Red Hat OpenStack Platform 16.1

Networking Guide

An advanced guide to Red Hat OpenStack Platform Networking

Last Updated: 2020-10-18

Red Hat OpenStack Platform 16.1 Networking Guide

An advanced guide to Red Hat OpenStack Platform Networking

OpenStack Team
rhos-docs@redhat.com

Abstract

A cookbook for common OpenStack Networking tasks.

Table of Contents

PREFACE

The OpenStack Networking service (codename *neutron*) is the software-defined networking component of Red Hat OpenStack Platform 16.1.

Software-defined networking (SDN)

Network administrators can use software-defined networking (SDN) to manage network services through abstraction of lower-level functionality. While server workloads have been migrated into virtual environments, they are still just servers that look for a network connection to send and receive data. SDN meets this need by moving networking equipment (such as routers and switches) into the same virtualized space. If you are already familiar with basic networking concepts, then it is easy to consider that these physical networking concepts have now been virtualized, just like the servers that they connect.

Topics covered in this book

- **Preface** - Offers a brief definition of software-defined networking (SDN).
- **Part 1** - Covers common administrative tasks and basic troubleshooting steps:
  - Adding and removing network resources
  - Troubleshooting basic networks
  - Troubleshooting tenant networks
- **Part 2** - Contains cookbook-style scenarios for advanced Red Hat OpenStack Platform Networking features, including:
  - Configuring Layer 3 High Availability for virtual routers
  - Configuring DVR and other networking features

CHAPTER 1. NETWORKING OVERVIEW

1.1. HOW NETWORKING WORKS

The term networking refers to the act of moving information from one computer to another. At the most basic level, this is performed by running a cable between two machines, each with network interface cards (NICs) installed. In the OSI networking model, the cable represents layer 1.

Now, if you want more than two computers to get involved in the conversation, you would need to scale out this configuration by adding a device called a switch. Enterprise switches have multiple Ethernet ports where you can connect additional machines. A network of multiple machines is called a Local Area Network (LAN).

Because they increase complexity, switches represent another layer of the OSI model, layer two. Each NIC has a unique MAC address number assigned to the hardware, and this number enables machines connected to the same switch to find each other. The switch maintains a list of which MAC addresses are plugged into which ports, so that when one computer attempts to send data to another, the switch knows where they are both situated, and adjusts entries in the CAM (Content Addressable Memory), which monitors of MAC-address-to-port mappings.

1.1.1. VLANs

You can use VLANs to segment network traffic for computers running on the same switch. This means that you can logically divide your switch by configuring the ports to be members of different networks — they are basically mini-LANs that you can use to separate traffic for security reasons.

For example, if your switch has 24 ports in total, you can assign ports 1-6 to VLAN200, and ports 7-18 to VLAN201. As a result, computers connected to VLAN200 are completely separate from those on VLAN201; they cannot communicate directly, and if they wanted to, the traffic must pass through a router as if they were two separate physical switches. Firewalls can also be useful for governing which VLANs can communicate with each other.

1.2. CONNECTING TWO LANS TOGETHER

If you have two LANs running on two separate switches, and you want them to share information with each other. You have two options for configuring this communication:

- **Use 802.1Q VLAN tagging to configure a single VLAN that spans across both physical switches:**
  You must connect one end of a network cable to a port on one switch, connect the other end to a port on the other switch, and then configure these ports as 802.1Q tagged ports (sometimes known as trunk ports). These two switches act as one big logical switch, and the connected computers can find each other.

  The downside to this option is scalability. You can only daisy-chain a limited number of switches until overhead becomes an issue.

- **Obtain a router and use cables to connect it to each switch:**
  The router is aware of the networks configured on both switches. Each end of the cable plugged into the switch receives an IP address, known as the default gateway for that network. A default gateway defines the destination where traffic is sent when it is clear that the destination machine is not on the same LAN as the source machine. By establishing a default gateway, each computer can send traffic to other computers without knowing specific information about the

destination. Each computer sends traffic to the default gateway, and the router determines which destination computer receives the traffic. Routing works on layer 3 of the OSI model, and is where the familiar concepts like IP addresses and subnets operate.

1.2.1. Firewalls

Firewalls can filter traffic across multiple OSI layers, including layer 7 (for inspecting actual content). Firewalls are often situated in the same network segments as routers, where they govern the traffic moving between all the networks. Firewalls refer to a predefined set of rules that prescribe which traffic can enter a network. These rules can become very granular, for example:

"Servers on **VLAN200** may only communicate with computers on **VLAN201**, and only on a Thursday afternoon, and only if they are sending encrypted web traffic (HTTPS) in one direction".

To help enforce these rules, some firewalls also perform Deep Packet Inspection (DPI) at layers 5-7, whereby they examine the contents of packets to ensure that the packets are legitimate. Hackers can exfiltrate data by having the traffic masquerade as something it is not. DPI is one of the means that you can use to mitigate that threat.

1.3. WORKING WITH OPENSTACK NETWORKING (NEUTRON)

These same networking concepts apply in OpenStack, where they are known as Software-defined networking (SDN). The OpenStack Networking (neutron) component provides the API for virtual networking capabilities, and includes switches, routers, and firewalls. The virtual network infrastructure allows your instances to communicate with each other and also externally using the physical network. The Open vSwitch bridge allocates virtual ports to instances, and can span across the network infrastructure to the physical network for incoming and outgoing traffic.

1.4. WORKING WITH CIDR FORMAT

IP addresses are generally first allocated in blocks of subnets. For example, the IP address range **192.168.100.0 - 192.168.100.255** with a subnet mask of **255.555.255.0** allows for **254** IP addresses (the first and last addresses are reserved).

These subnets can be represented in a number of ways:

- **Common usage:**
  Subnet addresses are traditionally displayed using the network address accompanied by the subnet mask:
  - Network Address: *192.168.100.0*
  - Subnet mask: *255.255.255.0*
- **CIDR format:**
  The subnet mask is shortened into its total number of active bits.

  For example, in **192.168.100.0/24**, **/24** is a shortened representation of **255.255.255.0**, and is a total of the number of flipped bits when converted to binary.

  Also, CIDR format can be used in **ifcfg-xxx** scripts instead of the **NETMASK** value:

  ```
  #NETMASK=255.255.255.0
  PREFIX=24
  ```

CHAPTER 2. OPENSTACK NETWORKING CONCEPTS

OpenStack Networking has system services to manage core services such as routing, DHCP, and metadata. Together, these services are included in the concept of the Controller node, which is a conceptual role assigned to a physical server. A physical server is typically assigned the role of *Network node* and dedicated to the task of managing Layer 3 routing for network traffic to and from instances. In OpenStack Networking, you can have multiple physical hosts performing this role, allowing for redundant service in the event of hardware failure. For more information, see the chapter on *Layer 3 High Availability*.

NOTE

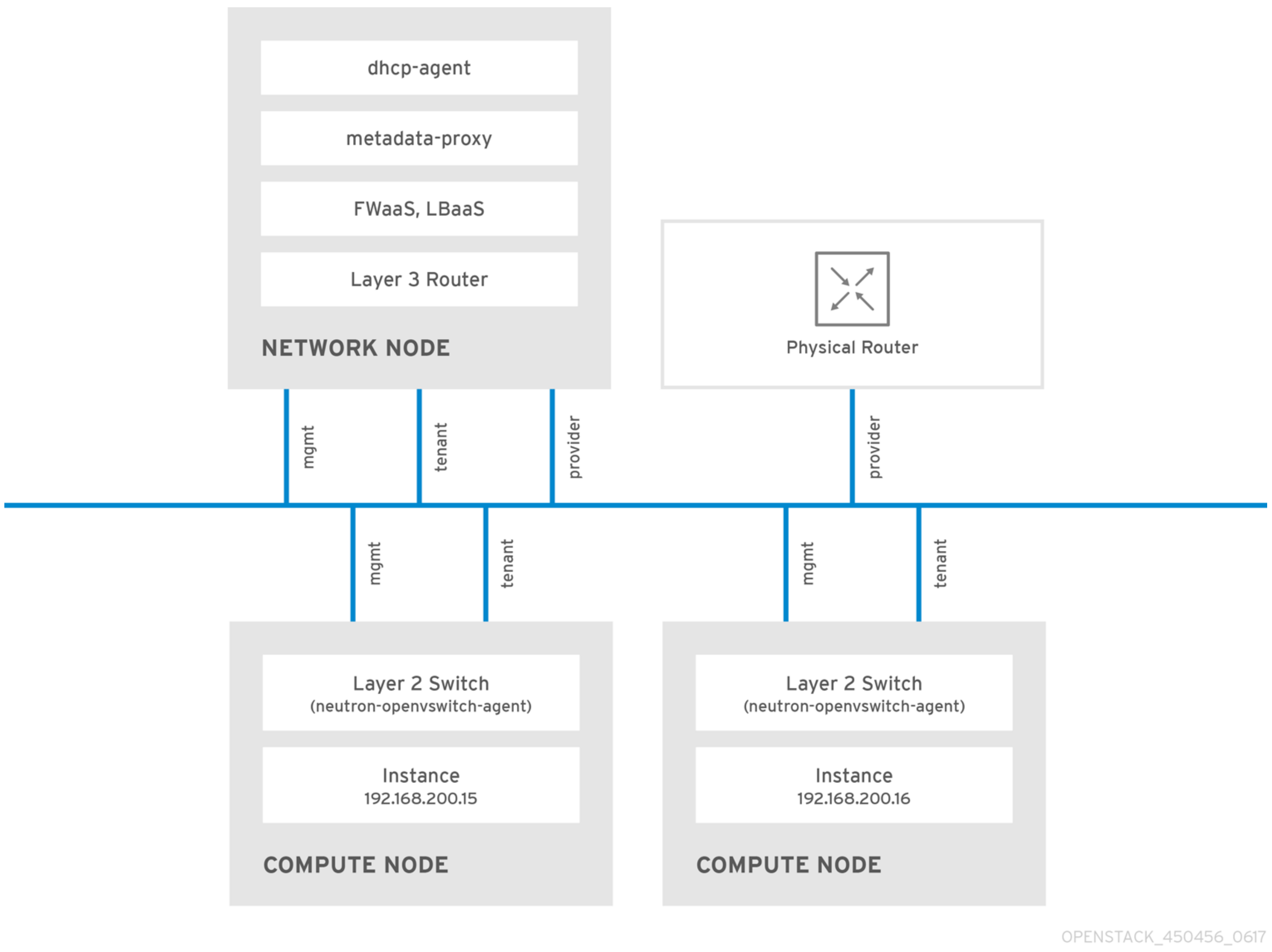
Red Hat OpenStack Platform 11 added support for composable roles, allowing you to separate network services into a custom role. However, for simplicity, this guide assumes that a deployment uses the default controller role.

2.1. INSTALLING OPENSTACK NETWORKING (NEUTRON)

The OpenStack Networking component is installed as part of a Red Hat OpenStack Platform director deployment. For more information about director deployment, see Director Installation and Usage.

2.2. OPENSTACK NETWORKING DIAGRAM

This diagram depicts a sample OpenStack Networking deployment, with a dedicated OpenStack Networking node performing layer 3 routing and DHCP, and running the advanced service load balancing as a Service (LBaaS). Two Compute nodes run the Open vSwitch (openvswitch-agent) and have two physical network cards each, one for tenant traffic, and another for management connectivity. The OpenStack Networking node has a third network card specifically for provider traffic:



2.3. SECURITY GROUPS

Security groups and rules filter the type and direction of network traffic that neutron ports send and receive. This provides an additional layer of security to complement any firewall rules present on the compute instance. The security group is a container object with one or more security rules. A single security group can manage traffic to multiple compute instances.

Ports created for floating IP addresses, OpenStack Networking LBaaS VIPs, and instances are associated with a security group. If you do not specify a security group, then the port is associated with the default security group. By default, this group drops all inbound traffic and allows all outbound traffic. However, traffic flows between instances that are members of the default security group, because the group has a remote group ID that points to itself.

To change the filtering behavior of the default security group, you can add security rules to the group, or create entirely new security groups.

2.4. OPEN VSWITCH

Open vSwitch (OVS) is a software-defined networking (SDN) virtual switch similar to the Linux software bridge. OVS provides switching services to virtualized networks with support for industry standard , OpenFlow, and sFlow. OVS can also integrate with physical switches using layer 2 features, such as STP, LACP, and 802.1Q VLAN tagging. Open vSwitch version 1.11.0-1.el6 or later also supports tunneling with VXLAN and GRE.

For more information about network interface bonds, see the Network Interface Bonding chapter of the Advanced Overcloud Customization guide.

NOTE

To mitigate the risk of network loops in OVS, only a single interface or a single bond can be a member of a given bridge. If you require multiple bonds or interfaces, you can configure multiple bridges.

2.5. MODULAR LAYER 2 (ML2) NETWORKING

ML2 is the OpenStack Networking core plug-in introduced in the OpenStack Havana release. Superseding the previous model of monolithic plug-ins, the ML2 modular design enables the concurrent operation of mixed network technologies. The monolithic Open vSwitch and Linux Bridge plug-ins have been deprecated and removed; their functionality is now implemented by ML2 mechanism drivers.

NOTE

ML2 is the default OpenStack Networking plug-in, with OVN configured as the default mechanism driver.

2.5.1. The reasoning behind ML2

Previously, OpenStack Networking deployments could use only the plug-in selected at implementation time. For example, a deployment running the Open vSwitch (OVS) plug-in was required to use the OVS plug-in exclusively. The monolithic plug-in did not support the simultaneously use of another plug-in such as linuxbridge. This limitation made it difficult to meet the needs of environments with heterogeneous requirements.

2.5.2. ML2 network types

Multiple network segment types can be operated concurrently. In addition, these network segments can interconnect using ML2 support for multi-segmented networks. Ports are automatically bound to the segment with connectivity; it is not necessary to bind ports to a specific segment. Depending on the mechanism driver, ML2 supports the following network segment types:

- flat
- GRE
- local
- VLAN
- VXLAN
- Geneve

Enable Type drivers in the ML2 section of the **ml2_conf.ini** file. For example:

```
[ml2]
type_drivers = local,flat,vlan,gre,vxlan,geneve
```

2.5.3. ML2 mechanism drivers

Plug-ins are now implemented as mechanisms with a common code base. This approach enables code reuse and eliminates much of the complexity around code maintenance and testing.

NOTE

For the list of supported mechanism drivers, see Release Notes.

The default mechanism driver is OVN. Mechanism drivers are enabled in the ML2 section of the **ml2_conf.ini** file. For example:

```
[ml2]
mechanism_drivers = ovn
```

NOTE

Red Hat OpenStack Platform director manages these settings. Do not change them manually.

2.6. ML2 TYPE AND MECHANISM DRIVER COMPATIBILITY

Mechanism Driver	Type Driver				
	flat	gre	vlan	vxlan	geneve
ovn	yes	no	yes	no	yes
openvswitch	yes	yes	yes	yes	no

2.7. LIMITS OF THE ML2/OVN MECHANISM DRIVER

Some features available with the ML2/OVS mechanism driver are not yet supported with the ML2/OVN mechanism driver.

2.7.1. ML2/OVS features not yet supported by ML2/OVN

Feature	Notes	Track this Feature
Fragmentation needed packets – ICMP/UDP Jumbo Frames	OVN does not yet support sending ICMP "fragmentation needed" packets. Larger ICMP/UDP packets that require fragmentation do not work with ML2/OVN as they would with the ML2/OVS driver implementation. TCP traffic is handled by maximum segment sized (MSS) clamping.	https://bugzilla.redhat.com/show_bug.cgi?id=1547074 (ovn-network)
Port Forwarding	OVN does not support port forwarding.	https://bugzilla.redhat.com/show_bug.cgi?id=1654608 https://blueprints.launchpad.net/neutron/+spec/port-forwarding

Feature	Notes	Track this Feature
Security Groups Logging API	ML2/OVN does not provide a log file that logs security group events such as an instance trying to execute restricted operations or access restricted ports in remote servers.	https://bugzilla.redhat.com/show_bug.cgi?id=1619266
Multicast	When using ML2/OVN as the integration bridge, multicast traffic is treated as broadcast traffic. The integration bridge operates in FLOW mode, so IGMP snooping is not available. To support this, core OVN must support IGMP snooping.	https://bugzilla.redhat.com/show_bug.cgi?id=1575512
Provisioning Baremetal Machines with OVN DHCP	The built-in DHCP server on OVN presently can not provision baremetal nodes. It cannot serve DHCP for the provisioning networks. Chainbooting iPXE requires tagging (**--dhcp-match** in dnsmasq), which is not supported in the OVN DHCP server.	https://bugzilla.redhat.com/show_bug.cgi?id=1622154
OVS_DPDK	OVS_DPDK is presently not supported with OVN.	https://bugzilla.redhat.com/show_bug.cgi?id=1844615

2.8. ML2/OVS TO ML2/OVN MIGRATION: VALIDATED AND PROHIBITED SCENARIOS

Migration of the ML2 mechanism driver is a substantial change with broad consequences in a deployment. Red Hat continues to test and refine migration scenarios. Make sure that your scenario is validated before attempting a migration.

WARNING

An ML2/OVS to ML2/OVN migration alters the environment in ways that can not be reversed without a backup and restore. A failed or interrupted migration can leave the OpenStack environment inoperable. Before migrating in a production environment, test the migration in a stage environment that closely resembles your production environment, and create a reliable backup of your deployment.

2.8.1. Validated ML2/OVS to ML2/OVN migration scenarios

DVR to DVR

Start: RHOSP 16.1.1 with OVS with DVR. Geneve project (tenant) networks.
End: RHOSP 16.1.1 with OVN with DVR. Geneve project (tenant) networks.

SR-IOV and TLS-everywhere were not present in the starting environment or added during or after the migration.

Centralized routing + SR-IOV with virtual function (VF) ports only

Start: RHOSP 16.1.1 with OVS (no DVR)and SR-IOV.
End: RHOSP 16.1.1 with OVN (no DVR) and SR-IOV.

Workloads used only SR-IOV virtual function (VF) ports. SR-IOV phyical function (PF) ports caused migration failure.

2.8.2. ML2/OVS to ML2/OVN migration scenarios that failed in tests

Do not perform the ML2/OVS to ML2/OVN migration in the following scenarios until Red Hat announces that the underlying issues are resolved.

Fast forward upgrade (FFU) from OSP13 to OSP16.1

Do not perform an ML2/OVS to ML2/OVN migration after an FFU from RHOPS13 to RHOSP16. To track progress on this issue, see https://bugzilla.redhat.com/show_bug.cgi?id=1879923.

SR-IOV with physical function (PF) ports

Migration tests failed when any workload uses an SR-IOV PF port. To track progress on this issue, see https://bugzilla.redhat.com/show_bug.cgi?id=1879546.

Transport layer security everywhere (TLS-e)

Migration tests failed when the OVS deployment had TLS-e enabled. If your OVS deployment has TLE-e enabled, do not perform an ML2/OVS to ML2/OVN migration. To track progress on this issue, see https://bugzilla.redhat.com/show_bug.cgi?id=1879097, https://bugzilla.redhat.com/show_bug.cgi?id=1877756, and https://bugzilla.redhat.com/show_bug.cgi?id=1872268.

OVS uses trunk ports

If you have an ML2/OVS deployment that uses trunk ports, do not perform an ML2/OVS to ML2/OVN migration. The migration does not properly set up the trunked ports in the OVN environment. To track progress on this issue, see https://bugzilla.redhat.com/show_bug.cgi?id=1857652.

Centralized routing and VLAN project (tenant) networks

If your ML2/OVS deployment uses centralized routing and VLAN project (tenant) networks, do not migrate to ML2/OVN with DVR. You can migrate to ML2/OVN with centralized routing. To track progress on this issue, see https://bugzilla.redhat.com/show_bug.cgi?id=1704596.

2.9. USING THE ML2/OVS MECHANISM DRIVER INSTEAD OF THE DEFAULT ML2/OVN DRIVER

If your application requires the ML2/OVS mechanism driver, you can deploy the overcloud with the environment file **neutron-ovs.yaml**, which disables the default ML2/OVN mechanism driver and enables ML2/OVS.

2.9.1. Using ML2/OVS in a new RHOSP 16.1 deployment

In the overcloud deployment command, include the environment file **neutron-ovs.yaml** as shown in the following example.

```
-e /usr/share/openstack-tripleo-heat-templates/environments/services/neutron-ovs.yaml
```

For more information about using environment files, see Including Environment Files in Overcloud Creation in the *Advanced Overcloud Customization* guide.

2.9.2. Upgrading from ML2/OVS in a previous RHOSP to ML2/OVS in RHOSP 16.1

To keep using ML2/OVS after an upgrade from a previous version of RHOSP that uses ML2/OVS, follow Red Hat's upgrade procedure as documented, and do not perform the ML2/OVS-to-ML2/OVN migration.

The upgrade procedure includes adding **-e /usr/share/openstack-tripleo-heat-templates/environments/services/neutron-ovs.yaml** to the overcloud deployment command.

2.10. CONFIGURING THE L2 POPULATION DRIVER

The L2 Population driver enables broadcast, multicast, and unicast traffic to scale out on large overlay networks. By default, Open vSwitch GRE and VXLAN replicate broadcasts to every agent, including those that do not host the destination network. This design requires the acceptance of significant network and processing overhead. The alternative design introduced by the L2 Population driver implements a partial mesh for ARP resolution and MAC learning traffic; it also creates tunnels for a particular network only between the nodes that host the network. This traffic is sent only to the necessary agent by encapsulating it as a targeted unicast.

To enable the L2 Population driver, complete the following steps:

1. Enable the L2 population driver by adding it to the list of mechanism drivers. You also must enable at least one tunneling driver: either GRE, VXLAN, or both. Add the appropriate configuration options to the ml2_conf.ini file:

```
[ml2]
type_drivers = local,flat,vlan,gre,vxlan,geneve
mechanism_drivers = l2population
```

NOTE

Neutron's Linux Bridge ML2 driver and agent were deprecated in Red Hat OpenStack Platform 11. The Open vSwitch (OVS) plugin OpenStack Platform director default, and is recommended by Red Hat for general usage.

2. Enable L2 population in the openvswitch_agent.ini file. Enable it on each node that contains the L2 agent:

```
[agent]
l2_population = True
```

NOTE

To install ARP reply flows, configure the **arp_responder** flag:

```
[agent]
l2_population = True
arp_responder = True
```

2.11. ENABLING SR-IOV WITH ML2/OVN AND NATIVE OVN DHCP

You can use SR-IOV in an ML2/OVN deployment with native OVN DHCP.

Limitations

The following limitations apply to the use of SR-IOV with ML2/OVN and native OVN DHCP in this release.

- All external ports are scheduled on a single gateway node because there is only one HA Chassis Group for all of the ports.
- Routing on VLAN tenant network does not work with SR-IOV because the external ports are not co-located with the logical router’s gateway ports. See https://bugs.launchpad.net/neutron/+bug/1875852.

Prerequisites

- You know how to deploy custom roles. For more information see Composable Services and Custom Roles in the Advanced Overcloud Customization guide.

Procedure

1. Ensure that the neutron DHCP agent is disabled in your deployment. For example, in the file **neutron-ovn-drv-ha.yaml** include the following line.

   ```
   OS::TripleO::Services::NeutronDhcpAgent: OS::Heat::None
   ```

2. Deploy the custom role that matches your node configuration.

Node Configuration	Custom role
Standalone networker and controller nodes	NetworkerSriov
Co-located networker and controller nodes	ControllerSriov

3. If you deployed NetworkerSriov in a configuration with standalone networker and controller nodes, perform the following parameter modifications in an environment file such as neutron-ovn-dvr-ha.yaml:

   a. Clear OVNCMSOptions on controller nodes.

      ```
      ControllerParameters:
        OVNCMSOptions: ""
      ```

b. Set NetworkerSriovParameters on networker nodes.

```
NetworkerSriovParameters:
   OVNCMSOptions: "enable-chassis-as-gw"
```

c. Remove NetworkerParameters from networker nodes if present.

```
NetworkerParameters:
   OVNCMSOptions: ""
```

4. Deploy the custom role ComputeSriov.

Verification steps

1. Ensure that ovn_metatdata_agent is running on controller and networker nodes.

```
[heat-admin@controller-0 ~]$ sudo podman ps | grep ovn_metadata
```

Expect output similar to the following example.

```
a65125d9588d  undercloud-0.ctlplane.localdomain:8787/rh-osbs/rhosp16-openstack-
neutron-metadata-agent-ovn:16.1_20200813.1  kolla_start          23 hours ago  Up 21 hours
ago         ovn_metadata_agent
```

2. Ensure that controller/networker nodes have been configured as gateways for OVS.

```
[heat-admin@controller-0 ~]$ sudo ovs-vsctl get Open_Vswitch .
```

Expect output similar to the following example.

```
external_ids:ovn-cms-options
   enable-chassis-as-gw
```

3. Ensure that neutron_sriov_agent is running on compute nodes.

```
[heat-admin@controller-0 ~]sudo podman ps | grep neutron_sriov_agent
```

Expect output similar to the following example.

```
f54cbbf4523a  undercloud-0.ctlplane.localdomain:8787/rh-osbs/rhosp16-openstack-neutron-
sriov-agent:16.1_20200813.1
kolla_start  23 hours ago  Up 21 hours ago         neutron_sriov_agent
```

4. Ensure that network-available SR-IOV NICs have been successfully detected.

```
[heat-admin@controller-0 ~]$ sudo podman exec -uroot galera-bundle-podman-0 mysql nova
-e 'select hypervisor_hostname,pci_stats from compute_nodes;'
```

Expect output similar to the following example.

```
computesriov-1.localdomain {... {"dev_type": "type-PF", "physical_network": "datacentre",
"trusted": "true"}, "count": 1}, ... {"dev_type": "type-VF", "physical_network": "datacentre",
"trusted": "true", "parent_ifname": "enp7s0f3"}, "count": 5}, ...}
computesriov-0.localdomain {... {"dev_type": "type-PF", "physical_network": "datacentre",
"trusted": "true"}, "count": 1}, ... {"dev_type": "type-VF", "physical_network": "datacentre",
"trusted": "true", "parent_ifname": "enp7s0f3"}, "count": 5}, ...}
```

2.12. OPENSTACK NETWORKING SERVICES

By default, Red Hat OpenStack Platform includes components that integrate with the ML2 and Open vSwitch plugin to provide networking functionality in your deployment:

2.12.1. L3 agent

The L3 agent is part of the **openstack-neutron** package. Use network namespaces to provide each project with its own isolated layer 3 routers, which direct traffic and provide gateway services for the layer 2 networks. The L3 agent assists with managing these routers. The nodes that host the L3 agent must not have a manually-configured IP address on a network interface that is connected to an external network. Instead there must be a range of IP addresses from the external network that are available for use by OpenStack Networking. Neutron assigns these IP addresses to the routers that provide the link between the internal and external networks. The IP range that you select must be large enough to provide a unique IP address for each router in the deployment as well as each floating IP.

2.12.2. DHCP agent

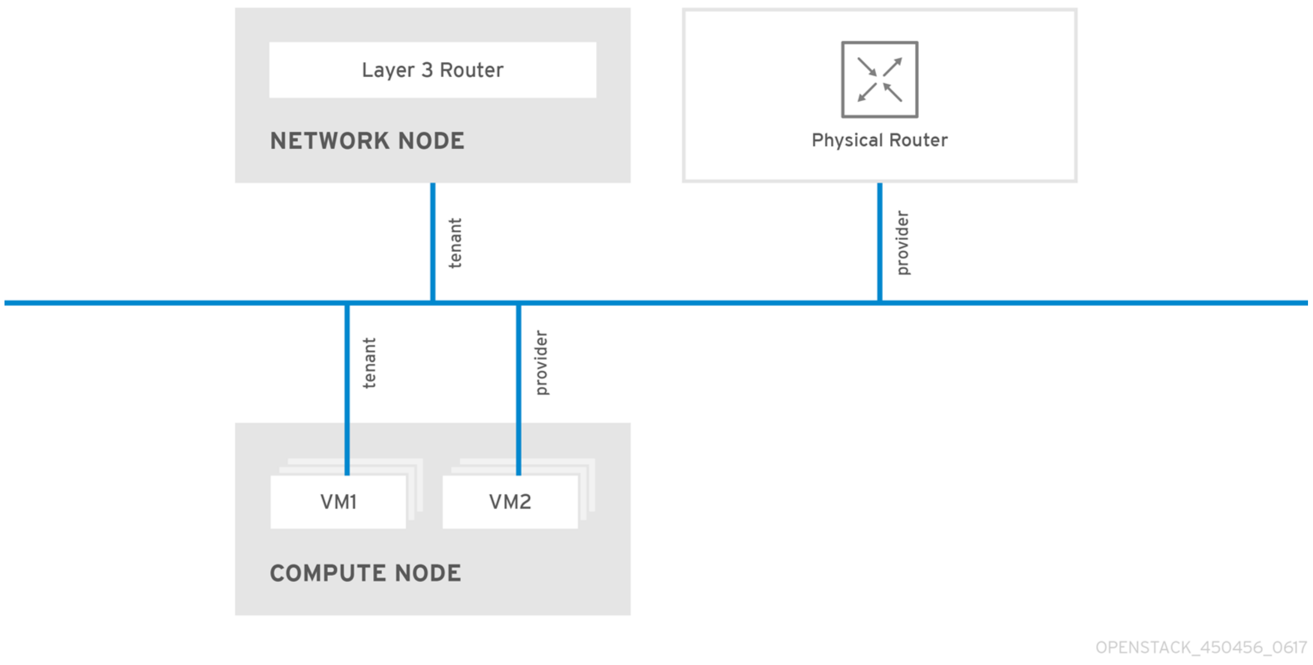
The OpenStack Networking DHCP agent manages the network namespaces that are spawned for each project subnet to act as DHCP server. Each namespace runs a dnsmasq process that can allocate IP addresses to virtual machines on the network. If the agent is enabled and running when a subnet is created then by default that subnet has DHCP enabled.

2.12.3. Open vSwitch agent

The Open vSwitch (OVS) neutron plug-in uses its own agent, which runs on each node and manages the OVS bridges. The ML2 plugin integrates with a dedicated agent to manage L2 networks. By default, Red Hat OpenStack Platform uses **ovs-agent**, which builds overlay networks using OVS bridges.

2.13. TENANT AND PROVIDER NETWORKS

The following diagram presents an overview of the tenant and provider network types, and illustrates how they interact within the overall OpenStack Networking topology:



2.13.1. Tenant networks

Users create tenant networks for connectivity within projects. Tenant networks are fully isolated by default and are not shared with other projects. OpenStack Networking supports a range of tenant network types:

- **Flat** - All instances reside on the same network, which can also be shared with the hosts. No VLAN tagging or other network segregation occurs.
- **VLAN** - OpenStack Networking allows users to create multiple provider or tenant networks using VLAN IDs (802.1Q tagged) that correspond to VLANs present in the physical network. This allows instances to communicate with each other across the environment. They can also communicate with dedicated servers, firewalls, load balancers and other network infrastructure on the same layer 2 VLAN.
- **VXLAN and GRE tunnels**- VXLAN and GRE use network overlays to support private communication between instances. An OpenStack Networking router is required to enable traffic to traverse outside of the GRE or VXLAN tenant network. A router is also required to connect directly-connected tenant networks with external networks, including the Internet; the router provides the ability to connect to instances directly from an external network using floating IP addresses. VXLAN and GRE type drivers are compatible with the ML2/OVS mechanism driver.
- **GENEVE tunnels** - GENEVE recognizes and accommodates changing capabilities and needs of different devices in network virtualization. It provides a framework for tunneling rather than being prescriptive about the entire system. Geneve defines the content of the metadata flexibly that is added during encapsulation and tries to adapt to various virtualization scenarios. It uses UDP as its transport protocol and is dynamic in size using extensible option headers. Geneve supports unicast, multicast, and broadcast. The GENEVE type driver is compatible with the ML2/OVN mechanism driver.

NOTE

You can configure QoS policies for tenant networks. For more information, see *Chapter 10, Configuring Quality of Service (QoS) policies* .

2.13.2. Provider networks

The OpenStack administrator creates provider networks. Provider networks map directly to an existing physical network in the data center. Useful network types in this category include flat (untagged) and VLAN (802.1Q tagged). You can also share provider networks among tenants as part of the network creation process.

2.13.2.1. Flat provider networks

You can use flat provider networks to connect instances directly to the external network. This is useful if you have multiple physical networks (for example, *physnet1* and *physnet2*) and separate physical interfaces (*eth0* → *physnet1* and *eth1* → *physnet2*), and intend to connect each Compute and Network node to those external networks. To use multiple vlan-tagged interfaces on a single interface to connect to multiple provider networks, see Section 7.3, "Using VLAN provider networks" .

2.13.2.2. Configuring networking for Controller nodes

1. Edit */etc/neutron/plugin.ini* (symbolic link to */etc/neutron/plugins/ml2/ml2_conf.ini*) to add **flat** to the existing list of values and set **flat_networks** to *****. For example:

```
type_drivers = vxlan,flat
flat_networks =*
```

2. Create an external network as a flat network and associate it with the configured *physical_network*. Configure it as a shared network (using **--share**) to let other users create instances that connect to the external network directly.

```
# openstack network create --share --provider-network-type flat --provider-physical-network physnet1
--external public01
```

3. Create a subnet using the **openstack subnet create** command, or the dashboard. For example:

```
# openstack subnet create --no-dhcp --allocation-pool start=192.168.100.20,end=192.168.100.100 --
gateway 192.168.100.1 --network public01 public_subnet
```

4. Restart the **neutron-server** service to apply the change:

```
systemctl restart tripleo_neutron_api
```

2.13.2.3. Configuring networking for the Network and Compute nodes

Complete the following steps on the Network node and Compute nodes to connect the nodes to the external network, and allow instances to communicate directly with the external network.

1. Create an external network bridge (br-ex) and add an associated port (eth1) to it:

Create the external bridge in */etc/sysconfig/network-scripts/ifcfg-br-ex*:

```
DEVICE=br-ex
TYPE=OVSBridge
DEVICETYPE=ovs
```

```
ONBOOT=yes
NM_CONTROLLED=no
BOOTPROTO=none
```

In */etc/sysconfig/network-scripts/ifcfg-eth1*, configure **eth1** to connect to **br-ex**:

```
DEVICE=eth1
TYPE=OVSPort
DEVICETYPE=ovs
OVS_BRIDGE=br-ex
ONBOOT=yes
NM_CONTROLLED=no
BOOTPROTO=none
```

Reboot the node or restart the network service for the changes to take effect.

2. Configure physical networks in */etc/neutron/plugins/ml2/openvswitch_agent.ini* and map bridges to the physical network:

```
bridge_mappings = physnet1:br-ex
```

NOTE

For more information on bridge mappings, see Chapter 11, *Configuring bridge mappings*.

