OpenStack IN ACTION

V. K. Cody Bumgardner

MEAP

MANNING

MEAP Edition
Manning Early Access Program

OpenStack in Action

Version 14

brief contents

6 Walking through a Networking deployment

This chapter covers

- Network node prerequisites
- Deploying OpenStack Networking core
- Setting up OpenStack Networking ML2 plug-in
- Configuring OpenStack Networking DHCP, Metadata, L3, and OVS agents

In chapter 5 you walked through the deployment of an OpenStack controller node, which provides the server-side management of OpenStack services. During the controller deployment, you made controller-side configurations for several OpenStack core services, including Networking, Compute, and Storage. We discussed the configurations for each core service in relation to the controller, but the services themselves weren't covered in detail.

Chapters 6 through 8 will walk you through the deployment of core OpenStack services on resource nodes. *Resource nodes* are nodes that provide a specific resource in relation to an OpenStack service. For instance, a server running OpenStack Compute (Nova) services (and all prerequisite requirements) would be considered a *compute resource node*. As you learned in chapter 2, it's possible for a specific node to provide multiple services, including Compute (Nova), Network (Neutron), and Block Storage (Cinder). But just like an exclusive node was used for the controller in chapter 5, exclusive resource nodes will be used for demonstration in chapters 6 (Networking), 7 (Block Storage), and 8 (Compute).

Take another look at the multi-node architecture introduced in chapter 5, shown in figure 5.8.

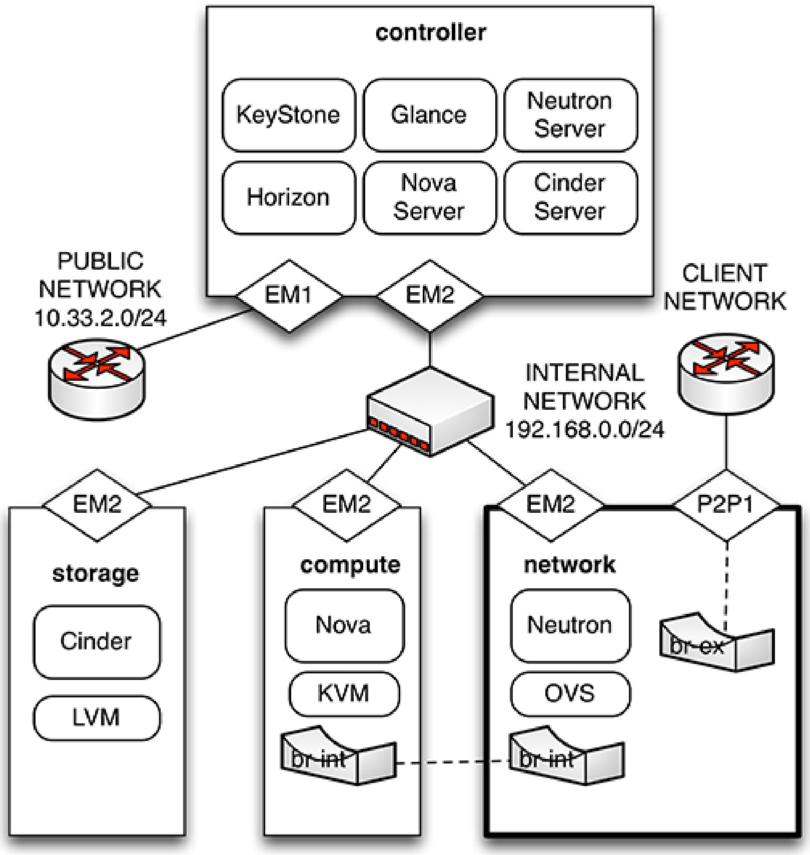


Figure 6.1 Multi-node architecture

In this chapter, you'll manually deploy the Networking components in the lower right of the figure on a standalone node.

Figure 6.16 shows your current status on your way to a working manual deployment. In this chapter, you'll first prepare the server to function as a network device. Next, you'll install and configure Neutron OSI Layer 2 (switching) components. Finally, you'll install and configure Neutron services that function on OSI Layer 3 (DHCP, Metadata, and so on). Network resources configured in this chapter will be used directly by VMs provided by OpenStack.

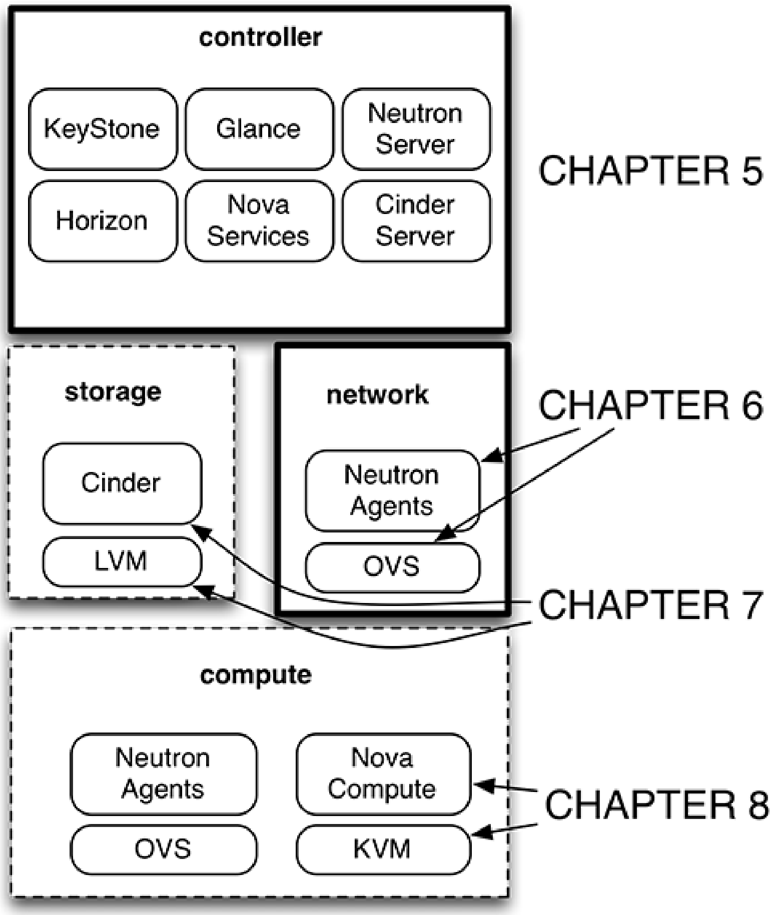


Figure 6.2 Deployment roadmap

For many people, this chapter will be the most difficult. Even if you have a deep background in traditional networking, you'll have to stop and think about how OpenStack Networking works. Overlay networks, or networks on top of other networks, are in many ways the network equivalent of the abstraction of virtual machines from bare-metal servers. This may be your first exposure to mesh/overlay/distributed networking, but these technologies are not exclusive to OpenStack. You'll learn more about overlay networks and their use in OpenStack in this chapter, but taking the time to understand the fundamental changes will be useful across many technologies.

6.1 Deploying network prerequisites

In the chapter 2 deployment, DevStack installed and configured OpenStack dependencies for you. In this chapter, you'll manually install these dependencies. Luckily, you can use a package management system to install the software: there's no compiling required, but you must still manually configure many of the components.

WARNING **Proceed with care**
Working in a multi-node environment greatly increases deployment complexity. A small, seemingly unrelated, mistake in the configuration of one component or dependency can cause issues that are very hard to track down. Read each section carefully, making sure you understand what you’re installing or configuring.

Many of the examples in this chapter include a verification step, which I highly recommend you follow. If a configuration can’t be verified, retrace your steps to the last verified point and start over. This practice will save you a great deal of frustration.

6.1.1 Preparing the environment

With the exception of the network configuration, environment preparation will be similar to preparing the controller node you deployed in chapter 5. Make sure you pay close attention to the network interfaces and addresses in the configurations. It’s easy to make a typo, and often hard to track down problems when you do.

6.1.2 Configuring the network interfaces

You want to configure the network with three interfaces:

- *Node interface*—Traffic not directly related to OpenStack. This interface will be used for administrative tasks like SSH console access, software updates, and even node-level monitoring.
- *Internal interface*—Traffic related to OpenStack component-to-component communication. This includes API and AMPQ type traffic.
- *VM interface*—Traffic related to OpenStack VM-to-VM and VM-to-external communication.

First, you’ll want to determine what interfaces already exist on the system.

REVIEWING THE NETWORK

The following command will list the interfaces on your server.

