matches protocols such as TCP, UDP, ICMP, and more
- `-s <ip_addr>`: Matches source IP address
- `-d <ip_addr>`: Matches destination IP address
- `--sport`: Matches source port
- `--dport`: Matches destination port
- `-I <interface>`: Matches the interface from which the packet entered
- `-o <interface>`: Matches the interface from which the packet exits

Neutron abstracts the implementation of security group rules from users, but understanding how it works is important for operators tasked with troubleshooting connectivity. For more information on iptables, please visit the following resources:

- <https://www.booleanworld.com/depth-guide-iptables-linux-firewall/>
- <https://help.ubuntu.com/community/IptablesHowTo>

Using ipset

In OpenStack releases prior to Juno, for every security group referenced in a rule that was created, an exponential number of iptables rules were created that corresponded to each source and destination pair of addresses and ports. This behavior resulted in poor L2 agent performance as well as race conditions where virtual machine instances were connected to the virtual bridge but were unable to successfully connect to the network.

Beginning with the Juno release, the ipset extension to iptables is utilized in an attempt to reduce the number of iptables rules required by creating groups of addresses and ports that are stored efficiently for fast lookup.

Without ipset, iptables rules that allow connections on port 80 to a set of web instances may resemble the following:

```
iptables -A INPUT -p tcp -d 1.1.1.1 --dport 80 -j RETURN
iptables -A INPUT -p tcp -d 2.2.2.2 --dport 80 -j RETURN
iptables -A INPUT -p tcp -d 3.3.3.3 --dport 80 -j RETURN
iptables -A INPUT -p tcp -d 4.4.4.4 --dport 80 -j RETURN
```

The match syntax `-d x.x.x.x` in the preceding code means "match packets whose destination address is `x.x.x.x`". To allow all four addresses, four separate iptables rules with four separate match specifications must be defined.

Alternatively, a combination of `ipset` and `iptables` commands can be used to achieve the same result:

```
ipset -N webset iphash
ipset -A webset 1.1.1.1
ipset -A webset 2.2.2.2
ipset -A webset 3.3.3.3
ipset -A webset 4.4.4.4
iptables -A INPUT -p tcp -m set --match-set webset dst --dport 80 -j RETURN
```

The `ipset` command creates a new set, a `webset`, with four addresses. The `iptables` command references the set with `--m set --match-set webset dst`, which means "match packets whose destination matches an entry within the set named `webset`".

By using an `ipset`, only one rule is required to accomplish what previously took four rules. The savings are small in this example, but as instances are added to security groups and security group rules are configured, the reduction in rules has a noticeable impact on performance and reliability.

Working with security groups

Security groups can be managed using the Neutron ReST API, the OpenStack CLI, or the Horizon dashboard. Both methods offer a pretty complete experience and are discussed in the following sections.

Managing security groups in the CLI

From within the OpenStack command-line client, a number of commands can be used to manage security groups. The primary commands associated with security group management that will be discussed in this chapter are listed in the following table:

Security Group Commands	Description
<code>security group create</code>	Creates a new security group
<code>security group delete</code>	Deletes security group(s)
<code>security group list</code>	Lists security group(s)

<code>security group rule create</code>	Creates a new security group rule
<code>security group rule delete</code>	Deletes security group rule(s)
<code>security group rule list</code>	Lists security group rules
<code>security group rule show</code>	Displays security group rule details
<code>security group set</code>	Sets security group properties
<code>security group show</code>	Displays security group details
<code>server add security group</code>	Adds a security group to a server
<code>server remove security group</code>	Removes a security group from a server

Creating security groups in the CLI

To create a security group, use the `openstack security group create` command as follows:

```
openstack security group create
[--description <description>]
[--project <project>]
[--project-domain <project-domain>]
<name>
```



By default, security groups in Neutron are pre-populated with two egress rules that allow all outbound traffic over IPv4 and IPv6. Inbound traffic is not permitted by default.

Deleting security groups in the CLI

To delete a security group, use the `openstack security group delete` command and specify the ID or name of the security group:

```
openstack security group delete <group> [<group> ...]
```

To delete a security group named `MySampleSecGrp`, you can enter the following command:

```
openstack security group delete MySampleSecGrp
```

Alternatively, you can use the group's ID:

```
openstack security group delete f8ch3db3-e25b-4760-972a-cb700d9a73dc
```

Multiple security groups can also be deleted simultaneously, as follows:

```
openstack security group delete MySampleSecGrp MyOtherSampleSecGrp
```

Neutron will successfully delete the security group(s) as long as ports utilizing the group(s) have been deleted prior to the request.

Listing security groups in the CLI

To obtain a listing of security groups, use the `openstack security group list` command as follows:

```
openstack security group list  
[--project <project>]  
[--project-domain <project-domain>]
```

The output returned includes the ID, name, and description of all security groups associated with the scoped or specified project. If run as a user with the admin role, all security groups across all projects will be listed.

Showing the details of a security group in the CLI

To display the details of a security group, use the `openstack security group show` command, as shown here:

```
openstack security group show <group>
```

Either the name or the ID of the security group can be specified. The output returned includes the description, ID, name, associated project ID, and the individual rules within the security group.

Updating security groups in the CLI

To update the attributes of a security group, use the `openstack security group set` command as follows:

```
openstack security group set
[--name <new-name>]
[--description <description>]
<group>
```

Attempting to modify the name of a default security group will result in an error.

Creating security group rules in the CLI

To create a security group rule, use the `openstack security group rule create` command as follows:

```
openstack security group rule create
[--remote-ip<ip-address> | --remote-group <group>]
[--description <description>]
[--dst-port <port-range>]
[--icmp-type <icmp-type>]
[--icmp-code <icmp-code>]
[--protocol <protocol>]
[--ingress | --egress]
[--ethertype<ethertype>]
[--project <project>]
[--project-domain <project-domain>]
<group>
```

The `--remote-ip` argument is optional and allows you to specify the source address or network the rule applies to. The address or network should be defined in CIDR format.

The `--remote-group` argument is optional and allows you to specify the name or ID of a security group the rule should apply to rather than individual IP addresses or networks. For example, when creating a rule to allow inbound SQL traffic to database servers, you can specify the ID of a security group that application servers are a member of without having to specify their individual IP addresses.

The `--description` argument applies a description to the security group rule.

The `--dst-port` argument is optional and allows you to specify a destination port or range of ports separated by a colon, as in 137:139. This option is required for TCP and UDP protocols.

The `--icmp-type` and `--icmp-code` arguments specify the ICMP type and code, respectively, for ICMP IP protocols. Allowing the entire ICMP protocol without specific types and codes satisfies most use cases.

The `--protocol` argument is optional and allows you to match traffic based on the IP protocol. Possible options include `ah`, `dccp`, `egp`, `esp`, `gre`, `icmp`, `igmp`, `ipv6-encap`, `ipv6-frag`, `ipv6-icmp`, `ipv6-nonxt`, `ipv6-opts`, `ipv6-route`, `ospf`, `pgm`, `rsvp`, `sctp`, `tcp`, `udp`, `udplite`, `vrp` or an IP protocol number [0-255]. The default is `tcp`.

The `--ingress` argument means that the rule applies to incoming traffic, whereas the `--egress` argument means that the rule applies to outgoing traffic from the instance. These arguments are mutually exclusive, meaning only one can be used to describe the rule's direction. If left unspecified, the default direction is `ingress`.

The `--ethertype` argument is optional and allows you to specify whether the rule applies to IPv4 or IPv6 traffic.

The `--project` and `--project-domain` arguments are optional and can be used to specify a project and domain other than the one associated with the user creating the rule.

The `group` argument is used to specify the name or ID of the security group the rule should be associated with.

Deleting security group rules in the CLI

To delete a security group rule, use the `openstack security group rule delete` command and specify the ID or name of the security group rule:

```
openstack security group rule delete <group> [<group> ...]
```

To delete a security group named `MySampleSecGrpRule80`, you can enter the following command:

```
openstack security group rule delete MySampleSecGrpRule80
```

Multiple security groups can also be deleted simultaneously, like so:

```
openstack security group delete MySampleSecGrpRuleTcp80
MySampleSecGrpRuleUDP123
```



While it is possible to delete the rules within the default security group, it is not possible to delete the group itself.

Listing security group rules in the CLI

To obtain a listing of security group rules and associated security groups, use the `openstack security group rule list` command, as follows:

```
openstack security group rule list
[--protocol <protocol>]
[--ingress | --egress] [--long]
[<group>]
```

All arguments are optional but can be used to filter results. The output returned includes the ID, protocol, IP range, remote security group, and associated security group for all security group rules associated with the scoped project. Specifying `--long` will provide additional details not available in the standard output, including direction and ether type. If run as a user with the admin role, all security group rules across all projects will be listed.

Showing the details of a security group rule in the CLI

To display the details of a security group rule, use the `openstack security group rule show` command, as shown here:

```
openstack security group rule show <group>
```

Applying security groups to instances and ports

Applying security groups to instances within the CLI is typically done at instance creation using the `openstack server create` command that's shown here:

```
openstack server create
--flavor <FLAVOR_ID>
```

```
--image <IMAGE_ID>
--nic net-id=<NETWORK_ID>
--security-group <SECURITY_GROUP_ID>
INSTANCE_NAME
```

Security groups can also be applied to running instances by using either the `openstack port set` or the `openstack server add security group` commands. The following examples demonstrate the use of the `openstack port set` command to apply security groups to a port.

In this example, the security group will be applied to the port, and the associated rules will be implemented immediately:

```
openstack port set <PORT> --security-group <SECURITY_GROUP>
```

Multiple security groups can be associated with a port simultaneously. To apply multiple security groups to a port, use the `--security-group` argument before each security group, as shown here:

```
openstack port set <PORT>
--security-group <SECURITY_GROUP_1>
--security-group <SECURITY_GROUP_2>
--security-group <SECURITY_GROUP_3>
```

The following example demonstrates the use of the `openstack server add group` command to apply a security group to an instance:

```
openstack server add security group <INSTANCE> <SECURITY_GROUP>
```

Because a port cannot be specified as part of the command, the security group will be applied to all ports associated with the instance.

Removing security groups from instances and ports in the CLI

To remove an individual security group, use the `openstack server remove security group` command, as shown in the following example:

```
openstack server remove security group <INSTANCE> <SECURITY_GROUP>
```


Alternatively, you can use the `openstack port unset` command to remove the group from a port:

```
openstack port unset --security-group <SECURITY GROUP> <PORT>
```

To remove all security groups from a port, including the default group, use the `openstack port set` command with the `--no-security-group` argument, as shown in the following example:

```
openstack port set --no-security-group <PORT>
```

Implementing security group rules

In the following example, an instance named `WEB1` has been created that acts as a web server running Apache on ports 80 and 443. Making a request to the web server at `192.168.206.6:80` eventually times out:

```
root@controller01:~# ip netns exec qdhcp-03327707-c369-4bd7-bd71-a42d9bcf49b8 curl http://192.168.206.6/ --connect-timeout 5
curl: (28) Connection timed out after 5001 milliseconds
```

To demonstrate how security group rules are implemented on a compute node, take note of the following `WEB_SERVERS` security group:

```
root@controller01:~# openstack security group list
```

ID	Name	Description	Project
1e82757a-848b-47de-a528-e8f8fb001df9	default	Default security group	fddf005c8b854dad90aaf8cf861c9598
25094e89-69e1-46af-9af4-588885714da9	default	Default security group	d2aefac55d5040e3b7d32a469f110d50
d8f1b1ec-bc06-4474-be55-436dce9797ea	WEB_SERVERS	WEB_SERVERS	9233b6b4f6a54386af63c0a7b8f043c2
fae5da02-5d96-43df-aebf-bf0210d353bc	default	Default security group	9233b6b4f6a54386af63c0a7b8f043c2

The following screenshot demonstrates two security group rules being added to the WEB_SERVERS security group using the `openstack security group rule create` command. The rules allow inbound connections on ports 80 and 443 from any remote host, as defined by the CIDR 0.0.0.0/0:

```
root@controller01:~# openstack security group rule create --protocol tcp --dst-port 80 --remote-ip 0.0.0.0/0 WEB_SERVERS
```

Field	Value
created_at	2018-03-23T14:54:27Z
description	
direction	ingress
ether_type	IPv4
id	9240f5af-e55b-4e5d-bef4-735eb583575c
name	None
port_range_max	80
port_range_min	80
project_id	9233b6b4f6a54386af63c0a7b8f043c2
protocol	tcp
remote_group_id	None
remote_ip_prefix	0.0.0.0/0
revision_number	0
security_group_id	d8f1b1ec-bc06-4474-be55-436dce9797ea
updated_at	2018-03-23T14:54:27Z

```
root@controller01:~# openstack security group rule create --protocol tcp --dst-port 443 --remote-ip 0.0.0.0/0 WEB_SERVERS
```

Field	Value
created_at	2018-03-23T14:54:46Z
description	
direction	ingress
ether_type	IPv4
id	37abc0fb-4340-46e8-880e-a9e6de5b01fd
name	None
port_range_max	443
port_range_min	443
project_id	9233b6b4f6a54386af63c0a7b8f043c2
protocol	tcp
remote_group_id	None
remote_ip_prefix	0.0.0.0/0
revision_number	0
security_group_id	d8f1b1ec-bc06-4474-be55-436dce9797ea
updated_at	2018-03-23T14:54:46Z

Using the `openstack server add security group` command, the WEB_SERVERS security group can be applied to the WEB1 instance, as shown in the following screenshot:

```

root@controller01:~# openstack server add security group WEB1 WEB_SERVERS
root@controller01:~# openstack server show WEB1

```

Field	Value
OS-DCF:diskConfig	MANUAL
OS-EXT-AZ:availability_zone	nova
OS-EXT-SRV-ATTR:host	compute01
OS-EXT-SRV-ATTR:hypervisor_hostname	compute01.learningneutron.com
OS-EXT-SRV-ATTR:instance_name	instance-00000014
OS-EXT-STS:power_state	Running
OS-EXT-STS:task_state	None
OS-EXT-STS:vm_state	active
OS-SRV-USG:launched_at	2018-03-23T15:06:40.000000
OS-SRV-USG:terminated_at	None
accessIPv4	
accessIPv6	
addresses	MyVLANNetwork=192.168.206.6
config_drive	
created	2018-03-23T15:06:10Z
flavor	small (280d07fb-b932-464a-9e26-817522286be7)
hostId	e1e220df487a0b933fb3ec56bef96d6b8ca8b3240f56ca76435b733d
id	e890bc04-0b6b-42ba-bc8c-1d4288ac86ed
image	ubuntu-xenial-16.04 (ffdb76f0-6cce-4d79-99a5-ccc6e76d530a)
key_name	james
name	WEB1
progress	0
project_id	9233b6b4f6a54386af63c0a7b8f043c2
properties	
security_groups	name='WEB_SERVERS'
	name='default'
status	ACTIVE
updated	2018-03-23T15:06:40Z
user_id	8a679d8b7b574e54a5cfe02c422bbe68
volumes_attached	

Once a security group has been applied to the corresponding port of an instance, a series of iptables rules and chains are implemented on the compute node hosting the instance. A quick connectivity check shows that the rule(s) work as expected:

```

root@controller01:~# ip netns exec qdhcp-03327707-c369-4bd7-bd71-a42d9bcf49b8 curl http://192.168.206.6/ --connect-timeout 5
Hello World! I am WEB1!

```

Stepping through the chains

The implementation of security group rules using iptables is similar in both Linux bridge and Open vSwitch-based environments. On `compute01`, a compute node running the Linux bridge agent and hosting the instance in this example, shows that iptables rules applied by the Neutron agent can be observed using the `iptables-save` or `iptables -L` commands.

For readability, only the raw and filter tables of rules related to the instance are shown in the following screenshot. Some comments have been removed or truncated to fit the page:

```

root@compute01:~# iptables-save
# Generated by iptables-save v1.6.0 on Fri Mar 23 15:36:48 2018
*raw
:PREROUTING ACCEPT [62:16623]
:OUTPUT ACCEPT [57:36069]
:neutron-linuxbri-OUTPUT - [0:0]
:neutron-linuxbri-PREROUTING - [0:0]
-A PREROUTING -j neutron-linuxbri-PREROUTING
-A OUTPUT -j neutron-linuxbri-OUTPUT
-A neutron-linuxbri-PREROUTING -m physdev --physdev-in brq03327707-c3 -m comment --comment "Set zone for dcc4b66-c1" -j CT --zone 1
-A neutron-linuxbri-PREROUTING -i brq03327707-c3 -m comment --comment "Set zone for dcc4b66-c1" -j CT --zone 1
-A neutron-linuxbri-PREROUTING -m physdev --physdev-in tapadcc4b66-c1 -m comment --comment "Set zone for dcc4b66-c1" -j CT --zone 1
COMMIT
# Completed on Fri Mar 23 15:36:48 2018
# Generated by iptables-save v1.6.0 on Fri Mar 23 15:36:48 2018
*filter
:INPUT ACCEPT [63:16675]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [64:38153]
:neutron-filter-top - [0:0]
:neutron-linuxbri-FORWARD - [0:0]
:neutron-linuxbri-INPUT - [0:0]
:neutron-linuxbri-OUTPUT - [0:0]
:neutron-linuxbri-iadcc4b66-c - [0:0]
:neutron-linuxbri-local - [0:0]
:neutron-linuxbri-oadcc4b66-c - [0:0]
:neutron-linuxbri-sadcc4b66-c - [0:0]
:neutron-linuxbri-sg-chain - [0:0]
:neutron-linuxbri-sg-fallback - [0:0]
-A INPUT -j neutron-linuxbri-INPUT
-A FORWARD -j neutron-filter-top
-A FORWARD -j neutron-linuxbri-FORWARD
-A OUTPUT -j neutron-filter-top
-A OUTPUT -j neutron-linuxbri-OUTPUT
-A neutron-filter-top -j neutron-linuxbri-local
-A neutron-linuxbri-FORWARD -m physdev --physdev-out tapadcc4b66-c1 --physdev-is-bridged -j neutron-linuxbri-sg-chain
-A neutron-linuxbri-FORWARD -m physdev --physdev-in tapadcc4b66-c1 --physdev-is-bridged -j neutron-linuxbri-sg-chain
-A neutron-linuxbri-INPUT -m physdev --physdev-in tapadcc4b66-c1 --physdev-is-bridged -j neutron-linuxbri-oadcc4b66-c
-A neutron-linuxbri-iadcc4b66-c -m state --state RELATED,ESTABLISHED -j RETURN
-A neutron-linuxbri-iadcc4b66-c -d 192.168.206.6/32 -p udp -m udp --sport 67 --dport 68 -j RETURN
-A neutron-linuxbri-iadcc4b66-c -d 255.255.255.255/32 -p udp -m udp --sport 67 --dport 68 -j RETURN
-A neutron-linuxbri-iadcc4b66-c -p tcp -m tcp --dport 443 -j RETURN
-A neutron-linuxbri-iadcc4b66-c -p tcp -m tcp --dport 80 -j RETURN
-A neutron-linuxbri-iadcc4b66-c -m set --match-set NIPv4fae5da02-5d96-43df-aebf- src -j RETURN
-A neutron-linuxbri-iadcc4b66-c -p icmp -j RETURN
-A neutron-linuxbri-iadcc4b66-c -p tcp -m tcp --dport 22 -j RETURN
-A neutron-linuxbri-iadcc4b66-c -m state --state INVALID -j DROP
-A neutron-linuxbri-iadcc4b66-c -j neutron-linuxbri-sg-fallback
-A neutron-linuxbri-oadcc4b66-c -s 0.0.0.0/32 -d 255.255.255.255/32 -p udp -m udp --sport 68 --dport 67 -j RETURN
-A neutron-linuxbri-oadcc4b66-c -j neutron-linuxbri-sadcc4b66-c
-A neutron-linuxbri-oadcc4b66-c -p udp -m udp --sport 68 --dport 67 -m comment --comment "Allow DHCP client traffic." -j RETURN
-A neutron-linuxbri-oadcc4b66-c -p udp -m udp --sport 67 --dport 68 -m comment --comment "Prevent DHCP Spoofing by VM." -j DROP
-A neutron-linuxbri-oadcc4b66-c -m state --state RELATED,ESTABLISHED -j RETURN
-A neutron-linuxbri-oadcc4b66-c -j RETURN
-A neutron-linuxbri-oadcc4b66-c -m state --state INVALID -j DROP
-A neutron-linuxbri-oadcc4b66-c -m comment --comment "Send unmatched traffic to the fallback chain." -j neutron-linuxbri-sg-fallback
-A neutron-linuxbri-sadcc4b66-c -s 192.168.206.6/32 -m mac --mac-source FA:16:3E:DC:D2:AB -j RETURN
-A neutron-linuxbri-sadcc4b66-c -m comment --comment "Drop traffic without an IP/MAC allow rule." -j DROP
-A neutron-linuxbri-sg-chain -m physdev --physdev-out tapadcc4b66-c1 --physdev-is-bridged -j neutron-linuxbri-iadcc4b66-c
-A neutron-linuxbri-sg-chain -m physdev --physdev-in tapadcc4b66-c1 --physdev-is-bridged -j neutron-linuxbri-oadcc4b66-c
-A neutron-linuxbri-sg-chain -j ACCEPT
-A neutron-linuxbri-sg-fallback -m comment --comment "Default drop rule for unmatched traffic." -j DROP
COMMIT
# Completed on Fri Mar 23 15:36:48 2018

```

First, the PREROUTING chain of the raw table is consulted. In the PREROUTING chain, we can find rules associating traffic traversing an instance's interface and/or related network to specific conntrack zone(s):

```
*raw
:PREROUTING ACCEPT [62:16623]
:OUTPUT ACCEPT [57:36069]
:neutron-linuxbri-OUTPUT - [0:0]
:neutron-linuxbri-PREROUTING - [0:0]
-A PREROUTING -j neutron-linuxbri-PREROUTING
-A OUTPUT -j neutron-linuxbri-OUTPUT
-A neutron-linuxbri-PREROUTING -m physdev --physdev-in brq03327707-c3 -m comment --comment "Set zone for dcc4b66-c1" -j CT --zone 1
-A neutron-linuxbri-PREROUTING -i brq03327707-c3 -m comment --comment "Set zone for dcc4b66-c1" -j CT --zone 1
-A neutron-linuxbri-PREROUTING -m physdev --physdev-in tapadcc4b66-c1 -m comment --comment "Set zone for dcc4b66-c1" -j CT --zone 1
```



Each Neutron network corresponds to a conntrack zone, which further helps distinguish traffic in the case of overlapping address space between projects. Conntrack is one piece of the Connection Tracking System and is utilized by iptables when performing stateful packet inspection.

Next, network traffic may traverse the INPUT chain of the filter table as follows:

```
-A INPUT -j neutron-linuxbri-INPUT
```

This rule jumps to a chain named neutron-linuxbri-INPUT, as shown here:

```
-A neutron-linuxbri-INPUT -m physdev
--physdev-in tapadcc4b66-c1
--physdev-is-bridged -j neutron-linuxbri-oadcc4b66-c
```

The rule states that traffic leaving the instance through the tapadcc4b66-c1 interface should be forwarded to the neutron-linuxbri-oadcc4b66-c chain, as shown here:

```
-A neutron-linuxbri-oadcc4b66-c -s 0.0.0.0/32 -d 255.255.255.255/32 -p udp -m udp --sport 68 --dport 67 -j RETURN
-A neutron-linuxbri-oadcc4b66-c -j neutron-linuxbri-sadcc4b66-c
-A neutron-linuxbri-oadcc4b66-c -p udp -m udp --sport 68 --dport 67 -m comment --comment "Allow DHCP client traffic." -j RETURN
-A neutron-linuxbri-oadcc4b66-c -p udp -m udp --sport 67 --dport 68 -m comment --comment "Prevent DHCP Spoofing by VM." -j DROP
-A neutron-linuxbri-oadcc4b66-c -m state --state RELATED,ESTABLISHED -j RETURN
-A neutron-linuxbri-oadcc4b66-c -j RETURN
-A neutron-linuxbri-oadcc4b66-c -m state --state INVALID -j DROP
-A neutron-linuxbri-oadcc4b66-c -m comment --comment "Send unmatched traffic to the fallback chain." -j neutron-linuxbri-sg-fallback
```

The rules of the neutron-linuxbri-oadcc4b66-c chain states that outgoing DHCP traffic should only be allowed when the client is attempting to obtain an address from a DHCP server. By default, instances are not allowed to act as DHCP servers and traffic will be dropped accordingly. The rules also state that traffic marked as RETURN or ESTABLISHED will be allowed, but INVALID packets will be dropped. Lastly, any traffic that does not match is dropped by a rule in the neutron-linuxbri-sg-fallback chain.



Iptables may process traffic using the INPUT or FORWARD chains depending on the destination address and whether it is local to the node or not. In most cases, the FORWARD chains will be used instead.

If traffic has survived the INPUT chains, it then moves on to the FORWARD chains seen here:

```
-A FORWARD -j neutron-filter-top
-A FORWARD -j neutron-linuxbri-FORWARD
...
-A neutron-filter-top -j neutron-linuxbri-local
-A neutron-linuxbri-FORWARD -m physdev --physdev-out tapadcc4b66-c1 --physdev-is-bridged -j neutron-linuxbri-sg-chain
-A neutron-linuxbri-FORWARD -m physdev --physdev-in tapadcc4b66-c1 --physdev-is-bridged -j neutron-linuxbri-sg-chain
```

The first rule causes iptables to jump to the `neutron-filter-top` chain. From there, iptables jumps to the `neutron-linuxbri-local` chain for further processing. Because there are no rules defined in that chain, iptables returns to the calling chain, `neutron-filter-top`. Once all rules have been processed, iptables returns to the previous calling chain, `FORWARD`.

The next rule in the `FORWARD` chain that is processed is as follows:

```
-A FORWARD -j neutron-linuxbri-FORWARD
```

The mentioned rule causes iptables to jump to the `neutron-linuxbri-FORWARD` chain seen here:

```
-A neutron-linuxbri-FORWARD -m physdev --physdev-out tapadcc4b66-c1 --physdev-is-bridged -j neutron-linuxbri-sg-chain
-A neutron-linuxbri-FORWARD -m physdev --physdev-in tapadcc4b66-c1 --physdev-is-bridged -j neutron-linuxbri-sg-chain
```

The `-m` flag followed by `physdev` is a directive to iptables to use an extended packet matching module that supports devices enslaved to a bridge device.



Remember: When the Linux bridge agent is used, tap interfaces for instances are connected to network bridges prefaced with `brq-*`. When the OVS agent is used, the tap interfaces are connected to their own Linux bridge prefaced with `qbr-*`.

The packet will match one of the two rules based on the direction the packet is headed through the interface. In either case, iptables jumps to the `neutron-linuxbri-sg-chain` chain, as shown here:

```
-A neutron-linuxbri-sg-chain -m physdev --physdev-out tapadcc4b66-c1 --physdev-is-bridged -j neutron-linuxbri-iadcc4b66-c
-A neutron-linuxbri-sg-chain -m physdev --physdev-in tapadcc4b66-c1 --physdev-is-bridged -j neutron-linuxbri-oadcc4b66-c
-A neutron-linuxbri-sg-chain -j ACCEPT
```

The direction of the packet will again dictate which rule is matched. Traffic centering the instance through the `tapadcc4b66-c1` interface will be processed by the `neutron-linuxbri-iadcc4b66-c` chain as follows:

```
-A neutron-linuxbri-iadcc4b66-c -m state --state RELATED,ESTABLISHED -j RETURN
-A neutron-linuxbri-iadcc4b66-c -d 192.168.206.6/32 -p udp -m udp --sport 67 --dport 68 -j RETURN
-A neutron-linuxbri-iadcc4b66-c -d 255.255.255.255/32 -p udp -m udp --sport 67 --dport 68 -j RETURN
-A neutron-linuxbri-iadcc4b66-c -p tcp -m tcp --dport 443 -j RETURN
-A neutron-linuxbri-iadcc4b66-c -p tcp -m tcp --dport 80 -j RETURN
-A neutron-linuxbri-iadcc4b66-c -m set --match-set NIPv4fae5da02-5d96-43df-aebf- src -j RETURN
-A neutron-linuxbri-iadcc4b66-c -p icmp -j RETURN
-A neutron-linuxbri-iadcc4b66-c -p tcp -m tcp --dport 22 -j RETURN
-A neutron-linuxbri-iadcc4b66-c -m state --state INVALID -j DROP
-A neutron-linuxbri-iadcc4b66-c -j neutron-linuxbri-sg-fallback
```



The name of a security group chain corresponds to the first 9 characters of the UUID of the Neutron port of which it is associated.

In the mentioned rule, iptables uses the state module to determine the state of the packet. Combined with connection tracking, iptables is able to track the connection and determine the following states of the packet: `INVALID`, `NEW`, `RELATED`, or `ESTABLISHED`. The state of the packet results in an appropriate action being taken. Traffic not matched by any rule is dropped by the `neutron-linuxbri-sg-fallback` chain:

```
-A neutron-linuxbri-sg-fallback -j DROP
```

Traffic exiting the instance through the `tapadcc4b66-c1` interface is processed by the `neutron-linuxbri-oadcc4b66-c` chain as follows:

```
-A neutron-linuxbri-oadcc4b66-c -s 0.0.0.0/32 -d 255.255.255.255/32 -p udp -m udp --sport 68 --dport 67 -j RETURN
-A neutron-linuxbri-oadcc4b66-c -j neutron-linuxbri-sadcc4b66-c
-A neutron-linuxbri-oadcc4b66-c -p udp -m udp --sport 68 --dport 67 -m comment --comment "Allow DHCP client traffic." -j RETURN
-A neutron-linuxbri-oadcc4b66-c -p udp -m udp --sport 67 --dport 68 -m comment --comment "Prevent DHCP Spoofing by VM." -j DROP
-A neutron-linuxbri-oadcc4b66-c -m state --state RELATED,ESTABLISHED -j RETURN
-A neutron-linuxbri-oadcc4b66-c -j RETURN
-A neutron-linuxbri-oadcc4b66-c -m state --state INVALID -j DROP
-A neutron-linuxbri-oadcc4b66-c -m comment --comment "Send unmatched traffic to the fallback chain." -j neutron-linuxbri-sg-fallback
```

The first UDP rule allows the instance to send `DHCPDISCOVER` and `DHCPREQUEST` broadcast packets on UDP port 67. All other traffic is then processed by the `neutron-linuxbri-sadcc4b66-c` chain as follows:

```
-A neutron-linuxbri-sadcc4b66-c -s 192.168.206.6/32 -m mac --mac-source FA:16:3E:DC:D2:AB -j RETURN
-A neutron-linuxbri-sadcc4b66-c -m comment --comment "Drop traffic without an IP/MAC allow rule." -j DROP
```

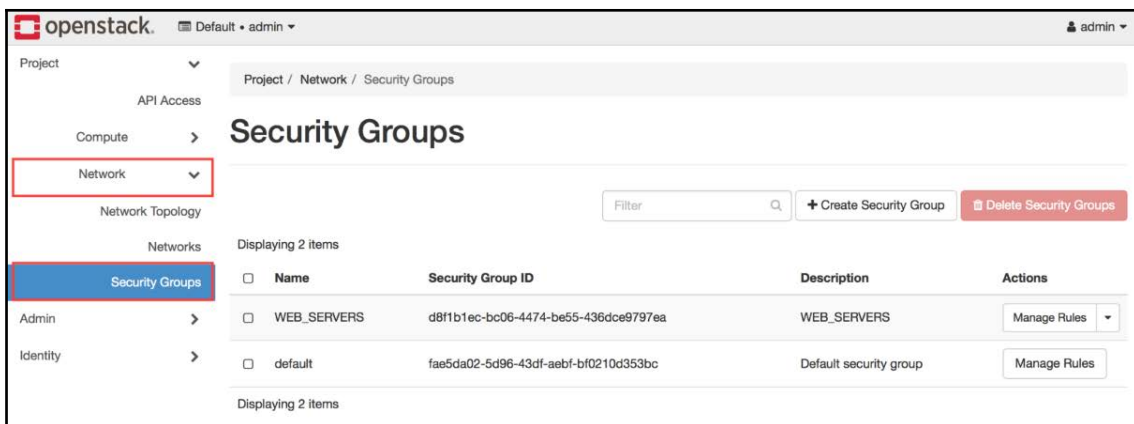
The preceding rule prevents an instance from performing IP and MAC address spoofing. Any traffic exiting the `tapadcc4b66-c1` interface must be sourced from IP address `192.168.206.6/32` and MAC address `FA:16:3E:DC:D2:AB`. To permit outbound traffic from additional IP or MAC addresses, use the `Neutron allowed-address-pairs` extension, which will be discussed later on in this chapter.

Further processing includes preventing DHCP spoofing and verifying the state of the packet and performing the appropriate action. Traffic eventually returns to the `neutron-linuxbri-sg-chain`, calling chain where it is allowed through:

```
-A neutron-linuxbri-sg-chain -j ACCEPT
```

Working with security groups in the dashboard

Within the Horizon dashboard, security groups are managed within the **Security Groups** section under the **Network** tab:



The screenshot shows the OpenStack dashboard interface. In the left sidebar, the 'Network' menu item is highlighted with a red box, and the 'Security Groups' sub-menu is also highlighted with a blue box. The main content area displays the 'Security Groups' page. At the top right, there are buttons for 'Filter', '+ Create Security Group', and 'Delete Security Groups'. Below these, a table lists two security groups:

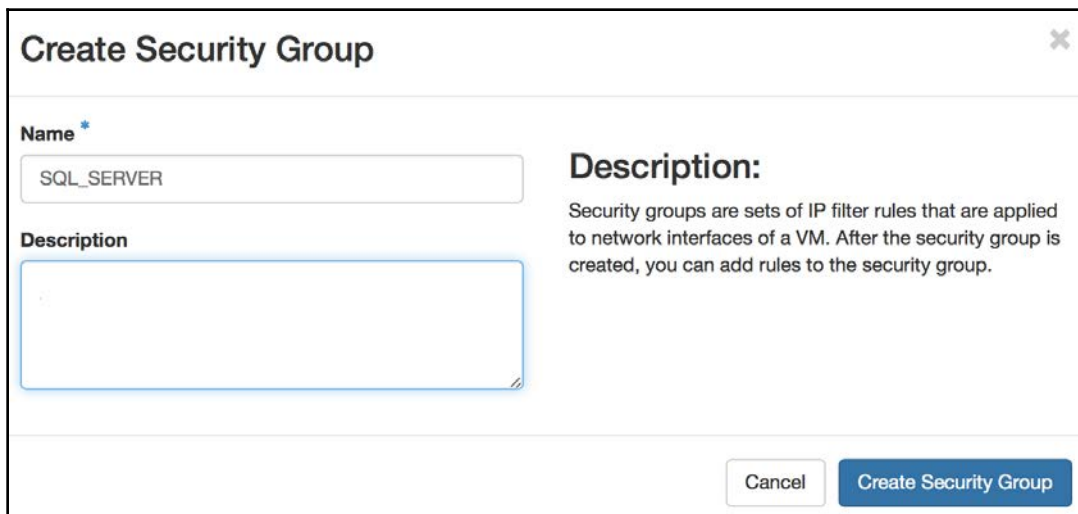
☐	Name	Security Group ID	Description	Actions
☐	WEB_SERVERS	d8f1b1ec-bc06-4474-be55-436dce9797ea	WEB_SERVERS	<button>Manage Rules</button>
☐	default	fae5da02-5d96-43df-aebf-bf0210d353bc	Default security group	<button>Manage Rules</button>

At the bottom of the table, it says 'Displaying 2 items'.

Creating a security group

To create a security group, perform the following steps.

Click on the **Create Security Group** button in the upper right-hand corner of the screen. A window will appear that will allow you to create a security group:

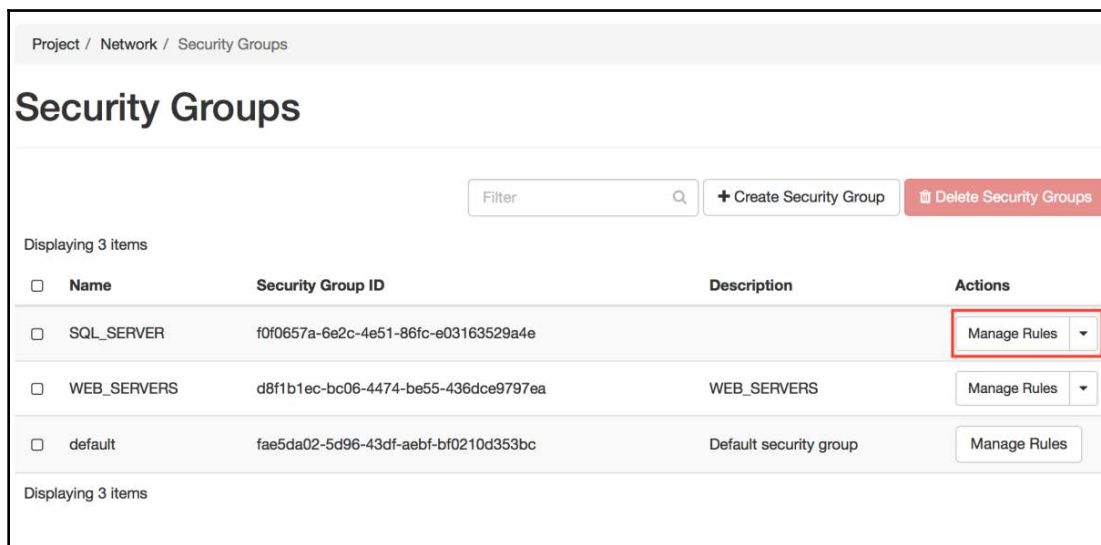


The 'Create Security Group' dialog box is shown. It has a title bar with a close button. The 'Name' field is required (indicated by an asterisk) and contains the text 'SQL_SERVER'. The 'Description' field is empty. To the right of the fields, there is a 'Description:' section with the text: 'Security groups are sets of IP filter rules that are applied to network interfaces of a VM. After the security group is created, you can add rules to the security group.' At the bottom right, there are two buttons: 'Cancel' and 'Create Security Group'.

The **Name** field is required. When you are ready to proceed, click on the blue **Create Security Group** button to create the security group.

Managing security group rules

Once created, you will be returned to the **Security Groups** section where you can add rules to the security group by clicking the **Manage Rules** button of the corresponding group:



Project / Network / Security Groups

Security Groups

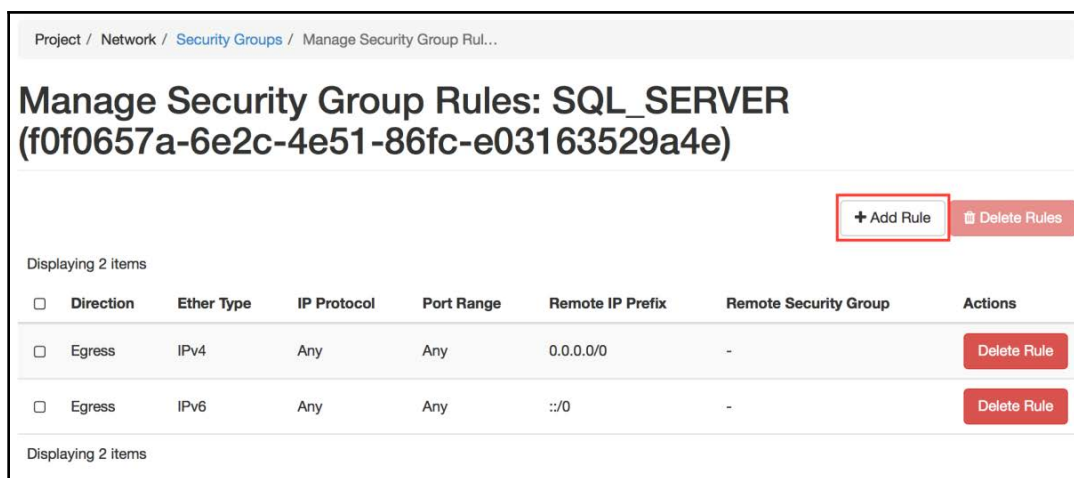
Filter + Create Security Group Delete Security Groups

Displaying 3 items

☐	Name	Security Group ID	Description	Actions
☐	SQL_SERVER	f0f0657a-6e2c-4e51-86fc-e03163529a4e		Manage Rules
☐	WEB_SERVERS	d8f1b1ec-bc06-4474-be55-436dce9797ea	WEB_SERVERS	Manage Rules
☐	default	fae5da02-5d96-43df-aebf-bf0210d353bc	Default security group	Manage Rules

Displaying 3 items

To delete a rule, click the red **Delete Rule** button next to the corresponding security group rule. To add a rule, click on the **Add Rule** button in the upper right-hand corner:



Project / Network / Security Groups / Manage Security Group Rules

Manage Security Group Rules: SQL_SERVER (f0f0657a-6e2c-4e51-86fc-e03163529a4e)

+ Add Rule Delete Rules

Displaying 2 items

☐	Direction	Ether Type	IP Protocol	Port Range	Remote IP Prefix	Remote Security Group	Actions
☐	Egress	IPv4	Any	Any	0.0.0.0/0	-	Delete Rule
☐	Egress	IPv6	Any	Any	:::0	-	Delete Rule

Displaying 2 items

A window will appear that will allow you to create rules. Within the rule list, you can choose from a predefined list of protocols or create a custom rule as follows:

Add Rule ✕

Rule *
Custom TCP Rule ▼

Direction
Ingress ▼

Open Port *
Port ▼

Port ?
3306

Remote * ?
CIDR ▼

CIDR ?
0.0.0.0/0

Ether Type
IPv4

Description:

Rules define which traffic is allowed to instances assigned to the security group. A security group rule consists of three main parts:

Rule: You can specify the desired rule template or use custom rules, the options are Custom TCP Rule, Custom UDP Rule, or Custom ICMP Rule.

Open Port/Port Range: For TCP and UDP rules you may choose to open either a single port or a range of ports. Selecting the "Port Range" option will provide you with space to provide both the starting and ending ports for the range. For ICMP rules you instead specify an ICMP type and code in the spaces provided.

Remote: You must specify the source of the traffic to be allowed via this rule. You may do so either in the form of an IP address block (CIDR) or via a source group (Security Group). Selecting a security group as the source will allow any other instance in that security group access to any other instance via this rule.

Cancel

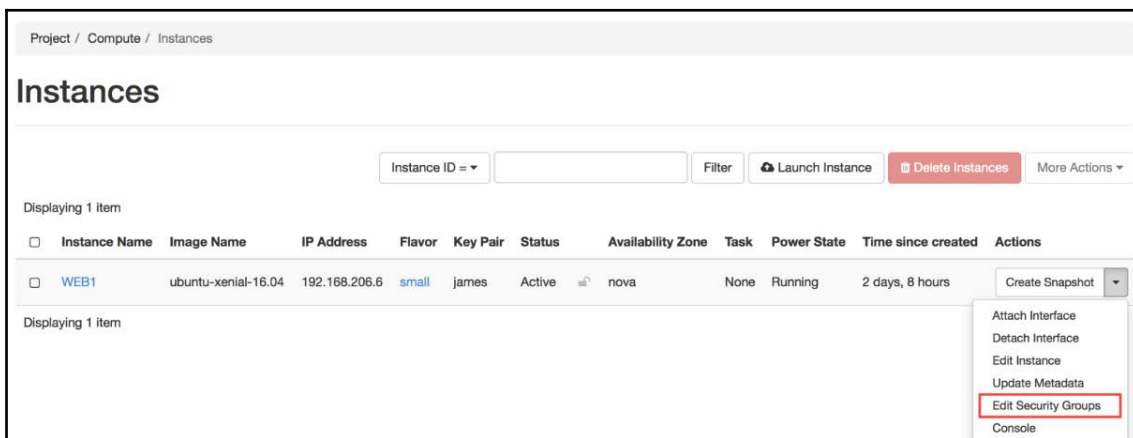
Add

The wizard allows you to specify the direction, port or range of ports, and the remote network or group. To complete the rule creation process, click on the blue **Add** button.

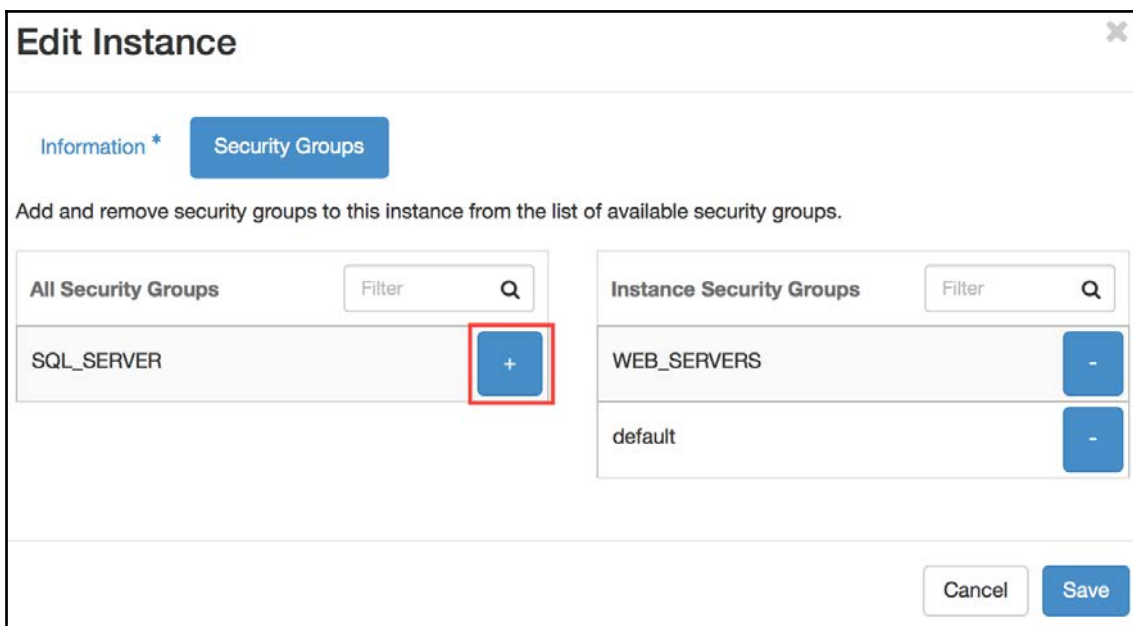
[230]

Applying security groups to instances

To apply a security group to an instance, return to the **Instances** section of the **Compute** panel. Click on the arrow under the **Actions** menu next to the instance and choose **Edit Security Groups**:



A window will appear that allows you to apply or remove security groups from an instance. Hit the blue plus sign next to the group to apply it to an instance:



Once clicked, the group will move to the **Instance Security Groups** column, as shown in the following screenshot:

The screenshot shows the 'Edit Instance' dialog box with a close button (X) in the top right corner. It has two tabs: 'Information *' and 'Security Groups'. The 'Security Groups' tab is active. Below the tabs, there is a text instruction: 'Add and remove security groups to this instance from the list of available security groups.' The dialog is divided into two main sections. The left section, titled 'All Security Groups', contains a 'Filter' input field and a search icon, and below it, the text 'No security groups found.' The right section, titled 'Instance Security Groups', also has a 'Filter' input field and a search icon. Below this, there is a list of security groups: 'WEB_SERVERS', 'default', and 'SQL_SERVER'. Each group has a blue button with a minus sign (-) to its right. The 'SQL_SERVER' group and its button are highlighted with a red rectangular box. At the bottom right of the dialog, there are two buttons: 'Cancel' and 'Save'. The 'Save' button is highlighted with a red rectangular box.

Click on the blue **Save** button to apply the changes to the instance.

Disabling port security

By default, Neutron applies anti-spoofing rules to all ports to ensure that unexpected or undesired traffic cannot originate from or pass through a port. This includes rules that prohibit instances from running DHCP servers or from acting as routers. To address the latter, the `allowed-address-pairs` extension can be used to allow additional IPs, subnets, and MAC addresses through the port. However, additional functionality may be required that cannot be addressed by the `allowed-address-pairs` extension.

In the Kilo release of OpenStack, the port security extension was introduced for the ML2 plugin, which allows all packet filtering to be disabled on a port. This includes default rules that prevent IP and MAC spoofing as well as security group functionality. This extension is especially useful when deploying instances for use as a router or a load balancer. The port security extension requires additional configuration that will be discussed in the following sections.