3. Restart the *neutron-openvswitch-agent* service on both the network and compute nodes to apply the changes:

```
systemctl restart neutron-openvswitch-agent
```

2.13.2.4. Configuring the Network node

1. Set **external_network_bridge =** to an empty value in */etc/neutron/l3_agent.ini*:

Setting **external_network_bridge =** to an empty value allows multiple external network bridges. OpenStack Networking creates a patch from each bridge to **br-int**.

```
external_network_bridge =
```

2. Restart **neutron-l3-agent** for the changes to take effect.

```
systemctl restart neutron-l3-agent
```

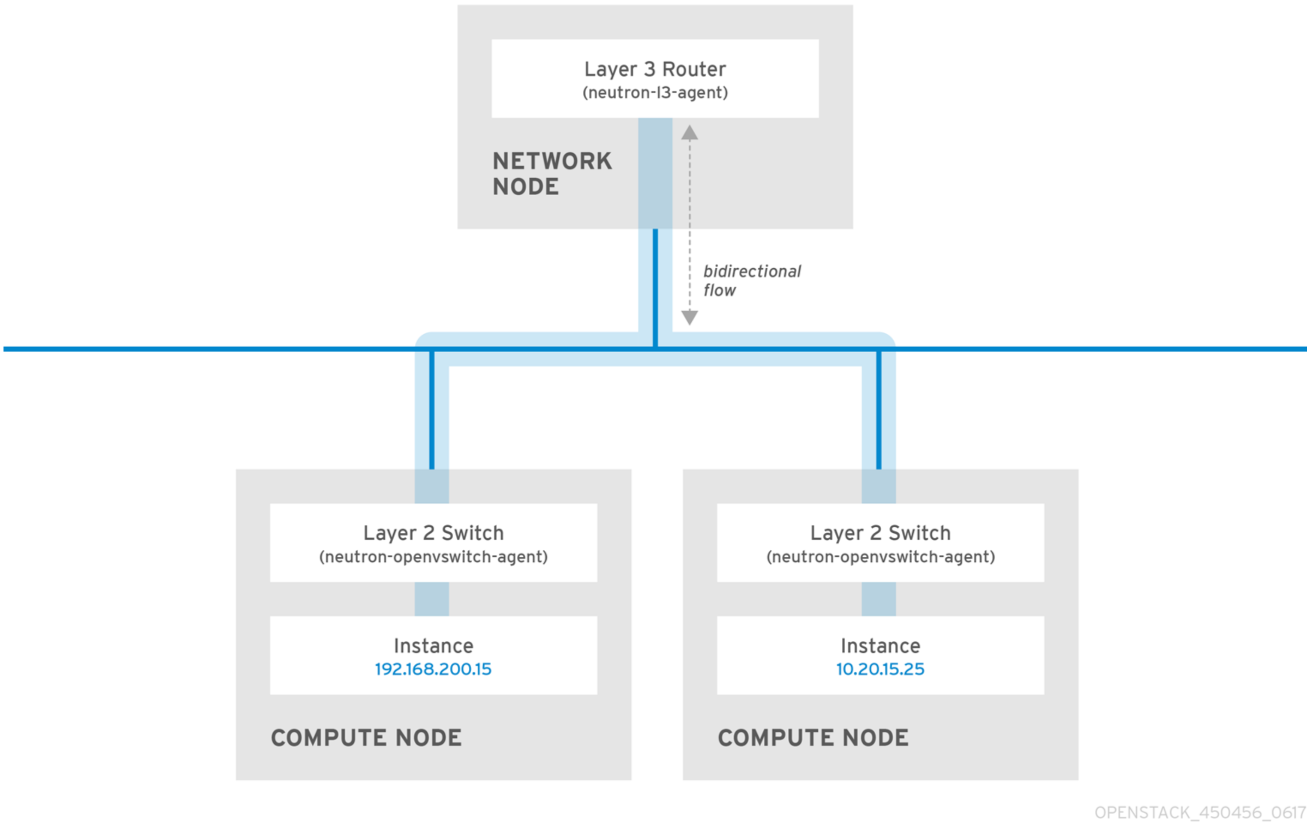
NOTE

If there are multiple flat provider networks, then each of them must have a separate physical interface and bridge to connect them to the external network. Configure the **ifcfg-*** scripts appropriately and use a comma-separated list for each network when specifying the mappings in the **bridge_mappings** option. For more information on bridge mappings, see Chapter 11, *Configuring bridge mappings*.

2.14. LAYER 2 AND LAYER 3 NETWORKING

When designing your virtual network, anticipate where the majority of traffic is going to be sent. Network traffic moves faster within the same logical network, rather than between multiple logical networks. This is because traffic between logical networks (using different subnets) must pass through a router, resulting in additional latency.

Consider the diagram below which has network traffic flowing between instances on separate VLANs:

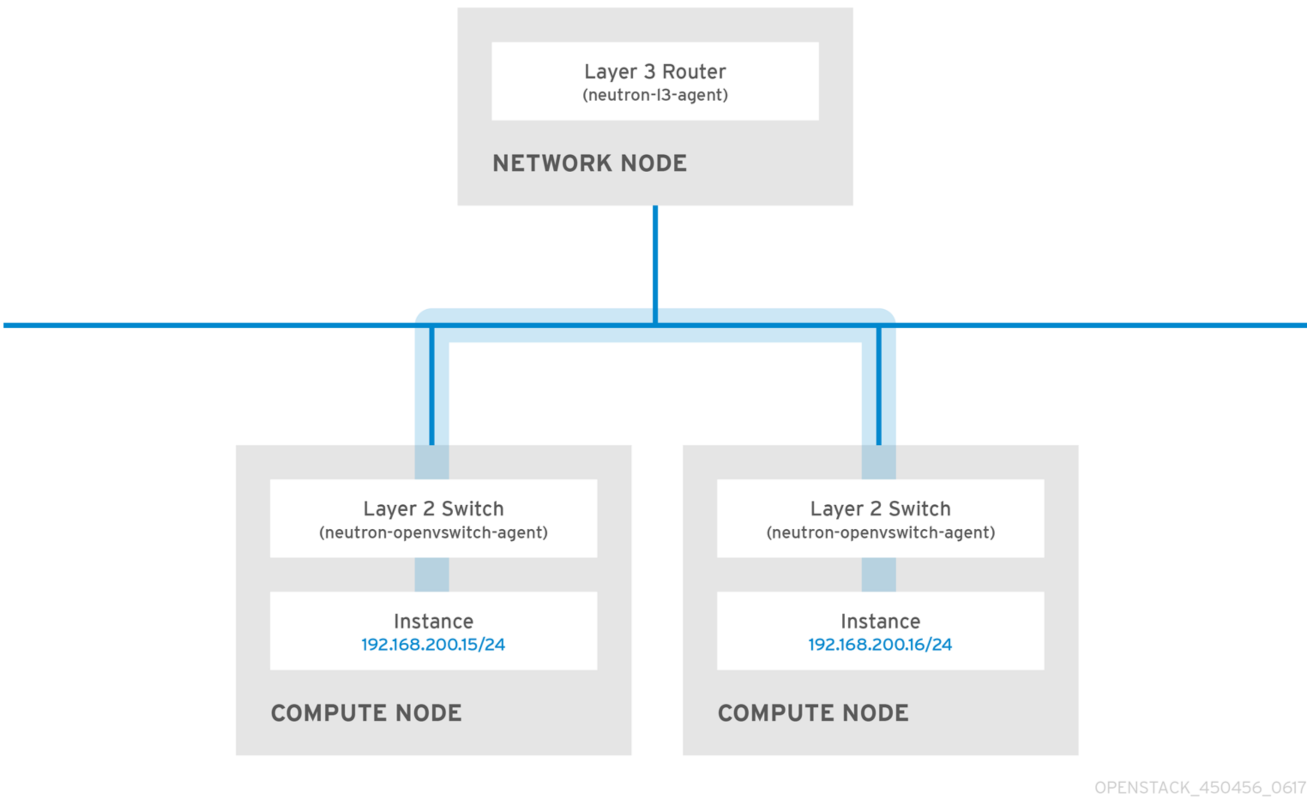


NOTE

Even a high performance hardware router adds latency to this configuration.

2.14.1. Use switching where possible

Because switching occurs at a lower level of the network (layer 2) it can function faster than the routing that occurs at layer 3. Design as few hops as possible between systems that communicate frequently. For example, the following diagram depicts a switched network that spans two physical nodes, allowing the two instances to communicate directly without using a router for navigation first. Note that the instances now share the same subnet, to indicate that they are on the same logical network:



To allow instances on separate nodes to communicate as if they are on the same logical network, use an encapsulation tunnel such as VXLAN or GRE. Red Hat recommends adjusting the MTU size from end-to-end to accommodate the additional bits required for the tunnel header, otherwise network performance can be negatively impacted as a result of fragmentation. For more information, see *Configure MTU Settings*.

You can further improve the performance of VXLAN tunneling by using supported hardware that features VXLAN offload capabilities. The full list is available here: https://access.redhat.com/articles/1390483

PART I. COMMON TASKS

Covers common administrative tasks and basic troubleshooting steps.

CHAPTER 3. COMMON ADMINISTRATIVE NETWORKING TASKS

OpenStack Networking (neutron) is the software-defined networking component of Red Hat OpenStack Platform. The virtual network infrastructure enables connectivity between instances and the physical external network.

This section contains information about common administration tasks, such as adding and removing subnets and routers to suit your Red Hat OpenStack Platform deployment.

3.1. CREATING A NETWORK

Create a network so that your instances can communicate with each other and receive IP addresses using DHCP. For more information about external network connections, see *Bridging the physical network*.

When creating networks, it is important to know that networks can host multiple subnets. This is useful if you intend to host distinctly different systems in the same network, and prefer a measure of isolation between them. For example, you can designate that only webserver traffic is present on one subnet, while database traffic traverses another. Subnets are isolated from each other, and any instance that wants to communicate with another subnet must have their traffic directed by a router. Consider placing systems that require a high volume of traffic amongst themselves in the same subnet, so that they do not require routing, and can avoid the subsequent latency and load.

1. In the dashboard, select **Project** > **Network** > **Networks**
2. Click **+Create Network** and specify the following values:

Field	Description
Network Name	Descriptive name, based on the role that the network will perform. If you are integrating the network with an external VLAN, consider appending the VLAN ID number to the name. For example, **webservers_122**, if you are hosting HTTP web servers in this subnet, and your VLAN tag is **122**. Or you might use **internal-only** if you intend to keep the network traffic private, and not integrate the network with an external network.
Admin State	Controls whether the network is immediately available. Use this field to create the network in a Down state, where it is logically present but inactive. This is useful if you do not intend to enter the network into production immediately.
Create Subnet	Determines whether to create a subnet. For example, you might not want to create a subnet if you intend to keep this network as a placeholder without network connectivity.

3. Click the **Next** button, and specify the following values in the **Subnet** tab:

Field	Description
Subnet Name	Enter a descriptive name for the subnet.
Network Address	Enter the address in CIDR format, which contains the IP address range and subnet mask in one value. To determine the address, calculate the number of bits masked in the subnet mask and append that value to the IP address range. For example, the subnet mask 255.255.255.0 has 24 masked bits. To use this mask with the IPv4 address range 192.168.122.0, specify the address 192.168.122.0/24.
IP Version	Specifies the internet protocol version, where valid types are IPv4 or IPv6. The IP address range in the Network Address field must match whichever version you select.
Gateway IP	IP address of the router interface for your default gateway. This address is the next hop for routing any traffic destined for an external location, and must be within the range that you specify in the Network Address field. For example, if your CIDR network address is 192.168.122.0/24, then your default gateway is likely to be 192.168.122.1.
Disable Gateway	Disables forwarding and isolates the subnet.

4. Click **Next** to specify **DHCP** options:

- **Enable DHCP** - Enables DHCP services for this subnet. You can use DHCP to automate the distribution of IP settings to your instances.
- **IPv6 Address** - Configuration Modes. If you create an IPv6 network, you must specify how to allocate IPv6 addresses and additional information:
    - **No Options Specified** - Select this option if you want to set IP addresses manually, or if you use a non OpenStack-aware method for address allocation.
    - **SLAAC (Stateless Address Autoconfiguration)** - Instances generate IPv6 addresses based on Router Advertisement (RA) messages sent from the OpenStack Networking router. Use this configuration to create an OpenStack Networking subnet with ra_mode set to slaac and address_mode set to slaac.
    - **DHCPv6 stateful** - Instances receive IPv6 addresses as well as additional options (for example, DNS) from the OpenStack Networking DHCPv6 service. Use this configuration to create a subnet with ra_mode set to dhcpv6-stateful and address_mode set to dhcpv6-stateful.
    - **DHCPv6 stateless** - Instances generate IPv6 addresses based on Router Advertisement (RA) messages sent from the OpenStack Networking router. Additional

options (for example, DNS) are allocated from the OpenStack Networking DHCPv6 service. Use this configuration to create a subnet with ra_mode set to dhcpv6-stateless and address_mode set to dhcpv6-stateless.

- **Allocation Pools** - Range of IP addresses that you want DHCP to assign. For example, the value 192.168.22.100,192.168.22.100 considers all up addresses in that range as available for allocation.
- **DNS Name Servers** - IP addresses of the DNS servers available on the network. DHCP distributes these addresses to the instances for name resolution.
- **Host Routes** - Static host routes. First, specify the destination network in CIDR format, followed by the next hop that you want to use for routing (for example, 192.168.23.0/24, 10.1.31.1). Provide this value if you need to distribute static routes to instances.

5. Click **Create**.
   You can view the complete network in the **Networks** tab. You can also click **Edit** to change any options as needed. When you create instances, you can configure them now to use its subnet, and they receive any specified DHCP options.

3.2. CREATING AN ADVANCED NETWORK

Advanced network options are available for administrators, when creating a network from the **Admin** view. Use these options to specify tenants and to define the network type that you want to use.

To create an advanced network, complete the following steps:

1. In the dashboard, select **Admin > Networks > Create Network > Project**.
2. Select the project that you want to host the new network with the **Project** drop-down list.
3. Review the options in **Provider Network Type**:
   - **Local** - Traffic remains on the local Compute host and is effectively isolated from any external networks.
   - **Flat** - Traffic remains on a single network and can also be shared with the host. No VLAN tagging or other network segregation takes place.
   - **VLAN** - Create a network using a VLAN ID that corresponds to a VLAN present in the physical network. This option allows instances to communicate with systems on the same layer 2 VLAN.
   - **GRE** - Use a network overlay that spans multiple nodes for private communication between instances. Traffic egressing the overlay must be routed.
   - **VXLAN** - Similar to GRE, and uses a network overlay to span multiple nodes for private communication between instances. Traffic egressing the overlay must be routed.
4. Click **Create Network**.
   Review the Project Network Topology to validate that the network has been successfully created.

3.3. ADDING NETWORK ROUTING

To allow traffic to be routed to and from your new network, you must add its subnet as an interface to an existing virtual router:

1. In the dashboard, select **Project > Network > Routers**
2. Select your virtual router name in the **Routers** list, and click **Add Interface**.
   In the Subnet list, select the name of your new subnet. You can optionally specify an IP address for the interface in this field.
3. Click **Add Interface**.
   Instances on your network can now communicate with systems outside the subnet.

3.4. DELETING A NETWORK

There are occasions where it becomes necessary to delete a network that was previously created, perhaps as housekeeping or as part of a decommissioning process. You must first remove or detach any interfaces where the network is still in use, before you can successfully delete a network.

To delete a network in your project, together with any dependent interfaces, complete the following steps:

1. In the dashboard, select **Project > Network > Networks**
   Remove all router interfaces associated with the target network subnets.

   To remove an interface, find the ID number of the network that you want to delete by clicking on your target network in the **Networks** list, and looking at the ID field. All the subnets associated with the network share this value in the **Network ID** field.
2. Navigate to **Project > Network > Routers**, click the name of your virtual router in the **Routers** list, and locate the interface attached to the subnet that you want to delete.
   You can distinguish this subnet from the other subnets by the IP address that served as the gateway IP. You can further validate the distinction by ensuring that the network ID of the interface matches the ID that you noted in the previous step.
3. Click the **Delete Interface** button for the interface that you want to delete.
4. Select **Project > Network > Networks**, and click the name of your network.
5. Click the **Delete Subnet** button for the subnet that you want to delete.

   NOTE

   If you are still unable to remove the subnet at this point, ensure it is not already being used by any instances.
6. Select **Project > Network > Networks**, and select the network you would like to delete.
7. Click **Delete Networks**.

3.5. PURGING THE NETWORKING FOR A TENANT

Use the neutron purge command to delete all neutron resources that belong to a particular project.

For example, to purge the neutron resources of the **test-project** project prior to deletion, run the following commands:

```
# openstack project list
+----------------------------------+--------------+
| ID                               | Name         |
+----------------------------------+--------------+
| 02e501908c5b438dbc73536c10c9aac0 | test-project |
| 519e6344f82e4c079c8e2eabb690023b | services     |
| 80bf5732752a41128e612fe615c886c6 | demo         |
| 98a2f53c20ce4d50a40dac4a38016c69 | admin        |
+----------------------------------+--------------+

# neutron purge 02e501908c5b438dbc73536c10c9aac0
Purging resources: 100% complete.
Deleted 1 security_group, 1 router, 1 port, 1 network.

# openstack project delete 02e501908c5b438dbc73536c10c9aac0
```

3.6. WORKING WITH SUBNETS

Use subnets to grant network connectivity to instances. Each instance is assigned to a subnet as part of the instance creation process, therefore it’s important to consider proper placement of instances to best accommodate their connectivity requirements.

You can create subnets only in pre-existing networks. Remember that tenant networks in OpenStack Networking can host multiple subnets. This is useful if you intend to host distinctly different systems in the same network, and prefer a measure of isolation between them.

For example, you can designate that only webserver traffic is present on one subnet, while database traffic traverse another.

Subnets are isolated from each other, and any instance that wants to communicate with another subnet must have their traffic directed by a router. Therefore, you can lessen network latency and load by grouping systems in the same subnet that require a high volume of traffic between each other.

3.6.1. Creating a subnet

To create a subnet, follow these steps:

1. In the dashboard, select **Project > Network > Networks**, and click the name of your network in the **Networks** view.

2. Click **Create Subnet**, and specify the following values:

Field	Description
Subnet Name	Descriptive subnet name.

Field	Description
Network Address	Address in CIDR format, which contains the IP address range and subnet mask in one value. To determine the CIDR address, calculate the number of bits masked in the subnet mask and append that value to the IP address range. For example, the subnet mask 255.255.255.0 has 24 masked bits. To use this mask with the IPv4 address range 192.168.122.0, specify the address 192.168.122.0/24.
IP Version	Internet protocol version, where valid types are IPv4 or IPv6. The IP address range in the Network Address field must match whichever protocol version you select.
Gateway IP	IP address of the router interface for your default gateway. This address is the next hop for routing any traffic destined for an external location, and must be within the range that you specify in the Network Address field. For example, if your CIDR network address is 192.168.122.0/24, then your default gateway is likely to be 192.168.122.1.
Disable Gateway	Disables forwarding and isolates the subnet.

3. Click **Next** to specify **DHCP** options:

   - **Enable DHCP** - Enables DHCP services for this subnet. You can use DHCP to automate the distribution of IP settings to your instances.

   - **IPv6 Address** - Configuration Modes. If you create an IPv6 network, you must specify how to allocate IPv6 addresses and additional information:

     - **No Options Specified** - Select this option if you want to set IP addresses manually, or if you use a non OpenStack-aware method for address allocation.

     - **SLAAC (Stateless Address Autoconfiguration)** - Instances generate IPv6 addresses based on Router Advertisement (RA) messages sent from the OpenStack Networking router. Use this configuration to create an OpenStack Networking subnet with ra_mode set to slaac and address_mode set to slaac.

     - **DHCPv6 stateful** - Instances receive IPv6 addresses as well as additional options (for example, DNS) from the OpenStack Networking DHCPv6 service. Use this configuration to create a subnet with ra_mode set to dhcpv6-stateful and address_mode set to dhcpv6-stateful.

     - **DHCPv6 stateless** - Instances generate IPv6 addresses based on Router Advertisement (RA) messages sent from the OpenStack Networking router. Additional options (for example, DNS) are allocated from the OpenStack Networking DHCPv6 service. Use this configuration to create a subnet with ra_mode set to dhcpv6-stateless and address_mode set to dhcpv6-stateless.

- **Allocation Pools** - Range of IP addresses that you want DHCP to assign. For example, the value 192.168.22.100,192.168.22.100 considers all up addresses in that range as available for allocation.
- **DNS Name Servers** - IP addresses of the DNS servers available on the network. DHCP distributes these addresses to the instances for name resolution.
- **Host Routes** - Static host routes. First, specify the destination network in CIDR format, followed by the next hop that you want to use for routing (for example, 192.168.23.0/24, 10.1.31.1). Provide this value if you need to distribute static routes to instances.

4. Click **Create**.
   You can view the subnet in the **Subnets** list. You can also click **Edit** to change any options as needed. When you create instances, you can configure them now to use its subnet, and they receive any specified DHCP options.

3.7. DELETING A SUBNET

You can delete a subnet if it is no longer in use. However, if any instances are still configured to use the subnet, the deletion attempt fails and the dashboard displays an error message.

Complete the following steps to delete a specific subnet in a network:

1. In the dashboard, select **Project > Network > Networks**
2. Click the name of your network.
3. Select the target subnet, and click **Delete Subnets**.

3.8. ADDING A ROUTER

OpenStack Networking provides routing services using an SDN-based virtual router. Routers are a requirement for your instances to communicate with external subnets, including those in the physical network. Routers and subnets connect using interfaces, with each subnet requiring its own interface to the router.

The default gateway of a router defines the next hop for any traffic received by the router. Its network is typically configured to route traffic to the external physical network using a virtual bridge.

To create a router, complete the following steps:

1. In the dashboard, select **Project > Network > Routers**, and click **Create Router**.
2. Enter a descriptive name for the new router, and click **Create router**.
3. Click **Set Gateway** next to the entry for the new router in the **Routers** list.
4. In the **External Network** list, specify the network that you want to receive traffic destined for an external location.
5. Click **Set Gateway**.
   After you add a router, you must configure any subnets you have created to send traffic using this router. You do this by creating interfaces between the subnet and the router.

IMPORTANT

The default routes for subnets must not be overwritten. When the default route for a subnet is removed, the L3 agent automatically removes the corresponding route in the router namespace too, and network traffic cannot flow to and from the associated subnet. If the existing router namespace route has been removed, to fix this problem, perform these steps:

1. Disassociate all floating IPs on the subnet.
2. Detach the router from the subnet.
3. Re-attach the router to the subnet.
4. Re-attach all floating IPs.

3.9. DELETING A ROUTER

You can delete a router if it has no connected interfaces.

To remove its interfaces and delete a router, complete the following steps:

1. In the dashboard, select **Project > Network > Routers**, and click the name of the router that you want to delete.
2. Select the interfaces of type **Internal Interface**, and click **Delete Interfaces**.
3. From the Routers list, select the target router and click **Delete Routers**.

3.10. ADDING AN INTERFACE

You can use interfaces to interconnect routers with subnets so that routers can direct any traffic that instances send to destinations outside of their intermediate subnet.

To add a router interface and connect the new interface to a subnet, complete these steps:

NOTE

This procedure uses the Network Topology feature. Using this feature, you can see a graphical representation of all your virtual routers and networks while you to perform network management tasks.

1. In the dashboard, select **Project > Network > Network Topology**.
2. Locate the router that you want to manage, hover your mouse over it, and click **Add Interface**.
3. Specify the Subnet that you want to connect to the router.
   You can also specify an IP address. The address is useful for testing and troubleshooting purposes, since a successful ping to this interface indicates that the traffic is routing as expected.
4. Click **Add interface**.
   The Network Topology diagram automatically updates to reflect the new interface connection between the router and subnet.

3.11. DELETING AN INTERFACE

You can remove an interface to a subnet if you no longer require the router to direct traffic for the subnet.

To delete an interface, complete the following steps:

1. In the dashboard, select **Project > Network > Routers**
2. Click the name of the router that hosts the interface that you want to delete.
3. Select the interface type (**Internal Interface**), and click **Delete Interfaces**.

3.12. CONFIGURING IP ADDRESSING

Follow the procedures in this section to manage your IP address allocation in OpenStack Networking.

3.12.1. Creating floating IP pools

You can use floating IP addresses to direct ingress network traffic to your OpenStack instances. First, you must define a pool of validly routable external IP addresses, which you can then assign to instances dynamically. OpenStack Networking routes all incoming traffic destined for that floating IP to the instance that you associate with the floating IP.

NOTE

OpenStack Networking allocates floating IP addresses to all projects (tenants) from the same IP ranges/CIDRs. As a result, all projects can consume floating IPs from every floating IP subnet. You can manage this behavior using quotas for specific projects. For example, you can set the default to *10* for *ProjectA* and *ProjectB*, while setting the quota for *ProjectC* to *0*.

When you create an external subnet, you can also define the floating IP allocation pool. If the subnet hosts only floating IP addresses, consider disabling DHCP allocation with the **--no-dhcp** option in the **openstack subnet create** command:

```
# openstack subnet create --no-dhcp --allocation-pool start=IP_ADDRESS,end=IP_ADDRESS --
gateway IP_ADDRESS --network SUBNET_RANGE NETWORK_NAME
```

For example:

```
# openstack subnet create --no-dhcp --allocation_pool start=192.168.100.20,end=192.168.100.100 --
gateway 192.168.100.1 --network 192.168.100.0/24 public
```

3.12.2. Assigning a specific floating IP

You can assign a specific floating IP address to an instance using the **nova** command.

```
# nova floating-ip-associate INSTANCE_NAME IP_ADDRESS
```

In this example, a floating IP address is allocated to an instance named **corp-vm-01**:

```
# nova floating-ip-associate corp-vm-01 192.168.100.20
```

3.12.3. Assigning a random floating IP

To dynamically allocate floating IP addresses to instances, complete these steps:

1. Enter the following **openstack** command:

   ```
   # openstack floating ip create
   ```

   In this example, you do not select a particular IP address, but instead request that OpenStack Networking allocate a floating IP address from the pool:

   ```
   # openstack floating ip create public
   +---------------------+--------------------------------------+
   | Field               | Value                                |
   +---------------------+--------------------------------------+
   | fixed_ip_address    |                                      |
   | floating_ip_address | 192.168.100.20                       |
   | floating_network_id | 7a03e6bc-234d-402b-9fb2-0af06c85a8a3 |
   | id                  | 9d7e2603482d                         |
   | port_id             |                                      |
   | router_id           |                                      |
   | status              | ACTIVE                               |
   | tenant_id           | 9e67d44eab334f07bf82fa1b17d824b6     |
   +---------------------+--------------------------------------+
   ```

   After you allocate the IP address, you can assign it to a particular instance.

2. Enter the following command to locate the port ID associated with your instance:

   ```
   # openstack port list
   ```

   (The port ID maps to the fixed IP address allocated to the instance.)

   ```
   # openstack port list
   +--------+------+-------------+--------------------------------------------------------+
   | id     | name | mac_address | fixed_ips                                              |
   +--------+------+-------------+--------------------------------------------------------+
   | ce8320 |      | 3e:37:09:4b | {"subnet_id": "361f27", "ip_address": "192.168.100.2"} |
   | d88926 |      | 3e:1d:ea:31 | {"subnet_id": "361f27", "ip_address": "192.168.100.5"} |
   | 8190ab |      | 3e:a3:3d:2f | {"subnet_id": "b74dbb", "ip_address": "10.10.1.25"}    |
   +--------+------+-------------+--------------------------------------------------------+
   ```

3. Associate the instance ID with the port ID of the instance:
   openstack server add floating ip *INSTANCE_NAME_OR_ID FLOATING_IP_ADDRESS*

   For example:

   ```
   # openstack server add floating ip VM1 172.24.4.225
   ```

4. Validate that you used the correct port ID for the instance by making sure that the MAC address (third column) matches the port on the instance.

```
# openstack port list
+--------+------+-------------+--------------------------------------------------------+
| id     | name | mac_address | fixed_ips                                              |
+--------+------+-------------+--------------------------------------------------------+
| ce8320 |      | 3e:37:09:4b | {"subnet_id": "361f27", "ip_address": "192.168.100.2"} |
| d88926 |      | 3e:1d:ea:31 | {"subnet_id": "361f27", "ip_address": "192.168.100.5"} |
| 8190ab |      | 3e:a3:3d:2f | {"subnet_id": "b74dbb", "ip_address": "10.10.1.25"}|
+--------+------+-------------+--------------------------------------------------------+
```

3.13. CREATING MULTIPLE FLOATING IP POOLS

OpenStack Networking supports one floating IP pool for each L3 agent. Therefore, you must scale your L3 agents to create additional floating IP pools.

NOTE

Make sure that in **/var/lib/config-data/neutron/etc/neutron/neutron.conf** the property **handle_internal_only_routers** is set to True for only one L3 agent in your environment. This option configures the L3 agent to manage only non-external routers.

3.14. BRIDGING THE PHYSICAL NETWORK

Bridge your virtual network to the physical network to enable connectivity to and from virtual instances.

In this procedure, the example physical interface, **eth0**, is mapped to the bridge, **br-ex**; the virtual bridge acts as the intermediary between the physical network and any virtual networks.

As a result, all traffic traversing **eth0** uses the configured Open vSwitch to reach instances.

For more information, see Chapter 11, *Configuring bridge mappings*.

To map a physical NIC to the virtual Open vSwitch bridge, complete the following steps:

1. Open **/etc/sysconfig/network-scripts/ifcfg-eth0** in a text editor, and update the following parameters with values appropriate for the network at your site:
   - IPADDR
   - NETMASK GATEWAY
   - DNS1 (name server)
     Here is an example:

     ```
     # vi /etc/sysconfig/network-scripts/ifcfg-eth0
     DEVICE=eth0
     TYPE=OVSPort
     DEVICETYPE=ovs
     OVS_BRIDGE=br-ex
     ONBOOT=yes
     ```

2. Open **/etc/sysconfig/network-scripts/ifcfg-br-ex** in a text editor and update the virtual bridge parameters with the IP address values that were previously allocated to eth0:

   ```
   # vi /etc/sysconfig/network-scripts/ifcfg-br-ex
   ```

```
DEVICE=br-ex
DEVICETYPE=ovs
TYPE=OVSBridge
BOOTPROTO=static
IPADDR=192.168.120.10
NETMASK=255.255.255.0
GATEWAY=192.168.120.1
DNS1=192.168.120.1
ONBOOT=yes
```

You can now assign floating IP addresses to instances and make them available to the physical network.

CHAPTER 4. PLANNING IP ADDRESS USAGE

An OpenStack deployment can consume a larger number of IP addresses than might be expected. This section contains information about correctly anticipating the quantity of addresses that you require, and where the addresses are used in your environment.

4.1. VLAN PLANNING

When you plan your Red Hat OpenStack Platform deployment, you start with a number of subnets, from which you allocate individual IP addresses. When you use multiple subnets you can segregate traffic between systems into VLANs.

For example, it is ideal that your management or API traffic is not on the same network as systems that serve web traffic. Traffic between VLANs travels through a router where you can implement firewalls to govern traffic flow.

You must plan your VLANs as part of your overall plan that includes traffic isolation, high availability, and IP address utilization for the various types of virtual networking resources in your deployment.

NOTE

The maximum number of VLANs in a single network, or in one OVS agent for a network node, is 4094. In situations where you require more than the maximum number of VLANs, you can create several provider networks (VXLAN networks) and several network nodes, one per network. Each node can contain up to 4094 private networks.

4.2. TYPES OF NETWORK TRAFFIC

You can allocate separate VLANs for the different types of network traffic that you want to host. For example, you can have separate VLANs for each of these types of networks. Only the External network must be routable to the external physical network. In this release, director provides DHCP services.

NOTE

You do not require all of the isolated VLANs in this section for every OpenStack deployment.. For example, if your cloud users do not create ad hoc virtual networks on demand, then you may not require a tenant network. If you want each VM to connect directly to the same switch as any other physical system, connect your Compute nodes directly to a provider network and configure your instances to use that provider network directly.

- **Provisioning network** - This VLAN is dedicated to deploying new nodes using director over PXE boot. OpenStack Orchestration (heat) installs OpenStack onto the overcloud bare metal servers. These servers attach to the physical network to receive the platform installation image from the undercloud infrastructure.

- **Internal API network** - The OpenStack services use the Internal API networkfor communication, including API communication, RPC messages, and database communication. In addition, this network is used for operational messages between controller nodes. When planning your IP address allocation, note that each API service requires its own IP address. Specifically, you must plan IP addresses for each of the following services:

  - vip-msg (ampq)

- vip-keystone-int
- vip-glance-int
- vip-cinder-int
- vip-nova-int
- vip-neutron-int
- vip-horizon-int
- vip-heat-int
- vip-ceilometer-int
- vip-swift-int
- vip-keystone-pub
- vip-glance-pub
- vip-cinder-pub
- vip-nova-pub
- vip-neutron-pub
- vip-horizon-pub
- vip-heat-pub
- vip-ceilometer-pub
- vip-swift-pub

NOTE

When using High Availability, Pacemaker moves VIP addresses between the physical nodes.

- **Storage** - Block Storage, NFS, iSCSI, and other storage services. Isolate this network to separate physical Ethernet links for performance reasons.
- **Storage Management** - OpenStack Object Storage (swift) uses this network to synchronise data objects between participating replica nodes. The proxy service acts as the intermediary interface between user requests and the underlying storage layer. The proxy receives incoming requests and locates the necessary replica to retrieve the requested data. Services that use a Ceph back end connect over the Storage Management network, since they do not interact with Ceph directly but rather use the front end service. Note that the RBD driver is an exception; this traffic connects directly to Ceph.
- **Tenant networks** - Neutron provides each tenant with their own networks using either VLAN segregation (where each tenant network is a network VLAN), or tunneling using VXLAN or GRE. Network traffic is isolated within each tenant network. Each tenant network has an IP subnet associated with it, and multiple tenant networks may use the same addresses.

- **External** - The External network hosts the public API endpoints and connections to the Dashboard (horizon). You can also use this network for SNAT. In a production deployment, it is common to use a separate network for floating IP addresses and NAT.

- **Provider networks** - Use provider networks to attach instances to existing network infrastructure. You can use provider networks to map directly to an existing physical network in the data center, using flat networking or VLAN tags. This allows an instance to share the same layer-2 network as a system external to the OpenStack Networking infrastructure.

4.3. IP ADDRESS CONSUMPTION

The following systems consume IP addresses from your allocated range:

- **Physical nodes** - Each physical NIC requires one IP address. It is common practice to dedicate physical NICs to specific functions. For example, allocate management and NFS traffic to distinct physical NICs, sometimes with multiple NICs connecting across to different switches for redundancy purposes.