Listing 6.1 List interfaces

```
$ ifconfig -a

em1       Link encap:Ethernet  HWaddr b8:2a:72:d5:21:c3
          inet addr:10.33.2.51  Bcast:10.33.2.255  Mask:255.255.255.0
          inet6 addr: fe80::ba2a:72ff:fed5:21c3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:9580 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1357 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:8716454 (8.7 MB)  TX bytes:183958 (183.9 KB)
          Interrupt:35

em2       Link encap:Ethernet  HWaddr b8:2a:72:d5:21:c4
```

```
          inet6 addr: fe80::ba2a:72ff:fed5:21c4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7732 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:494848 (494.8 KB)  TX bytes:680 (680.0 B)
          Interrupt:38
...
p2p1      Link encap:Ethernet  HWaddr a0:36:9f:44:e2:70
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

You might have configured your node interface, `em1`, during the initial installation. You'll use the `em1` interface to communicate with this node. Take a look at the two other interfaces, `em2` and `p2p1`. On the example systems used in writing this book, the `em2` interface will be used for internal OpenStack traffic and the add-on 10G adapter, whereas `p2p1` will be used for VM communication.

Next you'll review the network configuration for the example nodes, and you'll configure controller interfaces.

CONFIGURING THE NETWORK

Under Ubuntu, the interface configuration is maintained in the /etc/network/interfaces file. We'll build a working configuration based on the italicized addresses in table 6.1.

Table 6.1 Network address table

Node	Function	Interface	IP address
Controller	Pubic interface/node address	em1	10.33.2.50/24
Controller	OpenStack internal	em2	192.168.0.50/24
Network	*Node address*	*em1*	*10.33.2.51/24*
Network	*OpenStack internal*	*em2*	*192.168.0.51/24*
Network	*VM network*	*p2p1*	*None: assigned to OpenStack Networking*
Storage	Node address	em1	10.33.2.52/24
Storage	OpenStack internal	em2	192.168.0.52/24
Compute	Node address	em1	10.33.2.53/24
Compute	OpenStack internal	em2	192.168.0.53/24

In order to modify the network configuration, or any privileged configuration, you must use sudo privileges (`sudo vi /etc/network/interfaces`). Any text editor can be used in this process.

Modify your interfaces file as shown next.

Listing 6.2 Modify interface config /etc/network/interfaces

```
# The loopback network interface
auto lo
iface lo inet loopback

# The OpenStack Node Interface
auto em1   ❶
iface em1 inet static
        address 10.33.2.51
        netmask 255.255.255.0
        network 10.33.2.0
        broadcast 10.33.2.255
        gateway 10.33.2.1
        dns-nameservers 8.8.8.8
        dns-search testco.com

# The OpenStack Internal Interface
auto em2   ❷
iface em2 inet static
        address 192.168.0.51
        netmask 255.255.255.0

# The VM network interface
auto p2p1    ❸
iface p2p1 inet manual
```

❶ em1 is the public interface used for node administration.
❷ em2 is used primarily for AMPQ and API traffic between resource nodes and the controller.
❸ p2p1 virtual machine traffic between resource nodes and external networks

In your network configuration interface, `em1` will be used for node administration, such as SSH sessions to the actual server ❶. OpenStack shouldn't use this interface directly. The `em2` interface will be used primarily for AMPQ and API traffic between resource nodes and the controller ❷. The `p2p1` interface will be managed by Neutron. This interface will primarily carry virtual machine traffic between resource nodes and external networks ❸.

You should now refresh the network interfaces for which the configuration was changed. If you didn't change the settings of your primary interface, you shouldn't experience an interruption. If you changed the address of the primary interface, it's recommend you reboot the server at this point.

You can refresh the network configuration for a particular interface as shown here for interfaces `em2` and `p2p1`.

Listing 6.3 Refreshing Networking settings

```
sudo ifdown em2 && sudo ifup em2
sudo ifdown p2p1 && sudo ifup p2p1
```

The network configuration, from an operating system standpoint, should now be active. The interface will automatically be brought online based on your configuration. This process can be repeated for each interface that requires a configuration refresh. In order to confirm that the configuration was applied, you can once again check your interfaces.

Listing 6.4 Check network for updates

```
$ ifconfig -a

em1       Link encap:Ethernet  HWaddr b8:2a:72:d5:21:c3
          inet addr:10.33.2.51  Bcast:10.33.2.255  Mask:255.255.255.0
          inet6 addr: fe80::ba2a:72ff:fed5:21c3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10159 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1672 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:8803690 (8.8 MB)  TX bytes:247972 (247.9 KB)
          Interrupt:35

em2       Link encap:Ethernet  HWaddr b8:2a:72:d5:21:c4
          inet addr:192.168.0.51  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::ba2a:72ff:fed5:21c4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7913 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:506432 (506.4 KB)  TX bytes:680 (680.0 B)
          Interrupt:38
...
p2p1      Link encap:Ethernet  HWaddr a0:36:9f:44:e2:70
          inet6 addr: fe80::a236:9fff:fe44:e270/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:648 (648.0 B)
```

At this point you should be able to remotely access the network server, and the server should have internet access. The remainder of the install can be performed remotely using SSH or directly from the console.

6.1.3 Updating packages

The APT package index is a database of all available packages defined in the /etc/apt/sources.list file. You need to make sure your local database is synchronized with the latest packages available in the repository for your specific Linux distribution. Prior to installation, you should also upgrade any repository items, including the Linux kernel, that might be out of date.

Listing 6.5 Update and upgrade packages

```
sudo apt-get -y update
sudo apt-get -y upgrade
```

You now need to reboot the server to refresh any packages or configurations that might have changed.

Listing 6.6 Reboot server

```
sudo reboot
```

As of Ubuntu Server 14.04 (Trusty Tahr), the following OpenStack components are officially supported and included with the base distribution:

- *Nova*—Project name for OpenStack Compute; it works as an IaaS cloud fabric controller
- *Glance*—Provides services for virtual machine image, discovery, retrieval, and registration
- *Swift*—Provides highly scalable, distributed, object store services

- *Horizon*—Project name for OpenStack Dashboard; it provides a web-based admin/user GUI
- *Keystone*—Provides identity, token, catalog, and policy services for the OpenStack suite
- *Neutron*—Provides network management services for OpenStack components
- *Cinder*—Provides block storage as a service to OpenStack Compute

6.1.4 Software and configuration dependencies

In this section, you'll install a few software dependencies and make a few configuration changes in preparation for the install.

INSTALLING LINUX BRIDGE AND VLAN UTILITIES

You'll want to install the package bridge-utils, which provides a set of applications for working with network bridges on the system (OS) level. Network bridging on the OS level is critical to the operation of OpenStack Networking. For the time being, it's sufficient to think about network bridges under Linux as simply placing multiple interfaces on the same network segment (the same isolated VLAN). The default operation of Linux network bridging is to act like a switch, so you can certainly think of it this way.

In addition, you may want to install the vlan package, which provides the network subsystem the ability to work with Virtual Local Area Networks (VLANs) as defined by IEEE 802.1Q. VLANS allow you to segregate network traffic using VLAN IDs on virtual interfaces. This allows a single physical interface managed by your OS to isolate multiple networks using virtual interfaces. VLAN configuration won't be used in the examples, but you should be aware of the technology.

NOTE **Using VLANs with Neutron**
Instructions for installing the *vlan* package are included in listing 5.75 because the vast majority of deployments will make use of IEEE 802.1Q VLANs to deliver multiple networks to Neutron nodes. But, for the sake of clarity, the examples in this book will not use VLAN interfaces. Once you understand OpenStack Networking, the adoption of VLANs on the OS level is trivial.

In summary, VLANs *isolate* traffic and interfaces, whereas Linux bridges *aggregate* traffic and interfaces.

Listing 6.7 Install vlan and bridge-utils

```
$ sudo apt-get -y install vlan bridge-utils
...
Setting up bridge-utils (1.5-6ubuntu2) ...
Setting up vlan (1.9-3ubuntu10) ...
```

You now have the ability to create VLANs and Linux bridges.

SERVER-TO-ROUTER CONFIGURATION

OpenStack manages resources for providing virtual machines. One of those resources is the network used by the virtual machine to communicate with other virtual and physical machines. For OpenStack Networking to provide network services, at least one resource node that performs the functions of a network devices (routing, switching, and so on) must exist. You want this node to act as a router and switch for network traffic.

By default, the Linux kernel isn't set to allow the routing of traffic between interfaces. The command `sysctl` is used to modify kernel parameters, such as those related to basic network functions. You need to make several modifications to your kernel settings using this tool.

The first modification is related to the forwarding or routing (kernel IP forwarding) of traffic between network interfaces by the Linux kernel. You want traffic arriving on one interface to be forwarded or routed to another interface if the kernel determines that the destination network can be found on another interface maintained by the kernel. Take a look at figure 6.16, which shows a server with two interfaces.

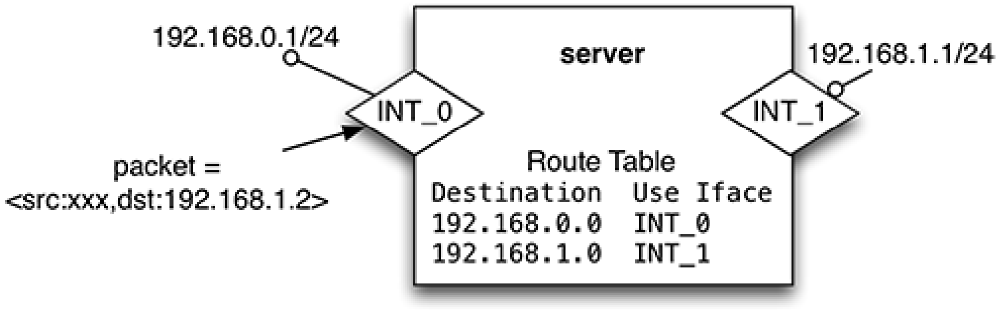


Figure 6.3 Linux IP routing

By default, the incoming packet shown in the figure will be dropped by interface `INT_0` because the address of this interface isn't the destination of the packet. But you want the server to inspect the packet's destination address, look in the server routing table, and, if a route is found, forward the packet to the appropriate interface. The `sysctl` setting `net.ipv4.ip_forward` instructing the kernel to forward traffic can be seen in listing 5.75.

In addition to enabling kernel IP forwarding, you also have to make a few other less-common kernel configuration changes. In the world of networking, there's something called *asymmetric routing*, where outgoing and incoming traffic paths/routes are not the same. There are legitimate reasons to do such things (such as terrestrial upload and satellite download; see www.google.com/patents/US6038594), but more often than not this ability was exploited by distributed denial of service (DDOS) attacks. RFC 3704, "Ingress Filtering for Multihomed Networks," also known as *reverse-path filtering*, was introduced to limit the impact of these DDOS attacks. By default, if the Linux kernel can't determine the source route of a packet, it will be dropped. OpenStack Networking is a complex platform that encompasses many layers of network resources, where the network resources themselves don't have a complete picture of the network. You must configure the kernel to disable reverse-path filtering, which leaves path management up to OpenStack.

The `sysctl` setting `net.ipv4.conf.all.rp_filter` that's used to disable reverse-path filtering for all existing interfaces is shown in listing 5.75. The `sysctl` setting `net.ipv4.conf.default.rp_filter` is used to disable reverse-path filtering for all future interfaces.

Apply the settings in the following listing to your OpenStack Network node.

Listing 6.8 Modify /etc/sysctl.conf

