

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.

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:

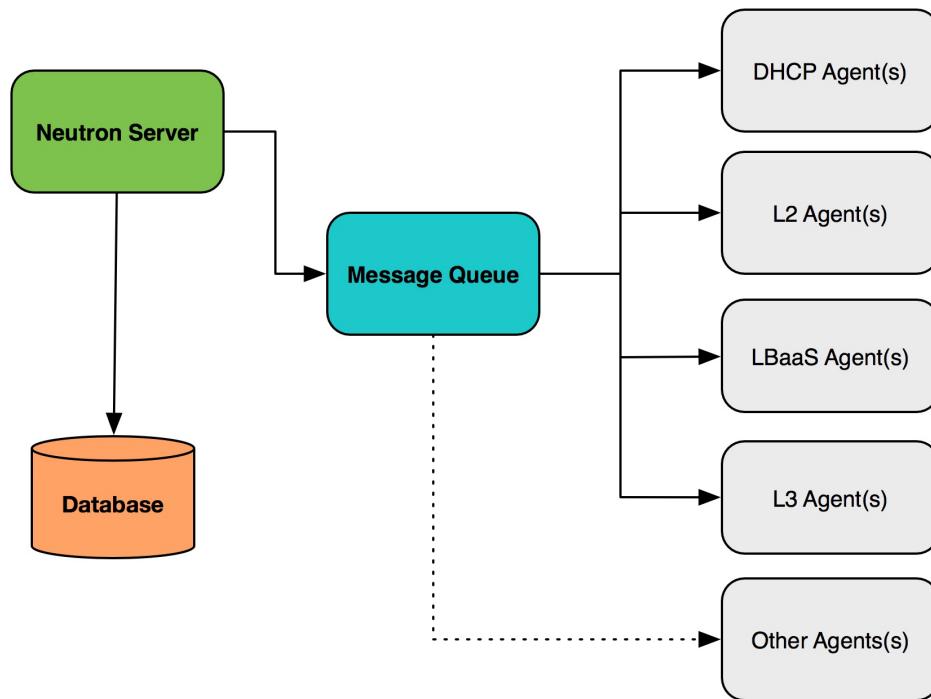


Figure 1.1

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:

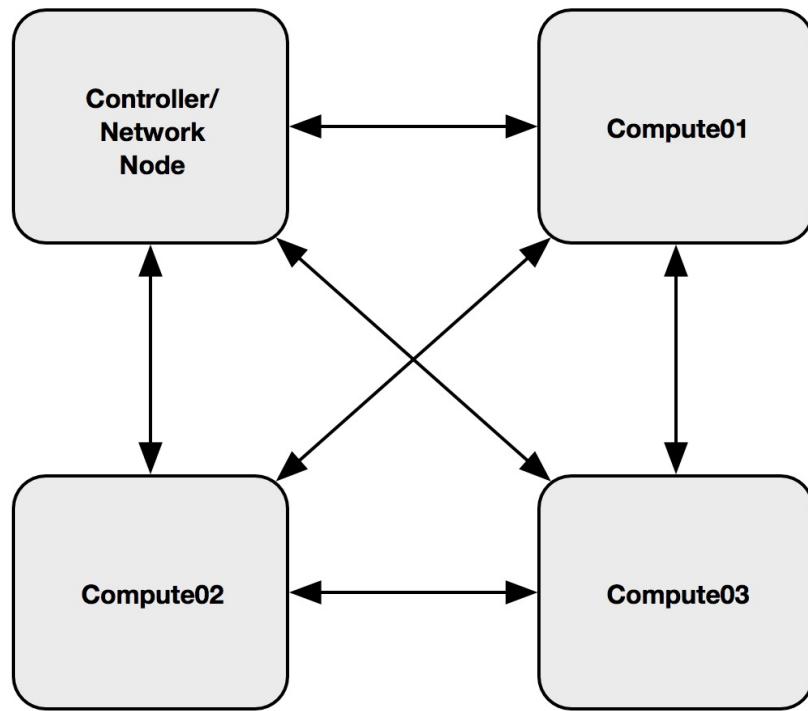


Figure 1.2

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:

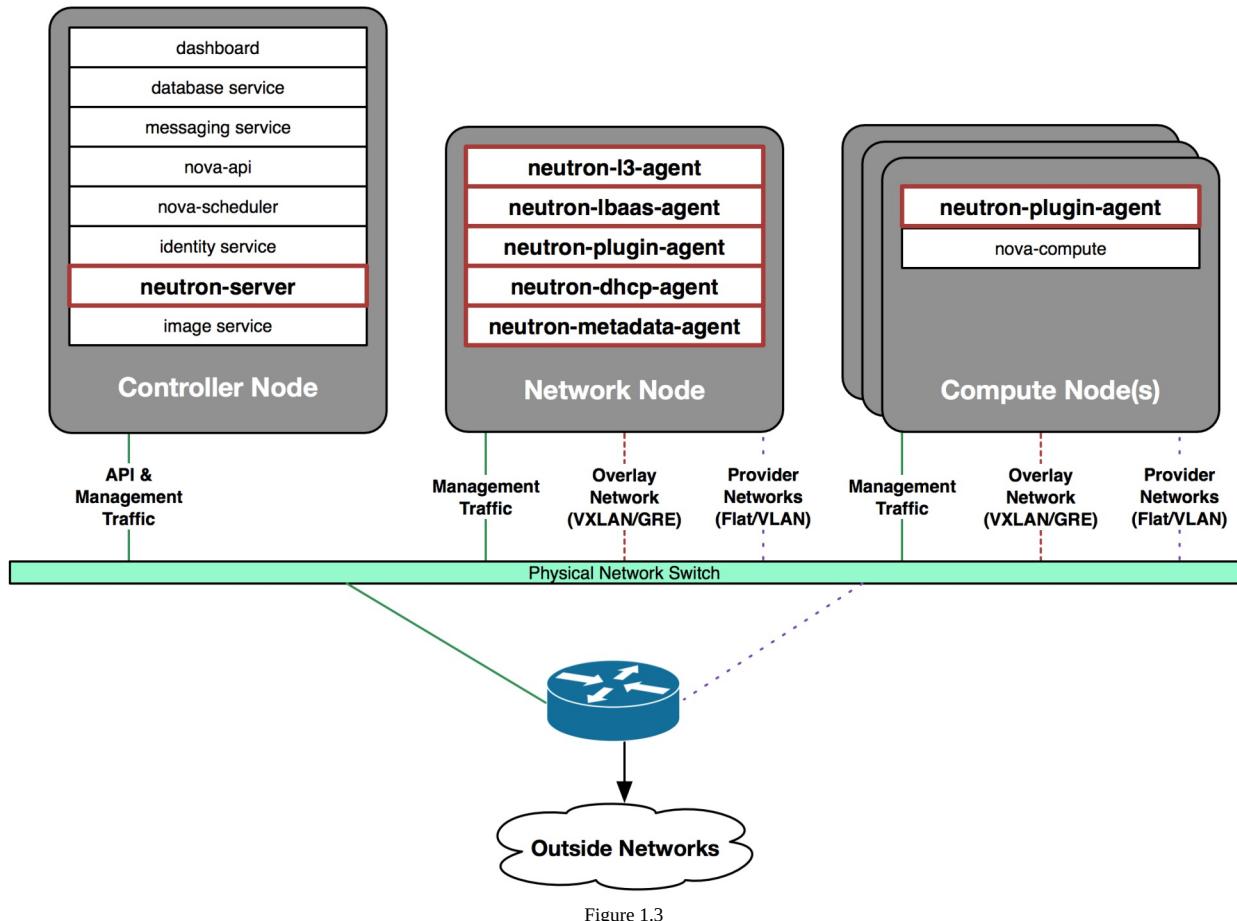


Figure 1.3

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
	Used to tunnel overlay (VXLAN, GRE,		

Overlay network	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:

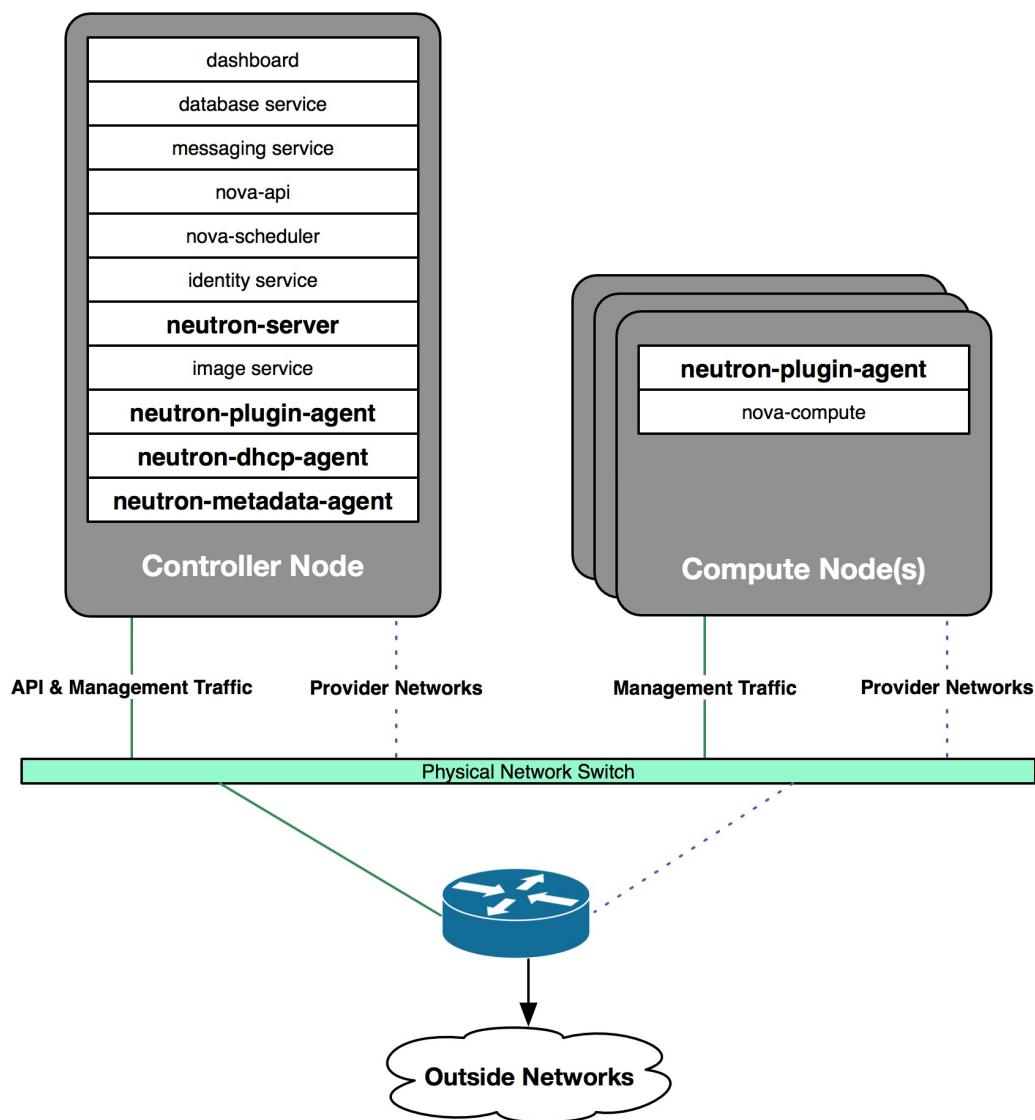


Figure 1.3

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:

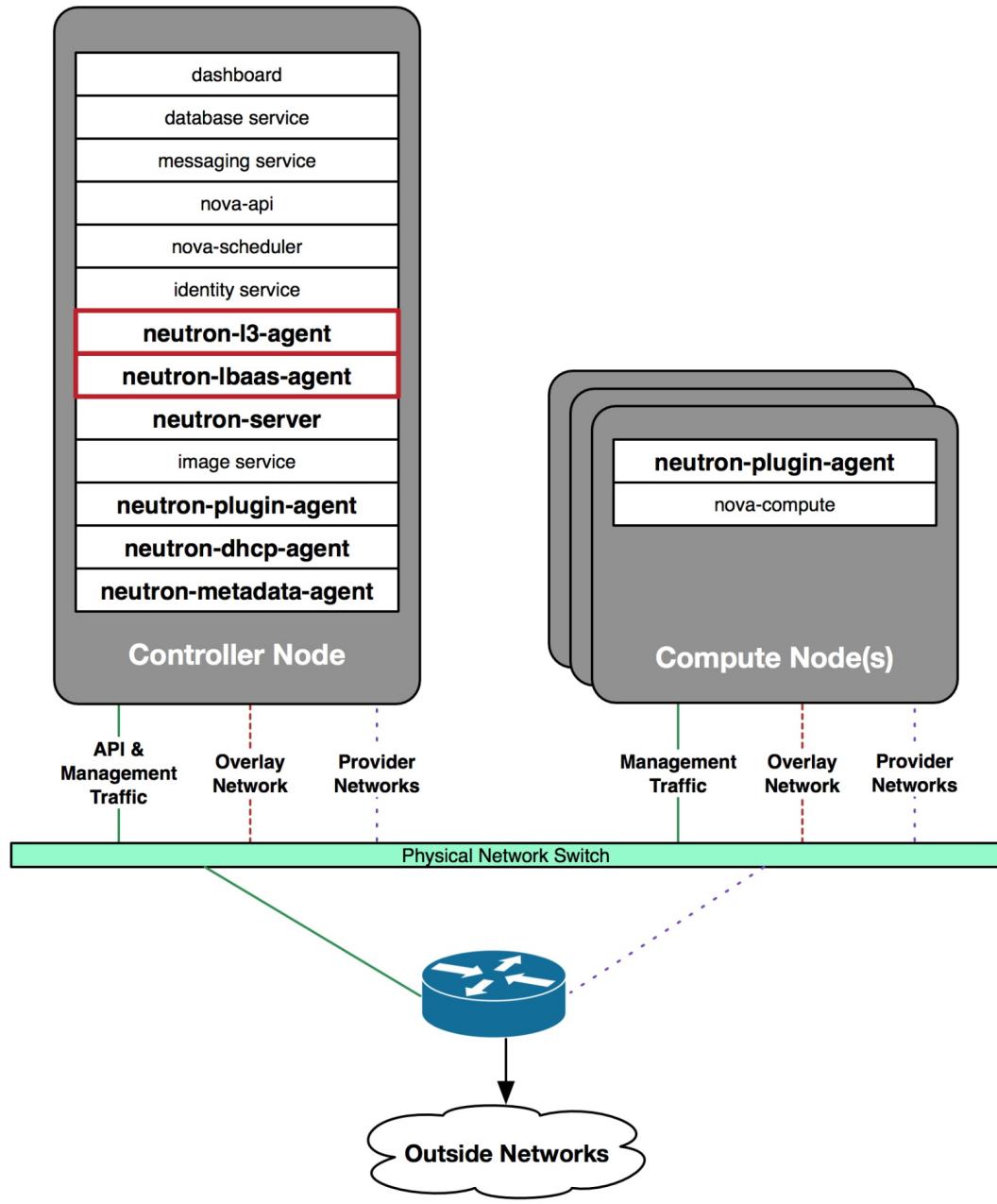


Figure 1.4

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:

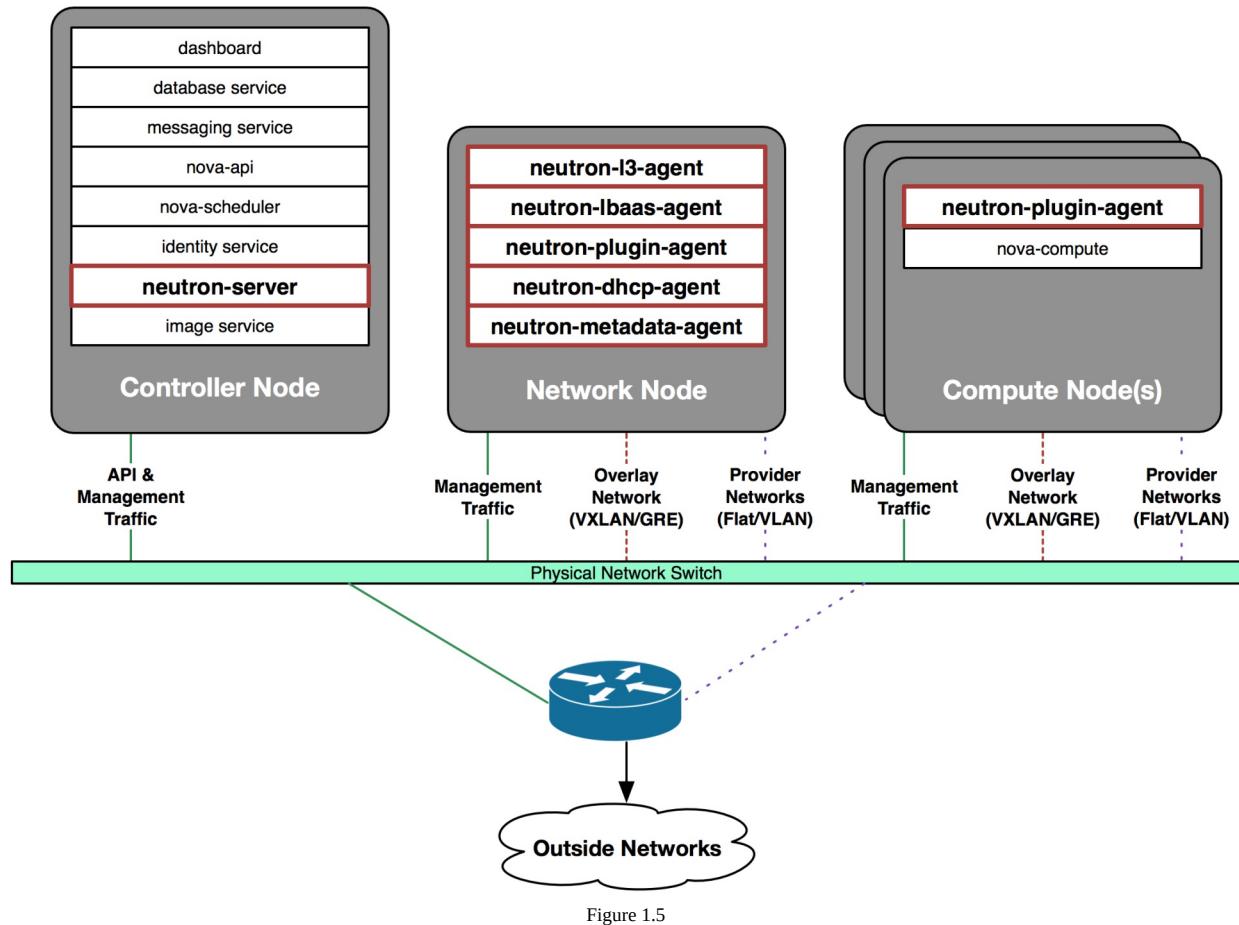


Figure 1.5

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*](#).

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	<p>Processor: 64-bit x86</p> <p>CPU count: 2-4</p> <p>Memory: 4+ GB RAM</p> <p>Disk space: 32+ GB</p> <p>Network: 3x 1-Gbps network interface cards</p>

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:

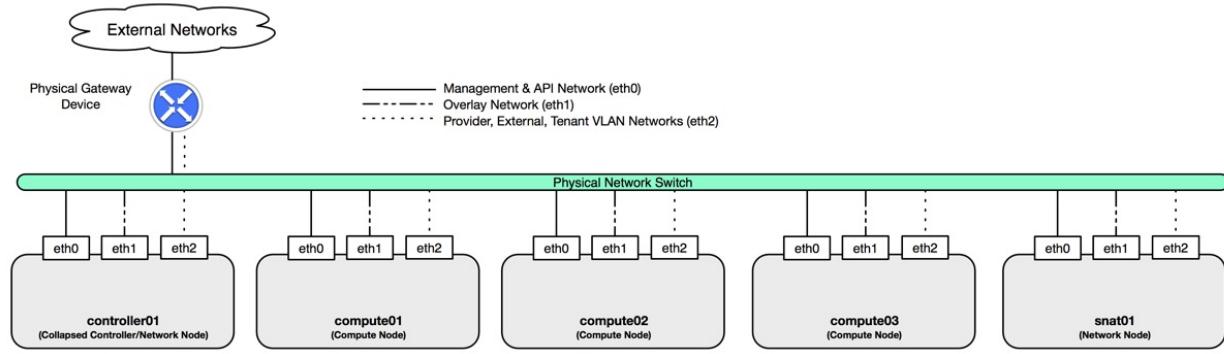


Figure 2.1

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)
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` 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/ |
| bfbdb2527b3b44d98b2214fc32d1cad2f | 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-aac3-7031dfc03fc0 | 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
```

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 `cell0` database using the `nova-manage` utility:

```
| # su -s /bin/sh -c "nova-manage cell_v2 map_cell0" nova
```

Create the `cell1` cell:

```
| # su -s /bin/sh -c "nova-manage cell_v2 create_cell
--name=cell1 --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:

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_vpnservice': 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

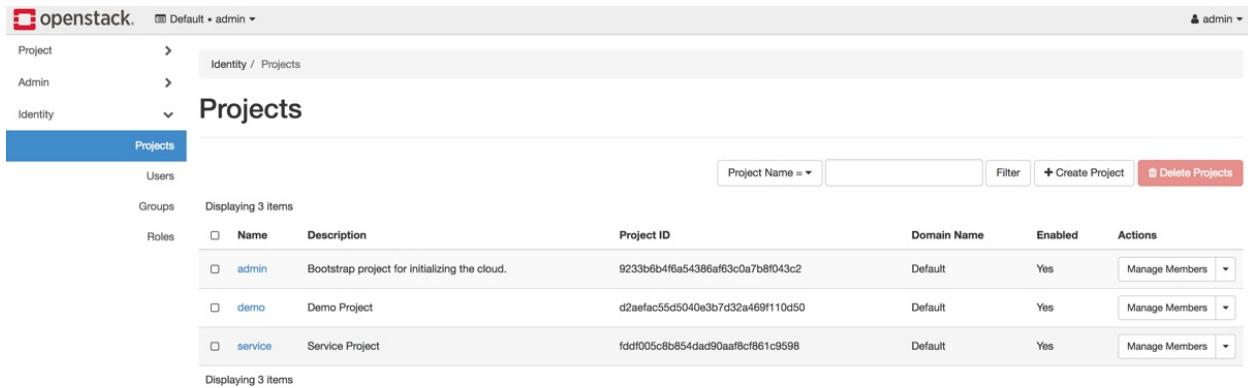
From a machine that has access to the management network of the `controller01` node, open the following URL in a web browser: <http://controller01/horizon/>.