- **Virtual IPs (VIPs) for High Availability**- Plan to allocate between one and three VIPs for each network that controller nodes share.

4.4. VIRTUAL NETWORKING

The following virtual resources consume IP addresses in OpenStack Networking. These resources are considered local to the cloud infrastructure, and do not need to be reachable by systems in the external physical network:

- **Tenant networks** - Each tenant network requires a subnet that it can use to allocate IP addresses to instances.

- **Virtual routers** - Each router interface plugging into a subnet requires one IP address. If you want to use DHCP, each router interface requires two IP addresses.

- **Instances** - Each instance requires an address from the tenant subnet that hosts the instance. If you require ingress traffic, you must allocate a floating IP address to the instance from the designated external network.

- **Management traffic** - Includes OpenStack Services and API traffic. All services share a small number of VIPs. API, RPC and database services communicate on the internal API VIP.

4.5. EXAMPLE NETWORK PLAN

This example shows a number of networks that accommodate multiple subnets, with each subnet being assigned a range of IP addresses:

Table 4.1. Example subnet plan

Subnet name	Address range	Number of addresses	Subnet Mask
Provisioning network	192.168.100.1 - 192.168.100.250	250	255.255.255.0
Internal API network	172.16.1.10 - 172.16.1.250	241	255.255.255.0

Subnet name	Address range	Number of addresses	Subnet Mask
Storage	172.16.2.10 - 172.16.2.250	241	255.255.255.0
Storage Management	172.16.3.10 - 172.16.3.250	241	255.255.255.0
Tenant network (GRE/VXLAN)	172.16.4.10 - 172.16.4.250	241	255.255.255.0
External network (incl. floating IPs)	10.1.2.10 - 10.1.3.222	469	255.255.254.0
Provider network (infrastructure)	10.10.3.10 - 10.10.3.250	241	255.255.252.0

CHAPTER 5. REVIEWING OPENSTACK NETWORKING ROUTER PORTS

Virtual routers in OpenStack Networking use ports to interconnect with subnets. You can review the state of these ports to determine whether they connect as expected.

5.1. VIEWING CURRENT PORT STATUS

Complete the following steps to lists all of the ports that attach to a particular router and to retrieve the current state of a port (DOWN or ACTIVE):

1. To view all the ports that attach to the router named **r1**, run the following command:

```
#  neutron router-port-list r1
```

Example result:

```
+--------------------------------------+------+-------------------+--------------------------------------------------------------------------------------+
| id                                   | name | mac_address       | fixed_ips                                                                            |
+--------------------------------------+------+-------------------+--------------------------------------------------------------------------------------+
| b58d26f0-cc03-43c1-ab23-ccdb1018252a |      | fa:16:3e:94:a7:df | {"subnet_id": "a592fdba-babd-48e0-96e8-2dd9117614d3", "ip_address": "192.168.200.1"} |
| c45e998d-98a1-4b23-bb41-5d24797a12a4 |      | fa:16:3e:ee:6a:f7 | {"subnet_id": "43f8f625-c773-4f18-a691-fd4ebfb3be54", "ip_address": "172.24.4.225"}  |
+--------------------------------------+------+-------------------+--------------------------------------------------------------------------------------+
```

2. To view the details of each port, run the following command. Include the port ID of the port that you want to view. The result includes the port status, indicated in the following example as having an **ACTIVE** state:

```
# openstack port show b58d26f0-cc03-43c1-ab23-ccdb1018252a
```

Example result:

```
+-----------------------+--------------------------------------------------------------------------------------+
| Field                 | Value                                                                                |
+-----------------------+--------------------------------------------------------------------------------------+
| admin_state_up        | True                                                                                 |
| allowed_address_pairs |                                                                                      |
| binding:host_id       | node.example.com                                                                     |
| binding:profile       | {}                                                                                   |
| binding:vif_details   | {"port_filter": true, "ovs_hybrid_plug": true}                                       |
| binding:vif_type      | ovs                                                                                  |
| binding:vnic_type     | normal                                                                               |
| device_id             | 49c6ebdc-0e62-49ad-a9ca-58cea464472f                                                 |
| device_owner          | network:router_interface                                                             |
| extra_dhcp_opts       |                                                                                      |
| fixed_ips             | {"subnet_id": "a592fdba-babd-48e0-96e8-2dd9117614d3", "ip_address":
"192.168.200.1"} |
| id                    | b58d26f0-cc03-43c1-ab23-ccdb1018252a                                              |
| mac_address           | fa:16:3e:94:a7:df                                                         |
| name                  |                                                                       |
| network_id            | 63c24160-47ac-4140-903d-8f9a670b0ca4
|
| security_groups       |                                                                       |
| status                | ACTIVE                                                                |
| tenant_id             | d588d1112e0f496fb6cac22f9be45d49                                              |
+-----------------------+--------------------------------------------------------------------------------------
+
```

3. Perform step 2 for each port to retrieve its status.

CHAPTER 6. TROUBLESHOOTING PROVIDER NETWORKS

A deployment of virtual routers and switches, also known as software-defined networking (SDN), may seem to introduce complexity. However, the diagnostic process of troubleshooting network connectivity in OpenStack Networking is similar to the diagnostic process for physical networks. If you use VLANs, you can consider the virtual infrastructure as a trunked extension of the physical network, rather than a wholly separate environment.

6.1. BASIC PING TESTING

The **ping** command is a useful tool for analyzing network connectivity problems. The results serve as a basic indicator of network connectivity, but might not entirely exclude all connectivity issues, such as a firewall blocking the actual application traffic. The ping command sends traffic to specific destinations, and then reports back whether the attempts were successful.

NOTE

The ping command is an ICMP operation. To use **ping**, you must allow ICMP traffic to traverse any intermediary firewalls.

Ping tests are most useful when run from the machine experiencing network issues, so it may be necessary to connect to the command line via the VNC management console if the machine seems to be completely offline.

For example, the following ping test command validates multiple layers of network infrastructure in order to succeed; name resolution, IP routing, and network switching must all function correctly:

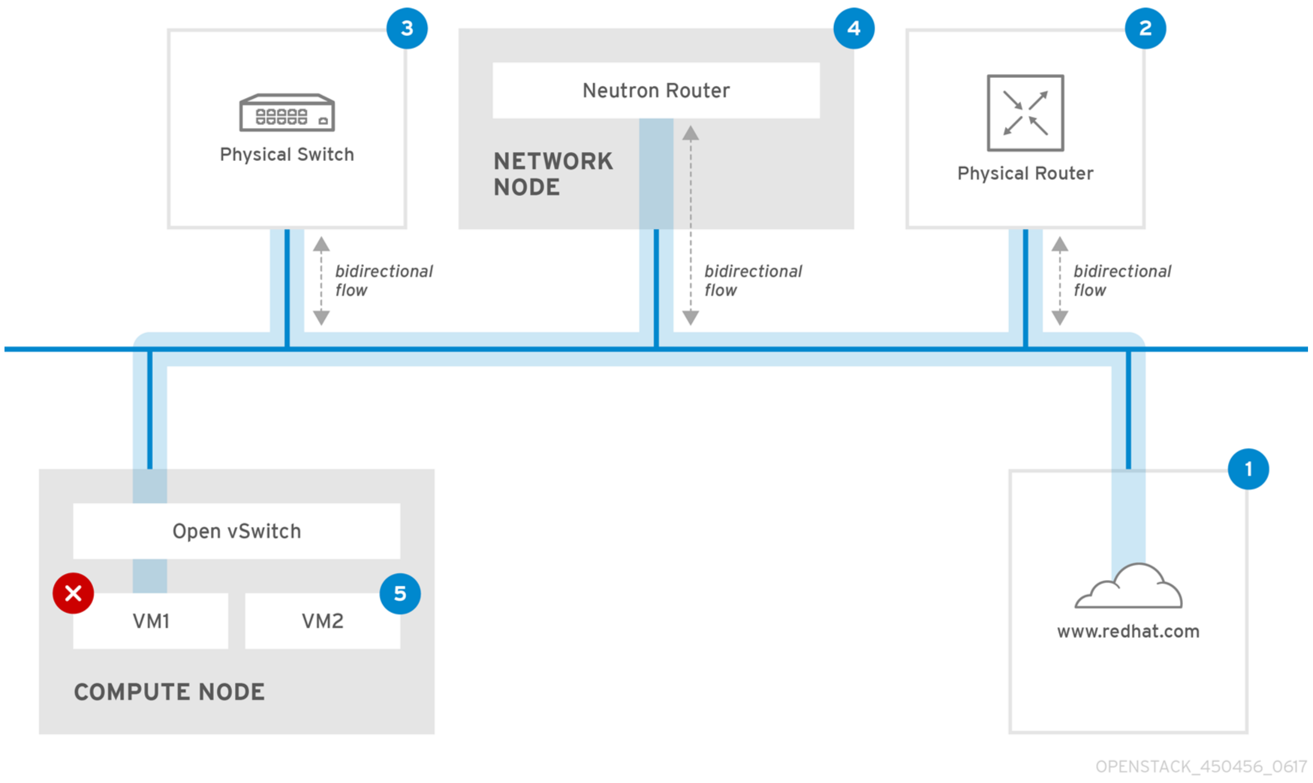
```
$ ping www.redhat.com

PING e1890.b.akamaiedge.net (125.56.247.214) 56(84) bytes of data.
64 bytes from a125-56.247-214.deploy.akamaitechnologies.com (125.56.247.214): icmp_seq=1
ttl=54 time=13.4 ms
64 bytes from a125-56.247-214.deploy.akamaitechnologies.com (125.56.247.214): icmp_seq=2
ttl=54 time=13.5 ms
64 bytes from a125-56.247-214.deploy.akamaitechnologies.com (125.56.247.214): icmp_seq=3
ttl=54 time=13.4 ms
^C
```

You can terminate the ping command with Ctrl-c, after which a summary of the results is presented. Zero percent packet loss indicates that the connection was stable and did not time out.

```
--- e1890.b.akamaiedge.net ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 13.461/13.498/13.541/0.100 ms
```

The results of a ping test can be very revealing, depending on which destination you test. For example, in the following diagram VM1 is experiencing some form of connectivity issue. The possible destinations are numbered in blue, and the conclusions drawn from a successful or failed result are presented:



1. The internet - a common first step is to send a ping test to an internet location, such as www.redhat.com.

- *Success*: This test indicates that all the various network points in between the machine and the Internet are functioning correctly. This includes the virtual and physical network infrastructure.

- *Failure*: There are various ways in which a ping test to a distant internet location can fail. If other machines on your network are able to successfully ping the internet, that proves the internet connection is working, and the issue is likely within the configuration of the local machine.

2. Physical router - This is the router interface that the network administrator designates to direct traffic onward to external destinations.

- *Success*: Ping tests to the physical router can determine whether the local network and underlying switches are functioning. These packets do not traverse the router, so they do not prove whether there is a routing issue present on the default gateway.

- *Failure*: This indicates that the problem lies between VM1 and the default gateway. The router/switches might be down, or you may be using an incorrect default gateway. Compare the configuration with that on another server that you know is functioning correctly. Try pinging another server on the local network.

3. Neutron router - This is the virtual SDN (Software-defined Networking) router that Red Hat OpenStack Platform uses to direct the traffic of virtual machines.

- *Success*: Firewall is allowing ICMP traffic, the Networking node is online.

- *Failure*: Confirm whether ICMP traffic is permitted in the security group of the instance. Check that the Networking node is online, confirm that all the required services are running, and review the L3 agent log (*/var/log/neutron/l3-agent.log*).

4. Physical switch - The physical switch manages traffic between nodes on the same physical network.

- *Success*: Traffic sent by a VM to the physical switch must pass through the virtual network infrastructure, indicating that this segment is functioning correctly.
- *Failure*: Check that the physical switch port is configured to trunk the required VLANs.

5. **VM2** - Attempt to ping a VM on the same subnet, on the same Compute node.

- *Success*: The NIC driver and basic IP configuration on VM1 are functional.
- *Failure*: Validate the network configuration on VM1. Or, firewall on VM2 might simply be blocking ping traffic. In addition, verify the virtual switching configuration and review the Open vSwitch (or Linux Bridge) log files.

6.2. TROUBLESHOOTING VLAN NETWORKS

OpenStack Networking can trunk VLAN networks through to the SDN switches. Support for VLAN-tagged provider networks means that virtual instances can integrate with server subnets in the physical network.

To troubleshoot connectivity to a VLAN Provider network, complete these steps:

1. Ping the gateway with **ping <gateway-IP-address>**.
   Consider this example, in which a network is created with these commands:

   ```
   # openstack network create --provider-network-type vlan --provider-physical-network phy-
   eno1 --provider-segment 120 provider
   # openstack subnet create --no-dhcp --allocation-pool
   start=192.168.120.1,end=192.168.120.153 --gateway 192.168.120.254 --network  provider
   public_subnet
   ```

   In this example, the gateway IP address is **192.168.120.254**.

   ```
   $ ping 192.168.120.254
   ```

2. If the ping fails, do the following:

   a. Confirm that you have network flow for the associated VLAN.
      It is possible that the VLAN ID has not been set. In this example, OpenStack Networking is configured to trunk VLAN 120 to the provider network. (See **--provider:segmentation_id=120** in the example in step 1.)

   b. Confirm the VLAN flow on the bridge interface using the command, **ovs-ofctl dump-flows <bridge-name>**.
      In this example the bridge is named **br-ex**:

      ```
      # ovs-ofctl dump-flows br-ex

       NXST_FLOW reply (xid=0x4):
        cookie=0x0, duration=987.521s, table=0, n_packets=67897, n_bytes=14065247,
      idle_age=0, priority=1 actions=NORMAL
        cookie=0x0, duration=986.979s, table=0, n_packets=8, n_bytes=648, idle_age=977,
      priority=2,in_port=12 actions=drop
      ```

6.2.1. Reviewing the VLAN configuration and log files

- **OpenStack Networking (neutron) agents** - Use the **openstack network agent list** command to verify that all agents are up and registered with the correct names:

```
(overcloud)[stack@undercloud~]$ openstack network agent list
+--------------------------------------+--------------------+-----------------------+-------+----------------+
| id                                   | agent_type         | host                  | alive | admin_state_up |
+--------------------------------------+--------------------+-----------------------+-------+----------------+
| a08397a8-6600-437d-9013-b2c5b3730c0c | Metadata agent     | rhelosp.example.com   | :-)
| True           |
| a5153cd2-5881-4fc8-b0ad-be0c97734e6a | L3 agent           | rhelosp.example.com   | :-)   |
True           |
| b54f0be7-c555-43da-ad19-5593a075ddf0 | DHCP agent         | rhelosp.example.com   | :-)
| True           |
| d2be3cb0-4010-4458-b459-c5eb0d4d354b | Open vSwitch agent | rhelosp.example.com   |
:-)   | True           |
+--------------------------------------+--------------------+-----------------------+-------+----------------+
```

- Review **/var/log/containers/neutron/openvswitch-agent.log** - this log should provide confirmation that the creation process used the **ovs-ofctl** command to configure VLAN trunking.

- Validate **external_network_bridge** in the **/etc/neutron/l3_agent.ini** file. If there is a hardcoded value in the **external_network_bridge** parameter, you cannot use a provider network with the L3-agent, and you cannot create the necessary flows. The **external_network_bridge** value must be in the format `external_network_bridge = "" `.

- Check the **network_vlan_ranges** value in the **/etc/neutron/plugin.ini** file. For provider networks, do not specify the numeric VLAN ID. Specify IDs only when using VLAN isolated tenant networks.

- Validate the OVS agent configuration file bridge mappings, confirm that the bridge mapped to **phy-eno1** exists and is properly connected to **eno1**.

6.3. TROUBLESHOOTING FROM WITHIN TENANT NETWORKS

In OpenStack Networking, all tenant traffic is contained within network namespaces so that tenants can configure networks without interfering with each other. For example, network namespaces allow different tenants to have the same subnet range of 192.168.1.1/24 without interference between them.

To begin troubleshooting a tenant network, first determine which network namespace contains the network:

1. List all the tenant networks using the **openstack network list** command:

```
# (overcloud)[stack@osp13-undercloud ~]$ openstack network list
+--------------------------------------+-------------+-------------------------------------------------------+
| id                                   | name        | subnets                                               |
+--------------------------------------+-------------+-------------------------------------------------------+
| 9cb32fe0-d7fb-432c-b116-f483c6497b08 | web-servers | 453d6769-fcde-4796-a205-
66ee01680bba 192.168.212.0/24 |
| a0cc8cdd-575f-4788-a3e3-5df8c6d0dd81 | private     | c1e58160-707f-44a7-bf94-
8694f29e74d3 10.0.0.0/24      |
| baadd774-87e9-4e97-a055-326bb422b29b | private     | 340c58e1-7fe7-4cf2-96a7-
96a0a4ff3231 192.168.200.0/24 |
| 24ba3a36-5645-4f46-be47-f6af2a7d8af2 | public      | 35f3d2cb-6e4b-4527-a932-
952a395c4bb3 172.24.4.224/28  |
+--------------------------------------+-------------+-----------------------------------------------------+
```

In this example,examine the **web-servers** network. Make a note of the id value in the web-server row (**9cb32fe0-d7fb-432c-b116-f483c6497b08**). This value is appended to the network namespace, which helps you identify the namespace in the next step.

2. List all the network namespaces using the **ip netns list** command:

```
# ip netns list
qdhcp-9cb32fe0-d7fb-432c-b116-f483c6497b08
qrouter-31680a1c-9b3e-4906-bd69-cb39ed5faa01
qrouter-62ed467e-abae-4ab4-87f4-13a9937fbd6b
qdhcp-a0cc8cdd-575f-4788-a3e3-5df8c6d0dd81
qrouter-e9281608-52a6-4576-86a6-92955df46f56
```

The output contains a namespace that matches the **web-servers** network id. In this example the namespace is **qdhcp-9cb32fe0-d7fb-432c-b116-f483c6497b08**.

3. Examine the configuration of the **web-servers** network by running commands within the namespace, prefixing the troubleshooting commands with **ip netns exec <namespace>**:

```
# ip netns exec qrouter-62ed467e-abae-4ab4-87f4-13a9937fbd6b route -n

Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.24.4.225    0.0.0.0         UG    0      0        0 qg-8d128f89-87
172.24.4.224    0.0.0.0         255.255.255.240 U     0      0        0 qg-8d128f89-87
192.168.200.0   0.0.0.0         255.255.255.0   U     0      0        0 qr-8efd6357-96
```

6.3.1. Performing advanced ICMP testing within the namespace

1. Capture ICMP traffic using the **tcpdump** command:

```
# ip netns exec qrouter-62ed467e-abae-4ab4-87f4-13a9937fbd6b tcpdump -qnntpi any icmp
```

2. In a separate command line window, perform a ping test to an external network:

```
# ip netns exec qrouter-62ed467e-abae-4ab4-87f4-13a9937fbd6b ping www.redhat.com
```

3. In the terminal running the **tcpdump** session, observe detailed results of the ping test.

```
tcpdump: listening on any, link-type LINUX_SLL (Linux cooked), capture size 65535 bytes
IP (tos 0xc0, ttl 64, id 55447, offset 0, flags [none], proto ICMP (1), length 88)
    172.24.4.228 > 172.24.4.228: ICMP host 192.168.200.20 unreachable, length 68
 IP (tos 0x0, ttl 64, id 22976, offset 0, flags [DF], proto UDP (17), length 60)
    172.24.4.228.40278 > 192.168.200.21: [bad udp cksum 0xfa7b -> 0xe235!] UDP, length 32
```

NOTE

When you perform a tcpdump analysis of traffic, you might observe the responding packets heading to the router interface rather than the instance. This is expected behavior, as the qrouter performs DNAT on the return packets.

CHAPTER 7. CONNECTING AN INSTANCE TO THE PHYSICAL NETWORK

This chapter contains information about using provider networks to connect instances directly to an external network.

7.1. OVERVIEW OF THE OPENSTACK NETWORKING TOPOLOGY

OpenStack Networking (neutron) has two categories of services distributed across a number of node types.

- **Neutron server** - This service runs the OpenStack Networking API server, which provides the API for end-users and services to interact with OpenStack Networking. This server also integrates with the underlying database to store and retrieve tenant network, router, and loadbalancer details, among others.
- **Neutron agents** - These are the services that perform the network functions for OpenStack Networking:
    - **neutron-dhcp-agent** - manages DHCP IP addressing for tenant private networks.
    - **neutron-l3-agent** - performs layer 3 routing between tenant private networks, the external network, and others.
- **Compute node** - This node hosts the hypervisor that runs the virtual machines, also known as instances. A Compute node must be wired directly to the network in order to provide external connectivity for instances. This node is typically where the l2 agents run, such as **neutron-openvswitch-agent**.

7.1.1. Service placement

The OpenStack Networking services can either run together on the same physical server, or on separate dedicated servers, which are named according to their roles:

- *Controller node* - The server that runs API service.
- *Network node* - The server that runs the OpenStack Networking agents.
- *Compute node* - The hypervisor server that hosts the instances.

The steps in this chapter apply to an environment that contains these three node types. If your deployment has both the Controller and Network node roles on the same physical node, then you must perform the steps from both sections on that server. This also applies for a High Availability (HA) environment, where all three nodes might be running the Controller node and Network node services with HA. As a result, you must complete the steps in sections applicable to Controller and Network nodes on all three nodes.

7.2. USING FLAT PROVIDER NETWORKS

The procedures in this section create flat provider networks that can connect instances directly to external networks. You would do this if you have multiple physical networks (for example, **physnet1**, **physnet2**) and separate physical interfaces (**eth0 -> physnet1**, and **eth1 -> physnet2**), and you need to connect each Compute node and Network node to those external networks.

NOTE

If you want to connect multiple VLAN-tagged interfaces (on a single NIC) to multiple provider networks, see Section 7.3, "Using VLAN provider networks" .

7.2.1. Configuring the Controller nodes

1. Edit **/etc/neutron/plugin.ini** (which is symlinked to **/etc/neutron/plugins/ml2/ml2_conf.ini**), add **flat** to the existing list of values, and set **flat_networks** to *:

```
type_drivers = vxlan,flat
flat_networks =*
```

2. Create a flat external network and associate it with the configured physical_network. Create this network as a shared network so that other users can connect their instances directly to it:

```
# openstack network create --provider-network-type flat --provider-physical-network physnet1 --
external public01
```

3. Create a subnet within this external network using the **openstack subnet create** command, or the OpenStack Dashboard:

```
# openstack subnet create --dhcp --allocation-pool start=192.168.100.20,end=192.168.100.100 --
gateway 192.168.100.1 --network public01 public_subnet
```

4. Restart the **neutron-server** service to apply this change:

```
# systemctl restart neutron-server.service
```

7.2.2. Configuring the Network node and Compute nodes

Complete the following steps on the Network node and the Compute nodes so that these nodes can connect to the external network, and can allow instances to communicate directly with the external network

1. Create the Open vSwitch bridge and port. Run the following command to create the external network bridge (br-ex) and add a corresponding port (eth1)

i. Edit **/etc/sysconfig/network-scripts/ifcfg-eth1**:

```
DEVICE=eth1
TYPE=OVSPort
DEVICETYPE=ovs
OVS_BRIDGE=br-ex
ONBOOT=yes
NM_CONTROLLED=no
BOOTPROTO=none
```

ii. Edit **/etc/sysconfig/network-scripts/ifcfg-br-ex**:

```
DEVICE=br-ex
TYPE=OVSBridge
DEVICETYPE=ovs
```

```
ONBOOT=yes
NM_CONTROLLED=no
BOOTPROTO=none
```

2. Restart the **network** service to apply these changes:

```
# systemctl restart network.service
```

3. Configure the physical networks in **/etc/neutron/plugins/ml2/openvswitch_agent.ini** and map the bridge to the physical network:

NOTE

For more information on configuring **bridge_mappings**, see Chapter 11, *Configuring bridge mappings*.

```
bridge_mappings = physnet1:br-ex
```

4. Restart the **neutron-openvswitch-agent** service on the Network and Compute nodes to apply these changes:

```
# systemctl restart neutron-openvswitch-agent
```

7.2.3. Configuring the Network node

1. Set the **external_network_bridge =** parameter to an empty value in **/etc/neutron/l3_agent.ini** to enable the use of external provider networks.

```
# Name of bridge used for external network traffic. This should be set to
# empty value for the linux bridge
external_network_bridge =
```

2. Restart **neutron-l3-agent** to apply these changes:

```
# systemctl restart neutron-l3-agent.service
```

NOTE

If you have multiple flat provider networks, ensure that each of them has a separate physical interface and bridge to connect them to the external network. Configure the *ifcfg-** scripts appropriately and use a comma-separated list for each network when specifying them in **bridge_mappings**. For more information on configuring **bridge_mappings**, see Chapter 11, *Configuring bridge mappings*.

7.2.4. Connecting an instance to the external network

After you create the external network, you can connect an instance to it and test connectivity:

1. Create a new instance.

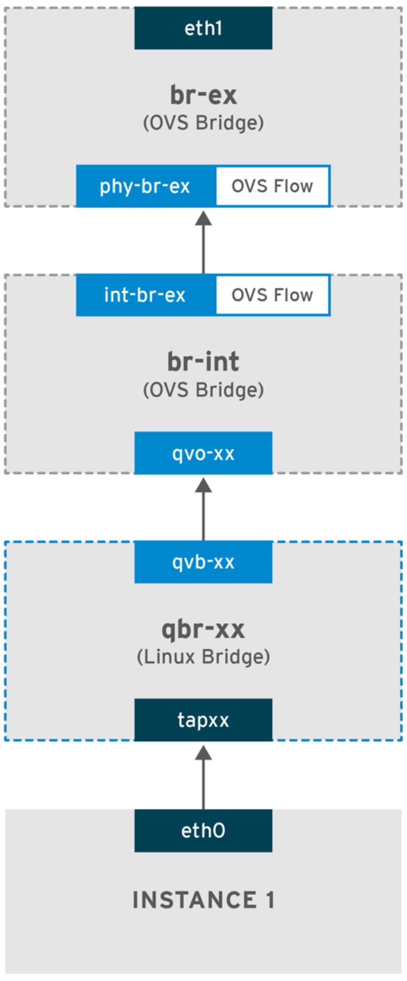
2. Use the **Networking** tab in the dashboard to add the new instance directly to the newly-created external network.

7.2.5. How does the flat provider network packet flow work?

This section describes in detail how traffic flows to and from an instance with flat provider network configuration.

The flow of outgoing traffic in a flat provider network

The following diagram describes the packet flow for traffic leaving an instance and arriving directly at an external network. After you configure the **br-ex** external bridge, add the physical interface to the bridge, and spawn an instance to a Compute node, the resulting configuration of interfaces and bridges resembles the configuration in the following diagram (if using the **iptables_hybrid** firewall driver):



OPENSTACK_450456_0617

1. Packets leave the **eth0** interface of the instance and arrive at the linux bridge **qbr-xx**.

2. Bridge **qbr-xx** is connected to **br-int** using veth pair **qvb-xx <-> qvo-xxx**. This is because the bridge is used to apply the inbound/outbound firewall rules defined by the security group.

3. Interface **qvb-xx** is connected to the **qbr-xx** linux bridge, and **qvoxx** is connected to the **br-int** Open vSwitch (OVS) bridge.

An example configuration of `qbr-xx`Linux bridge:

```
 # brctl show
qbr269d4d73-e7  8000.061943266ebb no  qvb269d4d73-e7
      tap269d4d73-e7
```

The configuration of **qvo-xx** on **br-int**:

```
# ovs-vsctl show
 Bridge br-int
      fail_mode: secure
          Interface "qvof63599ba-8f"
      Port "qvo269d4d73-e7"
          tag: 5
          Interface "qvo269d4d73-e7"
```

NOTE

Port **qvo-xx** is tagged with the internal VLAN tag associated with the flat provider network. In this example, the VLAN tag is **5**. When the packet reaches **qvo-xx**, the VLAN tag is appended to the packet header.

The packet is then moved to the **br-ex** OVS bridge using the patch-peer **int-br-ex <-> phy-br-ex**.

Example configuration of the patch-peer on **br-int**:

```
# ovs-vsctl show
   Bridge br-int
       fail_mode: secure
      Port int-br-ex
           Interface int-br-ex
               type: patch
               options: {peer=phy-br-ex}
```

Example configuration of the patch-peer on **br-ex**:

```
   Bridge br-ex
       Port phy-br-ex
           Interface phy-br-ex
               type: patch
               options: {peer=int-br-ex}
       Port br-ex
           Interface br-ex
               type: internal
```

When this packet reaches **phy-br-ex** on **br-ex**, an OVS flow inside **br-ex** strips the VLAN tag (5) and forwards it to the physical interface.

In the following example, the output shows the port number of **phy-br-ex** as **2**.

```
 # ovs-ofctl show br-ex
OFPT_FEATURES_REPLY (xid=0x2): dpid:00003440b5c90dc6
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: OUTPUT SET_VLAN_VID SET_VLAN_PCP STRIP_VLAN SET_DL_SRC SET_DL_DST
SET_NW_SRC SET_NW_DST SET_NW_TOS SET_TP_SRC SET_TP_DST ENQUEUE

 2(phy-br-ex): addr:ba:b5:7b:ae:5c:a2
     config:     0
     state:      0
     speed: 0 Mbps now, 0 Mbps max
```

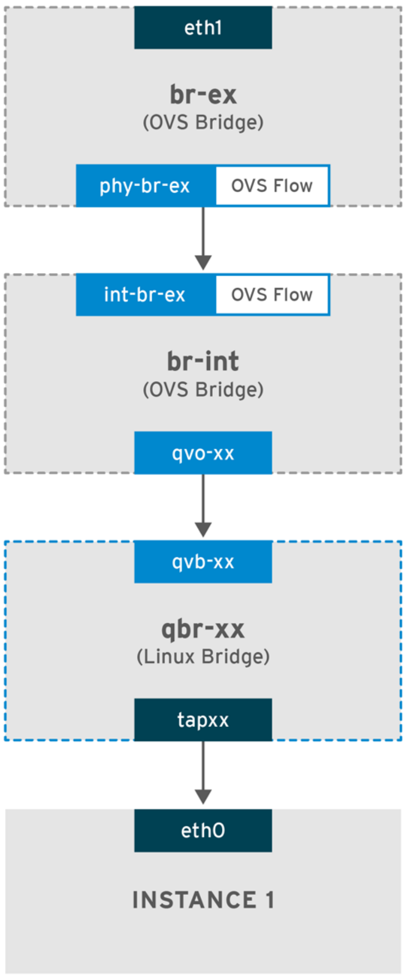
The following output shows any packet that arrives on **phy-br-ex** (**in_port=2**) with a VLAN tag of **5** (**dl_vlan=5**). In addition, an OVS flow in br-ex strips the VLAN tag and forwards the packet to the physical interface.

```
# ovs-ofctl dump-flows br-ex
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=4703.491s, table=0, n_packets=3620, n_bytes=333744, idle_age=0, priority=1
actions=NORMAL
 cookie=0x0, duration=3890.038s, table=0, n_packets=13, n_bytes=1714, idle_age=3764,
priority=4,in_port=2,dl_vlan=5 actions=strip_vlan,NORMAL
 cookie=0x0, duration=4702.644s, table=0, n_packets=10650, n_bytes=447632, idle_age=0,
priority=2,in_port=2 actions=drop
```

If the physical interface is another VLAN-tagged interface, then the physical interface adds the tag to the packet.

The flow of incoming traffic in a flat provider network

This section contains information about the flow of incoming traffic from the external network until it arrives at the interface of the instance.



1. Incoming traffic arrives at **eth1** on the physical node.

2. The packet passes to the **br-ex** bridge.

3. The packet moves to **br-int** via the patch-peer **phy-br-ex <--> int-br-ex**.

In the following example, **int-br-ex** uses port number **15**. See the entry containing **15(int-br-ex)**:

```
 # ovs-ofctl show br-int
OFPT_FEATURES_REPLY (xid=0x2): dpid:00004e67212f644d
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: OUTPUT SET_VLAN_VID SET_VLAN_PCP STRIP_VLAN SET_DL_SRC SET_DL_DST
SET_NW_SRC SET_NW_DST SET_NW_TOS SET_TP_SRC SET_TP_DST ENQUEUE
 15(int-br-ex): addr:12:4e:44:a9:50:f4
     config:     0
     state:      0
     speed: 0 Mbps now, 0 Mbps max
```

Observing the traffic flow on br-int

1. When the packet arrives at **int-br-ex**, an OVS flow rule within the **br-int** bridge amends the packet to add the internal VLAN tag **5**. See the entry for **actions=mod_vlan_vid:5**:

```
 # ovs-ofctl dump-flows br-int
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=5351.536s, table=0, n_packets=12118, n_bytes=510456, idle_age=0,
priority=1 actions=NORMAL
 cookie=0x0, duration=4537.553s, table=0, n_packets=3489, n_bytes=321696, idle_age=0,
priority=3,in_port=15,vlan_tci=0x0000 actions=mod_vlan_vid:5,NORMAL
 cookie=0x0, duration=5350.365s, table=0, n_packets=628, n_bytes=57892, idle_age=4538,
priority=2,in_port=15 actions=drop
 cookie=0x0, duration=5351.432s, table=23, n_packets=0, n_bytes=0, idle_age=5351, priority=0
actions=drop
```

2. The second rule manages packets that arrive on int-br-ex (in_port=15) with no VLAN tag (vlan_tci=0x0000): This rule adds VLAN tag 5 to the packet (**actions=mod_vlan_vid:5,NORMAL**) and forwards it to **qvoxxx**.

3. **qvoxxx** accepts the packet and forwards it to **qvbxx**, after stripping away the VLAN tag.

4. The packet then reaches the instance.

NOTE

VLAN tag 5 is an example VLAN that was used on a test Compute node with a flat provider network; this value was assigned automatically by **neutron-openvswitch-agent**. This value may be different for your own flat provider network, and can differ for the same network on two separate Compute nodes.

7.2.6. Troubleshooting instance-physical network connections on flat provider networks

The output provided in Section 7.2.5, “How does the flat provider network packet flow work?” - provides sufficient debugging information for troubleshooting a flat provider network, should anything go wrong. The following steps contain further information about the troubleshooting process.