```
net.ipv4.ip_forward=1
net.ipv4.conf.all.rp_filter=0
net.ipv4.conf.default.rp_filter=0
```

To enable the `sysctl` kernel changes without restarting the server, invoke the `sysctl -p` command.

Listing 6.9 Execute the `sysctl` command

```
$ sudo sysctl -p
net.ipv4.conf.default.rp_filter = 0
net.ipv4.conf.all.rp_filter = 0
net.ipv4.ip_forward = 1
```

The interfaces should now forward IPv4 traffic, and reverse-path filtering should be disabled.

In the next section, you'll add advanced network features to your user with the Open vSwitch package.

6.1.5 Installing Open vSwitch

OpenStack Networking takes advantage of the open source distributed virtual-switching package, Open vSwitch (OVS). OVS provides the same data-switching functions as a physical switch (L2 traffic on port A destined to port B is switched to port B), but it runs in software on servers.

SIDEBAR **What does a switch do?**

To understand what a switch does, you must first look at an Ethernet hub (you're likely using Ethernet in some form on all of your wired and wireless devices). "What is a hub?" you ask.

Circa early 1990s, there were several competing OSI Layer 1 (physical) Ethernet topologies. One such topology, IEEE 10Base2, worked (and looked) much like the cable TV in your house, where you could take a single cable and add network connections by splicing in T connectors (think *splitters*). Another common topology was 10BaseT (RJ45 connector twisted pair), which is the grandfather of what most of us think of as "Ethernet" today. The good thing about 10BaseT was that you could extend the network without interrupting network service; the bad thing was that this physical topology required a device to terminate the cable segments together. This device was a called a *hub*, and it also operated at the OSI Layer 1 (physical) level. If data was transmitted by a device on port A, it would be physically transmitted to all other ports on the hub.

Aside from the obvious security concerns related to transmitting all data to all ports, the operation of a hub wouldn't scale. Imagine thousands of devices connected to hundreds of interconnected switches. All traffic was flooded to all ports. To solve this issue, network switches were developed. Manufactures of Network Interface Cards (NICs) assigned a unique Ethernet Hardware Address (EHA) to every card. Switches kept track of the EHA addresses, commonly known as Media Access Control (MAC) addresses, on each port of the switch. If a packet with the destination MAC=xyz was transmitted to port A, and the switch had a record of xyz on port B, the packet was transmitted (switched) to port B. Switches operate on OSI Layer 2 (Link Layer) and switch traffic based on MAC destinations.

The examples in this book, from a network-switching standpoint, make exclusive use of the OVS switching platform.

SIDEBAR **OVS is not a strict OpenStack network dependency**

Without a doubt, OVS is used often with OpenStack Networking. But it’s not implicitly required by the framework. The following diagram, first introduced in chapter 4, shows where OVS fits into the OpenStack Network architecture.

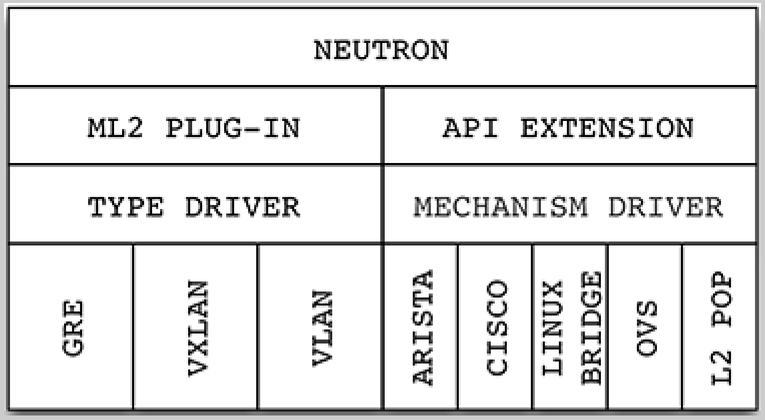


Figure 6.4 OVS is an L2 mechanism

You could use basic Linux bridging (the previously discussed virtual switch) or even a physical switch instead of OVS, as long as it’s supported by a vendor-specific Neutron plug-in or module.

At this point you have a server that can act like a basic network router (via IP kernel forwarding) and a basic switch (via Linux network bridging). You’ll now add advanced switching capabilities to your server by installing OVS. OVS could be the topic of an entire book, but it’s sufficient to say that the switching features provided by OVS rival offerings provided by standalone network vendors.

You can turn your server into an advanced switch with the following OVS install instructions.

Listing 6.10 Install OVS

```
$ sudo apt-get -y install openvswitch-switch
...
Setting up openvswitch-common ...
Setting up openvswitch-switch ...
openvswitch-switch start/running
```

The Open vSwitch install process will install a new OVS kernel module. In addition, the OVS kernel module will reference and load additional kernel models (GRE, VXLAN, and so on) as necessary to build network overlays.

SIDEBAR **What is a network overlay?**

For a minute, forget what you know about traditional networking. Forget the concept of servers on the same switch (VLAN/network) being on the same “network.” Imagine that you have a way to place any VM on any network, regardless of its physical location or underlying network topology. This is the value proposition for overlay networks.

At this point it’s sufficient to think about an overlay network as a fully meshed virtual private network (VPN) between all participating endpoints (all servers being on the same L2 network segment regardless of location). To create a network such as this, you’ll need technologies to tunnel traffic between endpoints. GRE, VXLAN, and other protocols provide the tunneling transports used by overlay networks. As usual, OpenStack simply manages these components. A network overlay is simply a method of extending L2 networks between hosts “overlaid” on top of other networks.

WARNING **Know thy kernel**

Ubuntu 14.04 LTS is the first Ubuntu release to ship with kernel support for OVS overlay networking technologies (GRE, VXLAN, and the like). In previous versions, additional steps had to be taken to build appropriate kernel modules. If you’re using another version of Ubuntu or another distribution altogether, make absolutely sure OVS kernel modules are loaded as shown in listing 5.75.

You want to be absolutely sure the Open vSwitch kernel modules were loaded. You can use the `lsmod` command in the following listing to confirm the presence of OVS kernel modules.

Listing 6.11 Verify OVS kernel modules

```
$ sudo lsmod | grep openvswitch
Module                       Size   Used by
openvswitch            66901  0
gre                    13796  1 openvswitch
vxlan                  37619  1 openvswitch
libcrc32c              12644  1 openvswitch
```

The output of the *lsmod* command should now show several resident modules related to OVS:

- `openvswitch`—This is the OVS module itself, which provides the interface between the kernel and OVS services.
- `gre`—Designated as “used by” the `openvswitch` module, it enables GRE functionality on

the kernel level.

- `vxlan`—Just like the GRE module, `vxlan` is used to provide VXLAN functions on the kernel level.
- `libcrc32c`—Provides kernel-level support for cyclic redundancy check (CRC) algorithms, including hardware offloading using Intel's CRC32C CPU instructions. Hardware offloading is important for the high-performance calculation of network flow hashes and other CRC functions common to network headers and data frames.

Having GRE and VXLAN support on the kernel level means that the transports used to create overlay networks are understood by the system kernel, and by relation the Linux network subsystem.

SIDEBAR **No modules? DKMS to the rescue!**

Dynamic Kernel Module Support (DKMS) was developed to make it easier to provide kernel-level drivers outside of the mainline kernel. DKMS has historically been used by OVS to provide kernel drivers for things such as overlay network devices (such as GRE and VXLAN), that were not included directly in the Linux kernel. The kernel that ships with Ubuntu 14.04 includes support for overlay devices built into the kernel, but depending on your distribution and release, you might not have a kernel with built-in support for the required network overlay technologies.

The following command will deploy the appropriate dependencies and build the OVS `datapath` module using the DKMS framework:

```
sudo apt-get -y install openvswitch-datapath-dkms
```

Only run this command if the modules couldn't be validated as shown in listing 5.75.

If you think the kernel module should have loaded, but you still don't see it, restart the system and see if it loads on restart. Additionally, you can try to load the kernel module with the command `modprobe openvswitch`. Check the kernel log, /var/log/kern/log, for any errors related to loading OVS kernel modules. OVS won't function for your purposes without the appropriate resident kernel modules.

6.1.6 Configuring Open vSwitch

You now need to add an internal `br-int` bridge and an external `br-ex` OVS bridge.

The `br-int` bridge interface will be used for communication within Neutron-managed networks. Virtual machines communicating within internal OpenStack Neutron-created networks will use this bridge for communication. This interface shouldn't be confused with the internal interface on the operating system level.

Listing 6.12 Configure internal OVS bridge

```
sudo ovs-vsctl add-br br-int
```

Now that `br-int` has been created, create the external bridge interface, `br-ex`. The external bridge interface will be used to bridge OVS-managed internal Neutron networks with physical external networks.

Listing 6.13 Configure external OVS bridge

```
sudo ovs-vsctl add-br br-ex
```

You'll also want to confirm that the bridges were successfully added to OVS and that they're visible to the underlying networking subsystem. You can do that with the following commands.

Listing 6.14 Verify OVS configuration

```
$ sudo ovs-vsctl show
8cff16ee-40a7-40fa-b4aa-fd6f1f864560
    Bridge br-int
        Port br-int
            Interface br-int
                type: internal
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
    ovs_version: "2.0.2"
```

Listing 6.15 Verify OVS OS integration

```
$ ifconfig -a
br-ex    Link encap:Ethernet  HWaddr d6:0c:1d:a8:56:4f     ❶
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)


br-int    Link encap:Ethernet  HWaddr e2:d9:b2:e2:00:4f     ❷
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
...
em1       Link encap:Ethernet  HWaddr b8:2a:72:d5:21:c3
          inet addr:10.33.2.51  Bcast:10.33.2.255  Mask:255.255.255.0
          inet6 addr: fe80::ba2a:72ff:fed5:21c3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:13483 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2763 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:12625608 (12.6 MB)  TX bytes:424893 (424.8 KB)
          Interrupt:35
...
ovs-system Link encap:Ethernet  HWaddr 96:90:8d:92:19:ab     ❶
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

❶ br-ex bridge
❷ br-int bridge
❸ ovs-system interface

Notice the addition of the `br-ex` ❷ and `br-int` ❷ bridges in your interface list. The new bridges will be used by OVS and the Neutron OVS module for internal and external traffic. In addition, the `ovs-system` interface ❸ was added. This is the OVS datapath interface, but you won't have to worry about working with this interface; it's simply an artifact of Linux kernel integration. Nevertheless, the presence of this interface is an indication that the OVS kernel modules are active.