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:

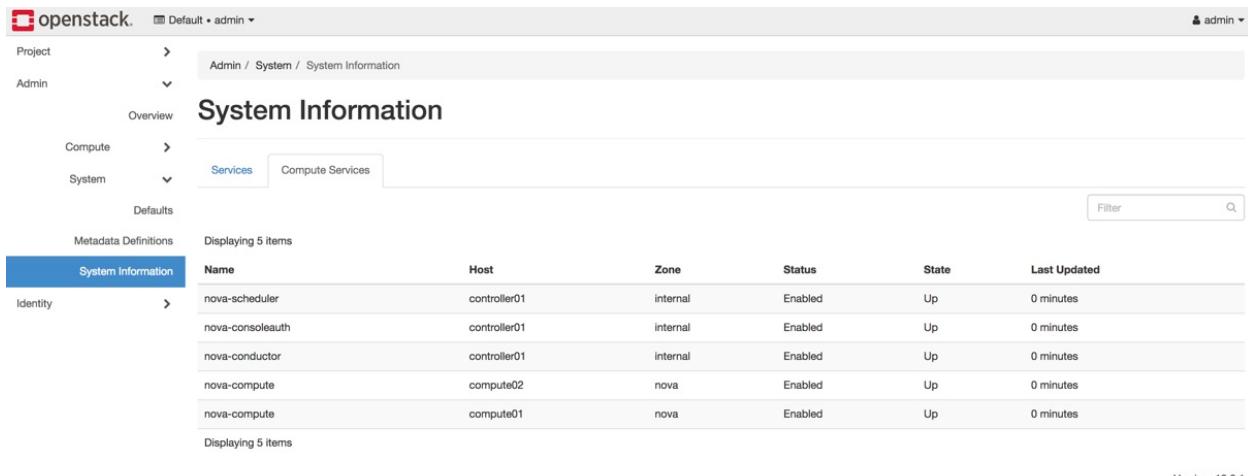


The screenshot shows the OpenStack Dashboard with the URL `openstack`. The top navigation bar shows the user is logged in as `admin`. The left sidebar has a tree structure: Project (selected), Admin, Identity. Under Identity, there are sub-options: Projects (selected), Users, Groups, Roles. The main content area is titled "Projects" and displays a table of three items:

Name	Description	Project ID	Domain Name	Enabled	Actions
admin	Bootstrap project for initializing the cloud.	9233b6b4f6a54386af63c0a7b8f043c2	Default	Yes	<button>Manage Members</button>
demo	Demo Project	d2aefac55d5040e3b7d32a469f110d50	Default	Yes	<button>Manage Members</button>
service	Service Project	fddff005c8b854dad90aaf8cf861c9598	Default	Yes	<button>Manage Members</button>

Below the table, it says "Displaying 3 items".

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 Dashboard with the URL `openstack`. The top navigation bar shows the user is logged in as `admin`. The left sidebar has a tree structure: Project, Admin (selected), System (selected). Under System, there are sub-options: Overview (selected), Compute, System, Defaults, Metadata Definitions. The main content area is titled "System Information" and shows two tabs: Services (selected) and Compute Services. Below the tabs, it says "Displaying 5 items".

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

Below the table, it says "Displaying 5 items".

Version: 12.0.1

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.

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:

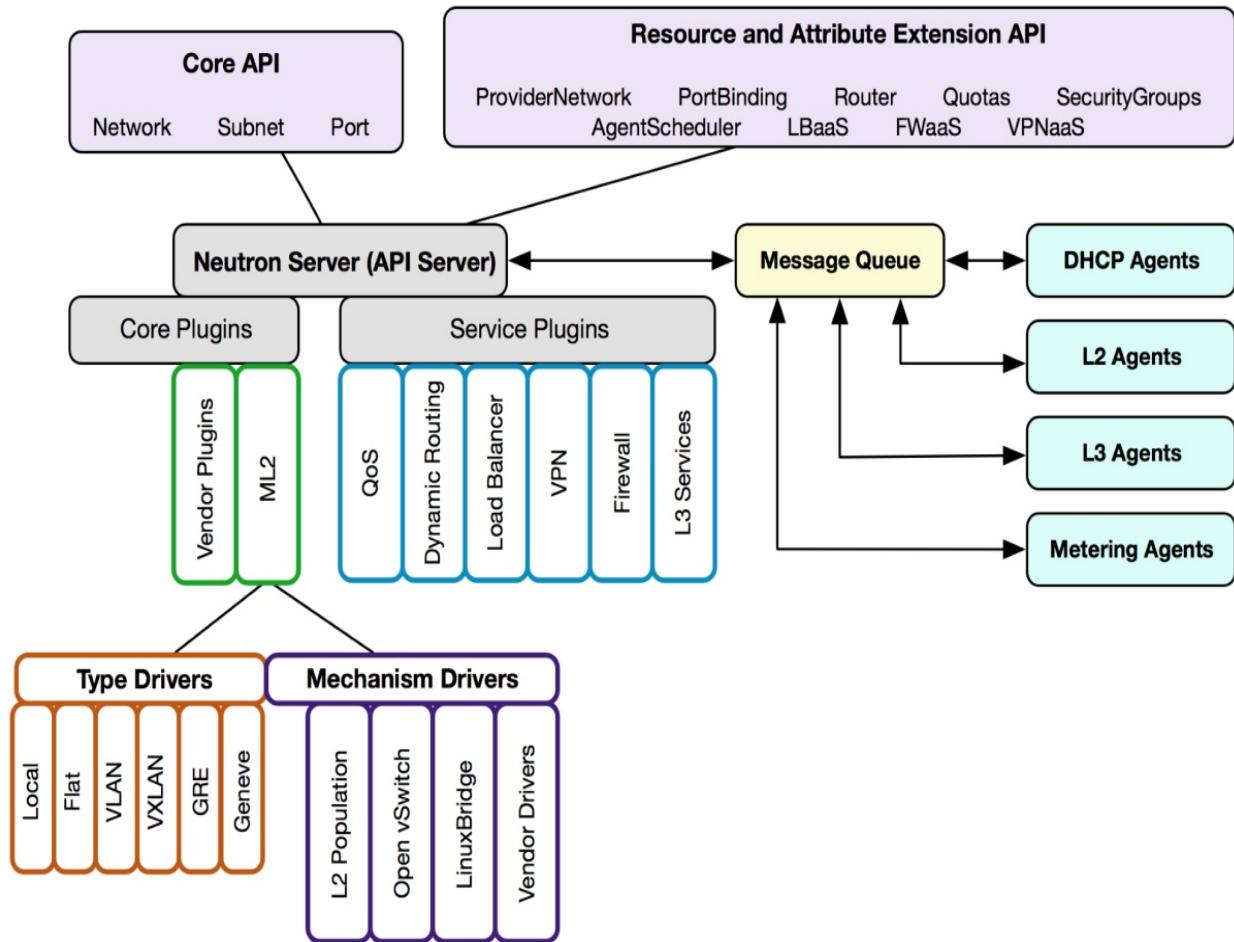


Figure 3.1

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 `qrouting` 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 `qrouting` 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.OVSIInterfaceDriver`

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 |  
+-----+-----+-----+-----+-----+-----+
```

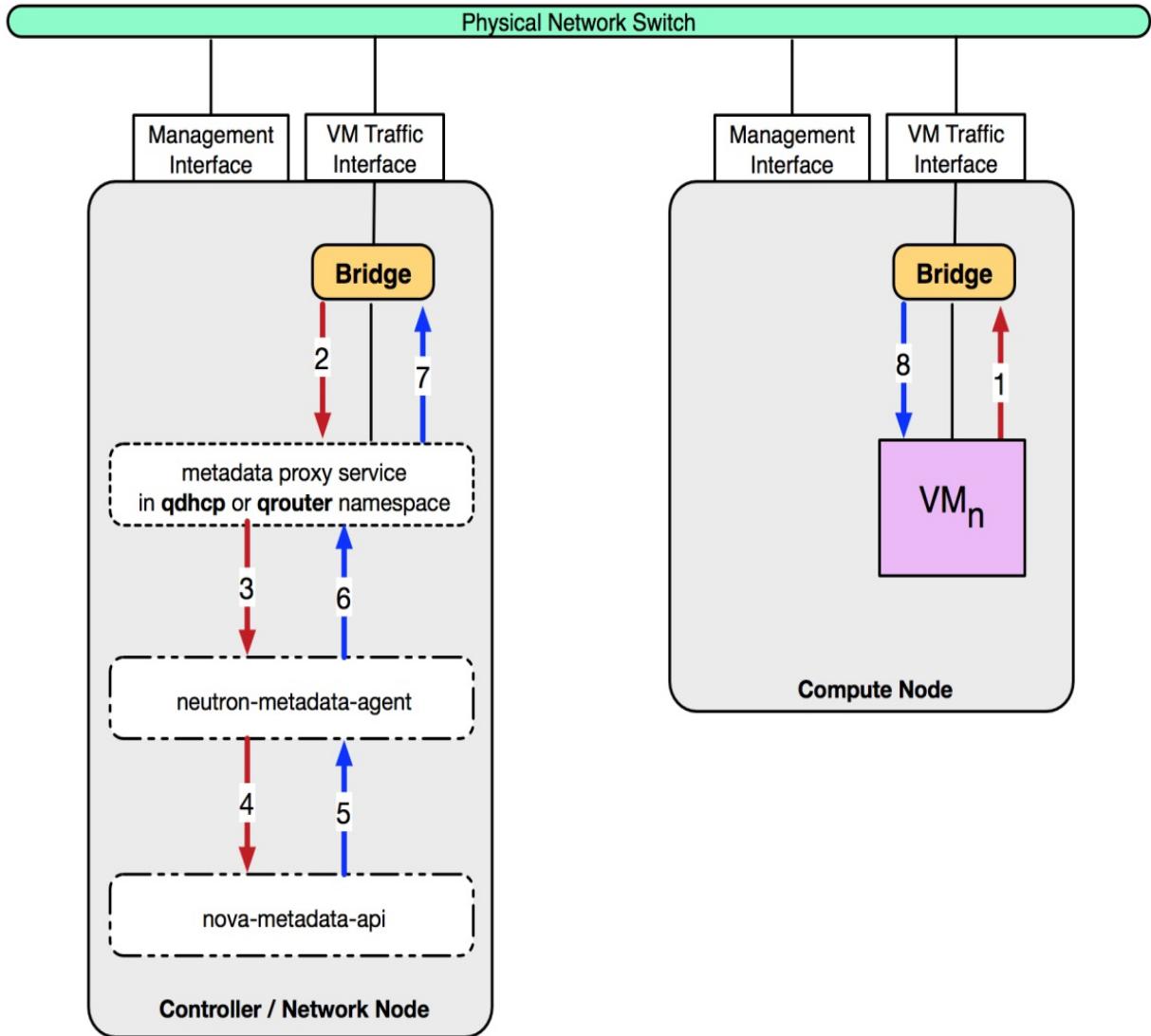
A smiley face under the `Alive` column means that the agent is properly communicating with the `neutron-server` service.

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 `169.254.269.254` 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 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: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-metadata)  
     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  
           └─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 parseable 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         | gAAAAABafFH8TfwDaM8NoFFXuS3sEBm3NnU6a5f1kRGSoAxAk9v1Aj-SEEUT9su8rwQvJ7jLu5LcX_8rdf0iuPo_WhcjL5lQd-hVo4GGLTC2snqASwxf-
|          | h7DHkgtmLUjsnIO2P0tPrD9tCL6fdCxnDzGRoq4sGeY4FJqhYD55P0cDGBPMjHQU
| project_id | 9233b6b4f6a54386af63c0a7b8f043c2
| user_id    | 8a679d8b7b574e54a5cf02c422bbe68
+-----+
```



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.