1. Review the bridge_mappings:

Verify that the physical network name you use (for example, **physnet1**) is consistent with the contents of the **bridge_mapping** configuration as shown in this example:

```
 # grep bridge_mapping /etc/neutron/plugins/ml2/openvswitch_agent.ini
bridge_mappings = physnet1:br-ex

 # openstack network show provider-flat
...
| provider:physical_network | physnet1
...
```

2. **Review the network configuration:**

Confirm that the network is created as **external**, and uses the **flat** type:

```
 # openstack network show provider-flat
...
| provider:network_type     | flat                                 |
| router:external           | True                                 |
...
```

3. **Review the patch-peer:**

Run the **ovs-vsctl show** command, and verify that **br-int** and **br-ex** are connected using a patch-peer **int-br-ex <--> phy-br-ex**.

```
 # ovs-vsctl show
   Bridge br-int
       fail_mode: secure
      Port int-br-ex
          Interface int-br-ex
              type: patch
              options: {peer=phy-br-ex}
```

Example configuration of the patch-peer on **br-ex**:

```
   Bridge br-ex
       Port phy-br-ex
           Interface phy-br-ex
               type: patch
               options: {peer=int-br-ex}
       Port br-ex
           Interface br-ex
               type: internal
```

This connection is created when you restart the **neutron-openvswitch-agent** service, if **bridge_mapping** is correctly configured in **/etc/neutron/plugins/ml2/openvswitch_agent.ini**. Re-check the **bridge_mapping** setting if the connection is not created after you restart the service.

NOTE

For more information on configuring **bridge_mappings**, see Chapter 11, *Configuring bridge mappings*.

4. **Review the network flows:**

Run **ovs-ofctl dump-flows br-ex** and **ovs-ofctl dump-flows br-int** and review whether the flows strip the internal VLAN IDs for outgoing packets, and add VLAN IDs for incoming packets. This flow is first added when you spawn an instance to this network on a specific Compute node.

- If this flow is not created after spawning the instance, verify that the network is created as **flat**, is **external**, and that the **physical_network** name is correct. In addition, review the **bridge_mapping** settings.
- Finally, review the **ifcfg-br-ex** and **ifcfg-ethx** configuration. Ensure that **ethX** is added as a port within **br-ex**, and that **ifcfg-br-ex** and **ifcfg-ethx** have an **UP** flag in the output of **ip a**.

The following output shows **eth1** is a port in **br-ex**:

```
    Bridge br-ex
        Port phy-br-ex
            Interface phy-br-ex
                type: patch
                options: {peer=int-br-ex}
        Port "eth1"
            Interface "eth1"
```

The following example demonstrates that **eth1** is configured as an OVS port, and that the kernel knows to transfer all packets from the interface, and send them to the OVS bridge **br-ex**. This can be observed in the entry: **master ovs-system**.

```
 # ip a
5: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq master ovs-system state
UP qlen 1000
```

7.3. USING VLAN PROVIDER NETWORKS

To connect multiple VLAN-tagged interfaces on a single NIC to multiple provider networks, complete the steps in this section to create VLAN provider networks that can connect instances directly to external networks.This example uses a physical network called **physnet1**, with a range of VLANs (**171-172**). The network nodes and compute nodes are connected to the physical network using physical interface **eth1**. The switch ports that these interfaces connect to must be configured to trunk the required VLAN ranges. Complete the following procedures to configure the VLAN provider networks using the example VLAN IDs and names.

7.3.1. Configuring the Controller nodes

1. Enable the *vlan* mechanism driver by editing **/etc/neutron/plugin.ini** (symlinked to **/etc/neutron/plugins/ml2/ml2_conf.ini**), and add *vlan* to the existing list of values:

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

2. Configure the **network_vlan_ranges** setting to reflect the physical network and VLAN ranges in use:

```
[ml2_type_vlan]
network_vlan_ranges=physnet1:171:172
```

3. Restart the *neutron-server* service to apply the changes:

```
systemctl restart neutron-server
```

4. Create the external networks as type *vlan*, and associate them with the configured **physical_network**. Used the *--shared* when you create the external networks so that other users can connect instances directly. Run the following example command to create two networks: one for VLAN 171, and another for VLAN 172:

```
openstack network create
  --provider-network-type vlan
  --external
  --provider-physical-network physnet1
  --segment 171
  --share

openstack network create
  --provider-network-type vlan
  --external
  --provider-physical-network physnet1
  --segment 172
  --share
```

5. Create a number of subnets and configure them to use the external network. You can use either **openstack subnet create** or the dashboard to create these subnets. Ensure that the external subnet details you have received from your network administrator are correctly associated with each VLAN. In this example, VLAN 171 uses subnet *10.65.217.0/24* and VLAN 172 uses *10.65.218.0/24*:

```
openstack subnet create
  --network provider-171
  --subnet-range 10.65.217.0/24
  --dhcp
  --gateway 10.65.217.254
  --subnet-provider-171

openstack subnet create
  --network provider-172
  --subnet-range 10.65.218.0/24
  --dhcp
  --gateway 10.65.218.254
  --subnet-provider-172
```

7.3.2. Configuring the Network and Compute nodes

Complete the following steps on the Network node and Compute nodes to connect the nodes to the external network, and permit instances to communicate directly with the external network.

1. Create an external network bridge (*br-ex*), and associate a port (*eth1*) with it:

- This example configures *eth1* to use *br-ex*:

```
/etc/sysconfig/network-scripts/ifcfg-eth1

DEVICE=eth1
TYPE=OVSPort
DEVICETYPE=ovs
```

```
OVS_BRIDGE=br-ex
ONBOOT=yes
NM_CONTROLLED=no
BOOTPROTO=none
```

- This example configures the *br-ex* bridge:

```
/etc/sysconfig/network-scripts/ifcfg-br-ex:

DEVICE=br-ex
TYPE=OVSBridge
DEVICETYPE=ovs
ONBOOT=yes
NM_CONTROLLED=no
BOOTPROTO=none
```

2. Reboot the node, or restart the *network* service to apply the networking changes:

```
# systemctl restart network
```

3. Configure the physical networks in **/etc/neutron/plugins/ml2/openvswitch_agent.ini** and map bridges according to the physical network:

```
bridge_mappings = physnet1:br-ex
```

NOTE

For more information on configuring **bridge_mappings**, see Chapter 11, *Configuring bridge mappings*.

4. Restart the **neutron-openvswitch-agent** service on the network nodes and compute nodes to apply the changes:

```
systemctl restart neutron-openvswitch-agent
```

7.3.3. Configuring the Network node

1. Set the **external_network_bridge =** parameter to an empty value in **/etc/neutron/l3_agent.ini** so that you can use provider external networks (not bridge based external networks) where you add **external_network_bridge = br-ex**:

```
# Name of bridge used for external network traffic. This should be set to
# empty value for the linux bridge
external_network_bridge =
```

2. Restart **neutron-l3-agent** to apply the changes.

```
systemctl restart neutron-l3-agent
```

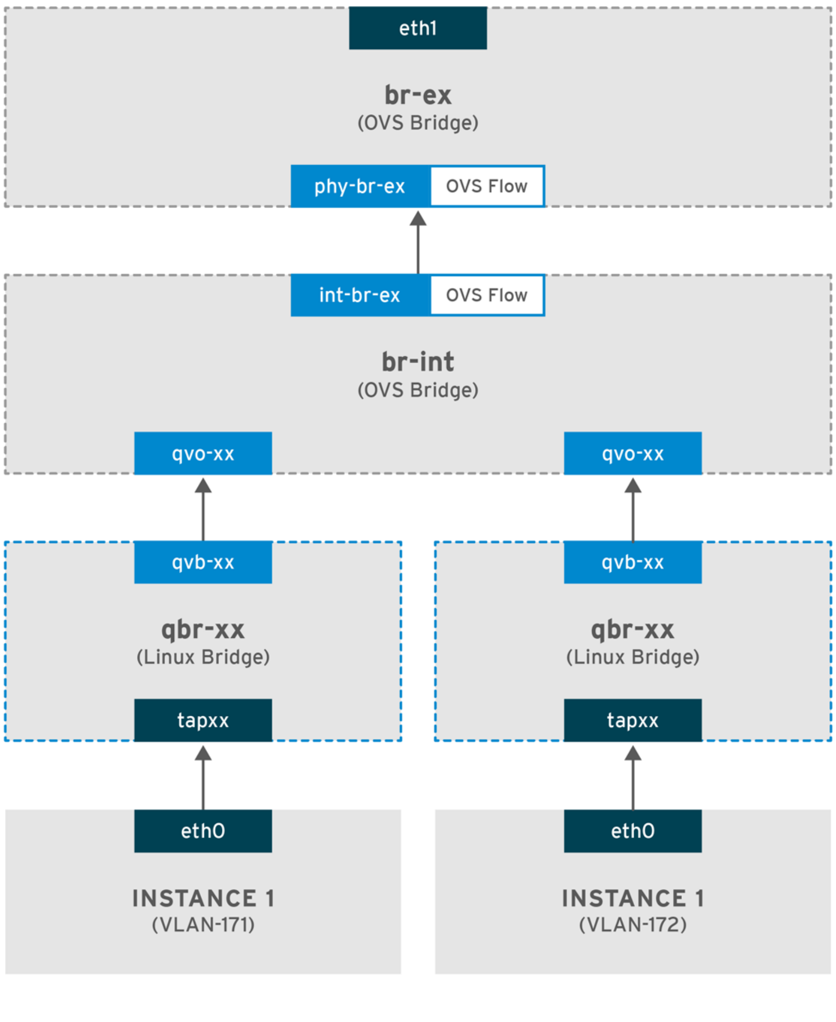
3. Create a new instance and use the **Networking** tab in the dashboard to add the new instance directly to the new external network.

7.3.4. How does the VLAN provider network packet flow work?

This section describes in detail how traffic flows to and from an instance with VLAN provider network configuration.

The flow of outgoing traffic in a VLAN provider network

The following diagram describes the packet flow for traffic leaving an instance and arriving directly to a VLAN provider external network. This example uses two instances attached to the two VLAN networks (171 and 172). After you configure *br-ex*, add a physical interface to it, and spawn an instance to a Compute node, the resulting configuration of interfaces and bridges resembles the configuration in the following diagram:



1. Packets leaving the *eth0* interface of the instance arrive at the linux bridge *qbr-xx* connected to the instance.

2. *qbr-xx* is connected to *br-int* using veth pair *qvbxx <→ qvoxxx*.

3. *qvbxx* is connected to the linux bridge *qbr-xx* and *qvoxx* is connected to the Open vSwitch bridge *br-int*.

Example configuration of *qbr-xx* on the Linux bridge.

This example features two instances and two corresponding linux bridges:

```
# brctl show
bridge name bridge id  STP enabled interfaces
```

```
qbr84878b78-63  8000.e6b3df9451e0 no  qvb84878b78-63
        tap84878b78-63

qbr86257b61-5d  8000.3a3c888eeae6 no  qvb86257b61-5d
        tap86257b61-5d
```

The configuration of *qvoxx* on *br-int*:

```
                options: {peer=phy-br-ex}
        Port "qvo86257b61-5d"
            tag: 3

            Interface "qvo86257b61-5d"
        Port "qvo84878b78-63"
            tag: 2
            Interface "qvo84878b78-63"
```

- **qvoxx** is tagged with the internal VLAN tag associated with the VLAN provider network. In this example, the internal VLAN tag 2 is associated with the VLAN provider network **provider-171** and VLAN tag 3 is associated with VLAN provider network **provider-172**. When the packet reaches *qvoxx*, the this VLAN tag is added to the packet header.
- The packet is then moved to the *br-ex* OVS bridge using patch-peer **int-br-ex** <→ **phy-br-ex**. Example patch-peer on *br-int*:

```
    Bridge br-int
        fail_mode: secure
       Port int-br-ex
            Interface int-br-ex
                type: patch
                options: {peer=phy-br-ex}
```

Example configuration of the patch peer on *br-ex*:

```
    Bridge br-ex
        Port phy-br-ex
            Interface phy-br-ex
                type: patch
                options: {peer=int-br-ex}
        Port br-ex
            Interface br-ex
                type: internal
```

- When this packet reaches *phy-br-ex* on *br-ex*, an OVS flow inside *br-ex* replaces the internal VLAN tag with the actual VLAN tag associated with the VLAN provider network.

The output of the following command shows that the port number of *phy-br-ex* is **4**:

```
# ovs-ofctl show br-ex
 4(phy-br-ex): addr:32:e7:a1:6b:90:3e
     config:     0
     state:      0
     speed: 0 Mbps now, 0 Mbps max
```

The following command shows any packet that arrives on phy-br-ex (**in_port=4**) which has VLAN tag 2 (**dl_vlan=2**). Open vSwitch replaces the VLAN tag with 171 (**actions=mod_vlan_vid:171,NORMAL**) and forwards the packet to the physical interface. The command also shows any packet that arrives on phy-br-ex (**in_port=4**) which has VLAN tag 3 (**dl_vlan=3**). Open vSwitch replaces the VLAN tag with 172 (**actions=mod_vlan_vid:172,NORMAL**) and forwards the packet to the physical interface. The neutron-openvswitch-agent adds these rules.

```
# ovs-ofctl dump-flows br-ex
NXST_FLOW reply (xid=0x4):
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=6527.527s, table=0, n_packets=29211, n_bytes=2725576, idle_age=0,
priority=1 actions=NORMAL
 cookie=0x0, duration=2939.172s, table=0, n_packets=117, n_bytes=8296, idle_age=58,
priority=4,in_port=4,dl_vlan=3 actions=mod_vlan_vid:172,NORMAL
 cookie=0x0, duration=6111.389s, table=0, n_packets=145, n_bytes=9368, idle_age=98,
priority=4,in_port=4,dl_vlan=2 actions=mod_vlan_vid:171,NORMAL
 cookie=0x0, duration=6526.675s, table=0, n_packets=82, n_bytes=6700, idle_age=2462,
priority=2,in_port=4 actions=drop
```

- This packet is then forwarded to physical interface *eth1*.

The flow of incoming traffic in a VLAN provider network

The following example flow was tested on a Compute node using VLAN tag 2 for provider network provider-171 and VLAN tag 3 for provider network provider-172. The flow uses port 18 on the integration bridge br-int.

Your VLAN provider network may require a different configuration. Also, the configuration requirement for a network may differ between two different Compute nodes.

The output of the following command shows *int-br-ex* with port number 18:

```
# ovs-ofctl show br-int
 18(int-br-ex): addr:fe:b7:cb:03:c5:c1
     config:     0
     state:      0
     speed: 0 Mbps now, 0 Mbps max
```

The output of the following command shows the flow rules on br-int.

```
# ovs-ofctl dump-flows br-int
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=6770.572s, table=0, n_packets=1239, n_bytes=127795, idle_age=106,
priority=1 actions=NORMAL

 cookie=0x0, duration=3181.679s, table=0, n_packets=2605, n_bytes=246456, idle_age=0,
 priority=3,in_port=18,dl_vlan=172 actions=mod_vlan_vid:3,NORMAL

 cookie=0x0, duration=6353.898s, table=0, n_packets=5077, n_bytes=482582, idle_age=0,
 priority=3,in_port=18,dl_vlan=171 actions=mod_vlan_vid:2,NORMAL

 cookie=0x0, duration=6769.391s, table=0, n_packets=22301, n_bytes=2013101, idle_age=0,
priority=2,in_port=18 actions=drop

 cookie=0x0, duration=6770.463s, table=23, n_packets=0, n_bytes=0, idle_age=6770, priority=0
actions=drop
```

▪

Incoming flow example

This example demonstrates the the following br-int OVS flow:

```
cookie=0x0, duration=3181.679s, table=0, n_packets=2605, n_bytes=246456, idle_age=0,
priority=3,in_port=18,dl_vlan=172 actions=mod_vlan_vid:3,NORMAL
```

- A packet with VLAN tag 172 from the external network reaches the *br-ex* bridge via *eth1* on the physical node
- The packet moves to *br-int* via the patch-peer **phy-br-ex <-> int-br-ex**.
- The packet matches the flow's criteria (**in_port=18,dl_vlan=172**)
- The flow actions (**actions=mod_vlan_vid:3,NORMAL**) replace the VLAN tag 172 with internal VLAN tag 3 and forwards the packet to the instance with normal Layer 2 processing.

7.3.5. Troubleshooting instance-physical network connections on VLAN provider networks

Refer to the packet flow described in Section 7.3.4, "How does the VLAN provider network packet flow work?" when troubleshooting connectivity in a VLAN provider network. In addition, review the following configuration options:

1. Verify that physical network name is used consistently. In this example, **physnet1** is used consistently while creating the network, and within the **bridge_mapping** configuration:

```
# grep bridge_mapping /etc/neutron/plugins/ml2/openvswitch_agent.ini
bridge_mappings = physnet1:br-ex

# openstack network show provider-vlan171
...
| provider:physical_network | physnet1
...
```

2. Confirm that the network was created as **external**, is type **vlan**, and uses the correct **segmentation_id** value:

```
# openstack network show provider-vlan171
...
| provider:network_type     | vlan                                  |
| provider:physical_network | physnet1                              |
| provider:segmentation_id  | 171                                   |
...
```

3. Run **ovs-vsctl show** and verify that *br-int* and *br-ex* are connected using the patch-peer **int-br-ex** <→ **phy-br-ex**.
This connection is created while restarting *neutron-openvswitch-agent*, provided that the **bridge_mapping** is correctly configured in **/etc/neutron/plugins/ml2/openvswitch_agent.ini**. Recheck the *bridge_mapping* setting if this is not created even after restarting the service.

4. To review the flow of outgoing packets, run **ovs-ofctl dump-flows br-ex** and **ovs-ofctl dump-flows br-int**, and verify that the flows map the internal VLAN IDs to the external VLAN ID (*segmentation_id*). For incoming packets, map the external VLAN ID to the internal VLAN ID.

This flow is added by the neutron OVS agent when you spawn an instance to this network for the first time. If this flow is not created after spawning the instance, ensure that the network is created as **vlan**, is **external**, and that the **physical_network** name is correct. In addition, re-check the **bridge_mapping** settings.

5. Finally, re-check the *ifcfg-br-ex* and *ifcfg-ethx* configuration. Ensure that *br-ex* includes port *ethX*, and that both *ifcfg-br-ex* and *ifcfg-ethx* have an **UP** flag in the output of the **ip a** command.
For example, the following output shows that *eth1* is a port in *br-ex*:

```
    Bridge br-ex
        Port phy-br-ex
            Interface phy-br-ex
                type: patch
                options: {peer=int-br-ex}
        Port "eth1"
            Interface "eth1"
```

The following command shows that *eth1* has been added as a port, and that the kernel is configured to move all packets from the interface to the OVS bridge *br-ex*. This is demonstrated by the entry: **master ovs-system**.

```
# ip a
5: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq master ovs-system state
UP qlen 1000
```

7.4. ENABLING COMPUTE METADATA ACCESS

Instances connected as described in this chapter are directly attached to the provider external networks, and have external routers configured as their default gateway. No OpenStack Networking (neutron) routers are used. This means that *neutron* routers cannot be used to proxy metadata requests from instances to the *nova-metadata* server, which may result in failures while running *cloud-init*. However, this issue can be resolved by configuring the dhcp agent to proxy metadata requests. You can enable this functionality in **/etc/neutron/dhcp_agent.ini**. For example:

```
enable_isolated_metadata = True
```

7.5. FLOATING IP ADDRESSES

You can use the same network to allocate floating IP addresses to instances, even if the floating IPs are already associated with private networks. The addresses that you allocate as floating IPs from this network are bound to the *qrouter-xxx* namespace on the Network node, and perform *DNAT-SNAT* to the associated private IP address. In contrast, the IP addresses that you allocate for direct external network access are bound directly inside the instance, and allow the instance to communicate directly with external network.

CHAPTER 8. CONFIGURING PHYSICAL SWITCHES FOR OPENSTACK NETWORKING

This chapter documents the common physical switch configuration steps required for OpenStack Networking. Vendor-specific configuration is included for the following switches:

- Cisco Catalyst
- Cisco Nexus
- Cumulus Linux
- Extreme Networks EXOS
- Juniper EX Series

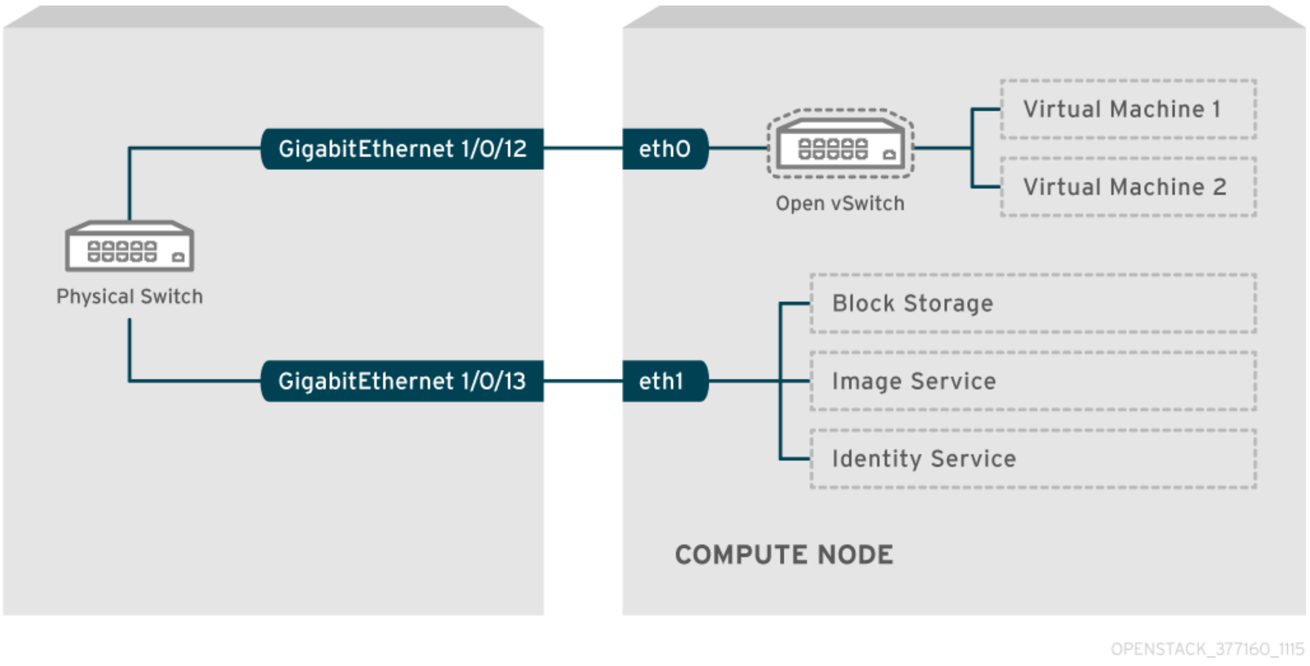
8.1. PLANNING YOUR PHYSICAL NETWORK ENVIRONMENT

The physical network adapters in your OpenStack nodes carry different types of network traffic, such as instance traffic, storage data, or authentication requests. The type of traffic these NICs carry affects how you must configure the ports on the physical switch.

First, you must decide which physical NICs on your Compute node you want to carry which types of traffic. Then, when the NIC is cabled to a physical switch port, you must configure the switch port to allow trunked or general traffic.

For example, the following diagram depicts a Compute node with two NICs, eth0 and eth1. Each NIC is cabled to a Gigabit Ethernet port on a physical switch, with eth0 carrying instance traffic, and eth1 providing connectivity for OpenStack services:

Sample network layout



NOTE

This diagram does not include any additional redundant NICs required for fault tolerance.

For information on network interface bonds, see the Network Interface Bonding chapter of the Advanced Overcloud Customization guide.

8.2. CONFIGURING A CISCO CATALYST SWITCH

8.2.1. About trunk ports

With OpenStack Networking you can connect instances to the VLANs that already exist on your physical network. The term *trunk* is used to describe a port that allows multiple VLANs to traverse through the same port. Using these ports, VLANs can span across multiple switches, including virtual switches. For example, traffic tagged as VLAN110 in the physical network reaches the Compute node, where the 8021q module directs the tagged traffic to the appropriate VLAN on the vSwitch.

8.2.2. Configuring trunk ports for a Cisco Catalyst switch

- If using a Cisco Catalyst switch running Cisco IOS, you might use the following configuration syntax to allow traffic for VLANs 110 and 111 to pass through to your instances.
  This configuration assumes that your physical node has an ethernet cable connected to interface GigabitEthernet1/0/12 on the physical switch.

  IMPORTANT

  These values are examples. You must change the values in this example to match those in your environment. Copying and pasting these values into your switch configuration without adjustment can result in an unexpected outage.

```
interface GigabitEthernet1/0/12
  description Trunk to Compute Node
  spanning-tree portfast trunk
  switchport trunk encapsulation dot1q
  switchport mode trunk
  switchport trunk native vlan 2
  switchport trunk allowed vlan 2,110,111
```

  Use the following list to understand these parameters:

Field	Description
interface GigabitEthernet1/0/12	The switch port that the NIC of the X node connects to. Ensure that you replace the **GigabitEthernet1/0/12** value with the correct port value for your environment. Use the show interface command to view a list of ports.
description Trunk to Compute Node	A unique and descriptive value that you can use to identify this interface.
spanning-tree portfast trunk	If your environment uses STP, set this value to instruct Port Fast that this port is used to trunk traffic.

Field	Description
switchport trunk encapsulation dot1q	Enables the 802.1q trunking standard (rather than ISL). This value varies depending on the configuration that your switch supports.
switchport mode trunk	Configures this port as a trunk port, rather than an access port, meaning that it allows VLAN traffic to pass through to the virtual switches.
switchport trunk native vlan 2	Set a native VLAN to instruct the switch where to send untagged (non-VLAN) traffic.
switchport trunk allowed vlan 2,110,111	Defines which VLANs are allowed through the trunk.

8.2.3. About access ports

Not all NICs on your Compute node carry instance traffic, and so you do not need to configure all NICs to allow multiple VLANs to pass through. Access ports require only one VLAN, and might fulfill other operational requirements, such as transporting management traffic or Block Storage data. These ports are commonly known as access ports and usually require a simpler configuration than trunk ports.

8.2.4. Configuring access ports for a Cisco Catalyst switch

- Using the example from the Sample network layout diagram, GigabitEthernet1/0/13 (on a Cisco Catalyst switch) is configured as an access port for **eth1**.
  In this configuration,your physical node has an ethernet cable connected to interface GigabitEthernet1/0/12 on the physical switch.

IMPORTANT

These values are examples. You must change the values in this example to match those in your environment. Copying and pasting these values into your switch configuration without adjustment can result in an unexpected outage.

```
interface GigabitEthernet1/0/13
 description Access port for Compute Node
 switchport mode access
 switchport access vlan 200
 spanning-tree portfast
```

These settings are described below:

Field	Description

Field	Description
interface GigabitEthernet1/0/13	The switch port that the NIC of the X node connects to. Ensure that you replace the **GigabitEthernet1/0/12** value with the correct port value for your environment. Use the show interface command to view a list of ports.
description Access port for Compute Node	A unique and descriptive value that you can use to identify this interface.
switchport mode access	Configures this port as an access port, rather than a trunk port.
switchport access vlan 200	Configures the port to allow traffic on VLAN 200. You must configure your Compute node with an IP address from this VLAN.
spanning-tree portfast	If using STP, set this value to instruct STP not to attempt to initialize this as a trunk, allowing for quicker port handshakes during initial connections (such as server reboot).

8.2.5. About LACP port aggregation

You can use LACP to bundle multiple physical NICs together to form a single logical channel. Also known as 802.3ad (or bonding mode 4 in Linux), LACP creates a dynamic bond for load-balancing and fault tolerance. You must configure LACP at both physical ends: on the physical NICs, and on the physical switch ports.

8.2.6. Configuring LACP on the physical NIC

1. Edit the */home/stack/network-environment.yaml* file:

```
- type: linux_bond
  name: bond1
  mtu: 9000
  bonding_options:{get_param: BondInterfaceOvsOptions};
  members:
    - type: interface
      name: nic3
      mtu: 9000
      primary: true
    - type: interface
      name: nic4
      mtu: 9000
```

2. Configure the Open vSwitch bridge to use LACP:

```
BondInterfaceOvsOptions:
    "mode=802.3ad"
```

For information on configuring network bonds, see the Network Interface Bonding chapter of the Advanced Overcloud Customization guide.

8.2.7. Configuring LACP for a Cisco Catalyst switch

In this example, the Compute node has two NICs using VLAN 100:

1. Physically connect both NICs on the Compute node to the switch (for example, ports 12 and 13).
2. Create the LACP port channel:

```
interface port-channel1
  switchport access vlan 100
  switchport mode access
  spanning-tree guard root
```

3. Configure switch ports 12 (Gi1/0/12) and 13 (Gi1/0/13):

```
sw01# config t
Enter configuration commands, one per line.  End with CNTL/Z.

sw01(config) interface GigabitEthernet1/0/12
   switchport access vlan 100
   switchport mode access
   speed 1000
   duplex full
   channel-group 10 mode active
   channel-protocol lacp

interface GigabitEthernet1/0/13
  switchport access vlan 100
  switchport mode access
  speed 1000
  duplex full
  channel-group 10 mode active
  channel-protocol lacp
```

4. Review your new port channel. The resulting output lists the new port-channel **Po1**, with member ports **Gi1/0/12** and **Gi1/0/13**:

```
sw01# show etherchannel summary
<snip>

Number of channel-groups in use: 1
Number of aggregators:           1

Group  Port-channel  Protocol    Ports
------+-------------+-----------+-----------------------------------------------
1      Po1(SD)         LACP      Gi1/0/12(D)  Gi1/0/13(D)
```

NOTE

Remember to apply your changes by copying the running-config to the startup-config: **copy running-config startup-config**.

8.2.8. About MTU settings

You must adjust your MTU size for certain types of network traffic. For example, jumbo frames (9000 bytes) are required for certain NFS or iSCSI traffic.

NOTE

You must change MTU settings from end-to-end on all hops that the traffic is expected to pass through, including any virtual switches. For information on changing the MTU in your OpenStack environment, see Chapter 9, *Configure maximum transmission unit (MTU) settings*.

8.2.9. Configuring MTU settings for a Cisco Catalyst switch

Complete the steps in this example procedure to enable jumbo frames on your Cisco Catalyst 3750 switch.

1. Review the current MTU settings:

```
sw01# show system mtu

System MTU size is 1600 bytes
System Jumbo MTU size is 1600 bytes
System Alternate MTU size is 1600 bytes
Routing MTU size is 1600 bytes
```

2. MTU settings are changed switch-wide on 3750 switches, and not for individual interfaces. Run the following commands to configure the switch to use jumbo frames of 9000 bytes. You might prefer to configure the MTU settings for individual interfaces, if your switch supports this feature.

```
sw01# config t
Enter configuration commands, one per line.  End with CNTL/Z.

sw01(config)# system mtu jumbo 9000
Changes to the system jumbo MTU will not take effect until the next reload is done
```

NOTE

Remember to save your changes by copying the running-config to the startup-config: **copy running-config startup-config**.

3. Reload the switch to apply the change.

IMPORTANT

Reloading the switch causes a network outage for any devices that are dependent on the switch. Therefore, reload the switch only during a scheduled maintenance period.

```
sw01# reload
Proceed with reload? [confirm]
```

4. After the switch reloads, confirm the new jumbo MTU size.
   The exact output may differ depending on your switch model. For example, **System MTU** might apply to non-Gigabit interfaces, and **Jumbo MTU** might describe all Gigabit interfaces.

```
sw01# show system mtu

System MTU size is 1600 bytes
System Jumbo MTU size is 9000 bytes
System Alternate MTU size is 1600 bytes
Routing MTU size is 1600 bytes
```

8.2.10. About LLDP discovery

The **ironic-python-agent** service listens for LLDP packets from connected switches. The collected information can include the switch name, port details, and available VLANs. Similar to Cisco Discovery Protocol (CDP), LLDP assists with the discovery of physical hardware during the director introspection process.

8.2.11. Configuring LLDP for a Cisco Catalyst switch

1. Run the **lldp run** command to enable LLDP globally on your Cisco Catalyst switch:

```
sw01# config t
Enter configuration commands, one per line.  End with CNTL/Z.

sw01(config)# lldp run
```

2. View any neighboring LLDP-compatible devices:

```
sw01# show lldp neighbor
Capability codes:
   (R) Router, (B) Bridge, (T) Telephone, (C) DOCSIS Cable Device
   (W) WLAN Access Point, (P) Repeater, (S) Station, (O) Other

Device ID           Local Intf     Hold-time  Capability      Port ID
DEP42037061562G3     Gi1/0/11       180        B,T            422037061562G3:P1

Total entries displayed: 1
```

NOTE

Remember to save your changes by copying the running-config to the startup-config: **copy running-config startup-config**.

8.3. CONFIGURING A CISCO NEXUS SWITCH

8.3.1. About trunk ports

With OpenStack Networking you can connect instances to the VLANs that already exist on your physical network. The term *trunk* is used to describe a port that allows multiple VLANs to traverse through the same port. Using these ports, VLANs can span across multiple switches, including virtual switches. For example, traffic tagged as VLAN110 in the physical network reaches the Compute node, where the 8021q module directs the tagged traffic to the appropriate VLAN on the vSwitch.

8.3.2. Configuring trunk ports for a Cisco Nexus switch

- If using a Cisco Nexus you might use the following configuration syntax to allow traffic for VLANs 110 and 111 to pass through to your instances.
  This configuration assumes that your physical node has an ethernet cable connected to interface **Ethernet1/12** on the physical switch.

  IMPORTANT

  These values are examples. You must change the values in this example to match those in your environment. Copying and pasting these values into your switch configuration without adjustment can result in an unexpected outage.

```
interface Ethernet1/12
  description Trunk to Compute Node
  switchport mode trunk
  switchport trunk allowed vlan 2,110,111
  switchport trunk native vlan 2
end
```

8.3.3. About access ports

Not all NICs on your Compute node carry instance traffic, and so you do not need to configure all NICs to allow multiple VLANs to pass through. Access ports require only one VLAN, and might fulfill other operational requirements, such as transporting management traffic or Block Storage data. These ports are commonly known as access ports and usually require a simpler configuration than trunk ports.

8.3.4. Configuring access ports for a Cisco Nexus switch

- Using the example from the Sample network layout diagram, Ethernet1/13 (on a Cisco Nexus switch) is configured as an access port for **eth1**. This configuration assumes that your physical node has an ethernet cable connected to interface **Ethernet1/13** on the physical switch.

  IMPORTANT

  These values are examples. You must change the values in this example to match those in your environment. Copying and pasting these values into your switch configuration without adjustment can result in an unexpected outage.

```
interface Ethernet1/13
 description Access port for Compute Node
 switchport mode access
 switchport access vlan 200
```

8.3.5. About LACP port aggregation

You can use LACP to bundle multiple physical NICs together to form a single logical channel. Also known as 802.3ad (or bonding mode 4 in Linux), LACP creates a dynamic bond for load-balancing and fault tolerance. You must configure LACP at both physical ends: on the physical NICs, and on the physical switch ports.