Configuring Neutron

To enable the port security extension, edit the ML2 configuration file on the `controller01` node at `/etc/neutron/plugins/ml2/ml2_conf.ini` and add the following to the `[ml2]` section:

```
[ml2]
...
extension_drivers = port_security
```

Restart the `neutron-server` service for the changes to take effect:

```
# systemctl restart neutron-server
```

You can verify that the port security extension is enabled by verifying its name in the extension list that's provided by the `openstack extension list --network` command, as shown here:

```
root@controller01:~# openstack extension list --network
```

Name	Alias	Description
Default Subnetpools	default-subnetpools	Provides ability to mark and use a subnetpool
Network IP Availability	network-ip-availability	Provides IP availability data for each network
Network Availability Zone	network_availability_zone	Availability zone support for network.
Network MTU (writable)	net-mtu-writable	Provides a writable MTU attribute for a network
Port Binding	binding	Expose port bindings of a virtual port to external
agent	agent	The agent management extension.
Subnet Allocation	subnet_allocation	Enables allocation of subnets from a subnet pool
DHCP Agent Scheduler	dhcp_agent_scheduler	Schedule networks among dhcp agents
Tag support	tag	Enables to set tag on resources.
Neutron external network	external-net	Adds external network attribute to network resource
Neutron Service Flavors	flavors	Flavor specification for Neutron advanced services
Network MTU	net-mtu	Provides MTU attribute for a network resource
Availability Zone	availability_zone	The availability zone extension.
Quota management support	quotas	Expose functions for quotas management per tenant
Tag support for: trunk, policy, security_group, floatingip	standard-attr-tag	Enables to set tag on resources with standard attributes
If-Match constraints based on revision_number	revision-if-match	Extension indicating that If-Match based on revision number
Provider Network	provider	Expose mapping of virtual networks to physical networks
Multi Provider Network	multi-provider	Expose mapping of virtual networks to multiple physical networks
Quota details management support	quota_details	Expose functions for quotas usage statistics
Address scope	address-scope	Address scopes extension.
Subnet service types	subnet-service-types	Provides ability to set the subnet service_type
Resource timestamps	standard-attr-timestamp	Adds created_at and updated_at fields to all resources
Neutron Service Type Management	service-type	API for retrieving service providers for Neutron
Tag support for resources: subnet, subnetpool, port, router	tag-ext	Extends tag support to more L2 and L3 resources
Neutron Extra DHCP options	extra_dhcp_opt	Extra options configuration for DHCP. For example, option 60
Resource revision numbers	standard-attr-revisions	This extension will display the revision number for resources
Pagination support	pagination	Extension that indicates that pagination is enabled
Sorting support	sorting	Extension that indicates that sorting is enabled
security-group	security-group	The security groups extension.
RBAC Policies	rbac-policies	Allows creation and modification of policies
standard-attr-description	standard-attr-description	Extension to add descriptions to standard attributes
Port Security	port-security	Provides port security
Allowed Address Pairs	allowed-address-pairs	Provides allowed address pairs
project_id field enabled	project-id	Extension that indicates that project_id field is enabled

Disabling port security for all ports on a network

Even without the port security extension in place, Neutron implements the default behavior of the extension by implementing DHCP, MAC address, and IP address spoofing rules on every port. The port security extension allows users with the admin role to disable port security on individual ports or network-wide. Port security can be disabled on all ports connected to a particular network by setting the `--disable-port-security` argument during network creation:

```
root@controller01:~# openstack network create --disable-port-security NetworkWithoutPortSecurity
```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-03-26T01:14:33Z
description	
dns_domain	None
id	eaf4b54e-fe28-42a4-8cb5-e44f4e5ef452
ipv4_address_scope	None
ipv6_address_scope	None
is_default	None
is_vlan_transparent	None
mtu	1450
name	NetworkWithoutPortSecurity
port security enabled	False
project_id	9233b6b4f6a54386af63c0a7b8f043c2
provider:network_type	vxlان
provider:physical_network	None
provider:segmentation_id	72
qos_policy_id	None
revision_number	2
router:external	Internal
segments	None
shared	False
status	ACTIVE
subnets	
tags	
updated_at	2018-03-26T01:14:33Z

When a port is created and attached to the network, its `port_security_enabled` attribute will be set to `False` automatically:

```
root@controller01:~# openstack port create --network NetworkWithoutPortSecurity TestPort
```

Field	Value
admin_state_up	UP
allowed_address_pairs	
binding_host_id	
binding_profile	
binding_vif_details	
binding_vif_type	unbound
binding_vnic_type	normal
created_at	2018-03-26T01:16:42Z
data_plane_status	None
description	
device_id	
device_owner	
dns_assignment	None
dns_name	None
extra_dhcp_opts	
fixed_ips	
id	1848fbc7-3124-403c-b283-4db41a30f6a2
ip_address	None
mac_address	fa:16:3e:64:e5:ea
name	TestPort
network_id	eaf4b54e-fe28-42a4-8cb5-e44f4e5ef452
option_name	None
option_value	None
port_security_enabled	False
project_id	9233b6b4f6a54386af63c0a7b8f043c2
qos_policy_id	None
revision_number	2
security_group_ids	
status	DOWN
subnet_id	None
tags	
trunk_details	None
updated_at	2018-03-26T01:16:42Z

It is important to know that when port security is disabled on a port, the API will not allow the port to be associated with any security groups. The lack of security group rules means all traffic is allowed in and out of a port. Disabling port security means any filtering must be implemented within the guest operating system.

Modifying port security on an individual port

When the port security extension is enabled, port security can be enabled or disabled on an individual port by setting the `port_security_enabled` attribute accordingly during the creation or update of a port.

The following screenshot demonstrates updating a port to enable port security:

```
root@controller01:~# openstack port show TestPort -c port_security_enabled
+-----+-----+
| Field                | Value |
+-----+-----+
| port_security_enabled | False |
+-----+-----+

root@controller01:~# openstack port set --enable-port-security TestPort
root@controller01:~# openstack port show TestPort -c port_security_enabled
+-----+-----+
| Field                | Value |
+-----+-----+
| port_security_enabled | True  |
+-----+-----+
```

By enabling port security, default anti-spoofing rules will be applied to the port, and security groups can be applied as needed. Disabling port security will remove the default anti-spoofing rules and security groups will not be allowed on the port.

Allowed address pairs

The `allowed-address-pairs` extension can be used to allow additional IPs, subnets, and MAC addresses, other than the fixed IP and MAC address associated with the port, to act as source addresses for traffic leaving a port or virtual interface. This is useful when treating an instance as a routing device or VPN concentrator, or when implementing high-availability between multiple instances using addresses that need to "float" between them, such as an `haproxy` and/or `keepalived` implementation.

Existing allowed address pairs can be found in the details of each port by using the `openstack port show` command. For every network and/or MAC address that should be allowed, the `openstack port set` command should be used with the `--allowed-address` argument, as shown here:

```
openstack port set <port>
--allowed-address ip_address=<IP_ADDR>,mac-address=<MAC_ADDR>
```



The MAC address value is optional. When a MAC address is not specified, the MAC address of the port is used.

Multiple allowed address pairs can be associated with a Neutron port simultaneously. To apply multiple allowed address pairs to a port, simply specify multiple `ip_address` and `mac_address` key/value pairs, as shown here:

```
openstack port set <PORT>
--allowed-address ip-address=<IP_ADDR>,mac-address=<MAC_ADDR>
--allowed-address ip-address=<IP_ADDR>,mac-address=<MAC_ADDR>
...
--allowed-address ip-address=<IP_ADDR>,mac-address=<MAC_ADDR>
```

To remove all allowed address pairs from the port, use the `openstack port set` command with the `--no-allowed-address` argument.

Summary

Security groups are fundamental for controlling access to instances by allowing users to create inbound and outbound rules that limit traffic to and from instances based on specific addresses, ports, protocols, and even other security groups. Default security groups are created by Neutron for every project that allows all outbound communication and restrict inbound communication to instances in the same default security group. Subsequent security groups are locked down even further, allowing only outbound communication and not allowing any inbound traffic at all unless modified by the user.

Security group rules are implemented on the compute nodes and are triggered when traffic enters or leaves a virtual network interface belonging to an instance. Users are free to implement additional firewalls within the guest operating system, but may find managing rules in both places a bit cumbersome. Many organizations still utilize and rely on physical firewall devices to provide additional filtering at the edge of the network, which may mean coordination is required between users of the cloud and traditional security teams to ensure proper rules and actions are in place in all locations.

In the next chapter, we will look at implementing Role-Based Access Control (RBAC) within Neutron as a method of controlling access to networks between projects from an API perspective.

9 Role-Based Access Control

The **Role-Based Access Control (RBAC)** policy framework enables both operators and users to grant access to resources for specific projects or tenants. Prior to RBAC, Neutron applied an all-or-nothing approach to the sharing of networks across projects. If a network was marked as shared, it was shared with all projects. Access control policies built using the Neutron RBAC API allow operators and users to share certain network resources with one or more projects using a more granular approach.

As of the Pike release of OpenStack, access that can be granted using access control policies includes the following:

- Regular port creation permissions on networks
- Attaching router gateways to networks
- Binding **Quality of Service (QoS)** policy permissions to networks or ports

In this chapter, we will focus on the concept and implementation of role-based access control in Neutron, including the following:

- A brief introduction to role-based access control for networking resources
- Creating and managing access control policies
- Demonstrating the use of access control policies

QoS is an advanced networking topic that's not covered in this book.

Working with access control policies

The workflow for managing RBAC policies follows the standard Create, Read, Update, and Delete (CRUD) model that is used throughout the Neutron API. The OpenStack command-line interface can be used to manage access control policies. As of the Pike release of OpenStack, Horizon support for RBAC has not been implemented. However, resources shared via a policy can be utilized within the dashboard.

Managing access control policies in the CLI

From within the openstack command-line client, a number of commands can be used to manage access control policies. The primary commands associated with access control policy management that will be discussed in this chapter are listed in the following table:

Role-Based Access Control Commands	Description
<code>network rbac create</code>	Creates the network RBAC policy
<code>network rbac delete</code>	Deletes the network RBAC policy
<code>network rbac list</code>	Lists network RBAC policies
<code>network rbac set</code>	Sets network RBAC policy properties
<code>network rbac show</code>	Displays network RBAC policy details

Creating access control policies in the CLI

To create an access control policy, use the `openstack network rbac create` command as follows:

```
openstack network rbac create
--type <type> --action <action>
--target-project <target-project>
[--target-project-domain <target-project-domain>]
[--project <project>]
[--project-domain <project-domain>]
<rbac-object>
```

The `--type` argument allows you to specify the type of object the policy affects. Possible options include `network` or `qos_policy`. Only network policies are described in this chapter.

The `--action` argument allows you to specify the action for the access policy. Possible options include `access_as_external` or `access_as_shared`. The former allows a network to be used as an external network for routers for a subset of projects, while the latter allows a network to be used as a shared network for a subset of projects.

The `--target-project` argument specifies the project to which the access policy will be enforced.

The `--target-project-domain` argument is optional and allows you to specify the domain the target project belongs to.

The `--project` argument is optional and allows you to specify the owner of the access policy.

The `--project-domain` argument is optional and allows you to specify the domain the owner project belongs to.

The `rbac-object` argument specifies the name of the object that the policy applies to. This may be a network name or ID if the type is network, or a QOS policy ID if the type is `qos_policy`.

Deleting access control policies in the CLI

To delete an access control policy, use the `openstack network rbac delete` command and specify the ID of the RBAC object, as follows:

```
openstack network rbac delete <RBAC Policy> [<RBAC Policy> ...]
```

Multiple access policies can also be deleted simultaneously, as follows:

```
openstack network rbac delete <RBAC Policy ID1> <RBAC Policy ID2>
```

Neutron will successfully delete the access policy as long as another project is not utilizing a shared resource.

Listing access control policies in the CLI

To obtain a listing of access control policies, use the `openstack network rbac list` command as follows:

```
openstack network rbac list  
[--type <type>] [--action <action>]
```

The results can be filtered by type and action. The output returned includes the action, ID, shared object ID, and shared object type.

Showing the details of an access control policy in the CLI

To display the details of an access policy, use the `openstack network rbac show` command that's shown here:

```
openstack network rbac show <RBAC Policy>
```

Updating access control policies in the CLI

To update the attributes of an access policy, use the `openstack network rbac set` command, as follows:

```
openstack network rbac set  
[--target-project <target-project>]  
[--target-project-domain <target-project-domain>]  
<RBAC Policy>
```

This command allows you to change the target project and domain of an existing access control policy.

Applying RBAC policies to projects

Before an RBAC policy can be created, you need to have a network or QoS policy that you wish to share among a subset of projects. In the following sections, we will create projects, roles, users, and networks, and I will demonstrate how an access control policy limits the visibility and use of shared resources between the projects versus the default behavior of the shared attribute.

Creating projects and users

Using the openstack client, let's create three projects named ProjectA, ProjectB, and ProjectC, as shown here:

```
root@controller01:~# openstack project create --description "Project A RBAC Demo" ProjectA
```

Field	Value
description	Project A RBAC Demo
domain_id	default
enabled	True
id	b013329775694d5d884fd2ce31c4e22f
is_domain	False
name	ProjectA
parent_id	default

```
root@controller01:~# openstack project create --description "Project B RBAC Demo" ProjectB
```

Field	Value
description	Project B RBAC Demo
domain_id	default
enabled	True
id	a228b048632140e2a9f610884fd9e27e
is_domain	False
name	ProjectB
parent_id	default

```
root@controller01:~# openstack project create --description "Project C RBAC Demo" ProjectC
```

Field	Value
description	Project C RBAC Demo
domain_id	default
enabled	True
id	e9cd39dc02ea470fb89ed78929fbfe4d
is_domain	False
name	ProjectC
parent_id	default

Using the openstack client, create a role named `rbacdemo`. Then, create a user in each project and apply the `rbacdemo` role to each user:

```

root@controller01:~# openstack role create rbacdemo
+-----+-----+
| Field | Value |
+-----+-----+
| domain_id | None |
| id | a13e045e9d1a42f08086ced1ddd0a16a |
| name | rbacdemo |
+-----+-----+

root@controller01:~# openstack user create UserA --password secrete
+-----+-----+
| Field | Value |
+-----+-----+
| domain_id | default |
| enabled | True |
| id | 50373e07321b4233bd74bfba75ab0717 |
| name | UserA |
| options | {} |
| password_expires_at | None |
+-----+-----+

root@controller01:~# openstack role add --project ProjectA --user UserA rbacdemo
root@controller01:~# openstack user create UserB --password secrete
+-----+-----+
| Field | Value |
+-----+-----+
| domain_id | default |
| enabled | True |
| id | 468372bdd067435d88daf30b7e570a57 |
| name | UserB |
| options | {} |
| password_expires_at | None |
+-----+-----+

root@controller01:~# openstack role add --project ProjectB --user UserB rbacdemo
root@controller01:~# openstack user create UserC --password secrete
+-----+-----+
| Field | Value |
+-----+-----+
| domain_id | default |
| enabled | True |
| id | 9153a96cdb814d91b25a227fa0aada0a |
| name | UserC |
| options | {} |
| password_expires_at | None |
+-----+-----+

root@controller01:~# openstack role add --project ProjectC --user UserC rbacdemo

```


Creating a network to share

Using the openstack client, create a network named `MySemiSharedNetwork` and a subnet named `MySemiSharedSubnet` as the admin user. This network will be shared among some of the newly created projects:

```

root@controller01:~# source adminrc
root@controller01:~# openstack network create MySemiSharedNetwork

```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-04-05T13:04:50Z
description	
dns_domain	None
id	ec7e4e1e-cc54-4e1a-9e57-5b3e2f752c50
ipv4_address_scope	None
ipv6_address_scope	None
is_default	None
is_vlan_transparent	None
mtu	1450
name	MySemiSharedNetwork
port_security_enabled	True
project_id	9233b6b4f6a54386af63c0a7b8f043c2
provider:network_type	vxlan
provider:physical_network	None
provider:segmentation_id	36
qos_policy_id	None
revision_number	2
router:external	Internal
segments	None
shared	False
status	ACTIVE
subnets	
tags	
updated_at	2018-04-05T13:04:50Z

```

root@controller01:~# openstack --debug subnet create --subnet-range 192.168.99.0/24 --network MySemiSharedNetwork MySemiSharedSubnet

```

Field	Value
allocation_pools	192.168.99.2-192.168.99.254
cidr	192.168.99.0/24
created_at	2018-04-05T15:13:19Z
description	
dns_nameservers	
enable_dhcp	True
gateway_ip	192.168.99.1
host_routes	
id	f31a8c48-cce7-475f-8287-0366ca74af2d
ip_version	4
ipv6_address_mode	None
ipv6_ra_mode	None
name	MySemiSharedSubnet
network_id	ec7e4e1e-cc54-4e1a-9e57-5b3e2f752c50
project_id	9233b6b4f6a54386af63c0a7b8f043c2
revision_number	0
segment_id	None
service_types	
subnetpool_id	None
tags	
updated_at	2018-04-05T15:13:19Z
use_default_subnet_pool	None

In the network details, observe that the `shared` attribute of the network is set to `False`. In this state, the network can only be used by users associated with the owning project. In this case, this is the `admin` project. Setting it to `True` would make the network usable by all projects, which is something we'd like to avoid.

Creating a policy

Creating an RBAC policy requires the following four details:

- Policy type
- Target project
- Action to perform
- Resource

By default, the `MySemiSharedNetwork` network created in the last section is limited to the `admin` project that created it. Using the `openstack` client, create a new RBAC policy that shares the network with the `ProjectA` project by using the `access_as_shared` action, as shown here:

```
root@controller01:~# openstack network rbac create \  
> --type network \  
> --action access_as_shared \  
> --target-project ProjectA \  
> MySemiSharedNetwork
```

Field	Value
action	access_as_shared
id	ce451cdc-abd6-41e8-973d-74b31d8e8ed4
name	None
object_id	ec7e4e1e-cc54-4e1a-9e57-5b3e2f752c50
object_type	network
project_id	9233b6b4f6a54386af63c0a7b8f043c2
target_project_id	b013329775694d5d884fd2ce31c4e22f

The access control policy can be viewed with the `openstack network rbac list` and/or `show` commands that can be seen here:

```
root@controller01:~# openstack network rbac list
```

ID	Object Type	Object ID
ce451cdc-abd6-41e8-973d-74b31d8e8ed4	network	ec7e4e1e-cc54-4e1a-9e57-5b3e2f752c50

```
root@controller01:~# openstack network rbac show ce451cdc-abd6-41e8-973d-74b31d8e8ed4
```

Field	Value
action	access_as_shared
id	ce451cdc-abd6-41e8-973d-74b31d8e8ed4
name	None
object_id	ec7e4e1e-cc54-4e1a-9e57-5b3e2f752c50
object_type	network
project_id	9233b6b4f6a54386af63c0a7b8f043c2
target_project_id	b013329775694d5d884fd2ce31c4e22f

Viewing the policy in action

As UserA in the ProjectA project, the `openstack network list` command can be used to confirm that the network is available for use, like so:

```
root@controller01:~# openstack --os-project-name ProjectA --os-username UserA --os-password secrete network list
```

ID	Name	Subnets
ec7e4e1e-cc54-4e1a-9e57-5b3e2f752c50	MySemiSharedNetwork	f31a8c48-cce7-475f-8287-0366ca74af2d

Users in the ProjectB and ProjectC projects, however, cannot see the network:

```
root@controller01:~# openstack --os-project-name ProjectB --os-username UserB --os-password secrete network list
root@controller01:~# openstack --os-project-name ProjectC --os-username UserC --os-password secrete network list
```

Running the query as a user in the ProjectB and ProjectC projects returns no results, and the network will not be usable by those projects. For every project that a network object must be shared with, a separate policy must be created. The `openstack network rbac create` command can be used to create additional policies by specifying different projects but sharing the same network resource.

Creating policies for external networks

External networks are used to attach Neutron routers to the physical network and help provide routing in and out of a project via routes and floating IPs. When a network's external attribute is set to `True`, the network is shared among all projects. Access policies can be used to limit external networks to a subset of projects. The syntax for creating a policy for external networks is very similar to the policy demonstrated in the last section. The action, however, will change from `access_as_shared` to `access_as_external`.

In this example, an external network named `MySemiSharedExternalNetwork` using VLAN 31 has been created with the goal of sharing it with a subset of projects. Notice that the shared attribute is `False` and that the external attribute is `internal`, meaning that it can only be used for internal networks:

```

openstack network create \
  --provider-network-type vlan \
  --provider-physical-network physnet1 \
  --provider-segment 31 \
  MySemiSharedExternalNetwork

```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-04-09T13:32:10Z
description	
dns_domain	None
id	b41d0d64-9c32-48c3-aa05-604697c7d38f
ipv4_address_scope	None
ipv6_address_scope	None
is_default	None
is_vlan_transparent	None
mtu	1500
name	MySemiSharedExternalNetwork
port_security_enabled	True
project_id	9233b6b4f6a54386af63c0a7b8f043c2
provider:network_type	vlan
provider:physical_network	physnet1
provider:segmentation_id	31
qos_policy_id	None
revision_number	2
router:external	Internal
segments	None
shared	False
status	ACTIVE
subnets	
tags	
updated_at	2018-04-09T13:32:11Z

As a result, the network will not be listed when running `openstack network list --external`, and is ineligible for use as an external network:

```
root@controller01:~# openstack network list --external
root@controller01:~#
```

In this example, the admin user has created an access policy that shares the network as an external network with the ProjectB project:

```
root@controller01:~# openstack network rbac create \
> --type network \
> --action access_as_external \
> --target-project ProjectB \
> MySemiSharedExternalNetwork
```

Field	Value
action	access_as_external
id	a4c124b3-4210-4a6f-9fc4-2baf6ae6e372
name	None
object_id	b41d0d64-9c32-48c3-aa05-604697c7d38f
object_type	network
project_id	9233b6b4f6a54386af63c0a7b8f043c2
target_project_id	a228b048632140e2a9f610884fd9e27e

As UserB in the ProjectB project, we can see the external network using the `openstack network list --external` command. The network will reflect its external status:

```
root@controller01:~# openstack --os-project-name ProjectB --os-username UserB --os-password secrete network list --external
```

ID	Name	Subnets
b41d0d64-9c32-48c3-aa05-604697c7d38f	MySemiSharedExternalNetwork	

```
root@controller01:~# openstack --os-project-name ProjectB --os-username UserB --os-password secrete network show MySemiSharedExternalNetwork
```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-04-09T13:32:10Z
description	
dns_domain	None
id	b41d0d64-9c32-48c3-aa05-604697c7d38f
ipv4_address_scope	None
ipv6_address_scope	None
is_default	None
is_vlan_transparent	None
mtu	1500
name	MySemiSharedExternalNetwork
port_security_enabled	True
project_id	9233b6b4f6a54386af63c0a7b8f043c2
provider:network_type	None
provider:physical_network	None
provider:segmentation_id	None
qos_policy_id	None
revision_number	4
router:external	External
segments	None
shared	False
status	ACTIVE
subnets	
tags	
updated_at	2018-04-09T13:39:55Z

Without a `shared_as_external` access policy in place, no other project will be able to see or use the network as an external network.

Summary

Access control policies created and managed by the role-based access control framework allow operators and users to provide access to certain network resources to individual projects rather than all projects. This ability opens up network design options that were not available in the past and provides a more secure approach to network resource sharing.

In the next chapter, we will explore standalone Neutron routers and their involvement in providing self-service networking to projects. In addition, we'll take a look at the configuration and use of floating IPs to provide external connectivity to instances. The configuration of highly available and distributed virtual routers will be covered in later chapters.

10

Creating Standalone Routers with Neutron

Neutron enables users to build routers that provide connectivity between networks created by users and external networks. In a reference implementation, the Neutron L3 agent provides IP routing and network address translation for virtual machine instances within the cloud by utilizing network namespaces to provide isolated routing instances. By creating networks and attaching them to routers, users can expose connected virtual machine instances and their applications to the internet.

Prior to the Juno release of OpenStack, users were limited to building standalone routers that acted as single points of failure in the network stack. Since the advent of distributed virtual routers in Juno and beyond, standalone routers are now referred to as legacy routers. While the preference may be to provide resiliency in the form of highly-available or distributed virtual routers, standalone routers provide the simplest implementation of the three options.

In previous chapters, we discovered the difference between provider and self-service project networks and demonstrated the process of booting an instance and connecting it to the network. In this chapter, we will work through the following:

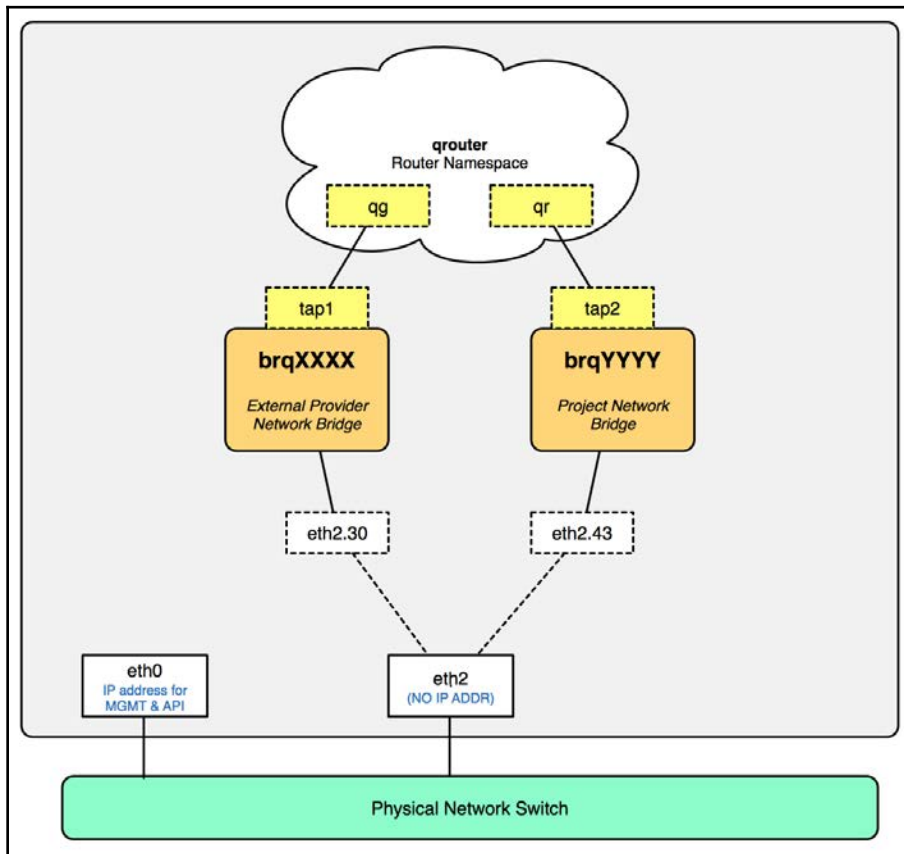
- Installing and configuring the L3 agent
- Creating an external provider network
- Creating a standalone router in the CLI and Horizon dashboard
- Attaching a router to both external and tenant networks
- Booting instances
- Demonstrating instance and namespace connectivity using Linux bridges
- Demonstrating SNAT and DNAT functionality provided by floating IPs

Routing traffic in the cloud

In a reference implementation, virtual routers created in Neutron exist as network namespaces that reside on nodes running the Neutron L3 agent service. A virtual router is often connected to a single external provider network and one or more project networks. The router interfaces connected to those networks can be identified as follows:

- **qg**: Gateway interface
- **qr**: Router interface

Neutron routers are responsible for providing inbound and outbound connectivity to and from project networks through the use of Network Address Translation, or NAT. The following diagram shows how a router namespace may be connected to multiple bridges in a Linux bridge-based implementation:



The preceding diagram demonstrates a Neutron router connected to multiple bridges in a Linux bridge-based implementation. In an Open vSwitch-based implementation, the router's interfaces are connected directly to the integration bridge. Traffic from project networks is routed in through **qr** interfaces and out the **qg** interface onto the external network. Routing tables within the namespace dictate how traffic is routed, and iptables rules dictate how traffic is translated, if necessary.

More on creating and configuring standalone Neutron routers, along with examples on how they are connected to the network and provide connectivity to instances, can be found later on in this chapter.

Installing and configuring the Neutron L3 agent

To install the Neutron L3 agent, run the following command on the `controller01` node:

```
# apt install neutron-l3-agent
```

Neutron stores the L3 agent configuration in the `/etc/neutron/l3_agent.ini` file. The most common configuration options will be covered in the following subsections.

Defining an interface driver

The Neutron L3 agent must be configured to use an interface driver that corresponds to the chosen mechanism driver. In a reference implementation, that can be either the Linux bridge or Open vSwitch drivers. In this environment, the linux bridge driver will be installed on `controller01`.

On the `controller01` node, update the Neutron L3 agent configuration file at `/etc/neutron/l3_agent.ini` and specify the following Linux bridge interface driver for this particular environment:

```
[DEFAULT]
...
interface_driver = linuxbridge
```

For your reference, the following can be used when the `network` node hosting routers is configured for Open vSwitch:

```
[DEFAULT]
...
interface_driver = openvswitch
```

Enabling the metadata proxy

When Neutron routers are used as the gateway for instances, requests for metadata are proxied by the router rather than the DHCP server and are forwarded to the Nova metadata service. This feature is enabled by default and can be disabled by setting the `enable_metadata_proxy` value to `False` in the `l3_agent.ini` configuration file and uncommenting the line. For this environment, leave the setting at its default `True` value.



If the metadata proxy is disabled, users may only be able to obtain metadata using config-drive.

Setting the agent mode

By default, the Neutron L3 agent works in legacy mode, which means that the L3 agent is deployed on a centralized node responsible for networking services. The default value for `agent_mode` is `legacy`, which shall remain unchanged for the remainder of this chapter.

Enabling the router service plugin

The router service plugin must be enabled before Neutron will accept API commands related to Neutron routers. On the `controller01` node, update the Neutron API server configuration file at `/etc/neutron/neutron.conf` and append `router` to the list of service plugins, as shown here:

```
service_plugins = router
```



The `service_plugins` configuration option may be disabled. Remove any leading hash (`#`) to enable the option.

Enabling router management in the dashboard

The Horizon dashboard can be used to manage routers, but the option must be enabled first.

On the `controller01` node, edit the OpenStack dashboard configuration file at `/etc/openstack-dashboard/local_settings.py` and navigate to the `OPENSTACK_NEUTRON_NETWORK` configuration option. Change the `enable_router` dictionary value from `False` to `True`, as shown here:

```
OPENSTACK_NEUTRON_NETWORK = {  
    'enable_router': True,  
    'enable_quotas': False,  
    'enable_ipv6': False,  
    'enable_distributed_router': False,  
    'enable_ha_router': False,  
    'enable_lb': False,  
    'enable_firewall': False,  
    'enable_vpn': False,  
    'enable_fip_topology_check': False,  
}
```

Close the file and proceed with the next section to restart services.

Restarting services

After making changes to the configuration of the Neutron L3 agent and API service, issue the following commands on the `controller01` node to restart the respective services:

```
# systemctl restart neutron-l3-agent neutron-server apache2
```

Verify that the agent is running:

```
# systemctl status neutron-l3-agent
```

The service should return a similar output to the following and be in an active (running) state:

```

root@controller01:~# systemctl status neutron-l3-agent
● neutron-l3-agent.service - OpenStack Neutron L3 agent
   Loaded: loaded (/lib/systemd/system/neutron-l3-agent.service; enabled; vendor preset: enabled)
   Active: active (running) since Mon 2018-07-16 16:17:40 UTC; 18s ago
     Process: 21902 ExecStartPre=/bin/chown neutron:adm /var/log/neutron (code=exited, status=0/SUCCESS)
     Process: 21897 ExecStartPre=/bin/chown neutron:neutron /var/lock/neutron /var/lib/neutron (code=exited, status=0/SUCCESS)
     Process: 21886 ExecStartPre=/bin/mkdir -p /var/lock/neutron /var/log/neutron /var/lib/neutron (code=exited, status=0/SUCCESS)
  Main PID: 21915 (neutron-l3-agent)
    Tasks: 1
   Memory: 106.8M
      CPU: 1.880s
   CGroup: /system.slice/neutron-l3-agent.service
           └─21915 /usr/bin/python /usr/bin/neutron-l3-agent --config-file=/etc/neutron/neutron.conf
             --config-file=/etc/neutron/l3_agent.ini
             --config-file=/etc/neutron/fwaas_driver.ini --log-file=/var/log/neutron/neutron-l3-agent

```

If the service remains stopped, troubleshoot any errors that may be indicated in the `/var/log/neutron/l3-agent.log` file.

Router management in the CLI

Neutron offers a number of commands that can be used to create and manage routers. The primary commands associated with router management include the following:

Router Management Commands	Description
<code>router create</code>	Creates a router
<code>router delete</code>	Deletes a router(s)
<code>router set</code>	Sets a router gateway and other properties
<code>router unset</code>	Unsets a router gateway and other properties
<code>router show</code>	Displays router details
<code>router list</code>	Lists routers
<code>router add port</code>	Adds an interface to a router using an existing port
<code>router add subnet</code>	Adds an interface to a router using an existing subnet
<code>router remove port</code>	Removes an interface from a router using a corresponding port ID
<code>router remove subnet</code>	Removes an interface from a router using a corresponding subnet ID
<code>network agent list</code>	Lists all network agents

Creating routers in the CLI

Routers in Neutron are associated with projects and can only be managed by users associated with the project. Unlike networks, routers cannot be shared among projects. However, shared networks can be attached to routers and potentially route traffic between different projects. Users with the admin role can associate routers with other projects during the router creation process.

To create a standalone router, use the `openstack router create` command shown here:

```
openstack router create
[--enable | --disable]
[--project <project>]
[--project-domain <project-domain>]
<name>
```

The router will be created without any interfaces attached and will immediately be scheduled to an L3 agent. A corresponding network namespace should be visible on the node hosting the respective L3 agent and can be found using the `ip netnslist` command.

Listing routers in the CLI

To display a list of existing routers, use the `openstack router list` command shown here:

```
openstack router list
[--name <name>] [--enable | --disable] [--long]
[--project <project>]
[--project-domain <project-domain>]
[--agent <agent-id>] [--tags <tag>[,<tag>,...]]
[--any-tags <tag>[,<tag>,...]]
[--not-tags <tag>[,<tag>,...]]
[--not-any-tags <tag>[,<tag>,...]]
```

Filters based on name, project, admin state, scheduled agent, and tags can be used to narrow returned results.



Users will only see routers associated with their project. When executed as a user with the admin role, Neutron will return a listing of all routers across all projects unless a project ID is specified.

Displaying router attributes in the CLI

To display the attributes of a router, use the `openstack router show` command shown here:

```
openstack router show <router>
```

Among the output returned is the admin state, the external network, the SNAT state, and project ID associated with the router. Two additional attributes, `distributed` and `HA`, are used to identify the router as being distributed or highly-available. For standalone routers, both attributes will be set to `False`.

Updating router attributes in the CLI

To update the attributes of a router, use the `openstack router set` or `openstack router unset` commands shown here:

```
openstack router set
[--name <name>] [--description <description>]
[--enable | --disable]
[--distributed | --centralized]
[--route destination=<subnet>,gateway=<ip-address>]
[--no-route] [--ha | --no-ha]
[--external-gateway <network>]
[--fixed-ip subnet=<subnet>,ip-address=<ip-address>]
[--enable-snat | --disable-snat] [--tag <tag>]
[--no-tag]
<router>
```

```
openstack router unset
[--route destination=<subnet>,gateway=<ip-address>]
[--external-gateway] [--tag <tag> | --all-tag]
<router>
```

Working with router interfaces in the CLI

Standalone Neutron routers have two types of interfaces: gateway and internal. The gateway interface of a Neutron router is analogous to the WAN interface of a physical router. It is the interface connected to an upstream device that provides connectivity to external resources. The internal interfaces of Neutron routers are analogous to the LAN interfaces of physical routers. Internal interfaces are connected to project networks and often serve as the gateway or next hop for instances in those networks.

Attaching internal interfaces to routers

To create an internal router interface and attach it to a subnet, use the `openstack router add subnet` command shown here:

```
openstack router add subnet <router> <subnet>
```

The `subnet` argument represents a subnet name or ID to be attached to the router. Neutron will assign the gateway IP of the subnet to the router when creating the `qr` interface and corresponding port.

To attach an existing port directly to the router, use the `openstack router add port` command shown here:

```
openstack router subnet add <router> <port>
```

The `port` keyword represents a port name or ID to be attached to the router.

A Neutron router can only have one interface in a given subnet, but can be attached to multiple subnets simultaneously. The L3 agent is responsible for connecting interfaces within the router namespace to the proper bridges on the host.



In Neutron, a network may contain multiple subnets. A router must be connected to all subnets in a network to properly route traffic for that network. Be sure not to attach an external network using this process, as traffic may be negatively impacted!

Attaching a gateway interface to a router

The external interface of a Neutron router is referred to as the gateway interface. A router is limited to a single gateway interface. To be eligible for use as an external network that can be used for router gateway interfaces, a provider network must have its router's external attribute set to `True` or `External`.

To attach a gateway interface to a router, use the `openstack router set` command shown here:

```
openstack router set --external-gateway <network> <router>
```

The `<network>` argument represents a network name or ID to be attached to the router as the gateway network. Neutron will assign an IP address from the external network to the router. By default, Neutron routers perform source network address translation, or SNAT, on all traffic from instances that lack floating IPs. NAT and SNAT will be covered in more detail later in this chapter. To disable this functionality, use the `--disable-snat` argument when using the `openstack router set` command.

Listing interfaces attached to routers

To list the interfaces attached to a router, use the `openstack port list` command shown here:

```
openstack port list --router <router>
```

The `<router>` argument represents the router name or ID.

Deleting internal interfaces

To delete an internal interface from a router, use the `openstack router remove port` or `openstack router remove subnet` commands shown here:

```
openstack router remove port <router> <port>
openstack router remove subnet <router> <subnet>
```

The `<port>` argument represents the name or ID of a particular port to be removed from the router, while the `<subnet>` argument represents the name or ID of a subnet to be removed from the router. In either case, deleting an interface based on port or subnet results in the respective Neutron port being removed from the database.

Clearing the gateway interface

Gateway interfaces cannot be removed from a router using the `openstack router remove` commands. Instead, the `openstack router unset` command must be used.

To clear the gateway of a router, use the `openstack router unset` command shown here:

```
openstack router unset --external-gateway <router>
```


Neutron performs checks that will prohibit the clearing of a gateway interface in the event floating IPs are associated with the router.

Deleting routers in the CLI

To delete a router, use the `openstack router delete` command and specify the name or ID of the router:

```
openstack router delete <router> [<router> ...]
```

Multiple routers can also be deleted simultaneously, as follows:

```
openstack router delete <router1> <router2>
```

Neutron will successfully delete the routers as long as all connected ports or subnets have been removed and all floating IPs have been disassociated or deleted.

Network address translation

Network address translation, or NAT, is a networking concept that was developed in the early 1990s in response to the rapid depletion of IP addresses throughout the world. Prior to NAT, every host connected to the internet had a unique IP address.

Standalone routers support two types of NAT:

- one-to-one
- many-to-one

A one-to-one NAT is a method in which one IP address is directly mapped to another. Commonly referred to as a static NAT, a one-to-one NAT is often used to map a unique public address to a privately addressed host. Floating IPs utilize one-to-one NAT concepts.

A many-to-one NAT is a method in which multiple addresses are mapped to a single address. A many-to-one NAT employs the use of port address translation, or PAT. Neutron uses PAT to provide external access to instances behind the router when floating IPs are not assigned.

For more information on network address translation, please visit the following Wikipedia page at http://en.wikipedia.org/wiki/Network_address_translation.

Floating IP addresses

Self-service project networks, when attached to a Neutron router, often utilize the router as their default gateway. By default, when a router receives traffic from an instance and routes it upstream, the router performs a port address translation and modifies the source address of the packet to appear as its own external interface address. When the translation occurs, the ephemeral source port is mapped to the original client address in a table that is referred to when the response packet is received. This ensures that the packet can be routed upstream and returned to the router, where the packet is modified and returned to the instance that initiated the connection. Neutron refers to this type of behavior as source NAT.

When users require direct inbound access to instances, a floating IP address can be utilized. A floating IP address in OpenStack is a one-to-one static NAT that maps an external address from an external network to an internal address in a project network. This method of NAT allows instances to be accessible from remote networks such as the internet. Floating IP addresses are configured on the external interface of the router that serves as the gateway for the instance, which is then responsible for modifying both the source and destination address of packets depending on their direction.

Floating IP management

The OpenStack command-line client offers a number of commands that can be used to create and manage floating IPs. The primary commands associated with floating IPs include the following:

Floating IP Commands	Description
<code>floating ip create</code>	Creates a floating IP
<code>floating ip delete</code>	Deletes a floating IP
<code>floating ip list</code>	Lists floating IP
<code>floating ip show</code>	Displays floating IP details
<code>floating ip set</code>	Sets floating IP properties
<code>floating ip unset</code>	Unsets floating IP properties
<code>floating ip pool list</code>	Lists pools of floating IP addresses
<code>server add floating ip</code>	Adds a floating IP address to a server
<code>server remove floating ip</code>	Removes a floating IP address from a server

Creating floating IPs in the CLI

If you recall from previous chapters, IP addresses are not assigned directly to instances. Instead, an IP address is associated with a Neutron port, and that port is logically mapped to an instance or other network resource.

When a floating IP is created, it will not be functional until it is associated with a Neutron port. To create a floating IP from within the CLI, use the `openstack floating ip create` command shown here:

```
openstack floating ip create
[--subnet <subnet>]
[--port <port>]
[--floating-ip-address <ip-address>]
[--fixed-ip-address <ip-address>]
[--description <description>]
[--project <project>]
[--project-domain <project-domain>]
<network>
```

Floating IP addresses can only be used within the project in which they were created. The `--project` argument can be used by an administrator so that they are able to specify the project associated with the floating IP.

The `--port` argument is optional and is used to specify the port to be associated with the floating IP upon creation.

Because a port can have multiple IP addresses associated with it, it may be necessary to define a specific fixed IP to associate the floating IP with. Use the `--fixed-ip-address` argument to specify the fixed IP address that should be associated with the floating IP.

In previous releases of OpenStack, floating IPs were automatically assigned from the allocation pool of the external network. This behavior made it difficult for users who required particular addresses, especially when upstream NATs or firewall rules were already in place that specified certain floating IP addresses. Beginning with Kilo, it is possible to create a floating IP using a specified address as long as it is available and not associated with another project. With this feature, users can specify a particular floating IP address and avoid modifying external systems.

Use the `--floating-ip-address` argument to specify a particular address from the external network for use as a floating IP.

Associating floating IPs with ports in the CLI

Once a floating IP has been created, it is available for use by any user within the project that created it. To associate a floating IP with an instance, it is first necessary to determine the Neutron port that is associated with the fixed IP of the instance.

The port ID associated with the fixed IP address of an instance can be determined in a couple of different ways. The quickest way may be to use the `openstack port list` command with a filter that can narrow down ports per instance.

For example, the ports of an instance whose ID is `3d577137-9658-4226-906e-88d3117e497e` can be determined in the following way:

```
root@controller01:~# openstack port list --server 3d577137-9658-4226-906e-88d3117e497e
```

ID	Name	MAC Address	Fixed IP Addresses	Status
be8df271-2892-49a1-899e-ef95c03bd0dd		fa:16:3e:e2:13:56	ip_address='192.168.206.11', subnet_id='02013907-1a5f-493f-8ece-ee1ee9e44afb'	ACTIVE

Once the port ID has been determined, use the `openstack floating ip set` command to associate the floating IP with the port:

```
openstack floating ip set --port <port>
[--fixed-ip-address <ip-address>]
<floating-ip>
```

Neutron uses the subnet ID of the specified port to determine the router in which to configure the floating IP address and respective NAT rules. The logic involved means that no more than one standalone router should be attached to a project network at any given time when floating IPs are used, otherwise unexpected results may occur.

Listing floating IPs in the CLI

To determine the association of floating IPs to Neutron ports and addresses, use the `openstack floating ip list` command shown here:

```
openstack floating ip list
[--network <network>] [--port <port>]
[--fixed-ip-address <ip-address>] [--long]
[--status <status>] [--project <project>]
[--project-domain <project-domain>]
[--router <router>]
```

The output returned includes the floating IP ID, fixed IP address, floating IP address, and port ID associated with the floating IP. All arguments are optional and help with filtering results.

Displaying floating IP attributes in the CLI

To display the attributes of a floating IP in the CLI, use the `openstack floating ip show` command shown here:

```
openstack floating ip show <floating-ip>
```

The output returned includes the floating IP address and the associated external network, fixed IP address, port, project, and router IDs.

Disassociating floating IPs in the CLI

To disassociate a floating IP from a port, use the `openstack floating ip unset` command shown here:

```
openstack floating ip unset [--port] <floating-ip>
```

Disassociating a floating IP from a port makes the floating IP available for use by other users within the project.

Deleting floating IPs in the CLI

To delete a floating IP, use the `openstack floating ip delete` command shown here:

```
openstack floating ip delete <floating-ip> [<floating-ip> ...]
```

Deleting a floating IP returns the IP address to the external network allocation pool where it can be allocated to other projects and used by other network resources, including routers and floating IPs.

Demonstrating traffic flow from an instance to the internet

This section is dedicated to a walkthrough that leverages fundamental Neutron concepts that have been discussed in this book so far. I will demonstrate the process of creating and connecting standalone Neutron routers to both project and external provider networks to provide network connectivity to instances.

A VLAN-based provider network will be created and used as an external gateway network for the Neutron router, while a VLAN-based project network will be created and used by instances. A Neutron router will be created and used to route traffic from instances in the project network to the internet, and floating IPs will be created and used to provide direct connectivity to instances.

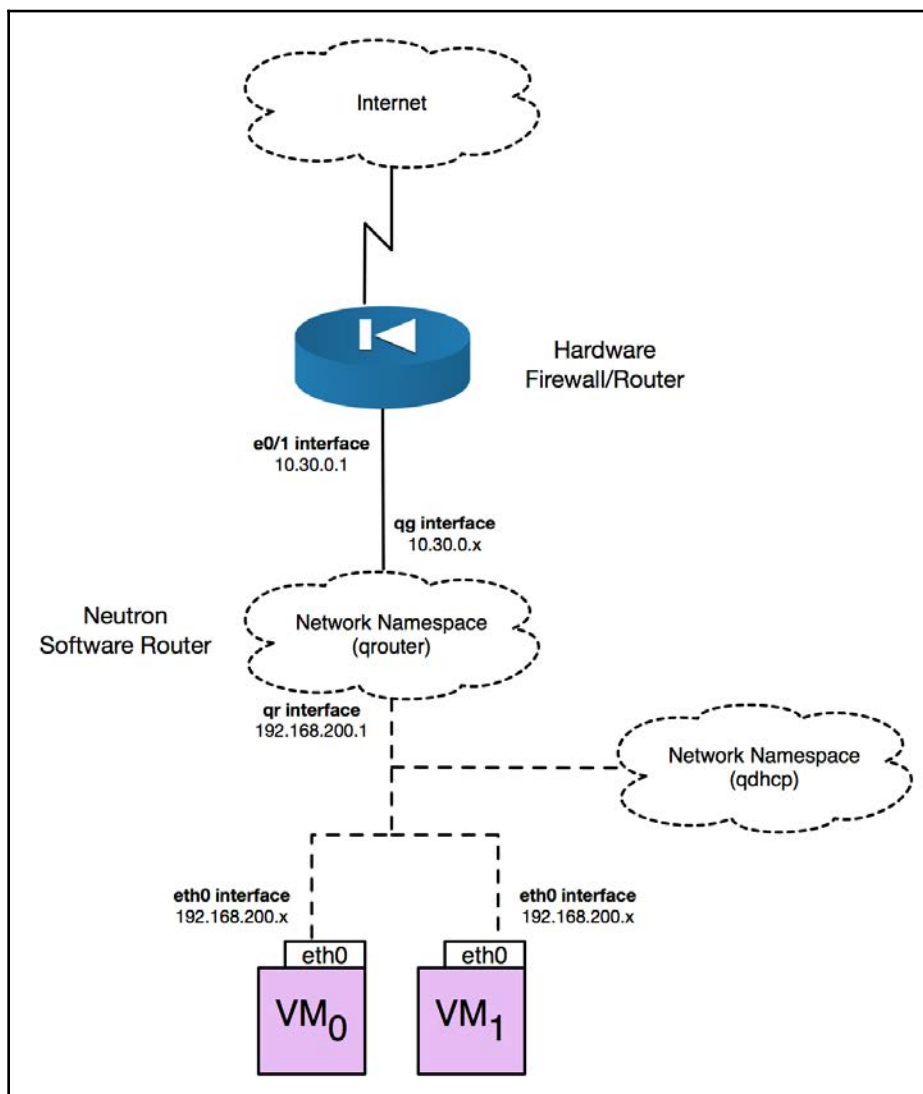
Setting the foundation

In this demonstration, a Cisco Adaptive Security Appliance (ASA) serves as the physical network gateway device and is connected to the internet. The following networks will be utilized:

VLAN name	VLAN ID	Network
GATEWAY_NET	30	10.30.0.0/24
PROJECT_NET	auto	192.168.200.0/24

An inside interface of the Cisco ASA has been configured with an IP address of 10.30.0.1/24 on VLAN 30 and will serve as the gateway for the external provider network created in the upcoming section.

The following diagram is the logical diagram of the network to be built as part of this demonstration:



In the preceding diagram, a Cisco ASA serves as the external network device in front of the OpenStack cloud.

Creating an external provider network

In order to provide instances with external connectivity, a Neutron router must be connected to a provider network eligible for use as an external network.

Using the `openstack network create` command, create a provider network in the admin project with the following attributes:

- Name: GATEWAY_NET
- Type: VLAN
- Segmentation ID: 30
- Physical Network: physnet1
- External: True

The following screenshot displays the resulting output of the `openstack network create` command:

```

root@controller01:~# openstack network create \
> --external \
> --provider-network-type vlan \
> --provider-physical-network physnet1 \
> --provider-segment 30 \
> GATEWAY_NET

```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-04-12T13:40:36Z
description	
dns_domain	None
id	758070f9-ecaf-4d2a-aa49-e119ce7943f6
ipv4_address_scope	None
ipv6_address_scope	None
is_default	None
is_vlan_transparent	None
mtu	1500
name	GATEWAY_NET
port_security_enabled	True
project_id	9233b6b4f6a54386af63c0a7b8f043c2
provider:network_type	vlan
provider:physical_network	physnet1
provider:segmentation_id	30
qos_policy_id	None
revision_number	3
router:external	External
segments	None
shared	False
status	ACTIVE
subnets	
tags	
updated_at	2018-04-12T13:40:36Z

Using the `openstack subnet create` command, create a subnet with the following attributes:

- Name: GATEWAY_SUBNET
- Network: 10.30.0.0
- Subnet Mask: 255.255.255.0
- Gateway: 10.30.0.1
- DHCP: Disabled
- Allocation Pool: 10.30.0.100 - 10.30.0.254

The following screenshot displays the resulting output of the `openstack subnet create` command:

```

root@controller01:~# openstack subnet create \
> --subnet-range 10.30.0.0/24 \
> --no-dhcp \
> --network GATEWAY_NET \
> --allocation-pool start=10.30.0.100,end=10.30.0.254 \
> GATEWAY_SUBNET
+-----+
| Field                                | Value                                |
+-----+-----+
| allocation_pools                     | 10.30.0.100-10.30.0.254            |
| cidr                                 | 10.30.0.0/24                       |
| created_at                           | 2018-04-12T13:44:44Z               |
| description                           |                                     |
| dns_nameservers                       |                                     |
| enable_dhcp                           | False                               |
| gateway_ip                           | 10.30.0.1                           |
| host_routes                           |                                     |
| id                                    | 0af5a767-fce0-4c4d-9df2-1aedbd259405 |
| ip_version                           | 4                                    |
| ipv6_address_mode                     | None                                 |
| ipv6_ra_mode                           | None                                 |
| name                                  | GATEWAY_SUBNET                      |
| network_id                            | 758070f9-ecaf-4d2a-aa49-e119ce7943f6 |
| project_id                            | 9233b6b4f6a54386af63c0a7b8f043c2   |
| revision_number                       | 0                                    |
| segment_id                            | None                                 |
| service_types                         |                                     |
| subnetpool_id                         | None                                 |
| tags                                  |                                     |
| updated_at                            | 2018-04-12T13:44:44Z               |
| use_default_subnet_pool                | None                                 |
+-----+-----+

```

Creating a Neutron router

Create a router using the `openstack router create` command with the following attribute:

- Name: MyLegacyRouter

The following screenshot displays the resulting output of the `openstack router create` command:

```
root@controller01:~# openstack router create MyLegacyRouter
```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-04-12T13:57:14Z
description	
distributed	False
external_gateway_info	None
flavor_id	None
ha	False
id	9ef2eed-4a55-4f64-b8be-4b07a43ec373
name	MyLegacyRouter
project_id	9233b6b4f6a54386af63c0a7b8f043c2
revision_number	None
routes	
status	ACTIVE
tags	
updated_at	2018-04-12T13:57:15Z

Attaching the router to an external network

When attaching a Neutron router an external network, the respective network must have its `router:external` attribute set to `True` to be eligible for use as an external network. Otherwise, the command will fail. The `GATEWAY_NETWORK` created previously meets this requirement.