At this point you have an operational OVS deployment and two bridges. The `br-int` (internal) bridge will be used by Neutron to attach virtual interfaces to the network bridge. These tap interfaces will be used as endpoints for the Generic Routing Encapsulation (GRE) tunnels. GRE tunnels are used to create point-to-point network connections (think VPN) between endpoints over the Internet Protocol (IP), and Neutron will configure GRE tunnels between compute and network nodes using OVS. These tunnels will provide a mesh of virtual networks between all possible resource locations and network drains in the topology. This mesh provides the functional equivalence of a single isolated OSI L2 network for the virtual machines on the same virtual network. The internal bridge won't need to be associated with a physical interface or be placed in an OS-level "UP" state to work.

The `br-ex` (external) bridge will be used to connect the OVS bridges and Neutron-derived virtual interfaces to the physical network. You must associate the external bridge with your `VM` interface as follows.

Listing 6.16 Add interface `p2p1` (VM) to bridge `br-ex`

```
sudo ovs-vsctl add-port br-ex p2p1
sudo ovs-vsctl br-set-external-id br-ex bridge-id br-ex
```

Now check that the `p2p1` interface was added to the `br-ex` bridge.

Listing 6.17 Verify OVS configuration

```
$ sudo ovs-vsctl show
8cff16ee-40a7-40fa-b4aa-fd6f1f864560
    Bridge br-int
        Port br-int
            Interface br-int
                type: internal
    Bridge br-ex   ❶
        Port br-ex
            Interface br-ex
                type: internal
        Port "p2p1"
            Interface "p2p1"   ❷
    ovs_version: "2.0.1"
```

❶ p2p1 interface
❷ br-ex bridge

Notice the `p2p1` interface ❸ listed as a port on the `br-ex` bridge ❷. This means that the `p2p1` interface is virtually connected to the OVS `br-ex` bridge interface.

Currently the `br-ex` and `br-int` bridges aren't connected. Neutron will configure ports on both the internal and external bridges, including taps between the two. Neutron will do all of the OVS configuration from this point forward.

6.2 Installing Neutron

In this section, you'll prepare the Neutron ML2 plug-in, L3 agent, DHCP agent, and Metadata agent for operation. The ML2 plug-in is installed on every physical node where Neutron interacts with OVS.

You'll install the ML2 plug-in and agent on all compute and network nodes. The ML2 plug-in will be used to build Layer 2 (data link layer, Ethernet layer, and so on) configurations and tunnels between network endpoints managed by OpenStack. You can think of these tunnels as virtual network cables connecting separate switches or VMs together.

The L3, Metadata, and DHCP agents are only installed on the network nodes. The L3 agent will provide Layer 3 routing of IP traffic on the established L2 network. Similarly, the Metadata and DHCP agents provide L3 services on the L2 network.

The agents and plug-in provide the following services:

- *ML2 plug-in*—The ML2 plug-in is the link between Neutron and OSI L2 services. The plug-in manages local ports and taps, and it generates remote connections over GRE tunnels. This agent will be installed on network and compute nodes. The plug-in will be configured to work with OVS.
- *L3 agent*—This agent provides Layer 3 routing services and is deployed on network nodes.
- *DHCP agent*—This agent provides DHCP services for Neutron-managed networks using DNSmasq. Normally this agent will be installed on a network node.

- *Metadata agent*—This agent provides cloud-init services for booting VMs and is typically installed on the network node.

6.2.1 Installing Neutron components

You're now ready to install Neutron software as follows.

Listing 6.18 Install Neutron components

```
$ sudo apt-get -y install neutron-plugin-ml2 \
neutron-plugin-openvswitch-agent neutron-l3-agent \
neutron-dhcp-agent
...
Adding system user `neutron' (UID 109) ...
Adding new user `neutron' (UID 109) with group `neutron' ...
...
Setting up neutron-dhcp-agent  ...
neutron-dhcp-agent start/running, process 14910
Setting up neutron-l3-agent  ...
neutron-l3-agent start/running, process 14955
Setting up neutron-plugin-ml2 ...
Setting up neutron-plugin-openvswitch-agent  ...
neutron-plugin-openvswitch-agent start/running, process 14994
```

Neutron plug-ins and agents should now be installed. You can continue on with the Neutron configuration.

6.2.2 Configuring Neutron

The next step is configuration. First, you must modify the /etc/neutron/neutron.conf file to define the service authentication, management communication, core network plug-in, and service strategies. In addition, you'll provide configuration and credentials to allow the Neutron client instance to communicate with the Neutron controller, which you deployed in chapter 5. Modify your neutron.conf file based on the values shown below. If any of these values doesn't exist, add it.

Listing 6.19 Modify /etc/neutron/neutron.conf

```
[DEFAULT]
verbose = True
auth_strategy = keystone

rpc_backend = neutron.openstack.common.rpc.impl_kombu
rabbit_host = 192.168.0.50
rabbit_password = openstack1

core_plugin = neutron.plugins.ml2.plugin.Ml2Plugin
allow_overlapping_ips = True
service_plugins = router,firewall,lbaas,vpnaas,metering

nova_url = http://127.0.0.1:8774/v2
nova_admin_username = admin
nova_admin_password = openstack1
nova_admin_tenant_id = b3c5ebecb36d4bb2916fecd8aed3aa1a
nova_admin_auth_url = http://10.33.2.50:35357/v2.0
```

```
[keystone_authtoken]
auth_url =  http://10.33.2.50:35357/v2.0
admin_tenant_name = service
admin_password = openstack1
auth_protocol = http
admin_user = neutron

[database]
connection = mysql://neutron_dbu:openstack1@192.168.0.50/neutron
```

Now that the core Neutron components are configured, you must configure the Neutron agents, which will allow Neutron to control network services.

6.2.3 Configuring the Neutron ML2 plug-in

The Neutron OVS agent allows Neutron to control the OVS switch.

This configuration can be made in the /etc/neutron/plugins/ml2/ml2_conf.ini file. The following listing provides the database information, along with ML2-specific switch configuration.

Listing 6.20 Modify /etc/neutron/plugins/ml2/ml2_conf.ini

```
[ml2]
type_drivers = gre
tenant_network_types = gre
mechanism_drivers = openvswitch

[ml2_type_gre]
tunnel_id_ranges = 1:1000

[ovs]
local_ip = 192.168.0.51
tunnel_type = gre
enable_tunneling = True

[securitygroup]
firewall_driver =
neutron.agent.linux.iptables_firewall.OVSHybridIptablesFirewallDriver
enable_security_group = True
```

Your Neutron ML2 plug-in configuration is now complete. Clear the log file, and then restart the service:

```
sudo rm /var/log/neutron/openvswitch-agent.log
sudo service neutron-plugin-openvswitch-agent restart
```

Your Neutron ML2 plug-in agent log should now look something like the following:

```
Logging enabled!
Connected to AMQP server on 192.168.0.50:5672
Agent initialized
successfully, now running...
```

You now have OSI L2 Neutron integration using OVS. In the next section, you'll configure the OSI L3 Neutron services.

6.2.4 Configuring the Neutron L3 agent

Next, you need to configure the Neutron L3 agent. This agent provides L3 services, such as routing, for VMs. The L3 agent will be configured to use Linux namespaces.

SIDEBAR **What is Linux namespace isolation?**
There's a feature built into the Linux kernel called *namespace isolation*. This feature allows you to separate processes and resources into multiple namespaces so that they don't interfere with each other. This is done internally by assigning namespace identifiers to each process and resource. From a network perspective, namespaces can be used to isolate network interfaces, firewall rules, routing tables, and so on. This is the underlying way in which multiple tenant networks, residing on the same Linux server, can have the same address ranges.

Go ahead and configure your L3 agent.

Listing 6.21 Modify /etc/neutron/l3_agent.ini

```
[DEFAULT]
interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver
use_namespaces = True
verbose = True
```

The L3 agent is now configured and will use Linux namespaces.
Clear the log file, and then restart the service:

```
sudo rm /var/log/neutron/l3-agent.log
sudo service neutron-l3-agent restart
```

Your Neutron L3 agent log should look something like the following:

```
Logging enabled!
Connected to AMQP server on 192.168.0.50:5672
L3 agent started
```

6.2.5 Configuring the Neutron DHCP agent

You'll next want to configure the DHCP agent, which provides DHCP services for VM images. Modify your dhcp_agent.ini as shown in the following listing.

Listing 6.22 Modify /etc/neutron/dhcp_agent.ini

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

```
interface_driver = neutron.agent.linux.interface.OVSInterfaceDriver
dhcp_driver = neutron.agent.linux.dhcp.Dnsmasq
use_namespaces = True
...
```

The DHCP agent is now configured and will use Linux namespaces. Clear the log file, and then restart the service:

```
sudo rm /var/log/neutron/dhcp-agent.log
sudo service neutron-dhcp-agent restart
```

Your Neutron DHCP agent log should look something like this:

```
Logging enabled!
Connected to AMQP server on 192.168.0.50:5672
DHCP agent started Synchronizing state
Synchronizing state complete
```

6.2.6 Configuring the Neutron Metadata agent

You'll next want to configure the Metadata agent, which provides environmental information to VM images. Cloud-init, which was originally created by Amazon for E2 services, is used to inject system-level settings on VM startup. To use Metadata services, you must use an image with a cloud-init–compatible agent installed and enabled.