8.3.6. Configuring LACP on the physical NIC

1. Edit the */home/stack/network-environment.yaml* file:

```
- type: linux_bond
  name: bond1
  mtu: 9000
  bonding_options:{get_param: BondInterfaceOvsOptions};
  members:
    - type: interface
      name: nic3
      mtu: 9000
      primary: true
    - type: interface
      name: nic4
      mtu: 9000
```

2. Configure the Open vSwitch bridge to use LACP:

```
BondInterfaceOvsOptions:
    "mode=802.3ad"
```

For information on configuring network bonds, see the Network Interface Bonding chapter of the Advanced Overcloud Customization guide.

8.3.7. Configuring LACP for a Cisco Nexus switch

In this example, the Compute node has two NICs using VLAN 100:

1. Physically connect the Compute node NICs to the switch (for example, ports 12 and 13).

2. Confirm that LACP is enabled:

```
(config)# show feature | include lacp
lacp                  1         enabled
```

3. Configure ports 1/12 and 1/13 as access ports, and as members of a channel group. Depending on your deployment, you can deploy trunk interfaces rather than access interfaces.

   For example, for Cisco UCI the NICs are virtual interfaces, so you might prefer to configure access ports exclusively. Often these interfaces contain VLAN tagging configurations.

```
interface Ethernet1/13
 description Access port for Compute Node
 switchport mode access
 switchport access vlan 200
 channel-group 10 mode active

interface Ethernet1/13
 description Access port for Compute Node
```

```
switchport mode access
switchport access vlan 200
channel-group 10 mode active
```

8.3.8. About MTU settings

You must adjust your MTU size for certain types of network traffic. For example, jumbo frames (9000 bytes) are required for certain NFS or iSCSI traffic.

NOTE

You must change MTU settings from end-to-end on all hops that the traffic is expected to pass through, including any virtual switches. For information on changing the MTU in your OpenStack environment, see Chapter 9, *Configure maximum transmission unit (MTU) settings*.

8.3.9. Configuring MTU settings for a Cisco Nexus 7000 switch

Apply MTU settings to a single interface on 7000-series switches.

- Run the following commands to configure interface 1/12 to use jumbo frames of 9000 bytes:

```
interface ethernet 1/12
  mtu 9216
  exit
```

8.3.10. About LLDP discovery

The **ironic-python-agent** service listens for LLDP packets from connected switches. The collected information can include the switch name, port details, and available VLANs. Similar to Cisco Discovery Protocol (CDP), LLDP assists with the discovery of physical hardware during the director introspection process.

8.3.11. Configuring LLDP for a Cisco Nexus 7000 switch

- You can enable LLDP for individual interfaces on Cisco Nexus 7000-series switches:

```
interface ethernet 1/12
  lldp transmit
  lldp receive
  no lacp suspend-individual
  no lacp graceful-convergence

interface ethernet 1/13
  lldp transmit
  lldp receive
  no lacp suspend-individual
  no lacp graceful-convergence
```

NOTE

Remember to save your changes by copying the running-config to the startup-config: **copy running-config startup-config**.

8.4. CONFIGURING A CUMULUS LINUX SWITCH

8.4.1. About trunk ports

With OpenStack Networking you can connect instances to the VLANs that already exist on your physical network. The term *trunk* is used to describe a port that allows multiple VLANs to traverse through the same port. Using these ports, VLANs can span across multiple switches, including virtual switches. For example, traffic tagged as VLAN110 in the physical network reaches the Compute node, where the 8021q module directs the tagged traffic to the appropriate VLAN on the vSwitch.

8.4.2. Configuring trunk ports for a Cumulus Linux switch

- Use the following configuration syntax to allow traffic for VLANs 100 and 200 to pass through to your instances.
  This configuration assumes that your physical node has transceivers connected to switch ports swp1 and swp2 on the physical switch.

  IMPORTANT

  These values are examples. You must change the values in this example to match those in your environment. Copying and pasting these values into your switch configuration without adjustment can result in an unexpected outage.

```
auto bridge
iface bridge
  bridge-vlan-aware yes
  bridge-ports glob swp1-2
  bridge-vids 100 200
```

8.4.3. About access ports

Not all NICs on your Compute node carry instance traffic, and so you do not need to configure all NICs to allow multiple VLANs to pass through. Access ports require only one VLAN, and might fulfill other operational requirements, such as transporting management traffic or Block Storage data. These ports are commonly known as access ports and usually require a simpler configuration than trunk ports.

8.4.4. Configuring access ports for a Cumulus Linux switch

- Using the example from the Sample network layout diagram, **swp1** (on a Cumulus Linux switch) is configured as an access port.
  This configuration assumes that your physical node has an ethernet cable connected to the interface on the physical switch. Cumulus Linux switches use **eth** for management interfaces and **swp** for access/trunk ports.

  IMPORTANT

  These values are examples. You must change the values in this example to match those in your environment. Copying and pasting these values into your switch configuration without adjustment can result in an unexpected outage.

```
auto bridge
iface bridge
```

```
  bridge-vlan-aware yes
  bridge-ports glob swp1-2
  bridge-vids 100 200


auto swp1
iface swp1
  bridge-access 100


auto swp2
iface swp2
  bridge-access 200
```

8.4.5. About LACP port aggregation

You can use LACP to bundle multiple physical NICs together to form a single logical channel. Also known as 802.3ad (or bonding mode 4 in Linux), LACP creates a dynamic bond for load-balancing and fault tolerance. You must configure LACP at both physical ends: on the physical NICs, and on the physical switch ports.

8.4.6. About MTU settings

You must adjust your MTU size for certain types of network traffic. For example, jumbo frames (9000 bytes) are required for certain NFS or iSCSI traffic.

NOTE

You must change MTU settings from end-to-end on all hops that the traffic is expected to pass through, including any virtual switches. For information on changing the MTU in your OpenStack environment, see Chapter 9, *Configure maximum transmission unit (MTU) settings*.

8.4.7. Configuring MTU settings for a Cumulus Linux switch

- This example enables jumbo frames on your Cumulus Linux switch.

```
auto swp1
iface swp1
  mtu 9000
```

NOTE

Remember to apply your changes by reloading the updated configuration: **sudo ifreload -a**

8.4.8. About LLDP discovery

The **ironic-python-agent** service listens for LLDP packets from connected switches. The collected information can include the switch name, port details, and available VLANs. Similar to Cisco Discovery Protocol (CDP), LLDP assists with the discovery of physical hardware during the director introspection process.

8.4.9. Configuring LLDP for a Cumulus Linux switch

By default, the LLDP service lldpd runs as a daemon and starts when the switch boots.

- To view all LLDP neighbors on all ports/interfaces, run the following command:

```
cumulus@switch$ netshow lldp
Local Port  Speed  Mode        Remote Port   Remote Host Summary
----------  ---    ---------    -----  -----  ----------- --------
eth0        10G    Mgmt        ====   swp6   mgmt-sw     IP: 10.0.1.11/24
swp51       10G    Interface/L3 ====   swp1   spine01     IP: 10.0.0.11/32
swp52       10G    Interface/L  ====   swp1   spine02     IP: 10.0.0.11/32
```

8.5. CONFIGURING A EXTREME EXOS SWITCH

8.5.1. About trunk ports

With OpenStack Networking you can connect instances to the VLANs that already exist on your physical network. The term *trunk* is used to describe a port that allows multiple VLANs to traverse through the same port. Using these ports, VLANs can span across multiple switches, including virtual switches. For example, traffic tagged as VLAN110 in the physical network reaches the Compute node, where the 8021q module directs the tagged traffic to the appropriate VLAN on the vSwitch.

8.5.2. Configuring trunk ports on an Extreme Networks EXOS switch

- If using an X-670 series switch, refer to the following example to allow traffic for VLANs 110 and 111 to pass through to your instances.
  This configuration assumes that your physical node has an ethernet cable connected to interface 24 on the physical switch. In this example, DATA and MNGT are the VLAN names.

IMPORTANT

These values are examples. You must change the values in this example to match those in your environment. Copying and pasting these values into your switch configuration without adjustment can result in an unexpected outage.

```
#create vlan DATA tag 110
#create vlan MNGT tag 111
#configure vlan DATA add ports 24 tagged
#configure vlan MNGT add ports 24 tagged
```

8.5.3. About access ports

Not all NICs on your Compute node carry instance traffic, and so you do not need to configure all NICs to allow multiple VLANs to pass through. Access ports require only one VLAN, and might fulfill other operational requirements, such as transporting management traffic or Block Storage data. These ports are commonly known as access ports and usually require a simpler configuration than trunk ports.

8.5.4. Configuring access ports for an Extreme Networks EXOS switch

- In this configuration example, on a Extreme Networks X-670 series switch, **10** is used as an access port for **eth1**.

This configuration assumes that your physical node has an ethernet cable connected to interface **10** on the physical switch.

IMPORTANT

These values are examples. You must change the values in this example to match those in your environment. Copying and pasting these values into your switch configuration without adjustment can result in an unexpected outage.

```
create vlan VLANNAME tag NUMBER
configure vlan Default delete ports PORTSTRING
configure vlan VLANNAME add ports PORTSTRING untagged
```

For example:

```
#create vlan DATA tag 110
#configure vlan Default delete ports 10
#configure vlan DATA add ports 10 untagged
```

8.5.5. About LACP port aggregation

You can use LACP to bundle multiple physical NICs together to form a single logical channel. Also known as 802.3ad (or bonding mode 4 in Linux), LACP creates a dynamic bond for load-balancing and fault tolerance. You must configure LACP at both physical ends: on the physical NICs, and on the physical switch ports.

8.5.6. Configuring LACP on the physical NIC

1. Edit the */home/stack/network-environment.yaml* file:

```
- type: linux_bond
  name: bond1
  mtu: 9000
  bonding_options:{get_param: BondInterfaceOvsOptions};
  members:
    - type: interface
      name: nic3
      mtu: 9000
      primary: true
    - type: interface
      name: nic4
      mtu: 9000
```

2. Configure the Open vSwitch bridge to use LACP:

```
BondInterfaceOvsOptions:
    "mode=802.3ad"
```

For information on configuring network bonds, see the Network Interface Bonding chapter of the Advanced Overcloud Customization guide.

8.5.7. Configuring LACP on an Extreme Networks EXOS switch

- In this example, the Compute node has two NICs using VLAN 100:

  ```
  enable sharing MASTERPORT grouping ALL_LAG_PORTS lacp
  configure vlan VLANNAME add ports PORTSTRING tagged
  ```

  For example:

  ```
  #enable sharing 11 grouping 11,12 lacp
  #configure vlan DATA add port 11 untagged
  ```

  NOTE

  You might need to adjust the timeout period in the LACP negotiation script. For more information, see https://gtacknowledge.extremenetworks.com/articles/How_To/LACP-configured-ports-interfere-with-PXE-DHCP-on-servers

8.5.8. About MTU settings

You must adjust your MTU size for certain types of network traffic. For example, jumbo frames (9000 bytes) are required for certain NFS or iSCSI traffic.

NOTE

You must change MTU settings from end-to-end on all hops that the traffic is expected to pass through, including any virtual switches. For information on changing the MTU in your OpenStack environment, see Chapter 9, *Configure maximum transmission unit (MTU) settings*.

8.5.9. Configuring MTU settings on an Extreme Networks EXOS switch

- Run the commands in this example to enable jumbo frames on an Extreme Networks EXOS switch and configure support for forwarding IP packets with 9000 bytes:

  ```
  enable jumbo-frame ports PORTSTRING
  configure ip-mtu 9000 vlan VLANNAME
  ```

  For example:

  ```
  # enable jumbo-frame ports 11
  # configure ip-mtu 9000 vlan DATA
  ```

8.5.10. About LLDP discovery

The **ironic-python-agent** service listens for LLDP packets from connected switches. The collected information can include the switch name, port details, and available VLANs. Similar to Cisco Discovery Protocol (CDP), LLDP assists with the discovery of physical hardware during the director introspection process.

8.5.11. Configuring LLDP settings on an Extreme Networks EXOS switch

- In this example, LLDP is enabled on an Extreme Networks EXOS switch. **11** represents the port string:

```
enable lldp ports 11
```

8.6. CONFIGURING A JUNIPER EX SERIES SWITCH

8.6.1. About trunk ports

With OpenStack Networking you can connect instances to the VLANs that already exist on your physical network. The term *trunk* is used to describe a port that allows multiple VLANs to traverse through the same port. Using these ports, VLANs can span across multiple switches, including virtual switches. For example, traffic tagged as VLAN110 in the physical network reaches the Compute node, where the 8021q module directs the tagged traffic to the appropriate VLAN on the vSwitch.

8.6.2. Configuring trunk ports for a Juniper EX Series switch

- If using a Juniper EX series switch running Juniper JunOS, use the following configuration syntax to allow traffic for VLANs 110 and 111 to pass through to your instances.
  This configuration assumes that your physical node has an ethernet cable connected to interface ge-1/0/12 on the physical switch.

  IMPORTANT

  These values are examples. You must change the values in this example to match those in your environment. Copying and pasting these values into your switch configuration without adjustment can result in an unexpected outage.

```
ge-1/0/12 {
          description Trunk to Compute Node;
              unit 0 {
                  family ethernet-switching {
                      port-mode trunk;
                      vlan {
                          members [110 111];
                           }
                      native-vlan-id 2;
                  }
              }
}
```

8.6.3. About access ports

Not all NICs on your Compute node carry instance traffic, and so you do not need to configure all NICs to allow multiple VLANs to pass through. Access ports require only one VLAN, and might fulfill other operational requirements, such as transporting management traffic or Block Storage data. These ports are commonly known as access ports and usually require a simpler configuration than trunk ports.

8.6.4. Configuring access ports for a Juniper EX Series switch

- This example on, a Juniper EX series switch, shows **ge-1/0/13** as an access port for **eth1**.

This configuration assumes that your physical node has an ethernet cable connected to interface **ge-1/0/13** on the physical switch.

IMPORTANT

These values are examples. You must change the values in this example to match those in your environment. Copying and pasting these values into your switch configuration without adjustment can result in an unexpected outage.

```
ge-1/0/13 {
        description Access port for Compute Node
            unit 0 {
                family ethernet-switching {
                    port-mode access;
                    vlan {
                        members 200;
                         }
                    native-vlan-id 2;
                }
            }
}
```

8.6.5. About LACP port aggregation

You can use LACP to bundle multiple physical NICs together to form a single logical channel. Also known as 802.3ad (or bonding mode 4 in Linux), LACP creates a dynamic bond for load-balancing and fault tolerance. You must configure LACP at both physical ends: on the physical NICs, and on the physical switch ports.

8.6.6. Configuring LACP on the physical NIC

1. Edit the */home/stack/network-environment.yaml* file:

```
- type: linux_bond
  name: bond1
  mtu: 9000
  bonding_options:{get_param: BondInterfaceOvsOptions};
  members:
    - type: interface
      name: nic3
      mtu: 9000
      primary: true
    - type: interface
      name: nic4
      mtu: 9000
```

2. Configure the Open vSwitch bridge to use LACP:

```
BondInterfaceOvsOptions:
    "mode=802.3ad"
```

For information on configuring network bonds, see the Network Interface Bonding chapter of the Advanced Overcloud Customization guide.

8.6.7. Configuring LACP for a Juniper EX Series switch

In this example, the Compute node has two NICs using VLAN 100.

1. Physically connect the Compute node’s two NICs to the switch (for example, ports 12 and 13).

2. Create the port aggregate:

```
chassis {
    aggregated-devices {
        ethernet {
            device-count 1;
        }
    }
}
```

3. Configure switch ports 12 (ge-1/0/12) and 13 (ge-1/0/13) to join the port aggregate **ae1**:

```
interfaces {
    ge-1/0/12 {
        gigether-options {
            802.3ad ae1;
        }
    }
    ge-1/0/13 {
        gigether-options {
            802.3ad ae1;
            }
        }
}
```

NOTE

For Red Hat OpenStack Platform director deployments, in order to PXE boot from the bond, you must configure one of the bond members as lacp force-up toensure that only one bond member comes up during introspection and first boot. The bond member that you configure with lacp force-up must be the same bond member that has the MAC address in *instackenv.json* (the MAC address known to ironic must be the same MAC address configured with force-up).

4. Enable LACP on port aggregate **ae1**:

```
interfaces {
    ae1 {
        aggregated-ether-options {
            lacp {
                active;
            }
        }
    }
}
```

5. Add aggregate **ae1** to VLAN 100:

```
interfaces {
    ae1 {
        vlan-tagging;
        native-vlan-id 2;
        unit 100 {
            vlan-id 100;
        }
    }
}
```

6. Review your new port channel. The resulting output lists the new port aggregate **ae1** with member ports **ge-1/0/12** and **ge-1/0/13**:

```
> show lacp statistics interfaces ae1

Aggregated interface: ae1
LACP Statistics: LACP Rx LACP Tx Unknown Rx Illegal Rx
ge-1/0/12 0 0 0 0
ge-1/0/13 0 0 0 0
```

NOTE

Remember to apply your changes by running the **commit** command.

8.6.8. About MTU settings

You must adjust your MTU size for certain types of network traffic. For example, jumbo frames (9000 bytes) are required for certain NFS or iSCSI traffic.

NOTE

You must change MTU settings from end-to-end on all hops that the traffic is expected to pass through, including any virtual switches. For information on changing the MTU in your OpenStack environment, see Chapter 9, *Configure maximum transmission unit (MTU) settings*.

8.6.9. Configuring MTU settings for a Juniper EX Series switch

This example enables jumbo frames on your Juniper EX4200 switch.

NOTE

The MTU value is calculated differently depending on whether you are using Juniper or Cisco devices. For example, **9216** on Juniper would equal to **9202** for Cisco. The extra bytes are used for L2 headers, where Cisco adds this automatically to the MTU value specified, but the usable MTU will be 14 bytes smaller than specified when using Juniper. So in order to support an MTU of **9000** on the VLANs, the MTU of **9014** would have to be configured on Juniper.

1. For Juniper EX series switches, MTU settings are set for individual interfaces. These commands configure jumbo frames on the **ge-1/0/14** and **ge-1/0/15** ports:

```
set interfaces ge-1/0/14 mtu 9216
set interfaces ge-1/0/15 mtu 9216
```

NOTE

Remember to save your changes by running the **commit** command.

2. If using a LACP aggregate, you will need to set the MTU size there, and not on the member NICs. For example, this setting configures the MTU size for the ae1 aggregate:

```
 set interfaces ae1 mtu 9216
```

8.6.10. About LLDP discovery

The **ironic-python-agent** service listens for LLDP packets from connected switches. The collected information can include the switch name, port details, and available VLANs. Similar to Cisco Discovery Protocol (CDP), LLDP assists with the discovery of physical hardware during the director introspection process.

8.6.11. Configuring LLDP for a Juniper EX Series switch

You can enable LLDP globally for all interfaces, or just for individual ones:

- Use the following too enable LLDP globally on your Juniper EX 4200 switch:

```
lldp {
 interface all{
  enable;
 }
 }
}
```

- Use the following to enable LLDP for the single interface **ge-1/0/14**:

```
lldp {
 interface ge-1/0/14{
  enable;
 }
 }
}
```

NOTE

Remember to apply your changes by running the **commit** command.

PART II. ADVANCED CONFIGURATION

Contains cookbook-style scenarios for advanced Red Hat OpenStack Platform Networking features.

CHAPTER 9. CONFIGURE MAXIMUM TRANSMISSION UNIT (MTU) SETTINGS

9.1. MTU OVERVIEW

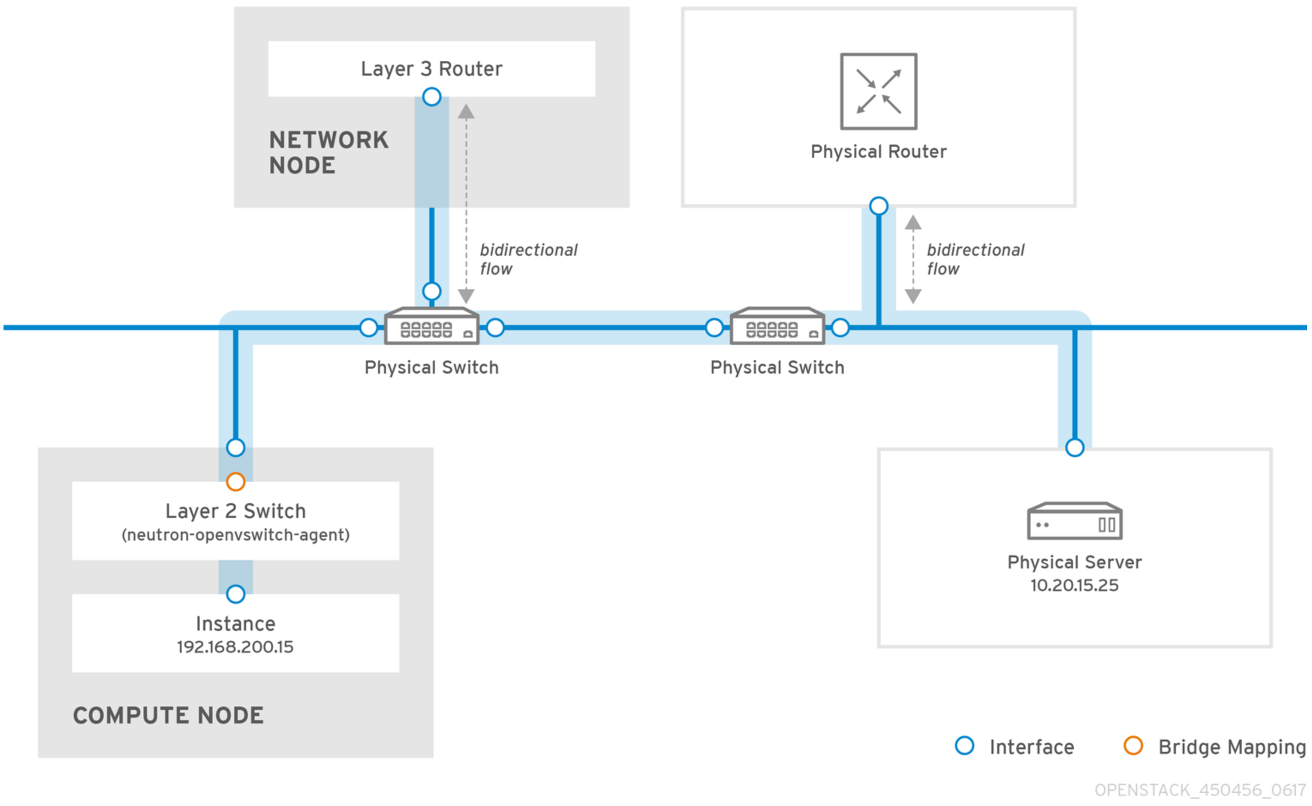
OpenStack Networking can calculate the largest possible maximum transmission unit (MTU) size that you can apply safely to instances. The MTU value specifies the maximum amount of data that a single network packet can transfer; this number is variable depending on the most appropriate size for the application. For example, NFS shares might require a different MTU size to that of a VoIP application.

NOTE

You can use the neutron net-show command to view the largest possible MTU values that OpenStack Networking calculates. **net-mtu** is a neutron API extension that is not present in some implementations. The MTU value that you require can be advertised to DHCPv4 clients for automatic configuration, if supported by the instance, as well as to IPv6 clients through Router Advertisement (RA) packets. To send Router Advertisements, the network must be attached to a router.

You must configure MTU settings consistently from end-to-end. This means that the MTU setting must be the same at every point the packet passes through, including the VM, the virtual network infrastructure, the physical network, and the destination server.

For example, the circles in the following diagram indicate the various points where an MTU value must be adjusted for traffic between an instance and a physical server. You must change the MTU value for very interface that handles network traffic to accommodate packets of a particular MTU size. This is necessary if traffic travels from the instance *192.168.200.15* through to the physical server *10.20.15.25*:



Inconsistent MTU values can result in several network issues, the most common being random packet loss that results in connection drops and slow network performance. Such issues are problematic to troubleshoot because you must identify and examine every possible network point to ensure it has the correct MTU value.

9.2. CONFIGURING MTU SETTINGS IN DIRECTOR

This example demonstrates how to set the MTU using the NIC config templates. You must set the MTU on the bridge, bond (if applicable), interface(s), and VLAN(s):

```
-
  type: ovs_bridge
  name: br-isolated
  use_dhcp: false
  mtu: 9000    # <--- Set MTU
  members:
    -
      type: ovs_bond
      name: bond1
      mtu: 9000    # <--- Set MTU
      ovs_options: {get_param: BondInterfaceOvsOptions}
      members:
        -
          type: interface
          name: ens15f0
          mtu: 9000    # <--- Set MTU
          primary: true
        -
          type: interface
          name: enp131s0f0
          mtu: 9000    # <--- Set MTU
    -
      type: vlan
      device: bond1
      vlan_id: {get_param: InternalApiNetworkVlanID}
      mtu: 9000    # <--- Set MTU
      addresses:
      -
        ip_netmask: {get_param: InternalApiIpSubnet}
    -
      type: vlan
      device: bond1
      mtu: 9000    # <--- Set MTU
      vlan_id: {get_param: TenantNetworkVlanID}
      addresses:
      -
        ip_netmask: {get_param: TenantIpSubnet}
```

9.3. REVIEWING THE RESULTING MTU CALCULATION

You can view the calculated MTU value, which is the largest possible MTU value that instances can use. Use this calculated MTU value to configure all interfaces involved in the path of network traffic.

```
# openstack network show <network>
```

CHAPTER 10. CONFIGURING QUALITY OF SERVICE (QOS) POLICIES

You can offer varying service levels for instances by using quality of service (QoS) policies to apply rate limits to egress and ingress traffic.

You can apply QoS policies to individual ports. You can also apply QoS policies to a project network, where ports with no specific policy attached inherit the policy.

NOTE

Internal network owned ports, such as DHCP and internal router ports, are excluded from network policy application.

You can apply, modify, or remove QoS policies dynamically. However, for guaranteed minimum bandwidth QoS policies, you can only apply modifications when there are no instances that use any of the ports the policy is assigned to.

10.1. QOS RULES

The following rule types define the limits of a particular quality of service (QoS) policy:

- **bandwidth_limit**: Provides bandwidth limitations on networks, ports or floating IPs. If implemented, any traffic that exceeds the specified rate is dropped.
- **minimum_bandwidth**: Provides minimum bandwidth constraints on certain types of traffic. If implemented, best efforts are made to provide no less than the specified bandwidth to each port on which the rule is applied.
- **dscp_marking**: Marks network traffic with a Differentiated Services Code Point (DSCP) value.

10.1.1. Related procedures

The following procedures describe how to create a QoS policy with each rule type:

- Creating and applying a bandwidth limit QoS policy and rule
- Creating and applying a guaranteed minimum bandwidth QoS policy and rule
- Creating and applying a DSCP marking QoS policy and rule for egress traffic

10.2. CREATING AND APPLYING A QOS POLICY AND RULE

To create a quality of service (QoS) policy and rule and apply the policy to a port, complete the following steps:

Procedure

1. Identify the ID of the project you want to create the QoS policy for:

```
(overcloud) $ openstack project list
```

```
-------------------------------------------+
| ID                               | Name    |
------------------------------------------+
| 8c409e909fb34d69bc896ab358317d60 | admin   |
| 92b6c16c7c7244378a062be0bfd55fa0 | service |
------------------------------------------+
```

2. Create the new QoS policy:

```
(overcloud) $ openstack network qos policy create --share --project <project_ID>
<policy_name>
```

3. Create new rules for the QoS policy:

```
(overcloud) $ openstack network qos rule create --type <rule-type> [rule properties]
<policy_name>
```

Table 10.1. Rule properties

Property	Description
max_kbps	The maximum rate (in Kbps) that the instance can send.
max_burst_kbps	The maximum amount of data (in kbits) that the port can send in an instant if the token buffer is full. The token buffer replenishes at a "max_kbps" rate. The burst value for TCP traffic can be set as 80% of desired bandwidth limit value. For example, if the bandwidth limit is set to 1000kbps then a burst value of 800kbps is sufficient. NOTE ● If the burst value is set too low, bandwidth usage is throttled even with a proper bandwidth limit setting, resulting in a lower than expected bandwidth. ● If the configured burst value is too high, too few packets could be limited, resulting in a higher than expected bandwidth limit.
min-kbps	The minimum bandwidth (in Kbps) guaranteed to an instance.

Property	Description
ingress/egress	The direction of traffic the rule is applied to. From the perspective of a cloud server, ingress indicates download, and egress indicates upload.
dscp-mark	Specifies the decimal value of a DSCP mark.

4. Configure a port or network to apply the policy to. You can update an existing port or network, or create a new port or network to apply the policy to:

   - Apply the policy to an existing port:

     ```
     (overcloud) $ openstack port set --qos-policy <policy_name> <port_name|port_ID>
     ```

   - Create a new port:

     ```
     (overcloud) $ openstack port create --qos-policy <policy_name> --network
     <network_name|network_ID> <port_name|port_ID>
     ```

   - Apply the policy to an existing network:

     ```
     (overcloud) $ openstack network set --qos-policy <policy_name>
     <network_name|network_ID>
     ```

   - Create a new network:

     ```
     (overcloud) $ openstack network create --qos-policy <policy_name> <network_name>
     ```

10.2.1. Creating and applying a bandwidth limit QoS policy and rule

You can create a QoS policy that limits the bandwidth on networks, ports or floating IPs, and drops any traffic that exceeds the specified rate. To create and apply a bandwidth limit QoS policy and rule, complete the following steps:

Procedure

1. If the **qos** extension is not already enabled for OpenStack Networking in */etc/neutron/plugins/ml2/<agent_name>_agent.ini*, then perform the following steps:

   a. Create a custom environment file, and add the following configuration:

   ```
   parameter_defaults:
     NeutronSriovAgentExtensions: 'qos'
   ```

   b. To apply this configuration, deploy the overcloud, adding your custom environment file to the stack along with your other environment files:

```
(undercloud) $ openstack overcloud deploy --templates \
  -e [your environment files]
  -e /home/stack/templates/<custom-environment-file>.yaml
```

For more information, see "Modifying the Overcloud Environment" in the *Director Installation and Usage* guide.

2. Identify the ID of the project you want to create the QoS policy for:

```
(overcloud) $ openstack project list
+----------------------------------+----------+
| ID                               | Name     |
+----------------------------------+----------+
| 4b0b98f8c6c040f38ba4f7146e8680f5 | auditors |
| 519e6344f82e4c079c8e2eabb690023b | services |
| 80bf5732752a41128e612fe615c886c6 | demo     |
| 98a2f53c20ce4d50a40dac4a38016c69 | admin    |
+----------------------------------+----------+
```

3. Create a QoS policy named "bw-limiter" in the **admin** project:

```
(overcloud) $ openstack network qos policy create --share --project
98a2f53c20ce4d50a40dac4a38016c69 bw-limiter
```

4. Configure the rules for the "bw-limiter" policy:

```
(overcloud) $ openstack network qos rule create --type bandwidth-limit --max_kbps 3000 --
max_burst_kbps 300 bw-limiter
```

5. Configure a port to apply the "bw-limiter" policy to:

```
(overcloud) $ openstack port set --qos-policy bw-limiter <port_name|port_ID>
```

10.2.2. Creating and applying a guaranteed minimum bandwidth QoS policy and rule

You can request bandwidth guarantees for ports on networks with **segmentation_type=flat** or **segmentation_type=vlan**, which are backed by a physical network (physnet).

NOTE

- Do not mix ports with and without bandwidth guarantees on the same physical interface, as the ports without a guarantee may starve. Create host aggregates to separate ports with bandwidth guarantees from those without.
- You can only modify a guaranteed minimum bandwidth QoS policy when there are no instances using any of the ports the policy is assigned to.

Supported drivers and agents¶

- SR-IOV (sriovnicswitch) vnic_types: direct, macvtap
- ML2/OVS (openvswitch) vnic_types: normal, direct

NOTE

ML2/OVN does not support minimum bandwidth.

Prerequisites

- The Placement service must support microversion 1.29.
- The Compute (nova) service must support microversion 2.72.
- The Networking (neutron) service must support the following API extensions:
  - **agent-resources-synced**
  - **port-resource-request**
  - **qos-bw-minimum-ingress**
- To query placement information using the OpenStack CLI, install the Placement service package, **python3-osc-placement**, on the undercloud.

Procedure

1. If the Placement service plugin is not already configured for Openstack networking, then perform the following steps:

   a. If the **NeutronServicePlugins** is already provided in a custom environment file, update the parameter to include "placement", otherwise create a custom environment file and add the following configuration:

   ```
   parameter_defaults:
    NeutronServicePlugins: 'router,qos,segments,trunk,placement'
   ```

   b. To apply this configuration, deploy the overcloud, adding your custom environment file to the stack along with your other environment files:

   ```
   (undercloud) $ openstack overcloud deploy --templates \
     -e [your environment files]
     -e /home/stack/templates/network-environment.yaml
   ```

   For more information, see "Modifying the Overcloud Environment" in the *Director Installation and Usage* guide.

2. (Optional) To black list **vnic_types** when multiple ML2 mechanism drivers support them by default and multiple agents are being tracked in Placement, add **vnic_type_blacklist** to **/etc/neutron/plugins/ml2/ml2_conf.ini** and restart the agent:

   ```
   [ovs_driver]
   vnic_type_blacklist = direct
   [sriov_driver]
   #vnic_type_blacklist = direct
   ```

3. Configure the resource provider ingress and egress bandwidths for the relevant agents on each Compute node that needs to provide a minimum bandwidth. You can configure only ingress or egress, or both, using the following formats:

- Configure only egress bandwidth, in kbps:

  ```
  resource_provider_bandwidths = <bridge0>:<egress_kbps>:,<bridge1>:
  <egress_kbps>:,...,<bridgeN>:<egress_kbps>:
  ```

- Configure only ingress bandwidth, in kbps:

  ```
  resource_provider_bandwidths = <bridge0>::<ingress_kbps>,<bridge1>::
  <ingress_kbps>,...,<bridgeN>::<ingress_kbps>
  ```

- Configure both egress and ingress bandwidth, in kbps:

  ```
  resource_provider_bandwidths = <bridge0>:<egress_kbps>:<ingress_kbps>,<bridge1>:
  <egress_kbps>:<ingress_kbps>,...,<bridgeN>:<egress_kbps>:<ingress_kbps>
  ```

  For example:

  - To configure the resource provider ingress and egress bandwidths for the OVS agent, add **resource_provider_bandwidths** to **/etc/neutron/plugins/ml2/openvswitch_agent.ini**:

    ```
    [ovs]
    bridge_mappings = physnet0:br-physnet0
    resource_provider_bandwidths = br-physnet0:10000000:10000000
    ```

  - To configure the resource provider ingress and egress bandwidths for the SRIOV agent, add **resource_provider_bandwidths** to **/etc/neutron/plugins/ml2/sriov_agent.ini**:

    ```
    [sriov_nic]
    physical_device_mappings = physnet0:ens5,physnet0:ens6
    resource_provider_bandwidths =
    ens5:40000000:40000000,ens6:40000000:40000000
    ```

To implement the resource provider bandwidths, restart the configured agents.