Using the `openstack router set` command, attach the router `MyLegacyRouter` to the `GATEWAY_NET` network:

```
openstack router set --external-gateway GATEWAY_NET MyLegacyRouter
```

No output will be returned upon successful completion of the command. Using the `openstack port list` command, determine the external IP address of the router:

```
root@controller01:~# openstack port list --router MyLegacyRouter
```

ID	Name	MAC Address	Fixed IP Addresses	Status
73e65609-f0ea-4f4b-9581-42fe4b7df8d8		fa:16:3e:8c:a0:3f	ip_address='10.30.0.106', subnet_id='0af5a767-fce0-4c4d-9df2-1aedbd259405'	ACTIVE

In this example, the IP address assigned to the router's external interface is `10.30.0.106`.

Identifying the L3 agent and namespace

Once the gateway interface has been added, the router will be scheduled to an eligible L3 agent. Using the `openstack network agent list` command, you can determine which L3 agent the router was scheduled to:

```
root@controller01:~# openstack network agent list --router MyLegacyRouter
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
9c75984a-2cfa-4db6-afc6-b7e1e4901bbd	L3 agent	controller01	nova	:~)	UP	neutron-l3-agent

In this example, the router was scheduled to the `controller01` node. In an environment running multiple L3 agents, a standalone router can be scheduled to any one of the agents, but will not be scheduled to more than one at any given time.

The L3 agent is responsible for creating a network namespace that acts as the virtual router. For easy identification, the name of the namespace incorporates the router's ID. The `ip netns` command can be used to list all network namespaces on a node:

```
root@controller01:~# ip netns
qrouter-9ef2eed-4a55-4f64-b8be-4b07a43ec373 (id: 3)
qdhcp-ec7e4e1e-cc54-4e1a-9e57-5b3e2f752c50 (id: 2)
qdhcp-7745a4a9-68a4-444e-a5ff-f9439e3aac6c (id: 1)
qdhcp-03327707-c369-4bd7-bd71-a42d9bcf49b8 (id: 0)
```

Inside the respective `qrouter namespace`, you will find an interface with a prefix of `qg`. The `qg` interface is the gateway, or external, interface of the router. Neutron automatically provisions an IP address to the `qg` interface from the allocation pool of the external network's subnet:

```
root@controller01:~# ip netns exec qrouter-9ef2eed-4a55-4f64-b8be-4b07a43ec373 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
2: qg-73e65609-f0@if16: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:8c:a0:3f brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.30.0.106/24 brd 10.30.0.255 scope global qg-73e65609-f0
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe8c:a03f/64 scope link
        valid_lft forever preferred_lft forever
```

In the preceding screenshot, the IP address **10.30.0.106** was automatically configured on the external interface inside the namespace.

When using the Open vSwitch interface driver, the `qg interface` is connected directly to the integration bridge. When using the Linux bridge interface driver, as in this example, the `qg` interface is one end of a veth pair whose other end is connected to a Linux bridge on the host.

Using `ethtool`, we can determine the peer index of the corresponding interface on the host. This can be useful in troubleshooting connectivity issues in and out of network namespaces:

```
root@controller01:~# ip netns exec qrouter-9ef2eed-4a55-4f64-b8be-4b07a43ec373 ethtool -S qg-73e65609-f0
NIC statistics:
    peer_ifindex: 16
```

Using `ip link show` on the host, the corresponding interface (peer index 16) can be found by searching for the index on the controller:

```
root@controller01:~# ip link show | grep 16: -A1
16: tap73e65609-f0@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master brq758070f9-ec state UP mode DEFAULT group default qlen 1000
    link/ether d2:47:a1:98:1a:24 brd ff:ff:ff:ff:ff:ff link-netnsid 3
```

The output conveniently reveals the corresponding network namespace using the `link-netnsid` identifier. In this example, the peer interface resides in the network namespace with an ID of 3, otherwise known as `qrouter-9ef2eed-4a55-4f64-b8be-4b07a43ec373`.



The link-netnsid ID from `ip link show` should correspond to a namespace provided in the output of the `ip netns list` command.

When using the Linux bridge interface driver, the veth interface is connected to a bridge that corresponds to the external network shown here:

root@controller01:~# brctl show			
bridge name	bridge id	STP enabled	interfaces
brq3f05db4d-c4	8000.000c291124cb	no	eth2.42 tapcfa2b6a1-d2
brq758070f9-ec	8000.000c291124cb	no	eth2.30 tap73e65609-f0
Neutron External Network ID			Router External Port ID
brqd8428f56-ba	8000.000c291124cb	no	eth2.41 tap9d3d1ae5-8c
brqdb527151-20	8000.000c291124cb	no	eth2.40 tapd5a55dff-9e



For easy identification, the bridge name includes the first ten characters of the Neutron network ID. In addition, each end of the veth pair includes the first ten characters of the port ID associated with the interface.

The namespace is able to communicate with other devices in the same subnet through the bridge. The other interface in the bridge, `eth2.30`, tags traffic as `VLAN 30` as it exits the bridge and out physical interface `eth2`.

Observe the route table within the namespace. The default gateway address corresponds to the address defined in the external provider subnet's `gateway_ip` attribute. In this case, it's 10.30.0.1:

```
root@controller01:~# ip netns exec qrouter-9ef2eeed-4a55-4f64-b8be-4b07a43ec373 ip route show
default via 10.30.0.1 dev qg-73e65609-f0
10.30.0.0/24 dev qg-73e65609-f0 proto kernel scope link src 10.30.0.106
```

In this example environment, 10.30.0.1 is configured on the Cisco ASA and will serve as the next hop gateway for outbound traffic from the Neutron router.

Testing gateway connectivity

To test external connectivity from the Neutron router, ping the edge gateway device from within the router namespace:

```
root@controller01:~# ip netns exec qrouter-9ef2eeed-4a55-4f64-b8be-4b07a43ec373 ping 10.30.0.1 -c 5
PING 10.30.0.1 (10.30.0.1) 56(84) bytes of data.
64 bytes from 10.30.0.1: icmp_seq=1 ttl=255 time=0.582 ms
64 bytes from 10.30.0.1: icmp_seq=2 ttl=255 time=0.738 ms
64 bytes from 10.30.0.1: icmp_seq=3 ttl=255 time=0.604 ms
64 bytes from 10.30.0.1: icmp_seq=4 ttl=255 time=0.623 ms
64 bytes from 10.30.0.1: icmp_seq=5 ttl=255 time=0.687 ms

--- 10.30.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4072ms
rtt min/avg/max/mdev = 0.582/0.646/0.738/0.065 ms
```

Successful ping attempts from the router namespace to the physical gateway device demonstrate proper external VLAN configuration on both physical and virtual networking components.

Creating an internal network

Within the `admin project`, create an internal network for instances. In this demonstration, a network will be created with the following attribute:

- Name: `PROJECT_NET`

The following screenshot demonstrates the resulting output of the `openstack network create` command:

```
root@controller01:~# openstack network create PROJECT_NET
```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-04-13T16:02:38Z
description	
dns_domain	None
id	ce1fe7cd-7b01-4d58-b1c8-414886b7d8f2
ipv4_address_scope	None
ipv6_address_scope	None
is_default	False
is_vlan_transparent	None
mtu	1500
name	PROJECT_NET
port_security_enabled	True
project_id	9233b6b4f6a54386af63c0a7b8f043c2
provider:network_type	vlan
provider:physical_network	physnet1
provider:segmentation_id	43
qos_policy_id	None
revision_number	2
router:external	Internal
segments	None
shared	False
status	ACTIVE
subnets	
tags	
updated_at	2018-04-13T16:02:39Z

Notice how Neutron automatically determined the network type, physical network, and segmentation ID for the network. Because the `openstack network create` command was executed without specific provider attributes, Neutron relied on the configuration found in the plugin configuration file to determine what type of network to create.

The following configuration options in the ML2 configuration file were used to determine the network type, physical network, and segmentation ID:

```
tenant_network_types = vlan,vxlan
network_vlan_ranges = physnet1:40:43
```



Remember, in this configuration, Neutron will consume all available VLAN segmentation IDs as networks are created before moving on to VXLAN networks.

Using the `openstack subnet create` command, create a subnet with the following attributes:

- Name: PROJECT_SUBNET
- Network: 192.168.200.0
- Subnet Mask: 255.255.255.0
- Gateway: <auto>
- DHCP Range: <auto>
- DNS Nameserver: 8.8.8.8

The output should resemble the following:

```

root@controller01:~# openstack subnet create \
> --subnet-range 192.168.200.0/24 \
> --network PROJECT_NET \
> --dns-nameserver 8.8.8.8 \
> PROJECT_SUBNET

```

Field	Value
allocation_pools	192.168.200.2-192.168.200.254
cidr	192.168.200.0/24
created_at	2018-04-13T16:04:56Z
description	
dns_nameservers	8.8.8.8
enable_dhcp	True
gateway_ip	192.168.200.1
host_routes	
id	f29a2257-1283-4047-a835-b207480aa6f3
ip_version	4
ipv6_address_mode	None
ipv6_ra_mode	None
name	PROJECT_SUBNET
network_id	ce1fe7cd-7b01-4d58-b1c8-414886b7d8f2
project_id	9233b6b4f6a54386af63c0a7b8f043c2
revision_number	0
segment_id	None
service_types	
subnetpool_id	None
tags	
updated_at	2018-04-13T16:04:56Z
use_default_subnet_pool	None

Attaching the router to the internal network

Using the `openstack router add subnet` command, attach the `PROJECT_SUBNET` subnet to `MyLegacyRouter`:

```
# openstack router add subnet MyLegacyRouter PROJECT_SUBNET
```

No output is provided if the command is successful. Using the `openstack port list` command, determine the internal IP of the router:

```
root@controller01:~# openstack port list --router MyLegacyRouter
```

ID	Name	MAC Address	Fixed IP Addresses	Status
73e65609-f0ea-4f4b-9581-42fe4b7df8d8		fa:16:3e:8c:a0:3f	ip_address='10.30.0.106', subnet_id='0af5a767-fce0-4c4d-9df2-1aedbd259405'	ACTIVE
a9327c59-a654-479f-9a0f-b706e81cd1ba		fa:16:3e:86:4e:53	ip_address='192.168.200.1', subnet_id='f29a2257-1283-4047-a835-b207480aa6f3'	ACTIVE

In this example, the IP address assigned to the router's internal interface is `192.168.200.1`. When a port ID or IP address is not specified when attaching the router to a subnet, the IP address assigned to the router's internal interface defaults to the address set as the `gateway_ip` for the subnet.

Inside the router namespace, a new interface has been added with a prefix of `qr`. A `qr` interface is an internal interface of the router:

```
root@controller01:~# ip netns exec qrouter-9ef2eed-4a55-4f64-b8be-4b07a43ec373 ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
2: qg-73e65609-f0@if16: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:8c:a0:3f brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.30.0.106/24 brd 10.30.0.255 scope global qg-73e65609-f0
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe8c:a03f/64 scope link
        valid_lft forever preferred_lft forever
3: qr-a9327c59-a6@if22: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:86:4e:53 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.200.1/24 brd 192.168.200.255 scope global qr-a9327c59-a6
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe86:4e53/64 scope link
        valid_lft forever preferred_lft forever
```

When using the Open vSwitch interface driver, the interface is connected directly to the integration bridge. When using the Linux bridge interface driver, as in this example, every `qr` interface is one end of a veth pair whose other end is connected to a bridge on the host:

```
root@controller01:~# brctl show
bridge name      bridge id        STP enabled      interfaces
brq3f05db4d-c4    8000.000c291124cb  no               eth2.42
                  tapcfa2b6a1-d2

brq758070f9-ec    8000.000c291124cb  no               eth2.30
                  tap73e65609-f0

brqce1fe7cd-7b    8000.000c291124cb  no               eth2.43
                  tapddb596d-d6
                  tapa9327c59-a6
                  DHCP Server Port ID
                  Router Internal Port ID

brqd8428f56-ba    8000.000c291124cb  no               eth2.41
                  tap9d3d1ae5-8c

brqdb527151-20    8000.000c291124cb  no               eth2.40
                  tapd5a55dff-9e
```



For easy identification, the bridge name includes the first ten characters of the respective Neutron network ID. In addition, each end of the veth pair includes the first ten characters of the Neutron port ID associated with the interface.

The router namespace is able to communicate with other devices in the same subnet through the bridge. The `eth2.43` interface in the bridge tags traffic as VLAN 43 as it exits the bridge out to the parent interface `eth2`.

Creating instances

Create two instances with the following characteristics:

Name	Network	Image	Flavor	Host
MyInstance1	PROJECT_NET	cirros-0.4.0	tiny	compute01
MyInstance2	PROJECT_NET	cirros-0.4.0	tiny	compute02

If necessary, use the `openstack image list` command to determine the ID of the CirrOS image downloaded in *Chapter 2, Installing OpenStack*:

Use the following `openstack server create` commands to boot two instances across two compute nodes in the `PROJECT_NET` network:

```
openstack server create \
--flavor tiny \
--image cirros-0.4.0 \
--nic net-id=PROJECT_NET \
--availability-zone nova:compute01 \
MyInstance1
```

```
openstack server create \
--flavor tiny \
--image cirros-0.4.0 \
--nic net-id=PROJECT_NET \
--availability-zone nova:compute02 \
MyInstance2
```

The `openstack server list` command can be used to return a list of instances and their IP addresses:

```
root@controller01:~# openstack server list
```

ID	Name	Status	Networks	Image	Flavor
bbfc2f0c-7e9f-46ea-aa30-4b80b39dcde	MyInstance2	ACTIVE	PROJECT_NET=192.168.200.10	cirros-0.4.0	tiny
b35a1be3-d330-44ae-a027-6ca072e575be	MyInstance1	ACTIVE	PROJECT_NET=192.168.200.6	cirros-0.4.0	tiny

On `compute01`, a Linux bridge has been created that corresponds to the `PROJECT_NET` network. When connected to the bridge, we can find a `VLAN` interface and the tap interface that corresponds to `MyInstance1`:

```
root@compute01:~# brctl show
```

bridge name	bridge id	STP enabled	interfaces
brqc1fe7cd-7b	8000.000c29fa3f25	no	eth2.43 tap7ae20af9-e3



When the Linux bridge agent is used, bridges correspond to individual networks, and the names will have a prefix of `brq`.

On `compute02`, we can find the tap interface that corresponds to `MyInstance2`; it is connected to the Linux bridge dedicated to the respective port:

```
root@compute02:~# brctl show
bridge name      bridge id        STP enabled      interfaces
qbr7e18b92b-ed   8000.76bb40c6d802  no               qvb7e18b92b-ed
                                                           tap7e18b92b-ed
```



When the Open vSwitch agent is used along with the `iptables_hybrid` firewall driver, bridges correspond to individual ports, and the names will have a prefix of `qbr`. These bridges are only used to overcome `iptables` limitations with Open vSwitch virtual switches.

Verifying instance connectivity

When a network and subnet are created with DHCP enabled, a network namespace is created by a DHCP agent that serves as a DHCP server for the network. On the host running the Neutron DHCP agent service, the `ip netns list` command can be used to reveal the namespace. For easy identification, the name of a DHCP namespace corresponds to the ID of the network it is serving:

```
root@controller01:~# ip netns list
qdhcp-ce1fe7cd-7b01-4d58-b1c8-414886b7d8f2 (id: 4)
qrouter-9ef2eed-4a55-4f64-b8be-4b07a43ec373 (id: 3)
qdhcp-ec7e4e1e-cc54-4e1a-9e57-5b3e2f752c50 (id: 2)
qdhcp-7745a4a9-68a4-444e-a5ff-f9439e3aac6c (id: 1)
qdhcp-03327707-c369-4bd7-bd71-a42d9bcf49b8 (id: 0)
```

DHCP Namespace
for PROJECT_NET
network

Inside the namespace, an interface with a prefix of `ns` has been created and assigned an address from the allocation pool of the subnet:

```
root@controller01:~# ip netns exec qdhcp-ce1fe7cd-7b01-4d58-b1c8-414886b7d8f2 ip addr list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
2: ns-ddbe596d-d6@if19: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:0d:49:68 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 169.254.169.254/16 brd 169.254.255.255 scope global ns-ddbe596d-d6
        valid_lft forever preferred_lft forever
    inet 192.168.200.2/24 brd 192.168.200.255 scope global ns-ddbe596d-d6
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe0d:4968/64 scope link
        valid_lft forever preferred_lft forever
```

When a DHCP agent is configured to use the Open vSwitch interface driver, the `ns` interface inside the namespace is connected directly to the integration bridge. When a DHCP agent is configured to use the Linux bridge interface driver, as in this example, the `ns` interface is one end of a veth pair whose other end is connected to a bridge on the host. The namespace is able to communicate with other devices in the same subnet through the respective bridge and VLAN.

As the instances come online, they send a DHCP request that is served by the `dnsmasq` process in the DHCP namespace. A populated ARP table within the namespace confirms instances are functioning in the VLAN at Layer 2:

```
root@controller01:~# ip netns exec qdhcp-celfe7cd-7b01-4d58-b1c8-414886b7d8f2 ip neigh show
192.168.200.6 dev ns-ddbe596d-d6 lladdr fa:16:3e:b5:60:b5 REACHABLE ← MyInstance1
192.168.200.10 dev ns-ddbe596d-d6 lladdr fa:16:3e:ef:7f:1b REACHABLE ← MyInstance2
```



The L2 population driver is used to pre-populate ARP and forwarding tables to reduce overhead on overlay networks and may not provide an accurate picture of connectivity in those networks. If an entry is in a PERMANENT state, it has been statically programmed and may not reflect actual reachability.

Before you can connect to the instances, security group rules must be updated to allow ICMP and SSH. [Chapter 8, Managing Security Groups](#), focuses on the implementation and administration of security group rules in more detail. To test connectivity, add ICMP and SSH access to security group applied to the instances. Use the following command to determine the security group for these particular instances:

```
# openstack port list --server MyInstance1 --long -c 'Security Groups'
# openstack port list --server MyInstance2 --long -c 'Security Groups'
```

The output may resemble the following:

```
root@controller01:~# openstack port list --server MyInstance1 --long -c 'Security Groups'
+-----+
| Security Groups |
+-----+
| fae5da02-5d96-43df-aebf-bf0210d353bc |
+-----+

root@controller01:~# openstack port list --server MyInstance2 --long -c 'Security Groups'
+-----+
| Security Groups |
+-----+
| fae5da02-5d96-43df-aebf-bf0210d353bc |
+-----+
```


Use the `openstack security group rule create` command to create rules within the respective security group that allow inbound ICMP and SSH:

```

root@controller01:~# openstack security group rule create \
> --protocol icmp \
> fae5da02-5d96-43df-aebf-bf0210d353bc

```

Field	Value
created_at	2018-04-16T16:50:17Z
description	
direction	ingress
ether_type	IPv4
id	ff8cf830-1596-4c12-8f15-3d74c35e82da
name	None
port_range_max	None
port_range_min	None
project_id	9233b6b4f6a54386af63c0a7b8f043c2
protocol	icmp
remote_group_id	None
remote_ip_prefix	0.0.0.0/0
revision_number	0
security_group_id	fae5da02-5d96-43df-aebf-bf0210d353bc
updated_at	2018-04-16T16:50:17Z

```

root@controller01:~# openstack security group rule create \
> --protocol tcp \
> --dst-port 22 \
> fae5da02-5d96-43df-aebf-bf0210d353bc

```

Field	Value
created_at	2018-04-16T16:50:55Z
description	
direction	ingress
ether_type	IPv4
id	1224a3f0-7af5-4e63-942e-36ec25a19e4e
name	None
port_range_max	22
port_range_min	22
project_id	9233b6b4f6a54386af63c0a7b8f043c2
protocol	tcp
remote_group_id	None
remote_ip_prefix	0.0.0.0/0
revision_number	0
security_group_id	fae5da02-5d96-43df-aebf-bf0210d353bc
updated_at	2018-04-16T16:50:55Z

Using the SSH command, connect to the instance `MyInstance1` from either the router or DHCP namespace. The CirrOS image has a built-in user named `cirros` with a password of `gocubsgo`:

```
root@controller01:~# ip netns exec qrouter-9ef2eed-4a55-4f64-b8be-4b07a43ec373 ssh cirros@192.168.200.6
The authenticity of host '192.168.200.6 (192.168.200.6)' can't be established.
ECDSA key fingerprint is SHA256:S7S5nf1XRi5ZErVMqBRHlf8LrzAeyEaL9MAhKVHWEWY.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.200.6' (ECDSA) to the list of known hosts.
cirros@192.168.200.6's password:
$ hostname
myinstance1
$ ip route
default via 192.168.200.1 dev eth0
169.254.169.254 via 192.168.200.1 dev eth0
192.168.200.0/24 dev eth0 src 192.168.200.6
$ exit
Connection to 192.168.200.6 closed.
```

Observe the routing table inside the instance, like so:

```
$ ip r
default via 192.168.200.1 dev eth0
169.254.169.254 via 192.168.200.1 dev eth0
192.168.200.0/24 dev eth0 src 192.168.200.6
```

The default gateway of `192.168.200.1` is the Neutron router that we created earlier in this chapter. Pinging an external resource from an instance should prove successful, provided external connectivity from the Neutron router exists:

```
root@controller01:~# ip netns exec qrouter-9ef2eed-4a55-4f64-b8be-4b07a43ec373 ssh cirros@192.168.200.6
cirros@192.168.200.6's password:
$ ping 8.8.8.8 -c5
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=55 time=30.296 ms
64 bytes from 8.8.8.8: seq=1 ttl=55 time=29.988 ms
64 bytes from 8.8.8.8: seq=2 ttl=55 time=29.158 ms
64 bytes from 8.8.8.8: seq=3 ttl=55 time=28.948 ms
64 bytes from 8.8.8.8: seq=4 ttl=55 time=28.524 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 28.524/29.382/30.296 ms
```

Observing default NAT behavior

The default behavior of the Neutron router is to source NAT traffic from instances that lack floating IPs when traffic egresses the external, or gateway, interface of the router.

Performing a packet capture on the `eth2.43` interface of the `controller` node that corresponds with the `PROJECT_NET` network, we can observe ICMP traffic from the instances sourcing from their real or fixed addresses as the traffic heads towards the router. The reply also references the same fixed IP address:

```
root@controller01:~# tcpdump -i eth2.43 -n icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth2.43, link-type EN10MB (Ethernet), capture size 262144 bytes
17:01:54.349267 IP 192.168.200.6 > 8.8.8.8: ICMP echo request, id 34049, seq 0, length 64
17:01:54.377051 IP 8.8.8.8 > 192.168.200.6: ICMP echo reply, id 34049, seq 0, length 64
```

From the `eth2.30` interface on the `controller` node that corresponds to the `GATEWAY_NET` network, we can observe ICMP traffic from the instances after it has traversed the router sourcing as the router's external address, `10.30.0.106`:

```
root@controller01:~# tcpdump -i eth2.30 -n icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth2.30, link-type EN10MB (Ethernet), capture size 262144 bytes
17:00:47.436765 IP 10.30.0.106 > 8.8.8.8: ICMP echo request, id 33793, seq 0, length 64
17:00:47.464786 IP 8.8.8.8 > 10.30.0.106: ICMP echo reply, id 33793, seq 0, length 64
```

A look at the iptables chains within the router namespace reveals the NAT rules responsible for this behavior:


```

root@controller01:~# ip netns exec qrouter-9ef2eed-4a55-4f64-b8be-4b07a43ec373 iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target prot opt source destination
neutron-l3-agent-PREROUTING all -- anywhere anywhere

Chain INPUT (policy ACCEPT)
target prot opt source destination

Chain OUTPUT (policy ACCEPT)
target prot opt source destination
neutron-l3-agent-OUTPUT all -- anywhere anywhere

Chain POSTROUTING (policy ACCEPT)
target prot opt source destination
neutron-l3-agent-POSTROUTING all -- anywhere anywhere
neutron-postrouting-bottom all -- anywhere anywhere

Chain neutron-l3-agent-OUTPUT (1 references)
target prot opt source destination

Chain neutron-l3-agent-POSTROUTING (1 references)
target prot opt source destination
ACCEPT all -- anywhere anywhere ! ctstate DNAT

Chain neutron-l3-agent-PREROUTING (1 references)
target prot opt source destination
REDIRECT tcp -- anywhere 169.254.169.254 tcp dpt:http redir ports 9697

Chain neutron-l3-agent-float-snat (1 references)
target prot opt source destination

Chain neutron-l3-agent-snat (1 references)
target prot opt source destination
neutron-l3-agent-float-snat all -- anywhere anywhere
SNAT all -- anywhere anywhere to:10.30.0.106
SNAT all -- anywhere anywhere mark match ! 0x2/0xffff ctstate DNAT to:10.30.0.106

Chain neutron-postrouting-bottom (1 references)
target prot opt source destination
neutron-l3-agent-snat all -- anywhere anywhere /* Perform source NAT on outgoing traffic. */

```

In this configuration, instances can communicate with outside resources through the router as long as the instances initiate the connection. Outside resources cannot initiate connections directly to instances via their fixed IP address.

Assigning floating IPs

To initiate connections to instances behind Neutron routers from outside networks, you must configure a floating IP address and associate it with the instance. In OpenStack, a floating IP is associated with a Neutron port that corresponds to an interface of the instance accepting connections.

Using the `openstack port list` command, determine the port ID of each instance recently booted. The command allows results to be filtered by device or instance ID, as shown in the following screenshot:

```
root@controller01:~# openstack port list --server MyInstance1
```

ID	Name	MAC Address	Fixed IP Addresses	Status
7ae20af9-e3d3-41ce-b74a-57cfaf41fe59		fa:16:3e:b5:60:b5	ip_address='192.168.200.6', subnet_id='f29a2257-1283-4047-a835-b207480aa6f3'	ACTIVE

```
root@controller01:~# openstack port list --server MyInstance2
```

ID	Name	MAC Address	Fixed IP Addresses	Status
7e18b92b-ed27-42d5-ad05-f58e2b0f5705		fa:16:3e:ef:7f:1b	ip_address='192.168.200.10', subnet_id='f29a2257-1283-4047-a835-b207480aa6f3'	ACTIVE

Using the `openstack floating ip create` command, create a single floating IP address and associate it with the port of the instance known as `MyInstance1`:

```
root@controller01:~# openstack floating ip create --port 7ae20af9-e3d3-41ce-b74a-57cfaf41fe59 GATEWAY_NET
```

Field	Value
created_at	2018-04-18T13:15:35Z
description	
fixed_ip_address	192.168.200.6
floating_ip_address	10.30.0.101
floating_network_id	758070f9-ecaf-4d2a-aa49-e119ce7943f6
id	9e08a332-5128-4e3c-a201-ee3e6d8d1b75
name	10.30.0.101
port_id	7ae20af9-e3d3-41ce-b74a-57cfaf41fe59
project_id	9233b6b4f6a54386af63c0a7b8f043c2
revision_number	0
router_id	9ef2eed-4a55-4f64-b8be-4b07a43ec373
status	DOWN
updated_at	2018-04-18T13:15:35Z



Upon creation, the floating IP may appear to be in a DOWN state. Once the changes have been applied to the network, the status should reflect an ACTIVE state.

From within the guest OS, verify that the instance can still communicate with outside resources:

```
$ ping 8.8.8.8 -c 2
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=55 time=28.491 ms
64 bytes from 8.8.8.8: seq=1 ttl=55 time=28.590 ms

--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 28.491/28.540/28.590 ms
```

Performing a packet capture on the `eth2.30` interface on the `controller01` node, we can observe ICMP traffic from the instance through the router having a source IP that corresponds to the floating IP address `10.30.0.101`:

```
root@controller01:~# tcpdump -i eth2.30 -n icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens224.30, link-type EN10MB (Ethernet), capture size 262144 bytes
13:19:00.598980 IP 10.30.0.101 > 8.8.8.8: ICMP echo request, id 40705, seq 0, length 64
13:19:00.626593 IP 8.8.8.8 > 10.30.0.101: ICMP echo reply, id 40705, seq 0, length 64
13:19:01.599335 IP 10.30.0.101 > 8.8.8.8: ICMP echo request, id 40705, seq 1, length 64
13:19:01.627029 IP 8.8.8.8 > 10.30.0.101: ICMP echo reply, id 40705, seq 1, length 64
^C
4 packets captured
4 packets received by filter
0 packets dropped by kernel
```

Within the router namespace, the floating IP has been configured as a secondary address on the `qg` interface:

```
root@controller01:~# ip netns exec qrouter-9ef2eed-4a55-4f64-b8be-4b07a43ec373 ip addr list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
2: qg-73e65609-f0@if16: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:8c:a0:3f brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.30.0.106/24 brd 10.30.0.255 scope global qg-73e65609-f0
        valid_lft forever preferred_lft forever
    inet 10.30.0.101/32 brd 10.30.0.101 scope global qg-73e65609-f0
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe8c:a03f/64 scope link
        valid_lft forever preferred_lft forever
3: qr-a9327c59-a6@if22: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:86:4e:53 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.200.1/24 brd 192.168.200.255 scope global qr-a9327c59-a6
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe86:4e53/64 scope link
        valid_lft forever preferred_lft forever
```

Floating IPs are configured with a /32 netmask

When the floating IP is configured as a secondary network address on the `qg` interface, the router is able to respond to ARP requests to the floating IP from the upstream gateway device and other Neutron routers or devices in the same external network. This allows inbound connectivity to the instance via the floating IP.

A look at the `iptables` chains within the router namespace show rules have been added to perform the 1:1 NAT translation from the floating IP to the fixed IP of `MyInstance1`, and vice versa:

```
root@controller01:~# ip netns exec qrouter-9ef2eed-4a55-4f64-b8be-4b07a43ec373 iptables -t nat -L
...
Chain neutron-l3-agent-OUTPUT (1 references)
target    prot opt source                destination
DNAT      all  --  anywhere              10.30.0.101           to:192.168.200.6

Chain neutron-l3-agent-POSTROUTING (1 references)
target    prot opt source                destination
ACCEPT    all  --  anywhere              anywhere              ! ctstate DNAT

Chain neutron-l3-agent-PREROUTING (1 references)
target    prot opt source                destination
REDIRECT  tcp  --  anywhere              169.254.169.254       tcp dpt:http redir ports 9697
DNAT      all  --  anywhere              10.30.0.101           to:192.168.200.6

Chain neutron-l3-agent-float-snat (1 references)
target    prot opt source                destination
SNAT      all  --  192.168.200.6         anywhere              to:10.30.0.101

Chain neutron-l3-agent-snat (1 references)
target    prot opt source                destination
neutron-l3-agent-float-snat all  --  anywhere              anywhere              anywhere
SNAT      all  --  anywhere              anywhere              to:10.30.0.106
SNAT      all  --  anywhere              anywhere              mark match ! 0x2/0xffff ctstate DNAT to:10.30.0.106

Chain neutron-postrouting-bottom (1 references)
target    prot opt source                destination
neutron-l3-agent-snat all  --  anywhere              anywhere              /* Perform source NAT on outgoing traffic. */
```

Provided our client workstation can route to the external provider network, traffic can be initiated directly to the instance via the floating IP:

```
workstation:~ jdenton$ ssh cirros@10.30.0.101
The authenticity of host '10.30.0.101 (10.30.0.101)' can't be established.
ECDSA key fingerprint is SHA256:S7S5nf1XRi5ZErVMqBRHlf8LrzAeyEaL9MAhKVHWEWY.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.30.0.101' (ECDSA) to the list of known hosts.
cirros@10.30.0.101's password:
$ who
cirros      pts/0      00:06  Apr 18 14:16:52  192.168.200.1
cirros      pts/1      00:00  Apr 18 14:25:45  192.168.1.75
$ ip route show
default via 192.168.200.1 dev eth0
169.254.169.254 via 192.168.200.1 dev eth0
192.168.200.0/24 dev eth0  src 192.168.200.6
$ exit
Connection to 10.30.0.101 closed.
```

Reassigning floating IPs

Neutron provides the ability to quickly disassociate a floating IP from an instance or other network resource and associate it with another. A list of floating IPs shows the current association:

```
root@controller01:~# openstack floating ip list
```

ID	Floating IP Address	Fixed IP Address	Port	Floating Network
9e08a332-5128-4e3c-a201-ee3e6d8d1b75	10.30.0.101	192.168.200.6	7ae20af9-e3d3-41ce-b74a-57cfaf41fe59	758070f9-ecaf-4d2a-aa49-e119ce7943f6

Using the `openstack floating ip unset` and `openstack floating ip set` commands, disassociate the floating IP from `MyInstance1` and associate it with `MyInstance2`. The disassociation can be seen here:

```
root@controller01:~# openstack floating ip unset --port 9e08a332-5128-4e3c-a201-ee3e6d8d1b75
root@controller01:~#
```

No output will be returned if the command is successful. The `openstack floating ip list` command shows that the floating IP is no longer associated with a port:

```
root@controller01:~# openstack floating ip list
```

ID	Floating IP Address	Fixed IP Address	Port	Floating Network
9e08a332-5128-4e3c-a201-ee3e6d8d1b75	10.30.0.101	None	None	758070f9-ecaf-4d2a-aa49-e119ce7943f6



The floating IP is still owned by the project that created it and cannot be assigned to another project without first being deleted.

Using the `openstack floating ip set` command, associate the floating IP with the port of `MyInstance2`, as shown in the following screenshot:

```
root@controller01:~# openstack port list --server MyInstance2
```

ID	Name	MAC Address	Fixed IP Addresses	Status
7e18b92b-ed27-42d5-ad05-f58e2b0f5705		fa:16:3e:ef:7f:1b	ip_address='192.168.200.10', subnet_id='f29a2257-1283-4047-a835-b207480aa6f3'	ACTIVE

```
root@controller01:~# openstack floating ip set --port 7e18b92b-ed27-42d5-ad05-f58e2b0f5705 9e08a332-5128-4e3c-a201-ee3e6d8d1b75
root@controller01:~#
```

Port of MyInstance2 Floating IP ID

No output will be returned if the command is successful. Observe the iptables rules within the router namespace. The NAT relationship has been modified, and traffic from MyInstance2 will now appear as the floating IP:

```
root@controller01:~# ip netns exec qrouter-9ef2eed-4a55-4f64-b8be-4b07a43ec373 iptables -t nat -L
...
Chain neutron-l3-agent-OUTPUT (1 references)
target    prot opt source                destination
DNAT      all  --  anywhere               10.30.0.101          to:192.168.200.10

Chain neutron-l3-agent-POSTROUTING (1 references)
target    prot opt source                destination
ACCEPT    all  --  anywhere               anywhere             ! ctstate DNAT

Chain neutron-l3-agent-PREROUTING (1 references)
target    prot opt source                destination
REDIRECT  tcp  --  anywhere               169.254.169.254      tcp dpt:http redir ports 9697
DNAT      all  --  anywhere               10.30.0.101          to:192.168.200.10

Chain neutron-l3-agent-float-snat (1 references)
target    prot opt source                destination
SNAT      all  --  192.168.200.10        anywhere             to:10.30.0.101
The source IP has changed from .6 to .10

Chain neutron-l3-agent-snat (1 references)
target    prot opt source                destination
neutron-l3-agent-float-snat all -- anywhere             anywhere
SNAT      all  --  anywhere             anywhere             to:10.30.0.106
SNAT      all  --  anywhere             anywhere             mark match ! 0x2/0xffff ctstate DNAT to:10.30.0.106

Chain neutron-postrouting-bottom (1 references)
target    prot opt source                destination
neutron-l3-agent-snat all -- anywhere             anywhere             /* Perform source NAT on outgoing traffic. */
```

As a result of the new association, attempting an SSH connection to the floating IP may result in the following message on the client machine:

```
workstation:~ jdenton$ ssh cirros@10.30.0.101
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!    @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
SHA256:DHBSR4jrZAu3IBHoLBF+gy7czdq3gHzBdystllLfslg.
Please contact your system administrator.
Add correct host key in /Users/jdenton/.ssh/known_hosts to get rid of this message.
Offending ECDSA key in /Users/jdenton/.ssh/known_hosts:9
ECDSA host key for 10.30.0.101 has changed and you have requested strict checking.
Host key verification failed.
```

The warning message is a good indicator that traffic is being sent to a different host. Clearing the offending key and logging into the instance reveals it to be `MyInstance2`:

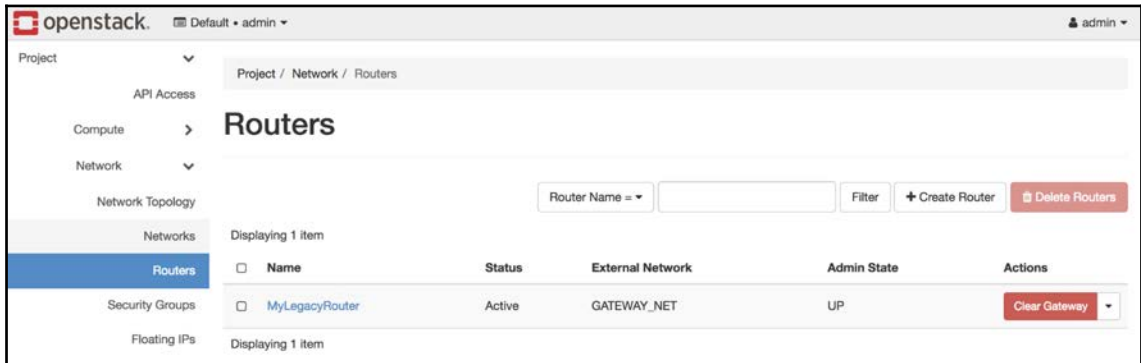
```
workstation:~ jdenton$ ssh-keygen -R 10.30.0.101
# Host 10.30.0.101 found: line 9
/Users/jdenton/.ssh/known_hosts updated.
Original contents retained as /Users/jdenton/.ssh/known_hosts.old