Cloud-init is supported in most modern Linux distributions. Either download an image that has cloud-init preinstalled or install the package from your distribution.

Modify your metadata_agent.ini file to include the following information.

Listing 6.23 Modify /etc/neutron/metadata_agent.ini

```
[DEFAULT]
auth_url =  http://10.33.2.50:35357/v2.0
auth_region = RegionOne
admin_tenant_name = service
admin_password = openstack1
auth_protocol = http
admin_user = neutron
nova_metadata_ip = 192.168.0.50
metadata_proxy_shared_secret = openstack1
```

The Neutron Metadata agent is now configured and will use Linux namespaces. Clear the log file, and then restart the service:

```
sudo rm /var/log/neutron/metadata-agent.log
sudo service neutron-metadata-agent restart
```

Your Neutron Metadata agent log should look something like this:

```
Logging enabled!
```

```
(11074) wsgi starting up on http:///:v/
Connected to AMQP server on 192.168.0.50:5672
```

6.2.7 Restarting and verifying Neutron agents

It's a good idea at this point to restart all Neutron services, as shown in the following listing. Alternatively, you could simply restart the server.

Listing 6.24 Restart Neutron agents

```
$ cd /etc/init.d/; for i in $( ls neutron-* ); \
do sudo service $i restart; done
neutron-dhcp-agent stop/waiting
neutron-dhcp-agent start/running, process 16259
neutron-l3-agent stop/waiting
neutron-l3-agent start/running, process 16273
neutron-metadata-agent stop/waiting
neutron-metadata-agent start/running, process 16283
neutron-ovs-cleanup stop/waiting
neutron-ovs-cleanup start/running
```

You'll want to check the Neutron logs to make sure each service started successfully and is listening for requests. The logs can be found in the /var/log/neutron or /var/log/upstart/neutron-* directory.

Review the logs, checking for connections to the AMQP (RabbitMQ) server, and ensure there are no errors. The log files should exist even if they're empty. Ensure that there are no errors about unsupported OVS tunnels in the file /var/log/neutron/openvswitch-agent.log. If you experience such errors, restart the operating system and see if reloading the kernel modules and OVS takes care of the problem.

If you continue to experience problems starting Neutron services, you can increase the verbosity of the services through the /etc/neutron/neutron.conf file or the corresponding agent file.

6.2.8 Creating Neutron networks

In chapter 3 you were introduced to OpenStack Networking. This section reviews items presented in that chapter as they relate to the components you've deployed in this chapter.

Before you start creating networks using OpenStack Networking, you need to recall the basic differences between traditional "flat" networks, typically used for virtual and physical machines, and how OpenStack Networking works.

The term *flat* in *flat network* alludes to the absence of a virtual routing tier; the VM has direct access to a network, just as if you plugged a physical device into a physical network switch. Figure 3.17 shows an example of a flat network connected to a physical router.

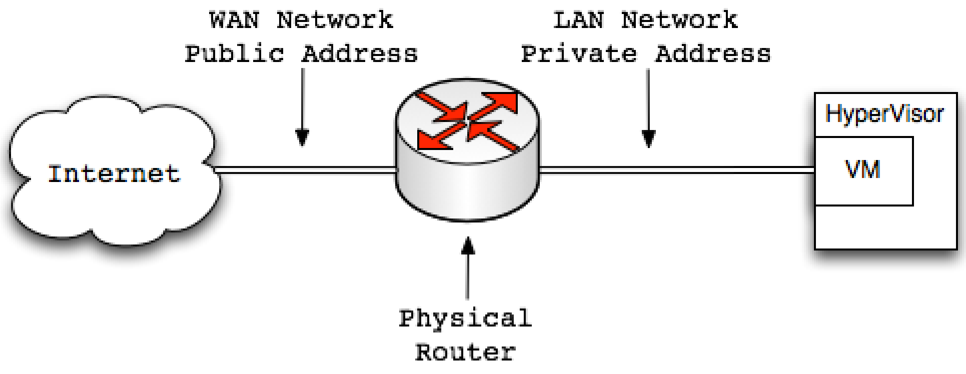


Figure 6.5 Traditional flat network

In this type of deployment, all network services (DHCP, load balancing, routing, and so on) beyond simple switching (OSI Model, Layer 2) must be provided outside of the virtual environment. For most systems administrators, this type of configuration will be very familiar, but this is not how we're going to demonstrate the power of OpenStack. You can make OpenStack Networking behave like a traditional flat network, but this approach will limit the benefits of the OpenStack framework.

In this section, you'll build a tenant network from scratch. Figure 3.17 illustrates an OpenStack tenant network, with virtual isolation from the physical external network.

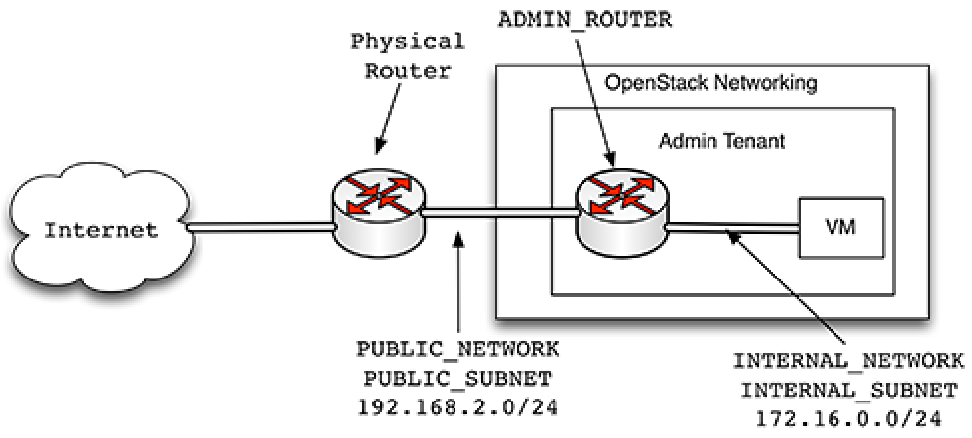


Figure 6.6 OpenStack tenant network

SIDEBAR **Set your environment variables**

The configurations in the following subsections require OpenStack authentication. In the previous examples, command-line arguments were provided for credentials. For the sake of simplicity, though, the following examples will use environment variables instead of command-line arguments.

To set your environment variables, execute the following commands in your shell:

```
$ export OS_USERNAME=admin
$ export OS_PASSWORD=openstack1
$ export OS_TENANT_NAME=admin
$ export OS_AUTH_URL=http://10.33.2.50:5000/v2.0
```

NETWORK (NEUTRON) CONSOLE

Neutron commands can be entered through the Neutron console (which is like a command line for a network router or switch) or directly through the CLI. The console is very handy if you know what you're doing, and it's a natural choice for those familiar with the Neutron command set. For the sake of clarity, however, this book demonstrates each action as a separate command, using CLI commands.

The distinction between the Neutron console and the Neutron CLI will be made clear in the following subsections. There are many things you can do with the Neutron CLI and console that you can't do in the Dashboard. Although the demonstrations will be executed using the CLI, you'll still need to know how to access the Neutron console. As you can see from the following, it's quite simple. Using the `neutron` command without arguments will take you to the console. All of the subcommands will be listed using the command shown in the following listing.

Listing 6.25 Access Neutron console

```
devstack@devstack:~/devstack$ neutron
(neutron) help

Shell commands (type help <topic>):
===================================
...
(neutron)
```

You now have the ability to access the interactive Neutron console. Any CLI configurations can be made either in the console or directly on the command line.

In the next subsection, you'll create a new network.

INTERNAL NETWORKS

The first step you'll take in providing a tenant-based network is to configure the internal network. The internal network is used directly by instances in your tenant. The internal network works on the ISO Layer 2, so for the network types, this is the virtual equivalent of providing a network switch to be used exclusively for a particular tenant.

In order to create an internal network for a tenant, you must first determine your tenant ID:

```
$ keystone tenant-list
+----------------------------------+---------+---------+
|                id                |   name  | enabled |
+----------------------------------+---------+---------+
| 55bd141d9a29489d938bb492a1b2884c |  admin  |   True  |
| b3c5ebecb36d4bb2916fecd8aed3aa1a | service |   True  |
+----------------------------------+---------+---------+
```

By using the commands in listing 3.27, you can create a new network for your tenant. First, you tell OpenStack Networking (Neutron) to create a new network ❶. Then you specify the `admin tenant-id` on the command line ❷. Finally, you specify the name of the tenant network ❸.

Listing 6.26 Create internal network

```
$ neutron net-create \    ❶
--tenant-id 55bd141d9a29489d938bb492a1b2884c \    ❷
INTERNAL_NETWORK    ❸
Created a new network:
+---------------------------+--------------------------------------+
| Field                     | Value                                |
+---------------------------+--------------------------------------+
| admin_state_up            | True                                 |
| id                        | 5b04a1f2-1676-4f1e-a265-adddc5c589b8 |
| name                      | INTERNAL_NETWORK                     |
| provider:network_type     | gre                                  |
| provider:physical_network |                                      |
| provider:segmentation_id  | 1                                    |
| shared                    | False                                |
| status                    | ACTIVE                               |
| subnets                   |                                      |
| tenant_id                 | 55bd141d9a29489d938bb492a1b2884c     |
+---------------------------+--------------------------------------+
```

❶ Tells Neutron to create a new network
❷ Specifies the admin tenant-id
❸ Specifies the network name

Figure 3.17 illustrates the `INTERNAL_NETWORK` you created for your tenant. The figure shows the network you just created connected to a VM (if one was in the tenant).