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:

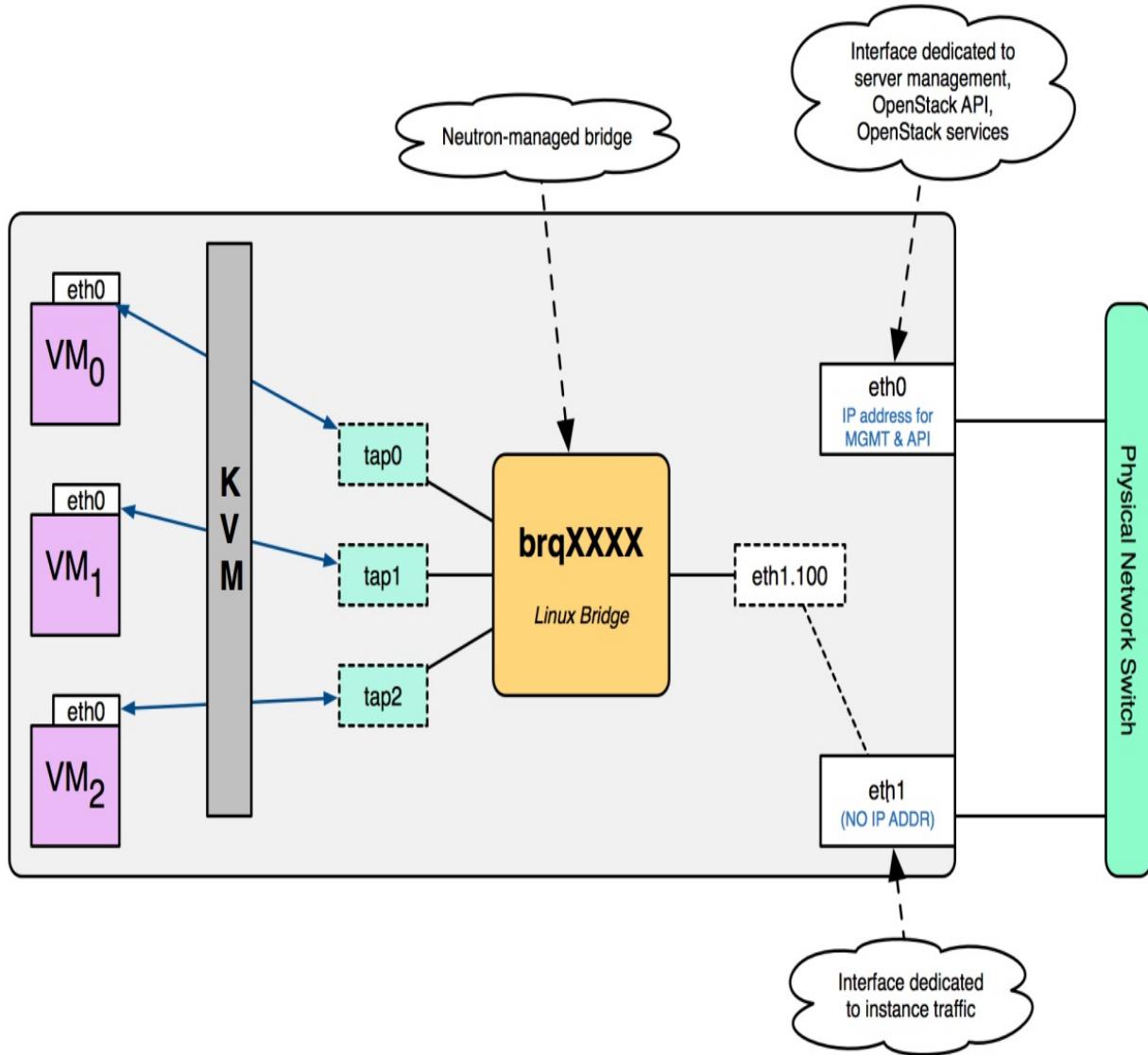


Figure 4.1

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:

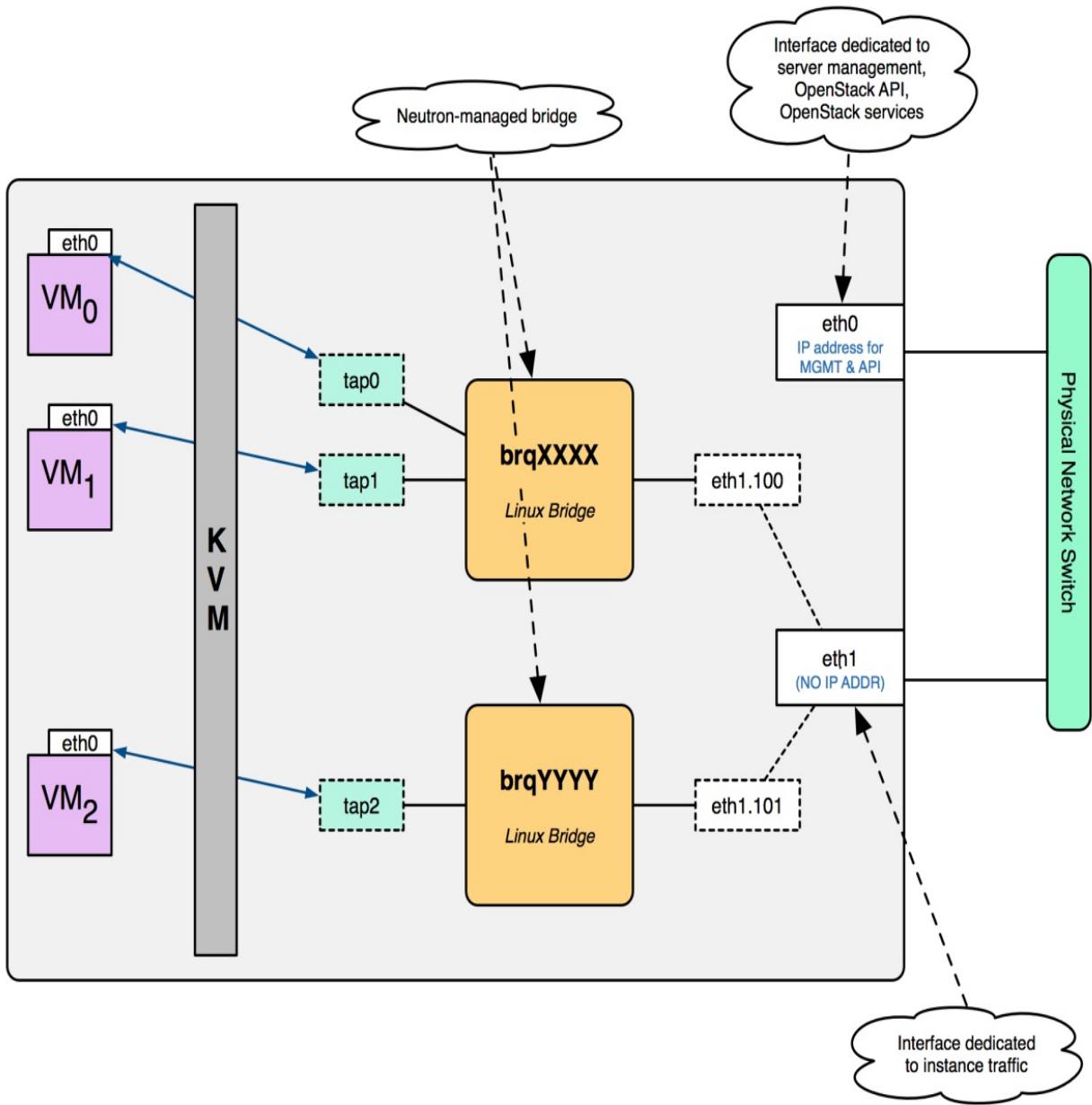


Figure 4.2

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:

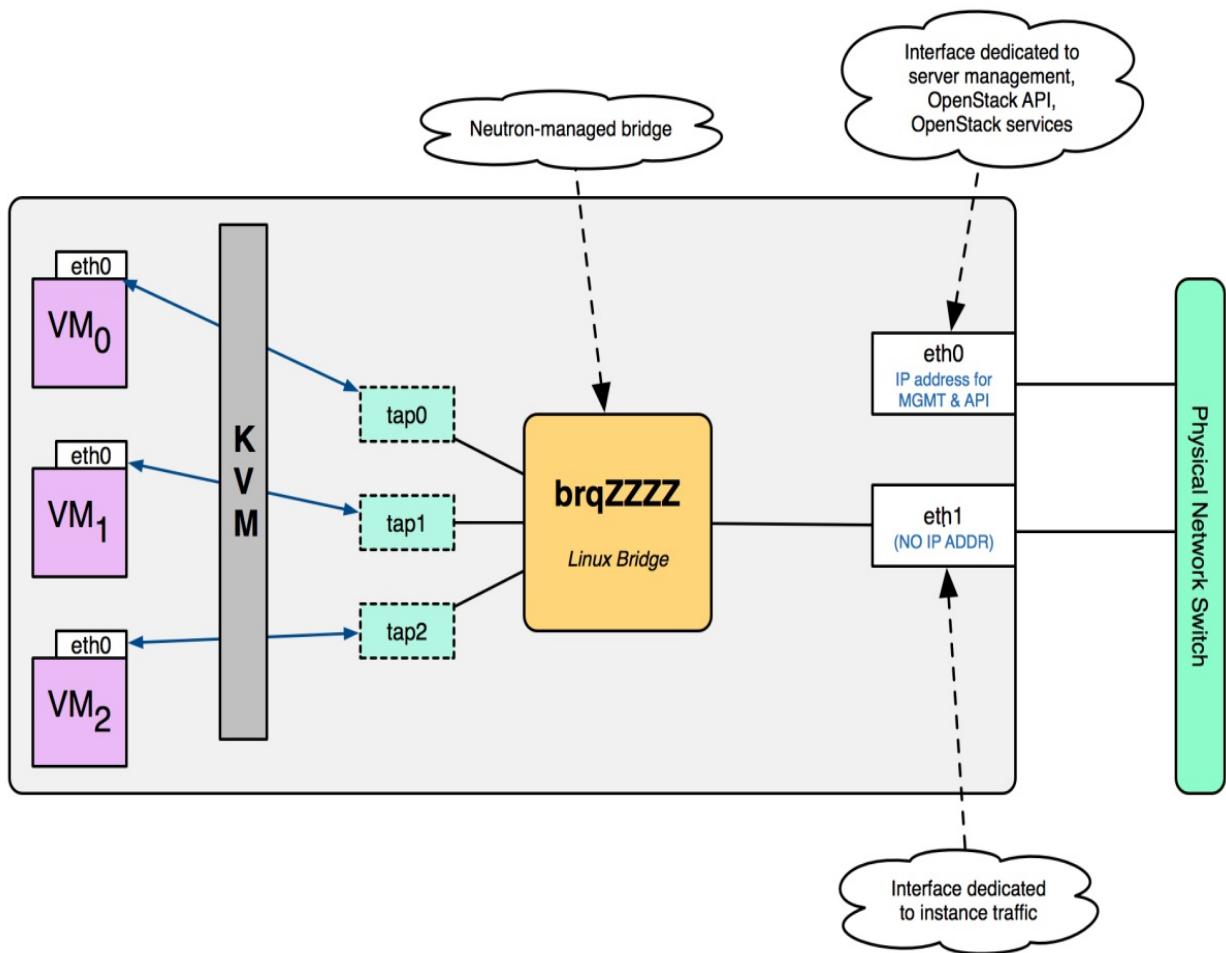


Figure 4.3

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
brqZZZZ          <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:

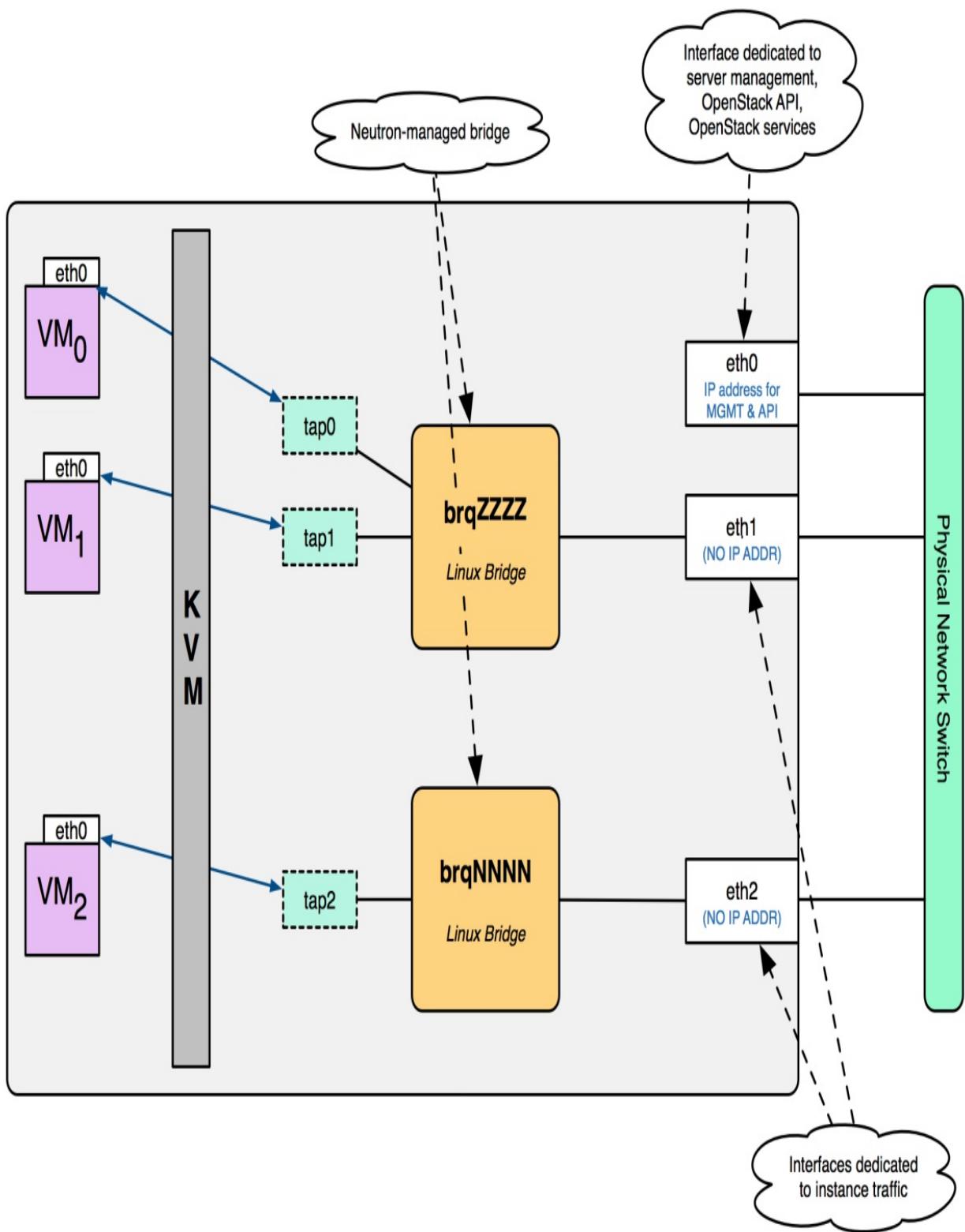


Figure 4.4

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

bridge name      bridge id      STP enabled      interfaces
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:

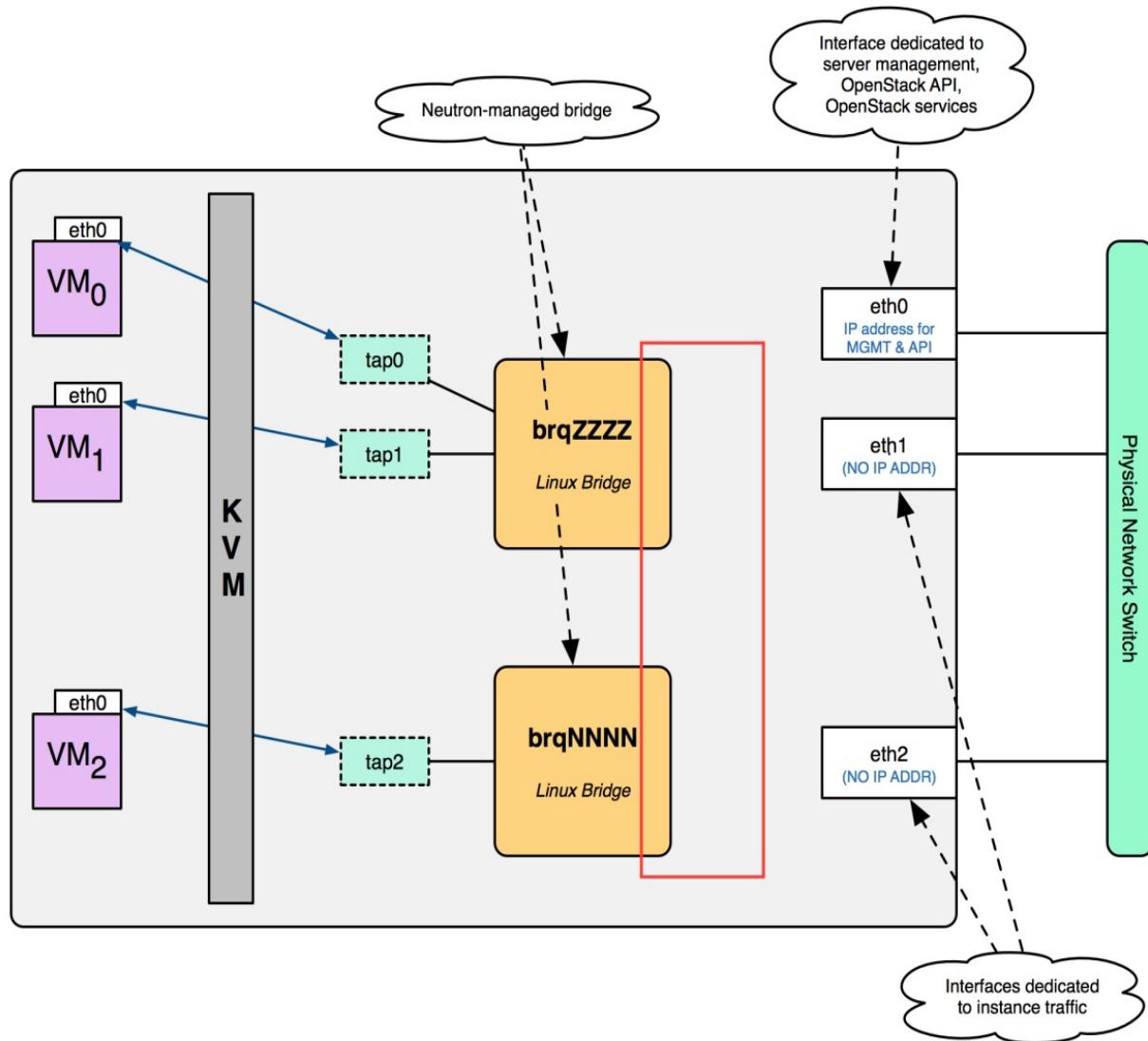


Figure 4.5

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>	10.20.0.100
<code>compute01</code>	10.20.0.101

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 and 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 `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:

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
dcc2ba98c-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.

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:

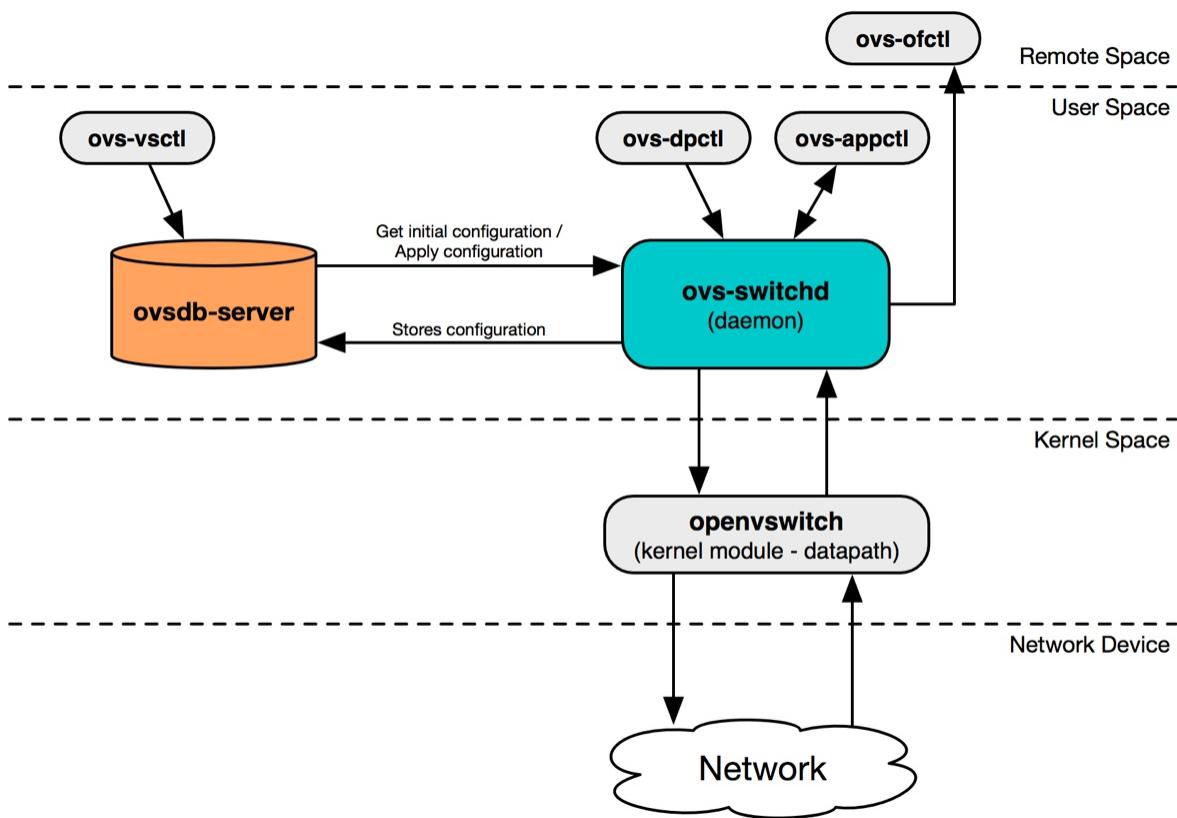
- Local
- Flat
- VLAN
- VXLAN
- GRE

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-vctl` 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:

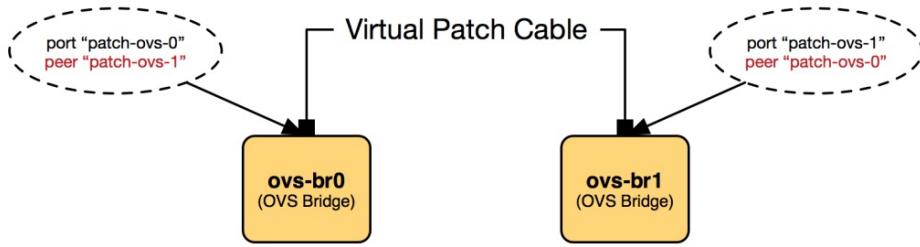


Figure 5.1

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:

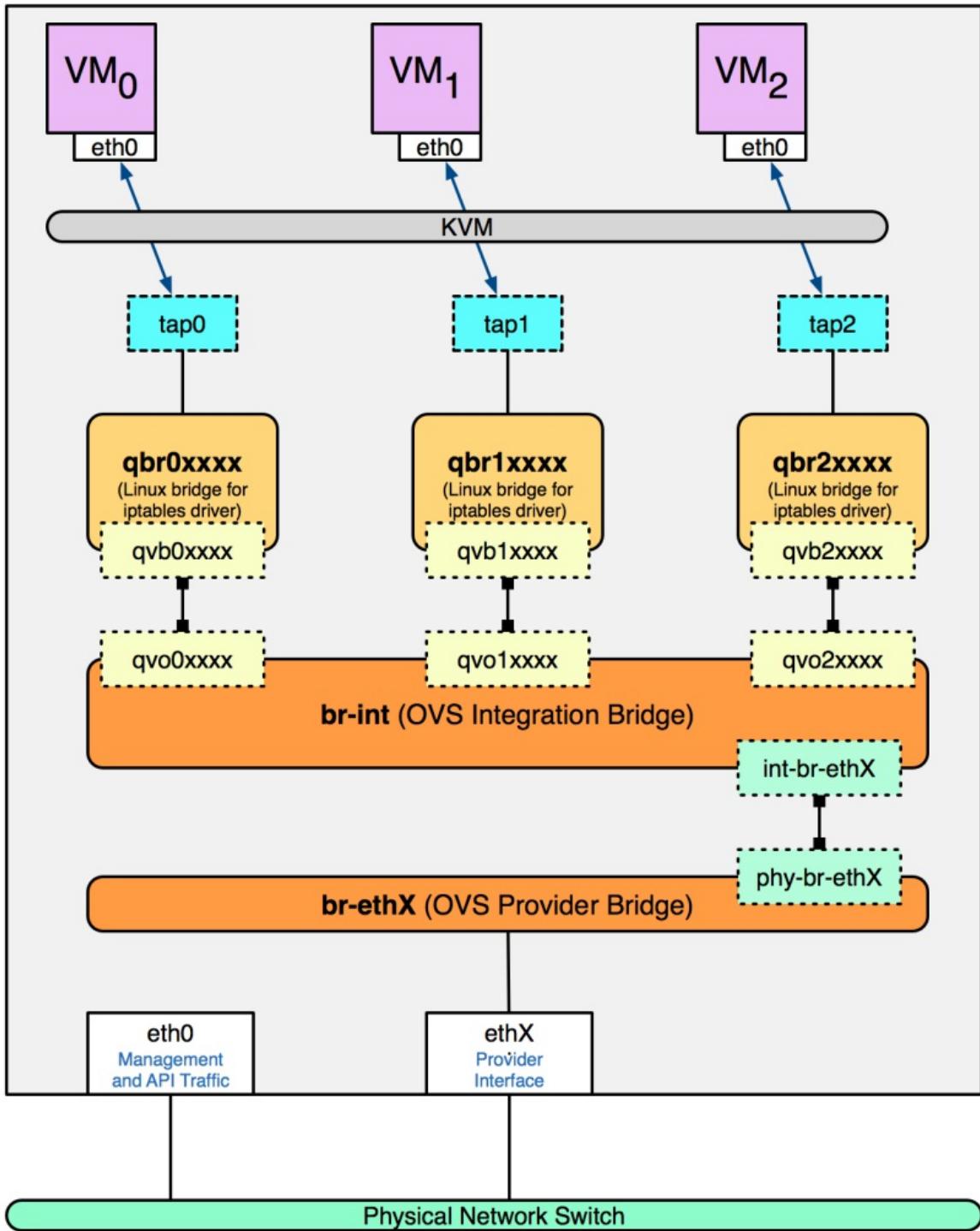


Figure 5.2

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:

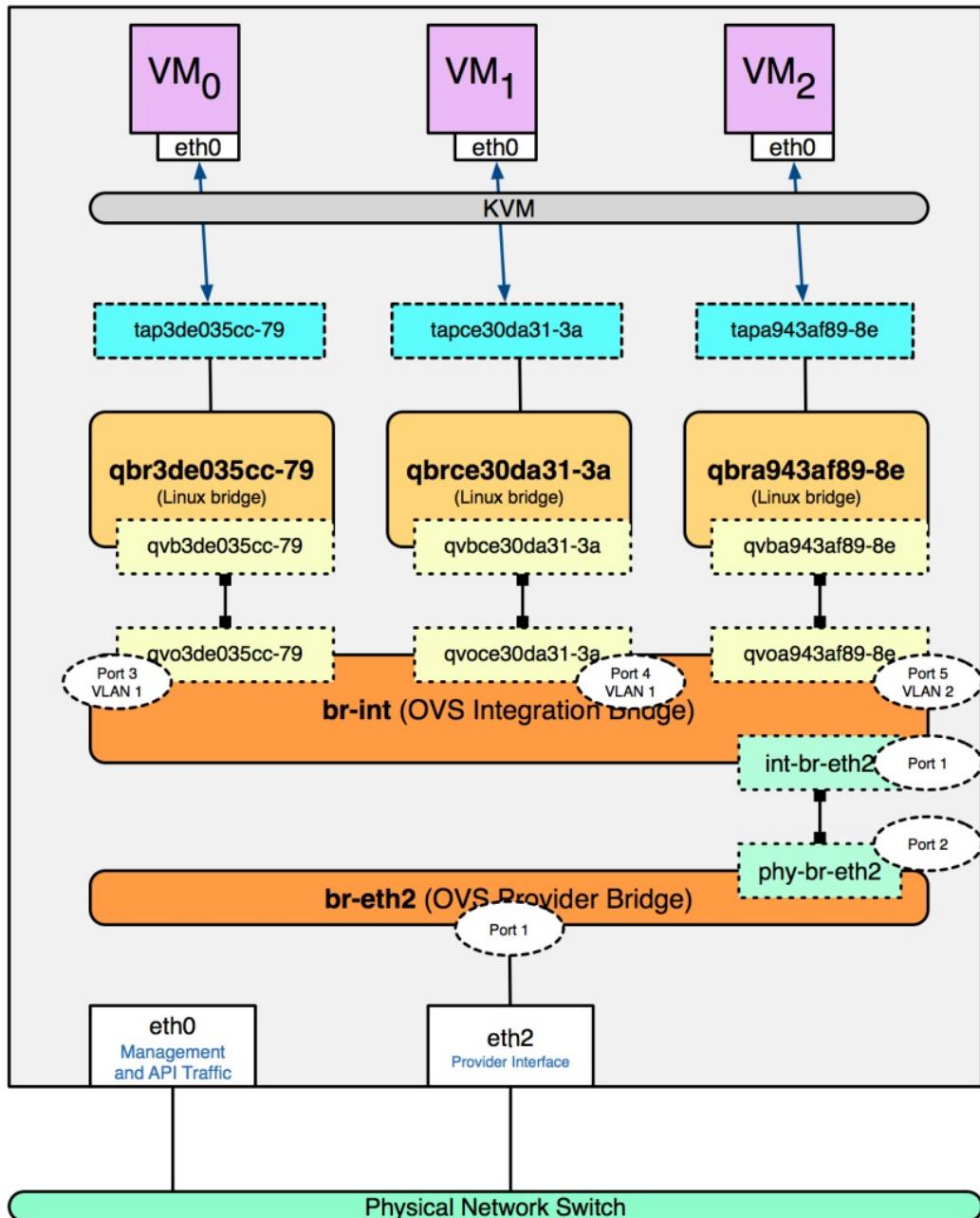


Figure 5.3

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=630, n_bytes=0, 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=0 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=f80::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=f80::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=f80::f816:3eff:fedc: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,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:de:9e:dc 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
...


i 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	vxlan
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 --rsort
table=20, n_packets=0, n_bytes=0, priority=2,dl_vlan=4,dl_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,dl_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:0XM_OF_IN_PORT[],output:"patch-int"
table=21, n_packets=0, n_bytes=0, priority=1,arp,dl_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_dl_src:fa:16:3e:f1:b0:49,IN_PORT
table=22, n_packets=0, n_bytes=0, priority=1,dl_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,dl_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,dl_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 `vxlan0a140064`, 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 `[m12_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:

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
69eed18-4079-45ba-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 `8021q` 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.

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

network set	Sets network properties
network unset	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-allocations.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:

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.

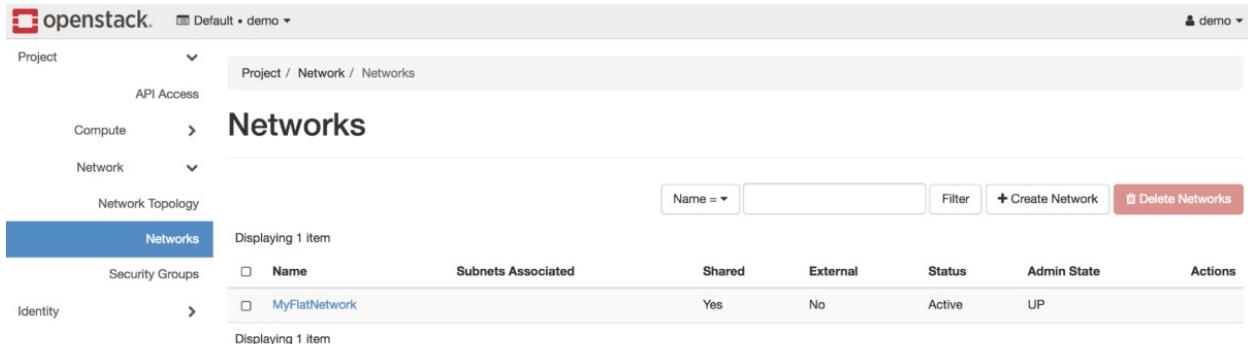
Creating networks in the dashboard

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:

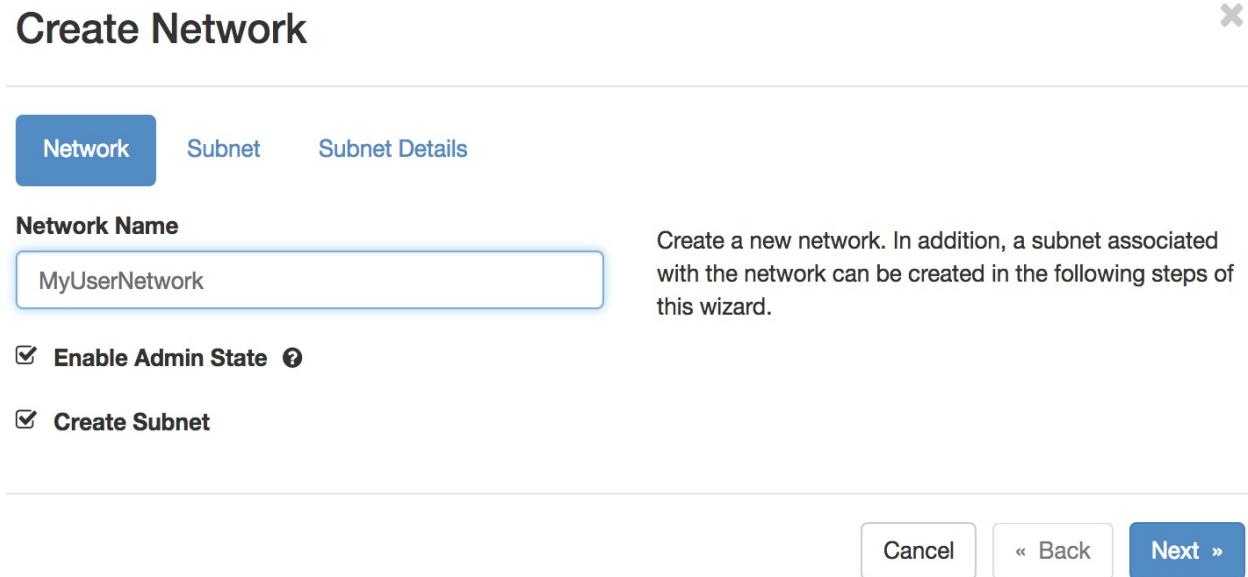
1. Navigate to Project | Network | Networks:



The screenshot shows the OpenStack Project panel with the URL [openstack](#) and the project set to "demo". The navigation bar includes "Project", "API Access", "Compute", "Network", "Network Topology", "Security Groups", and "Identity". The "Network" tab is selected. The main content area is titled "Networks" and displays a table with one item: "MyFlatNetwork". The table columns are "Name", "Subnets Associated", "Shared", "External", "Status", "Admin State", and "Actions". The "Actions" column contains a small icon. Below the table, it says "Displaying 1 item".

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 the "Create Network" wizard. The title bar has a close button. The tabs at the top are "Network" (selected), "Subnet", and "Subnet Details".
Network Name: The input field contains "MyUserNetwork".
Enable Admin State: The checkbox is checked.
Create Subnet: The checkbox is checked.
A descriptive text on the right says: "Create a new network. In addition, a subnet associated with the network can be created in the following steps of this wizard."
At the bottom 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:

Create Network

Network

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

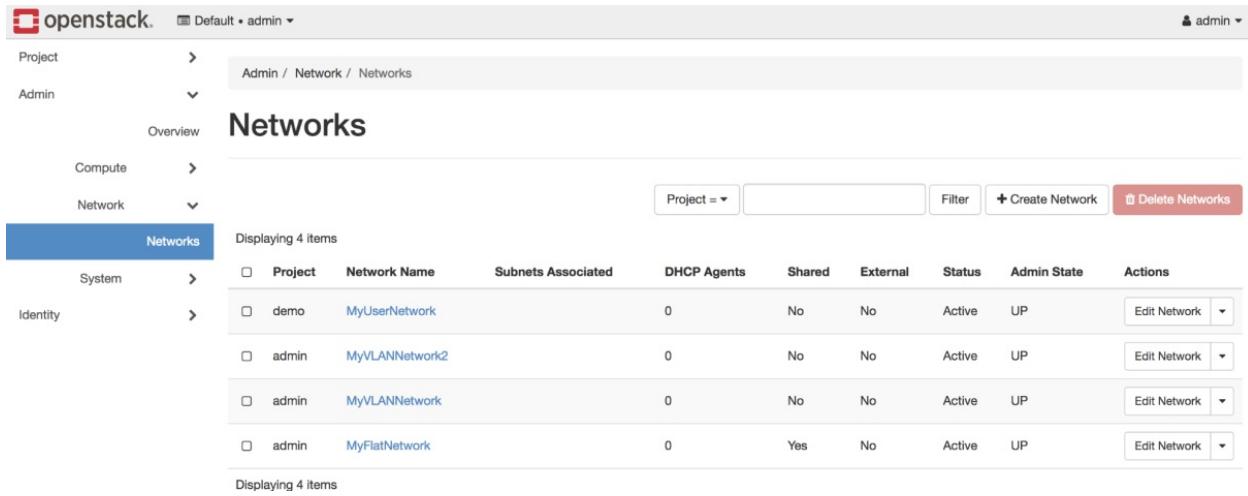
Cancel **« Back** **Create**

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:



The screenshot shows the OpenStack Admin dashboard with the following details:

- Header:** openstack. Default • admin. admin ▾
- Sidebar (Project Admin):** Project, Admin, Overview, Compute, Network, Networks (selected).
- Breadcrumbs:** Admin / Network / Networks
- Title:** Networks
- Search and Filter:** Project = ▾, Filter, + Create Network, Delete Networks
- Data Table:** Displays 4 items.
- Table Headers:** Project, Network Name, Subnets Associated, DHCP Agents, Shared, External, Status, Admin State, Actions.
- Table Data:**

Project	Network Name	Subnets Associated	DHCP Agents	Shared	External	Status	Admin State	Actions
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		0	Yes	No	Active	UP	Edit Network ▾
- Footer:** Displaying 4 items

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

Create a new network. In addition, a subnet associated with the network can be created in the following steps of this wizard.

Project *

Provider Network Type *

Enable Admin State

Shared

External Network

Create Subnet

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)

<code>subnet pool list</code>	Lists subnet pools
<code>subnet pool set</code>	Sets subnet pool properties
<code>subnet pool show</code>	Displays subnet pool details
<code>subnet pool unset</code>	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.html>. 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:

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.

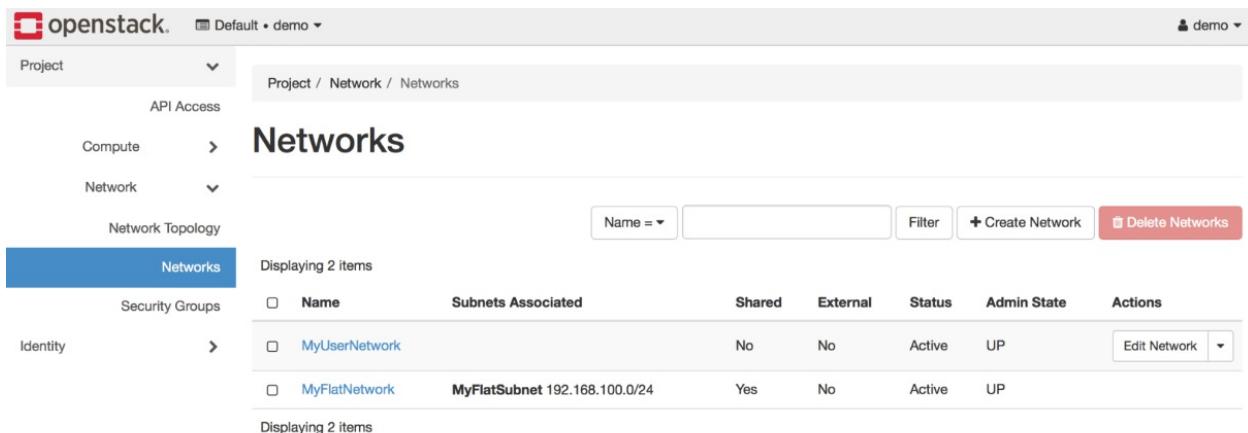
Creating subnets in the dashboard

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

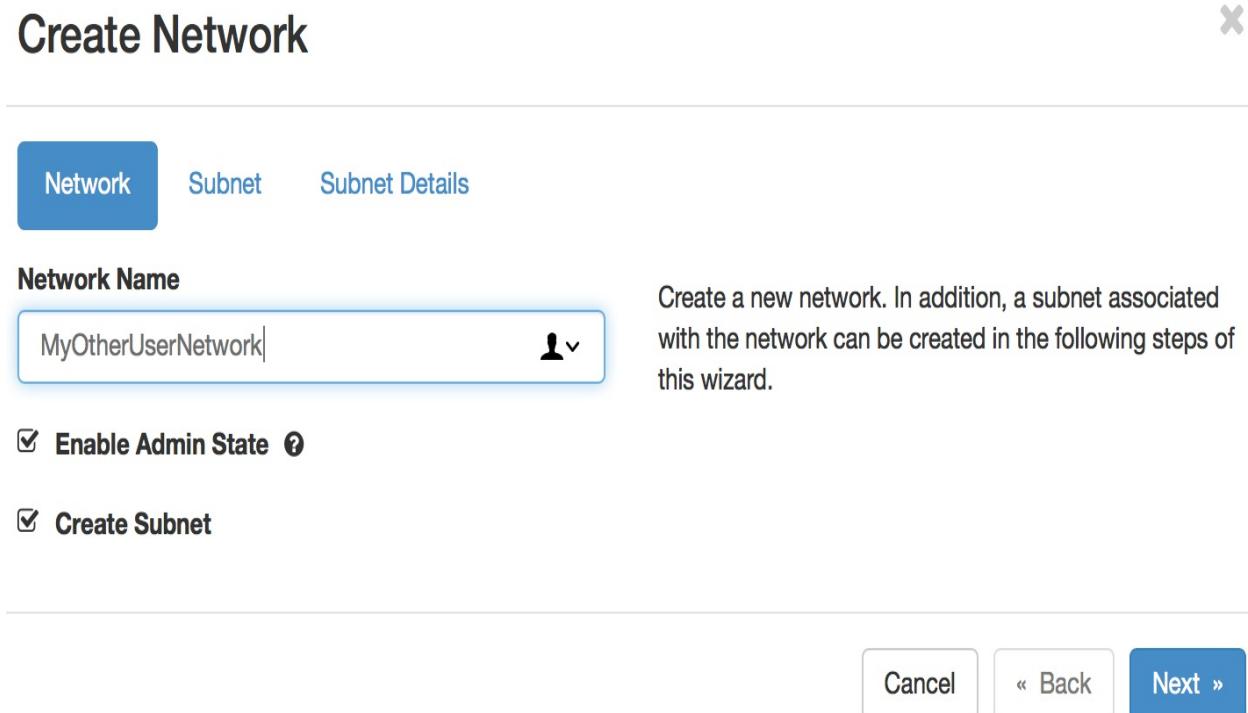
Users can create subnets at the time of network creation using a wizard located within the Project tab in the dashboard. They can also add new subnets to existing networks. To create a network and subnet, login as the user in the demo project and perform the following steps:

1. Navigate to Project | Network | Networks and click on the Create Network button:



The screenshot shows the OpenStack dashboard with the 'demo' project selected. The navigation bar includes 'Project / Network / Networks'. The main content area is titled 'Networks' and displays two items: 'MyUserNetwork' and 'MyFlatNetwork'. 'MyUserNetwork' is associated with 'MyFlatSubnet 192.168.100.0/24'. A red 'Create Network' button is visible at the top right of the table.