4. Identify the ID of the project you want to create the QoS policy for:

   ```
   (overcloud) $ openstack project list
   +----------------------------------+----------+
   | ID                               | Name     |
   +----------------------------------+----------+
   | 4b0b98f8c6c040f38ba4f7146e8680f5 | auditors |
   | 519e6344f82e4c079c8e2eabb690023b | services |
   | 80bf5732752a41128e612fe615c886c6 | demo     |
   | 98a2f53c20ce4d50a40dac4a38016c69 | admin    |
   +----------------------------------+----------+
   ```

5. Create a QoS policy named "guaranteed_min_bw" in the **admin** project:

   ```
   (overcloud) $ openstack network qos policy create --share --project
   98a2f53c20ce4d50a40dac4a38016c69 guaranteed_min_bw
   ```

6. Configure the rules for the "guaranteed_min_bw" policy:

```
(overcloud) $ openstack network qos rule create --type minimum-bandwidth --min-kbps
40000000 --ingress guaranteed_min_bw
(overcloud) $ openstack network qos rule create --type minimum-bandwidth --min-kbps
40000000 --egress guaranteed_min_bw
```

7. Configure a port to apply the "guaranteed_min_bw" policy to:

```
(overcloud) $ openstack port set --qos-policy guaranteed_min_bw <port_name|port_ID>
```

Verification

1. List all of the available resource providers:

```
(undercloud) $ openstack --os-placement-api-version 1.17 resource provider list
```

Example output:

```
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------
| uuid                                 | name                                          | generation |
root_provider_uuid                   | parent_provider_uuid                 |
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------
| 31d3d88b-bc3a-41cd-9dc0-fda54028a882 | dell-r730-014.localdomain                     |
28 | 31d3d88b-bc3a-41cd-9dc0-fda54028a882 | None                                 |
| 6b15ddce-13cf-4c85-a58f-baec5b57ab52 | dell-r730-063.localdomain                     |
18 | 6b15ddce-13cf-4c85-a58f-baec5b57ab52 | None                                 |
| e2f5082a-c965-55db-acb3-8daf9857c721 | dell-r730-063.localdomain:NIC Switch agent
|          0 | 6b15ddce-13cf-4c85-a58f-baec5b57ab52 | 6b15ddce-13cf-4c85-a58f-
baec5b57ab52 |
| d2fb0ef4-2f45-53a8-88be-113b3e64ba1b | dell-r730-014.localdomain:NIC Switch agent
|          0 | 31d3d88b-bc3a-41cd-9dc0-fda54028a882 | 31d3d88b-bc3a-41cd-9dc0-
fda54028a882 |
| f1ca35e2-47ad-53a0-9058-390ade93b73e | dell-r730-063.localdomain:NIC Switch
agent:enp6s0f1 |         13 | 6b15ddce-13cf-4c85-a58f-baec5b57ab52 | e2f5082a-c965-55db-
acb3-8daf9857c721 |
| e518d381-d590-5767-8f34-c20def34b252 | dell-r730-014.localdomain:NIC Switch
agent:enp6s0f1 |         19 | 31d3d88b-bc3a-41cd-9dc0-fda54028a882 | d2fb0ef4-2f45-53a8-
88be-113b3e64ba1b |
---------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------
```

2. Check the bandwidth a specific resource provides:

```
(undercloud) $ openstack --os-placement-api-version 1.17 resource provider inventory list
<rp_uuid>
```

The following example output shows the bandwidth provided by interface enp6s0f1 on dell-r730-014:

```
[stack@dell-r730-014 nova]$ openstack --os-placement-api-version 1.17 resource provider
inventory list e518d381-d590-5767-8f34-c20def34b252
------------------------------------------------------------------------------------------------
```

```
| resource_class            | allocation_ratio | min_unit |   max_unit | reserved | step_size |
total |
----------------------------------------------------------------------------------------------------
| NET_BW_EGR_KILOBIT_PER_SEC |            1.0 |        1 | 2147483647 |        0 |         1 |
10000000 |
| NET_BW_IGR_KILOBIT_PER_SEC |            1.0 |        1 | 2147483647 |        0 |         1 |
10000000 |
----------------------------------------------------------------------------------------------------
```

3. To check claims against the resource provider when instances are running, run the following command:

```
(undercloud) $ openstack --os-placement-api-version 1.17 resource provider show --
allocations  <rp_uuid>
```

Example output:

```
[stack@dell-r730-014 nova]$ openstack --os-placement-api-version 1.17 resource provider
show --allocations  e518d381-d590-5767-8f34-c20def34b252 -f value -c allocations
{3cbb9e07-90a8-4154-8acd-b6ec2f894a83: {resources:
{NET_BW_EGR_KILOBIT_PER_SEC: 1000000, NET_BW_IGR_KILOBIT_PER_SEC:
1000000}}, 8848b88b-4464-443f-bf33-5d4e49fd6204: {resources:
{NET_BW_EGR_KILOBIT_PER_SEC: 1000000, NET_BW_IGR_KILOBIT_PER_SEC:
1000000}}, 9a29e946-698b-4731-bc28-89368073be1a: {resources:
{NET_BW_EGR_KILOBIT_PER_SEC: 1000000, NET_BW_IGR_KILOBIT_PER_SEC:
1000000}}, a6c83b86-9139-4e98-9341-dc76065136cc: {resources:
{NET_BW_EGR_KILOBIT_PER_SEC: 3000000, NET_BW_IGR_KILOBIT_PER_SEC:
3000000}}, da60e33f-156e-47be-a632-870172ec5483: {resources:
{NET_BW_EGR_KILOBIT_PER_SEC: 1000000, NET_BW_IGR_KILOBIT_PER_SEC:
1000000}}, eb582a0e-8274-4f21-9890-9a0d55114663: {resources:
{NET_BW_EGR_KILOBIT_PER_SEC: 3000000, NET_BW_IGR_KILOBIT_PER_SEC:
3000000}}}
```

10.2.3. DSCP marking for egress traffic

You can use differentiated services code point (DSCP) to implement quality-of-service (QoS) policies on your network by embedding relevant values in the IP headers. OpenStack Networking (neutron) QoS policies can use DSCP marking to manage egress traffic on neutron ports and networks. At present, DSCP is available only for VLAN and flat provider networks using Open vSwitch (OVS).

Complete the steps in this example procedure to create a policy, define DSCP rules, and apply the rules to the policy. These rules use the **--dscp-mark** parameter, which specifies the decimal value of a DSCP mark.

1. Create a new QoS policy:

```
openstack network qos policy create --project 98a2f53c20ce4d50a40dac4a38016c69
qos_policy_name
```

2. Create a DSCP rule and apply it to the **qos-web-servers** policy, using DSCP mark **18**:

```
openstack network qos rule create --type dscp-marking --dscp-mark 18 qos_policy_name
Created a new dscp_marking_rule:
+-----------+--------------------------------------+
| Field     | Value                                |
+-----------+--------------------------------------+
| dscp_mark | 18                                   |
| id        | d7f976ec-7fab-4e60-af70-f59bf88198e6 |
+-----------+--------------------------------------+
```

3. View the DSCP rules for QoS policy, **qos-web-servers**:

```
openstack network qos rule list qos-web-servers
+-----------+--------------------------------------+
| dscp_mark | id                                   |
+-----------+--------------------------------------+
|        18 | d7f976ec-7fab-4e60-af70-f59bf88198e6 |
+-----------+--------------------------------------+
```

4. View the details of the DSCP rule assigned to policy, **qos-web-servers**:

```
openstack network qos rule show qos-web-servers d7f976ec-7fab-4e60-af70-f59bf88198e6
+-----------+--------------------------------------+
| Field     | Value                                |
+-----------+--------------------------------------+
| dscp_mark | 18                                   |
| id        | d7f976ec-7fab-4e60-af70-f59bf88198e6 |
+-----------+--------------------------------------+
```

5. Change the DSCP value assigned to a rule:

```
openstack network qos rule set --dscp-mark 22 qos-web-servers d7f976ec-7fab-4e60-af70-
f59bf88198e6
```

6. Delete a DSCP rule:

```
openstack network qos rule delete qos-web-servers d7f976ec-7fab-4e60-af70-f59bf88198e6
```

10.2.4. How to verify the application of your QoS policy and rules

Use the following commands to verify the creation and application of your quality of service (QoS) policy and rules.

Action	Command
List the available QoS policies	**$ openstack network qos policy list**
Show details of a specific QoS policy	**$ openstack network qos policy show <policy_name>**
List the available QoS rules	**$ openstack network qos rule type list**
List the rules of a specific QoS policy	**$ openstack network qos rule list <policy_name>**

Action	Command
Show details of a specific rule	**$ openstack network qos rule type show <rule_id>**
List the available ports	**$ openstack port list**
Show details of a specific port	**$ openstack port show <port_ID/port_name>**

10.3. RBAC FOR QOS POLICIES

You can add a role-based access control (RBAC) for quality-of-service (QoS) policies. As a result, you can now make QoS policies available to certain projects.

For example, you can now create a QoS policy that allows for lower-priority network traffic, and have it only apply to certain projects. Run the following command to assign the **bw-limiter** policy to the tenant, **demo**:

```
# openstack network rbac create --type qos_policy --target-project
80bf5732752a41128e612fe615c886c6 --action access_as_shared bw-limiter
```

CHAPTER 11. CONFIGURING BRIDGE MAPPINGS

This chapter contains information about configuring bridge mappings in Red Hat OpenStack Platform.

11.1. OVERVIEW OF BRIDGE MAPPINGS

A bridge mapping associates a physical network name (an interface label) to a bridge created with OVS or OVN. In this example, the physical name (datacentre) is mapped to the external bridge (br-ex):

```
bridge_mappings = datacentre:br-ex
```

Bridge mappings allow provider network traffic to reach the physical network. Traffic leaves the provider network from the **qg-xxx** interface of the router and arrives at **br-int**. For OVS, a patch port between **br-int** and **br-ex** then allows the traffic to pass through the bridge of the provider network and out to the physical network. OVN creates a patch port on a hypervisor only when there is a VM bound to the hypervisor that requires the port.

You configure bridge mappings on the network node on which the router is scheduled. Router traffic can egress using the correct physical network, as represented by the provider network.

11.2. TRAFFIC FLOW

Each external network is represented by an internal VLAN ID, which is tagged to the router **qg-xxx** port. When a packet reaches **phy-br-ex**, the **br-ex** port strips the VLAN tag and moves the packet to the physical interface and then to the external network.

The return packet from the external network arrives on **br-ex** and moves to **br-int** using **phy-br-ex <-> int-br-ex**. When the packet is going through **br-ex** to **br-int**, the packet's external vlan ID is replaced by an internal vlan tag in **br-int**, and this allows **qg-xxx** to accept the packet.

In the case of egress packets, the packet's internal vlan tag is replaced with an external vlan tag in br-ex (or in the external bridge that is defined in the **network_vlan_ranges** parameter).

11.3. CONFIGURING BRIDGE MAPPINGS

Red Hat OpenStack Platform (RHOSP) director uses predefined NIC templates to install and configure your initial networking configuration.

You can customize aspects of your initial networking configuration, such as bridge mappings, by using the **NeutronBridgeMappings** parameter in a customized environment file. You call the environment file in the **openstack overcloud deploy** command.

Prerequisites

- You must configure bridge mappings on the network node on which the router is scheduled.
- For both ML2/OVS and ML2/OVN DVR configurations, you must configure bridge mappings for the compute nodes, too.

Procedure

1. Create a custom environment file and add the **NeutronBridgeMappings** heat parameter with values that are appropriate for your site.

    ■

```
parameter_defaults:
  NeutronBridgeMappings: "datacentre:br-ex,tenant:br-tenant"
```

The **NeutronBridgeMappings** heat parameter associates a physical name (**datacentre**) to a bridge (**br-ex**).

NOTE

When the **NeutronBridgeMappings** parameter is not used, the default maps the external bridge on hosts (br-ex) to a physical name (datacentre).

2. To apply this configuration, deploy the overcloud, adding your custom environment file to the stack along with your other environment files.

```
(undercloud) $ openstack overcloud deploy --templates \
  -e [your environment files]
  -e /home/stack/templates/<custom-environment-file>.yaml
```

3. You are ready for the next steps, which are the following:

   a. Using the network VLAN ranges, create the provider networks that represent the corresponding external networks. (You use the physical name when creating neutron provider networks or floating IP networks.)

   b. Connect the external networks to your tenant networks with router interfaces.

Additional resources

- Network environment parameters in the *Advanced Overcloud Customization* guide
- Including Environment Files in Overcloud Creation in the *Advanced Overcloud Customization* guide

11.4. MAINTAINING BRIDGE MAPPINGS FOR OVS

After removing any OVS bridge mappings, you must perform a subsequent cleanup to ensure that the bridge configuration is cleared of any associated patch port entries. You can perform this operation in the following ways:

- Manual port cleanup - requires careful removal of the superfluous patch ports. No outages of network connectivity are required.
- Automated port cleanup - performs an automated cleanup, but requires an outage, and requires that the necessary bridge mappings be re-added. Choose this option during scheduled maintenance windows when network connectivity outages can be tolerated.

NOTE

When OVN bridge mappings are removed, the OVN controller automatically cleans up any associated patch ports.

11.4.1. Cleaning up OVS patch ports manually

After removing any OVS bridge mappings, you must also remove the associated patch ports.

Prerequisites

- The patch ports that you are cleaning up must be Open Virtual Switch (OVS) ports.
- A system outage is **not** required to perform a manual patch port cleanup.
- You can identify the patch ports to cleanup by their naming convention:
  - In **br-$external_bridge** patch ports are named **phy-<external bridge name>** (for example, phy-br-ex2).
  - In **br-int** patch ports are named **int-<external bridge name>** (for example, **int-br-ex2**).

Procedure

1. Use **ovs-vsctl** to remove the OVS patch ports associated with the removed bridge mapping entry:

```
# ovs-vsctl del-port br-ex2 datacentre
# ovs-vsctl del-port br-tenant tenant
```

2. Restart **neutron-openvswitch-agent**:

```
# service neutron-openvswitch-agent restart
```

11.4.2. Cleaning up OVS patch ports automatically

After removing any OVS bridge mappings, you must also remove the associated patch ports.

NOTE

When OVN bridge mappings are removed, the OVN controller automatically cleans up any associated patch ports.

Prerequisites

- The patch ports that you are cleaning up must be Open Virtual Switch (OVS) ports.
- Cleaning up patch ports automatically with the **neutron-ovs-cleanup** command causes a network connectivity outage, and should be performed only during a scheduled maintenance window.
- Use the flag **--ovs_all_ports** to remove all patch ports from **br-int**, cleaning up tunnel ends from **br-tun**, and patch ports from bridge to bridge.
- The **neutron-ovs-cleanup** command unplugs all patch ports (instances, qdhcp/qrouter, among others) from all OVS bridges.

Procedure

1. Run the **neutron-ovs-cleanup** command with the **--ovs_all_ports** flag.

IMPORTANT

Perfoming this step will result in a total networking outage.

```
# /usr/bin/neutron-ovs-cleanup
--config-file /etc/neutron/plugins/ml2/openvswitch_agent.ini
--log-file /var/log/neutron/ovs-cleanup.log --ovs_all_ports
```

2. Restore connectivity by redeploying the overcloud.
   When you rerun the **openstack overcloud deploy** command, your bridge mapping values are reapplied.

NOTE

After a restart, the OVS agent does not interfere with any connections that are not present in bridge_mappings. So, if you have **br-int** connected to **br-ex2**, and **br-ex2** has some flows on it, removing **br-int** from the bridge_mappings configuration does not disconnect the two bridges when you restart the OVS agent or the node.

Additional resources

- Network environment parameters in the *Advanced Overcloud Customization* guide
- Including Environment Files in Overcloud Creation in the *Advanced Overcloud Customization* guide

CHAPTER 12. VLAN-AWARE INSTANCES

12.1. OVERVIEW OF VLAN-AWARE INSTANCES

Instances can send and receive VLAN-tagged traffic over a single vNIC. This is particularly useful for NFV applications (VNFs) that expect VLAN-tagged traffic, allowing a single vNIC to serve multiple customers or services.

For example, the tenant data network can use VLANs, or tunneling (VXLAN/GRE) segmentation, while the instances see the traffic tagged with VLAN IDs. As a result, network packets are tagged just before they are injected to the instance and do not need to be tagged throughout the entire network.

To implement VLAN-tagged traffic, create a parent port and attach the new port to an existing neutron network. When you attach the new port, OpenStack Networking adds a trunk connection to the parent port you created. Next, create subports. These subports connect VLANs to instances, which allow connectivity to the trunk. Within the instance operating system, you must also create a sub-interface that tags traffic for the VLAN associated with the subport.

12.2. REVIEWING THE TRUNK PLUG-IN

During a Red Hat openStack deployment, the trunk plug-in is enabled by default. You can review the configuration on the controller nodes:

- On the controller node, confirm that the trunk plug-in is enabled in the /var/lib/config-data/neutron/etc/neutron/neutron.conf file:

```
service_plugins=router,qos,trunk
```

12.3. CREATING A TRUNK CONNECTION

1. Identify the network that requires the trunk port connection. This would be the network that will contain the instance that requires access to the trunked VLANs. In this example, this is the *public* network:

```
openstack network list
+--------------------------------------+---------+--------------------------------------+
| ID                                   | Name    | Subnets                              |
+--------------------------------------+---------+--------------------------------------+
| 82845092-4701-4004-add7-838837837621 | private | 434c7982-cd96-4c41-a8c9-
b93adbdcb197 |
| 8d8bc6d6-5b28-4e00-b99e-157516ff0050 | public  | 3fd811b4-c104-44b5-8ff8-
7a86af5e332c |
+--------------------------------------+---------+--------------------------------------+
```

2. Create the parent trunk port, and attach it to the network that the instance connects to. In this example, create a neutron port named parent-trunk-port on the *public* network. This trunk is the *parent* port, as you can use it to create *subports*.

```
openstack port create --network public parent-trunk-port
+-----------------------+-----------------------------------------------------------------------------+
| Field                 | Value                                                                       |
+-----------------------+-----------------------------------------------------------------------------+
| admin_state_up        | UP                                                                          |
| allowed_address_pairs |                                                                      |
| binding_host_id       |                                                                      |
| binding_profile       |                                                                      |
| binding_vif_details   |                                                                      |
| binding_vif_type      | unbound                                                              |
| binding_vnic_type     | normal                                                               |
| created_at            | 2016-10-20T02:02:33Z                                                 |
| description           |                                                                      |
| device_id             |                                                                      |
| device_owner          |                                                                      |
| extra_dhcp_opts       |                                                                      |
| fixed_ips             | ip_address='172.24.4.230', subnet_id='dc608964-9af3-4fed-9f06-
6d3844fb9b9b' |
| headers               |                                                                      |
| id                    | 20b6fdf8-0d43-475a-a0f1-ec8f757a4a39                                 |
| mac_address           | fa:16:3e:33:c4:75                                                    |
| name                  | parent-trunk-port                                                    |
| network_id            | 871a6bd8-4193-45d7-a300-dcb2420e7cc3                                 |
| project_id            | 745d33000ac74d30a77539f8920555e7                                     |
| project_id            | 745d33000ac74d30a77539f8920555e7                                     |
| revision_number       | 4                                                                    |
| security_groups       | 59e2af18-93c6-4201-861b-19a8a8b79b23                                 |
| status                | DOWN                                                                 |
| updated_at            | 2016-10-20T02:02:33Z                                                 |
+-----------------------+----------------------------------------------------------------------+
```

3. Create a trunk using the port that you created in step 2. In this example the trunk is named **parent-trunk**.

```
openstack network trunk create --parent-port parent-trunk-port parent-trunk
+-----------------+--------------------------------------+
| Field           | Value                                |
+-----------------+--------------------------------------+
| admin_state_up  | UP                                   |
| created_at      | 2016-10-20T02:05:17Z                 |
| description     |                                      |
| id              | 0e4263e2-5761-4cf6-ab6d-b22884a0fa88 |
| name            | parent-trunk                         |
| port_id         | 20b6fdf8-0d43-475a-a0f1-ec8f757a4a39 |
| revision_number | 1                                    |
| status          | DOWN                                 |
| sub_ports       |                                      |
| tenant_id       | 745d33000ac74d30a77539f8920555e7     |
| updated_at      | 2016-10-20T02:05:17Z                 |
+-----------------+--------------------------------------+
```

4. View the trunk connection:

```
openstack network trunk list
+--------------------------------------+--------------+--------------------------------------+-------------+
| ID                                   | Name         | Parent Port                          | Description |
+--------------------------------------+--------------+--------------------------------------+-------------+
| 0e4263e2-5761-4cf6-ab6d-b22884a0fa88 | parent-trunk | 20b6fdf8-0d43-475a-a0f1-
ec8f757a4a39 |             |
+--------------------------------------+--------------+--------------------------------------+-------------+
```

5. View the details of the trunk connection:

```
openstack network trunk show parent-trunk
+-----------------+--------------------------------------+
| Field           | Value                                |
+-----------------+--------------------------------------+
| admin_state_up  | UP                                   |
| created_at      | 2016-10-20T02:05:17Z                 |
| description     |                                      |
| id              | 0e4263e2-5761-4cf6-ab6d-b22884a0fa88 |
| name            | parent-trunk                         |
| port_id         | 20b6fdf8-0d43-475a-a0f1-ec8f757a4a39 |
| revision_number | 1                                    |
| status          | DOWN                                 |
| sub_ports       |                                      |
| tenant_id       | 745d33000ac74d30a77539f8920555e7     |
| updated_at      | 2016-10-20T02:05:17Z                 |
+-----------------+--------------------------------------+
```

12.4. ADDING SUBPORTS TO THE TRUNK

1. Create a neutron port.
   This port is a subport connection to the trunk. You must also specify the MAC address that you assigned to the parent port:

```
openstack port create --network private --mac-address fa:16:3e:33:c4:75 subport-trunk-port
+-----------------------+-----------------------------------------------------------------------------+
| Field                 | Value                                                                       |
+-----------------------+-----------------------------------------------------------------------------+
| admin_state_up        | UP                                                                          |
| allowed_address_pairs |                                                                             |
| binding_host_id       |                                                                             |
| binding_profile       |                                                                             |
| binding_vif_details   |                                                                             |
| binding_vif_type      | unbound                                                                     |
| binding_vnic_type     | normal                                                                      |
| created_at            | 2016-10-20T02:08:14Z                                                        |
| description           |                                                                             |
| device_id             |                                                                             |
| device_owner          |                                                                             |
| extra_dhcp_opts       |                                                                             |
| fixed_ips             | ip_address='10.0.0.11', subnet_id='1a299780-56df-4c0b-a4c0-
c5a612cef2e8' |
| headers               |                                                                             |
| id                    | 479d742e-dd00-4c24-8dd6-b7297fab3ee9                                        |
| mac_address           | fa:16:3e:33:c4:75                                                           |
| name                  | subport-trunk-port                                                          |
| network_id            | 3fe6b758-8613-4b17-901e-9ba30a7c4b51                                        |
| project_id            | 745d33000ac74d30a77539f8920555e7                                            |
| project_id            | 745d33000ac74d30a77539f8920555e7                                            |
| revision_number       | 4                                                                           |
| security_groups       | 59e2af18-93c6-4201-861b-19a8a8b79b23                                        |
| status                | DOWN                                                                        |
| updated_at            | 2016-10-20T02:08:15Z                                                        |
+-----------------------+-----------------------------------------------------------------------------+
```

NOTE

If you receive the error **HttpException: Conflict**, confirm that you are creating the subport on a different network to the one that has the parent trunk port. This example uses the public network for the parent trunk port, and private for the subport.

2. Associate the port with the trunk (**parent-trunk**), and specify the VLAN ID (**55**):

```
openstack network trunk set --subport port=subport-trunk-port,segmentation-
type=vlan,segmentation-id=55 parent-trunk
```

12.5. CONFIGURING AN INSTANCE TO USE A TRUNK

You must configure the instance operating system to use the MAC address that neutron assigned to the subport. You can also configure the subport to use a specific MAC address during the subport creation step.

1. Review the configuration of your network trunk:

```
$ openstack network trunk list
+--------------------------------------+--------------+--------------------------------------+-------------+
| ID                                   | Name         | Parent Port                          | Description |
+--------------------------------------+--------------+--------------------------------------+-------------+
| 0e4263e2-5761-4cf6-ab6d-b22884a0fa88 | parent-trunk | 20b6fdf8-0d43-475a-a0f1-
ec8f757a4a39 |             |
+--------------------------------------+--------------+--------------------------------------+-------------+


$ openstack network trunk show parent-trunk
+-----------------+-----------------------------------------------------------------------------------------------
---+
| Field           | Value                                                                                        |
+-----------------+-----------------------------------------------------------------------------------------------
---+
| admin_state_up  | UP                                                                                           |
| created_at      | 2016-10-20T02:05:17Z                                                                         |
| description     |                                                                                              |
| id              | 0e4263e2-5761-4cf6-ab6d-b22884a0fa88                                                         |
| name            | parent-trunk                                                                                 |
| port_id         | 20b6fdf8-0d43-475a-a0f1-ec8f757a4a39                                                         |
| revision_number | 2                                                                                            |
| status          | DOWN                                                                                         |
| sub_ports       | port_id='479d742e-dd00-4c24-8dd6-b7297fab3ee9', segmentation_id='55',
segmentation_type='vlan' |
| tenant_id       | 745d33000ac74d30a77539f8920555e7
|
| updated_at      | 2016-10-20T02:10:06Z                                                                         |
+-----------------+-----------------------------------------------------------------------------------------------
---+
```

2. Create an instance that uses the parent **port-id** as its vNIC:

```
nova boot --image cirros --flavor m1.tiny testInstance --security-groups default --key-name
sshaccess --nic port-id=20b6fdf8-0d43-475a-a0f1-ec8f757a4a39


+--------------------------------------+-----------------------------------------------+
| Property                             | Value                                         |
+--------------------------------------+-----------------------------------------------+
| OS-DCF:diskConfig                    | MANUAL                                        |
| OS-EXT-AZ:availability_zone          |                                               |
| OS-EXT-SRV-ATTR:host                 | -                                             |
| OS-EXT-SRV-ATTR:hostname             | testinstance                                  |
| OS-EXT-SRV-ATTR:hypervisor_hostname  | -                                             |
| OS-EXT-SRV-ATTR:instance_name        |                                               |
| OS-EXT-SRV-ATTR:kernel_id            |                                               |
| OS-EXT-SRV-ATTR:launch_index         | 0                                             |
| OS-EXT-SRV-ATTR:ramdisk_id           |                                               |
| OS-EXT-SRV-ATTR:reservation_id       | r-juqco0el                                    |
| OS-EXT-SRV-ATTR:root_device_name     | -                                             |
| OS-EXT-SRV-ATTR:user_data            | -                                             |
| OS-EXT-STS:power_state               | 0                                             |
| OS-EXT-STS:task_state                | scheduling                                    |
| OS-EXT-STS:vm_state                  | building                                      |
| OS-SRV-USG:launched_at               | -                                             |
| OS-SRV-USG:terminated_at             | -                                             |
| accessIPv4                           |                                               |
| accessIPv6                           |                                               |
| adminPass                            | uMyL8PnZRBwQ                                  |
| config_drive                         |                                               |
| created                              | 2016-10-20T03:02:51Z                          |
| description                          | -                                             |
| flavor                               | m1.tiny (1)                                   |
| hostId                               |                                               |
| host_status                          |                                               |
| id                                   | 88b7aede-1305-4d91-a180-67e7eac8b70d          |
| image                                | cirros (568372f7-15df-4e61-a05f-10954f79a3c4) |
| key_name                             | sshaccess                                     |
| locked                               | False                                         |
| metadata                             | {}                                            |
| name                                 | testInstance                                  |
| os-extended-volumes:volumes_attached | []                                            |
| progress                             | 0                                             |
| security_groups                      | default                                       |
| status                               | BUILD                                         |
| tags                                 | []                                            |
| tenant_id                            | 745d33000ac74d30a77539f8920555e7              |
| updated                              | 2016-10-20T03:02:51Z                          |
| user_id                              | 8c4aea738d774967b4ef388eb41fef5e              |
+--------------------------------------+-----------------------------------------------+
```

12.6. UNDERSTANDING TRUNK STATES

- **ACTIVE**: The trunk is working as expected and there are no current requests.
- **DOWN**: The virtual and physical resources for the trunk are not in sync. This can be a temporary state during negotiation.

- **BUILD**: There has been a request and the resources are being provisioned. After successful completion the trunk returns to **ACTIVE**.
- **DEGRADED**: The provisioning request did not complete, so the trunk has only been partially provisioned. It is recommended to remove the subports and try again.
- **ERROR**: The provisioning request was unsuccessful. Remove the resource that caused the error to return the trunk to a healthier state. Do not add more subports while in the **ERROR** state, as this can cause more issues.

CHAPTER 13. CONFIGURING RBAC POLICIES

13.1. OVERVIEW OF RBAC POLICIES

Role-based access control (RBAC) policies in OpenStack Networking allow granular control over shared *neutron* networks. OpenStack Networking uses a RBAC table to control sharing of *neutron* networks among tenants, allowing an administrator to control which tenants are granted permission to attach instances to a network.

As a result, cloud administrators can remove the ability for some tenants to create networks and can instead allow them to attach to pre-existing networks that correspond to their project.

13.2. CREATING RBAC POLICIES

This example procedure demonstrates how to use a role-based access control (RBAC) policy to grant a project access to a shared network.

1. View the list of available networks:

```
# openstack network list
+--------------------------------------+-------------+-------------------------------------------------------+
| id                                   | name        | subnets                                               |
+--------------------------------------+-------------+-------------------------------------------------------+
| fa9bb72f-b81a-4572-9c7f-7237e5fcabd3 | web-servers | 20512ffe-ad56-4bb4-b064-
2cb18fecc923 192.168.200.0/24 |
| bcc16b34-e33e-445b-9fde-dd491817a48a | private     | 7fe4a05a-4b81-4a59-8c47-
82c965b0e050 10.0.0.0/24      |
| 9b2f4feb-fee8-43da-bb99-032e4aaf3f85 | public      | 2318dc3b-cff0-43fc-9489-
7d4cf48aaab9 172.24.4.224/28  |
+--------------------------------------+-------------+-------------------------------------------------------+
```

2. View the list of projects:

```
# openstack project list
+----------------------------------+----------+
| ID                               | Name     |
+----------------------------------+----------+
| 4b0b98f8c6c040f38ba4f7146e8680f5 | auditors |
| 519e6344f82e4c079c8e2eabb690023b | services |
| 80bf5732752a41128e612fe615c886c6 | demo     |
| 98a2f53c20ce4d50a40dac4a38016c69 | admin    |
+----------------------------------+----------+
```

3. Create a RBAC entry for the **web-servers** network that grants access to the *auditors* project (**4b0b98f8c6c040f38ba4f7146e8680f5**):

```
# openstack network rbac create --type network --target-project
4b0b98f8c6c040f38ba4f7146e8680f5 --action access_as_shared web-servers
Created a new rbac_policy:
+----------------+--------------------------------------+
| Field          | Value                                |
+----------------+--------------------------------------+
| action         | access_as_shared                     |
| id             | 314004d0-2261-4d5e-bda7-0181fcf40709 |
| object_id      | fa9bb72f-b81a-4572-9c7f-7237e5fcabd3 |
| object_type    | network                              |
| target_project | 4b0b98f8c6c040f38ba4f7146e8680f5     |
| project_id     | 98a2f53c20ce4d50a40dac4a38016c69     |
+----------------+--------------------------------------+
```

As a result, users in the *auditors* project can connect instances to the **web-servers** network.

13.3. REVIEWING RBAC POLICIES

1. Run the **openstack network rbac list** command to retrieve the ID of your existing role-based access control (RBAC) policies:

```
# openstack network rbac list
+--------------------------------------+-------------+--------------------------------------+
| id                                   | object_type | object_id                            |
+--------------------------------------+-------------+--------------------------------------+
| 314004d0-2261-4d5e-bda7-0181fcf40709 | network     | fa9bb72f-b81a-4572-9c7f-
7237e5fcabd3 |
| bbab1cf9-edc5-47f9-aee3-a413bd582c0a | network     | 9b2f4feb-fee8-43da-bb99-
032e4aaf3f85 |
+--------------------------------------+-------------+--------------------------------------+
```

2. Run the **openstack network rbac-show** command to view the details of a specific RBAC entry:

```
# openstack network rbac show 314004d0-2261-4d5e-bda7-0181fcf40709
+----------------+--------------------------------------+
| Field          | Value                                |
+----------------+--------------------------------------+
| action         | access_as_shared                     |
| id             | 314004d0-2261-4d5e-bda7-0181fcf40709 |
| object_id      | fa9bb72f-b81a-4572-9c7f-7237e5fcabd3 |
| object_type    | network                              |
| target_project | 4b0b98f8c6c040f38ba4f7146e8680f5     |
| project_id     | 98a2f53c20ce4d50a40dac4a38016c69     |
+----------------+--------------------------------------+
```

13.4. DELETING RBAC POLICIES

1. Run the **openstack network rbac list** command to retrieve the ID of your existing role-based access control (RBAC) policies:

```
# openstack network rbac list
+--------------------------------------+-------------+--------------------------------------+
| id                                   | object_type | object_id                            |
+--------------------------------------+-------------+--------------------------------------+
| 314004d0-2261-4d5e-bda7-0181fcf40709 | network     | fa9bb72f-b81a-4572-9c7f-
7237e5fcabd3 |
| bbab1cf9-edc5-47f9-aee3-a413bd582c0a | network     | 9b2f4feb-fee8-43da-bb99-
032e4aaf3f85 |
+--------------------------------------+-------------+--------------------------------------+
```

2. Run the **openstack network rbac delete** command to delete the RBAC, using the ID of the RBAC that you want to delete:

```
# openstack network rbac delete 314004d0-2261-4d5e-bda7-0181fcf40709
Deleted rbac_policy: 314004d0-2261-4d5e-bda7-0181fcf40709
```

13.5. GRANTING RBAC POLICY ACCESS FOR EXTERNAL NETWORKS

You can grant role-based access control (RBAC) policy access to external networks (networks with gateway interfaces attached) using the **--action access_as_external** parameter.