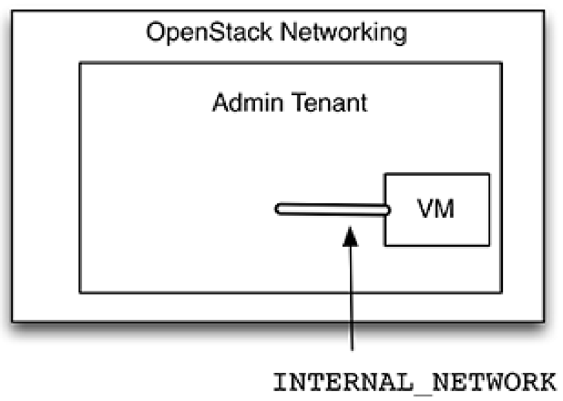


Figure 6.7 Created internal network

You’ve now created an internal network. In the next subsection, you’ll create an internal subnet for this network.

INTERNAL SUBNETS

In the previous subsection, you created an internal network. The internal network you created inside your tenant is completely isolated from other tenants. This will be a strange concept to those who work with physical servers, or even those who generally expose their virtual machines directly to physical networks. Most people are used to connecting their servers to the network, and network services are provided on a data center or enterprise level. We don’t typically think about networking and computation being controlled under the same framework.

As previously mentioned, OpenStack can be configured to work in a flat network configuration. But there are many advantages to letting OpenStack manage the network stack. In this subsection, you’ll create a subnet for your tenant. This can be thought of as an ISO Layer 3 (L3) provisioning of the tenant. You might be thinking to yourself, “What are you talking about? You can’t just provision L3 services on the network!” or “I already have L3 services centralized in my data center. I don’t want OpenStack to do this for me!” By the end of this section, or perhaps by the end of the book, you’ll have your own answers to these questions. For the time being, just trust that OpenStack offers benefits that are either enriched by these features or that are not possible without them.

What does it mean to create a new subnet for a specific network? Basically, you describe the network you want to work with, and then you describe the address ranges you plan to use on that network. In this case, you’ll assign the new subnet to the `ADMIN_NETWORK`, in the `ADMIN` tenant. You must also provide an address range for the subnet. You can use your own address range as long as it doesn’t exist in the tenant or a shared tenant. One of the interesting things about OpenStack is that through the use of Linux namespaces, you could use the same address range for every internal subnet in every tenant.

Enter the command in the following listing.

Listing 6.27 Creating an internal subnet for the network

```
$ neutron subnet-create \    ❶
--tenant-id 55bd141d9a29489d938bb492a1b2884c \    ❷
INTERNAL_NETWORK 172.16.0.0/24    ❸
Created a new subnet:
+------------------+------------------------------------------------+
| Field            | Value                                          |
+------------------+------------------------------------------------+
| allocation_pools | {"start": "172.16.0.2", "end": "172.16.0.254"} |
| cidr             | 172.16.0.0/24                                  |
| dns_nameservers  |                                                |
| enable_dhcp      | True                                           |
| gateway_ip       | 172.16.0.1                                     |
| host_routes      |                                                |
| id               | eb0c84d3-ea66-437f-9d1a-9defe8cccd06           |
| ip_version       | 4                                              |
| name             |                                                |
| network_id       | 5b04a1f2-1676-4f1e-a265-adddc5c589b8           |
| tenant_id        | 55bd141d9a29489d938bb492a1b2884c               |
+------------------+------------------------------------------------+
```

❶ Creates new subnet
❷ Specifies admin tenant-id
❸ Specifies network name and subnet range

First you tell OpenStack Networking (Neutron) to create a new subnet ❶. Then you specify the `admin tenant-id` on the command line ❷. Finally you specify the name of the network where the subnet should be created and the subnet range to be used on the internal network in CIDR notation❸. Don't forget, if you need to find the `admin tenant-id`, use the Keystone `tenant-id` command.

You now have a new subnet assigned to your `INTERNAL_NETWORK`. Figure 3.17 illustrates the assignment of the subnet to the `INTERNAL_NETWORK`. Unfortunately, this subnet is still isolated, but you're one step closer to connecting your private network to a public network.

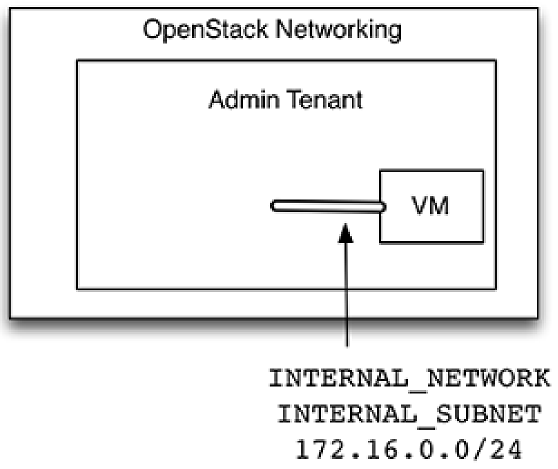


Figure 6.8 Created internal subnet

In the next subsection, you'll add a router to the subnet you just created. Make a note of your `subnet-id`—it will be needed in the following sections.

NOTE **CIDR notation**
As previously mentioned, CIDR is a compact way to represent subnets. For internal subnets, it's common to use a private class C address range. One of the most commonly used private ranges for internal or private networks is 192.168.0.0/24, which provides the range 192.168.0.1–192.168.0.254.

ROUTERS

Routers, put simply, route traffic between interfaces. In this case, you have an isolated network on your tenant and you want to be able to communicate with other tenant networks or networks outside of OpenStack. The following listing shows you how to create a new tenant router.

Listing 6.28 Create router

```
$ neutron router-create \    ❶
--tenant-id 55bd141d9a29489d938bb492a1b2884c \    ❷
ADMIN_ROUTER    ❸
Created a new router:
+-----------------------+--------------------------------------+
| Field                 | Value                                |
+-----------------------+--------------------------------------+
| admin_state_up        | True                                 |
| external_gateway_info |                                      |
| id                    | 5d7f2acd-cfc4-41bd-b5be-ba6d8e04f1e9 |
| name                  | ADMIN_ROUTER                         |
| status                | ACTIVE                               |
| tenant_id             | 55bd141d9a29489d938bb492a1b2884c     |
+-----------------------+--------------------------------------+
```

❶ Creates new router
❷ Specifies admin tenant-id
❸ Specifies router name

First, you tell OpenStack Networking (Neutron) to create a new router ❶. Then, you specify the `admin tenant-id` on the command line ❷. Finally, you specify the name of the router ❸.

Figure 3.17 illustrates the router you created in your tenant.

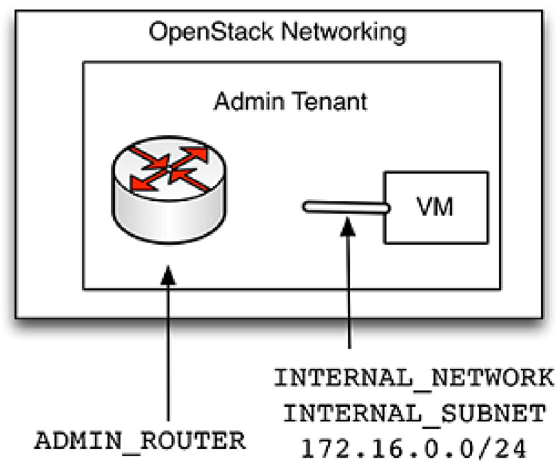


Figure 6.9 Created internal router

Now you have a new router, but your tenant router and subnet aren’t connected. The next listing shows how to connect your subnet to your router.

Listing 6.29 Adding router to internal subnet

```
$ neutron router-interface-add \     ❶
5d7f2acd-cfc4-41bd-b5be-ba6d8e04f1e9 \     ❷
eb0c84d3-ea66-437f-9d1a-9defe8cccd06     ❸

Added interface 54f0f944-06ce-4c04-861c-c059bc38fe59
    to router 5d7f2acd-cfc4-41bd-b5be-ba6d8e04f1e9.
```

❶ Adds internal subnet
❷ Specifies router-id
❸ Specifies subnet-id

First, you tell OpenStack Networking (Neutron) to add an internal subnet to your router ❶. Then, you specify the `router-id` of the router ❷. Finally, you specify the `subnet-id` of the subnet ❸.

If you need to look up Neutron-associated object IDs, you can access the Neutron console by running the Neutron CLI application without arguments: `neutron`. Once in the Neutron console, you can use the `help` command to navigate through the commands.

Figure 3.17 illustrates your router, `ADMIN_ROUTER`, connected to your internal network, `INTERNAL_NETWORK`.

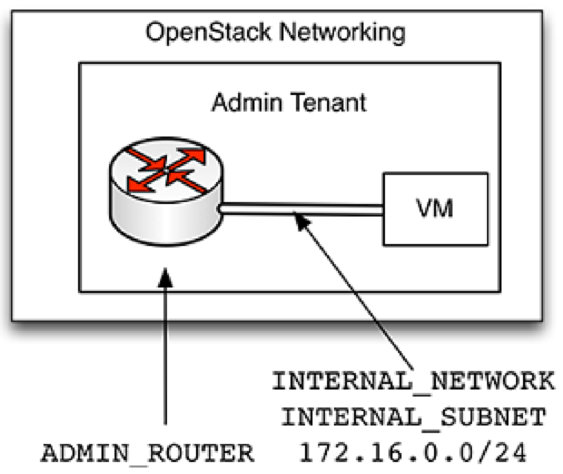


Figure 6.10 Router connected router to internal network

The process of adding a router to a subnet will actually create a *port* on the local virtual switch. You can think of a port on a virtual switch the same way you'd think of a port on a physical switch. In this case, the device is the `ADMIN_ROUTER`, the network is `INTERNAL_NETWORK`, and the subnet is `172.16.0.0/24`. The router will use the address specified during subnet creation (it defaults to first available address). When you create an instance (VM), you should be able to communicate with the router address on the 172.16.0.1 address, but you won't yet be able to route packets to external networks.