workstation:~ jdenton$ ssh cirros@10.30.0.101
The authenticity of host '10.30.0.101 (10.30.0.101)' can't be established.
ECDSA key fingerprint is SHA256:DHBSR4jrZAu3IBHoLBF+gy7czdqV3gHzBdystLLfslg.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.30.0.101' (ECDSA) to the list of known hosts.
cirros@10.30.0.101's password:
$ hostname
myinstance2
$ ip addr list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue 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
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether fa:16:3e:ef:7f:1b brd ff:ff:ff:ff:ff:ff
    inet 192.168.200.10/24 brd 192.168.200.255 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:feef:7f1b/64 scope link
        valid_lft forever preferred_lft forever
$ exit
Connection to 10.30.0.101 closed.
```

At this point, we have successfully deployed two instances behind a single virtual router and have verified connectivity to and from the instances using floating IPs. In the next section, we will explore how those same tasks can be accomplished within the Horizon dashboard.

Router management in the dashboard

From the Horizon dashboard, routers can be created and managed within the **Project | Network | Routers** pane:



Routers available to the logged-in user can be seen on this page.

Creating a router in the dashboard

From the **Routers** page, click **Create Router** in the upper right-hand corner to create a router. A wizard will open that resembles the following:

The screenshot shows the 'Create Router' wizard. It has a title bar with a close button. The form contains several fields: 'Router Name' with the value 'MyLegacyDashboardRouter', a checked checkbox for 'Enable Admin State', and an 'External Network' dropdown menu. The dropdown menu is open, showing a list of networks: 'Select network', 'GATEWAY_NET' (which is highlighted with a red rectangle), and 'MySemiSharedExternalNetwork'. To the right of the form, there is a 'Description:' section with the text 'Creates a router with specified parameters.' At the bottom right, there are two buttons: 'Cancel' and 'Create Router'.

Enter the name of the router, select its admin state, and choose the appropriate external network. Click the blue **Create Router** button to complete the operation.

Attaching internal interfaces in the dashboard

To attach internal interfaces to routers in the dashboard, click the router to reveal the **Router Details** page shown here:

Project / Network / Routers / MyLegacyDashboardRouter

MyLegacyDashboardRouter

[Clear Gateway](#)

Overview **Interfaces** Static Routes

Name	MyLegacyDashboardRouter
ID	acda1cab-2e4c-48e7-83db-13f516d499fd
Description	
Project ID	9233b6b4f6a54386af63c0a7b8f043c2
Status	Active
Admin State	UP
Availability Zones	<ul style="list-style-type: none"> nova

External Gateway

Network Name	GATEWAY_NET
Network ID	758070f9-ecaf-4d2a-aa49-e119ce7943f6
External Fixed IPs	<ul style="list-style-type: none"> Subnet ID 0af5a767-fce0-4c4d-9df2-1a6dbd259405 IP Address 10.30.0.100
SNAT	Enabled

Click the **Interfaces** tab to reveal details of the router's interfaces:

Project / Network / Routers / MyLegacyDashboardRouter

MyLegacyDashboardRouter

[Clear Gateway](#)

Overview **Interfaces** Static Routes

[+ Add Interface](#) [Delete Interfaces](#)

Displaying 1 item

☐	Name	Fixed IPs	Status	Type	Admin State	Actions
☐	(9296f277-b2b3)	<ul style="list-style-type: none"> 10.30.0.100 	Down	External Gateway	UP	Delete Interface

Displaying 1 item

Clicking the **Add Interface** button reveals a wizard that allows you to select details of the interface to be added:

Add Interface ✕

Subnet *

Select Subnet ▾

IP Address (optional) ?

Description:

You can connect a specified subnet to the router.

If you don't specify an IP address here, the gateway's IP address of the selected subnet will be used as the IP address of the newly created interface of the router. If the gateway's IP address is in use, you must use a different address which belongs to the selected subnet.

Cancel

Submit

Select a subnet you wish to attach to the router from the **Subnet** menu and click the blue **Submit** button to attach the interface. The newly attached interface will be revealed on the **Interface** pane shown here:

Project / Network / Routers / MyLegacyDashboardRouter

MyLegacyDashboardRouter Clear Gateway ▾

Overview

Interfaces

Static Routes

+ Add Interface

Delete Interfaces

Displaying 2 items

☐	Name	Fixed IPs	Status	Type	Admin State	Actions
☐	(9296f277-b2b3)	• 10.30.0.100	Active	External Gateway	UP	Delete Interface
☐	(cecf02e-5ea6)	• 192.168.1.1	Down	Internal Interface	UP	Delete Interface

Displaying 2 items

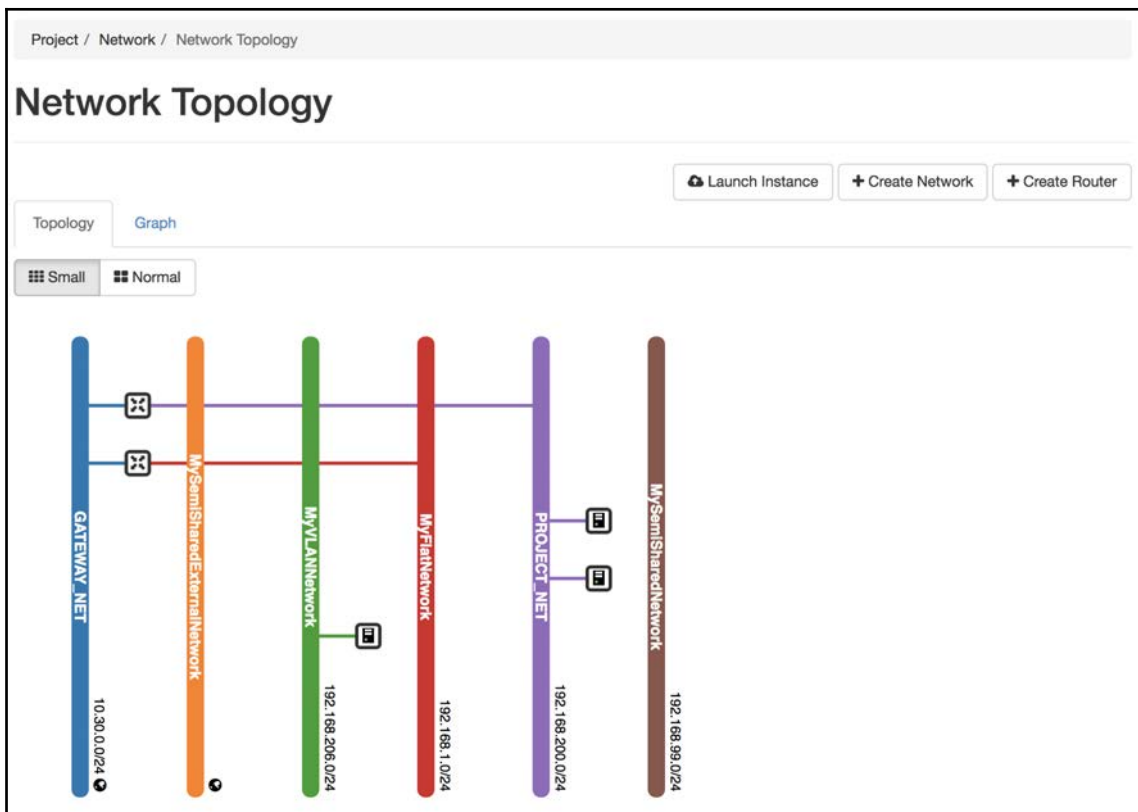


It is normal for an interface's status to be Down immediately after adding the interface to the router. Neutron will not mark the interface as Active until the agents have completed their tasks. Refreshing the dashboard will update the status accordingly.

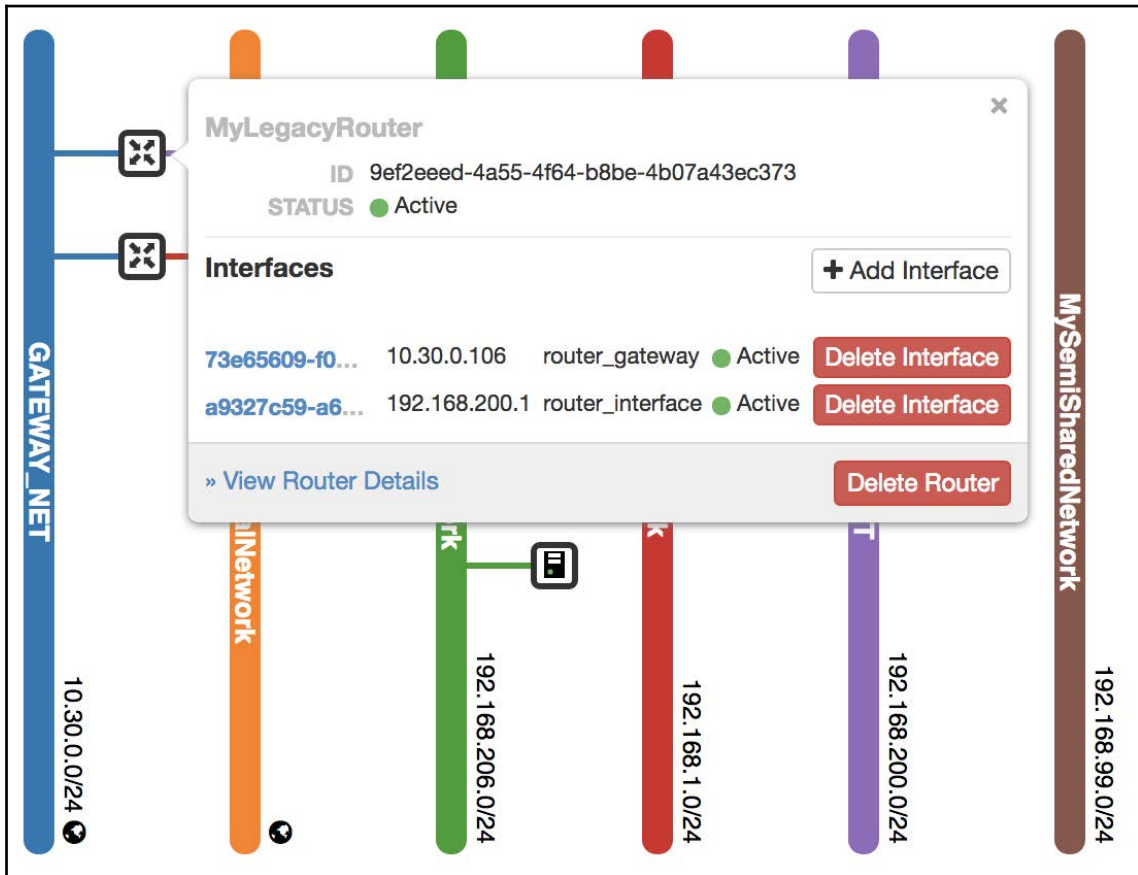
Viewing the network topology in the dashboard

From within the dashboard, users can view a logical topology of the network based on the network configuration managed by Neutron.

Click on **Network Topology** under the **Project | Network** pane to find a logical diagram based on the networks, router, and instances that we created earlier:



Hovering over the router icon reveals a popup displaying details about the router such as connected ports, IPs, and port status:



Associating floating IPs to instances in the dashboard

Floating IPs in the dashboard are managed on the **Instances** page found within the **Project | Compute** pane. Click the menu under the **Actions** column next to the instance you wish to assign a floating IP to:

☐	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Time since created	Actions
☐	MyInstance2	cirros-0.4.0	192.168.200.10 Floating IPs: 10.30.0.101	tiny	-	Active	nova	None	Running	1 day, 21 hours	Create Snapshot
☐	MyInstance1	cirros-0.4.0	192.168.200.6	tiny	-	Active	nova	None	Running	1 day, 21 hours	Create Snapshot
☐	TestInstance1	cirros-0.4.0	192.168.206.11	tiny	-	Active	nova	None	Running	1 week	Associate Floating IP Attach Interface Detach Interface Edit Instance Update Metadata Edit Security Groups

Displaying 3 items

To assign a floating IP, click **Associate Floating IP**. A wizard will be revealed that allows you to manage floating IP allocations:

Manage Floating IP Associations

IP Address

No floating IP addresses allocated

+

Select the IP address you wish to associate with the selected instance or port.

Port to be associated

MyInstance1: 192.168.200.6

Cancel

Associate

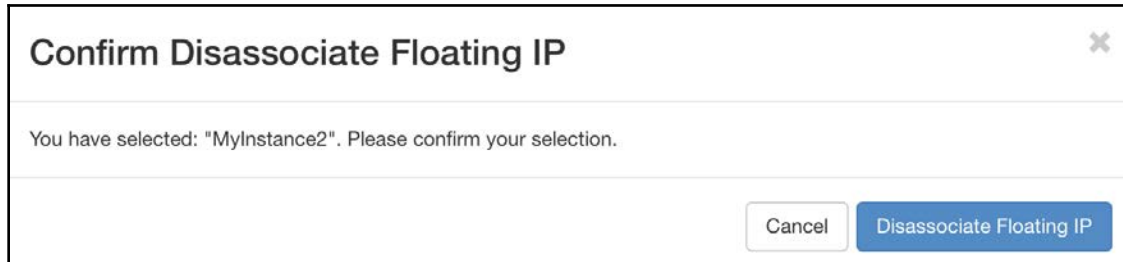
If there are no floating IP addresses available for allocation, click the plus (+) to create one. Click the **Associate** button to associate the floating IP with the port.

Disassociating floating IPs in the dashboard

To disassociate a floating IP from an instance in the dashboard, click the menu under the **Actions** column that corresponds to the instance and select **Disassociate Floating IP**:

☐	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Time since created	Actions
☐	MyInstance2	cirros-0.4.0	192.168.200.10 Floating IPs: 10.30.0.101	tiny	-	Active	nova	None	Running	3 days, 1 hour	Create Snapshot
☐	MyInstance1	cirros-0.4.0	192.168.200.6	tiny	-	Active	nova	None	Running	3 days, 1 hour	Disassociate Floating IP Attach Interface Detach Interface

A message will appear that warns you of the pending action:



Click the blue **Disassociate Floating IP** button to proceed with the action.



While the floating IP has been disassociated with the instance, it remains under the ownership of the project and is not returned to the allocation pool until it is deleted.

Summary

Neutron routers are a core component of networking in OpenStack and provide users the flexibility to design the network to best suit their application. The use of floating IPs allows users to quickly and programmatically provide direct connectivity to applications while preserving limited IPv4 address space through the use of network address translation.

Standalone routers are easy to implement but are a single point of failure in any network design. In the event of an L3 agent failure, all routers scheduled to the agent may become unavailable or unreliable. In the next chapter, we will discuss how Neutron implements highly available routers using the Virtual Router Redundancy Protocol, or VRRP, to solve many of the shortcomings of legacy standalone routers.

11

Router Redundancy Using VRRP

In the Juno release of OpenStack, the Neutron community introduced two methods of attaining high availability in routing. This chapter focuses on a method that uses the Virtual Routing Redundancy Protocol, also known as VRRP, to implement redundancy between two or more Neutron routers. High availability using distributed virtual routers, otherwise known as DVR, will be discussed in *Chapter 12, Distributed Virtual Routers*.

In the previous chapter, we explored the concept of standalone routers and how they allow users to route traffic between project networks and external networks as well as provide network address translation for instances managed by the user. In this chapter, we will cover the following topics:

- High availability of routing using keepalived and VRRP
- Installing and configuring additional L3 agents
- Demonstrating the creation and management of a highly-available router

Using keepalived and VRRP to provide redundancy

Keepalived is a software package for Linux that provides load balancing and high availability to Linux-based software and infrastructures.

The Virtual Router Redundancy Protocol, or VRRP, is a first hop redundancy protocol that aims to provide high availability of a network's default gateway by allowing two or more routers to provide backup for that address. If the active router fails, a backup router will take over the address within a brief period of time. VRRP is an open standard and is based on the proprietary Hot Standby Router Protocol, or HSRP, developed by Cisco.

Neutron uses keepalived, which utilizes VRRP, to provide failover between multiple sets of router namespaces.

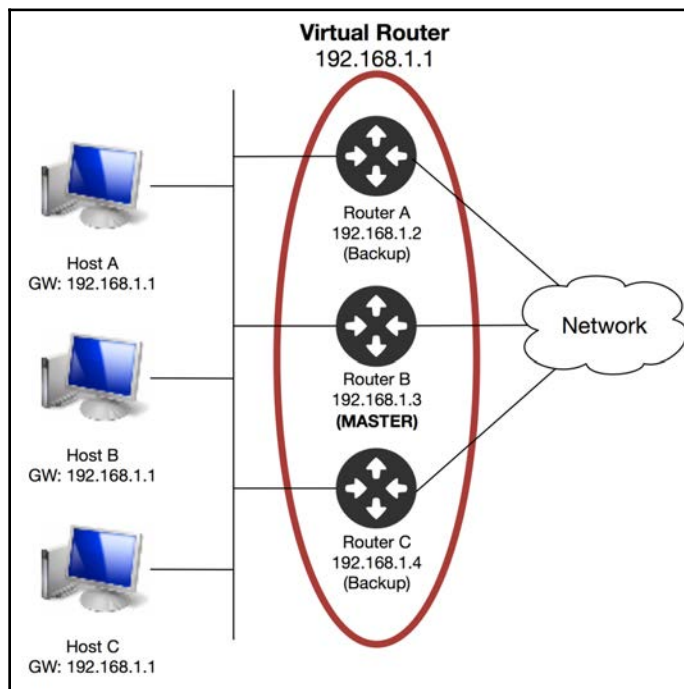
VRRP groups

With VRRP, a group of routers can be configured to act as a single virtual router. Routers in the VRRP group elect a master to act as the active gateway device, and hosts in the network only need to configure the virtual router address as their default network gateway or next hop address. When a failover occurs, another router in the group will take over routing duties while the configuration of hosts in the network never changes.



A VRRP virtual router should not be confused with a Neutron virtual router. The former represents a logical entity, while the latter actually exists as a virtualized routing device, typically implemented as a network namespace.

In the following diagram, **Router A**, **Router B**, and **Router C** form a single virtual router. In this configuration, each router has its own IP address and the virtual router has its own IP address. Hosts in the network use the virtual router address as their default gateway:



As the master router, **Router B** in the preceding diagram is responsible for the virtual address 192.168.1.1, and routes traffic for hosts using that address as their gateway. The master router sends VRRP advertisements to the group, which include the priority and state of the master router using the multicast address 224.0.0.18. The backup routers use a variety of timers and configuration options to determine when a master router has failed and change their state accordingly.



Routers that make up the virtual router communicate between themselves using the multicast IP address 224.0.0.18 and IP protocol number 112. The network setup for this communication is automatically configured and is not impacted by security group rules or manageable by users or operators.

VRRP priority

Routers in the VRRP group elect a master router according to their priorities. The router with the highest priority is elected master, while the other routers in the group are relegated to backup duties. When a master router fails to send its VRRP advertisements to the group, the backup routers in the VRRP group elect a new master to replace the failed master.

VRRP priorities range from 0 to 255, with 255 being the highest priority. Neutron configures each router in a group with the same priority of 50. Because the priority is the same between routers, in the event of a failover, the election process falls back to the highest IP address. That is to say that a backup router with IP address 192.168.1.200 on the HA interface will be elected master over a router with IP address 192.168.1.100.

VRRP working mode

A router in a VRRP group works in one of two modes: preemptive and non-preemptive.

Preemptive

In preemptive mode, when a master router fails, it becomes the master router again when it returns to a group and has a higher priority than the newly elected master.

Non-preemptive

In non-preemptive mode, when a router in a VRRP group becomes the master, it continues to operate as the master under normal working conditions. If a backup router is assigned a higher priority later, the active master router will continue to operate as the master until it fails.

As of the Kilo release of OpenStack, Neutron configures each router to act in non-preemptive mode, although this may change in the future. In the event of a failure of the HA network, which is used for communication between the routers, a failed master router may not detect that it has failed. It can continue to operate as a master router, even though another router has been elected master. The lack of connectivity between the routers means that all routers may not receive VRRP advertisements. When connectivity is re-established, the routers may engage in an election to determine a single master router. This failure scenario is not common, but one to keep in mind when troubleshooting issues with highly-available routers.

VRRP timers

Timers that are used within VRRP include an advertisement interval timer and a preemption delay timer.

Advertisement interval timer

The master router in a VRRP group periodically sends advertisements on an interval established by the advertisement interval timer to inform other routers in the group that it is operating properly. If a backup router does not receive advertisements in a period of three times the interval, the backup regards itself as the master and sends VRRP advertisements to start a new master election process. Neutron routers in a master state are configured to send advertisements every two seconds.

Preemption delay timer

After a backup router receives an advertisement with a priority lower than itself, it waits for a period of time established by the preemption delay timer before sending out VRRP advertisements to start a new master election. This delay helps the routers avoid frequent state changes among members of the VRRP group in cases of network flapping. Because preemption is not enabled within Neutron routers, this timer is not configured.

Networking of highly available routers

When a highly-available router is created, Neutron creates a VRRP group composed of at least two router namespaces by default. The namespaces are spread across multiple hosts running the Neutron L3 agent, and each runs the keepalived service with an automatically generated configuration. Traffic between the routers uses a dedicated network interface, which is discussed in the following section.

Dedicated HA network

Routers in a VRRP group communicate among one another over a dedicated HA network. An HA router is automatically configured with an interface prefixed with that is only used for this communication.

The first time an HA router is created in a project, Neutron configures a network and subnet using the CIDR `169.254.192.0/18`. The network type used is based on the default project network type. Only one HA network is created per project, and it is used by all HA routers that are created by that project. If all HA routers in a project are deleted, the HA network will remain and will be reused for all other HA routers that get created by the project in the future.



Networks created by Neutron for VRRP communication between routers are not actually assigned to projects. As a result, these networks are hidden from regular users in the CLI and GUI. The name of the network reflects the associated project, however, and is used by the L3 agent for identification purposes. Under normal conditions, an HA network should remain hidden and not be modified by users or administrators.

Limitations

VRRP utilizes a virtual router identifier, or VRID, to exchange VRRP protocol messages over multicast with other routers using the same VRID to determine which is the master router. The VRID is 8 bits in length and the valid range is 1-255. Because each project uses a single administrative network for VRRP communication between routers, individual projects are limited to 255 highly-available routers.

Virtual IP

A VRRP virtual router has a virtual IP address that can serve as the default gateway for hosts in the network. The master router owns the IP address until a failover event occurs, at which time a backup router becomes the new master and takes over the IP and associated routing duties.

Due to limitations with keepalived, Neutron HA routers do not completely follow the VRRP networking conventions described up to this point. Neutron assigns a single virtual IP to an HA router, and that virtual IP is only configured on the master router in the group at any given time. While the address does fail over between routers during a failover event, it is not actually used as a gateway address for any network. As HA routers are created, a new virtual IP address is assigned to the respective group.



Neutron assigns virtual IP addresses from the 169.254.0.0/24 network by default. If an HA router's VRID is 5, then the assigned virtual IP would be 169.254.0.5. Using the VRID in the virtual IP assignment process assures that the address is consistent among HA router instances on different nodes without having to be stored in the database.

Instead of using virtual addresses for each connected subnet, Neutron uses the `virtual_ipaddress_excluded` configuration section found within the `keepalived` configuration file to specify routes, addresses, and their respective interfaces that should be configured when a router becomes master for the group. Likewise, the interface configuration will be removed once the router becomes a backup router. The following screenshot demonstrates various interfaces and routes that will be modified:

```
vrp_instance VR_1 {
    state BACKUP
    interface ha-4d66b09f-3b
    virtual_router_id 1
    priority 50
    garp_master_delay 60
    nopreempt
    advert_int 2
    track_interface {
        ha-4d66b09f-3b
    }
    virtual_ipaddress {
        169.254.0.1/24 dev ha-4d66b09f-3b
    }
    virtual_ipaddress_excluded {
        10.30.0.104/24 dev qg-3edbfff06-35
        192.168.200.1/24 dev qr-1eace53f-69
        fe80::f816:3eff:fe12:140c/64 dev qr-1eace53f-69 scope link
        fe80::f816:3eff:fe17:4f8/64 dev qg-3edbfff06-35 scope link
    }
    virtual_routes {
        0.0.0.0/0 via 10.30.0.1 dev qg-3edbfff06-35
    }
}
```

The `keepalived` configuration file for an HA router will be discussed in further detail later in this chapter.



The reason for this behavior is due to the `keepalived` service being limited to 20 configured virtual addresses, which could artificially limit the number of subnets attached to a Neutron router. The use of `virtual_ipaddress_excluded` is a known workaround of that limitation.

Determining the master router

In the following screenshot, an HA router without any connected gateway or project networks is scheduled across three L3 agents running on a single controller and two compute nodes:

```

root@controller01:~# ip netns exec qrouter-44e8f6a5-c9c6-44c1-854c-c503efccc1b2 ip addr show
..
2: ha-a4a3ef03-4a@if28: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:1c:0c:09 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 169.254.192.6/18 brd 169.254.255.255 scope global ha-a4a3ef03-4a
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe1c:c09/64 scope link
        valid_lft forever preferred_lft forever
..
BACKUP

root@compute01:~# ip netns exec qrouter-44e8f6a5-c9c6-44c1-854c-c503efccc1b2 ip addr show
..
2: ha-4d66b09f-3b@if17: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:f9:de:12 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 169.254.192.9/18 brd 169.254.255.255 scope global ha-4d66b09f-3b
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fef9:de12/64 scope link
        valid_lft forever preferred_lft forever
..
BACKUP

root@compute02:~# ip netns exec qrouter-44e8f6a5-c9c6-44c1-854c-c503efccc1b2 ip addr show
..
16: ha-80d242d7-b3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:79:e2:33 brd ff:ff:ff:ff:ff:ff
    inet 169.254.192.2/18 brd 169.254.255.255 scope global ha-80d242d7-b3
        valid_lft forever preferred_lft forever
    inet 169.254.0.1/24 scope global ha-80d242d7-b3 ← Virtual IP Address
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe79:e233/64 scope link
        valid_lft forever preferred_lft forever
..
MASTER

```



The Neutron L3 agent is commonly installed on controller or network nodes, but can be installed on compute nodes as well. This is especially true when distributed virtual routers are configured.

In the preceding example, one router acts as the master while the other two are relegated to backup duties. The HA interfaces are used for communication between the routers. At any given time, only the master router should have the virtual IP address 169.254.0.1 configured on its HA interface.

Installing and configuring additional L3 agents

To configure HA routers, two or more L3 agents are required. The L3 agent was installed on the `controller01` node in the previous chapter. On all remaining `compute` nodes, run the following command to install the L3 agent:

```
# apt install neutron-l3-agent
```

Defining an interface driver

Both the Linux bridge and Open vSwitch mechanism drivers support HA routers, and the Neutron L3 agent must be configured to use the interface driver that corresponds to the chosen mechanism driver.

Update the Neutron L3 configuration file on the `compute` nodes at `/etc/neutron/l3_agent.ini` and specify one of the following interface drivers:

On `compute01` running the Linux bridge agent:

```
[DEFAULT]
...
interface_driver = linuxbridge
```

On `compute02` and running the Open vSwitch agent:

```
[DEFAULT]
...
interface_driver = openvswitch
```

Setting the agent mode

The Neutron L3 agent considers HA routers as legacy routers, as many of the same mechanisms used for standalone routers are shared with HA routers. Therefore, the default value for `agent_mode` of `legacy` shall remain unchanged for the remainder of this chapter.

Restarting the Neutron L3 agent

After making changes to the configuration of the Neutron L3 agent, issue the following command on the compute nodes to restart the agent:

```
# systemctl restart neutron-l3-agent
```

After a restart of the services, the additional agents should check in. Use the following `openstack network agent list` command to return a listing of all L3 agents:

```
# openstack network agent list --agent-type l3
```

The output should resemble the following:

```
root@controller01:~# openstack network agent list --agent-type l3
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
ee380688-0ea7-4e54-9889-c44ddd933678	L3 agent	compute01	nova	:~)	UP	neutron-l3-agent
f9f8e9a3-4332-4693-a652-40837ffdbd51	L3 agent	compute02	nova	:~)	UP	neutron-l3-agent
7cecebf-a1da-479b-ab72-d4a18582f424	L3 agent	compute03	nova	:~)	UP	neutron-l3-agent
9c75984a-2cfa-4db6-afc6-b7e1e4901bbd	L3 agent	controller01	nova	:~)	UP	neutron-l3-agent

If an agent is not listed in the output as expected, troubleshoot any errors that may be indicated in the `/var/log/neutron/l3-agent.log` log file on the respective node.

Configuring Neutron

Neutron uses default settings to determine the type of routers that users are allowed to create. In addition, the settings also specify the number of routers that make up a VRRP group, the default HA network CIDR, and the default network type.

The following settings are specified within the Neutron configuration file at `/etc/neutron/neutron.conf`, and only need to be modified on the host(s) running the Neutron API service. In this environment, the `neutron-server` service runs on the `controller01` node:

```
# Enable HA mode for virtual routers. (boolean value)
#l3_ha = false
#
# Maximum number of L3 agents which a HA router will be scheduled
# on. If it is to 0 then the router will be scheduled on every
# agent. (integer value)
#max_l3_agents_per_router = 3
#
```



```
# Subnet used for the l3 HA admin network. (string value)
#l3_ha_net_cidr = 169.254.192.0/18
#
# The network type to use when creating the HA network for an HA
# router. By default or if empty, the first 'tenant_network_types'
# is used. This is helpful when the VRRP traffic should use a
# specific network which is not the default one. (string value)
#l3_ha_network_type =
#
```

To set HA routers as the default router type for projects, set the `l3_ha` configuration option to `True` in `neutron.conf`. For this demonstration, the default value of `False` is sufficient.



With a value of `False`, only users with the admin role can create HA routers. Ordinary users will be limited to the default router type, which in most cases is standalone.

To set a maximum number of L3 agents used for a virtual router, set `max_l3_agents_per_router` accordingly. For this demonstration, the default value is sufficient and will mean that three network namespaces will be constructed to make up the VRRP virtual router.

To modify the network type used for HA routers, change the value of `l3_ha_network_type`. By default, the default project network type is used and is sufficient for this demonstration.



If the default project network type is `vlan`, VLANs may be consumed at a faster-than-expected rate, as each project can consume a single VLAN for the HA network.

Once the changes have been made, restart the `neutron-server` service on the controller node for the changes to take effect.

Working with highly available routers

With a few exceptions, creating and managing an HA router is no different from its standalone counterpart. Neutron's router management commands were covered in the previous chapter, and the exceptions can be found in the following sections.

Creating highly-available routers

Users with the admin role can create highly-available routers using the `--ha` argument with the `openstack router create` command shown here:

```
openstack router create --ha ROUTER
```

Users without the admin role do not have the ability to override the default router type and cannot specify the `--ha` argument.

Deleting highly-available routers

To delete a highly-available router, simply use the `openstack router delete` command shown here:

```
openstack router delete ROUTER
```



When all HA routers in a project are removed, the HA network used for communication between routers stays behind. That network will be reused upon the creation of HA routers at a later time.

Decomposing a highly available router

Using concepts demonstrated in previous chapters, let's walk through the creation and decomposition of a highly available router. In the following example, I've started out with an external provider network named `GATEWAY_NET` and a project network named `PROJECT_NET`:

```
root@controller01:~# openstack network list
```

ID	Name	Subnets
758070f9-ecaf-4d2a-aa49-e119ce7943f6	GATEWAY_NET	0af5a767-fce0-4c4d-9df2-1aedbd259405
ce1fe7cd-7b01-4d58-b1c8-414886b7d8f2	PROJECT_NET	f29a2257-1283-4047-a835-b207480aa6f3

Using the `openstack router create` command with the `--ha` argument, we can create an HA router named `MyHighlyAvailableRouter`:

```
root@controller01:~# openstack router create --ha MyHighlyAvailableRouter
```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-04-27T13:39:23Z
description	
distributed	False
external_gateway_info	None
flavor_id	None
ha	True
id	44e8f6a5-c9c6-44c1-854c-c503efccc1b2
name	MyHighlyAvailableRouter
project_id	9233b6b4f6a54386af63c0a7b8f043c2
revision_number	None
routes	
status	ACTIVE
tags	
updated_at	2018-04-27T13:39:23Z

Upon creation of the HA router, a network namespace was created on up to three hosts running the Neutron L3 agent. In this demonstration, the L3 agent is running on the `controller01` and two compute nodes.

In the following screenshot, a router namespace that corresponds to the `MyHighlyAvailableRouter` router can be observed on each host:

```
root@controller01:~# ip netns
qrouter-44e8f6a5-c9c6-44c1-854c-c503efccc1b2 (id: 0)
qdhcp-ce1fe7cd-7b01-4d58-b1c8-414886b7d8f2 (id: 4)
qrouter-9ef2eeed-4a55-4f64-b8be-4b07a43ec373 (id: 3)

root@compute01:~# ip netns
qrouter-44e8f6a5-c9c6-44c1-854c-c503efccc1b2 (id: 0)

root@compute02:~# ip netns
qrouter-44e8f6a5-c9c6-44c1-854c-c503efccc1b2
```

Neutron automatically created a network reserved for communication between the routers upon creation of the first HA router within a project using the network defined by the `l3_ha_net_cidr` configuration option in the L3 agent configuration file:

```
root@controller01:~# openstack network list
```

ID	Name	Subnets
471ba196-e74e-412f-9a4f-390e0ed710c2	HA network tenant 9233b6b4f6a54386af63c0a7b8f043c2	4efc3fb0-645c-4372-8886-290c2398c458
758070f9-ecaf-4d2a-aa49-e119ce7943f6	GATEWAY_NET	0af5a767-fce0-4c4d-9df2-1aedbd259405
ce1fe7cd-7b01-4d58-b1c8-414886b7d8f2	PROJECT_NET	f29a2257-1283-4047-a835-b207480aa6f3

The HA network is not directly associated with the project and is not visible by anyone but an administrator, who is able to see all networks. The output here shows the details of the network:

```
root@controller01:~# openstack network show 471ba196-e74e-412f-9a4f-390e0ed710c2
```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	nova
created_at	2018-04-27T13:39:22Z
description	
dns_domain	None
id	471ba196-e74e-412f-9a4f-390e0ed710c2
ipv4_address_scope	None
ipv6_address_scope	None
is_default	None
is_vlan_transparent	None
mtu	1500
name	HA network tenant 9233b6b4f6a54386af63c0a7b8f043c2
port_security_enabled	True
project_id	Not Used
provider:network_type	vlan
provider:physical_network	physnet1
provider:segmentation_id	40
qos_policy_id	None
revision_number	3
router:external	Internal
segments	None
shared	False
status	ACTIVE
subnets	4efc3fb0-645c-4372-8886-290c2398c458
tags	
updated_at	2018-04-27T13:39:23Z

As you can see, the `project_id` field is blank, while the name of the HA network includes the ID of the project. The network is used by Neutron for all HA routers created by that project in the future.



The HA network utilizes the default project network type and will consume a segmentation ID of that type.

Both a gateway and internal interface have been attached to the router. The `openstack port list` command reveals the gateway, internal, and HA ports associated with the router:

```
root@controller01:~# openstack port list --router 44e8f6a5-c9c6-44c1-854c-c503efccclb2
```

ID	Name	MAC Address	Fixed IP Addresses	Status
1eace53f-69ab-4ccd-991d-...	Internal Network Interface	fa:16:3e:12:14:0c	ip_address='192.168.200.1', subnet_id='f29a2257-1283-4047-ab35-b207480aa6f3'	ACTIVE
3edbf06-351e-4ed1-8153-...	External Network Interface	fa:16:3e:17:04:f8	ip_address='10.30.0.104', subnet_id='0af5a767-fce0-4c4d-9df2-1aedbd259405'	ACTIVE
4d66b09f-3b4a-4808-8dfe-...	HA port tenant 9233b6b4f6a54386af63c0a7b8f043c2	fa:16:3e:f9:de:12	ip_address='169.254.192.9', subnet_id='4efc3fb0-645c-4372-8886-290c2398c458'	ACTIVE
88d242d7-b3d7-4348-95e8-...	HA port tenant 9233b6b4f6a54386af63c0a7b8f043c2	fa:16:3e:79:e2:33	ip_address='169.254.192.2', subnet_id='4efc3fb0-645c-4372-8886-290c2398c458'	ACTIVE
a4a3ef03-4a58-42c7-a65a-...	HA port tenant 9233b6b4f6a54386af63c0a7b8f043c2	fa:16:3e:1c:0c:09	ip_address='169.254.192.6', subnet_id='4efc3fb0-645c-4372-8886-290c2398c458'	ACTIVE

The three HA ports were created automatically by Neutron and are used for communication between the router namespaces distributed across the hosts. Inside the network namespaces, we can find the corresponding interfaces:

```
root@controller01:~# ip netns exec qrouter-44e8f6a5-c9c6-44c1-854c-c503efccclb2 ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
   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
2: ha-a4a3ef03-4a@if28: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
   link/ether fa:16:3e:1c:0c:09 brd ff:ff:ff:ff:ff:ff link-netnsid 0
   inet 169.254.192.6/18 brd 169.254.255.255 scope global ha-a4a3ef03-4a
       valid_lft forever preferred_lft forever
   inet6 fe80::f816:3eff:fe1c:09/64 scope link
       valid_lft forever preferred_lft forever
3: qr-1eace53f-69@if31: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
   link/ether fa:16:3e:12:14:0c brd ff:ff:ff:ff:ff:ff link-netnsid 0
4: qg-3edbf06-35@if32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
   link/ether fa:16:3e:17:04:f8 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```



```

root@compute01:~# ip netns exec qrouter-44e8f6a5-c9c6-44c1-854c-c503efccccb2 ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
2: ha-4d66b09f-3b@if17: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:f9:de:12 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 169.254.192.9/18 brd 169.254.255.255 scope global ha-4d66b09f-3b
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe9:de12/64 scope link
        valid_lft forever preferred_lft forever
3: qr-1eace53f-69@if20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:12:14:0c brd ff:ff:ff:ff:ff:ff link-netnsid 0
4: qg-3edbff06-35@if21: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:17:04:f8 brd ff:ff:ff:ff:ff:ff link-netnsid 0

```

```

root@compute02:~# ip netns exec qrouter-44e8f6a5-c9c6-44c1-854c-c503efccccb2 ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
16: ha-80d242d7-b3 <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:79:e2:33 brd ff:ff:ff:ff:ff:ff
    inet 169.254.192.2/18 brd 169.254.255.255 scope global ha-80d242d7-b3
        valid_lft forever preferred_lft forever
    inet 169.254.0.1/24 scope global ha-80d242d7-b3
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe79:e233/64 scope link
        valid_lft forever preferred_lft forever
17: qr-1eace53f-69: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:12:14:0c brd ff:ff:ff:ff:ff:ff
    inet 192.168.200.1/24 scope global qr-1eace53f-69
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe12:140c/64 scope link
        valid_lft forever preferred_lft forever
18: qg-3edbff06-35: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:17:04:f8 brd ff:ff:ff:ff:ff:ff
    inet 10.30.0.104/24 scope global qg-3edbff06-35
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe17:4f8/64 scope link
        valid_lft forever preferred_lft forever

```

VRRP
Virtual IP

Active qr
interface

Active qg
interface

In the preceding screenshots, the router namespace on `compute02` has been elected as the master router, as evidenced by the virtual IP, `169.254.0.1/24`, being configured on the HA interface within the namespace. In addition to the HA interface, the `qg` and `qr` interfaces are only fully configured on the master router. If more than one router owns the virtual IP, or if you see the `qg` and `qr` interfaces fully configured on more than one router, there may be communication issues between the routers on the HA network.



Based on Neutron's configuration of `keepalived`, the virtual IP is not used as a gateway address and is only used as a standardized address that can failover to other routers as part of the VRRP failover mechanisms. Neutron does not treat addresses on the `qg` or `qr` interfaces as virtual IPs. Along with the virtual IP, the `qg` and `qr` interfaces should only be configured and active on the master router at any given time.

You can use the `openstack network agent list` command with the `--router` and `--long` arguments to determine the agent(s) a router is scheduled to as well as see the HA state for the given agent, if applicable:

```
root@controller01:~# openstack network agent list --router 44e8f6a5-c9c6-44c1-854c-c503efccc1b2 --long
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary	HA State
ee380688-0ea7-4e54-9889-c44ddd933678	L3 agent	compute01	nova	:-)	UP	neutron-l3-agent	standby
f9f8e9a3-4332-4693-a652-40837ffdbd51	L3 agent	compute02	nova	:-)	UP	neutron-l3-agent	active
9c75984a-2cfa-4db6-afc6-b7e1e4901bbd	L3 agent	controller01	nova	:-)	UP	neutron-l3-agent	standby

Examining the keepalived configuration

Neutron has configured `keepalived` in each `qrouter` namespace so that, together, the namespaces can act as a single virtual router. A `keepalived` service runs in each namespace and uses the following configuration file located on the underlying host running the L3 agent:

```
/var/lib/neutron/ha_confs/<router_id>/keepalived.conf
```

A look at the configuration file for the active router on `compute02` shows the `keepalived` and VRRP configurations currently in use:

```
root@compute02:~# cat /var/lib/neutron/ha_confs/44e8f6a5-c9c6-44c1-854c-c503efcc1b2/keepalived.conf
global_defs {
    notification_email_from neutron@openstack.local
    router_id neutron
}
vrrp_instance VR_1 {
    state BACKUP
    interface ha-80d242d7-b3
    virtual_router_id 1
    priority 50
    garp_master_delay 60
    nopreempt
    advert_int 2
    track_interface {
        ha-80d242d7-b3
    }
    virtual_ipaddress {
        169.254.0.1/24 dev ha-80d242d7-b3
    }
    virtual_ipaddress_excluded {
        10.30.0.104/24 dev qg-3edbff06-35
        192.168.200.1/24 dev qr-leace53f-69
        fe80::f816:3eff:fe12:140c/64 dev qr-leace53f-69 scope link
        fe80::f816:3eff:fe17:4f8/64 dev qg-3edbff06-35 scope link
    }
    virtual_routes {
        0.0.0.0/0 via 10.30.0.1 dev qg-3edbff06-35
    }
}
```

Annotations in the original image:

- `virtual_router_id 1`: Virtual Router ID (VRID)
- `priority 50`: Router priority
- `nopreempt`: Preempt mode
- `advert_int 2`: Advertisement interval
- `169.254.0.1/24 dev ha-80d242d7-b3`: Virtual IP address
- `10.30.0.104/24 dev qg-3edbff06-35`: External IP to be configured when master
- `192.168.200.1/24 dev qr-leace53f-69`: Internal IP to be configured when master
- `0.0.0.0/0 via 10.30.0.1 dev qg-3edbff06-35`: Route to be configured when master

Executing a failover

Under normal circumstances, a node is not likely to fail, and router failover is unlikely to occur. A failure scenario can be recreated by manually rebooting the node hosting the active router or by putting its physical interfaces into a down state.

Failover actions are logged in the following location within the router namespaces:

```
/var/lib/neutron/ha_confs/<router_id>/neutron-keepalived-state-change.log
```

An example of a router going from a backup state to master state once failure is detected can be observed in the following screenshot:

```
root@controller01:~# tail -f /var/lib/neutron/ha_confs/44e8f6a5-c9c6-44c1-854c-c503efcc1b2/neutron-keepalived-state-change.log
2018-04-27 13:39:46.177 12167 INFO neutron.common.config [-] Logging enabled!
2018-04-27 13:39:46.179 12167 INFO neutron.common.config [-] /usr/bin/neutron-keepalived-state-change version 11.0.3
...
root@compute01:~# tail -f /var/lib/neutron/ha_confs/44e8f6a5-c9c6-44c1-854c-c503efcc1b2/neutron-keepalived-state-change.log
2018-04-27 13:39:42.903 18938 INFO neutron.common.config [-] Logging enabled!
2018-04-27 13:39:42.904 18938 INFO neutron.common.config [-] /usr/bin/neutron-keepalived-state-change version 11.0.3
...
2018-04-27 16:38:59.505 18986 DEBUG neutron.agent.l3.keepalived_state_change [-] Wrote router 44e8f6a5-c9c6-44c1-854c-c503efcc1b2 state master write_state_change ...
2018-04-27 16:39:00.497 18986 DEBUG neutron.agent.l3.keepalived_state_change [-] Notified agent router 44e8f6a5-c9c6-44c1-854c-c503efcc1b2, state master notify_agent ...
```


In the preceding screenshot, the router on `compute01` became the new master router. In the log, this was identified by the `state_master` message. Once the former master router regains connectivity and detects a new master has been elected, it moves to a backup state, as indicated by the `state_backup` message seen here:

```
root@compute02:~# tail -f /var/lib/neutron/ha_confs/44e8f6a5-c9c6-44c1-854c-c503efccclb2/neutron-keepalived-state-change.log
2018-04-27 18:21:44.485 27028 DEBUG neutron.agent.l3.keepalived_state_change [-] Wrote router 44e8f6a5-c9c6-44c1-854c-c503efccclb2 state_backup write_state_change ...
2018-04-27 18:21:45.526 27028 DEBUG neutron.agent.l3.keepalived_state_change [-] Notified agent router 44e8f6a5-c9c6-44c1-854c-c503efccclb2, state_backup notify_agent ...
```

Summary

Highly available routers can be created and managed using the same router command set discussed in the previous chapter. Neutron L3 agents are responsible for configuring the routers in a VRRP group, and the routers are left to elect a master router and implement their respective keepalived configuration based on their master or backup state at that time.

While HA routers provide a level of redundancy over their standalone counterparts, they are not without their drawbacks. A single node hosting a master router is still a bottleneck for traffic traversing that router. In addition, if the network used for dedicated VRRP traffic between routers experiences a loss of connectivity, the routers can become split-brained. This can cause two or more routers to become master routers and potentially cause ARP and MAC flapping issues in the network. Connection tracking between routers has not been implemented as of the Pike release, which means that connections to and from instances may be severed during a failover event. These shortcomings are being actively worked on and look to be addressed in future releases of OpenStack.

In the next chapter, we will look at how virtual routers can be distributed across `compute` nodes and serve as the gateway for their respective instances using a technology referred to as Distributed Virtual Routers, or DVR.

12

Distributed Virtual Routers

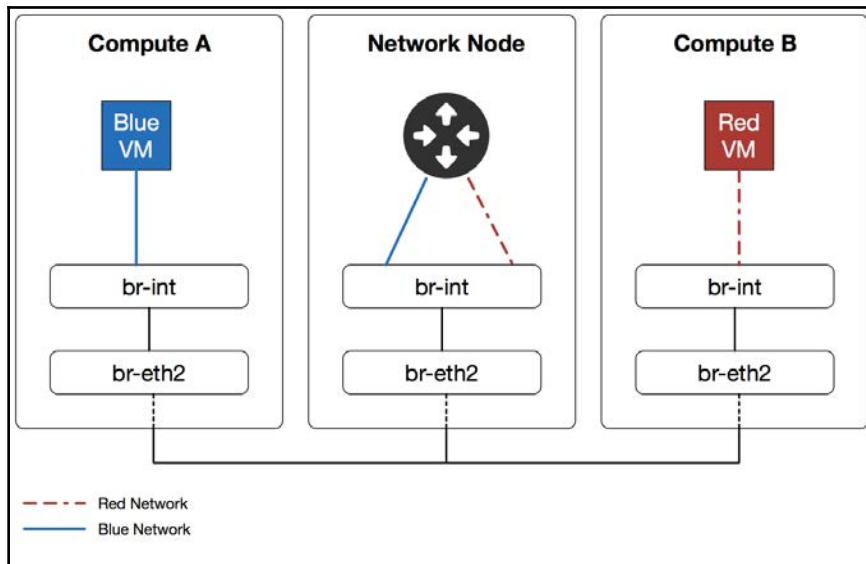
Prior to the introduction of Neutron in the Folsom release of OpenStack, all network management was built in to the Nova API and was known as nova-network. Nova-network provided floating IP functionality, and network failure domains were limited to an individual compute node – something that was lacking in the early releases of Neutron. Nova-network has since been deprecated and most of its functionality has been implemented and improved upon in the latest releases of Neutron. In the last chapter, we looked at using VRRP to provide high-availability using active-standby routers. In this chapter, we will look at how distributed virtual routers borrow many concepts from the nova-network multi-host model to provide high-availability and smaller network failure domains while retaining support for many of the advanced networking features provided by Neutron.

Legacy routers, including standalone and active-standby, are compatible with multiple mechanism drivers, including the Linux bridge and Open vSwitch drivers. Distributed virtual routers, on the other hand, require Open vSwitch and are only supported by the Open vSwitch mechanism driver and agent. Other drivers and agents, such as those for OVN or OpenContrail, may provide similar distributed routing functionality, but are out of the scope of this book.

Distributing routers across the cloud

Much like nova-network did with its multi-host functionality, Neutron can distribute a virtual router across compute nodes in an effort to isolate the failure domain to a particular compute node rather than a centralized network node. By eliminating a centralized Layer 3 agent, routing that was performed on a single node is now handled by the compute nodes themselves.

Legacy routing using a centralized network node resembles the following diagram:



In the legacy model, traffic from the blue virtual machine to the red virtual machine on a different network would traverse a centralized network node hosting the router. If the node hosting the router were to fail, traffic between the instances and external networks or the instances themselves would be dropped.

In this chapter, I will discuss the following:

- Installing and configuring additional L3 agents to support distributed virtual routers
- Demonstrating the creation and management of a distributed virtual router
- Routing between networks behind the same router
- Outbound connectivity using SNAT
- Inbound and outbound connectivity using floating IPs

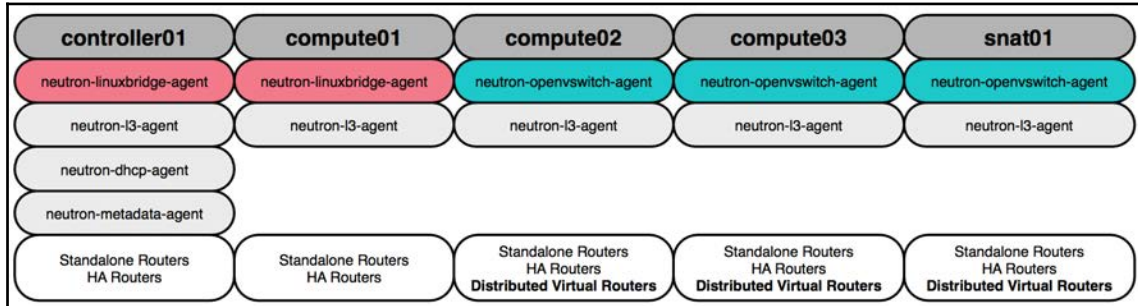
Installing and configuring Neutron components

To configure distributed virtual routers, there are a few requirements that must be met, including the following:

- ML2 plugin
- L2 population mechanism driver
- Open vSwitch mechanism driver
- Layer 3 agent installed on all networks and compute nodes

In the environment built out in this book, a single controller node handles OpenStack API services, DHCP and metadata services, and runs the Linux bridge agent for use with standalone and HA routers. `compute01` runs the Linux bridge agent, while `compute02` and `compute03` run the Open vSwitch agent. Another node, `snat01`, will run the Open vSwitch agent and be responsible for outbound SNAT traffic when distributed virtual routers are used.

A diagram of this configuration can be seen here:



While the mixing of drivers and agents between nodes is possible thanks to the ML2 core plugin, a design like this is not typical in a production environment. Instances deployed on `compute01` may experience connectivity issues if deployed on a network using a distributed virtual router.

Installing additional L3 agents

Run the following command on `snat01` to install the L3 agent:

```
# apt install neutron-l3-agent
```

Defining an interface driver

Open vSwitch and the Open vSwitch mechanism driver are required to enable and utilize distributed virtual routers.

Update the Neutron L3 agent configuration file at `/etc/neutron/l3_agent.ini` on `snat01` and specify the following interface driver:

```
[DEFAULT]
...
interface_driver = openvswitch
```

Enabling distributed mode

The ML2 plugin is required to operate distributed virtual routers and must be configured accordingly.

Update the OVS configuration file at `/etc/neutron/plugins/ml2/openvswitch_agent.ini` on `compute02`, `compute03`, and `snat01` to enable the OVS agent to support distributed virtual routing and L2 population:

```
[agent]
...
enable_distributed_routing = True
l2_population = True
```

Setting the agent mode

When using distributed virtual routers, a node can operate in one of two modes: `dvr` or `dvr_snat`. A node configured in `dvr_snat` mode handles north-south SNAT traffic, while a node in `dvr` mode handles north-south DNAT (for example, floating IP) traffic and east-west traffic between instances.

Compute nodes running the Open vSwitch agent run in `dvr` mode. A centralized network node typically runs in `dvr_snat` mode, and can potentially acts as a single point of failure for the network for instances not leveraging floating IPs.



Neutron supports deploying highly-available `dvr_snat` nodes using VRRP, but doing so is outside the scope of this book.

In this book, the `snat01` node will be dedicated to handling SNAT traffic when using distributed virtual routers, as `controller01` has been configured with the Linux bridge agent and is not an eligible host for DVR-related functions.

On the `snat01` node, configure the L3 agent to operate in `dvr_snat` mode by modifying the `agent_mode` option in the L3 agent configuration file:

```
[DEFAULT]
...
agent_mode = dvr_snat
```

On the `compute02` and `compute03` nodes, modify the L3 agent to operate in `dvr` mode from `legacy` mode:

```
[DEFAULT]
...
agent_mode = dvr
```

On the `snat01`, `compute02`, and `compute03` nodes, set the `handle_internal_only_routers` configuration option to `false`:

```
[DEFAULT]
...
handle_internal_only_routers = false
```

Configuring Neutron

Neutron uses default settings to determine the type of routers that users are allowed to create as well as the number of routers that should be deployed across L3 agents.

The following default settings are specified within the `neutron.conf` configuration file and only need to be modified on the host running the Neutron API service. In this environment, the `neutron-server` service runs on the `controller01` node. The default values can be seen here:

```
# System-wide flag to determine the type of router
# that tenants can create.
# Only admin can override. (boolean value)
#
# router_distributed = false
```

To set distributed routers as the default router type for projects, set the `router_distributed` configuration option to `true`. For this demonstration, the default value of `false` is sufficient.