Complete the steps in the following example procedure to create a RBAC for the web-servers network and grant access to the engineering project (c717f263785d4679b16a122516247deb):

- Create a new RBAC policy using the **--action access_as_external** option:

```
# openstack network rbac create --type network --target-project
c717f263785d4679b16a122516247deb --action access_as_external web-servers
 Created a new rbac_policy:
+----------------+--------------------------------------+
| Field          | Value                                |
+----------------+--------------------------------------+
| action         | access_as_external                   |
| id             | ddef112a-c092-4ac1-8914-c714a3d3ba08 |
| object_id      | 6e437ff0-d20f-4483-b627-c3749399bdca |
| object_type    | network                              |
| target_project | c717f263785d4679b16a122516247deb     |
| project_id     | c717f263785d4679b16a122516247deb     |
+----------------+--------------------------------------+
```

As a result, users in the engineering project are able to view the network or connect instances to it:

```
$ openstack network list
+--------------------------------------+-------------+------------------------------------------------------+
| id                                   | name        | subnets                                              |
+--------------------------------------+-------------+------------------------------------------------------+
| 6e437ff0-d20f-4483-b627-c3749399bdca | web-servers | fa273245-1eff-4830-b40c-
57eaeac9b904 192.168.10.0/24 |
+--------------------------------------+-------------+------------------------------------------------------+
```

CHAPTER 14. CONFIGURING DISTRIBUTED VIRTUAL ROUTING (DVR)

14.1. UNDERSTANDING DISTRIBUTED VIRTUAL ROUTING (DVR)

When you deploy Red Hat OpenStack Platform you can choose between a centralized routing model or DVR.

Each model has advantages and disadvantages. Use this document to carefully plan whether centralized routing or DVR better suits your needs.

DVR is enabled by default in new ML2/OVN deployments and disabled by default in new ML2/OVS deployments. The Heat template for the OpenStack Networking (neutron) API (**deployment/neutron/neutron-api-container-puppet.yaml**) contains a parameter to enable and disable Distributed Virtual Routing (DVR). To disable DVR, use the following in an environment file:

```
parameter_defaults:
  NeutronEnableDVR: false
```

14.1.1. Overview of Layer 3 routing

OpenStack Networking (neutron) provides routing services for project networks. Without a router, instances in a project network can communicate with other instances over a shared L2 broadcast domain. Creating a router and assigning it to a tenant network allows the instances in that network to communicate with other project networks or upstream (if an external gateway is defined for the router).

14.1.2. Routing flows

Routing services in OpenStack can be categorized into three main flows:

- **East-West routing** - routing of traffic between different networks in the same tenant. This traffic does not leave the OpenStack deployment. This definition applies to both IPv4 and IPv6 subnets.
- **North-South routing with floating IPs** - Floating IP addressing is a one-to-one NAT that can be modified and that floats between instances. While floating IPs are modeled as a one-to-one association between the floating IP and a neutron port, they are implemented by association with a neutron router that performs the NAT translation. The floating IPs themselves are taken from the uplink network that provides the router with external connectivity. As a result, instances can communicate with external resources (such as endpoints in the internet) or the other way around. Floating IPs are an IPv4 concept and do not apply to IPv6. It is assumed that the IPv6 addressing used by projects uses Global Unicast Addresses (GUAs) with no overlap across the projects, and therefore can be routed without NAT.
- **North-South routing without floating IPs** (also known as *SNAT*) - Neutron offers a default port address translation (PAT) service for instances that do not have allocated floating IPs. With this service, instances can communicate with external endpoints through the router, but not the other way around. For example, an instance can browse a website in the Internet, but a web browser outside cannot browse a website hosted within the instance. SNAT is applied for IPv4 traffic only. In addition, neutron project networks that are assigned GUAs prefixes do not require NAT on the neutron router external gateway port to access the outside world.

14.1.3. Centralized routing

Originally, neutron was designed with a centralized routing model where a project's virtual routers, managed by the neutron L3 agent, are all deployed in a dedicated node or cluster of nodes (referred to as the Network node, or Controller node). This means that each time a routing function is required (east/west, floating IPs or SNAT), traffic would traverse through a dedicated node in the topology. This introduced multiple challenges and resulted in sub-optimal traffic flows. For example:

- Traffic between instances flows through a Controller node - when two instances need to communicate with each other using L3, traffic has to hit the Controller node. Even if the instances are scheduled on the same Compute node, traffic still has to leave the Compute node, flow through the Controller, and route back to the Compute node. This negatively impacts performance.

- Instances with floating IPs receive and send packets through the Controller node - the external network gateway interface is available only at the Controller node, so whether the traffic is originating from an instance, or destined to an instance from the external network, it has to flow through the Controller node. Consequently, in large environments the Controller node is subject to heavy traffic load. This would affect performance and scalability, and also requires careful planning to accommodate enough bandwidth in the external network gateway interface. The same requirement applies for SNAT traffic.

To better scale the L3 agent, neutron can use the L3 HA feature, which distributes the virtual routers across multiple nodes. In the event that a Controller node is lost, the HA router will failover to a standby on another node and there will be packet loss until the HA router failover completes.

14.2. DVR OVERVIEW

Distributed Virtual Routing (DVR) offers an alternative routing design. DVR isolates the failure domain of the Controller node and optimizes network traffic by deploying the L3 agent and schedule routers on every Compute node. DVR has these characteristics:

- East-West traffic is routed directly on the Compute nodes in a distributed fashion.

- North-South traffic with floating IP is distributed and routed on the Compute nodes. This requires the external network to be connected to every Compute node.

- North-South traffic without floating IP is not distributed and still requires a dedicated Controller node.

- The L3 agent on the Controller node uses the **dvr_snat** mode so that the node serves only SNAT traffic.

- The neutron metadata agent is distributed and deployed on all Compute nodes. The metadata proxy service is hosted on all the distributed routers.

14.3. DVR KNOWN ISSUES AND CAVEATS

NOTE

For Red Hat OpenStack Platform 16.1, do not use DVR unless your kernel version is at least **kernel-3.10.0-514.1.1.el7**

- Support for DVR is limited to the ML2 core plug-in and the Open vSwitch (OVS) mechanism driver or ML2/OVN mechanism driver. Other back ends are not supported.

- On OVS DVR deployments, network traffic for the Red Hat OpenStack Platform Load-balancing service (octavia) goes through the Controller and network nodes, instead of the compute nodes.
- SNAT (source network address translation) traffic is not distributed, even when DVR is enabled. SNAT does work, but all ingress/egress traffic must traverse through the centralized Controller node.
- IPv6 traffic is not distributed, even when DVR is enabled. IPv6 routing does work, but all ingress/egress traffic must traverse through the centralized Controller node. If you use IPv6 routing extensively, do not use DVR.
- DVR is not supported in conjunction with L3 HA. If you use DVR with Red Hat OpenStack Platform 16.1 director, L3 HA is disabled. This means that routers are still scheduled on the Network nodes (and load-shared between the L3 agents), but if one agent fails, all routers hosted by this agent fail as well. This affects only SNAT traffic. The **allow_automatic_l3agent_failover** feature is recommended in such cases, so that if one network node fails, the routers are rescheduled to a different node.
- DHCP servers, which are managed by the neutron DHCP agent, are not distributed and are still deployed on the Controller node. The DHCP agent is deployed in a highly available configuration on the Controller nodes, regardless of the routing design (centralized or DVR).
- To work with floating IPs, each Compute node requires an interface on the *External* network. In addition, each Compute node requires one additional IP address. This is due to the implementation of the external gateway port and the floating IP network namespace.
- VLAN, GRE, and VXLAN are all supported for project data separation. When you use GRE or VXLAN, you must enable the L2 Population feature. The Red Hat OpenStack Platform director enforces L2 Population during installation.

14.4. SUPPORTED ROUTING ARCHITECTURES

- Red Hat Enterprise Linux OpenStack Platform 8 through to Red Hat OpenStack Platform 15 for centralized, HA routing.
- Red Hat OpenStack Platform 12 or later with distributed routing.
- Upgrading from a Red Hat OpenStack Platform 8 or later deployment running centralized HA routing to Red Hat OpenStack Platform 10 or later with only distributed routing.

14.5. DEPLOYING DVR WITH ML2 OVS

The **neutron-ovs-dvr.yaml** environment file contains the configuration for the required DVR-specific parameters. Configuring DVR for arbitrary deployment configuration requires additional consideration. The requirements are the following:

A. You must configure the interface connected to the physical network for external network traffic on both the Compute and Controller nodes.

B. You must create a bridge on Compute and Controller nodes, with an interface for external network traffic.

C. You must configure Neutron to allow X to use this bridge.

The host networking configuration (A. and B.) are controlled by Heat templates that pass configuration to the Heat-managed nodes for consumption by the os-net-config process. This is essentially automation of provisioning host networking. You must also configure Neutron (C.) to match the provisioned networking environment. The defaults are not expected to work in production environments.

A proof-of-concept environment using the typical defaults might be similar to the following example:

1. Verify that the value for **OS::TripleO::Compute::Net::SoftwareConfig** in the *environments/neutron-ovs-dvr.yaml* file is the same as the current **OS::TripleO::Controller::Net::SoftwareConfig** value.
   Normally, you can find this value in the network environment file that you use when deploying the overcloud, for example, *environments/net-multiple-nics.yaml*. This value creates the appropriate external network bridge for the Compute node L3 agent.

   NOTE

   If you customize the network configuration of the Compute node, it may be necessary to add the appropriate configuration to those files instead.

2. Include the *environments/neutron-ovs-dvr.yaml* file in the deployment command when deploying the overcloud:

```
$ openstack overcloud deploy --templates -e /usr/share/openstack-tripleo-heat-
templates/environments/neutron-ovs-dvr.yaml
```

3. Verify that L3 HA is disabled.
   For production environments (or test environments that require special customization, for example, involving network isolation, dedicated NICs, among others) you can use the example environments as a guide. Ensure that the bridge mapping type parameters that the L2 agents use are correct, as well as any reference to external facing bridges for other agents,such as the L3 agent.

   NOTE

   The external bridge configuration for the L3 agent, while currently still provided, is deprecated and will be removed in the future.

14.6. MIGRATING CENTRALIZED ROUTERS TO DISTRIBUTED ROUTING

This section contains information about upgrading to distributed routing for Red Hat OpenStack Platform deployments that use L3 HA centralized routing.

1. Upgrade your deployment and validate that it is working correctly.

2. Run the director stack update to configure DVR.
   For more information, see Deploying DVR with ML2 OVS.

3. Confirm that routing functions correctly through the existing routers.

4. You cannot transition a L3 HA router to *distributed* directly. Instead, for each router, disable the L3 HA option, and then enable the distributed option:

a. Disable the router:

```
$ openstack router set --disable
```

b. Clear high availability:

```
$ openstack router set --no-ha
```

c. Configure the router to use DVR:

```
$ openstack router set --distributed
```

a. Enable the router:

```
$ openstack router set --enable
```

b. Confirm that distributed routing functions correctly.

CHAPTER 15. LOAD BALANCING-AS-A-SERVICE (LBAAS) WITH OCTAVIA

The OpenStack Load-balancing service (Octavia) provides a Load Balancing-as-a-Service (LBaaS) version 2 implementation for Red Hat OpenStack platform director installations. This section contains information about enabling Octavia and assumes that Octavia services are hosted on the same nodes as the Networking API server. By default, the load-balancing services are on the Controller nodes.

NOTE

Red Hat does not support a migration path from Neutron-LBaaS to Octavia. However, there are some unsupported open source tools that are available. For more information, see https://github.com/nmagnezi/nlbaas2octavia-lb-replicator/tree/stable_1.

15.1. OVERVIEW OF OCTAVIA

Octavia uses a set of instances on a Compute node called *amphorae* and communicates with the amphorae over a load-balancing management network (**lb-mgmt-net**).

Octavia includes the following services:

API Controller(octavia_api container)

Communicates with the controller worker for configuration updates and to deploy, monitor, or remove amphora instances.

Controller Worker(octavia_worker container)

Sends configuration and configuration updates to amphorae over the LB network.

Health Manager

Monitors the health of individual amphorae and handles failover events if amphorae fail unexpectedly.

IMPORTANT

Health monitors of type **PING** only check if the member is reachable and responds to ICMP echo requests. **PING** does not detect if your application running on that instance is healthy or not. **PING** should only be used in specific cases where an ICMP echo request is a valid health check.

Housekeeping Manager

Cleans up stale (deleted) database records, manages the spares pool, and manages amphora certificate rotation.

Loadbalancer

The top API object that represents the load balancing entity. The VIP address is allocated when the loadbalancer is created. When creating the loadbalancer, an Amphora instance launches on the compute node.

Amphora

The instance that performs the load balancing. Amphorae are typically instances running on Compute nodes and are configured with load balancing parameters according to the listener, pool, health monitor, L7 policies, and members configuration. Amphorae send a periodic heartbeat to the Health Manager.

Listener

The listening endpoint,for example HTTP, of a load balanced service. A listener might refer to several pools and switch between pools using layer 7 rules.

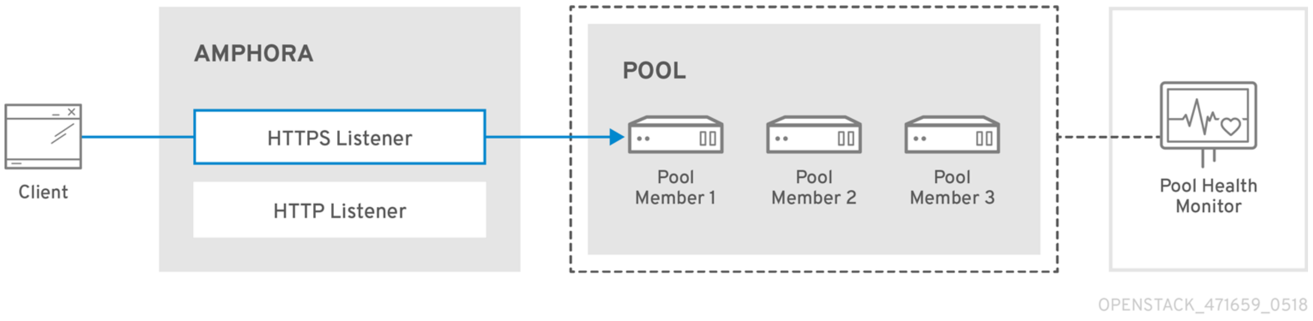
Pool

A group of members that handle client requests from the load balancer (amphora). A pool is associated with only one listener.

Member

Compute instances that serve traffic behind the load balancer (amphora) in a pool.

The following diagram shows the flow of HTTPS traffic through to a pool member:



15.2. OCTAVIA SOFTWARE REQUIREMENTS

Octavia requires that you configure the following core OpenStack components:

- Compute (nova)
- Networking (enable **allowed_address_pairs**)
- Image (glance)
- Identity (keystone)
- RabbitMQ
- MySQL

15.3. PREREQUISITES FOR THE UNDERCLOUD

This section assumes that:

- your undercloud is already installed and ready to deploy an overcloud with Octavia enabled.
- only container deployments are supported.
- Octavia runs on your Controller node.

NOTE

If you want to enable the Octavia service on an existing overcloud deployment, you must prepare the undercloud. Failure to do so results in the overcloud installation being reported as successful yet without Octavia running. To prepare the undercloud, see Transitioning to Containerized Services.

15.3.1. Octavia feature support matrix

Table 15.1. Octavia feature support matrix

Feature	Support level in RHOSP 16.1
ML2/OVS L3 HA	Full support
ML2/OVS DVR	Full support
ML2/OVS L3 HA + composable network node [Network node with OVS, metadata, DHCP, L3 and Octavia (worker, health monitor, house keeping]	Full support
ML2/OVS DVR + composable network node [Network node with OVS, metadata, DHCP, L3 and Octavia (worker, health monitor, house keeping]	Full support
ML2/OVN L3 HA	Full support
ML2/OVN DVR	Full support
Amphora active-standby	Full support—16.1 and later
Terminated HTTPS load balancers (with barbican)	Full support—16.0.1 and later
Amphora spare pool	Technology Preview only
UDP	Full support—16.1 and later
Backup members	Technology Preview only
Provider framework	Technology Preview only
TLS client authentication	Technology Preview only
TLS backend encryption	Technology Preview only
Octavia flavors	Full support
Object tags - API only	Full support 16.1 and later
Listener API timeouts	Full support
Log offloading	Technology Preview only
VIP access control list	Full support

Feature	Support level in RHOSP 16.1
Volume-based amphora	No support

15.4. PLANNING YOUR OCTAVIA DEPLOYMENT

Red Hat OpenStack Platform provides a workflow task to simplify the post-deployment steps for the Load-balancing service. This Octavia workflow runs a set of Ansible playbooks to provide the following post-deployment steps as the last phase of the overcloud deployment:

- Configure certificates and keys.
- Configure the load-balancing management network between the amphorae and the Octavia Controller worker and health manager.

NOTE

Do not modify the OpenStack heat templates directly. Create a custom environment file (for example, **octavia-environment.yaml**) to override default parameter values.

Amphora Image

On pre-provisioned servers, you must install the amphora image on the undercloud before deploying octavia:

```
$ sudo dnf install octavia-amphora-image-x86_64.noarch
```

On servers that are not pre-provisioned, Red Hat OpenStack Platform director automatically downloads the default amphora image, uploads it to the overcloud Image service, and then configures Octavia to use this amphora image. During a stack update or upgrade, director updates this image to the latest amphora image.

NOTE

Custom amphora images are not supported.

15.4.1. Configuring Octavia certificates and keys

Octavia containers require secure communication with load balancers and with each other. You can specify your own certificates and keys or allow these to be automatically generated by Red Hat OpenStack Platform director. We recommend you allow director to automatically create the required private certificate authorities and issue the necessary certificates.

If you must use your own certificates and keys, complete the following steps:

1. On the machine on which you run the **openstack overcloud deploy** command, create a custom YAML environment file.

   ```
   $ vi /home/stack/templates/octavia-environment.yaml
   ```

2. In the YAML environment file, add the following parameters with values appropriate for your site:

- **OctaviaCaCert**:
  The certificate for the CA Octavia will use to generate certificates.

- **OctaviaCaKey**:
  The private CA key used to sign the generated certificates.

- **OctaviaClientCert**:
  The client certificate and un-encrypted key issued by the Octavia CA for the controllers.

- **OctaviaCaKeyPassphrase**:
  The passphrase used with the private CA key above.

- **OctaviaGenerateCerts**:
  The Boolean that instructs director to enable (true) or disable (false) automatic certificate and key generation.

  Here is an example:

  NOTE

  The certificates and keys are multi-line values, and you must indent all of the lines to the same level.

```
parameter_defaults:
   OctaviaCaCert: |
     -----BEGIN CERTIFICATE-----

MIIDgzCCAmugAwIBAgIJAKk46qw6ncJaMA0GCSqGSIb3DQEBCwUAMFgxCzAJBgNV
     [snip]
     sFW3S2roS4X0Af/kSSD8mlBBTFTCMBAj6rtLBKLaQbIxEpIzrgvp
     -----END CERTIFICATE-----

   OctaviaCaKey: |
     -----BEGIN RSA PRIVATE KEY-----
     Proc-Type: 4,ENCRYPTED
     [snip]
     -----END RSA PRIVATE KEY-----[

   OctaviaClientCert: |
     -----BEGIN CERTIFICATE-----

MIIDmjCCAoKgAwIBAgIBATANBgkqhkiG9w0BAQsFADBcMQswCQYDVQQGEwJVUzEP

     [snip]
     270l5ILSnfejLxDH+vI=
     -----END CERTIFICATE-----
     -----BEGIN PRIVATE KEY-----

MIIEvgIBADANBgkqhkiG9w0BAQEFAASCBKgwggSkAgEAAoIBAQDU771O8MTQV8RY

     [snip]
     KfrjE3UqTF+ZaaIQaz3yayXW
     -----END PRIVATE KEY-----

   OctaviaCaKeyPassphrase:
```

```
  b28c519a-5880-4e5e-89bf-c042fc75225d

OctaviaGenerateCerts: false
[rest of file snipped]
```

WARNING

If you use the default certificates that director generates, and if you set the **OctaviaGenerateCerts** parameter to **false**, then your certificates do not renew automatically when they expire.

15.5. ENABLING AMPHORA ACTIVE-STANDBY TOPOLOGY

By default, the amphora provider driver is configured for a single amphora topology with limited support for high availability (HA). However, you are able to make amphora highly available when you implement an active-standby topology for amphora. In this topology, Octavia boots an active and standby amphora for each load balancer, and maintains session persistence between each. Should the active amphora become unhealthy, the amphora automatically fails over to the standby amphora making it active. The octavia health manager automatically rebuilds an amphora that has failed.

Procedure

1. On the machine on which you run the openstack overcloud deploy command, create a custom YAML environment file.

   ```
   $ vi /home/stack/templates/octavia-environment.yaml
   ```

2. In the custom environment file, add the following parameter:

   ```
   parameter_defaults:
     OctaviaLoadBalancerTopology: "ACTIVE_STANDBY"
   ```

3. Run the RHOSP director command, **openstack overcloud deploy** and include all your current environment files and this custom new one.

Verification steps

- After the deployment is complete and you have created a load balancer, run the following command:

  ```
  $ openstack loadbalancer amphora list
  ```

  If your amphora HA configuration is successful, you see output for two amphorae. One amphora has the **MASTER** role, and the other amphora has the **BACKUP** role.

Additional resources

- Modifying the overcloud environment in the *Director Installation and Usage* guide.

15.6. DEPLOYING OCTAVIA

You deploy Octavia using the Red Hat OpenStack Platform (RHOSP) director. Director uses heat templates to deploy Octavia (and other RHOSP components).

Ensure that your environment has access to the Octavia image. For more information about image registry methods, see the Containerization section of the *Director Installation and Usage* guide.

To deploy Octavia in the overcloud:

```
$ openstack overcloud deploy --templates -e \
/usr/share/openstack-tripleo-heat-templates/environments/services/octavia.yaml
```

NOTE

The director updates the amphora image to the latest amphora image during a stack update or upgrade.

15.7. CHANGING OCTAVIA DEFAULT SETTINGS

If you want to override the default parameters that director uses to deploy Octavia, you can specify your values in one or more custom, YAML-formatted environment files (for example, **octavia-environment.yaml**).

IMPORTANT

As a best practice, always make Octavia configuration changes to the appropriate Heat templates and re-run Red Hat OpenStack Platform director. You risk losing ad hoc configuration changes when you manually change individual files and do not use director.

The parameters that director uses to deploy and configure Octavia are pretty straightforward. Here are a few examples:

- **OctaviaControlNetwork**
  The name for the neutron network used for the amphora control network.
- **OctaviaControlSubnetCidr**
  The subnet for amphora control subnet in CIDR form.
- **OctaviaMgmtPortDevName**
  The name of the Octavia management network interface used for communication between octavia worker/health-manager with the amphora machine.
- **OctaviaConnectionLogging**
  A Boolean that enables (true) and disables connection flow logging in load-balancing instances (amphorae). As the amphorae have log rotation on, logs are unlikely to fill up disks. When disabled, performance is marginally impacted.

For the list of Octavia parameters that director uses, consult the following file on the undercloud:

```
/usr/share/openstack-tripleo-heat-templates/deployment/octavia/octavia-deployment-config.j2.yaml
```

Your environment file must contain the keywords **parameter_defaults:**. Put your parameter value pairs after the **parameter_defaults:** keyword. Here is an example:

```
parameter_defaults:
    OctaviaMgmtPortDevName: "o-hm0"
    OctaviaControlNetwork: 'lb-mgmt-net'
    OctaviaControlSubnet: 'lb-mgmt-subnet'
    OctaviaControlSecurityGroup: 'lb-mgmt-sec-group'
    OctaviaControlSubnetCidr: '172.24.0.0/16'
    OctaviaControlSubnetGateway: '172.24.0.1'
    OctaviaControlSubnetPoolStart: '172.24.0.2'
    OctaviaControlSubnetPoolEnd: '172.24.255.254'
```

TIP

YAML files are extremely sensitive about where in the file a parameter is placed. Make sure that **parameter_defaults:** starts in the first column (no leading whitespace characters), and your parameter value pairs start in column five (each parameter has four whitespace characters in front of it).

15.8. SECURE A LOAD BALANCER WITH AN ACCESS CONTROL LIST

You can use the Octavia API to create an access control list (ACL) to limit incoming traffic to a listener to a set of allowed source IP addresses. Any other incoming traffic is rejected.

Prerequisites

This document uses the following assumptions for demonstrating how to secure an Octavia load balancer:

- The back-end servers, 192.0.2.10 and 192.0.2.11, are on subnet named **private-subnet**, and have been configured with a custom application on TCP port 80.
- The subnet, **public-subnet**, is a shared external subnet created by the cloud operator that is reachable from the internet.
- The load balancer is a basic load balancer that is accessible from the internet and distributes requests to the back-end servers.
- The application on TCP port 80 is accessible to limited-source IP addresses (192.0.2.0/24 and 198.51.100/24).

Procedure

1. Create a load balancer (**lb1**) on the subnet (**public-subnet**):

   NOTE

   Replace the names in parentheses () with names that your site uses.

   ```
   $ openstack loadbalancer create --name lb1 --vip-subnet-id public-subnet
   ```

2. Re-run the following command until the load balancer (**lb1**) shows a status that is **ACTIVE** and **ONLINE**:

```
$ openstack loadbalancer show lb1
```

3. Create listener (**listener1**) with the allowed CIDRs:

```
$ openstack loadbalancer listener create --name listener1 --protocol TCP --protocol-port 80 -
-allowed-cidr 192.0.2.0/24 --allowed-cidr 198.51.100/24 lb1
```

4. Create the default pool (**pool1**) for the listener (**listener1**):

```
$ openstack loadbalancer pool create --name pool1 --lb-algorithm ROUND_ROBIN --listener
listener1 --protocol TCP
```

5. Add members 192.0.2.10 and 192.0.2.11 on the subnet (**private-subnet**) to the pool (**pool1**) that you created:

```
$ openstack loadbalancer member create --subnet-id private-subnet --address 192.0.2.10 --
protocol-port 80 pool1

$ openstack loadbalancer member create --subnet-id private-subnet --address 192.0.2.11 --
protocol-port 80 pool1
```

Verification steps

1. Enter the following command:

```
$ openstack loadbalancer listener show listener1
```

You should see output similar to the following:

```
+-----------------------------+--------------------------------------+
| Field                       | Value                                |
+-----------------------------+--------------------------------------+
| admin_state_up              | True                                 |
| connection_limit            | -1                                   |
| created_at                  | 2019-12-09T11:38:05                  |
| default_pool_id             | None                                 |
| default_tls_container_ref   | None                                 |
| description                 |                                      |
| id                          | d26ba156-03c3-4051-86e8-f8997a202d8e |
| insert_headers              | None                                 |
| l7policies                  |                                      |
| loadbalancers               | 2281487a-54b9-4c2a-8d95-37262ec679d6 |
| name                        | listener1                            |
| operating_status            | ONLINE                               |
| project_id                  | 308ca9f600064f2a8b3be2d57227ef8f     |
| protocol                    | TCP                                  |
| protocol_port               | 80                                   |
| provisioning_status         | ACTIVE                               |
| sni_container_refs          | []                                   |
| timeout_client_data         | 50000                                |
| timeout_member_connect      | 5000                                 |
| timeout_member_data         | 50000                                |
| timeout_tcp_inspect         | 0                                    |
| updated_at                | 2019-12-09T11:38:14                  |
| client_ca_tls_container_ref | None                               |
| client_authentication     | NONE                                 |
| client_crl_container_ref  | None                                 |
| allowed_cidrs             | 192.0.2.0/24                         |
|                           | 198.51.100/24                        |
+---------------------------+--------------------------------------+
```

The parameter, **allowed_cidrs**, is set to allow traffic only from 192.0.2.0/24 and 198.51.100/24.

2. To verify that the load balancer is secure, try to make a request to the listener from a client whose CIDR is not in the **allowed_cidrs** list; the request should not succeed. You should see output similar to the following:

```
curl: (7) Failed to connect to 10.0.0.226 port 80: Connection timed out
curl: (7) Failed to connect to 10.0.0.226 port 80: Connection timed out
curl: (7) Failed to connect to 10.0.0.226 port 80: Connection timed out
curl: (7) Failed to connect to 10.0.0.226 port 80: Connection timed out
```

15.9. CONFIGURING AN HTTP LOAD BALANCER

To configure a simple HTTP load balancer, complete the following steps:

1. Create the load balancer on a subnet:

```
$ openstack loadbalancer create --name lb1 --vip-subnet-id private-subnet
```

2. Monitor the state of the load balancer:

```
$ openstack loadbalancer show lb1
```

When you see a status of *Active* and *ONLINE*, the load balancer is created and running and you can go to the next step.

NOTE

To check load balancer status from the Compute service (nova), use the **openstack server list --all | grep amphora** command. Creating load balancers can be a slow process (status displaying as *PENDING*) because load balancers are virtual machines (VMs) and not containers.

3. Create a listener:

```
$ openstack loadbalancer listener create --name listener1 --protocol HTTP --protocol-port 80
lb1
```

4. Create the listener default pool:

```
$ openstack loadbalancer pool create --name pool1 --lb-algorithm ROUND_ROBIN --listener
listener1 --protocol HTTP
```

5. Create a health monitor on the pool to test the “/healthcheck” path:

```
$ openstack loadbalancer healthmonitor create --delay 5 --max-retries 4 --timeout 10 --type
HTTP --url-path /healthcheck pool1
```

6. Add load balancer members to the pool:

```
$ openstack loadbalancer member create --subnet-id private-subnet --address 192.0.2.10 --
protocol-port 80 pool1
$ openstack loadbalancer member create --subnet-id private-subnet --address 192.0.2.11 --
protocol-port 80 pool1
```

7. Create a floating IP address on a public subnet:

```
$ openstack floating ip create public
```

8. Associate this floating IP with the load balancer VIP port:

```
$ openstack floating ip set --port `_LOAD_BALANCER_VIP_PORT_` `_FLOATING_IP_`
```

TIP

To locate *LOAD_BALANCER_VIP_PORT*, run the **openstack loadbalancer show lb1** command.

15.10. VERIFYING THE LOAD BALANCER

To verify the load balancer, complete the following steps:

1. Run the **openstack loadbalancer show** command to verify the load balancer settings:

```
(overcloud) [stack@undercloud-0 ~]$ openstack loadbalancer show lb1
+---------------------+--------------------------------------+
| Field               | Value                                |
+---------------------+--------------------------------------+
| admin_state_up      | True                                 |
| created_at          | 2018-04-18T12:28:34                  |
| description         |                                      |
| flavor              |                                      |
| id                  | 788fe121-3dec-4e1b-8360-4020642238b0 |
| listeners           | 09f28053-fde8-4c78-88b9-0f191d84120e |
| name                | lb1                                  |
| operating_status    | ONLINE                               |
| pools               | 627842b3-eed8-4f5f-9f4a-01a738e64d6a |
| project_id          | dda678ca5b1241e7ad7bf7eb211a2fd7     |
| provider            | octavia                              |
| provisioning_status | ACTIVE                               |
| updated_at          | 2018-04-18T14:03:09                  |
| vip_address         | 192.168.0.11                         |
| vip_network_id      | 9bca13be-f18d-49a5-a83d-9d487827fd16 |
| vip_port_id         | 69a85edd-5b1c-458f-96f2-b4552b15b8e6 |
| vip_qos_policy_id   | None                                 |
| vip_subnet_id       | 5bd7334b-49b3-4849-b3a2-b0b83852dba1 |
+---------------------+--------------------------------------+
```

2. Run the **amphora list** command to find the UUID of the amphora associated with load balancer lb1:

```
(overcloud) [stack@undercloud-0 ~]$ openstack loadbalancer amphora list | grep <UUID of
loadbalancer lb1>
```

3. Run the **amphora show** command with the amphora UUID to view amphora information:

```
(overcloud) [stack@undercloud-0 ~]$ openstack loadbalancer amphora show 62e41d30-
1484-4e50-851c-7ab6e16b88d0
+-----------------+--------------------------------------+
| Field           | Value                                |
+-----------------+--------------------------------------+
| id              | 62e41d30-1484-4e50-851c-7ab6e16b88d0 |
| loadbalancer_id | 53a497b3-267d-4abc-968f-94237829f78f |
| compute_id      | 364efdb9-679c-4af4-a80c-bfcb74fc0563 |
| lb_network_ip   | 192.168.0.13                         |
| vrrp_ip         | 10.0.0.11                            |
| ha_ip           | 10.0.0.10                            |
| vrrp_port_id    | 74a5c1b4-a414-46b8-9263-6328d34994d4 |
| ha_port_id      | 3223e987-5dd6-4ec8-9fb8-ee34e63eef3c |
| cert_expiration | 2020-07-16T12:26:07                  |
| cert_busy       | False                                |
| role            | BACKUP                               |
| status          | ALLOCATED                            |
| vrrp_interface  | eth1                                 |
| vrrp_id         | 1                                    |
| vrrp_priority   | 90                                   |
| cached_zone     | nova                                 |
| created_at      | 2018-07-17T12:26:07                  |
| updated_at      | 2018-07-17T12:30:36                  |
| image_id        | a3f9f3e4-92b6-4a27-91c8-ddc69714da8f |
+-----------------+--------------------------------------+
```

4. Run the **openstack loadbalancer listener show** command to view the listener details:

```
(overcloud) [stack@undercloud-0 ~]$ openstack loadbalancer listener show listener1
------------------------------------------------------------------------------------------------+
| Field                     | Value                                                            |
------------------------------------------------------------------------------------------------+
| admin_state_up            | True                                                             |
| connection_limit          | -1                                                               |
| created_at                | 2018-04-18T12:51:25                                              |
| default_pool_id           | 627842b3-eed8-4f5f-9f4a-01a738e64d6a                             |
| default_tls_container_ref | http://10.0.0.101:9311/v1/secrets/7eafeabb-b4a1-4bc4-8098-
b6281736bfe2 |
| description               |                                                                  |
| id                        | 09f28053-fde8-4c78-88b9-0f191d84120e                             |
| insert_headers            | None                                                             |
| l7policies                |                                                                  |
| loadbalancers             | 788fe121-3dec-4e1b-8360-4020642238b0                             |
| name                      | listener1                                                        |
| operating_status          | ONLINE                                                           |
| project_id                | dda678ca5b1241e7ad7bf7eb211a2fd7                                 |
| protocol                  | TERMINATED_HTTPS                                                 |
| protocol_port             | 443                                              |
| provisioning_status       | ACTIVE                                           |
| sni_container_refs        | []                                               |
| updated_at                | 2018-04-18T14:03:09                              |
------------------------------------------------------------------------------------------------+
```

5. Run the **openstack loadbalancer pool show** command to view the pool and load-balancer members:

```
(overcloud) [stack@undercloud-0 ~]$ openstack loadbalancer pool show pool1
+---------------------+--------------------------------------+
| Field               | Value                                |
+---------------------+--------------------------------------+
| admin_state_up      | True                                 |
| created_at          | 2018-04-18T12:53:49                  |
| description         |                                      |
| healthmonitor_id    |                                      |
| id                  | 627842b3-eed8-4f5f-9f4a-01a738e64d6a |
| lb_algorithm        | ROUND_ROBIN                          |
| listeners           | 09f28053-fde8-4c78-88b9-0f191d84120e |
| loadbalancers       | 788fe121-3dec-4e1b-8360-4020642238b0 |
| members             | b85c807e-4d7c-4cbd-b725-5e8afddf80d2 |
|                     | 40db746d-063e-4620-96ee-943dcd351b37 |
| name                | pool1                                |
| operating_status    | ONLINE                               |
| project_id          | dda678ca5b1241e7ad7bf7eb211a2fd7     |
| protocol            | HTTP                                 |
| provisioning_status | ACTIVE                               |
| session_persistence | None                                 |
| updated_at          | 2018-04-18T14:03:09                  |
+---------------------+--------------------------------------+
```

6. If a health monitor is present, you can check the status of each member.
   A working member should have an **ONLINE** value for its **operating_status**.

```
(overcloud) [stack@undercloud-0 ~]$ openstack loadbalancer member show <member
UUID>
+---------------------+--------------------------------------+
| Field               | Value                                |
+---------------------+--------------------------------------+
| address             | 192.168.0.32                         |
| admin_state_up      | True                                 |
| created_at          | 2018-04-18T14:03:09                  |
| id                  | <member UUID>                        |
| name                |                                      |
| operating_status    | ONLINE                               |
| project_id          | dda678ca5b1241e7ad7bf7eb211a2fd7     |
| protocol_port       | 80                                   |
| provisioning_status | ACTIVE                               |
| subnet_id           | 5bd7334b-49b3-4849-b3a2-b0b83852dba1 |
| updated_at          | 2018-04-18T14:03:09                  |
| weight              | 1                                    |
| monitor_port        | None                                 |
| monitor_address     | None                                 |
| backup              | False                                |
+---------------------+--------------------------------------+
```

7. Run the **openstack floating ip list** command to verify the floating IP address:

```
(overcloud) [stack@undercloud-0 ~]$ openstack floating ip list
-------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------+
| ID                                   | Floating IP Address | Fixed IP Address | Port                                 |
Floating Network                     | Project                          |
-------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------+
| 89661971-fa65-4fa6-b639-563967a383e7 | 10.0.0.213          | 192.168.0.11     | 69a85edd-
5b1c-458f-96f2-b4552b15b8e6 | fe0f3854-fcdc-4433-bc57-3e4568e4d944 |
dda678ca5b1241e7ad7bf7eb211a2fd7 |
-------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------+
```

8. Verify HTTPS traffic flows across the load balancer:

```
(overcloud) [stack@undercloud-0 ~]$ curl -v https://10.0.0.213 --insecure
* About to connect() to 10.0.0.213 port 443 (#0)
*   Trying 10.0.0.213...
* Connected to 10.0.0.213 (10.0.0.213) port 443 (#0)
* Initializing NSS with certpath: sql:/etc/pki/nssdb
* skipping SSL peer certificate verification
* SSL connection using TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
* Server certificate:
*  subject: CN=www.example.com,O=Dis,L=Springfield,ST=Denial,C=US
*  start date: Apr 18 09:21:45 2018 GMT
*  expire date: Apr 18 09:21:45 2019 GMT
*  common name: www.example.com
*  issuer: CN=www.example.com,O=Dis,L=Springfield,ST=Denial,C=US
> GET / HTTP/1.1
> User-Agent: curl/7.29.0
> Host: 10.0.0.213
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 30
<
* Connection #0 to host 10.0.0.213 left intact
```

15.11. OVERVIEW OF TLS-TERMINATED HTTPS LOAD BALANCER

When a TLS-terminated HTTPS load balancer is implemented, web clients communicate with the load balancer over Transport Layer Security (TLS) protocols. The load balancer terminates the TLS session and forwards the decrypted requests to the back end servers.