NOTE **DHCP agent**
In past versions of OpenStack Networking, you had to manually add DHCP agents to your network. The DHCP agent is used to provide your instances with an IP address. In current versions, the agent is automatically added the first time you create an instance. In advanced configurations, however, it's still helpful to know that agents (of all kinds) can be manipulated through Neutron.

A router isn't much good when it's only connected to one network, so your next step is to create a public network that can be connected to the router you just created.

EXTERNAL NETWORK

In the subsection "Internal networks," you created a network that was specifically for your tenant. Here you'll create a public network that can be used by multiple tenants. This public network can be attached to a private router and will function as a network gateway for the internal network created in the previous section.

Only the `admin` user can create external networks. If a tenant isn't specified, the new external network will be created in the `admin` tenant. Create a new external network as shown in the next listing.

Listing 6.30 Create external network

```
neutron net-create \     ❶
PUBLIC_NETWORK  ❷
--router:external=True    ❸Created a new network:
+---------------------------+-------------------------------------+
| Field                     | Value                               |
+---------------------------+-------------------------------------+
| admin_state_up            | True                                |
| id                        | 64d44339-15a4-4231-95cc-ee04bffbc459 |
| name                      | PUBLIC_NETWORK                      |
| provider:network_type     | gre                                 |
| provider:physical_network |                                     |
| provider:segmentation_id  | 2                                   |
| router:external           | True                                |
| shared                    | False                               |
| status                    | ACTIVE                              |
| subnets                   |                                     |
| tenant_id                 | 55bd141d9a29489d938bb492a1b2884c    |
+---------------------------+-------------------------------------+
```

❶ Creates a new network
❷ Specifies network name
❸ Designates as external network

First, you tell Neutron to create a new network ❶ and you specify the network name ❷. Then, you designate this network as an external network ❸.

You now have a network designated as an external network. As shown in figure 3.17, this network will reside in the `admin` tenant.

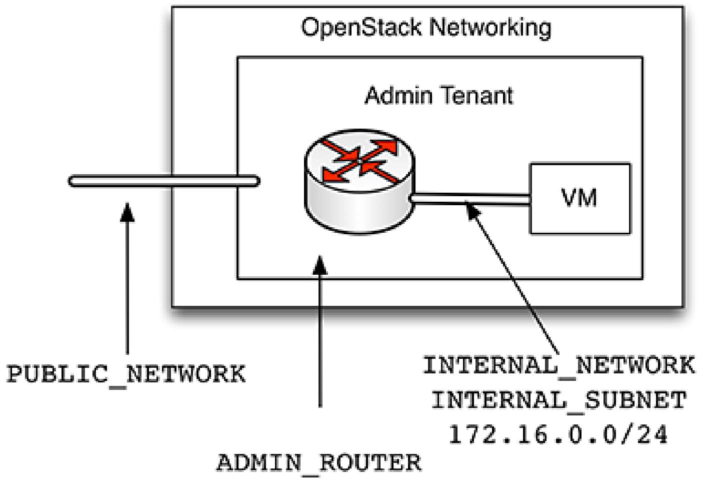


Figure 6.11 Created external network

Before you can use this network as a gateway for your tenant router (as shown in the subsection "Routers") you must first add a subnet to the external network you just created. That's what you'll do next.

EXTERNAL SUBNET

You must now create an external subnet, as shown in the following listing.

Listing 6.31 Create external subnet

```
neutron subnet-create \    ❶
--gateway 192.168.2.1 \    ❷
--allocation-pool start=192.168.2.100,end=192.168.2.250 \    ❸
PUBLIC_NETWORK \    ❹
192.168.2.0/24 \    ❺
--enable_dhcp=False    ❻

Created a new subnet:
+------------------+----------------------------------------------------+
| Field            | Value                                              |
+------------------+----------------------------------------------------+
| allocation_pools | {"start": "192.168.2.100", "end": "192.168.2.250"} |
| cidr             | 192.168.2.0/24                                     |
| dns_nameservers  |                                                    |
| enable_dhcp      | False                                              |
| gateway_ip       | 192.168.2.1                                        |
| host_routes      |                                                    |
| id               | ee91dd59-2673-4bce-8954-b6cedbf8e920               |
| ip_version       | 4                                                  |
| name             |                                                    |
| network_id       | 64d44339-15a4-4231-95cc-ee04bffbc459               |
| tenant_id        | 55bd141d9a29489d938bb492a1b2884c                   |
+------------------+----------------------------------------------------+
```

❶ Creates new subnet
❷ Sets gateway address
❸ Sets address range
❹ Defines external network
❺ Defines subnet
❻ Don't provide DHCP services

You first tell Neutron to create a new subnet ❶. You set the gateway address to the first available address ❷ and then define the range of addresses available for allocation in the subnet ❸. You then define the external network where the subnet will be assigned ❹. In CIDR format, you define the subnet ❺. Finally, you specify that OpenStack should not provide DHCP services for this subnet ❻.

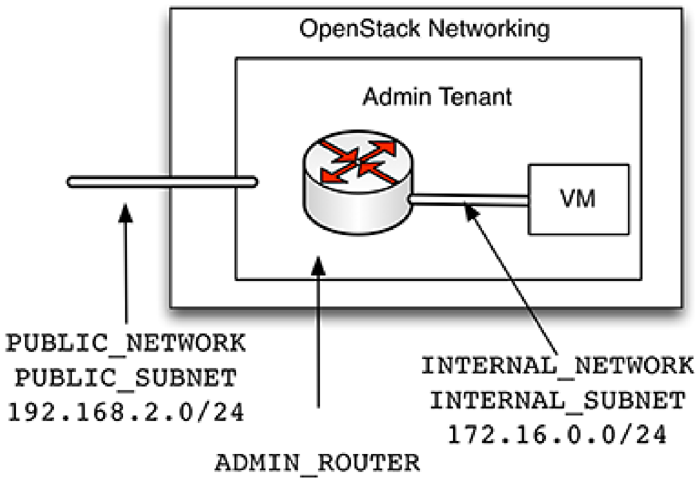


Figure 6.12 Created external subnet

In figure 3.17, you can see that you now have the subnet 192.168.2.0/24 assigned to the external network `PUBLIC_NETWORK`. The subnet and external network you just created can now be used by an OpenStack Networking router as a gateway network. In the next step, you'll assign your newly created external network as the gateway address of your internal network.

SIDEBAR **List routers to obtain router-id**

To list all the routers in the system, you can use the `neutron router-list` command:

```
devstack@devstack:~/devstack$ neutron router-list
+--------+---------------+-----------------------+
| id     | name          | external_gateway_info  |
+--------+---------------+-----------------------+
| 5d..e9 | ADMIN_ROUTER  | null                   |
+--------+---------------+-----------------------+
```

You can assign an external subnet as a gateway as follows.

Listing 6.32 Add new external network as router gateway

```
neutron router-gateway-set \    ❶
5d7f2acd-cfc4-41bd-b5be-ba6d8e04f1e9 \    ❷
64d44339-15a4-4231-95cc-ee04bffbc459    ❸

Set gateway for router
15d7f2acd-cfc4-41bd-b5be-ba6d8e04f1e9
```

❶ Uses router-gateway-set command
❷ Specifies router-id
❸ Specifies external-network-id

Figure 3.17 illustrates the assignment of the `PUBLIC_NETWORK` network as the gateway for the `ADMIN_ROUTER` in the `ADMIN` tenant. You can confirm this setting by running the command `neutron router-show <router-id>`, where the `<router-id>` is the ID of the `ADMIN_ROUTER`. The command will return the `external_gateway_info`, which lists the currently assigned gateway network. Optionally, you can log in to the OpenStack Dashboard and look at your tenant network.

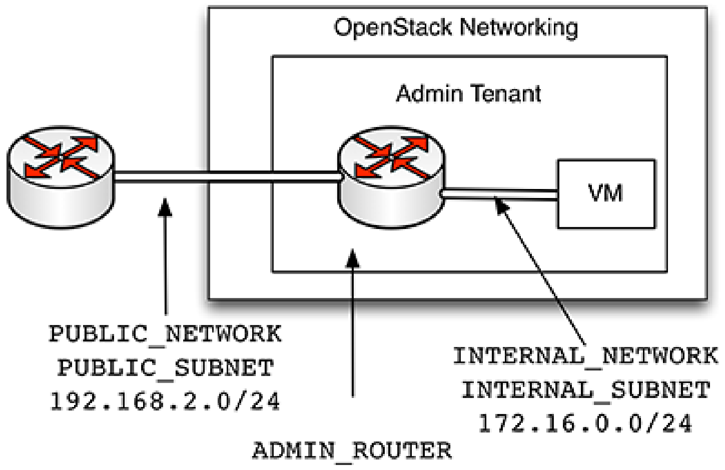


Figure 6.13 Assigned public network as router gateway

6.2.9 Relating Linux, OVS, and Neutron

At this point you should have a working Neutron environment and even a functioning network or two. But something will inevitably break and you'll need to troubleshoot the problem. Naturally, you'll turn up the log level in the suspected Neutron component. If you're lucky, there will be an obvious error. If you're not so lucky, there could be a problem with the underlying systems that Neutron depends on. Throughout this chapter, those dependencies and component relations have been explained. In many cases, you've created networks that make use of Linux namespaces, which you might not be used to working with. Now you'll work with Linux namespaces to relate the components you created on the network and systems layers.