Once the changes have been made, restart the `neutron-server` service on `controller01` for the changes to take effect:

```
# systemctl restart neutron-server
```

Restarting the Neutron L3 and Open vSwitch agent

After making changes to the configuration of the Neutron L3 and L2 agents, issue the following command on `compute02`, `compute03`, and `snat01` to restart the respective agents:

```
# systemctl restart neutron-l3-agent neutron-openvswitch-agent
```

After a restart of the services, the additional agents should check in. Use the following `openstack network agent list` command to return a listing of all L3 agents:

```
# openstack network agent list --agent-type="l3"
```

The output should resemble the following:

```
root@controller01:~# openstack network agent list --agent-type="l3"
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
ee380688-0ea7-4e54-9889-c44ddd933678	L3 agent	compute01	nova	:~)	UP	neutron-l3-agent
f9f8e9a3-4332-4693-a652-40837ffdbd51	L3 agent	compute02	nova	:~)	UP	neutron-l3-agent
7cdeb762-ab03-41a3-b924-9e64dde50621	L3 agent	compute03	nova	:~)	UP	neutron-l3-agent
9c75984a-2cfa-4db6-afc6-b7e1e4901bbd	L3 agent	controller01	nova	:~)	UP	neutron-l3-agent
63214965-cd41-4aa1-a98f-a3e202929106	L3 agent	snat01	nova	:~)	UP	neutron-l3-agent

If an agent is not listed in the output as expected, troubleshoot any errors that may be indicated in the `/var/log/neutron/l3-agent.log` log file on the respective node.

Managing distributed virtual routers

With a few exceptions, managing a distributed router is no different from its standalone counterpart. Neutron's router management commands were covered in *Chapter 10, Creating Standalone Routers with Neutron*. The exceptions are covered in the following section.

Creating distributed virtual routers

Users with the admin role can create distributed virtual routers using the `--distributed` argument with the `openstack router create` command, as shown here:

```
openstack router create --distributed NAME
```

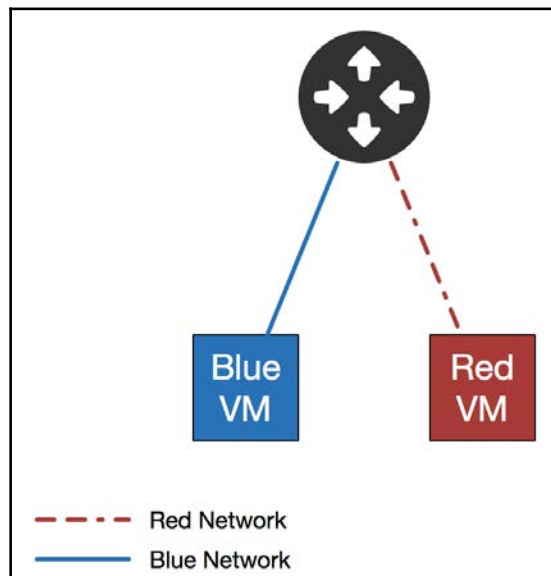
Users without the admin role are limited the router type specified by the `router_distributed` configuration option in the Neutron configuration file. Users do not have the ability to override the default router type and cannot specify the `--distributed` argument.

Routing east-west traffic between instances

In the network world, east-west traffic is traditionally defined as server-to-server traffic. In Neutron, as it relates to distributed virtual routers, east-west traffic is traffic between instances in different networks owned by the same project. In Neutron's legacy routing model, traffic between different networks traverses a virtual router located on a centralized network node. With DVR, the same traffic avoids the network node and is routed directly between the compute nodes hosting the virtual machine instances.

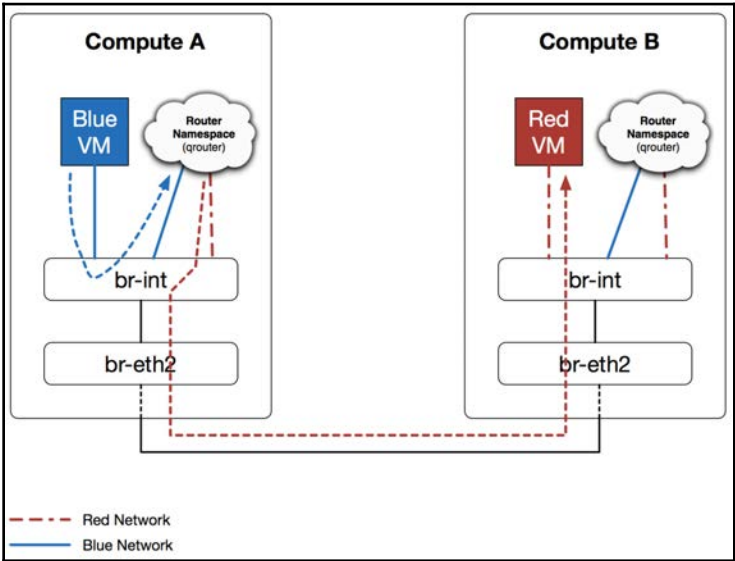
Reviewing the topology

Logically speaking, a distributed virtual router is a single router object connecting two or more project networks, as shown in the following diagram:



In the following example, a distributed virtual router named `MyDistributedRouter` has been created and connected to two project networks: `BLUE_NET` and `RED_NET`. Virtual machine instances in each network use their respective default gateways to route traffic to the other network through the same router. The virtual machine instances are unaware of where the router is located.

A look under the hood, however, tells a different story. In the following example, the blue VM pings the red VM and traffic is routed and forwarded accordingly:



As far as the user is concerned, the router connecting the two networks is a single entity known as MyDistributedRouter:

```
root@controller01:~# openstack router list
```

ID	Name	Status	State	Distributed	HA	Project
9822b80c-7548-43f7-8a03-70c5baf6c9c0	MyDistributedRouter	ACTIVE	UP	True	False	9233b6b4f6a54386af63c0a7b8f043c2

Using the `ip netns exec` command, we can see that the `qr` interfaces within the namespaces on each compute node and the `SNAT` node share the same interface names, IP addresses, and MAC addresses:

```
root@compute02:~# ip netns exec qrouter-9822b80c-7548-43f7-8a03-70c5baf6c9c0 ip a
...
30: qr-841d9818-bf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:73:ce:50 brd ff:ff:ff:ff:ff:ff
    inet 192.168.150.1/24 brd 192.168.150.255 scope global qr-841d9818-bf
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe73:ce50/64 scope link
        valid_lft forever preferred_lft forever
31: qr-d2ce8f82-d8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:56:50:10 brd ff:ff:ff:ff:ff:ff
    inet 172.16.0.1/24 brd 172.16.0.255 scope global qr-d2ce8f82-d8
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe56:5010/64 scope link
        valid_lft forever preferred_lft forever

root@compute03:~# ip netns exec qrouter-9822b80c-7548-43f7-8a03-70c5baf6c9c0 ip a
...
18: qr-841d9818-bf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:73:ce:50 brd ff:ff:ff:ff:ff:ff
    inet 192.168.150.1/24 brd 192.168.150.255 scope global qr-841d9818-bf
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe73:ce50/64 scope link
        valid_lft forever preferred_lft forever
19: qr-d2ce8f82-d8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:56:50:10 brd ff:ff:ff:ff:ff:ff
    inet 172.16.0.1/24 brd 172.16.0.255 scope global qr-d2ce8f82-d8
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe56:5010/64 scope link
        valid_lft forever preferred_lft forever

root@snat01:~# ip netns exec qrouter-9822b80c-7548-43f7-8a03-70c5baf6c9c0 ip a
...
17: qr-841d9818-bf: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:73:ce:50 brd ff:ff:ff:ff:ff:ff
    inet 192.168.150.1/24 brd 192.168.150.255 scope global qr-841d9818-bf
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe73:ce50/64 scope link
        valid_lft forever preferred_lft forever
19: qr-d2ce8f82-d8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:56:50:10 brd ff:ff:ff:ff:ff:ff
    inet 172.16.0.1/24 brd 172.16.0.255 scope global qr-d2ce8f82-d8
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe56:5010/64 scope link
        valid_lft forever preferred_lft forever
```

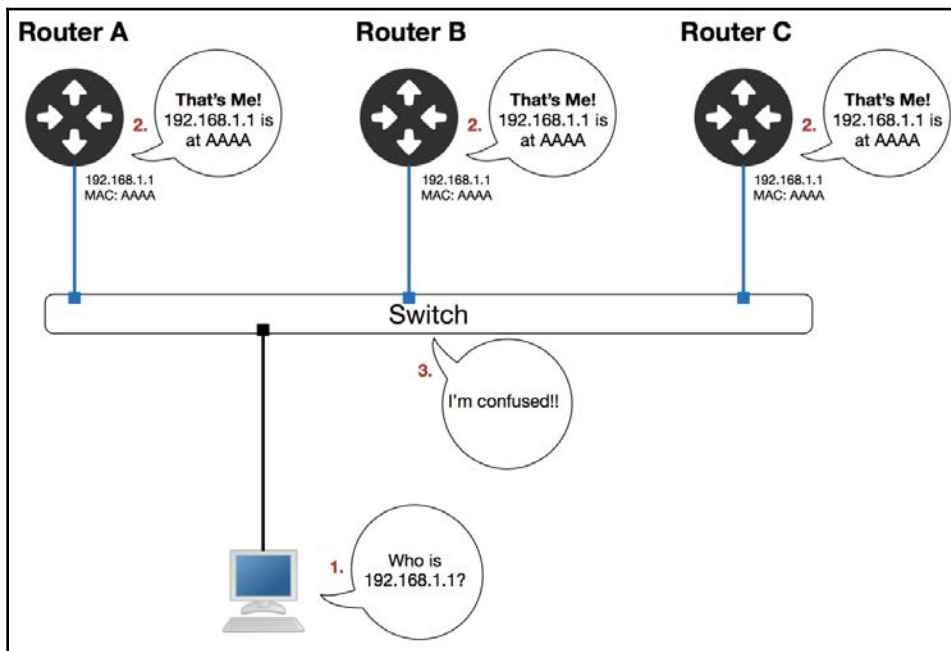
In the preceding screenshot, the `qrouter` namespaces on the `snat01` and `compute` nodes that correspond to the distributed router contain the same `qr-841d9818-bf` and `qr-d2ce8f82-d8` interfaces and addresses that correspond to the `BLUE_NET` and `RED_NET` networks. A creative use of routing tables and Open vSwitch flow rules allows traffic between instances behind the same distributed router to be routed directly between compute nodes. The tricks behind this functionality will be discussed in the following sections and throughout this chapter.

Plumbing it up

When a distributed virtual router is connected to a subnet using the OpenStack `router add subnet` or `router add port` commands, the router is scheduled to all nodes hosting ports on the subnet and running the Open vSwitch agent, including any controller or network node hosting DHCP or load balancer namespaces and any compute node hosting virtual machine instances in the subnet. The L3 agents are responsible for creating the respective `qrouter` network namespace on each node, and the Open vSwitch agent connects the router interfaces to the bridges and configures the appropriate flows.

Distributing router ports

Without proper precautions, distributing ports with the same IP and MAC addresses across multiple compute nodes presents major issues in the network. Imagine a physical topology that resembles the following diagram:



In most networks, an environment consisting of multiple routers with the same IP and MAC address connected to a switch would result in the switches learning and relearning the location of the MAC addresses across different switch ports. This behavior is often referred to as MAC flapping and results in network instability and unreliability.

Virtual switches can exhibit the same behavior regardless of segmentation type, as the virtual switch may learn that a MAC address exists both locally on the compute node and remotely, resulting in similar behavior that is observed on the physical switch.

Making it work

To work around this expected network behavior, Neutron allocates a unique MAC address to each compute node that is used whenever traffic from a distributed virtual router leaves the node. The following screenshot shows the unique MAC addresses that have been allocated to the nodes in this demonstration:

```
root@controller01:~# mysql -e 'use neutron; select * from dvr_host_macs;'
```

host	mac_address
snat01	FA-16-3F-75-10-80
compute03	FA-16-3F-8B-A5-69
compute02	FA-16-3F-C9-6D-0F

Open vSwitch flow rules are used to rewrite the source MAC address of a packet as it leaves a router interface with the unique MAC address allocated to the respective host. In the following screenshot, a look at the flows on the provider bridge of `compute02` demonstrates the rewriting of the non-unique `qr` interface MAC address with the unique MAC address assigned to `compute02`:

```
root@compute02:~# ovs-ofctl dump-flows br-eth2
```

Source is non-unique qr MAC address Modify to unique host MAC address

```
...
table=1, n_packets=2, n_bytes=260, priority=1,dl_vlan=2,dl_src=fa:16:3e:56:50:10 actions=mod_dl_src:fa:16:3f:c9:6d:0f, resubmit(,2)
table=1, n_packets=2, n_bytes=260, priority=1,dl_vlan=1,dl_src=fa:16:3e:73:ce:50 actions=mod_dl_src:fa:16:3f:c9:6d:0f, resubmit(,2)
table=1, n_packets=8, n_bytes=512, priority=0 actions=resubmit(,2)
table=2, n_packets=2, n_bytes=260, priority=4,in_port="phy-br-eth2",dl_vlan=2 actions=mod_vlan_vid:42,NORMAL
table=2, n_packets=2, n_bytes=260, priority=4,in_port="phy-br-eth2",dl_vlan=1 actions=mod_vlan_vid:41,NORMAL
table=2, n_packets=8, n_bytes=512, priority=4,in_port="phy-br-eth2",dl_vlan=3 actions=mod_vlan_vid:30,NORMAL
table=2, n_packets=0, n_bytes=0, priority=2,in_port="phy-br-eth2" actions=drop
table=3, n_packets=4, n_bytes=536, priority=2,dl_src=fa:16:3f:75:10:80 actions=output:"phy-br-eth2"
table=3, n_packets=4, n_bytes=456, priority=2,dl_src=fa:16:3f:8b:a5:69 actions=output:"phy-br-eth2"
table=3, n_packets=16444, n_bytes=2558076, priority=1 actions=NORMAL
...
```

Likewise, when traffic comes in to a compute node that matches a local virtual machine instance's MAC address and segmentation ID, the source MAC address is rewritten from the unique source host MAC address to the local instance's gateway MAC address:

```

root@compute02:~# ovs-ofctl dump-flows br-int
...
table=0, n_packets=4, n_bytes=536, priority=4, in_port="int-br-eth2", dl_src=fa:16:3f:75:10:80 actions=resubmit(,2)
table=0, n_packets=0, n_bytes=0, priority=2, in_port="patch-tun", dl_src=fa:16:3f:75:10:80 actions=resubmit(,1)
table=0, n_packets=4, n_bytes=456, priority=4, in_port="int-br-eth2", dl_src=fa:16:3f:8b:a5:69 actions=resubmit(,2)
table=0, n_packets=0, n_bytes=0, priority=2, in_port="patch-tun", dl_src=fa:16:3f:8b:a5:69 actions=resubmit(,1)
table=0, n_packets=16, n_bytes=1304, priority=3, in_port="int-br-eth2", dl_vlan=42 actions=mod_vlan_vid:2, resubmit(,60)
table=0, n_packets=16, n_bytes=1304, priority=3, in_port="int-br-eth2", dl_vlan=41 actions=mod_vlan_vid:1, resubmit(,60)
table=0, n_packets=69, n_bytes=6186, priority=3, in_port="int-br-eth2", dl_vlan=30 actions=mod_vlan_vid:3, resubmit(,60)
table=0, n_packets=24, n_bytes=1536, priority=0 actions=resubmit(,60)
table=1, n_packets=0, n_bytes=0, priority=1 actions=drop
table=2, n_packets=0, n_bytes=0, priority=4, dl_vlan=41, dl_dst=fa:16:3e:3b:c9:4b actions=mod_dl_src:fa:16:3e:73:ce:50, resubmit(,60)
table=2, n_packets=8, n_bytes=992, priority=1 actions=drop
...

```

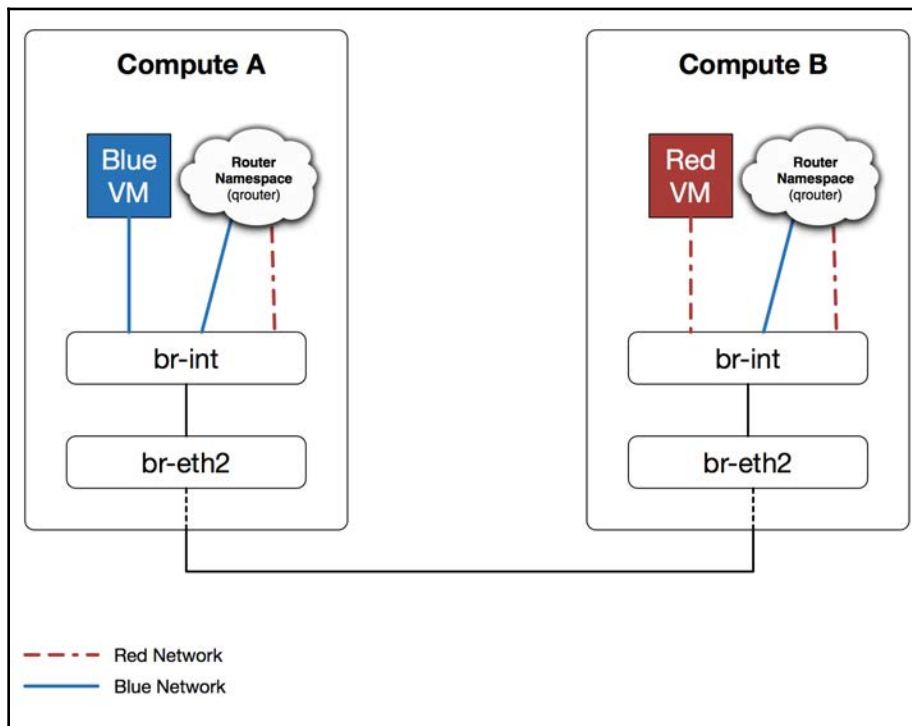
When traffic arrives from provider bridge from a unique host MAC, go to table 2

If the dest MAC is that of BLUE_VM, modify the src MAC to that of the qr interface of the router interface on compute02

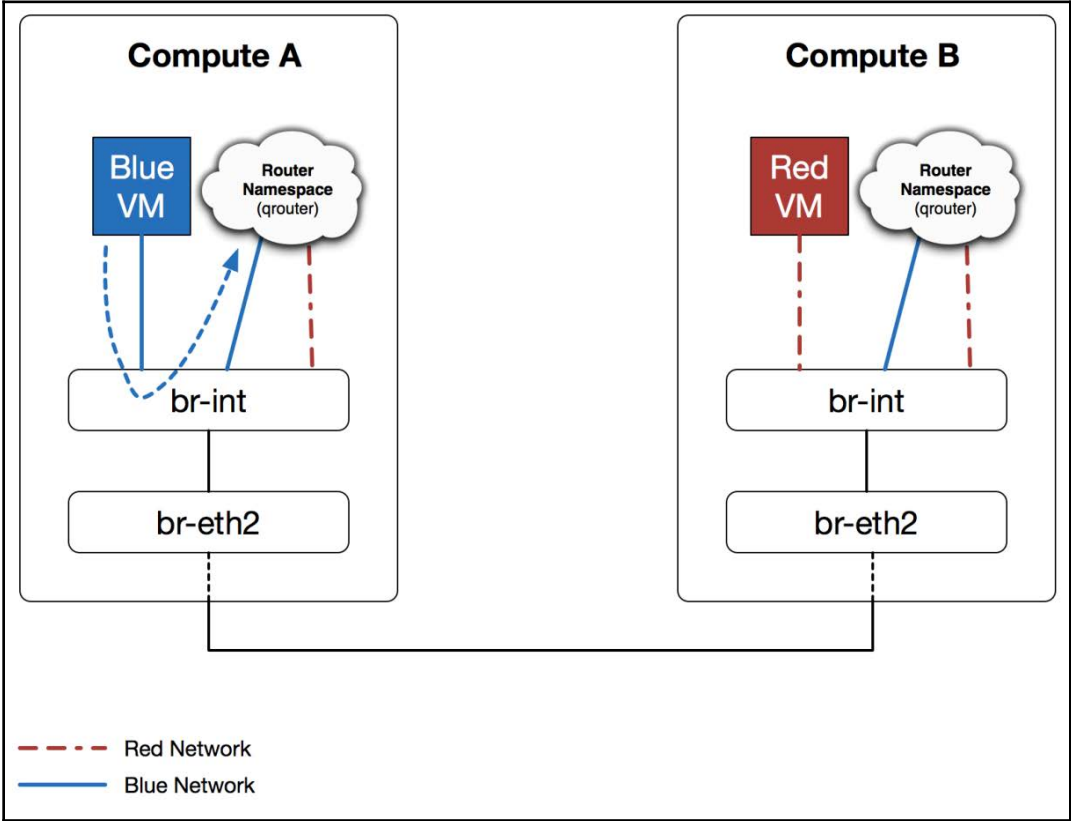
Because the Layer 2 header rewrites occur before traffic enters and after traffic leaves the virtual machine instance, the instance is unaware of the changes made to the frames and operates normally. The following section demonstrates this process in further detail.

Demonstrating traffic between instances

Imagine a scenario where virtual machines in different networks exist on two different compute nodes, as demonstrated in the following diagram:



Traffic from the blue virtual machine instance on **Compute A** to the red virtual machine instance on **Compute B** will first be forwarded from the instance to its local gateway through the integration bridge and to the router namespace, as shown here:

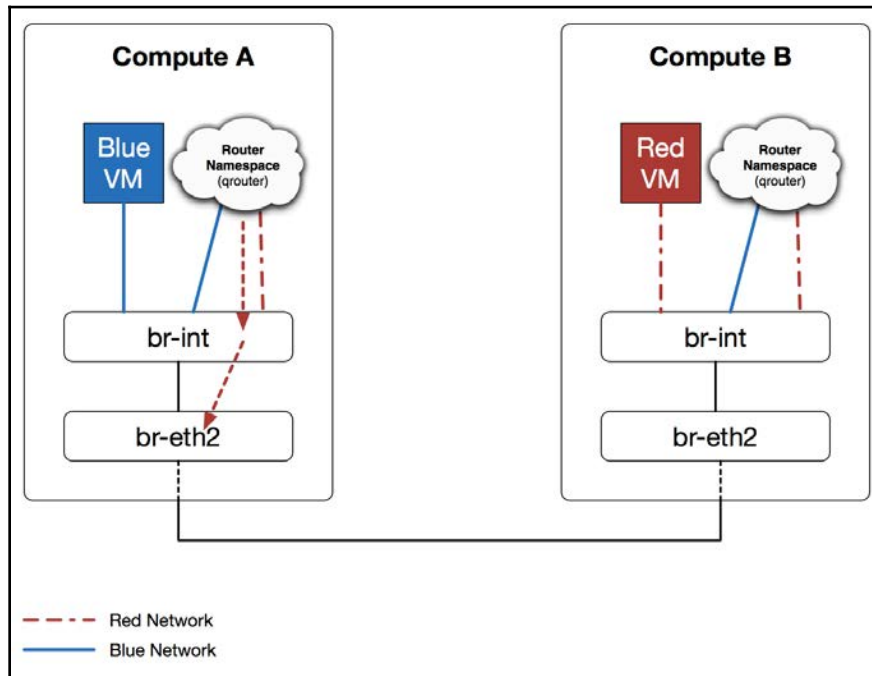


Source MAC	Destination MAC	Source IP	Destination IP
Blue VM	Blue router interface	Blue VM	Red VM

The router on **Compute A** will route the traffic from the **blue VM** to the **red VM**, replacing the source MAC address with its red interface and the destination MAC address to that of the **red VM** in the process:

Source MAC	Destination MAC	Source IP	Destination IP
Red router interface	Red VM	Blue VM	Red VM

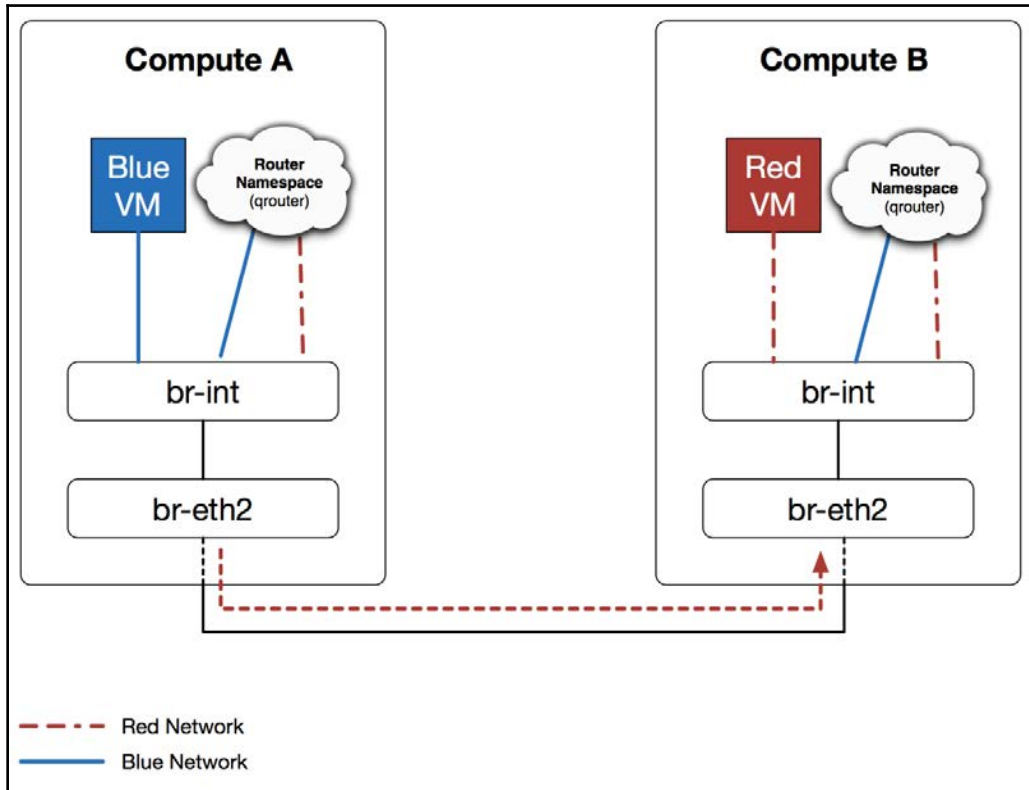
The router then sends the packet back to the integration bridge, which then forwards it to the provider bridge, as shown here:



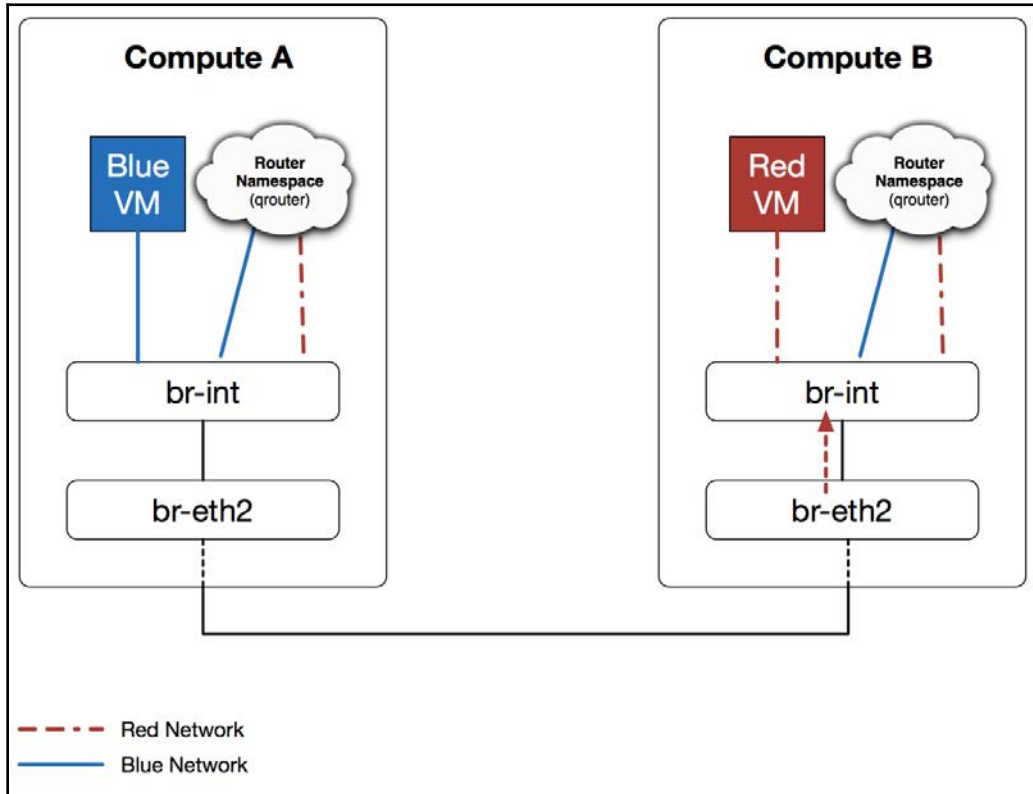
As traffic arrives at the provider bridge of **ComputeA**, a series of flow rules are processed, resulting in the source MAC address being changed from the red interface of the router to the unique MAC address of the host:

Source MAC	Destination MAC	Source IP	Destination IP
Source host (Compute A)	Red VM	Blue VM	Red VM

The traffic is then forwarded out onto the physical network and over to **Compute B**:

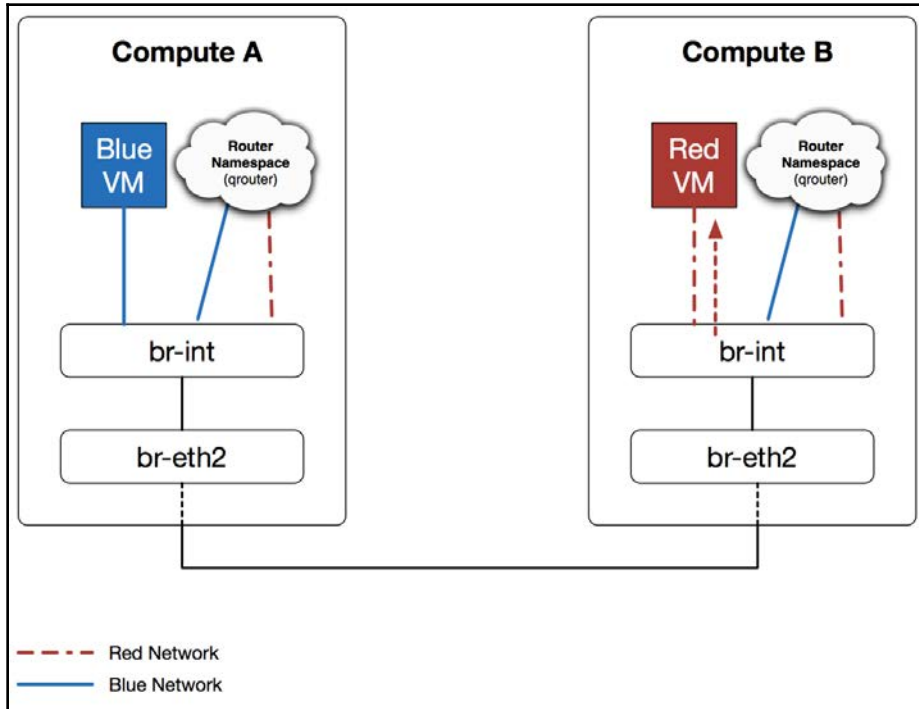


When traffic arrives at **Compute B**, it is forwarded through the provider bridge. A flow rule adds a local VLAN header that allows traffic to be matched when it is forwarded to the integration bridge:



Source MAC	Destination MAC	Source IP	Destination IP
Source host (Compute A)	Red VM	Blue VM	Red VM

In the integration bridge, a flow rule strips the local VLAN tag and changes the source MAC address back to that of the router's red interface. The packet is then forwarded to the red VM:



Source MAC	Destination MAC	Source IP	Destination IP
Red router interface	Red VM	Blue VM	Red VM

Return traffic from the red VM to the blue VM undergoes a similar routing path through the respective routers and bridges on each compute node.

Centralized SNAT

Source NAT, or SNAT for short, is the method of changing the source address of a packet as it leaves the interface of a router. When a Neutron router is allocated an IP address from an external network, that IP is used to represent traffic that originates from virtual machine instances behind the router that do not have a floating IP. All routers in Neutron, whether they are standalone, highly-available, or distributed, support SNAT and masquerade traffic originating behind the router when floating IPs are not used.



By default, routers that handle SNAT are centralized on a single node and are not highly available, resulting in a single point of failure for a given network. As a workaround, multiple nodes may be configured in `dvr_snat` mode. Neutron supports the ability to leverage VRRP to provide highly-available SNAT routers, however, the feature is experimental and is not discussed in this book.

Reviewing the topology

In this DVR SNAT demonstration, the following provider and project networks will be used:

```
root@controller01:~# openstack network list
```

ID	Name	Subnets
758070f9-ecaf-4d2a-aa49-e119ce7943f6	GATEWAY_NET	0af5a767-fce0-4c4d-9df2-1aedbd259405
db527151-2088-491b-80f3-b06b10715527	GREEN_NET	4a2b4fcc-07b3-45ae-a5e2-dfec3a8ec2e6

Using the `--distributed` argument, a distributed virtual router has been created:

```
root@controller01:~# openstack router create --distributed MyOtherDistributedRouter
```

Field	Value
admin_state_up	UP
availability_zone_hints	
availability_zones	
created_at	2018-05-07T14:32:51Z
description	
distributed	True
external_gateway_info	None
flavor_id	None
ha	False
id	d932cc76-4227-4c87-a24d-9736826e525d
name	MyOtherDistributedRouter
project_id	9233b6b4f6a54386af63c0a7b8f043c2
revision_number	None
routes	
status	ACTIVE
tags	
updated_at	2018-05-07T14:32:51Z

In this environment, the L3 agent on the host `snat01` is in `dvr_snat` mode and serves as the centralized SNAT node. Attaching the router to the project network `GREEN_NET` results in the router being scheduled to the `snat01` host:

```
root@controller01:~# openstack router set --external GATEWAY_NET MyOtherDistributedRouter
root@controller01:~#
root@controller01:~# openstack network agent list --router=MyOtherDistributedRouter
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
63214965-cd41-4aa1-a98f-a3e202929106	L3 agent	snat01	nova	:~)	UP	neutron-l3-agent

When an instance is spun up in the `GREEN_NET` network, the router is also scheduled to the respective compute node.

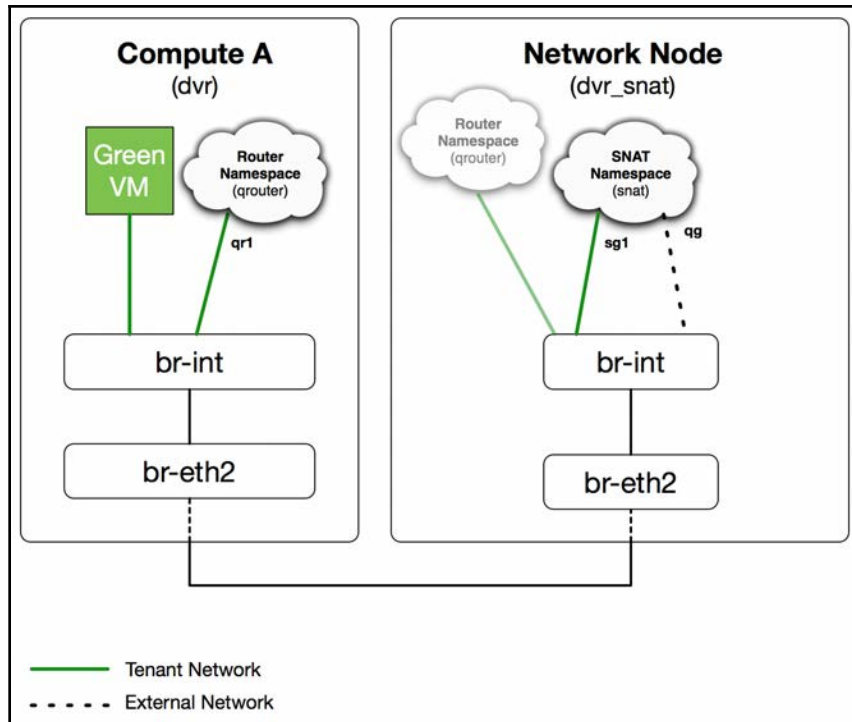
At this point, both the `snat01` and `compute03` nodes each have a `qrouter` namespace that corresponds to the `MyOtherDistributedRouter` router. Attaching the router to the external network results in the creation of a `snat` namespace on the `snat01` node. Now, on the `snat01` node, two namespaces exist for the same router – `snat` and `qrouter`:

```

root@snat01:~# ip netns
snat-d932cc76-4227-4c87-a24d-9736826e525d
qrouter-d932cc76-4227-4c87-a24d-9736826e525d
...

```

This configuration can be represented by the following diagram:



The `qrouter` namespace on the `snat01` node is configured similarly to the `qrouter` namespace on the `compute03` node. The `snat` namespace is for the centralized SNAT service.

On the `snat01` node, observe the interfaces inside the `qrouter` namespace:

```
root@snat01:~# ip netns exec qrouter-d932cc76-4227-4c87-a24d-9736826e525d ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
2: rfp-d932cc76-4@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 8a:f2:78:41:ed:5b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 169.254.106.114/31 scope global rfp-d932cc76-4
        valid_lft forever preferred_lft forever
    inet6 fe80::88f2:78ff:fe41:ed5b/64 scope link
        valid_lft forever preferred_lft forever
22: qr-13222cba-eb: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:be:8c:e9 brd ff:ff:ff:ff:ff:ff
    inet 172.24.100.1/24 brd 172.24.100.255 scope global qr-13222cba-eb
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:febe:8ce9/64 scope link
        valid_lft forever preferred_lft forever
```

Unlike the `qrouter` namespace of a legacy router, there is no `qg` interface, even though the router was attached to the external network.

However, taking a look inside the `snat` namespace, we can find the `qg` interface that is used to handle outgoing traffic from instances:

```
root@snat01:~# ip netns exec snat-d932cc76-4227-4c87-a24d-9736826e525d ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
21: qg-b752de9c-a3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:be:e2:7d brd ff:ff:ff:ff:ff:ff
    inet 10.30.0.104/24 brd 10.30.0.255 scope global qg-b752de9c-a3
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:febe:e27d/64 scope link
        valid_lft forever preferred_lft forever
23: sg-ea07f243-be: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:04:a6:08 brd ff:ff:ff:ff:ff:ff
    inet 172.24.100.11/24 brd 172.24.100.255 scope global sg-ea07f243-be
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe04:a608/64 scope link
        valid_lft forever preferred_lft forever
```

In addition to the `qg` interface, there is now a new interface with the prefix of `sg`. A virtual router will have a `qr` interface and the new `sg` interface for every internal network it is connected to. The `sg` interfaces are used as an extra hop when traffic is source NAT'd, which will be explained in further detail in the following sections.

Using the routing policy database

When a virtual machine instance without a floating IP sends traffic destined to an external network such as the internet, it hits the local `qrouter` namespace on the compute node and is routed to the `snat` namespace on the centralized `network` node. To accomplish this task, special routing rules are put in place within the `qrouter` namespaces.

Linux offers a routing policy database made up of multiple routing tables and rules that allow for intelligent routing based on destination and source addresses, IP protocols, ports, and more. There are source routing rules for every subnet a virtual router is attached to.

In this demonstration, the router is attached to a single project network:

172.24.100.0/24. Take a look at the main routing within the `qrouter` namespace on `compute01`:

```
root@compute03:~# ip netns exec qrouter-d932cc76-4227-4c87-a24d-9736826e525d ip route
169.254.106.114/31 dev rfp-d932cc76-4 proto kernel scope link src 169.254.106.114
172.24.100.0/24 dev qr-13222cba-eb proto kernel scope link src 172.24.100.1
```

Notice how there is no default route in the main routing table.

On the compute node, use the `ip rule` command from within the `qrouter` namespace to list additional routing tables and rules created by the Neutron agent:

```
root@compute03:~# ip netns exec qrouter-d932cc76-4227-4c87-a24d-9736826e525d ip rule
0: from all lookup local
32766: from all lookup main
32767: from all lookup default
2887279617: from 172.24.100.1/24 lookup 2887279617
```

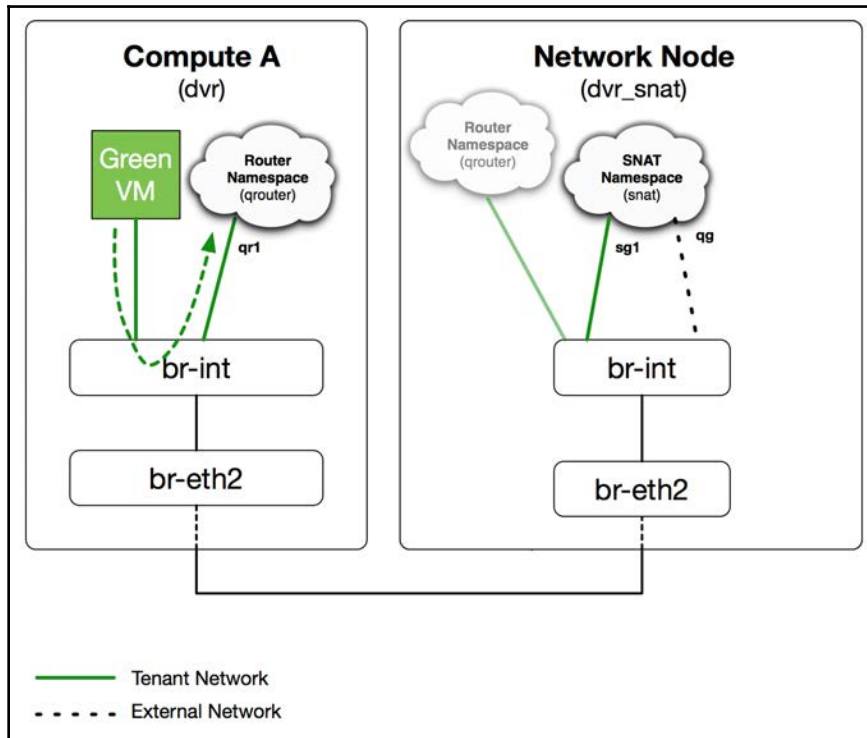
The table numbered `2887279617` was created by Neutron. The additional routing table is consulted and a default route is found:

```
root@compute03:~# ip netns exec qrouter-d932cc76-4227-4c87-a24d-9736826e525d ip route show table 2887279617
default via 172.24.100.11 dev qr-13222cba-eb
```

From that output, we can see that `172.24.100.11` is the default gateway address and corresponds to the `sq` interface within the `snat` namespace on the centralized node. When traffic reaches the `snat` namespace, the source NAT is performed and the traffic is routed out of the `qg` interface.

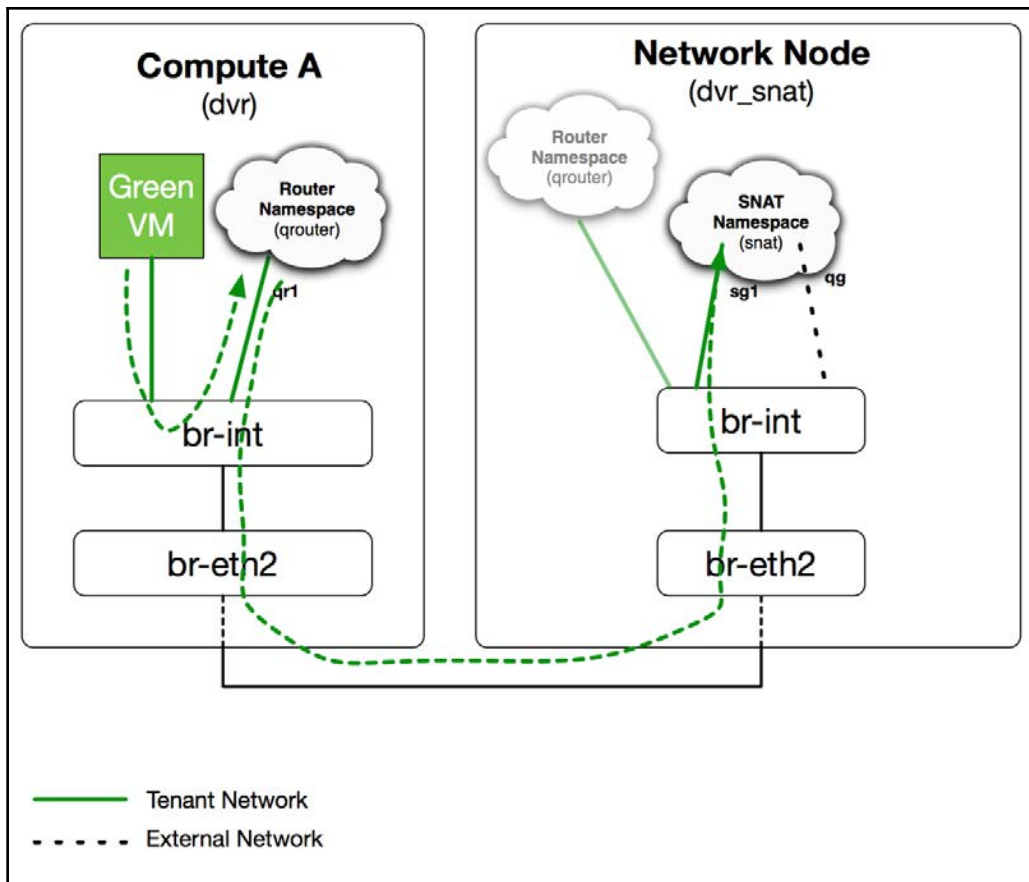
Tracing a packet through the SNAT namespace

In the following example, the green VM sends traffic to 8.8.8.8, a Google DNS server:



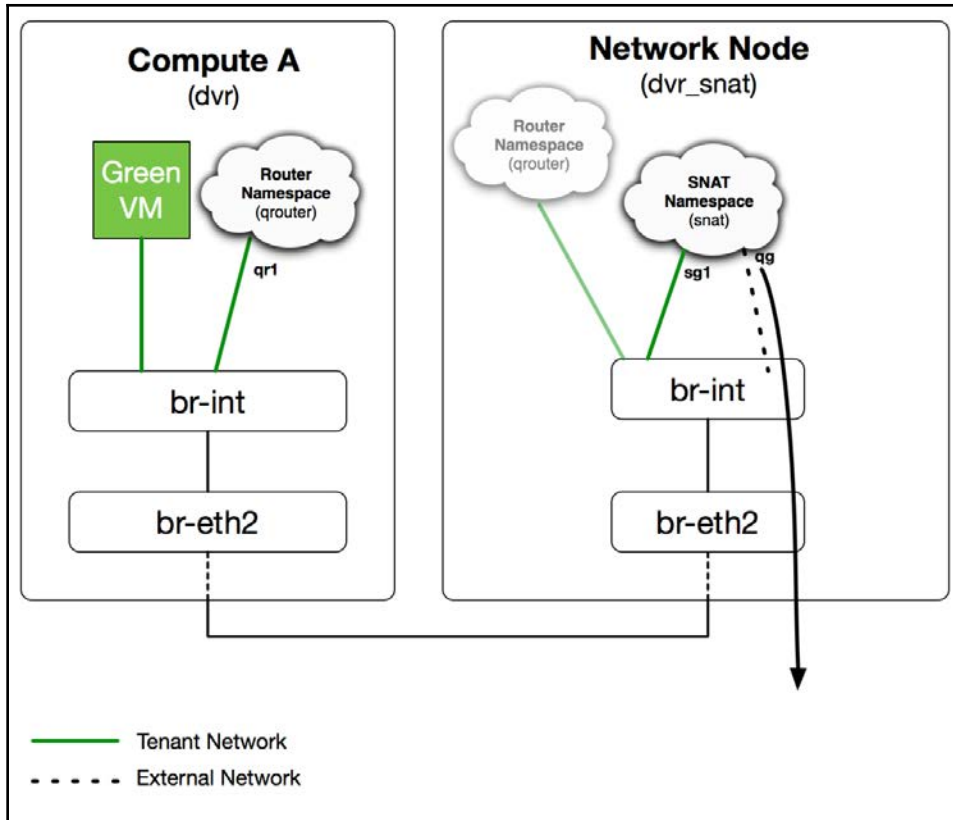
Source MAC	Destination MAC	Source IP	Destination IP
Green VM	Green Router Interface (qr1)	Green VM	8.8.8.8 (Google DNS)

When traffic arrives at the local qrouter namespace, the main routing table is consulted. The destination IP, 8.8.8.8, does not match any directly connected subnet, and a default route does not exist. Secondary routing tables are then consulted, and a match is found based on the source interface. The router then routes the traffic from the green VM to the green interface of the SNAT namespace, sg1, through the east-west routing mechanisms covered earlier in this chapter:



Source MAC	Destination MAC	Source IP	Destination IP
Green Router Interface (qr1)	Green SNAT Interface (sg1)	Green VM	8.8.8.8 (Google DNS)

When traffic enters the snat namespace, it is routed out to the `qg` interface. The `iptables` rules within the namespace change the source IP and MAC address to that of the `qg` interface to ensure traffic is routed back properly:



Source MAC	Destination MAC	Source IP	Destination IP
External SNAT Interface (<code>qg</code>)	Physical Default Gateway	External SNAT Interface (<code>qg</code>)	8.8.8.8 (Google DNS)

When the remote destination responds, a combination of flow rules on the centralized network node and compute node, along with data stored in the connection tracking and NAT tables, ensures the response is routed back to the green VM with the proper IP and MAC addresses in place.

Floating IPs through distributed virtual routers

In the network world, north-south traffic is traditionally defined as client-to-server traffic. In Neutron, as it relates to distributed virtual routers, north-south traffic is traffic that originates from an external network to virtual machine instances using floating IPs, or vice versa.

In the legacy model, all traffic to or from external clients traverses a centralized network node hosting a router with floating IPs. With DVR, the same traffic avoids the network node and is routed directly to the compute node hosting the virtual machine instance. This functionality requires compute nodes to be connected directly to external networks through an external bridge – a configuration that up until now has only been seen on nodes hosting standalone or highly-available routers.

Introducing the FIP namespace

Unlike SNAT traffic, traffic through a floating IP with DVR is handled on the individual compute nodes rather than a centralized node. When a floating IP is attached to a virtual machine instance, the L3 agent on the compute node creates a new `fip` namespace that corresponds to the external network the floating IP belongs to if one doesn't already exist:

```
root@compute03:~# ip netns
...
qrouter-d932cc76-4227-4c87-a24d-9736826e525d
fip-758070f9-ecaf-4d2a-aa49-e119ce7943f6 External Network ID
```

Any router namespace on a compute node connected to the same external network shares a single `fip` namespace and is connected to the namespace using a veth pair. The veth pairs are treated as point-to-point links between the `fip` namespace and individual `qrouter` namespaces, and are addressed as `/31` networks using a common `169.254/16` link-local address space. Because the network connections between the namespaces exist only within the nodes themselves and are used as point-to-point links, a Neutron project network allocation is not required.

In the `qrouter` namespace, one end of the veth pair has the prefix `rfp`, meaning router-to-FIP:

```
root@compute03:~# ip netns exec qrouter-d932cc76-4227-4c87-a24d-9736826e525d ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
2: rfp-d932cc76-4@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether ce:00:bb:db:0b:a1 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 169.254.106.114/31 scope global rfp-d932cc76-4
        valid_lft forever preferred_lft forever
    inet6 fe80::cc00:bbff:fedb:ba1/64 scope link
        valid_lft forever preferred_lft forever
21: qr-13222cba-eb: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:be:8c:e9 brd ff:ff:ff:ff:ff:ff
    inet 172.24.100.1/24 brd 172.24.100.255 scope global qr-13222cba-eb
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:febe:8ce9/64 scope link
        valid_lft forever preferred_lft forever
```

Inside the `fip` namespace, the other end of the veth pair has the prefix `fpr`, meaning FIP-to-router:

```
root@compute03:~# ip netns exec fip-758070f9-ecaf-4d2a-aa49-e119ce7943f6 ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
4: fpr-9822b80c-7@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether e6:62:fd:b7:96:2a brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 169.254.109.47/31 scope global fpr-9822b80c-7
        valid_lft forever preferred_lft forever
    inet6 fe80::e462:fdff:feb7:962a/64 scope link
        valid_lft forever preferred_lft forever
5: fpr-d932cc76-4@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether a6:62:ea:86:6c:1d brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet 169.254.106.115/31 scope global fpr-d932cc76-4
        valid_lft forever preferred_lft forever
    inet6 fe80::a462:eaff:fe86:6c1d/64 scope link
        valid_lft forever preferred_lft forever
14: fg-9f3f75e4-cd: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:19:fa:9f brd ff:ff:ff:ff:ff:ff
    inet 10.30.0.113/24 brd 10.30.0.255 scope global fg-9f3f75e4-cd
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe19:fa9f/64 scope link
        valid_lft forever preferred_lft forever
```

In addition to the `fpr` interface, a new interface with the prefix `fg` can be found inside the FIP namespace. The `rfp`, `fpr`, and `fg` interfaces will be discussed in the following sections.

Tracing a packet through the FIP namespace

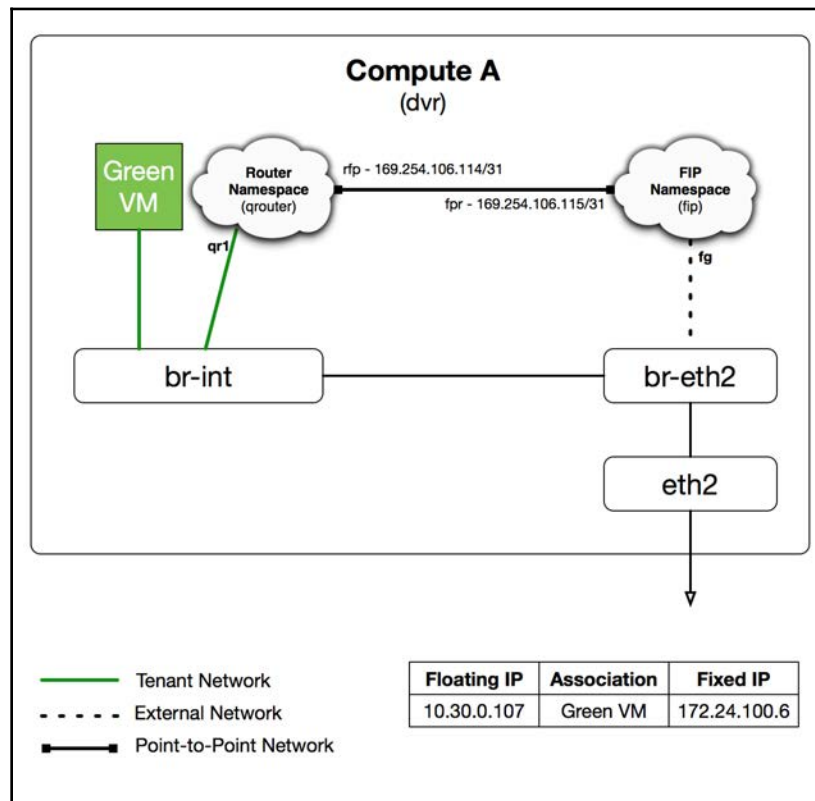
When a floating IP is assigned to an instance, a couple of things occur:

- A fip namespace for the external network is created on the compute node if one doesn't exist.
- The route table within the qrouter namespace on the compute node is modified.

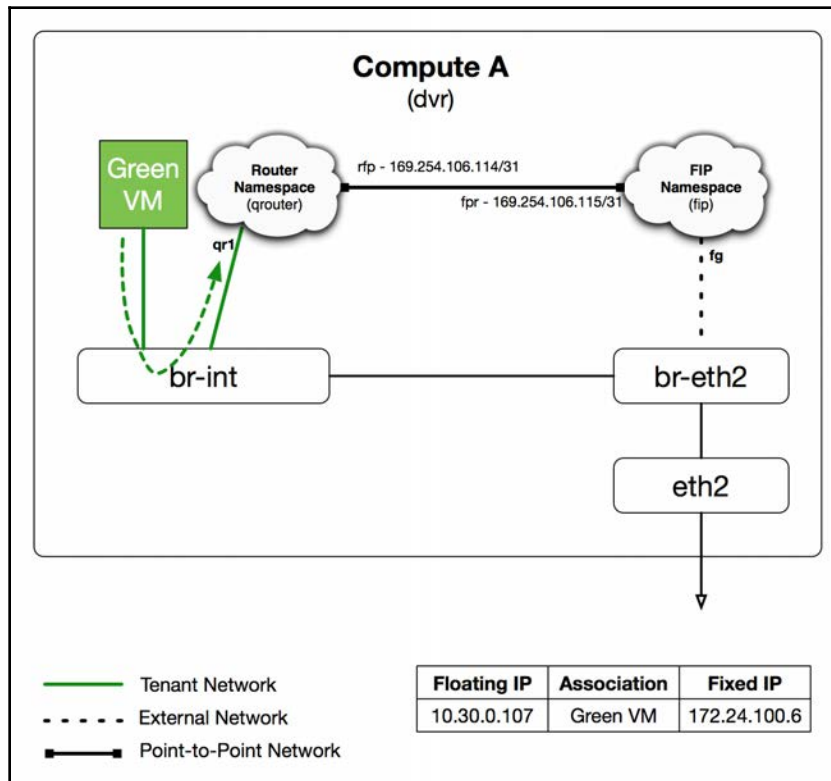
The following sections demonstrate how traffic to and from floating IPs is processed.