2. Clicking on Create Network will open a window where you can specify the network and subnet details:



The screenshot shows the 'Create Network' wizard. The tabs at the top are 'Network' (selected), 'Subnet', and 'Subnet Details'. The 'Network Name' field contains 'MyOtherUserNetwork'. A note to the right 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 network name, there are two checked checkboxes: 'Enable Admin State' and 'Create Subnet'. At the bottom are 'Cancel', '« Back', and 'Next »' buttons.

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

X

Network Subnet Subnet Details

Subnet Name
MyOtherUserSubnet

Network Address ⓘ
192.168.204.0/24

Gateway IP ⓘ
192.168.204.1

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](#)

[« Back](#)

[Create](#)

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:

Networks

Name = ▾ Filter [Create Network](#) [Delete Networks](#)

Displaying 3 items

<input type="checkbox"/>	Name	Subnets Associated	Shared	External	Status	Admin State	Actions
<input type="checkbox"/>	MyOtherUserNetwork	MyOtherUserSubnet 192.168.204.0/24	No	No	Active	UP	 Edit Network ▾
<input type="checkbox"/>	MyUserNetwork		No	No	Active	UP	 Edit Network ▾
<input type="checkbox"/>	MyFlatNetwork	MyFlatSubnet 192.168.100.0/24	Yes	No	Active	UP	Create Subnet Delete Network

Displaying 3 items

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:

The screenshot shows the OpenStack Admin interface. The top navigation bar includes the OpenStack logo, a dropdown for 'Default * admin', and a user icon. The left sidebar has sections for Project, Admin (with Overview, Compute, and Network), and Networks (which is selected). The main content area is titled 'Networks' and displays a table of networks. The table columns are: Project, Network Name, Subnets Associated, DHCP Agents, Shared, External, Status, Admin State, and Actions. There are five entries listed:

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	<button>Edit Network</button>
demo	MyUserNetwork		0	No	No	Active	UP	<button>Edit Network</button>
admin	MyVLANNetwork2		0	No	No	Active	UP	<button>Edit Network</button>
admin	MyVLANNetwork		0	No	No	Active	UP	<button>Edit Network</button>
admin	MyFlatNetwork	MyFlatSubnet 192.168.100.0/24	1	Yes	No	Active	UP	<button>Edit Network</button>

At the bottom of the table, it says '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:

The screenshot shows the details for the 'MyVLANNetwork'. The top navigation bar shows 'Admin / Network / Networks / MyVLANNetwork'. The main content area is titled 'MyVLANNetwork' and has tabs for Overview, Subnets, Ports, and DHCP Agents. The Overview tab is selected. It displays the following details:

Name	MyVLANNetwork
ID	b2ca3d41-c79c-416a-8e86-0bb01060ddcc
Project ID	877f949397ad428ceaaac4e5123bad83
Status	Active
Admin State	UP
Shared	No
External Network	No
MTU	1500
Provider Network	Network Type: vlan Physical Network: physnet1 Segmentation ID: 300

On the right side of the details page, there is a button labeled 'Edit Network'.

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 a network management interface for a VLAN network. At the top, there's a breadcrumb navigation: Admin / Network / Networks / MyVLANNetwork. On the right, there's an 'Edit Network' button with a dropdown arrow. Below the header, there are four tabs: Overview (selected), Subnets (highlighted in blue), Ports, and DHCP Agents. The main content area is titled 'Subnets' and contains a table with the following columns: Name, CIDR, IP Version, Gateway IP, Used IPs, Free IPs, and Actions. A search bar with a 'Filter' placeholder and a magnifying glass icon is positioned above the table. A '+ Create Subnet' button is located to the right of the search bar. The table body displays the message 'No items to display.'

4. A wizard will appear that allows you to define the properties of the new subnet:

Create Subnet



Subnet

Subnet Details

Subnet Name

MyVLANSubnet

Create a subnet associated with the network. Advanced configuration is available by clicking on the "Subnet Details" tab.

Network Address

192.168.206.0/24

Gateway IP

Disable Gateway

Cancel

« Back

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](#)

[« Back](#)

[Create](#)

6. Click on the blue Create button to complete the creation of the subnet. The Subnets panel will be updated accordingly:

MyVLANNetwork

Edit Network ▾

Overview Subnets Ports DHCP Agents

Subnets

Filter



+ Create Subnet

✖ Delete Subnets

Displaying 1 item

<input type="checkbox"/>	Name	CIDR	IP Version	Gateway IP	Used IPs	Free IPs	Actions
<input type="checkbox"/>	MyVLANSubnet	192.168.206.0/24	IPv4	192.168.206.1	1	252	<button>Edit Subnet ▾</button>

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              | e49703d8-27f4-4a16-9bf4-91a6cf00ffff3 |
| 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   | e49703d8-27f4-4a16-9bf4-91a6cf00ffff3 |
| 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:

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='6cfffc42-0366-447f-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:

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.

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 `DHCPCACK` 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: 0.0.0.68 > 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 tapee3c29d0-75 -ne port 67 or port 68
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on tapee3c29d0-75, 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: 0.0.0.68 > 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: 0.0.0.68 > 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 `dhcpdump` utility, we can observe more verbose output of the DHCP lease cycle. To install `dhcpdump`, issue the following command on all nodes:

```
| # apt install dhcpcdump
```

Within the network namespace hosting the DHCP server for the `MyVLANNetwork` network, run `dhcpdump`, as shown here:

```
| root@controller01:~# ip netns exec qdhcp-03327707-c369-4bd7-bd71-a42d9bcf49b8 dhcpcdump -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
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 **DHCPoffer** 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
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:

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
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 `dhcpdump`, 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.

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



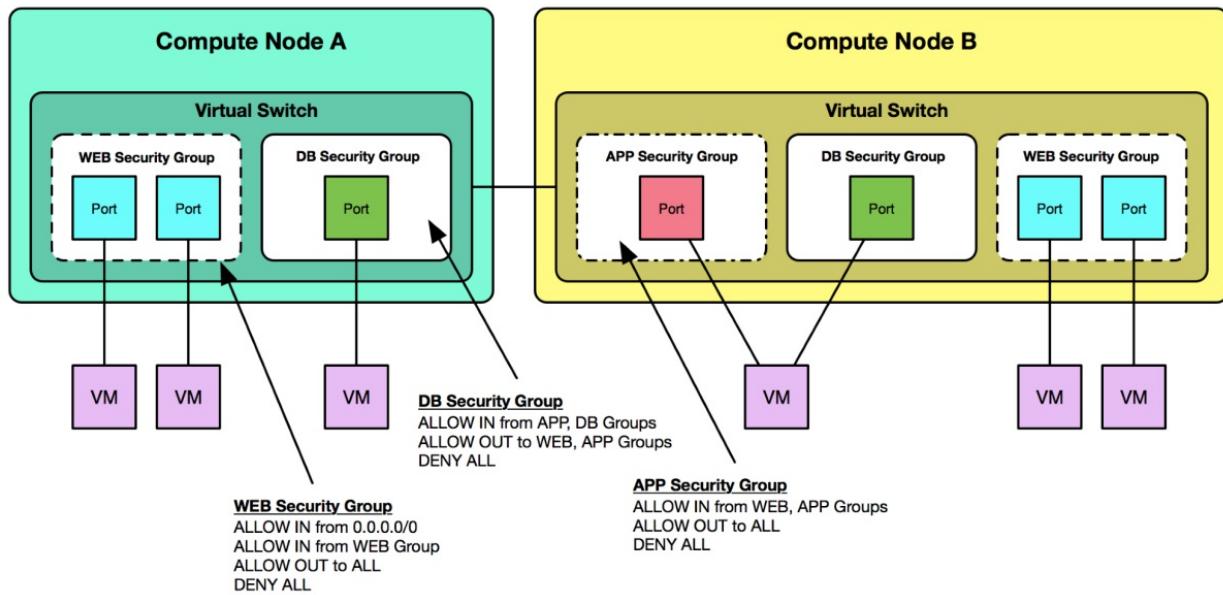
Due to its experimental status as of the Pike and Queens releases of OpenStack, the `openvswitch` firewall driver will not be discussed in this book.

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:

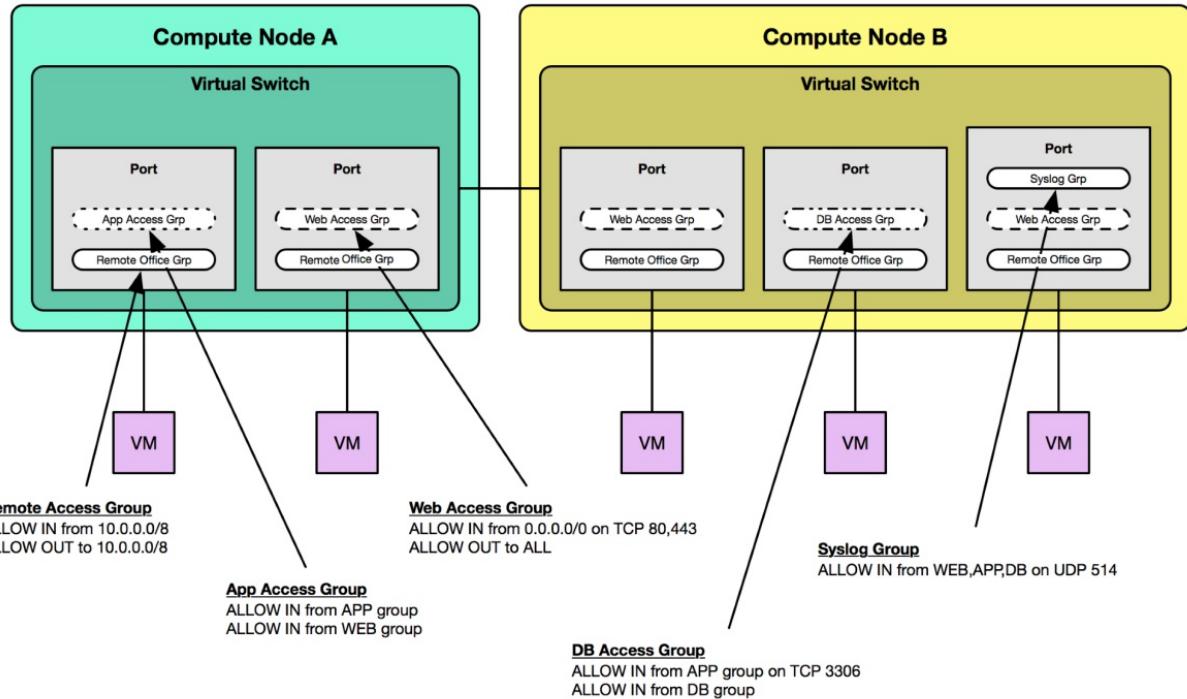


In the preceding diagram, ports within the virtual switch belong to one of three security groups: WEB, DB, or APP. When a change is made to a security group, such as adding or removing rules to the group, corresponding iptables rule changes are made automatically on the `compute` nodes.

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
security group create	Creates a new security group
security group delete	Deletes security group(s)
security group list	Lists security group(s)
security group rule create	Creates a new security group rule
security group rule delete	Deletes security group rule(s)
security group rule list	Lists security group rules
security group rule show	Displays security group rule details
security group set	Sets security group properties
security group show	Displays security group details
server add security group	Adds a security group to a server