Start by looking at the Linux network namespaces on the Neutron node:

```
$ sudo ip netns list
qrouter-5d7f2acd-cfc4-41bd-b5be-ba6d8e04f1e9
```

This result suggests you should look at the namespace `qrouter-5d7f2acd-cfc4-41bd-b5be-ba6d8e04f1e9`. Referencing the namespace, you'll display all network interface adapters:

```
sudo ip netns exec qrouter-5d7f2acd-cfc4-41bd-b5be-ba6d8e04f1e9\
 ifconfig -a

qg-896674d7-52 Link encap:Ethernet  HWaddr fa:16:3e:3b:fd:28     ❶
          inet addr:192.168.2.100  Bcast:192.168.2.255  Mask:255.255.255.0
          inet6 addr: fe80::f816:3eff:fe3b:fd28/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:738 (738.0 B)

qr-54f0f944-06 Link encap:Ethernet  HWaddr fa:16:3e:e7:f3:35     ❷
          inet addr:172.16.0.1  Bcast:172.16.0.255  Mask:255.255.255.0
          inet6 addr: fe80::f816:3eff:fee7:f335/64 Scope:Link
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:738 (738.0 B)
```

❶ Interface qg-896674d7-52
❷ Interface qr-54f0f944-06

Whether you knew it or not, this feature has been lurking in your Linux distribution for some time. Notice that the interface `qg-896674d7-52` ❶ has the same address range as the Neutron `PUBLIC_INTERFACE`, and the interface `qr-54f0f944-06` ❷ has the same range as the Neutron `INTERNAL_INTERFACE`. In fact, these are the router interfaces for their respective networks.

SIDEBAR **Working with Linux network namespaces**

To work with network namespaces, you must preface each command with `ip netns <function> <namespace_id>`:

```
sudo ip netns <function> <namespace_id> <command>
```

For more information about `ip netns`, consult the online man pages (which list it as "ip-netns"): http://man7.org/linux/man-pages/man8/ip-netns.8.html.

OK. You have some interfaces in a namespace, and these interfaces are related to the router interfaces you created earlier in the chapter. At some point, you'll want to communicate either between VM instances on OpenStack Neutron networks or with networks external to OpenStack Neutron. This is where OVS comes in.

Take a look at your OVS instance:

```
$ sudo ovs-vsctl show
    Bridge br-int
...
        Port "qr-54f0f944-06"
            tag: 1
            Interface "qr-54f0f944-06"
                type: internal
...
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
        Port "p2p1"
            Interface "p2p1"
        Port "qg-896674d7-52"
            Interface "qg-896674d7-52"
                type: internal
...
```

Some things have been added to OVS since you last saw it in listing 6.33. Notice that the interface `qr-54f0f944-06` shows up as `Port "qr-54f0f944-06"` on the internal bridge, `br-int`. Likewise, the interface `qg-896674d7-52` shows up as `Port "qg-896674d7-52"` on the external bridge, `be-ex`.

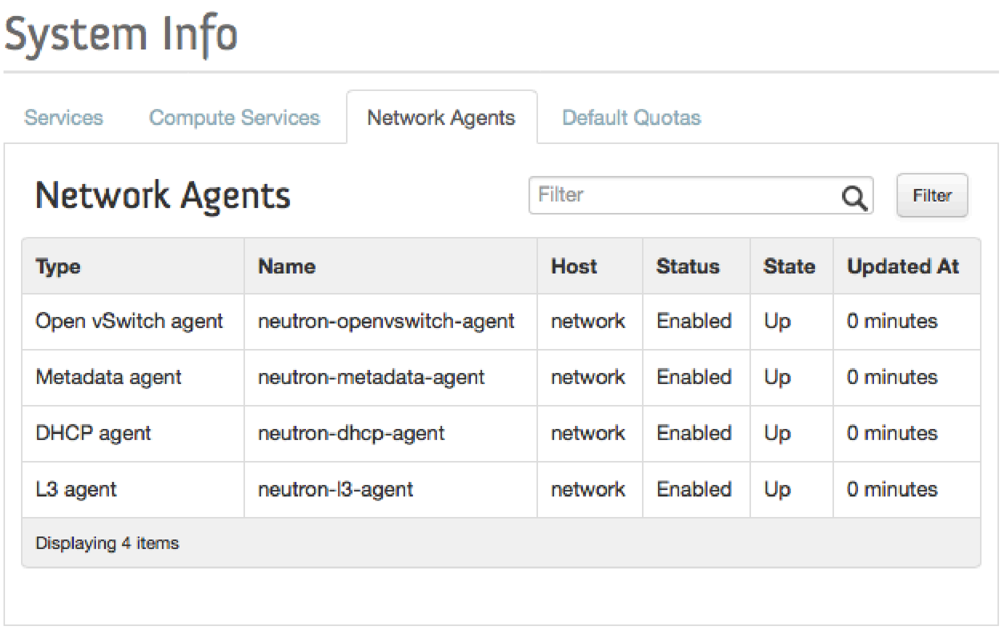
What does this mean? The external interface of the router in your configuration is on the same bridge, `br-ex`, as the physical interface, `p2p1`. This means that the OpenStack Neutron network `PUBLIC_NETWORK` will use the physical interface `br-ex` to communicate with networks external to OpenStack.

Now that all of the pieces are tied together, you can move on to the next section, where you can graphically admire your newly created networks.

6.2.10 Checking Horizon

In chapter 5 you deployed the OpenStack Dashboard. The Dashboard should now be available at http://<controller address>/horizon.

It’s a good idea to log in at this point to make sure that components are reported in the Dashboard. Log in as `admin` with the password `openstack1`. Once logged in to Horizon, select the Admin tab on the left toolbar. Next, click System Info and look under the Network Agents tab, which should look similar to figure 6.16. If you followed the instructions in the previous sections, your network should be visible in the Dashboard.



System Info

Services | Compute Services | Network Agents | Default Quotas

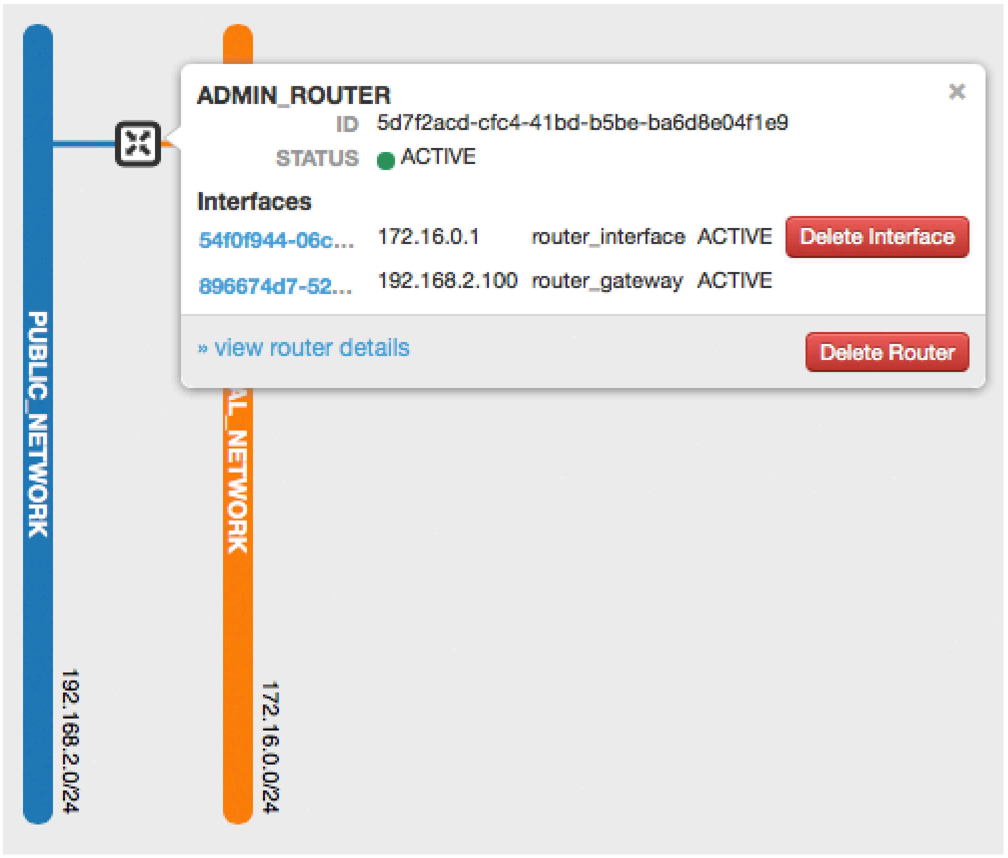
Network Agents

Type	Name	Host	Status	State	Updated At
Open vSwitch agent	neutron-openvswitch-agent	network	Enabled	Up	0 minutes
Metadata agent	neutron-metadata-agent	network	Enabled	Up	0 minutes
DHCP agent	neutron-dhcp-agent	network	Enabled	Up	0 minutes
L3 agent	neutron-l3-agent	network	Enabled	Up	0 minutes

Displaying 4 items

Figure 6.14 Dashboard System Info

Now make sure you're in the `admin` tenant and select the Project tab on the left toolbar. Next, click Network and then Network Topology. Your Network Topology screen should look like figure 6.15.



ADMIN_ROUTER

ID 5d7f2acd-cfc4-41bd-b5be-ba6d8e04f1e9

STATUS ● ACTIVE

Interfaces

54f0f944-06c...	172.16.0.1	router_interface	ACTIVE
896674d7-52...	192.168.2.100	router_gateway	ACTIVE

Figure 6.15 Network topology of PUBLIC/INTERNAL/ADMIN network

The figure shows your public network, tenant router, and tenant network in relation to your tenant. If you've made it to this screen, you've successfully manually deployed your network node.

6.3 Summary

- A separate physical network interface will be used for VM traffic.
- Neutron nodes function as routers and switches.
- Open vSwitch can be used to enable advanced switching features on a typical server.
- Network routing is included as part of the Linux kernel.
- Overlay networks use GRE, VXLAN, and other such tunnels to connect endpoints like VMs and other Neutron router instances.
- OpenStack Networking can be configured to build overlay networks for communication between VMs on separate hypervisors.
- Neutron provides both OSI L2 and L3 services.
- Neutron agents can be configured to provide DHCP, Metadata, and other services on Neutron networks.
- Neutron can be configured to use Linux networking namespaces in conjunction with OVS to provide a fully virtualized network environment.
- Internally, all tenants can use the same network IP ranges without conflict, because they're separated by using Linux namespaces.
- Neutron routers are used to route traffic between internal and external Neutron networks.