Sending traffic from an instance with a floating IP

Imagine a scenario where a floating IP, 10.30.0.107, has been assigned to the green VM represented in the following diagram:



When the green virtual machine instance at 172.24.100.6 sends traffic to an external resource, it first arrives at the local `qrouter` namespace:



Source MAC	Destination MAC	Source IP	Destination IP
Green VM	Green qr interface	Green VM Fixed IP (172.24.100.6)	8.8.8.8 (Google DNS)

When traffic arrives at the local `qrouter` namespace, the routing policy database is consulted so that traffic may be routed accordingly. Upon association of the floating IP to a port, a source routing rule is added to the route table within the `qrouter` namespace:

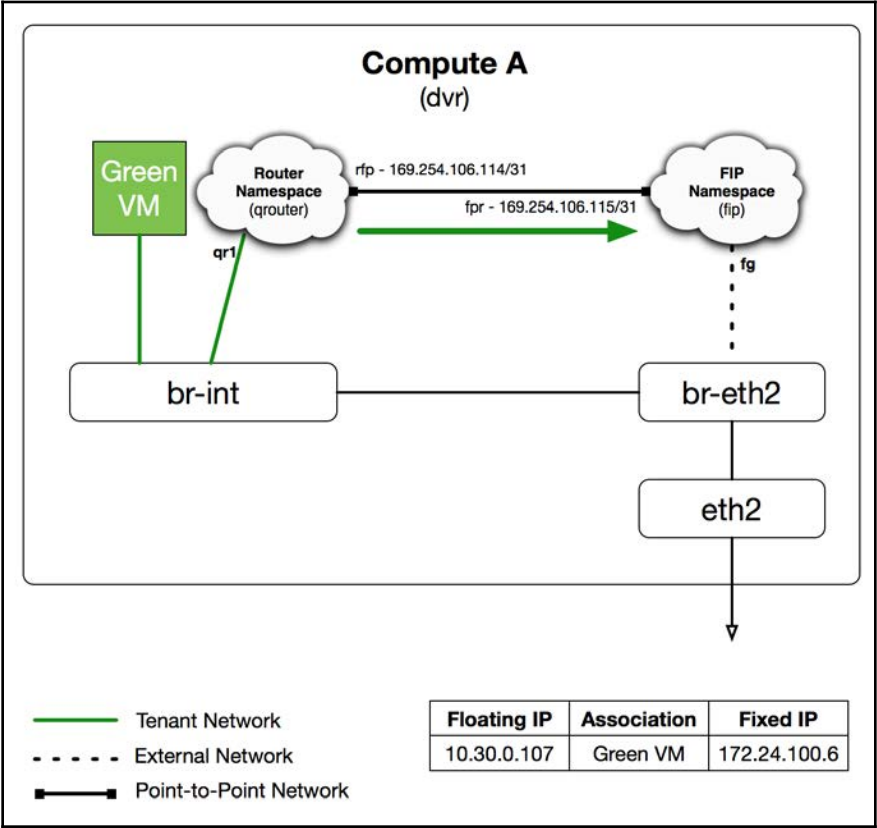
```
root@compute03:~# ip netns exec qrouter-d932cc76-4227-4c87-a24d-9736826e525d ip rule
0: from all lookup local
32766: from all lookup main
32767: from all lookup default
57481: from 172.24.100.6 lookup 16
2887279617: from 172.24.100.1/24 lookup 2887279617
```

The main routing table inside the qrouter namespace with a higher priority does not have a default route, so the 57481: from 172.24.100.6 lookup 16 rule is matched instead.

A look at the referenced routing table, table 16, shows the fip namespace's fpr interface is the default route for traffic sourced from the fixed IP of the instance:

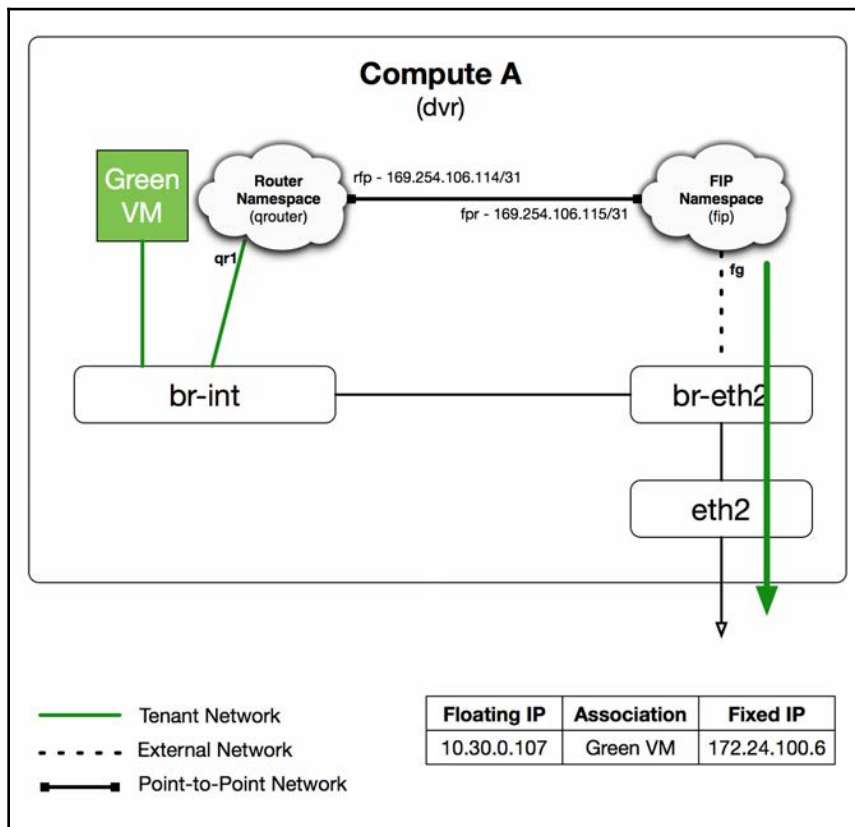
```
root@compute03:~# ip netns exec qrouter-d932cc76-4227-4c87-a24d-9736826e525d ip route show table 16
default via 169.254.106.115 dev rfp-d932cc76-4
```

The qrouter namespace performs the NAT translation of the fixed IP to the floating IP and sends the traffic to the fip namespace, as demonstrated in the following diagram:



Source MAC	Destination MAC	Source IP	Destination IP
r <code>fp</code> interface	f <code>pr</code> interface	Green VM Floating IP (10.30.0.107)	8.8.8.8 (Google DNS)

Once traffic arrives at the `fip` namespace, it is forwarded thru the `fg` interface to its default gateway:



Source MAC	Destination MAC	Source IP	Destination IP
f <code>g</code> interface	Physical Default Gateway	Green VM Floating IP (10.30.0.107)	8.8.8.8 (Google DNS)

Returning traffic to the floating IP

If you recall from earlier in this chapter, a single `fip` namespace on a compute node is shared by every `qrouter` namespace on that node connected to the external network. Much like a standalone or highly-available router has an IP address from the external network on its `qg` interface, each `fip` namespace has a single IP address from the external network configured on its `fg` interface.

On `compute03`, the IP address is `10.30.0.113`:

```
root@compute03:~# ip netns exec fip-758070f9-ecaf-4d2a-aa49-e119ce7943f6 ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
4: fpr-9822b80c-7@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether e6:62:fd:b7:96:2a brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 169.254.109.47/31 scope global fpr-9822b80c-7
        valid_lft forever preferred_lft forever
    inet6 fe80::e462:fdff:feb7:962a/64 scope link
        valid_lft forever preferred_lft forever
5: fpr-d932cc76-4@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether a6:62:ea:86:6c:1d brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet 169.254.106.115/31 scope global fpr-d932cc76-4
        valid_lft forever preferred_lft forever
    inet6 fe80::a462:eaff:fe86:6c1d/64 scope link
        valid_lft forever preferred_lft forever
14: fg-9f3f75e4-cd: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether fa:16:3e:19:fa:9f brd ff:ff:ff:ff:ff:ff
    inet 10.30.0.113/24 brd 10.30.0.255 scope global fg-9f3f75e4-cd
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe19:fa9f/64 scope link
        valid_lft forever preferred_lft forever
```

Unlike a legacy router, the `qrouter` namespaces of distributed routers do not have direct connectivity to the external network. However, the `qrouter` namespace is still responsible for performing the NAT from the fixed IP to the floating IP. Traffic is then routed to the `fip` namespace and, from there on out, to the external network.

Using proxy ARP

Floating IPs are configured on the `rfp` interface within the `qrouter` namespace, but are not directly reachable from the gateway of the external network, since the `fip` namespace sits between the `qrouter` namespace and the external network.

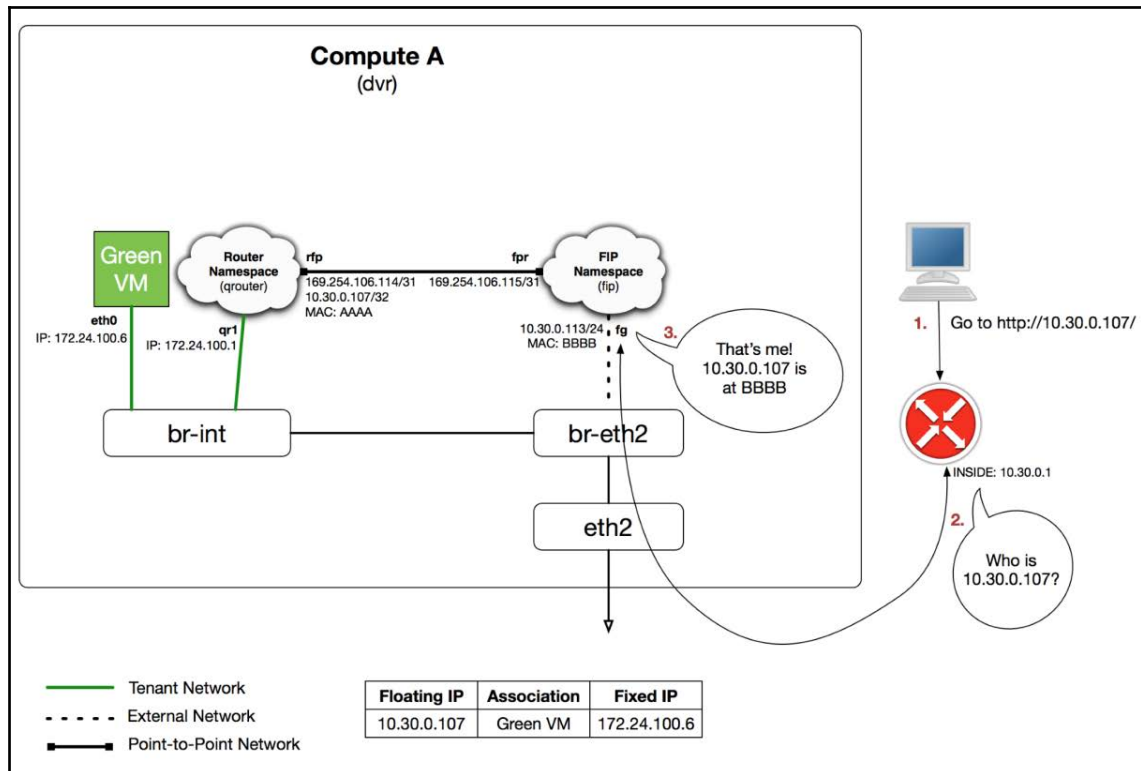
To allow for the routing of traffic through the `fip` namespace back to the `qrouter` namespace, Neutron relies on the use of proxy arp. By automatically enabling proxy arp on the `fg` interface, the `fip` namespace is able to respond to ARP requests for the floating IP, on behalf of the floating IP, from the upstream gateway device.

When traffic is routed from the gateway device to the `fip` namespace, the routing table is consulted and traffic is routed to the respective `qrouter` namespace:

```
root@compute03:~# ip netns exec fip-758070f9-ecaf-4d2a-aa49-e119ce7943f6 ip route
10.30.0.0/24 dev fg-9f3f75e4-cd proto kernel scope link src 10.30.0.113
10.30.0.107 via 169.254.106.114 dev fpr-d932cc76-4
169.254.106.114/31 dev fpr-d932cc76-4 proto kernel scope link src 169.254.106.115
169.254.109.46/31 dev fpr-9822b80c-7 proto kernel scope link src 169.254.109.47
```

← Route to fpr interface of qrouter namespace

The following diagram demonstrates how proxy arp works in this scenario:



The `fg` interface within the `fip` namespace responds on behalf of the `qrouter` namespace since `qrouter` is not directly connected to the external network. The use of a single `fip` namespace and proxy arp eliminates the need to provide each `qrouter` namespace with its own IP address from the external network, which reduces unnecessary IP address consumption and makes more floating IPs available for use by virtual machine instances and other network resources.

Summary

Distributed virtual routers have a positive impact on the network architecture as a whole by avoiding bottlenecks and single points of failure seen in the legacy model. Both east/west and north/south traffic can be routed and forwarded between compute nodes, resulting in a more efficient and resilient network. SNAT traffic is limited to a centralized node, but highly-available SNAT routers are currently available in an experimental status and will be production-ready in future releases of OpenStack.

While distributed virtual routers help provide parity with nova-network's multi-host capabilities, they are operationally complex and considerably more difficult to troubleshoot if things go wrong when compared to a standalone or highly-available router.

In the next chapter, we will look at the advanced networking service known as load balancing as-a-service, or LBaaS, and its reference architecture using the `haproxy` plugin. LBaaS allows users to create and manage load balancers that can distribute workloads across multiple virtual machine instances. Using the Neutron API, users can quickly scale their application while providing resiliency and high availability.

13

Load Balancing Traffic to Instances

The Neutron load-balancing-as-a-service plugin, known as LBaaS v2, provides users with the ability to load balance traffic to applications running on virtual instances in the cloud. Neutron provides an API to manage load balancers, listeners, pools, members, and health monitors.

In this chapter, we will cover some fundamental load balancing concepts, including the following:

- Listeners, pools, and pool members
- Load balancing algorithms
- Health Monitors
- Session Persistence

Neutron uses drivers to interact with hardware or software load balancers. In Pike, the reference driver deploys HAProxy within network namespaces. HAProxy is a free, open source load balancer that is available for most Linux-based operating systems. Octavia is another open source load balancing solution that is compatible with the LBaaS v2 API. Rather than use network namespaces, however, Octavia spins up and manages a fleet of virtual machines, containers, or bare-metal servers known as amphorae that act as load balancers. This allows Octavia to scale horizontally and on-demand. Deploying Octavia and other third-party drivers from vendors such as F5, Citrix, Radware, and more is possible, but outside the scope of this book.

In this chapter, we will work through the following:

- Integrating load balancers into the network
- Installing and configuring the LBaaS v2 agent
- Creating a load balancer
- Demonstrating load balanced traffic to instances

Fundamentals of load balancing

A load balancer is an object that represents a virtual IP and is associated with a Neutron port. A virtual IP is often exposed to the internet and mapped to a domain name to provide access to an internet-facing load balanced service. Virtual IPs can also be created for services limited to internal clients. Traffic to the virtual IP is distributed among pool members and provides scaling and resiliency to the application.

There are four major components to a load balancer in Neutron:

- Pool Member(s)
- Pool
- Health Monitor
- Listener(s)

A pool member is a Layer 4 object that represents the IP address and listening port of a service or application. For example, a pool member might be a web server with a configured IP address of 192.168.0.2 listening on TCP port 80.

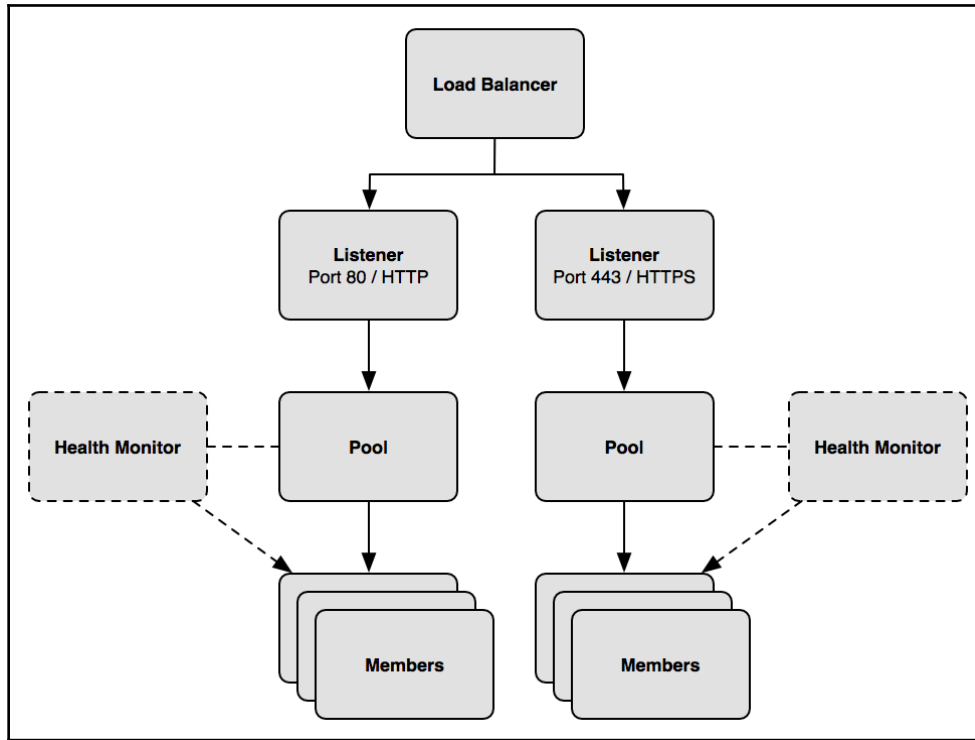
A pool is a group of pool members that typically serve identical content. A pool composed of web servers may resemble the following membership:

- Server A:192.168.0.2:80
- Server B:192.168.0.4:80
- Server C:192.168.0.6:80

A health monitor is an object that represents a health check operation against pool member(s) that run(s) at a defined interval. When the health monitor fails, the pool member is marked DOWN and is removed from the pool until it passes a subsequent check. In the meantime, traffic is directed to healthy members.

A listener is an object that represents a Layer 4 port associated with the Virtual IP that listens for incoming connections to the VIP. Load balancers can listen for connections on multiple ports, and each port is defined by a listener.

The following diagram demonstrates the relationship between these objects:



Load balancing algorithms

In version 2 of the LBaaS API, the following load balancing algorithms can be applied to a pool:

- Round robin
- Least connections
- Source IP

With the round robin algorithm, the load balancer passes each new connection to the next server in line. Over time, all connections will be distributed evenly across all machines being load balanced. Round robin is the least resource-intensive algorithm and has no mechanism to determine when a machine is being overwhelmed by connections. To avoid overwhelming a pool member, all members should be equal in terms of processing speed, connection speed, and memory.

With the least connections algorithm, the load balancer passes a new connection to a server that has the least number of current connections. It is considered a dynamic algorithm, as the system keeps track of the number of connections attached to each server and balances traffic accordingly. Pool members of higher capabilities will likely receive more traffic, as they are able to process connections quicker.

With the source IP algorithm, all connections originating from the same source IP address are sent to the same pool member. Connections are initially balanced using the round robin algorithm and are then tracked in a table for future lookup with subsequent connections from the same IP address. This algorithm is useful in cases where the application requires clients to persist to a particular server for all requests, such as an online shopping cart that stores session information on the local web server.

Monitoring

The LBaaS v2 API supports multiple monitor types, including TCP, HTTP, and HTTPS. The TCP monitor tests connectivity to pool members at Layer 4, while HTTP and HTTPS monitors tests the health of pool members based on Layer 7 HTTP status codes.

Session persistence

LBaaS v2 supports session persistence on virtual IPs. Session persistence is a method of load balancing that forces multiple client requests of the same protocol to be directed to the same node. This feature is commonly used with many web applications that do not share application states between pool members.

The types of session persistence supported with the HAProxy driver include the following:

- SOURCE_IP
- HTTP_COOKIE
- APP_COOKIE

Using the `SOURCE_IP` persistence type configures HAProxy with the following settings within the backend pool configuration:

```
stick-table type ip size 10k
stick on src
```


The first time a client connects to the virtual IP, HAProxy creates an entry in a sticky table based on the client's IP address and sends subsequent connections from the same IP address to the same backend pool member. Based on the configuration, up to 10,000 sticky entries can exist in the sticky table. This persistence method can cause load imbalance between pool members if users connect from behind a proxy server that misidentifies multiple clients as a single address.

Using the `HTTP_COOKIE` persistence type configures HAProxy with the following settings within the backend pool configuration:

```
cookie SRV insert indirect nocache
```

The first time a client connects to the virtual IP, HAProxy balances the connection to the next pool member in line. When the pool member sends its response, HAProxy injects a cookie named `SRV` into the response before sending it to the client. The value of the `SRV` cookie is a unique server identifier. When the client sends subsequent requests to the virtual IP, HAProxy strips the cookie from the request header and sends the traffic directly to the pool member identified in the cookie. This persistence method is recommended over source IP persistence, as it is not reliant on the IP address of the client. However, it may not be compatible with all clients.

Using the `APP_COOKIE` persistence type configures HAProxy with the following settings within the backend pool configuration:

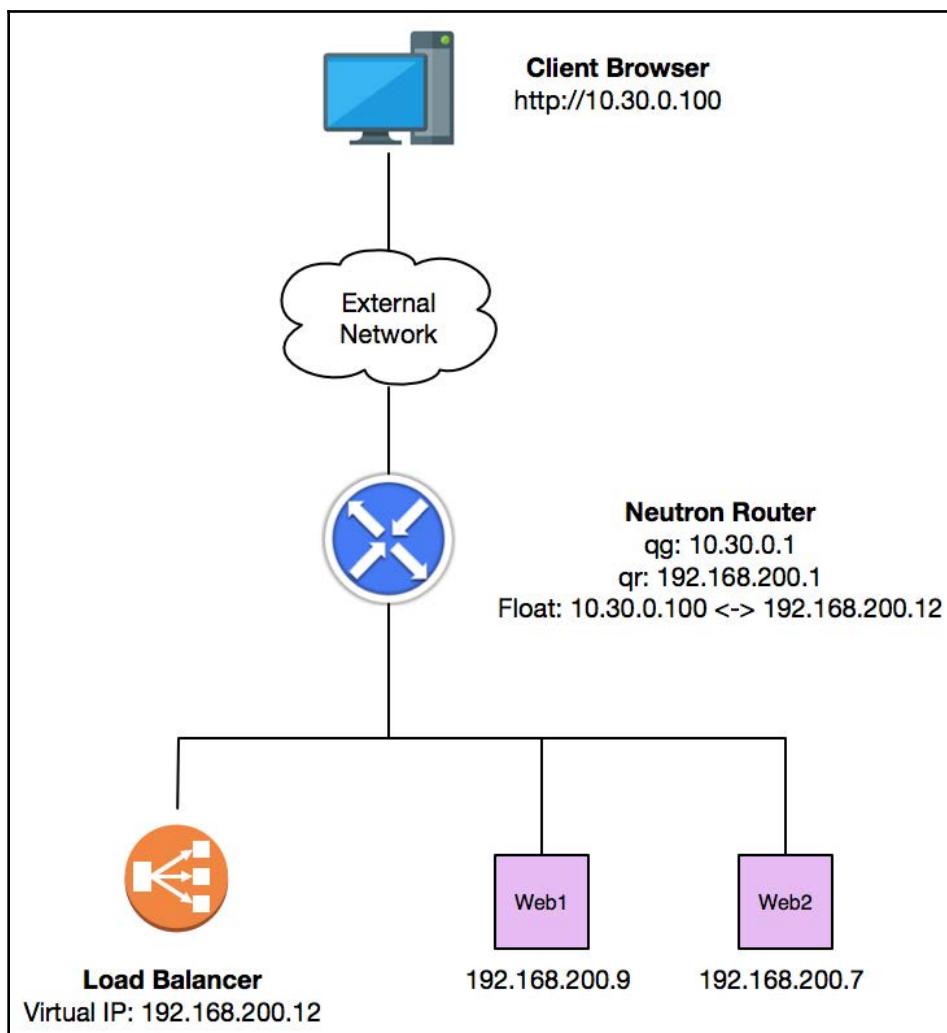
```
appsession <CookieName> len 56 timeout 3h
```

When an application cookie is defined in a backend configuration, HAProxy will check server responses for the cookie, stores its value in a table, and associates it with the server's identifier. Up to 56 characters from the value will be retained. On subsequent client connections, HAProxy will look for the cookie in the client's request. If a known value is found, the client is directed to the pool member associated with the value. Otherwise, the round-robin load balancing algorithm is applied. Cookies are automatically removed from memory when they have gone unused for a duration longer than three hours.

Integrating load balancers into the network

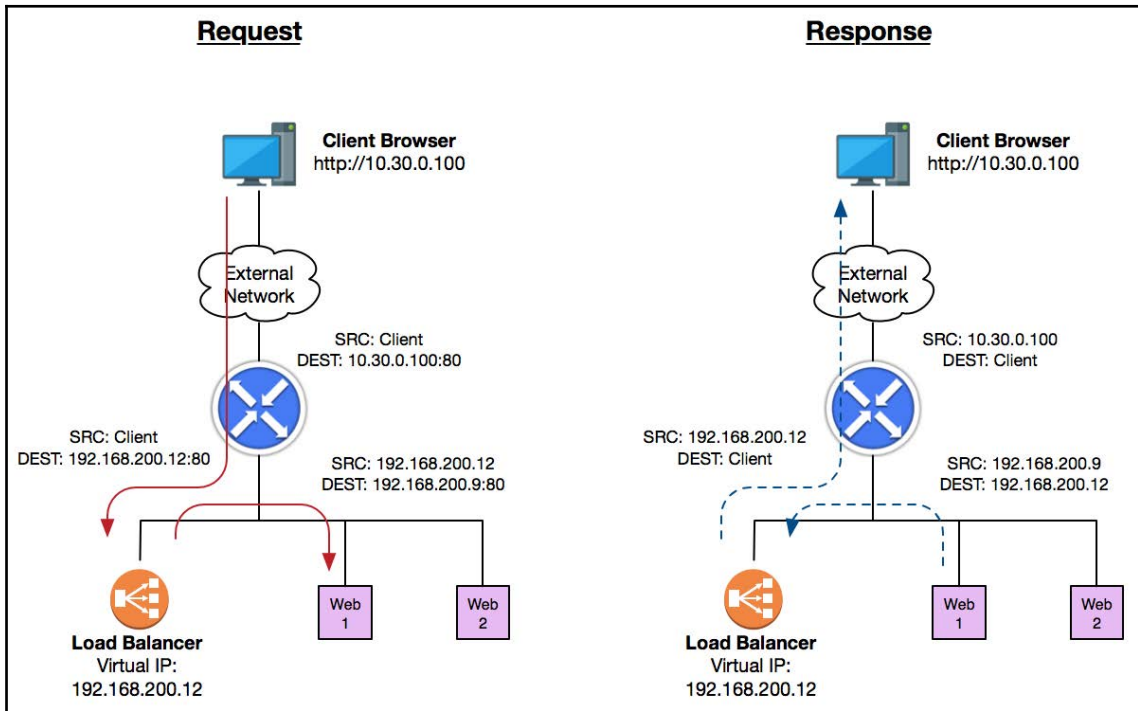
When using the HAProxy driver, load balancers are implemented in one-arm mode. In one-arm mode, the load balancer is not in the path of normal traffic to the pool members. The load balancer has a single interface for ingress and egress traffic to and from clients and pool members.

A logical diagram of a load balancer in one-arm mode can be seen here:



In the preceding diagram a load balancer is configured in one-arm mode and resides in the same subnet as the servers it is balancing traffic to.

Because a load balancer in one-arm mode is not the gateway for pool members it is sending traffic to, it must rely on the use of source NAT to ensure return traffic from the members to the client is sent back through the load balancer. An example of the traffic flow can be seen in the following diagram:



In the preceding diagram, the load balancer receives a request from the client and forwards it to **web1**. The load balancer will modify the source IP of the request to its own address, **192.168.200.12**, before forwarding the request to the pool member. This ensures that the member sends the response back to the load balancer, which will then rewrite the destination IP as the client address. If the server were to send the response directly to the client, the client would reject the packet.

Neutron configures HAProxy to send an HTTP `X-Forwarded-For` header to the pool member, which allows the member to see the original client address. Without this header, all traffic will be identified as coming from the load balancer, which may skew application reporting data and persistence efforts.

Alternatives to one-arm mode include routed mode and transparent mode. In routed mode, the load balancer acts as a gateway between the client and pool member. The source addresses of packets do not need to be manipulated in most cases, as the load balancer servers as the gateway for pool members.

In transparent mode, the load balancer acts as a network bridge between two VLANs configured with the same subnet(s). Using this mode allows users to introduce a load balancer to the network with minimal disruption, as pool members do not need to change their gateway.



There is currently no way to change the way an HAProxy-based load balancer is integrated into the network. Some third-party drivers, however, may not be limited to one-arm mode and may function in any mode.

Network namespaces

Neutron relies on network namespaces to provide individual load balancers when using the HAProxy driver. Every load balancer has a corresponding network namespace. Load balancers are scheduled to LBaaS v2 agents in the environment that are responsible for creating a corresponding network namespace and appropriate configuration. Namespaces used for load balancers are prefaced with `qlbaas` in the `ip netns list` output.

Installing LBaaS v2

The `neutron-lbaasv2-agent` service typically runs on one or more network nodes. In this environment, the service will be installed on `controller01`.

Issue the following command on the controller node to install the LBaaS v2 agent and its dependencies, including HAProxy:

```
# apt install neutron-lbaasv2-agent
```

Configuring the Neutron LBaaS agent service

Neutron stores the LBaaS v2 agent configuration in the `/etc/neutron/lbaas_agent.ini` file. The most common configuration options will be covered in the upcoming sections.

Defining an interface driver

Just like the previously installed agents, the Neutron LBaaS v2 agent must be configured to use an interface driver that corresponds to the chosen networking driver. In this configuration, there are two options:

- Linux bridge
- Open vSwitch

Update the Neutron LBaaS agent configuration file node at `/etc/neutron/lbaas_agent.ini` on the controller to use one of the following drivers.

For Linux bridge:

```
[DEFAULT]
...
interface_driver = linuxbridge
```

For Open vSwitch:

```
[DEFAULT]
...
interface_driver = openvswitch
```



In this book, we specified that the controller node runs the Linux bridge agent. Therefore, the LBaaS v2 agent should be configured to use the Linux bridge interface driver.

Defining a device driver

To manage a load balancer, the Neutron LBaaS v2 agent must be configured to use a device driver that provides the interface between the Neutron API and the programming of the load balancer itself.

Update the Neutron LBaaS v2 agent configuration file node at `/etc/neutron/lbaas_agent.ini` on the controller to use the HAProxy device driver:

```
[DEFAULT]
...
device_driver =
neutron_lbaas.drivers.haproxy.namespace_driver.HaproxyNSDriver
```

Defining a user group

Depending on the operating system in use, including Ubuntu 16.04 LTS, which is used throughout this book, the LBaaS v2 agent may need to be configured to operate HAProxy with a certain user group name. On the `controller01` node, update the LBaaS v2 configuration file with the following configuration:

```
[haproxy]
...
user_group = haproxy
```

Configuring Neutron

In addition to configuring the LBaaS agent, Neutron must be configured to use an LBaaS service plugin and driver before the API can be utilized to create LBaaS objects.

Defining a service plugin

On the `controller01` node, add the LBaaS v2 service plugin to the existing `service_plugins` list found in the Neutron configuration file at `/etc/neutron/neutron.conf`:

```
[DEFAULT]
...
service_plugins =
router,neutron_lbaas.services.loadbalancer.plugin.LoadBalancerPluginv2
```



Be sure to append to the existing list of plugins, rather than replacing the list content, to avoid issues with the API and existing network objects.

Defining a service provider

Since the Kilo release of OpenStack, many advanced service configuration options have moved out of the main Neutron configuration file into their own files. On the `controller01` node, create or update the Neutron LBaaS configuration file at `/etc/neutron/neutron_lbaas.conf` and define the HAProxy service provider driver for LBaaS:

```
[service_providers]
```

```
...
service_provider =
LOADBALANCERV2:Haproxy:neutron_lbaas.drivers.haproxy.plugin_driver.HaproxyO
nHostPluginDriver:default
```

Updating the database schema

Before restarting the Neutron API server to provide the LBaaS API, the database schema must be updated to support load balancing objects.

On the `controller01` node, run the following command to run database migrations:

```
# neutron-db-manage --subproject neutron-lbaas upgrade head
```

Restarting the Neutron LBaaS agent and API service

Once the database migration is complete, the `neutron-lbaasv2-agent` and `neutron-server` services must be restarted for the changes to take effect. Issue the following commands on the controller node to restart the services:

```
# systemctl restart neutron-server neutron-lbaasv2-agent
```

Use the `openstack network agent list` command to verify that the agent is running and has checked in:

```
root@controller01:~# openstack network agent list
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
...						
93ae43ad-e419-47ba-9671-26e417aea960	Loadbalancerv2 agent	controller01	None	:~)	UP	neutron-lbaasv2-agent
...						

If you encounter any issues, be sure to check the LBaaS v2 agent log found at `/var/log/neutron/neutron-lbaasv2-agent.log` before proceeding.

Load balancer management in the CLI

Neutron offers a number of commands that can be used to create and manage listeners, virtual IPs, pools, members, and health monitors for load balancing purposes. As of the Pike release of OpenStack, however, load balancer-related commands are not available in the openstack client. Instead, the neutron client or API should be used.

The workflow to create a basic load balancer is as follows:

- Create a load balancer object
- Create and associate a pool
- Create and associate pool member(s)
- Create and associate health monitor(s) (optional)
- Create and associate a listener

Managing load balancers in the CLI

A load balancer is an object that occupies a Neutron port and has an IP assigned from a subnet that acts as a Virtual IP. The following commands are used to manage load balancer objects in the CLI:

Load Balancer Management Commands	Description
<code>lbaas-loadbalancer-create</code>	Creates a load balancer
<code>lbaas-loadbalancer-delete</code>	Deletes a given load balancer
<code>lbaas-loadbalancer-list</code>	Lists load balancers that belong to a given tenant
<code>lbaas-loadbalancer-list-on-agent</code>	Lists the load balancers on a load balancer v2 agent
<code>lbaas-loadbalancer-show</code>	Shows information of a given load balancer
<code>lbaas-loadbalancer-stats</code>	Retrieves stats for a given load balancer
<code>lbaas-loadbalancer-status</code>	Retrieves the status for a given load balancer
<code>lbaas-loadbalancer-update</code>	Updates a given load balancer
<code>lbaas-agent-hosting-loadbalancer</code>	Gets the lbaas v2 agent hosting a load balancer

Creating load balancers in the CLI

To create a load balancer, use the `neutron lbaas-loadbalancer-create` command shown here:

```
neutron lbaas-loadbalancer-create
[--tenant-id TENANT_ID]
[--description DESCRIPTION]
[--name NAME]
[--admin-state-down]
[--provider PROVIDER]
[--flavor FLAVOR]
[--vip-address VIP_ADDRESS]
VIP_SUBNET
```

The `--tenant-id` argument is optional and can be used to associate the load balancer with a project other than the creator.

The `--description` argument is optional and can be used to provide a description for the load balancer.

The `--name` argument is optional and can be used to specify a name for the load balancer.

The `--admin-state-down` argument is optional and can be used to set the load balancer in a DOWN state at creation.

The `--provider` argument is optional and can be used to specify the load balancer provider driver in lieu of the default. The default provider in this environment is HAProxy.

The `--vip-address` argument is optional and can be used to specify a particular IP address for the VIP.

The `VIP_SUBNET` argument is used to specify the subnet from which the load balancer VIP will be procured.

Deleting load balancers in the CLI

To delete a load balancer, use the `neutron lbaas-loadbalancer-delete` command shown here:

```
neutron lbaas-loadbalancer-delete LOADBALANCER [LOADBALANCER...]
```

The `LOADBALANCER` argument specifies the name or ID of the load balancer to delete. Multiple load balancers can be deleted simultaneously using a space-separated list.

Listing load balancers in the CLI

To list all load balancers, use the `neutron lbaas-loadbalancer-list` command shown here:

```
neutron lbaas-loadbalancer-list
```

Showing load balancer details in the CLI

To show the details of a load balancer, use the `neutron lbaas-loadbalancer-show` command shown here:

```
neutron lbaas-loadbalancer-show LOADBALANCER
```

The `LOADBALANCER` argument specifies the name or ID of the load balancer to show.

Showing load balancer statistics in the CLI

To show the statistics of a load balancer, use the `neutron lbaas-loadbalancer-stats` command shown here:

```
neutron lbaas-loadbalancer-stats LOADBALANCER
```

The `LOADBALANCER` argument specifies the name or ID of the load balancer to show.

Showing the load balancer's status in the CLI

To show the status of a load balancer, use the `neutron lbaas-loadbalancer-status` command shown here:

```
neutron lbaas-loadbalancer-status LOADBALANCER
```

The `LOADBALANCER` argument specifies the name or ID of the load balancer to show.

Updating a load balancer in the CLI

To update the attributes of a load balancer, use the `neutron lbaas-loadbalancer-update` command shown here:

```
neutron lbaas-loadbalancer-update
[--description DESCRIPTION]
[--name NAME]
LOADBALANCER
```

The `--description` argument is optional and can be used to update the description of the load balancer.

The `--name` argument is optional and can be used to update the name of a load balancer.

The `LOADBALANCER` argument specifies the name or ID of the load balancer to update.

Managing pools in the CLI

A pool is a set of devices, such as web servers, that are grouped together to receive and process traffic. When traffic is sent to a virtual IP, the load balancer sends the request to any of the servers that are members of that pool.

The following commands are used to manage load balancer pools in the CLI:

Pool Commands	Description
<code>lbaas-pool-create</code>	Creates a pool
<code>lbaas-pool-delete</code>	Deletes a given pool
<code>lbaas-pool-list</code>	Lists pools that belong to a given tenant
<code>lbaas-pool-show</code>	Shows the information of a given pool
<code>lbaas-pool-update</code>	Updates a given pool

Creating a pool in the CLI

To create a pool, use the `neutron lbaas-pool-create` command shown here:

```
neutron lbaas-pool-create
[--tenant-id TENANT_ID]
[--description DESCRIPTION]
[--name NAME] --lb-algorithm
{ROUND_ROBIN, LEAST_CONNECTIONS, SOURCE_IP}
```

```
[--admin-state-down]
[--listener LISTENER]
[--loadbalancer LOADBALANCER] --protocol {HTTP,HTTPS,TCP}
[--session-persistence type=TYPE[,cookie_name=COOKIE_NAME]]
```

The `--tenant-id` argument is optional and can be used to associate the pool with a project other than the creator. The tenant or project ID specified should match the respective load balancer.

The `--description` argument is optional and can be used to provide a description for the pool.

The `--name` argument is optional and can be used to set a name for the pool.

The `--lb-algorithm` argument is required and is used to specify the load balancing algorithm used to distribute traffic among the pool members. Possible options include `ROUND_ROBIN`, `LEAST_CONNECTIONS`, and `SOURCE_IP`.

The `--admin-state-down` argument is optional and can be used to set the pool in a `DOWN` state upon creation.

The `--listener` argument is optional and is used to associate the pool with a listener.

The `--loadbalancer` argument is optional and can be used to specify the associated load balancer.

The `--protocol` argument is required and is used to specify the protocol for balancing. Possible options include `HTTP`, `HTTPS`, and `TCP`.

The `--session-persistence` argument is optional and is used to specify the session persistence method and/or cookie type.

Deleting a pool in the CLI

To delete a pool, use the `neutron lbaas-pool-delete` command shown here:

```
neutron lbaas-pool-delete POOL [POOL ...]
```

The `POOL` argument specifies the name or ID of the pool to delete. Multiple pools can be deleted simultaneously using a space-separated list.

Listing pools in the CLI

To list all pools, use the `neutron lbaas-pool-list` command shown here:

```
neutron lbaas-pool-list
```

Showing pool details in the CLI

To show the details of a pool, use the `neutron lbaas-pool-show` command shown here:

```
neutron lbaas-pool-show POOL
```

The `POOL` argument specifies the name or ID of the pool to show.

Updating a pool in the CLI

To update a pool, use the `neutron lbaas-pool-update` command shown here:

```
neutron lbaas-pool-update
[--admin-state-up {True,False}]
[--session-persistence type=TYPE [,cookie_name=COOKIE_NAME]
| --no-session-persistence]
[--description DESCRIPTION]
[--name NAME]
[--lb-algorithm {ROUND_ROBIN,LEAST_CONNECTIONS,SOURCE_IP}]
POOL
```

The `--admin-state-up` argument is optional and can be used to change the state of a pool.

The `--session-persistence` argument is optional and is used to modify the session persistence method and/or cookie type.

The `--no-session-persistence` argument is optional and is used to remove session persistence from the pool.

The `--description` argument is optional and can be used to update the description of the pool.

The `--name` argument is optional and can be used to update the name of the pool.

The `--lb-algorithm` argument is optional and is used to modify the load balancing algorithm used to distribute traffic among the pool members.

The `POOL` argument specifies the name or ID or the pool to update.

Managing pool members in the CLI

The following commands are used to manage pool members in the CLI:

Pool Member Command	Description
<code>lbaas-member-create</code>	Creates a member
<code>lbaas-member-delete</code>	Deletes a given member
<code>lbaas-member-list</code>	Lists members that belong to a given pool
<code>lbaas-member-show</code>	Shows information of a given member
<code>lbaas-member-update</code>	Updates a given member

Creating pool members in the CLI

To create a pool member, use the `neutron lbaas-member-create` command as follows:

```
neutron lbaas-member-create
[--tenant-id TENANT_ID]
[--name NAME]
[--weight WEIGHT]
[--admin-state-down]
--subnet SUBNET
--address ADDRESS
--protocol-port
PROTOCOL_PORT
POOL
```

The `--tenant-id` argument is optional and can be used to associate the pool member with a project other than the creator. The tenant or project ID specified should match the respective load balancer.

The `--name` argument is optional and can be used to set the name of the pool member.

The `--weight` argument allows you to associate a weight with the pool member. When set, a pool member may receive more or less traffic than other members in the same pool. For example, a pool member with a weight of 2 will receive twice the amount of traffic as a pool member with a weight of 1. A pool member with a weight of 3 will receive three times the traffic as a pool member with a weight of 1, and so on.

The `--admin-state-down` argument is optional and can be used to set the pool member **DOWN** on creation.

The `--subnet` argument is required and is used to specify the subnet of the pool member.

The `--address` argument is required and is used to specify the IP address of the pool member. The IP address must fall within the specified subnet CIDR.

The `--protocol-port` argument is required and is used to specify the listening port of the application being balanced. For example, if you are balancing HTTP traffic, the listening port specified would be 80. For SSL traffic, the port specified would be 443. In most cases, the VIP associated with the pool will utilize the same application port number.

The `POOL` argument is required and associates the pool member with the specified pool.

Deleting pool members

To delete a pool member, use the `neutron lbaas-member-delete` command shown here:

```
neutron lbaas-member-delete MEMBER [MEMBER ...] POOL
```

The `MEMBER` argument specifies the name or ID of the pool member to delete from the specified pool. Multiple pool members can be deleted simultaneously using a space-separated list.

Listing pool members

To obtain a list of pool members for a particular pool, use the `neutron lbaas-member-list` command shown here:

```
neutron lbaas-member-list POOL
```

Showing pool member details

To show the details of a pool member, use the `neutron lbaas-member-show` command shown here:

```
neutron lbaas-member-show MEMBER POOL
```

The `MEMBER` argument represents the ID or name of the member of the specified pool.

Updating a pool member

To update the attributes of a pool member, use the `neutron lbaas-member-update` command as follows:

```
neutron lbaas-member-update
[--admin-state-up {True,False}]
[--name NAME] [--weight WEIGHT]
MEMBER POOL
```

The `--admin-state-up` argument is optional and can be used to change the state of a pool member.

The `--name` argument is optional and can be used to update the name of the pool member.

The `--weight` argument is optional and can be used to update the weight of the pool member.

The `MEMBER` argument is required and represents the ID or name of the member of the specified pool.

Managing health monitors in the CLI

LBaaS in Neutron provides the ability to monitor the health of pool members as a method of ensuring the availability of an application. If a pool member is not in a healthy state, Neutron can pull a member out of rotation, limiting the impact of issues between the client and the application.

The following commands are used to manage health monitors in the CLI:

Health Monitor Commands	Description
<code>lbaas-healthmonitor-create</code>	Creates a health monitor
<code>lbaas-healthmonitor-delete</code>	Deletes a given health monitor
<code>lbaas-healthmonitor-list</code>	Lists health monitors that belong to a given tenant
<code>lbaas-healthmonitor-show</code>	Shows the information of a given health monitor
<code>lbaas-healthmonitor-update</code>	Updates a given health monitor

Creating a health monitor in the CLI

To create a health monitor, use the `neutron lbaas-healthmonitor-create` command shown here:

```
neutron lbaas-healthmonitor-create
[--tenant-id TENANT_ID] --delay DELAY
[--name NAME] --timeout TIMEOUT
[--http-method HTTP_METHOD]
[--url-path URL_PATH] --max-retries MAX_RETRIES
[--expected-codes EXPECTED_CODES]
[--admin-state-down]
--type {PING,TCP,HTTP,HTTPS} --pool
POOL
```

The `--tenant-id` argument is optional and can be used to associate the health monitor with a project other than the creator. The tenant or project ID specified should match the respective load balancer.

The `--delay` argument is required and is used to specify the period between each health check sent to members (in seconds). A common starting value is 5 seconds.

The `--name` argument is optional and can be used to set the name of the health monitor.

The `--timeout` argument is required and is used to specify the number of seconds for a monitor to wait for a connection to be established. The value must be less than the delay value.

The `--http-method` argument is optional and is used in conjunction with `--expected-codes` and `--url-path`. It is used to specify the type of HTTP request being made. Common types include GET and POST. The default value is GET.

The `--url-path` argument is optional and is used in conjunction with `--expected-codes` and `--http-method`. When specified, the system will perform an HTTP request defined by `--http-method` for the URL against the pool member. The default value is root or /.

The `--max-retries` argument is required and is used to specify the maximum number of consecutive failures before a pool member is marked as DOWN. A common starting value is 3 retries.

The `--expected-codes` argument is optional and allows you to specify the HTTP status code(s) that indicate that a pool member is working as expected when the monitor sends an HTTP request to the pool member for the specified URL. For example, if a `GET` request for a URL is sent to a pool member, the server is expected to return a 200 OK status upon successful retrieval of the page. If 200 is listed as an expected code, the monitor would mark the pool member as UP. As a result, the pool member would be eligible to receive connections. If a 500 status code were returned, it could indicate that the server is not properly processing connections. The health monitor would mark the pool member as DOWN and temporarily remove it from the pool. The default value is 200.

The `--type` argument is required and is used to specify the type of monitor being configured. The four types include the following:

- **PING:** The simplest of all monitor types, PING uses ICMP to confirm connectivity to pool members.



The PING type is not supported by the HAProxy driver and results in the same behaviour as the TCP monitor type.

- **TCP:** This instructs the load balancer to send a TCP SYN packet to the pool member. Upon receiving a SYN ACK back, the load balancer resets the connection. This type of monitor is commonly referred to as a half-open TCP monitor.
- **HTTP:** This instructs the monitor to initiate an HTTP request to a pool member based on the `expected_codes`, `url_path`, and `http_method` attributes described here.
- **HTTPS:** This instructs the monitor to initiate an HTTPS request to a pool member based on the `expected_codes`, `url_path`, and `http_method` attributes described here.

The `--pool` argument is required and is used to associate the health monitor with the given pool. Only one health monitor per pool is allowed.

Deleting a health monitor in the CLI

To delete a health monitor, use the `neutron lbaas-healthmonitor-delete` command shown here:

```
neutron lbaas-healthmonitor-delete HEALTHMONITOR
```

The `HEALTHMONITOR` argument specifies the name or ID of the health monitor to delete. Multiple health monitors can be deleted simultaneously using a space-separated list.

Listing health monitors in the CLI

To obtain a list of health monitors, use the `neutron lbaas-healthmonitor-list` command shown here:

```
neutron lbaas-healthmonitor-list
```

Showing health monitor details

To show the details of a health monitor, use the `neutron lbaas-healthmonitor-show` command shown here:

```
neutron lbaas-healthmonitor-show HEALTHMONITOR
```

The details returned include delay, expected codes, HTTP method, max retries, pools, timeout, type, and URL path.

Updating a health monitor

To update the attributes of a health monitor, use the `neutron lbaas-healthmonitor-update` command as follows:

```
neutron lbaas-healthmonitor-update  
[--delay DELAY] [--name NAME]  
[--timeout TIMEOUT]  
[--http-method HTTP_METHOD]  
[--url-path URL_PATH]  
[--max-retries MAX_RETRIES]  
[--expected-codes EXPECTED_CODES]  
[--admin-state-up {True,False}]  
HEALTHMONITOR
```

Updateable attributes include delay, expected codes, HTTP method, max retries, timeout, and URL path.

Managing listeners in the CLI

The following commands are used to manage listeners in the CLI:

Listener Commands	Description
<code>lbaas-listener-create</code>	Creates a listener
<code>lbaas-listener-delete</code>	Deletes a given listener
<code>lbaas-listener-list</code>	Lists listeners that belong to a given tenant
<code>lbaas-listener-show</code>	Shows the information of a given listener
<code>lbaas-listener-update</code>	Updates a given listener

Creating listeners in the CLI

To create a listener, use the `neutron lbaas-listener-create` command shown here:

```
neutron lbaas-listener-create
[--tenant-id TENANT_ID]
[--description DESCRIPTION]
[--connection-limit CONNECTION_LIMIT]
[--default-pool DEFAULT_POOL]
[--admin-state-down]
[--name NAME]
[--default-tls-container-ref DEFAULT_TLS_CONTAINER_REF]
[--sni-container-refs SNI_CONTAINER_REFS [SNI_CONTAINER_REFS ...]]
[--loadbalancer LOADBALANCER] --protocol
{TCP,HTTP,HTTPS,TERMINATED_HTTPS}
--protocol-port PORT
```

The `--tenant-id` argument is optional and can be used to associate the listener with a project other than the creator. The tenant or project ID specified should match the respective load balancer.

The `--description` argument is optional and can be used to provide a description for the listener.

The `--connection-limit` argument is optional and can be used to limit connections per second to the listener. The default is unlimited (-1).

The `--default-pool` argument is optional and sets the default pool for the listener.

The `--admin-state-down` argument is optional and can be used to set the listener in a DOWN state at creation.

The `--name` argument is optional and can be used to set a name for the listener.

The `--loadbalancer` argument is optional and can be used to specify the associated load balancer.

The `--protocol` argument is required and is used to specify the protocol for the listener. Options include `TCP`, `HTTP`, `HTTPS`, and `TERMINATED_HTTPS`.

The `--protocol-port` argument is required and is used to specify the port for the listener.

Deleting listeners in the CLI

To delete a load balancer, use the `neutron lbaas-loadbalancer-delete` command shown here:

```
neutron lbaas-listener-delete LISTENER [LISTENER ...]
```

The `LISTENER` argument specifies the name or ID or the listener to delete. Multiple listeners can be deleted simultaneously by using a space-separated list.

Listing listeners in the CLI

To list all listeners, use the `neutron lbaas-listener-list` command shown here:

```
neutron lbaas-listener-list
```

Showing listener details in the CLI

To show the details of a listener, use the `neutron lbaas-listener-show` command shown here:

```
neutron lbaas-listener-show LISTENER
```

The `LISTENER` argument specifies the name or ID or the listener to show.

Updating a listener in the CLI

To update the attributes of a listener, use the `neutron lbaas-listener-update` command shown here:

```
neutron lbaas-listener-update
[--description DESCRIPTION]
[--connection-limit CONNECTION_LIMIT]
[--default-pool DEFAULT_POOL]
[--name NAME]
[--admin-state-up {True,False}]
LISTENER
```

The `--description` argument is optional and can be used to update the description of the listener.

The `--connection-limit` argument is optional and can be used to update the connection limit.

The `--default-pool` argument is optional and can be used to update the default pool of the listener.

The `--name` argument is option and can be used to set a name for the listener.

The `--admin-state-up` argument is optional and can be used to set the listener's state.

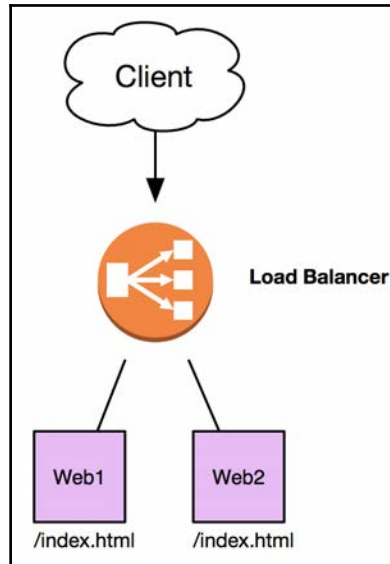
The `LISTENER` argument specifies the name or ID or the listener to update.

Building a load balancer

To demonstrate the creation and use of load balancers in Neutron, this section is dedicated to building a functional load balancer based on the following scenario:

"A project has a simple Neutron network architecture composed of a router attached to both an external provider network and internal tenant network. The user would like to load balance HTTP traffic between two instances, each running a web server on port 80. Each instance has been configured with an `index.html` page containing a unique server identifier."

A diagram of the requested topology can be seen here:



This demonstration assumes that two instances named **web1** and **web2** have been deployed in the environment and are connected to a project network. The network is connected to a Neutron router that provides outbound access and inbound access via floating IPs. Concepts explained in earlier chapters can be used to build this topology.

To eliminate the installation and configuration of a web server, you can mimic the behavior of one by using the `SimpleHTTPServer` Python module on the instances, as shown here:

```
ubuntu@web1:~$ echo "This is web1" > ~/index.html
ubuntu@web1:~$ sudo python -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
```

Repeat the mentioned commands for the second instance, substituting `web2` for `web1` in the `index.html` file.

Creating a load balancer

The first step to building a functional load balancer is to first create the load balancer object. With the neutron client, create a load balancer object with the following attributes:

- Name: `LB_WEB`
- Subnet ID: <Subnet ID of the pool members>

The following command can be used:

```
neutron lbaas-loadbalancer-create PROJECT_SUBNET --name LB_WEB
```

The output will resemble the following:

```
root@controller01:~# neutron lbaas-loadbalancer-create PROJECT_SUBNET --name LB_WEB
Created a new loadbalancer:
```

Field	Value
admin_state_up	True
description	
id	137292b2-79cf-447a-8259-39ac9aef6575
listeners	
name	LB_WEB
operating_status	OFFLINE
pools	
provider	haproxy
provisioning_status	PENDING_CREATE
tenant_id	9233b6b4f6a54386af63c0a7b8f043c2
vip_address	192.168.200.12
vip_port_id	28d1e4f9-c81a-4c82-90b5-20bf1eb5aa3d
vip_subnet_id	f29a2257-1283-4047-a835-b207480aa6f3

When a load balancer is created, OpenStack assigns an IP address known as a virtual IP, or VIP. The VIP will be used by clients to access the load-balanced application. Creating a load balancer object is only the first step in load balancing traffic to instances.

Creating a pool

Pools are associated with load balancers and are objects that represent a collection of instances that receive traffic sent to the VIP. With the neutron client, create a pool with the following attributes:

- Name: WEB_POOL
- Load balancing method: ROUND_ROBIN
- Protocol: HTTP
- Subnet ID: <Subnet ID where pool members reside>

The following command can be used:

```
neutron lbaas-pool-create \
--lb-algorithm ROUND_ROBIN \
--protocol HTTP \
--loadbalancer LB_WEB \
--name POOL_WEB
```

The output will resemble the following:

```
root@controller01:~# neutron lbaas-pool-create \
> --lb-algorithm ROUND_ROBIN \
> --protocol HTTP \
> --loadbalancer LB_WEB \
> --name POOL_WEB
neutron CLI is deprecated and will be removed in the future. Use openstack CLI instead.
Created a new pool:
```

Field	Value
admin_state_up	True
description	
healthmonitor_id	
id	05762f48-64e8-4e5a-a972-ddc2516c0153
lb_algorithm	ROUND_ROBIN
listeners	
loadbalancers	{"id": "137292b2-79cf-447a-8259-39ac9aef6575"}
members	
name	POOL_WEB
protocol	HTTP
session_persistence	
tenant_id	9233b6b4f6a54386af63c0a7b8f043c2

Creating pool members

The next step to building a functional load balancer is to create and associate pool members with the pool. Pool members are objects that represent a backend application listening on a particular IP and port.

In this environment, there are two instances eligible for use in the pool:

```
root@controller01:~# openstack server list
```

ID	Name	Status	Networks	Image	Flavor
d0ce5716-f1dc-4d5f-a0c9-0f0db7721556	web2	ACTIVE	PROJECT_NET=192.168.200.7	ubuntu-xenial-16.04	small
6157572c-2b6d-4f66-a66e-b0beaae0e0bf	web1	ACTIVE	PROJECT_NET=192.168.200.9	ubuntu-xenial-16.04	small

With the neutron client, create pool members with the following attributes:

Name	Address	Port	Pool
web1	192.168.200.9	80	POOL_WEB
web2	192.168.200.7	80	POOL_WEB

The following screenshot demonstrates the process of creating the first pool member:

```
root@controller01:~# neutron lbaas-member-create --name web1 \
> --subnet PROJECT_SUBNET \
> --address 192.168.200.9 \
> --protocol-port 80 \
> POOL_WEB
Created a new member:
```

Field	Value
address	192.168.200.9
admin_state_up	True
id	9554ae8b-8ced-4074-ab5f-037f0ef440cc
name	web1
protocol_port	80
subnet_id	f29a2257-1283-4047-a835-b207480aa6f3
tenant_id	9233b6b4f6a54386af63c0a7b8f043c2
weight	1

Repeat the process shown in the preceding screenshot to create the second pool member, as shown here:

```
root@controller01:~# neutron lbaas-member-create --name web2 \
> --subnet PROJECT_SUBNET \
> --address 192.168.200.7 \
> --protocol-port 80 \
> POOL_WEB
Created a new member:
```

Field	Value
address	192.168.200.7
admin_state_up	True
id	0ec91151-6ef1-4289-ad53-db9e46dc9d60
name	web2
protocol_port	80
subnet_id	f29a2257-1283-4047-a835-b207480aa6f3
tenant_id	9233b6b4f6a54386af63c0a7b8f043c2
weight	1

The `neutron lbaas-member-list` command can be used to return the pool members for the `POOL_WEB` pool:

```
root@controller01:~# neutron lbaas-member-list POOL_WEB
```

id	name	address	protocol_port	weight	admin_state_up
0ec91151-6ef1-4289-ad53-db9e46dc9d60	web2	192.168.200.7	80	1	True
9554ae8b-8ced-4074-ab5f-037f0ef440cc	web1	192.168.200.9	80	1	True

Creating a health monitor

To provide high availability of an application to clients, it is recommended to create and apply a health monitor to a pool. Without a monitor, the load balancer will continue to send traffic to members that may not be available.

Using the `neutron lbaas-healthmonitor-create` command, create a health monitor with the following attributes:

- Name: MONITOR_WEB
- Delay: 5
- Max retries:3
- Timeout: 4
- Type: TCP
- Pool: POOL_WEB

```
root@controller01:~# neutron lbaas-healthmonitor-create \
> --name MONITOR_WEB \
> --delay 5 \
> --max-retries 3 \
> --timeout 4 \
> --type HTTP \
> --pool POOL_WEB
Created a new healthmonitor:
```

Field	Value
admin_state_up	True
delay	5
expected_codes	200
http_method	GET
id	98cc9c87-d001-4d0a-b317-3e77d21c0689
max_retries	3
max_retries_down	3
name	MONITOR_WEB
pools	{"id": "05762f48-64e8-4e5a-a972-ddc2516c0153"}
tenant_id	9233b6b4f6a54386af63c0a7b8f043c2
timeout	4
type	HTTP
url_path	/

Creating a listener

The last step in creating a function load balancer is to create the listener. Using the `neutron lbaas-listener-create` command, create a listener with the following attributes:

- Name: `LISTENER_WEB`
- Port: 80
- Protocol: HTTP
- Pool: `POOL_WEB`

```
root@controller01:~# neutron lbaas-listener-create \
> --name LISTENER_WEB \
> --loadbalancer LB_WEB \
> --protocol HTTP \
> --protocol-port 80 \
> --default-pool POOL_WEB
Created a new listener:
```

Field	Value
admin_state_up	True
connection_limit	-1
default_pool_id	05762f48-64e8-4e5a-a972-ddc2516c0153
default_tls_container_ref	
description	
id	d416b3cb-612f-4a68-a77e-e14cf8bda245
loadbalancers	{"id": "137292b2-79cf-447a-8259-39ac9aef6575"}
name	LISTENER_WEB
protocol	HTTP
protocol_port	80
sni_container_refs	
tenant_id	9233b6b4f6a54386af63c0a7b8f043c2

Multiple listeners can be created for each load balancer, allowing users to balance traffic for multiple protocols and applications on the same Virtual IP.

The LBaaS network namespace

A listing of the network namespaces on the host running the LBaaS v2 agent reveals a network namespace that corresponds to the load balancer we just created:

```
root@controller01:~# ip netns
qrouter-f94e2ec9-1c18-49ab-82bd-2fbcd5608cc4 (id: 5)
qlbaas-137292b2-79cf-447a-8259-39ac9aef6575 (id: 3)
qdhcp-db527151-2088-491b-80f3-b06b10715527 (id: 2)
qdhcp-d8428f56-ba13-4aa7-97bb-76a6feb308bf (id: 1)
qdhcp-3f05db4d-c4ec-4cee-a616-2cfffbbd296e (id: 0)
qdhcp-ce1fe7cd-7b01-4d58-b1c8-414886b7d8f2 (id: 4)
```

The IP configuration within the namespace reveals an interface that corresponds to the subnet of the virtual IP:

```
root@controller01:~# ip netns exec qlbaas-137292b2-79cf-447a-8259-39ac9aef6575 ip addr list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
2: ns-28d1e4f9-c8@if42: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:9a:46:69 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.200.12/24 brd 192.168.200.255 scope global ns-28d1e4f9-c8
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe9a:4669/64 scope link
        valid_lft forever preferred_lft forever
```

Neutron creates an HAProxy configuration file specific to every load balancer that is created by users. The load balancer configuration files can be found in the `/var/lib/neutron/lbaas/v2/` directory on the host(s) running the LBaaS v2 agent.

The configuration file for this load balancer built by Neutron can be seen in the following screenshot:

```

root@controller01:~# cat /var/lib/neutron/lbaas/v2/137292b2-79cf-447a-8259-39ac9aef6575/haproxy.conf
# Configuration for LB_WEB
global
    daemon
    user nobody
    group haproxy
    log /dev/log local0
    log /dev/log local1 notice
    maxconn 2000
    stats socket /var/lib/neutron/lbaas/v2/137292b2-79cf-447a-8259-39ac9aef6575/haproxy_stats.sock mode 0666 level user

defaults
    log global
    retries 3
    option redispatch
    timeout connect 5000
    timeout client 50000
    timeout server 50000

frontend d416b3cb-612f-4a68-a77e-e14cf8bda245
    option tcplog
    option forwardfor
    bind 192.168.200.12:80
    mode http
    default_backend 05762f48-64e8-4e5a-a972-ddc2516c0153

backend 05762f48-64e8-4e5a-a972-ddc2516c0153
    mode http
    balance roundrobin
    timeout check 4s
    option httpchk GET /
    http-check expect rstatus 200
    server 0ec91151-6ef1-4289-ad53-db9e46dc9d60 192.168.200.7:80 weight 1 check inter 5s fall 3
    server 9554ae8b-8ced-4074-ab5f-037f0ef440cc 192.168.200.9:80 weight 1 check inter 5s fall 3

```

Confirming load balancer functionality

From within the `qlbaas` namespace, confirm direct connectivity to `web1` and `web2` by using `curl`:

```

root@controller01:~# ip netns exec qlbaas-137292b2-79cf-447a-8259-39ac9aef6575 curl http://192.168.200.9
This is web1

root@controller01:~# ip netns exec qlbaas-137292b2-79cf-447a-8259-39ac9aef6575 curl http://192.168.200.7
This is web2

```

By connecting to the VIP address rather than the individual pool members, you can observe the default round-robin load balancing algorithm in effect:

```

root@controller01:~# ip netns exec qlbaas-137292b2-79cf-447a-8259-39ac9aef6575 curl http://192.168.200.12
This is web1
root@controller01:~# ip netns exec qlbaas-137292b2-79cf-447a-8259-39ac9aef6575 curl http://192.168.200.12
This is web2
root@controller01:~# ip netns exec qlbaas-137292b2-79cf-447a-8259-39ac9aef6575 curl http://192.168.200.12
This is web1
root@controller01:~# ip netns exec qlbaas-137292b2-79cf-447a-8259-39ac9aef6575 curl http://192.168.200.12
This is web2

```


With round-robin load balancing, every connection is evenly distributed among the two pool members.

Observing health monitors

A packet capture on `web1` reveals that the load balancer is performing TCP health checks every 5 seconds:

```
ubuntu@web1:~$ sudo tcpdump -i any port 80
sudo: unable to resolve host web1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 262144 bytes
20:33:27.562511 IP 192.168.200.12.51200 > 192.168.200.9.http: Flags [S], seq 4
20:33:27.562568 IP 192.168.200.9.http > 192.168.200.12.51200: Flags [S.], seq 3
20:33:27.563384 IP 192.168.200.12.51200 > 192.168.200.9.http: Flags [.], ack 1
20:33:27.563425 IP 192.168.200.12.51200 > 192.168.200.9.http: Flags [P.], seq 3
20:33:27.563440 IP 192.168.200.9.http > 192.168.200.12.51200: Flags [.], ack 3
20:33:27.564620 IP 192.168.200.9.http > 192.168.200.12.51200: Flags [P.], seq 3
20:33:27.565113 IP 192.168.200.12.51200 > 192.168.200.9.http: Flags [.], ack 1

20:33:32.567675 IP 192.168.200.12.51204 > 192.168.200.9.http: Flags [S], seq 4
20:33:32.567733 IP 192.168.200.9.http > 192.168.200.12.51204: Flags [S.], seq 3
20:33:32.569304 IP 192.168.200.12.51204 > 192.168.200.9.http: Flags [.], ack 1
20:33:32.570146 IP 192.168.200.12.51204 > 192.168.200.9.http: Flags [P.], seq 3
20:33:32.570271 IP 192.168.200.9.http > 192.168.200.12.51204: Flags [.], ack 3
20:33:32.570970 IP 192.168.200.9.http > 192.168.200.12.51204: Flags [P.], seq 3
20:33:32.571436 IP 192.168.200.9.http > 192.168.200.12.51204: Flags [FP.], seq 3
20:33:32.571892 IP 192.168.200.12.51204 > 192.168.200.9.http: Flags [.], ack 1
20:33:32.571908 IP 192.168.200.12.51204 > 192.168.200.9.http: Flags [R.], seq 3
20:33:32.571919 IP 192.168.200.12.51204 > 192.168.200.9.http: Flags [R], seq 3

20:33:37.573026 IP 192.168.200.12.51208 > 192.168.200.9.http: Flags [S], seq 2
20:33:37.573085 IP 192.168.200.9.http > 192.168.200.12.51208: Flags [S.], seq 2
20:33:37.574143 IP 192.168.200.12.51208 > 192.168.200.9.http: Flags [.], ack 1
20:33:37.574740 IP 192.168.200.12.51208 > 192.168.200.9.http: Flags [P.], seq 2
20:33:37.574762 IP 192.168.200.9.http > 192.168.200.12.51208: Flags [.], ack 3
20:33:37.576198 IP 192.168.200.9.http > 192.168.200.12.51208: Flags [P.], seq 2
20:33:37.576834 IP 192.168.200.12.51208 > 192.168.200.9.http: Flags [.], ack 1
20:33:37.576860 IP 192.168.200.9.http > 192.168.200.12.51208: Flags [P.], seq 2
20:33:37.576940 IP 192.168.200.12.51208 > 192.168.200.9.http: Flags [R.], seq 2
20:33:37.577335 IP 192.168.200.12.51208 > 192.168.200.9.http: Flags [R], seq 2
```

In the preceding output, the load balancer is initiating a connection every 5 seconds and performing the health check prescribed by the health monitor: "a GET request: against/".

To observe the monitor removing a pool member from eligibility, stop the web service on `web1` and observe the packet captures on `web1` and logs on the `controller` node:


```
ubuntu@web1:~$ sudo tcpdump -i any port 80
sudo: unable to resolve host web1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 262144 bytes
20:05:28 920882 IP 192.168.200.12.38842 > 192.168.200.9.http: Flags [S], seq 1190505067, win 29200, d
20:05:28 920932 IP 192.168.200.9.http > 192.168.200.12.38842: Flags [R.], seq 0, ack 1190505068, win
20:05:33 924053 IP 192.168.200.12.38846 > 192.168.200.9.http: Flags [S], seq 1440376660, win 29200, d
20:05:33 924105 IP 192.168.200.9.http > 192.168.200.12.38846: Flags [R.], seq 0, ack 1440376661, win
20:05:38 925611 IP 192.168.200.12.38850 > 192.168.200.9.http: Flags [S], seq 1530860699, win 29200, d
20:05:38 925662 IP 192.168.200.9.http > 192.168.200.12.38850: Flags [R.], seq 0, ack 1530860700, win
```

In the preceding output, the web service is stopped and connections to port 80 are refused. Immediately following the third failure, the load balancer marks the pool member as `DOWN`:

```
root@controller01:~# tail -f /var/log/haproxy.log
May 18 20:05:18 controller01 haproxy[27950]: Server
05762f48-64e8-4e5a-a972-ddc2516c0153/9554ae8b-8ced-4074-ab5f-037f0ef440cc is DOWN, reason: Layer4 connection
problem, info: "Connection refused", check duration: 1ms. 1 active and 0 backup servers left. 0 sessions active, 0
requested, 0 remaining in queue.
```

While `web1` is down, all subsequent connections to the VIP are sent to `web2`:

```
root@controller01:~# ip netns exec qlbaas-137292b2-79cf-447a-8259-39ac9aef6575 curl http://192.168.200.12
This is web2
root@controller01:~# ip netns exec qlbaas-137292b2-79cf-447a-8259-39ac9aef6575 curl http://192.168.200.12
This is web2
root@controller01:~# ip netns exec qlbaas-137292b2-79cf-447a-8259-39ac9aef6575 curl http://192.168.200.12
This is web2
root@controller01:~# ip netns exec qlbaas-137292b2-79cf-447a-8259-39ac9aef6575 curl http://192.168.200.12
This is web2
```

After restarting the web service on `web1`, the load balancer places the server back in the pool upon the next successful health check:

```
May 18 20:16:49 controller01 haproxy[27950]: Server 05762f48-64e8-4e5a-a972-ddc2516c0153/9554ae8b-8ced-4074-ab5f-037f0ef440cc is UP
reason: Layer7 check passed, code: 200, info: "HTTP status check returned code <3C>200<3E>", check duration: 2ms. 2 active and 0 backup
servers online. 0 sessions requested, 0 total in queue.
```

Connecting to the virtual IP externally

To connect to a virtual IP externally, a floating IP must be associated with the VIP since the virtual IP exists within a subnet behind the router and is not reachable directly.

Using the `openstack floating ip create` command, assign a floating IP to be used with the virtual IP:

```
root@controller01:~# openstack floating ip create --port 28d1e4f9-c81a-4c82-90b5-20bf1eb5aa3d GATEWAY_NET
```

Field	Value
created_at	2018-05-18T20:25:03Z
description	
fixed_ip_address	192.168.200.12
floating_ip_address	10.30.0.100
floating_network_id	758070f9-ecaf-4d2a-aa49-e119ce7943f6
id	b3b729d5-e36a-4c4a-bcc7-ab05c69ea8d7
name	10.30.0.100
port_id	28d1e4f9-c81a-4c82-90b5-20bf1eb5aa3d
project_id	9233b6b4f6a54386af63c0a7b8f043c2
revision_number	0
router_id	9822b80c-7548-43f7-8a03-70c5baf6c9c0
status	DOWN
updated_at	2018-05-18T20:25:03Z

A test from a workstation to the floating IP confirms external connectivity to the load balancer and its pool members:

```
workstation:~ jdenton$ curl http://10.30.0.100
This is web2
workstation:~ jdenton$ curl http://10.30.0.100
This is web1
workstation:~ jdenton$ curl http://10.30.0.100
This is web2
workstation:~ jdenton$ curl http://10.30.0.100
This is web1
```



If the connection is not successful, be sure to confirm security groups allow connections to port 80.

Load balancer management in the dashboard

Out of the box, the LBaaS panels for load balancer management in Horizon are not installed. To install the panels, perform the following:

On the controller node, install the `python-neutron-lbaas-dashboard` package:

```
# apt install python-neutron-lbaas-dashboard
```

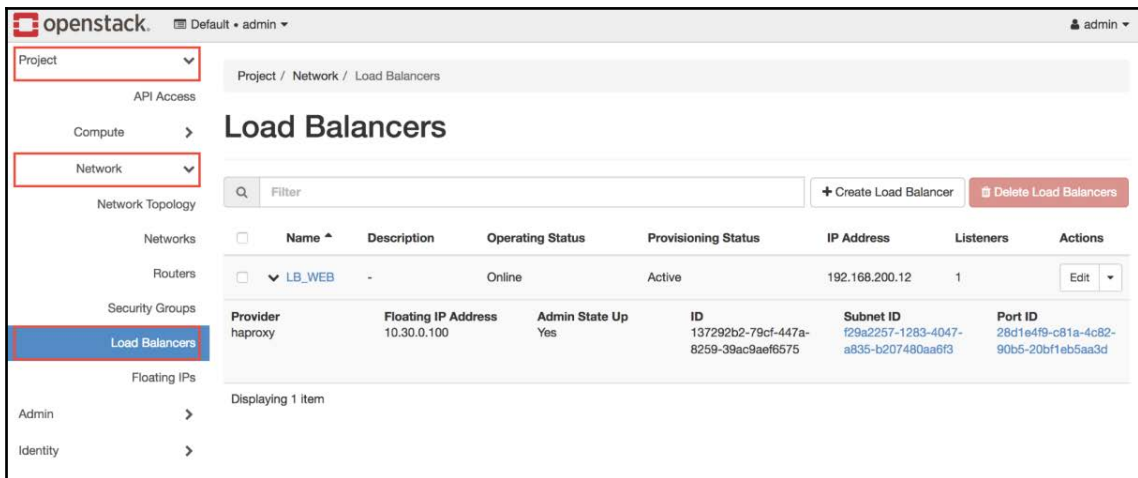
Then, update the OpenStack Dashboard configuration file at `/etc/openstack-dashboard/local_settings.py` to enable the panels. Search for `enable_lb` in the `OPENSTACK_NEUTRON_NETWORK` dictionary and change the value from `False` to `True`:

```
OPENSTACK_NEUTRON_NETWORK = {
    'enable_router': True,
    'enable_quotas': False,
    'enable_ipv6': False,
    'enable_distributed_router': False,
    'enable_ha_router': False,
    'enable_lb': True,
    'enable_firewall': False,
    'enable_vpn': False,
    'enable_fip_topology_check': False,
}
```

Lastly, restart the Apache service:

```
# systemctl restart apache2
```

Once the panels have been installed and activated, load balancers can be managed from the **Project** panel by clicking on **Network | Load Balancers** in the menu on the left-hand side of the screen:



The screenshot shows the OpenStack Dashboard interface. The left sidebar contains a menu with 'Project' and 'Network' highlighted by red boxes. The main content area displays the 'Load Balancers' panel, which includes a table with the following data:

Name	Description	Operating Status	Provisioning Status	IP Address	Listeners	Actions
LB_WEB	-	Online	Active	192.168.200.12	1	Edit

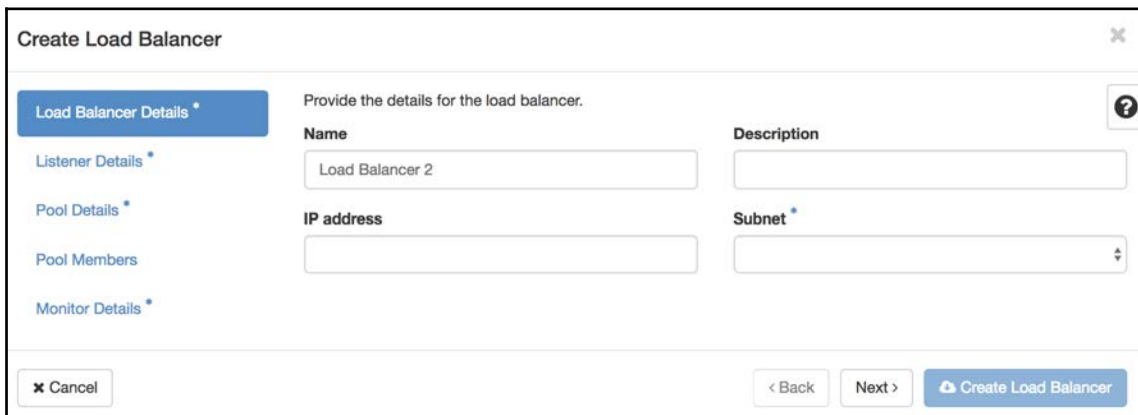
Below the table, there is a section for 'Provider haproxy' with details for Floating IP Address, Admin State Up, ID, Subnet ID, and Port ID.

In Horizon, load balancers are created and managed as a single entity that includes pool, members, and monitors. This is unlike LBaaS v1, where members and monitors may have been managed separately in their own panel.

Creating a load balancer in the dashboard

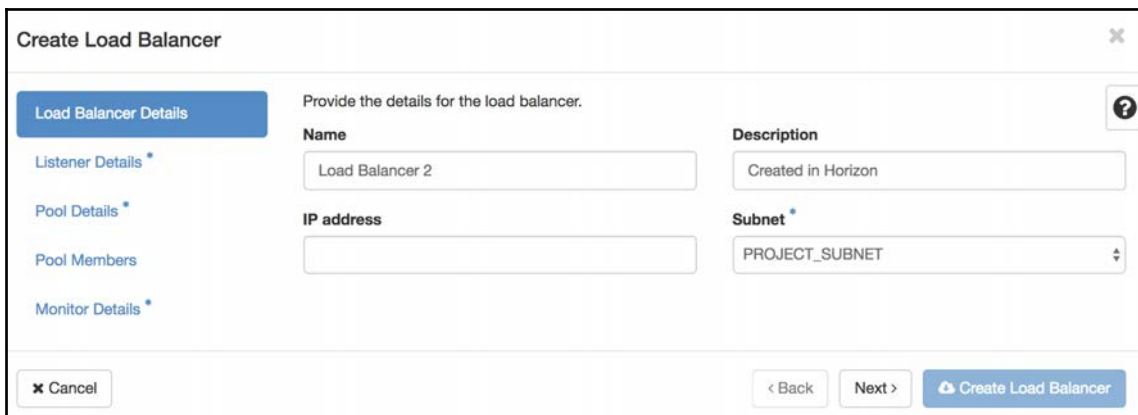
To create a load balancer, perform the following steps:

1. Click on the **Create Load Balancer** button within the **Load Balancers** panel. A wizard will appear that resembles the one shown in the following screenshot:



The screenshot shows the 'Create Load Balancer' wizard with the 'Load Balancer Details' panel selected. The wizard is titled 'Create Load Balancer' and has a close button (X) in the top right corner. On the left, there is a sidebar with five tabs: 'Load Balancer Details' (selected), 'Listener Details', 'Pool Details', 'Pool Members', and 'Monitor Details'. The main area is titled 'Provide the details for the load balancer.' and contains four fields: 'Name' (text input with 'Load Balancer 2'), 'Description' (text input), 'IP address' (text input), and 'Subnet' (dropdown menu). At the bottom, there are three buttons: 'Cancel', '< Back', and 'Next >', followed by a blue 'Create Load Balancer' button.

2. From the **Load Balancer Details** panel, you can enter a name for the load balancer, a description, an IP address, and select the respective subnet for the Virtual IP:



This screenshot shows the same 'Create Load Balancer' wizard as the previous one, but with the fields filled out. The 'Name' field contains 'Load Balancer 2', the 'Description' field contains 'Created in Horizon', the 'IP address' field is empty, and the 'Subnet' dropdown menu is set to 'PROJECT_SUBNET'. The 'Create Load Balancer' button is still present at the bottom right.

3. From the **Listener** panel, you can enter a name for the listener, a description, and choose the protocol and port:

The screenshot shows the 'Create Load Balancer' dialog with the 'Listener Details' panel selected. The panel title is 'Provide the details for the listener.' It contains four input fields: 'Name' (Listener 1), 'Description' (Created in Horizon), 'Protocol' (HTTP), and 'Port' (80). The 'Protocol' and 'Port' fields have dropdown arrows. At the bottom, there are three buttons: 'Cancel', '< Back', and 'Next >', followed by a blue 'Create Load Balancer' button with a cloud icon.

Create Load Balancer

Load Balancer Details

Listener Details

Pool Details *

Pool Members

Monitor Details *

Provide the details for the listener.

Name

Listener 1

Description

Created in Horizon

Protocol *

HTTP

Port *

80

Cancel

< Back

Next >

Create Load Balancer

4. From the **Pool Details** panel, you can enter a name for the pool, a description, and select the load balancing method:

The screenshot shows the 'Create Load Balancer' dialog with the 'Pool Details' panel selected. The panel title is 'Provide the details for the pool.' It contains three input fields: 'Name' (Pool 1), 'Description' (Created in Horizon), and 'Method' (ROUND_ROBIN). The 'Method' field has a dropdown arrow. At the bottom, there are three buttons: 'Cancel', '< Back', and 'Next >', followed by a blue 'Create Load Balancer' button with a cloud icon.

Create Load Balancer

Load Balancer Details

Listener Details

Pool Details

Pool Members

Monitor Details *

Provide the details for the pool.

Name

Pool 1

Description

Created in Horizon

Method *

ROUND_ROBIN

Cancel

< Back

Next >

Create Load Balancer

- From the **Pool Members** panel, you can select the pool members that will make up the pool:

▼ Available Instances		
<input type="text" value="Filter"/>		
Name	IP Address	
web2	192.168.200.7	<button>Add</button>
web1	192.168.200.9	<button>Add</button>
GREEN_VM	172.24.100.6	<button>Add</button>
RED_VM	172.16.0.4	<button>Add</button>
BLUE_VM	192.168.150.13	<button>Add</button>

- Click the **Add** button next to the pool member to add it to the pool:

Create Load Balancer

Load Balancer Details

Listener Details

Pool Details

Pool Members

Monitor Details *

Add members to the load balancer pool.

▼ Allocated Members 2

IP Address *	Subnet *	Port *	Weight	
192.168.200.9	PROJECT_SUBNET	80	1	<button>Remove</button>
192.168.200.7	PROJECT_SUBNET	80	1	<button>Remove</button>

Add external member

▼ Available Instances

Name	IP Address	
web2	192.168.200.7	<button>Add</button>
web1	192.168.200.9	<button>Add</button>
GREEN_VM	172.24.100.6	<button>Add</button>
RED_VM	172.16.0.4	<button>Add</button>
BLUE_VM	192.168.150.13	<button>Add</button>

✕ Cancel

< BackNext >Create Load Balancer

7. From the **Monitor Details** panel, you can set the monitor type, interval, retries, and timeout values:

The screenshot shows the 'Create Load Balancer' wizard with the 'Monitor Details' panel selected. The panel contains the following fields:

- Monitor type**: A dropdown menu with 'TCP' selected.
- Interval (sec)**: A text input field with the value '5'.
- Retries**: A text input field with the value '3'.
- Timeout (sec)**: A text input field with the value '5'.

At the bottom of the wizard, there are three buttons: 'Cancel', '< Back', and 'Next >', followed by a large blue 'Create Load Balancer' button.

8. Once the load balancer has been configured, click the blue **Create Load Balancer** button to complete the wizard. The load balancer will appear in the list of load balancers:

The screenshot shows the 'Load Balancers' page in the AWS Management Console. The page has a breadcrumb trail: 'Project / Network / Load Balancers'. Below the title, there is a search bar and two buttons: '+ Create Load Balancer' and 'Delete Load Balancers'. The main content is a table with the following columns: Name, Description, Operating Status, Provisioning Status, IP Address, Listeners, and Actions.

Name ^	Description	Operating Status	Provisioning Status	IP Address	Listeners	Actions
> LB_WEB	-	Online	Active	192.168.200.12	1	Edit
▼ Load Balancer 2	Created in Horizon	Online	Active	192.168.200.3	1	Edit

Below the table, there is a section for detailed information:

Provider	Floating IP Address	Admin State Up	ID	Subnet ID	Port ID
haproxy	None	Yes	f1792f8b-1c9c-438e-a432-91c927212e44	f29a2257-1283-4047-a835-b207480aa6f3	c3210bdc-688b-4df2-a4bf-c803e9445c41

At the bottom, it says 'Displaying 2 items'.

Assigning a floating IP to the load balancer

To assign a floating IP to a load balancer within the Horizon dashboard:

1. Click the arrow under the **Actions** menu next to the load balancer and select **Associate Floating IP**:

Project / Network / Load Balancers

Load Balancers

Q

Filter

+ Create Load Balancer

Delete Load Balancers

☐	Name ^	Description	Operating Status	Provisioning Status	IP Address	Listeners	Actions
☐	> LB_WEB	-	Online	Active	192.168.200.12	1	Edit ▾
☐	▼ Load Balancer 2	Created in Horizon	Online	Active	192.168.200.3	1	Edit ▾
Provider haproxy	Floating IP Address None	Admin State Up Yes	ID f1792f8b-1c9c-438e-a432-91c927212e44	Subnet ID f29a2257-1283-4041-a835-b207480aa6f3	<div>Associate Floating IP</div> <div>Delete Load Balancer</div>		

Displaying 2 items

2. A wizard will pop up that allows you to select an existing floating IP or a network from which a new one can be created:

Associate Floating IP Address

Select a floating IP address to associate with the load balancer or a floating IP pool in which to allocate a new floating IP address.

Floating IP address or pool *

✓

Floating IP addresses

10.30.0.105

10.30.0.101

Floating IP pools

GATEWAY_NET

✕ Cancel

✓ Associate

3. Once a floating IP or network has been chosen, click on the blue **Associate** button to associate the floating IP with the virtual IP of the load balancer.

The load balancer details will be updated to reflect the new floating IP.

Summary

Load-balancing-as-a-Service provides users with the ability to scale their application programmatically through the Neutron API. Users can balance traffic to pools consisting of multiple application servers and can provide high availability of their application through the use of intelligent health monitors. The LBaaS v2 API even supports SSL offloading with certificates managed by Barbican, another OpenStack project, as well as certificate bundles and SNI support.

The HAProxy driver offers functionality that can address basic load balancing needs, but may not be enough for most production environments. The load balancers are not highly-available and may present a weakness in the application's network architecture. The use of Octavia over the HAProxy driver is recommended for production-grade clouds and is compatible with the LBaaS v2 API.

More information on Octavia and how it may be implemented can be found at the following URL: <https://docs.openstack.org/octavia/pike/reference/introduction.html>.

In the next chapter, we will take a look at a few other advanced Neutron features, including 802.1q VLAN tagging (vlan-aware-vms) and BGP speaker functionality.

14

Advanced Networking Topics

OpenStack Networking provides many networking functions that enable users to develop topologies that best support their applications. While this book focuses on many of the core features of OpenStack Networking, there are times when certain use cases require advanced functionality. In this chapter, we will look at some advanced OpenStack Networking features, including the following:

- VLAN-aware VMs
- BGP Speaker
- Network availability zones

VLAN-aware VMs

VLAN tagging is a method in which a VLAN tag is added to an Ethernet header to help distinguish traffic from multiple networks carried over the same interface. In the architectures described so far in this book, an instance connected to multiple networks has a corresponding interface for each network. This works at small scale, but PCI limitations may cap the number of interfaces that can be attached to an instance. In addition, hot-plugging interfaces to running VMs when attaching new networks may have unexpected results.

The following diagram visualizes the concept of one vNIC per network:

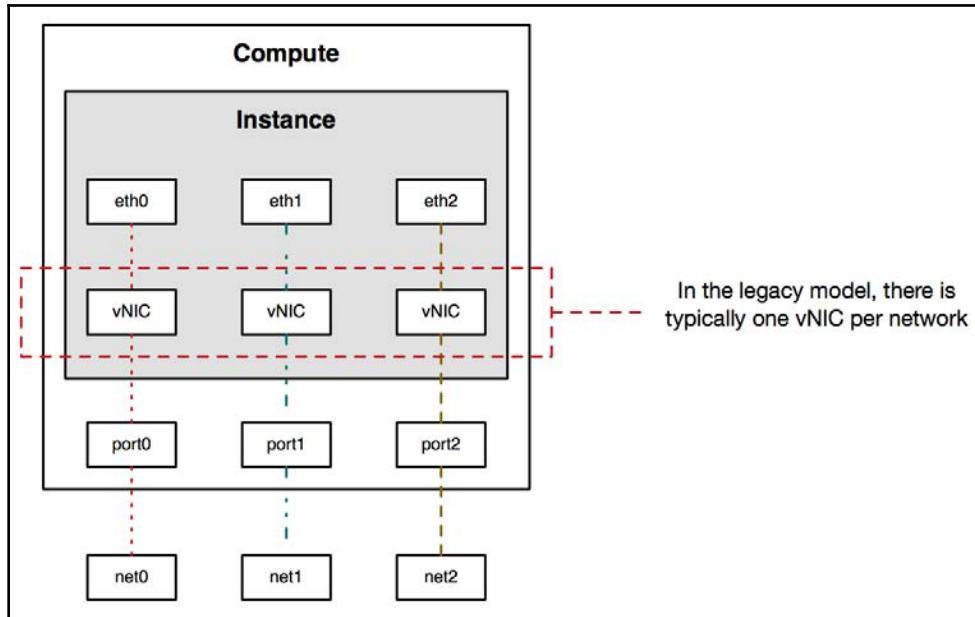


Figure 14.1

In Figure 14.1, a single vNIC is associated with a Neutron port. Neutron typically performs VLAN tagging at the virtual switch based on the `segmentation_id` provider attribute of the respective VLAN network. In this case, instances are not expected to perform any VLAN tagging themselves and any tagged traffic from the instance may be dropped by the compute node.

The trunk service plugin for Neutron allows VLAN-aware VM instances to tag traffic within the guest operating system, enabling the quick addition and removal of network sub-interfaces and setting up advanced use cases such as **Network Function Virtualization (NFV)**.

Configuring the trunk plugin

The trunk service plugin was first introduced in the Newton release of OpenStack Networking and supports both the Open vSwitch and Linux bridge ML2 mechanism drivers. Other drivers may be supported but are outside the scope of this book.

To install the trunk service plugin, modify the Neutron configuration file on the controller node hosting the Neutron API service. In this environment, API services are running on `controller01`. Append `trunk` to the list of existing service plugins as shown here:

```
[DEFAULT]
...
service_plugins = router,
neutron_lbaas.services.loadbalancer.plugin.LoadBalancerPluginv2, trunk
```



Be sure to append to the existing list of plugins, rather than replacing the list contents, to avoid issues with the API and existing network objects.

Close the file and restart the `neutron-server` service:

```
# systemctl restart neutron-server
```

Defining the workflow

Trunking in OpenStack Networking introduces some concepts that are important to the workflow needed when attaching trunks to an instance. Those concepts include the following:

- Trunks
- Parent ports
- Sub-ports

A trunk is an object that binds a parent port to sub-ports.

A parent port is a port that is attached to the instance and represents the trunk link from within the guest operating system. The interface looks and feels like a normal interface and inherits the MAC and IP addresses of the parent port. Any traffic sent over the associated interface inside the guest operating system is considered untagged and follows normal Neutron port behavior.

A sub-port is associated with a network and subnet and is not directly attached to an instance. Instead, the sub-port is associated with the trunk object. A VLAN sub-interface can be configured within the guest operating system using the properties of the sub-port and network, including a unique MAC address, IP address, and 802.1q VLAN tag. Traffic sent over the sub-interface is tagged by the guest operating system and forwarded through the parent port.

The following diagram visualizes the concept of one vNIC per instance using 802.1q VLAN encapsulation:

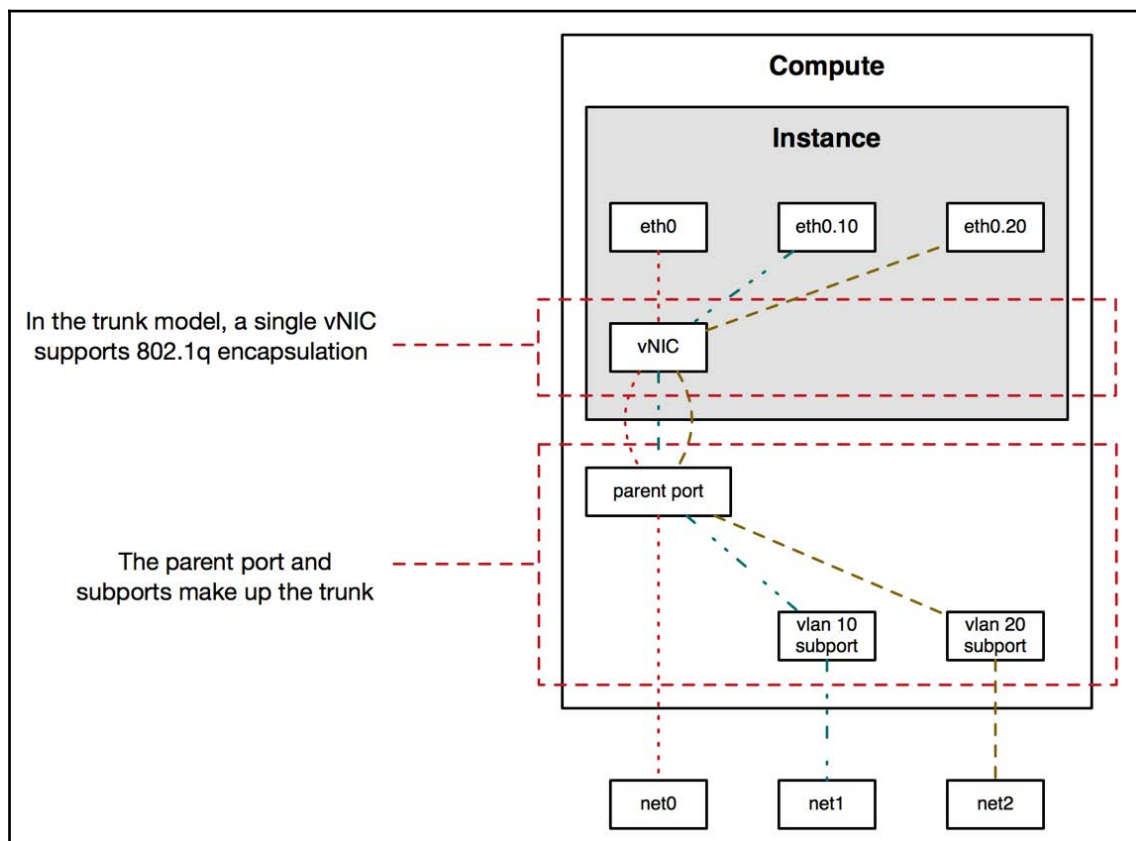


Figure 14.2

In Figure 14.2, a single vNIC is associated with a parent port. The parent port and multiple sub-ports are associated with a trunk. Instances can tag traffic using 802.1q encapsulation and the Neutron agent configures the underlying bridges to support this tagged traffic.

When leveraging trunking within OpenStack, the following workflow should be used:

1. Create networks and subnets for the trunk and sub-ports
2. Create the trunk
3. Add sub-ports to the trunk
4. Launch an instance connected to the trunk

Once the instance is active, additional sub-ports can be associated with the trunk and configured within the instance.

Managing trunks in the CLI

The following commands are used to manage trunk objects in the CLI:

Trunk Management Commands	Description
<code>network trunk create</code>	Create a trunk.
<code>network trunk delete</code>	Delete a given trunk.
<code>network trunk list</code>	List trunks that belong to a given project.
<code>network trunk set</code>	Update properties of a given trunk.
<code>network trunk show</code>	Show details of a given trunk.
<code>network trunk unset</code>	Unset properties of a given trunk.

Creating trunks in the CLI

To create a trunk, use the `openstack network trunk create` command shown here:

```
openstack network trunk create
[--description <description>]
--parent-port <parent-port>
[--subport <port=,segmentation-type=,segmentation-id=>]
[--enable | --disable]
[--project <project>]
[--project-domain <project-domain>]
<name>
```

The `--description` argument is optional and can be used to provide a description for the trunk.

The `--parent-port` argument is required and is used to associate a parent port with the trunk. The parent port is the port attached to the instance and acts as the trunk within the guest operating system.

The `--subport` argument is optional and is used to associate a sub-port to the trunk. A sub-port corresponds to an `802.1q` sub-interface within the guest operating system. If sub-ports are not specified, all tagged traffic from the instance will be dropped.

The `--project` and `--project-domain` arguments are optional and can be used to associate the trunk with a project and/or domain other than the creating project.

The `name` argument is required and can be used to specify a name for the trunk.

Deleting trunks in the CLI

To delete a trunk, use the `openstack network trunk delete` command shown here:

```
openstack network trunk delete <trunk> [<trunk> ...]
```

The `trunk` argument specifies the name or ID of the trunk to delete. Multiple trunks can be deleted simultaneously using a space-separated list.

Listing trunks in the CLI

To list all load balances, use the `openstack network trunk list` command shown here:

```
openstack network trunk list
```

Showing trunk details in the CLI

To show the details of a trunk, use the `openstack network trunk show` command shown here:

```
openstack network trunk show <trunk>
```

The `trunk` argument specifies the name or ID of the trunk to show.

Updating a trunk in the CLI

To update the attributes of a trunk, use the `openstack network trunk set` or `openstack network trunk unset` commands shown here:

```
openstack network trunk set
[--name <name>]
[--description <description>]
[--subport <port=,segmentation-type=,segmentation-id=>]
[--enable | --disable]
<trunk>

openstack network trunk unset --subport <subport> <trunk>
```

The `set` and `unset` commands can be used to add and remove sub-ports from a trunk without impacting the running instance.

Building a trunk

To demonstrate the creation and use of trunks within OpenStack Networking, I have configured the following networks:

ID	Name	Status	Subnets	Network Type	Segmentation ID
3f05db4d-c4ec-4cee-a616-2cffffbbd296e	RED_NET	ACTIVE	172.16.0.0/24	vlan	42
d8428f56-ba13-4aa7-97bb-76a6feb308bf	BLUE_NET	ACTIVE	192.168.150.0/24	vlan	41
db527151-2088-491b-80f3-b06b10715527	GREEN_NET	ACTIVE	172.24.100.0/24	vlan	40

The guest instance will run the Ubuntu 16.04 LTS operating system and will be connected to a single interface.

Creating the parent port

The first step to building a functional trunk is to first create the parent port. The parent port should be associated with a network that handles untagged traffic, in other words, the native VLAN. With the OpenStack client, create a port with the following attributes:

- Name: `parent0`
- Network: `GREEN_NET`

The following command can be used:

```
openstack port create --network GREEN_NET parent0
```

The output will resemble the following:

```
root@controller01:~# openstack port create --network GREEN_NET parent0
```

Field	Value
admin_state_up	UP
allowed_address_pairs	
binding_host_id	
binding_profile	
binding_vif_details	
binding_vif_type	unbound
binding_vnic_type	normal
created_at	2018-06-04T15:38:09Z
data_plane_status	None
description	
device_id	
device_owner	
dns_assignment	None
dns_name	None
extra_dhcp_opts	
fixed_ips	ip_address='172.24.100.7', subnet_id='4a2b4fcc-07b3-45ae-a5e2-dfec3a8ec2e6'
id	a6546d52-bc8b-4ca0-bd3a-bda1afcde6bb
ip_address	None
mac_address	fa:16:3e:1c:3d:d3
name	parent0
network_id	db527151-2088-491b-80f3-b06b10715527
option_name	None
option_value	None
port_security_enabled	True
project_id	9233b6b4f6a54386af63c0a7b8f043c2
qos_policy_id	None
revision_number	3
security_group_ids	fae5da02-5d96-43df-aebf-bf0210d353bc
status	DOWN
subnet_id	None
tags	
trunk_details	None
updated_at	2018-06-04T15:38:09Z

Creating a sub-port

Sub-ports should be associated with networks that will be tagged using 802.1q VLAN encapsulation inside the instance. Sub-ports are then associated with a trunk and correspond to tagged sub-interfaces within the guest operating system. With the openstack client, create a sub-port with the following attributes:

- Name: child-p0c1
- Network: RED_NET

The following command can be used:

```
openstack port create --network RED_NET child-p0c1
```

The output will resemble the following:

```
root@controller01:~# openstack port create --network RED_NET child-p0c1
```

Field	Value
admin_state_up	UP
allowed_address_pairs	
binding_host_id	
binding_profile	
binding_vif_details	
binding_vif_type	unbound
binding_vnic_type	normal
created_at	2018-06-04T15:41:50Z
data_plane_status	None
description	
device_id	
device_owner	
dns_assignment	None
dns_name	None
extra_dhcp_opts	
fixed_ips	ip_address='172.16.0.11', subnet_id='bfe04cde-2c80-413e-9d69-b10a1aabb2f7'
id	cd6850a0-186d-47ac-8981-6fa52088088e
ip_address	None
mac_address	fa:16:3e:a9:48:cc
name	child-p0c1
network_id	3f05db4d-c4ec-4cee-a616-2cfffbbd296e
option_name	None
option_value	None
port_security_enabled	True
project_id	9233b6b4f6a54386af63c0a7b8f043c2
qos_policy_id	None
revision_number	3
security_group_ids	fae5da02-5d96-43df-aebf-bf0210d353bc
status	DOWN
subnet_id	None
tags	
trunk_details	None
updated_at	2018-06-04T15:41:51Z



Like any other port, when a sub-port is created, Neutron dynamically assigns a MAC address. However, when creating VLAN sub-interfaces inside an instance, the sub-interface may inherit the MAC address of the parent interface. This behavior is acceptable since the interfaces are on two different networks and MAC addresses do not pass the Layer 2 boundary. However, it may be problematic from a port security standpoint. When creating sub-interfaces in an instance, you will need to specify the MAC address Neutron assigned for the sub-port or create the sub-port with the same MAC address of the parent port.

Creating a trunk

The last step to building a functional trunk is to create a trunk object and associate parent and sub-ports. The information required for the trunk includes a name, a parent port with segment details, and sub-port(s) with segment details. With the OpenStack client, create a trunk object with the following attributes:

- Name: `trunk0`
- Parent port: `parent0`
- Sub-port: `child-p0c1`
- Sub-port VLAN: 42

The following command can be used:

```
openstack network trunk create \
--parent-port parent0 \
--subport port=child-p0c1,segmentation-type=vlan,segmentation-id=42 \
trunk0
```

The output will resemble the following:

```
root@controller01:~# openstack network trunk create \
> --parent-port parent0 \
> --subport port=child-p0c1,segmentation-type=vlan,segmentation-id=42 \
> trunk0
```

Field	Value
admin_state_up	UP
created_at	2018-06-04T15:58:08Z
description	
id	82597947-6e0a-40f3-b4ff-ccedd8c74ff8
name	trunk0
port_id	a6546d52-bc8b-4ca0-bd3a-bda1afcde6bb
project_id	9233b6b4f6a54386af63c0a7b8f043c2
revision_number	0
status	DOWN
sub_ports	port_id='cd6850a0-186d-47ac-8981-6fa52088088e', segmentation_id='42', segmentation_type='vlan'
tags	[]
tenant_id	9233b6b4f6a54386af63c0a7b8f043c2
updated_at	2018-06-04T15:58:08Z

Booting an instance with a trunk

Now that the trunk has been created and associated with a parent and sub-port, an instance can be booted and attached only to the parent port. Since a VIF is not being created for the subinterface(s), there's no need to attach them at boot. Logic within Neutron will associate traffic from sub-interfaces inside the instance with the parent interface.

The following OpenStack command syntax can be used to boot the instance with a parent port attached:

```
openstack server create \
--image <image> \
--flavor<flavor> \
--key-name <keypair name> \
--nic port-id=<parent port> \
<name>
```

```
root@controller01:~# openstack server create \
> --image 'ubuntu-xenial-16.04' \
> --flavor small \
> --key-name james \
> --nic port-id=parent0 \
> ServerWithTrunk
```

Field	Value
OS-DCF:diskConfig	MANUAL
OS-EXT-AZ:availability_zone	
OS-EXT-SRV-ATTR:host	None
OS-EXT-SRV-ATTR:hypervisor_hostname	None
OS-EXT-SRV-ATTR:instance_name	
OS-EXT-STS:power_state	NOSTATE
OS-EXT-STS:task_state	scheduling
OS-EXT-STS:vm_state	building
OS-SRV-USG:launched_at	None
OS-SRV-USG:terminated_at	None
accessIPv4	
accessIPv6	
addresses	
adminPass	Gh9zhNumek47
config_drive	
created	2018-06-04T16:06:59Z
flavor	small (280d07fb-b932-464a-9e26-817522286be7)
hostId	
id	09c21c1f-1b0e-4794-a0a6-ae37d0fd243a
image	ubuntu-xenial-16.04 (ffdb76f0-6cce-4d79-99a5-ccc6e76d530a)
key_name	james
name	ServerWithTrunk
progress	0
project_id	9233b6b4f6a54386af63c0a7b8f043c2
properties	
security_groups	name='default'
status	BUILD
updated	2018-06-04T16:07:01Z
user_id	8a679d8b7b574e54a5cfe02c422bbe68
volumes_attached	

Configuring the instance

Once booted, the instance should be available via the IP address of the parent port and can be accessed from a namespace or workstation if the proper routing is in place. From within the respective `qdhcp` namespace, confirm connectivity to the instance:

```
root@controller01:~# ip netns exec qdhcp-db527151-2088-491b-80f3-b06b10715527 ssh -i james.key ubuntu@172.24.100.7
The authenticity of host '172.24.100.7 (172.24.100.7)' can't be established.
ECDSA key fingerprint is SHA256:WJ1h0Xdboy8iaz5t4lsjm2ves9eYStYsvU3Lqmx6eyk.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '172.24.100.7' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-112-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.

ubuntu@serverwithtrunk:~$ sudo su
root@serverwithtrunk:~#
root@serverwithtrunk:~# ip addr list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default 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
2: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether fa:16:3e:1c:3d:d3 brd ff:ff:ff:ff:ff:ff
    inet 172.24.100.7/24 brd 172.24.100.255 scope global ens3
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe1c:3dd3/64 scope link
        valid_lft forever preferred_lft forever
```

The `ip addr list` command confirms the IP address of the parent port has been configured by DHCP, and our ability to connect to the instance demonstrates traffic over the parent port of the trunk is untagged and behaves like an ordinary port.

Using utilities from the `iproute2` package, we can configure a sub-interface using the attributes of our sub-port shown here:

- MAC address: `fa:16:3e:a9:48:cc`
- VLAN ID: 42

For convenience, an 802.1q VLAN sub-interface can be created with the `ip link` command. Or, on Ubuntu systems, the `/etc/network/interfaces` file can be modified for persistent interface configuration.

Using the `ip link add` command, create the sub-interface and modify the MAC address using the following commands:

```
# ip link add link ens3 name ens3.42 type vlan id 42
# ip link set dev ens3.42 address fa:16:3e:a9:48:cc
# ip link set ens3.42 up
```

Using the `ip addr list` command, the newly configured interface should be visible in an UP state and have the specified MAC address:

```
root@serverwithtrunk:~# ip addr list
***
2: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether fa:16:3e:1c:3d:d3 brd ff:ff:ff:ff:ff:ff
    inet 172.24.100.7/24 brd 172.24.100.255 scope global ens3
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe1c:3dd3/64 scope link
        valid_lft forever preferred_lft forever
3: ens3.42@ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:a9:48:cc brd ff:ff:ff:ff:ff:ff
    inet6 fe80::f816:3eff:fea9:48cc/64 scope link
        valid_lft forever preferred_lft forever
```

The interface does not yet have an IP address, so connectivity cannot be confirmed just yet. There are two options available for IP address configuration:

- DHCP
- Static address configuration

To utilize DHCP, simply run the `dhclient ens3.42` command as root or with `sudo` privileges. When operating properly, the Neutron DHCP server should return the assigned address and the interface will be configured automatically as shown here:

```
root@serverwithtrunk:~# ip a
***
2: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether fa:16:3e:1c:3d:d3 brd ff:ff:ff:ff:ff:ff
    inet 172.24.100.7/24 brd 172.24.100.255 scope global ens3
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fe1c:3dd3/64 scope link
        valid_lft forever preferred_lft forever
3: ens3.42@ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fa:16:3e:a9:48:cc brd ff:ff:ff:ff:ff:ff
    inet 172.16.0.11/24 brd 172.16.0.255 scope global ens3.42
        valid_lft forever preferred_lft forever
    inet6 fe80::f816:3eff:fea9:48cc/64 scope link
        valid_lft forever preferred_lft forever
```



The specifics of a dynamic or static, but persistent, interface configuration will vary based on the installed guest operating system.

A quick ping to the DHCP server of the RED_NET network demonstrates tagged traffic from the instance's interface traverses the network appropriately:

```
root@serverwithtrunk:~# ping 172.16.0.2 -c 2
PING 172.16.0.2 (172.16.0.2) 56(84) bytes of data.
64 bytes from 172.16.0.2: icmp_seq=1 ttl=64 time=0.567 ms
64 bytes from 172.16.0.2: icmp_seq=2 ttl=64 time=1.29 ms

--- 172.16.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.567/0.933/1.299/0.366 ms
```

Reviewing the network plumbing

On the compute node hosting instance, a quick look at the bridges using `brctl show` does not reveal anything unique about the configuration that would support tagging within the instance:

bridge name	bridge id	STP enabled	interfaces
brq3f05db4d-c4	8000.000c29fa3f25	no	ens224.42 tapcd6850a0-18 Child Port
brq471ba196-e7	8000.000000000000	no	
brq758070f9-ec	8000.000c29fa3f25	no	ens224.30
brqce1fe7cd-7b	8000.000c29fa3f25	no	ens224.43
brqdb527151-20	8000.000c29fa3f25	no	ens224.40 tapa6546d52-bc Parent Port

However, a deeper look at the `tap` interface that corresponds to the sub-port reveals that the interface has been configured as a VLAN interface off the parent port using VLAN ID 42:

```
root@compute01:~# ip -d link show tapcd6850a0-18
33: tapcd6850a0-18@tapa6546d52-bc: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master brq3f05db4d-c4
link/ether fe:16:3e:1c:3d:d3 brd ff:ff:ff:ff:ff:ff promiscuity 1
vlan protocol 802.1Q id 42 <REORDER_HDR>
bridge_slave state forwarding priority 32 cost 100 hairpin off guard off root_block off fastleave off learning
```

Performing a packet capture on the `tap` interface associated with the parent port, we can see traffic leaving the instance's sub-interface and through the parent interface is tagged with 802.1q VLAN ID 42:

```
root@compute01:~# tcpdump -i tapa6546d52-bc -ne icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on tapa6546d52-bc, link-type EN10MB (Ethernet), capture size 262144 bytes
18:06:08.085655 fa:16:3e:a9:48:cc > fa:16:3e:2d:cc:46, ethertype 802.1Q (0x8100), length 102: vlan 42 p 0, ethertype IPv4, 172.16.0.11 > 172.16.0.2:
18:06:08.086188 fa:16:3e:2d:cc:46 > fa:16:3e:a9:48:cc, ethertype 802.1Q (0x8100), length 102: vlan 42 p 0, ethertype IPv4, 172.16.0.2 > 172.16.0.11:
18:06:09.087545 fa:16:3e:a9:48:cc > fa:16:3e:2d:cc:46, ethertype 802.1Q (0x8100), length 102: vlan 42 p 0, ethertype IPv4, 172.16.0.11 > 172.16.0.2:
18:06:09.088131 fa:16:3e:2d:cc:46 > fa:16:3e:a9:48:cc, ethertype 802.1Q (0x8100), length 102: vlan 42 p 0, ethertype IPv4, 172.16.0.2 > 172.16.0.11:
^C
4 packets captured
4 packets received by filter
0 packets dropped by kernel
```

The `compute` node strips the VLAN ID and forwards the traffic out the `tap` interface associated with the sub-port untagged:

```
root@compute01:~# tcpdump -i tapcd6850a0-18 -ne icmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on tapcd6850a0-18, link-type EN10MB (Ethernet), capture size 262144 bytes
18:06:08.085673 fa:16:3e:a9:48:cc > fa:16:3e:2d:cc:46, ethertype IPv4 (0x0800), length 98: 172.16.0.11 > 172.16.0.2: ICMP echo request
18:06:08.086182 fa:16:3e:2d:cc:46 > fa:16:3e:a9:48:cc, ethertype IPv4 (0x0800), length 98: 172.16.0.2 > 172.16.0.11: ICMP echo reply
18:06:09.087566 fa:16:3e:a9:48:cc > fa:16:3e:2d:cc:46, ethertype IPv4 (0x0800), length 98: 172.16.0.11 > 172.16.0.2: ICMP echo request
18:06:09.088122 fa:16:3e:2d:cc:46 > fa:16:3e:a9:48:cc, ethertype IPv4 (0x0800), length 98: 172.16.0.2 > 172.16.0.11: ICMP echo reply
^C
4 packets captured
4 packets received by filter
0 packets dropped by kernel
```

At this point, normal Linux bridge-related traffic operations take place and traffic is forwarded over the physical network infrastructure as described earlier in this book.



When an Open vSwitch network agent is used, flow rules may be used instead of the methods described here.

BGP dynamic routing

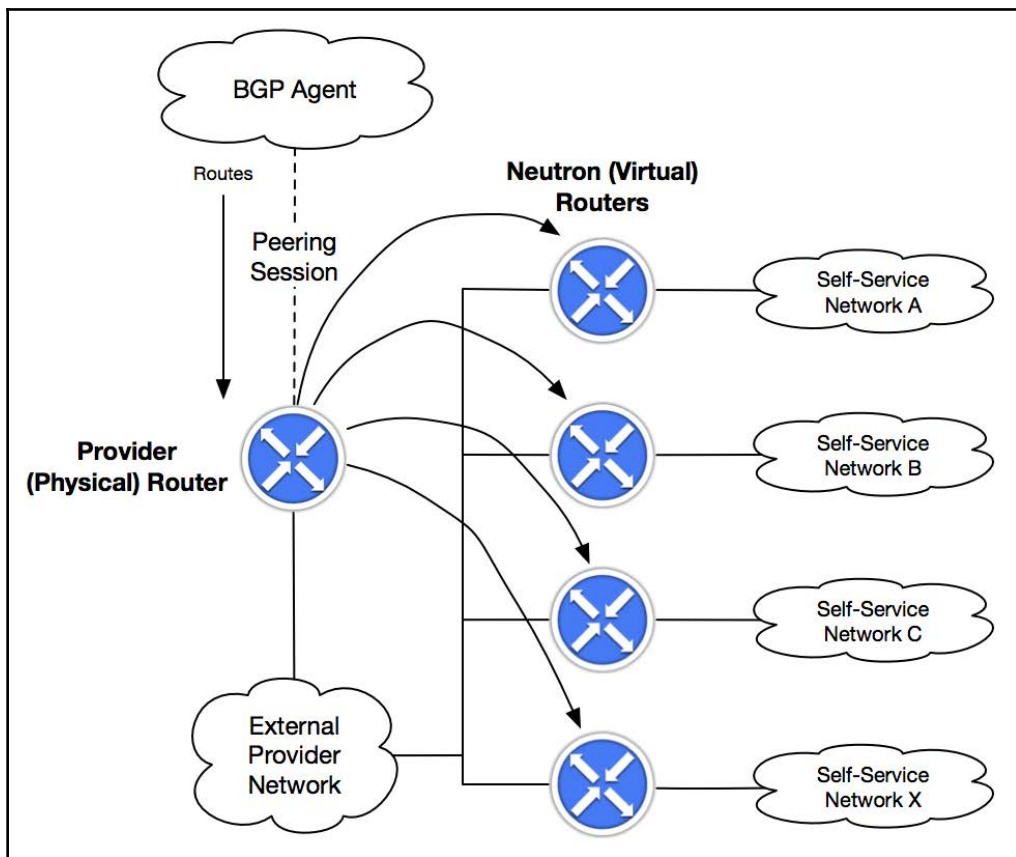
BGP dynamic routing enables the advertisement of self-service IPv4 and IPv6 network prefixes to network devices that support BGP, including many physical and virtual router and firewall devices. By advertising self-service networks attached to Neutron routers, the use of floating IPs can be avoided.



BGP speaker functionality relies on address scopes and requires knowledge of their operation for proper deployment.

BGP dynamic routing consists of a Neutron API service plugin that implements the Networking service extension and an agent that manages BGP peering sessions. A cloud administrator creates and configures a BGP speaker using the CLI or API and manually schedules it to one or more hosts running the agent.

The following diagram demonstrates the BGP agent's peering relationship with a physical router that enables the physical route to reach self-service networks behind Neutron routers via a common provider network:



Prefix advertisement requirements

BGP dynamic routing advertises prefixes for self-service networks and host routes for floating IP addresses. The following conditions must be met to advertise a self-service network:

- The external and self-service network must reside in the same address scope.
- The Neutron router must contain an interface on the self-service subnet and a gateway on the external provider network.
- The BGP speaker must peer with a device connected to the same external provider network as the Neutron router.
- The BGP speaker has the `advertise_tenant_networks` attribute set to `True`.

Advertisement of a floating IP address requires satisfying the following conditions:

- The BGP speaker must peer with a device connected to the same external provider network as the Neutron router hosting the floating IP.
- The BGP speaker has the `advertise_floating_ip_host_routes` attribute set to `True`.

Operations with distributed virtual routers

In deployments using distributed virtual routers, the BGP speaker advertises floating IP addresses and self-service networks differently. For floating IP addresses, the BGP speaker advertises the floating IP agent gateway on the corresponding compute node as the next-hop IP address. For self-service networks using SNAT, the BGP speaker advertises the DVR SNAT node as the next-hop IP address. This means that traffic directed to a self-service network behind a distributed virtual router must traverse a `network` node, while traffic to a floating IP will avoid routing through a `network` node.

Configuring BGP dynamic routing

The BGP dynamic routing plugin was first introduced in the Mitaka release of OpenStack Networking and supports the native L3 agent provided by Neutron.

To install the BGP service plugin, modify the Neutron configuration file on the controller node hosting the Neutron API service. In this environment, API services are running on `controller01`. Add the BGP plugin to the list of existing service plugins as shown here:

```
[DEFAULT]
...
service_plugins = router,
neutron_lbaas.services.loadbalancer.plugin.LoadBalancerPluginv2,
trunk,neutron_dynamic_routing.services.bgp.bgp_plugin.BgpPlugin
```



Be sure to append to the existing list of plugins, rather than replacing the list contents, to avoid issues with the API and existing network objects.

Installing the agent

To install the Neutron BGP agent, run the following command on the `controller01` node:

```
# apt install neutron-bgp-dragent
```

Neutron stores the BGP agent configuration in the `/etc/neutron/bgp_dragent.ini` file. The most common configuration options will be covered in the following sections.

Configuring the agent

Edit the BGP agent configuration file and add the following configuration, substituting the `bgp_router_id` value with the IP address of the respective controller node, if necessary:

```
[bgp]
...
bgp_speaker_driver =
neutron_dynamic_routing.services.bgp.agent.driver.ryu.driver.RyuBgpDriver
bgp_router_id = 10.10.0.100
```



In this example, the management IP address of the host is used. Using an IP address and interface dedicated to router advertisements is possible but outside the scope of this book.

Restarting services

For the configuration changes to take effect, restart the Neutron API service and the Neutron BGP agent with the following command:

```
# systemctl restart neutron-server neutron-bgp-dragent
```

Use the OpenStack client to verify the BGP agent is checked in and ready for use:

```
root@controller01:~# openstack network agent list --agent-type bgp
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
c5f503bd-7ad4-42c6-ab67-303ffad75cb8	BGP dynamic routing agent	controller01	None	:~)	UP	neutron-bgp-dragent

Managing BGP speakers in the CLI

As of the Pike release of the OpenStack client, BGP speaker-related commands are not yet available. The following neutron client commands are used to manage BGP speaker objects in the CLI:

BGP speaker management Commands	Description
bgp-dragent-list-hosting-speaker	Lists dynamic routing agents hosting a BGP speaker.
bgp-dragent-speaker-add	Adds a BGP speaker to a dynamic routing agent.
bgp-dragent-speaker-remove	Removes a BGP speaker from a dynamic routing agent.
bgp-peer-create	Creates a BGP peer.
bgp-peer-delete	Deletes a BGP peer.
bgp-peer-list	List BGP peers.
bgp-peer-show	Shows information of a given BGP peer.
bgp-peer-update	Updates BGP peer information.
bgp-speaker-advertiseroute-list	Lists routes advertised by a given BGP speaker.
bgp-speaker-create	Creates a BGP speaker.
bgp-speaker-delete	Deletes a BGP speaker.
bgp-speaker-list	Lists BGP speakers.
bgp-speaker-list-on-dragent	Lists BGP speakers hosted by a dynamic routing agent.
bgp-speaker-network-add	Adds a network to the BGP speaker.

bgp-speaker-network-remove	Removes a network from the BGP speaker.
bgp-speaker-peer-add	Adds a peer to the BGP speaker.
bgp-speaker-peer-remove	Removes a peer from the BGP speaker.
bgp-speaker-show	Shows information of a given BGP speaker.
bgp-speaker-update	Updates BGP speaker information.

Please refer to the upstream documentation on configuring address scopes and subnet pools for use with BGP dynamic routing. The following URL provides working examples that demonstrate BGP dynamic

routing: <https://docs.openstack.org/neutron/pike/admin/config-bgp-dynamic-routing.html>.

Network availability zones

A network availability zone is a logical construct used to define a group of network nodes that share similar power, network, and cooling systems. Network resources can be scheduled to multiple availability zones to ensure a high level of reliability when a single zone fails. This is similar to how Nova availability zones work, as they are used to group compute nodes in a similar way to ensure components of an application are not in the same failure domain.

Configuring network availability zones

Availability zone support for Neutron was first introduced in the Mitaka release of OpenStack Networking, and supports both the DHCP and L3 agents included with Neutron. Other drivers may be supported but are outside the scope of this book.

In the environment built throughout this book, the Neutron L3 agent has been installed on most of the nodes to demonstrate standalone, highly available, and distributed virtual routers. A quick look at the Neutron agent list shows they all share a common availability zone based on default agent configuration options:

```
root@controller01:~# openstack network agent list --agent-type='l3'
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
ee380688-0ea7-4e54-9889-c44ddd933678	L3 agent	compute01	nova	:-)	UP	neutron-l3-agent
f9f8e9a3-4332-4693-a652-40837ffdbd51	L3 agent	compute02	nova	:-)	UP	neutron-l3-agent
7cdeb762-ab03-41a3-b924-9e64dde50621	L3 agent	compute03	nova	:-)	UP	neutron-l3-agent
9c75984a-2cfa-4db6-afc6-b7e1e4901bbd	L3 agent	controller01	nova	:-)	UP	neutron-l3-agent
63214965-cd41-4aa1-a98f-a3e202929106	L3 agent	snat01	nova	:-)	UP	neutron-l3-agent

To modify the availability zone for a particular L3 agent, edit the respective agent configuration file at `/etc/neutron/l3_agent.ini` and modify the `availability_zone` configuration option as shown here:

```
[agent]
...
# Availability zone of this node (string value)
# availability_zone = nova
availability_zone = AZ1
```

Restart the L3 agent with the following command:

```
# systemctl restart neutron-l3-agent
```

A fresh look at the agent list reflects the change:

```
root@controller01:~# openstack network agent list --agent-type='l3'
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
ee380688-0ea7-4e54-9889-c44ddd933678	L3 agent	compute01	AZ1	:-)	UP	neutron-l3-agent
f9f8e9a3-4332-4693-a652-40837ffdbd51	L3 agent	compute02	nova	:-)	UP	neutron-l3-agent
7cdeb762-ab03-41a3-b924-9e64dde50621	L3 agent	compute03	nova	:-)	UP	neutron-l3-agent
9c75984a-2cfa-4db6-afc6-b7e1e4901bbd	L3 agent	controller01	nova	:-)	UP	neutron-l3-agent
63214965-cd41-4aa1-a98f-a3e202929106	L3 agent	snat01	nova	:-)	UP	neutron-l3-agent

DHCP agents can also be associated with availability zones in a similar manner. To modify the availability zone for a particular DHCP agent, edit the respective agent configuration file at `/etc/neutron/dhcp_agent.ini` and modify the `availability_zone` configuration option as shown here:

```
[agent]
...
# Availability zone of this node (string value)
# availability_zone = nova
availability_zone = AZ1
```

Restart the DHCP agent with the following command:

```
# systemctl restart neutron-dhcp-agent
```

Scheduling routers to availability zones

Neutron routers can be directed to a particular availability zone during the router creation process. Using the `openstack router create` command, we can schedule a router to availability zone AZ1 with the `--availability-zone-hint` argument as shown here:

```
root@controller01:~# openstack router create Router01 --availability-zone-hint AZ1
```

Field	Value
admin_state_up	UP
availability_zone_hints	AZ1
availability_zones	
created_at	2018-06-05T12:56:12Z
description	
distributed	False
external_gateway_info	None
flavor_id	None
ha	False
id	608c793a-0d18-4244-87e9-7a548a605fb1
name	Router01
project_id	9233b6b4f6a54386af63c0a7b8f043c2
revision_number	None
routes	
status	ACTIVE
tags	
updated_at	2018-06-05T12:56:12Z

After attaching the router to the external provider network `GATEWAY_NET`, we can see the router was scheduled to an L3 agent in availability zone AZ1:

```
root@controller01:~# openstack network agent list --router Router01
```

ID	Agent Type	Host	Availability Zone	Alive	State	Binary
ee380688-0ea7-4e54-9889-c44ddd933678	L3 agent	compute01	AZ1	:~)	UP	neutron-l3-agent

Directing a router to a particular availability zone does not ensure it will be scheduled to an L3 agent in that zone, however, as the directive is more of a scheduler hint than an outright requirement. Neutron performs best effort scheduling using availability zone hints. Multiple availability zones can be specified, if desired, by repeating the `--availability-zone-hint` argument as necessary.

Scheduling DHCP services to availability zones

The DHCP service for a given network is something often taken for granted and is not directly seen, let alone configurable, by users or operators via the API. Neutron has scheduled DHCP services across multiple DHCP agents for some time using the following configuration option in the Neutron configuration file at `/etc/neutron/neutron.conf`:

```
# Number of DHCP agents scheduled to host a tenant network.
# If this number is greater than 1, the scheduler automatically
# assigns multiple DHCP agents for a given tenant network,
# providing high availability for DHCP service.
# (integer value)
# dhcp_agents_per_network = 1
```

When `dhcp_agents_per_network` is greater than 1, and more than one DHCP agent exists in the environment, Neutron will automatically schedule a network to multiple DHCP agents up to the specified value. Using the `openstack network create` command with the `--availability-zone-hint` argument, we can suggest to the scheduler that DHCP services be split amongst multiple availability zones as shown here:

```
openstack network create network01
--availability-zone-hint AZ1
--availability-zone-hint AZ2
```

Directing a network's DHCP services to a particular availability zone does not ensure it will be scheduled to a DHCP agent in that zone, however, as the directive is more of a scheduler hint than an outright requirement. Neutron performs best effort scheduling using availability zone hints. Multiple availability zones can be specified, if desired, by repeating the `--availability-zone-hint` argument as necessary.

More information on the use of network availability zones can be found in upstream documentation available at the following URL: <https://docs.openstack.org/neutron/pike/admin/config-az.html>.

Summary

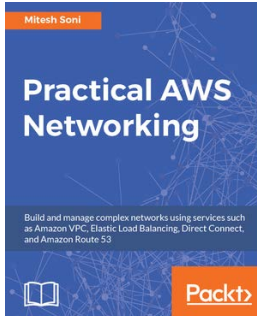
As workloads for OpenStack-based clouds have evolved, so have the features provided by OpenStack Networking. VLAN-aware VMs, BGP speakers, and network availability zones are just a few of the advanced network features that have been introduced in Neutron over the last few release cycles and there are more to come in future releases. Used in tandem, these features can provide users with the ability to define rich network topologies and provide high-availability to a number of different applications and workloads in the cloud.

The release notes for OpenStack Networking are a great place to find information on recently released features as well as bug fixes for existing functions. To find the release notes for a given release, please visit the following

URL: <https://docs.openstack.org/releasenotes/neutron/index.html>.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

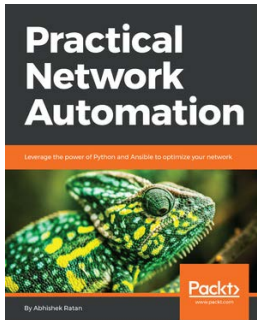


Practical AWS Networking

Mitesh Soni

ISBN: 978-1-78839-829-9

- Overview of all networking services available in AWS.
- Gain Work with load balance application across different regions.
- Learn auto scale instance based on the increase and decrease of the traffic.
- Deploy application in highly available and fault tolerant manner.
- Configure Route 53 for a web application.
- Troubleshooting tips and best practices at the end.

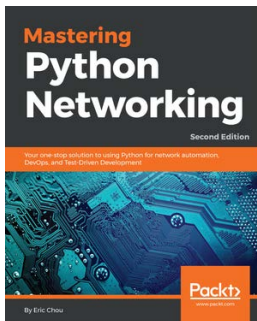


Practical Network Automation

Abhishek Ratan

ISBN: 978-1-78829-946-6

- Get the detailed analysis of Network automation
- Trigger automations through available data factors
- Improve data center robustness and security through specific access and data digging
- Get an Access to APIs from Excel for dynamic reporting
- Set up a communication with SSH-based devices using netmiko
- Make full use of practical use cases and best practices to get accustomed with the various aspects of network automation



Mastering Python Networking - Second Edition

Eric Chou

ISBN: 978-1-78913-599-2

- Use Python libraries to interact with your network
- Integrate Ansible 2.5 using Python to control Cisco, Juniper, and Arista eAPI network devices
- Leverage existing frameworks to construct high-level APIs
- Learn how to build virtual networks in the AWS Cloud
- Understand how Jenkins can be used to automatically deploy changes in your network
- Use PyTest and Unittest for Test-Driven Network Development

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

access control policies

- creating, in CLI 239, 240
- deleting, in CLI 240
- details, displaying in CLI 241
- listing, in CLI 240
- managing, in CLI 239
- updating, in CLI 241
- working with 239

Advanced Message Queue Protocol (AMQP) 40

aggregation groups 23

allowed address pairs 236

allowed-address-pairs extension 236

API network 20

B

BGP dynamic routing

- about 411
- agent, configuring 414
- agent, installing 414
- BGP speakers, managing in CLI 415
- configuring 413
- operations, with distributed virtual routers 413
- prefix advertisement, requisites 413
- services, restarting 415

C

CentOS

- URL 49

centralized SNAT

- about 335
- packet, tracing through namespace 340, 342
- routing policy database, used 339
- topology, reviewing 335, 338

CLI

- access control policies, creating 239, 240

access control policies, deleting 240

access control policies, listing 240

access control policies, managing 239

access control policies, updating 241

details of access control policies, displaying 241

details of security group rule, displaying 218

details, displaying of security groups 215

security group rules, creating 216

security groups, creating 214

security groups, deleting 215

security groups, listing 215

security groups, managing 213

security groups, removing from instances/ports 220

security groups, updating 216

Compute service

additional compute tasks 56

additional controller tasks 53

authentication settings, configuring 52

compute node components, configuring 54

compute node components, installing 54

compute node, adding to cell database 57

configuring 50

controller node components, configuring 51

controller node components, installing 51

database, configuring 50

defining 51

endpoints, defining 51

installing 50

Nova user, defining 51

control plane 19

core plugin 63

D

dashboard

security group rules, managing 229

security group, creating 228

- Data plane traffic 19
- dedicated HA network 302
- dedicated network node
 - using 27
- default chains 211
- DHCP agent
 - agents, verifying 109
 - services, restarting 109
- DHCP namespace
 - about 204
 - DHCP, used for injecting route 206
 - manual route, adding 205

E

- east-west traffic
 - routing, between instances 324
- external networks
 - about 20
 - policies, creating for 247, 248, 249

F

- Firewall as a Service (FWaaS) 11
- flat network 65, 90, 94
- floating IP management
 - about 261
 - attributes, displaying in CLI 264
 - floating IPs, associating with ports in CLI 263
 - floating IPs, creating in CLI 262
 - floating IPs, deleting in CLI 264
 - floating IPs, disassociating in CLIs 264
 - floating IPs, listing in CLI 263
- floating IPs
 - assigning 284, 287
 - associating, to instances in dashboard 295
 - disassociating, in dashboard 296
 - namespace 343
 - packet, tracing through namespace 345
 - proxy ARP, using 349
 - reassigning 288
 - traffic, returning 349
 - traffic, sending from instance 345, 347
 - using, for distributed virtual routers 343

G

- Generic Network Virtualization Encapsulation (GENEVE) 16, 65
- Generic Router Encapsulation (GRE) 16, 65
- Glance
 - additional images, installing 49
 - additional settings, configuring 48
 - authentication settings, configuring 47
 - components, configuring 46
 - components, installing 46
 - configuring 45
 - database, configuring 45
 - endpoints, defining 46
 - image service installation, verifying 48
 - installing 45
 - service, defining 46
 - user, defining 46
- groups, VRRP 299, 300
- guest network 21

H

- highly-available routers
 - creating 309
 - decomposing 309, 310, 311, 312, 314
 - deleting 309
 - failover, executing 315
 - keepalived configuration, examining 314
 - networking 302
 - working with 308
- hostnames
 - setting 37

I

- identity service
 - configuring 41
 - database, configuring 41
 - installing 41
- instances
 - addresses, obtaining 196, 198
 - attaching, to network at creation 192
 - attaching, to networks 191
 - DHCP lease cycle, viewing 198, 203
 - DHCP namespace 204
 - DHCP, troubleshooting 203

- east-west traffic, routing 324
- metadata, retrieving 204
- multiple interfaces, attaching 193
- network interfaces, attach to running instances 194
- network interfaces, detaching 195
- network, specifying 193
- port, specifying 193
- router ports, distributing 327
- security groups, applying to 219, 231
- topology, reviewing 324, 326
- traffic, demonstrating 329, 331, 334
- working 328
- integration bridge 116
- internal network 19
- ipset
 - using 212, 213
- iptables
 - about 210
 - URL 212

K

- keepalived
 - using 298
- Keystone
 - Apache HTTP server, configuring 43
 - API endpoint, defining 44
 - drivers, configuring 42
 - environment variables, setting 43
 - identity service, bootstrapping 43
 - installing 42
 - projects, defining 45
 - roles, defining 45
 - services, defining 44
 - tokens, configuring 42
 - used, for configuring Neutron 70
 - users, defining 45

L

- L3 agents
 - agent mode, setting 306
 - configuring 306
 - installing 306
 - interface driver, defining 306
- Linux bridge driver

- using 85
- Linux bridge
 - agent configuration file, updating 106
 - agent, configuring 105
 - agent, installing 105
 - driver, configuring 105
 - driver, using with DHCP agent configuration 109
 - firewall driver 108
 - flat network 90, 94
 - L2 population 107
 - local IP 108
 - local network 96, 97
 - physical interface mappings 106
 - traffic flow, visualizing 86, 90
 - VLAN 87
 - VXLAN 94
 - VXLAN, enabling 107
- listeners
 - creating, in CLI 375
 - deleting, in CLI 376
 - details, displaying in CLI 376
 - listing, in CLI 376
 - managing, in CLI 375
 - updating, in CLI 377
- load balancer management
 - creating, in dashboard 391
 - floating IP, assigning 395
 - health monitor details, displaying 374
 - health monitor, creating in CLI 372, 373
 - health monitor, deleting in CLI 373
 - health monitor, updating 374
 - health monitors, listing in CLI 374
 - health monitors, managing in CLI 371
 - in CLI 363
 - in dashboard 389
 - listeners, managing in CLI 375
 - load balancer details, displaying in CLI 365
 - load balancer statistics, displaying in CLI 365
 - load balancer, updating in CLI 366
 - load balancers, creating in CLI 364
 - load balancers, deleting in CLI 364
 - load balancers, listing in CLI 365
 - pool details, displaying in CLI 368
 - pool member details, displaying 370
 - pool member, updating 371

- pool members, creating in CLI 369
- pool members, deleting 370
- pool members, listing 370
- pool members, managing in CLI 369
- pool, creating in CLI 366
- pool, deleting in CLI 367
- pool, updating in CLI 368
- pools, listing in CLI 368
- pools, managing in CLI 366
- status, displaying in CLI 365
- load balancer
 - building 377
 - creating 378
 - health monitor, creating 382
 - health monitors, observing 387
 - LBaaS network namespace 385
 - listener, creating 384
 - load balancer functionality, confirming 386
 - pool members, creating 380
 - pool, creating 379
 - virtual IP, connecting externally 388
- Load Balancing as a Service (LBaaS v2)
 - about 11
 - installing 359
 - Neutron LBaaS agent service, configuring 359
 - Neutron, configuring 361
 - user group, defining 361
- load balancing
 - algorithms 354
 - API service, restarting 362
 - database schema, updating 362
 - fundamentals 353
 - integrating, into network 356, 359
 - monitoring 355
 - network namespace 359
 - Neutron LBaaS agent, restarting 362
 - session persistence 355
- local network 64

M

- management network 19
- maximum transmission unit (MTU) 95
- Mechanism driver
 - about 65
 - agent-based 65

- controller-based 65
- Top-of-Rack 65
- memcached service
 - configuring 41
 - installing 41
- mesh network 14
- messaging server
 - configuring 40
 - installing 40
- metadata 76
- ML2 networking plugin
 - bridge interface, configuring 98, 134
 - configuration options 136
 - configuring 98, 134
 - flat networks 103, 136
 - L2 population driver, using 101
 - mechanism drivers 101, 136
 - network VLAN ranges 103, 137
 - options 100
 - overlay interface, configuring 99, 134
 - security groups 105, 139
 - tenant network types 102
 - tunnel ID ranges 137
 - type drivers 101
 - VNI ranges 104, 138
- Modular Layer 2 (ML2) plugin
 - about 64
 - architecture 66
 - drivers 64
 - Mechanism driver 65
 - TypeDrivers 64
- Multi-Chassis Link Aggregation (MLAG) 23
- multihoming 194
- MySQL database server
 - configuring 39
 - installing 39

N

- network address translation (NAT)
 - about 260
 - floating IP addresses 261
- network availability zone
 - about 416
 - configuring 416
 - DHCP services, scheduling 419

- routers, scheduling 418
- network configuration
 - about 31
 - example networks 31, 34
 - interface configuration 34, 36
- Network Function Virtualization (NFV) 12, 398
- network management
 - about 148
 - flat network, creating in CLI 153, 155
 - in CLI 151, 153
 - local network, creating in CLI 158
 - network attributes, updating in CLI 160
 - network properties, displaying in CLI 159
 - network, creating via Admin panel 164
 - network, creating via project panel 162
 - networks, creating in dashboard 162
 - networks, deleting in CLI 161
 - networks, listing in CLI 158
 - provider network 149
 - tenant network 149
 - VLAN network, creating in CLI 155, 157
- network namespace 67, 68, 359
- network ports
 - managing 187, 189
 - port, creating 189
- Network Time Protocol (NTP)
 - configuring 38
 - installing 37
- Neutron components
 - agent mode, setting 320
 - configuring 319
 - distributed mode, enabling 320
 - installing 319
 - interface driver, defining 320
 - L3 agents, installing 320
 - Neutron L3, restarting 322
 - Neutron, configuring 321
- Neutron L3 agent
 - agent mode, setting 253
 - configuring 252
 - installing 252
 - interface driver, defining 252
 - metadata proxy, enabling 253
 - restarting 307
 - router management, enabling 254

- router service plugin, enabling 253
 - services, restarting 254
- Neutron LBaaS agent service
 - configuring 359
 - device driver, defining 360
 - interface driver, defining 360
- Neutron services
 - configuration, for notifying Nova 72
 - configuration, for using Keystone 70
 - configuration, for using message service 71
 - configuring 68, 72, 74
 - database, creating 68
 - installing 68
 - metadata agent, configuring 76
 - metadata agent, restarting 78
 - Neutron DHCP agent, configuring 74, 75
 - Neutron DHCP agent, restarting 76
 - neutron-server, initiating 74
 - Nova configuration 71
 - packages, installing 69
 - role, configuring in Keystone 69
 - user, configuring in Keystone 69
- Neutron
 - configuring 233, 307, 308, 361
 - network 62
 - networking elements 62
 - port 63
 - service plugin, defining 361
 - service provider, defining 361
 - subnet 63
- nodes
 - dedicated network node, using 27
 - services, separating 24
 - single controller node, using 24

O

- Open vSwitch agent
 - configuration file, updating 140
 - configuring 139
 - installing 140
 - verifying 146
- Open vSwitch Database Server (OVSDB) 112
- Open vSwitch driver
 - bridge mappings 143
 - bridge, configuring 144

- configuring 139
- database server 112
- firewall driver 144
- flat network, flow rules 126, 129
- flow rules, programming 122
- integration bridge 142
- kernel module 112
- L2 population 141
- local IP 142
- local networks, flow rules 133
- local VLANs, identifying associated with ports 121
- overlay networks, flow rules 129, 133
- ports, identifying on virtual switch 118
- return traffic 125
- services, restarting 145
- tunnel bridge 142
- tunnel types 140
- used, for visualizing traffic flow 116
- using 111, 113
- using, with DHCP configuration 145
- VLAN networks, flow rules 122, 125
- vSwitch daemon 112
- VXLAN UDP port 141
- Openstack client
 - network, creating to share 244
 - projects, creating 242
 - users, creating 242
- OpenStack Dashboard
 - Apache, reloading 59
 - API version configuration, updating 57
 - connectivity, testing 59
 - default Ubuntu theme, uninstalling 58
 - familiarizing 59
 - host, updating 57
 - installing 57
 - Keystone settings, configuring 58
 - network configuration, modifying 58
- OpenStack Networking
 - about 8
 - cURL utility, using 81
 - features 9
 - firewalling 11
 - interfacing 79
 - load balancing 11

- network functions virtualization (NFV) 12
- Neutron command-line interface, using 80
- OpenStack command-line interface, using 80
- OpenStack Python SDK, using 81
- overlay networks 14
- project/tenant network 12
- provider network 12
- resources 12
- routing 10
- switching 10
- virtual network interfaces 13
- virtual network switches 14
- virtual private network (VPN) 11
- OpenStack repository
 - configuring 36
- OpenStack
 - installing 38
- OpenvSwitch commands
 - about 114
 - base commands 114
 - ovs-appctl 115
 - ovs-dpctl 115
 - ovs-ofctl 115
 - ovs-vsctl 114
- overlay interface 99, 134
- overlay networks
 - about 14
 - Generic Network Virtualization Encapsulation (GENEVE) 16
 - Generic Router Encapsulation (GRE) 16
 - Virtual Extensible Local Area Network (VXLAN) 15

P

- patch port 113
- permissions 36
- physical infrastructure
 - API network 20
 - compute node 17
 - configuring 19
 - controller node 17
 - external network 20
 - guest network 21
 - network node 17
 - physical server connections 21

- preparing 17
- storage node 17
- physical server connections
 - about 21
 - bonding 22
 - multiple interfaces 22
 - single interface 21
- plugins
 - Modular Layer 2 (ML2) plugin 64
 - used, for extending functionality 63
- policies
 - creating, for external networks 247, 248, 249
- port security
 - disabling 232
 - disabling, for ports on network 234, 235
 - modifying, on individual port 236
- ports
 - security groups, applying to 219
- project network 12
- projects
 - RBAC policies, applying to 241
- provider bridge 116
- provider interface 98, 134
- provider label 136
- provider network 12, 149

Q

Quality of Service (QoS) 238

R

- RBAC policies
 - applying, to projects 241
 - creating 245, 246
 - viewing 246
- Role-Based Access Control (RBAC) 13, 238
- router management
 - floating IPs, associating to instance in dashboard 295
 - floating IPs, disassociating in dashboard 296
 - gateway interface, attaching to router 258
 - gateway interface, clearing 259
 - in CLI 255
 - in dashboard 291
 - interfaces, listing attached to routers 259
 - internal interfaces, attaching in dashboard 292

- internal interfaces, attaching to routers 258
- internal interfaces, deleting 259
- network topology, viewing in dashboard 294
- router attributes, displaying in CLI 257
- router attributes, updating in CLI 257
- router interfaces, working with 257
- router, creating in dashboard 291
- routers, creating in CLI 256
- routers, deleting in CLI 260
- routers, listing in CLI 256

routers

- distributed virtual routers, creating 323
- distributed virtual routers, managing 323
- distributing, across cloud 317
- Open vSwitch agent, restarting 322

S

security group rules

- about 209
- chains 222, 224, 225, 226, 227
- creating, in CLI 216
- deleting, in CLI 217
- details, displaying in CLI 218
- implementing 220, 221
- listing, in CLI 218
- managing, in dashboard 229

security groups

- about 208, 209
- applying, to instances 218, 231
- applying, to ports 218
- creating, in CLI 214
- creating, in dashboard 228
- deleting, in CLI 215
- details, displaying in CLI 215
- in dashboard 227
- listing, in CLI 215
- managing, in CLI 213
- removing, from instances 219
- removing, from ports 219
- updating, in CLI 216
- working with 213
- service plugin 63
- service
 - separating, across nodes 24
- single controller node

- using 24
- subnet management
 - about 165
 - default subnet pool, assigning 185, 186
 - IPv4 addresses, working with 166
 - IPv6 addresses, working with 167
 - subnet pool, creating 182
 - subnet pool, deleting 185
 - subnet pools, managing 182
 - subnet properties, displaying in CLI 172
 - subnet, creating from pool 184
 - subnet, updating in CLI 173, 174
 - subnets, creating in CLI 167, 170
 - subnets, creating via Admin tab 178, 182
 - subnets, creating via Project tab 175, 178
 - subnets, listing in CLI 172
- system requisites
 - about 30
 - operating system requisites 30
- system
 - rebooting 38
 - upgrading 37

T

- tap interface 13, 85
- tenant network 149
- timers, VRRP
 - advertisement interval timer 301
 - preemption delay timer 302
- topology
 - reviewing 326
- traffic flow
 - default NAT behavior, observing 283
 - demonstrating, to internet 265
 - external provider network, creating 267
 - floating IPs, assigning 284
 - floating IPs, associating 287
 - floating IPs, reassigning 288
 - foundation, setting 265
 - gateway connectivity, testing 273
 - instance connectivity, verifying 279, 282
 - instances, creating 277
 - internal network, creating 273
 - L3 agent, identifying 270, 273
 - namespace, identifying 270, 273

- Neutron router, creating 269
- router, attaching to external network 269
- router, attaching to internal network 276
- traffic
 - routing, in cloud 251

U

- Ubuntu
 - URL 49

V

- veth interface 118
- Virtual Extensible Local Area Network (VXLAN)
 - about 15, 65, 94
 - potential issues 95
- virtual function (VF) 12
- virtual network interface (VNI) 65, 142
- virtual private network (VPN) 11
- Virtual Router Redundancy Protocol (VRRP)
 - about 298
 - groups 299, 300
 - master router, determining 305
 - priority 300
 - timers 301
 - virtual IP 303
 - working mode 300
- virtual switch 10
- virtual tunnel endpoint 132
- VLAN networks 65, 87
- VLAN-aware VMs
 - about 397
 - instance, booting with trunk 406
 - instance, configuring 408, 410
 - network plumb, reviewing 410, 411
 - parent port, creating 403
 - sub-port, creating 404
 - trunk details, displaying in CLI 402
 - trunk plugin, configuring 398
 - trunk, building 403
 - trunk, creating 406
 - trunk, updating in CLI 403
 - trunks, creating in CLI 401
 - trunks, deleting in CLI 402
 - trunks, listing in CLI 402
 - trunks, managing in CLI 401, 402

workflow, defining 399, 401
VXLAN Network Identifier (VNI) 86

W

working mode, VRRP
non-preemptive 301
preemptive 301