By terminating the TLS session on the load balancer, we offload the CPU-intensive encryption operations to the load balancer, and allows the load balancer to use advanced features such as Layer 7 inspection.

15.12. CREATING A TLS-TERMINATED HTTPS LOAD BALANCER

This procedure describes how to configure a TLS-terminated HTTPS load balancer that is accessible from the internet through Transport Layer Security (TLS) and distributes requests to the back end servers over the non-encrypted HTTP protocol.

Prerequisites

- A private subnet that contains back end servers that host non-secure HTTP applications on TCP port 80.
- A shared external (public) subnet that is reachable from the internet.
- TLS public-key cryptography has been configured with the following characteristics:
  - A TLS certificate, key, and intermediate certificate chain have been obtained from an external certificate authority (CA) for the DNS name assigned to the load balancer VIP address (for example, www.example.com).
  - The certificate, key, and intermediate certificate chain reside in separate files in the current directory.
  - The key and certificate are PEM-encoded.
  - The key is not encrypted with a passphrase.
  - The intermediate certificate chain is contains multiple certificates that are PEM-encoded and concatenated together.
- You must configure the Load-balancing service (octavia) to use the Key Manager service (barbican). For more information, see the Manage Secrets with OpenStack Key Manager guide .

Procedure

1. Combine the key (**server.key**), certificate (**server.crt**), and intermediate certificate chain (**ca-chain.crt**) into a single PKCS12 file (**server.p12**).

   NOTE

   The values in parentheses are provided as examples. Replace these with values appropriate for your site.

   ```
   $ openssl pkcs12 -export -inkey server.key -in server.crt -certfile ca-chain.crt -passout pass:
   -out server.p12
   ```

2. Using the Key Manager service, create a secret resource (**tls_secret1**) for the PKCS12 file.

   ```
   $ openstack secret store --name='tls_secret1' -t 'application/octet-stream' -e 'base64' --
   payload="$(base64 < server.p12)"
   ```

3. Create a load balancer (**lb1**) on the public subnet (**public-subnet**).

   ```
   $ openstack loadbalancer create --name lb1 --vip-subnet-id public-subnet
   ```

4. Before you can proceed, the load balancer that you created (**lb1**) must be in an active and online state.
   Run the **openstack loadbalancer show** command, until the load balancer responds with an **ACTIVE** and **ONLINE** status. (You might need to run this command more than once.)

   ```
   $ openstack loadbalancer show lb1
   ```

5. Create a TERMINATED_HTTPS listener (**listener1**), and reference the secret resource as the default TLS container for the listener.

   ```
   $ openstack loadbalancer listener create --protocol-port 443 --protocol
   TERMINATED_HTTPS --name listener1 --default-tls-container=$(openstack secret list | awk
   '/ tls_secret1 / {print $2}') lb1
   ```

6. Create a pool (**pool1**) and make it the default pool for the listener.

   ```
   $ openstack loadbalancer pool create --name pool1 --lb-algorithm ROUND_ROBIN --listener
   listener1 --protocol HTTP
   ```

7. Add the non-secure HTTP back end servers (**192.0.2.10** and **192.0.2.11**) on the private subnet (**private-subnet**) to the pool.

   ```
   $ openstack loadbalancer member create --subnet-id private-subnet --address 192.0.2.10 --
   protocol-port 80 pool1
   $ openstack loadbalancer member create --subnet-id private-subnet --address 192.0.2.11 --
   protocol-port 80 pool1
   ```

15.13. CREATING A TLS-TERMINATED HTTPS LOAD BALANCER WITH SNI

This procedure describes how to configure a TLS-terminated HTTPS load balancer that is accessible from the internet through Transport Layer Security (TLS) and distributes requests to the back end servers over the non-encrypted HTTP protocol. In this configuration, there is one listener that contains multiple TLS certificates and implements Server Name Indication (SNI) technology.

Prerequisites

- A private subnet that contains back end servers that host non-secure HTTP applications on TCP port 80.
- A shared external (public) subnet that is reachable from the internet.
- TLS public-key cryptography has been configured with the following characteristics:
  - Multiple TLS certificates, keys, and intermediate certificate chains have been obtained from an external certificate authority (CA) for the DNS names assigned to the load balancer VIP address (for example, www.example.com and www2.example.com).
  - The certificates, keys, and intermediate certificate chains reside in separate files in the current directory.
  - The keys and certificates are PEM-encoded.
  - The keys are not encrypted with passphrases.

  - The intermediate certificate chains contain multiple certificates that are PEM-encoded and are concatenated together.

- You must configure the Load-balancing service (octavia) to use the Key Manager service (barbican). For more information, see the Manage Secrets with OpenStack Key Manager guide .

Procedure

1. For each of the TLS certificates in the SNI list, combine the key (**server.key**), certificate (**server.crt**), and intermediate certificate chain (**ca-chain.crt**) into a single PKCS12 file (**server.p12**).
   In this example, you create two PKCS12 files (**server.p12** and **server2.p12**) one for each certificate (**www.example.com** and **www2.example.com**).

   NOTE

   The values in parentheses are provided as examples. Replace these with values appropriate for your site.

   ```
   $ openssl pkcs12 -export -inkey server.key -in server.crt -certfile ca-chain.crt -passout pass:
   -out server.p12

   $ openssl pkcs12 -export -inkey server2.key -in server2.crt -certfile ca-chain2.crt -passout
   pass: -out server2.p12
   ```

2. Using the Key Manager service, create secret resources (**tls_secret1** and **tls_secret2**) for the PKCS12 file.

   ```
   $ openstack secret store --name='tls_secret1' -t 'application/octet-stream' -e 'base64' --
   payload="$(base64 < server.p12)"
   $ openstack secret store --name='tls_secret2' -t 'application/octet-stream' -e 'base64' --
   payload="$(base64 < server2.p12)"
   ```

3. Create a load balancer (**lb1**) on the public subnet (**public-subnet**).

   ```
   $ openstack loadbalancer create --name lb1 --vip-subnet-id public-subnet
   ```

4. Before you can proceed, the load balancer that you created (**lb1**) must be in an active and online state.
   Run the **openstack loadbalancer show** command, until the load balancer responds with an **ACTIVE** and **ONLINE** status. (You might need to run this command more than once.)

   ```
   $ openstack loadbalancer show lb1
   ```

5. Create a TERMINATED_HTTPS listener (**listener1**), and reference both the secret resources using SNI.
   (Reference **tls_secret1** as the default TLS container for the listener.)

   ```
   $ openstack loadbalancer listener create --protocol-port 443 \
   --protocol TERMINATED_HTTPS --name listener1 \
   --default-tls-container=$(openstack secret list | awk '/ tls_secret1 / {print $2}') \
   --sni-container-refs $(openstack secret list | awk '/ tls_secret1 / {print $2}') \
   $(openstack secret list | awk '/ tls_secret2 / {print $2}') -- lb1
   ```

6. Create a pool (**pool1**) and make it the default pool for the listener.

```
$ openstack loadbalancer pool create --name pool1 --lb-algorithm ROUND_ROBIN --listener
listener1 --protocol HTTP
```

7. Add the non-secure HTTP back end servers (**192.0.2.10** and **192.0.2.11**) on the private subnet (**private-subnet**) to the pool.

```
$ openstack loadbalancer member create --subnet-id private-subnet --address 192.0.2.10 --
protocol-port 80 pool1
$ openstack loadbalancer member create --subnet-id private-subnet --address 192.0.2.11 --
protocol-port 80 pool1
```

15.14. CREATING HTTP AND TLS-TERMINATED HTTPS LOAD BALANCERS ON THE SAME BACK END

This procedure describes how to configure a non-secure listener and a TLS-terminated HTTPS listener on the same load balancer (and same IP address). You would use this configuration when you want to respond to web clients with the exact same content, regardless if the client is connected with secure or non-secure HTTP protocol.

Prerequisites

- A private subnet that contains back end servers that host non-secure HTTP applications on TCP port 80.
- A shared external (public) subnet that is reachable from the internet.
- TLS public-key cryptography has been configured with the following characteristics:
  - A TLS certificate, key, and optional intermediate certificate chain have been obtained from an external certificate authority (CA) for the DNS name assigned to the load balancer VIP address (for example, www.example.com).
  - The certificate, key, and intermediate certificate chain reside in separate files in the current directory.
  - The key and certificate are PEM-encoded.
  - The key is not encrypted with a passphrase.
  - The intermediate certificate chain is contains multiple certificates that are PEM-encoded and concatenated together.
- You must configure the Load-balancing service (octavia) to use the Key Manager service (barbican). For more information, see the Manage Secrets with OpenStack Key Manager guide .
- The non-secure HTTP listener is configured with the same pool as the HTTPS TLS-terminated load balancer.

Procedure

1. Combine the key (**server.key**), certificate (**server.crt**), and intermediate certificate chain (**ca-chain.crt**) into a single PKCS12 file (**server.p12**).

NOTE

The values in parentheses are provided as examples. Replace these with values appropriate for your site.

```
$ openssl pkcs12 -export -inkey server.key -in server.crt -certfile ca-chain.crt -passout pass:
-out server.p12
```

2. Using the Key Manager service, create a secret resource (**tls_secret1**) for the PKCS12 file.

```
$ openstack secret store --name='tls_secret1' -t 'application/octet-stream' -e 'base64' --
payload="$(base64 < server.p12)"
```

3. Create a load balancer (**lb1**) on the public subnet (**public-subnet**).

```
$ openstack loadbalancer create --name lb1 --vip-subnet-id public-subnet
```

4. Before you can proceed, the load balancer that you created (**lb1**) must be in an active and online state.
Run the **openstack loadbalancer show** command, until the load balancer responds with an **ACTIVE** and **ONLINE** status. (You might need to run this command more than once.)

```
$ openstack loadbalancer show lb1
```

5. Create a TERMINATED_HTTPS listener (**listener1**), and reference the secret resource as the default TLS container for the listener.

```
$ openstack loadbalancer listener create --protocol-port 443 --protocol
TERMINATED_HTTPS --name listener1 --default-tls-container=$(openstack secret list | awk
'/ tls_secret1 / {print $2}') lb1
```

6. Create a pool (**pool1**) and make it the default pool for the listener.

```
$ openstack loadbalancer pool create --name pool1 --lb-algorithm ROUND_ROBIN --listener
listener1 --protocol HTTP
```

7. Add the non-secure HTTP back end servers (**192.0.2.10** and **192.0.2.11**) on the private subnet (**private-subnet**) to the pool.

```
$ openstack loadbalancer member create --subnet-id private-subnet --address 192.0.2.10 --
protocol-port 80 pool1
$ openstack loadbalancer member create --subnet-id private-subnet --address 192.0.2.11 --
protocol-port 80 pool1
```

8. Create a non-secure, HTTP listener (**listener2**), and make its default pool, the same as the secure listener.

```
$ openstack loadbalancer listener create --protocol-port 80 --protocol HTTP --name listener2
--default-pool pool1 lb1
```

15.15. CREATING A UDP LOAD BALANCER WITH A HEALTH MONITOR

You can use the Red Hat OpenStack Platform Load-balancing service (octavia) to balance network traffic on UDP ports.

While it is possible to set up a listener without a health monitor, if a back end pool member goes down, octavia does not remove the failed server from the pool. To avoid service disruption, always configure load balancers to use a health monitor on production systems.

Prerequisites

- A private subnet that contains back end servers that host one or more applications configured to use UDP ports.
- A shared external (public) subnet that is reachable from the internet.
- A basic load balancer that is reachable from the internet, and that distributes requests to its members (back end servers).
- Establish a UDP health check to ensure that the back end servers are available with a UDP-CONNECT health monitor.
- Ensure that no security rules block ICMP Destination Unreachable messages (ICMP type 3).

Procedure

1. Create a load balancer (lb1) on a private subnet (private-subnet):

   NOTE

   Values inside parentheses are sample values that are used in the example commands in this procedure. Substitute these sample values with ones that are appropriate for your site.

   ```
   $ openstack loadbalancer create --name lb1 --vip-subnet-id private-subnet
   ```

2. Monitor the state of the load balancer:

   ```
   $ openstack loadbalancer show lb1
   ```

   When you see a status of *Active* and *ONLINE*, the load balancer is created and running and you can go to the next step.

3. Create a listener (listener1) on a port (1234):

   ```
   $ openstack loadbalancer listener create --name listener1 --protocol UDP --protocol-port
   1234 lb1
   ```

4. Create the listener default pool (pool1):

   ```
   $ openstack loadbalancer pool create --name pool1 --lb-algorithm ROUND_ROBIN --listener
   listener1 --protocol UDP
   ```

5. Create a health monitor on the pool that connects to the back end servers:

```
$ openstack loadbalancer healthmonitor create --delay 5 --max-retries 2 --timeout 3 --type
UDP-CONNECT pool1
```

6. Add load balancer members (192.0.2.10 and 192.0.2.11) on the private subnet to the default pool:

```
$ openstack loadbalancer member create --subnet-id private-subnet --address 192.0.2.10 --
protocol-port 1234 pool1
$ openstack loadbalancer member create --subnet-id private-subnet --address 192.0.2.11 --
protocol-port 1234 pool1
```

Verification steps

- See Verifying the load balancer.

Additional resources

- See loadbalancer in the *Command Line Interface Reference*.

15.16. ACCESSING AMPHORA LOGS

Amphora is the instance that performs load balancing. You can view Amphora logging information in the systemd journal.

1. Start the ssh-agent, and add your user identity key to the agent:

```
[stack@undercloud-0] $ eval `ssh-agent -s`
[stack@undercloud-0] $ ssh-add
```

2. Use SSH to connect to the Amphora instance:

```
[stack@undercloud-0] $ ssh -A -t heat-admin@<controller node IP address> ssh cloud-
user@<IP address of Amphora in load-balancing management network>
```

3. View the systemd journal:

```
[cloud-user@amphora-f60af64d-570f-4461-b80a-0f1f8ab0c422 ~] $ sudo journalctl
```

   Refer to the journalctl man page for information about filtering journal output.

4. When you are finished viewing the journal, and have closed your connections to the Amphora instance and the Controller node, make sure that you stop the SSH agent:

```
[stack@undercloud-0] $ exit
```

15.17. UPDATING RUNNING AMPHORA INSTANCES

15.17.1. Overview

Periodically, you must update a running load balancing instance (amphora) with a newer image. For example, you must update an amphora instance during the following events:

- An update or upgrade of Red Hat OpenStack Platform.
- A security update to your system.
- A change to a different flavor for the underlying virtual machine.

To update an amphora image, you must fail over the load balancer and then wait for the load balancer to regain an active state. When the load balancer is again active, it is running the new image.

15.17.2. Prerequisites

New images for amphora are available during an OpenStack update or upgrade.

15.17.3. Update amphora instances with new images

During an OpenStack update or upgrade, director automatically downloads the default amphora image, uploads it to the overcloud Image service (glance), and then configures Octavia to use the new image. When you failover the load balancer, you are forcing Octavia to start the new amphora image.

1. Make sure that you have reviewed the prerequisites before you begin updating amphora.
2. List the IDs for all of the load balancers that you want to update:

   ```
   $ openstack loadbalancer list -c id -f value
   ```

3. Fail over each load balancer:

   ```
   $ openstack loadbalancer failover <loadbalancer_id>
   ```

   NOTE

   When you start failing over the load balancers, monitor system utilization, and as needed, adjust the rate at which you perform failovers. A load balancer failover creates new virtual machines and ports, which might temporarily increase the load on OpenStack Networking.

4. Monitor the state of the failed over load balancer:

   ```
   $ openstack loadbalancer show <loadbalancer_id>
   ```

   The update is complete when the load balancer status is **ACTIVE**.

CHAPTER 16. TENANT NETWORKING WITH IPV6

16.1. OVERVIEW OF TENANT NETWORKING

This chapter contains information about implementing IPv6 subnets in a Red Hat OpenStack Platform (RHOSP) tenant network. In addition to tenant networking, with RHOSP director, you can configure IPv6-native deployments for the overcloud nodes.

RHOSP supports IPv6 in tenant networks. IPv6 subnets are created within existing tenant networks, and support a number of address assignment modes: **Stateless Address Autoconfiguration (SLAAC)**, **Stateful DHCPv6**, and **Stateless DHCPv6**.

16.2. IPV6 SUBNET OPTIONS

Create IPv6 subnets using the **openstack subnet create** command. You can also specify the address mode and the Router Advertisement mode. Use the following list to understand the possible combinations of options that you can include with the **openstack subnet create** command.

RA Mode	Address Mode	Result
ipv6_ra_mode=not set	ipv6-address-mode=slaac	The instance receives an IPv6 address from the external router (not managed by OpenStack Networking) using **SLAAC**.
ipv6_ra_mode=not set	ipv6-address-mode=dhcpv6-stateful	The instance receives an IPv6 address and optional information from OpenStack Networking (dnsmasq) using **DHCPv6 stateful**.
ipv6_ra_mode=not set	ipv6-address-mode=dhcpv6-stateless	The instance receives an IPv6 address from the external router using SLAAC, and optional information from OpenStack Networking (dnsmasq) using **DHCPv6 stateless**.
ipv6_ra_mode=slaac	ipv6-address-mode=not-set	The instance uses SLAAC to receive an IPv6 address from OpenStack Networking (**radvd**).
ipv6_ra_mode=dhcpv6-stateful	ipv6-address-mode=not-set	The instance receives an IPv6 address and optional information from an external DHCPv6 server using **DHCPv6 stateful**.

RA Mode	Address Mode	Result
ipv6_ra_mode=dhcpv6-stateless	ipv6-address-mode=not-set	The instance receives an IPv6 address from OpenStack Networking (**radvd**) using SLAAC, and optional information from an external DHCPv6 server using **DHCPv6 stateless**.
ipv6_ra_mode=slaac	ipv6-address-mode=slaac	The instance receives an IPv6 address from OpenStack Networking (**radvd**) using **SLAAC**.
ipv6_ra_mode=dhcpv6-stateful	ipv6-address-mode=dhcpv6-stateful	The instance receives an IPv6 address from OpenStack Networking (**dnsmasq**) using **DHCPv6 stateful**, and optional information from OpenStack Networking (**dnsmasq**) using **DHCPv6 stateful**.
ipv6_ra_mode=dhcpv6-stateless	ipv6-address-mode=dhcpv6-stateless	The instance receives an IPv6 address from OpenStack Networking (**radvd**) using **SLAAC**, and optional information from OpenStack Networking (**dnsmasq**) using **DHCPv6 stateless**.

16.3. CREATE AN IPV6 SUBNET USING STATEFUL DHCPV6

Complete the steps in this procedure to create an IPv6 subnet in a tenant network using some of the options in section 17.1. First, gather the necessary tenant and network information, then include this information in the **openstack subnet create** command.

NOTE

OpenStack Networking supports only EUI-64 IPv6 address assignment for SLAAC. This allows for simplified IPv6 networking, as hosts self-assign addresses based on the base 64-bits plus the MAC address. You cannot create subnets with a different netmask and *address_assign_type* of SLAAC.

1. Retrieve the tenant ID of the Project where you want to create the IPv6 subnet. These values are unique between OpenStack deployments, so your values differ from the values in this example.

```
# openstack project list
+----------------------------------+----------+
| ID                               | Name     |
+----------------------------------+----------+
| 25837c567ed5458fbb441d39862e1399 |    QA    |
| f59f631a77264a8eb0defc898cb836af |  admin   |
| 4e2e1951e70643b5af7ed52f3ff36539 |   demo   |
| 8561dff8310e4cd8be4b6fd03dc8acf5 | services |
+----------------------------------+----------+
```

2. Retrieve a list of all networks present in OpenStack Networking (neutron), and note the name of the network that you want to host the IPv6 subnet:

```
# openstack network list
+--------------------------------------+------------------+---------------------------------------------------
---------+
| id                                   | name             | subnets                                          |
+--------------------------------------+------------------+---------------------------------------------------
---------+
| 8357062a-0dc2-4146-8a7f-d2575165e363 | private          | c17f74c4-db41-4538-af40-
48670069af70 10.0.0.0/24             |
| 31d61f7d-287e-4ada-ac29-ed7017a54542 | public           | 303ced03-6019-4e79-a21c-
1942a460b920 172.24.4.224/28         |
| 6aff6826-4278-4a35-b74d-b0ca0cbba340 | database-servers |
|
+--------------------------------------+------------------+---------------------------------------------------
---------+
```

3. Include the tenant ID and network name in the **openstack subnet create** command:

```
# openstack subnet create --ip-version 6 --ipv6-address-mode dhcpv6-stateful --project
25837c567ed5458fbb441d39862e1399 --network database-servers --subnet-range
fdf8:f53b:82e4::53/125 subnet_name

Created a new subnet:
+-------------------+--------------------------------------------------------------+
| Field             | Value                                                        |
+-------------------+--------------------------------------------------------------+
| allocation_pools  | {"start": "fdf8:f53b:82e4::52", "end": "fdf8:f53b:82e4::56"} |
| cidr              | fdf8:f53b:82e4::53/125                                       |
| dns_nameservers   |                                                              |
| enable_dhcp       | True                                                         |
| gateway_ip        | fdf8:f53b:82e4::51                                           |
| host_routes       |                                                              |
| id                | cdfc3398-997b-46eb-9db1-ebbd88f7de05                         |
| ip_version        | 6                                                            |
| ipv6_address_mode | dhcpv6-stateful                                              |
| ipv6_ra_mode      |                                                              |
| name              |                                                              |
| network_id        | 6aff6826-4278-4a35-b74d-b0ca0cbba340                         |
| tenant_id         | 25837c567ed5458fbb441d39862e1399                             |
+-------------------+--------------------------------------------------------------+
```

4. Validate this configuration by reviewing the network list. Note that the entry for *database-servers* now reflects the newly created IPv6 subnet:

```
# openstack network list
+--------------------------------------+------------------+---------------------------------------------------
```

```
---------+
| id                                   | name             | subnets                                          |
+--------------------------------------+------------------+--------------------------------------------------
---------+
| 6aff6826-4278-4a35-b74d-b0ca0cbba340 | database-servers | cdfc3398-997b-46eb-9db1-
ebbd88f7de05 fdf8:f53b:82e4::50/125 |
| 8357062a-0dc2-4146-8a7f-d2575165e363 | private          | c17f74c4-db41-4538-af40-
48670069af70 10.0.0.0/24           |
| 31d61f7d-287e-4ada-ac29-ed7017a54542 | public           | 303ced03-6019-4e79-a21c-
1942a460b920 172.24.4.224/28       |
+--------------------------------------+------------------+--------------------------------------------------
---------+
```

As a result of this configuration, instances that the QA tenant creates can receive a DHCP IPv6 address when added to the database-servers subnet:

```
# openstack server list
+--------------------------------------+------------+--------+------------+-------------+---------------------
----------------+
| ID                                   | Name       | Status | Task State | Power State | Networks
|
+--------------------------------------+------------+--------+------------+-------------+---------------------
----------------+
| fad04b7a-75b5-4f96-aed9-b40654b56e03 | corp-vm-01 | ACTIVE | -          | Running     |
database-servers=fdf8:f53b:82e4::52 |
+--------------------------------------+------------+--------+------------+-------------+---------------------
----------------+
```

CHAPTER 17. MANAGING TENANT QUOTAS

17.1. CONFIGURING TENANT QUOTAS

OpenStack Networking (neutron) supports the use of quotas to constrain the number of resources created by tenants/projects.

- You can set tenant quotas for various network components in the /var/lib/config-data/neutron/etc/neutron/neutron.conf file.
  For example, to limit the number of routers that a tenant can create, change the **quota_router** value:

  ```
  quota_router = 10
  ```

  In this example, each tenant is limited to a maximum of 10 routers.

For a listing of the quota settings, see sections that immediately follow.

17.2. L3 QUOTA OPTIONS

Here are quota options available for layer 3 (L3) networking:

- **quota_floatingip** - The number of floating IPs available to a tenant.
- **quota_network** - The number of networks available to a tenant.
- **quota_port** - The number of ports available to a tenant.
- **quota_router** - The number of routers available to a tenant.
- **quota_subnet** - The number of subnets available to a tenant.
- **quota_vip** - The number of virtual IP addresses available to a tenant.

17.3. FIREWALL QUOTA OPTIONS

Here are quota options available for managing firewalls for tenants:

- **quota_firewall** - The number of firewalls available to a tenant.
- **quota_firewall_policy** - The number of firewall policies available to a tenant.
- **quota_firewall_rule** - The number of firewall rules available to a tenant.

17.4. SECURITY GROUP QUOTA OPTIONS

Here are quota options available for managing the number of security groups that tenants can create:

- **quota_security_group** - The number of security groups available to a tenant.
- **quota_security_group_rule** - The number of security group rules available to a tenant.

17.5. MANAGEMENT QUOTA OPTIONS

Here are additonal options available to administrators for managing quotas for tenants:

- **default_quota*** - The default number of resources available to a tenant.
- **quota_health_monitor*** - The number of health monitors available to a tenant.
  Health monitors do not consume resources, however the quota option is available because OpenStack Networking considers health monitors as resource consumers.
- **quota_member** - The number of pool members available to a tenant.
  Pool members do not consume resources, however the quota option is available because OpenStack Networking considers pool members as resource consumers.
- **quota_pool** - The number of pools available to a tenant.

CHAPTER 18. DEPLOYING ROUTED PROVIDER NETWORKS

18.1. ADVANTAGES OF ROUTED PROVIDER NETWORKS

In Red Hat OpenStack Platform, operators can create routed provider networks (RPNs). RPNs are typically used in edge deployments, and rely on multiple layer 2 network segments instead of traditional networks that have only one segment.

RPNs simplify the cloud for end users because they see only one network. For cloud operators, RPNs deliver scalabilty and fault tolerance. For example, if a major error occurs, only one segment is impacted instead of the entire network failing.

Before routed provider networks (RPNs), operators typically had to choose from one of the following architectures:

- A single, large layer 2 network
- Multiple, smaller layer 2 networks

Single, large layer 2 networks become complex when scaling and reduce fault tolerance (increase failure domains).

Multiple, smaller layer 2 networks scale better and shrink failure domains, but can introduce complexity for end users.

Starting with Red Hat OpenStack Platform 16.1.1 and later, you can deploy RPNs using the ML2/OVS or the SR-IOV mechanism drivers.

Additional resources

- Fundamentals of routed provider networks
- Limitations of routed provider networks
- Preparing for a routed provider network
- Creating a routed provider network
- Migrating a non-routed to a routed provider network

18.2. FUNDAMENTALS OF ROUTED PROVIDER NETWORKS

The association between a segment and a subnet distinguishes a routed provider network (RPN) from other types of networks. Each network segment requires at least one subnet that explicitly belongs to that segment. The OpenStack Networking service (neutron) enforces that either zero or all subnets on a particular network associate with a segment. For example, attempting to create a subnet without a segment on a network containing subnets with segments generates an error.

With RPNs, the IP addresses available to virtual machine (VM) instances depend on the segment of the network available on the particular compute node. The Networking service port can be associated with only one network segment.

Similar to conventional networking, layer 2 (switching) handles transit of traffic between ports on the same network segment and layer 3 (routing) handles transit of traffic between segments.

The Networking service does not provide layer 3 services between segments. Instead, it relies on physical network infrastructure to route subnets. Thus, both the Networking service and physical network infrastructure must contain configuration for routed provider networks, similar to conventional provider networks.

Because the Compute service (nova) scheduler is not network segment aware, when you deploy RPNs, you must map each leaf or rack segment or DCN edge site to a Compute service host-aggregate or availability zone.

If you require a DHCP-metadata service, you must define an availability zone for each edge site or network segment, to ensure that the local DHCP agent is deployed.

Additional resources

- Advantages of routed provider networks
- Limitations of routed provider networks
- Preparing for a routed provider network
- Creating a routed provider network
- Migrating a non-routed to a routed provider network

18.3. LIMITATIONS OF ROUTED PROVIDER NETWORKS

Routed provider networks (RPNs) are not supported by all mechanism drivers and there are restrictions with the Compute service scheduler and other software as noted in the following list:

- Routed provider networks are supported only by the ML2/OVS and SR-IOV mechanism drivers. Open Virtual Network (OVN) is not supported.
- North-south routing with central SNAT or a floating IP is not supported.
- When using SR-IOV or PCI pass-through, physical network (physnet) names must be the same in central and remote sites or segments. You cannot reuse segment IDs.
- The Compute service (nova) scheduler is not segment-aware. (You must map each segment or edge site to a Compute host-aggregate or availability zone.) Currently, there are only two VM instance boot options available:
  - Boot using **port-id** and no IP address, specifying Compute availability zone (segment or edge site).
  - Boot using **network-id**, specifying the Compute availability zone (segment or edge site).
- Cold or live migration works only when you specify the destination Compute availability zone (segment or edge site).

18.4. PREPARING FOR A ROUTED PROVIDER NETWORK

There are several tasks that you must perform before you can create a routed provider network in Red Hat OpenStack Platform.

Procedure

1. Within a network, use a unique physical network name for each segment. This enables reuse of the same segmentation details between subnets.
   For example, use the same VLAN ID across all segments of a particular provider network.

2. Implement routing between segments.
   Each subnet on a segment must contain the gateway address of the router interface on that particular subnet.

   Table 18.1. Sample segments for routing

Segment	Version	Addresses	Gateway
segment1	4	203.0.113.0/24	203.0.113.1
segment1	6	fd00:203:0:113::/64	fd00:203:0:113::1
segment2	4	198.51.100.0/24	198.51.100.1
segment2	6	fd00:198:51:100::/64	fd00:198:51:100::1

3. Map segments to compute nodes.
   Routed provider networks imply that compute nodes reside on different segments. Ensure that every compute host in a router provider network has direct connectivity to one of its segments.

   Table 18.2. Sample segment to Compute node mappings

Host	Rack	Physical network
compute0001	rack 1	segment 1
compute0002	rack 1	segment 1
...	...	...
compute0101	rack 2	segment 2
compute0102	rack 2	segment 2
compute0102	rack 2	segment 2
...	...	...

4. Deploy at least one DHCP agent per segment.
   Unlike conventional provider networks, a DHCP agent cannot support more than one segment within a network. Deploy DHCP agents on the compute nodes containing the segments rather than on the network nodes to reduce the node count.

   Table 18.3. Sample DCHP agent per segment mapping

Host	Rack	Physical network
network0001	rack 1	segment 1
network0002	rack 1	segment 1
...	...	...

You deploy a DCHP agent and a Networking service metadata agent on the Compute nodes by using a custom roles file.

Here is an example:

```
###############################################################################
####
# Role: ComputeSriov                                                          #
###############################################################################
####
- name: ComputeSriov
  description: |
    Compute SR-IOV Role
  CountDefault: 1
  networks:
    External:
      subnet: external_subnet
    InternalApi:
      subnet: internal_api_subnet
    Tenant:
      subnet: tenant_subnet
    Storage:
      subnet: storage_subnet
  RoleParametersDefault:
    TunedProfileName: "cpu-partitioning"
  update_serial: 25
  ServicesDefault:
    - OS::TripleO::Services::Aide
    - OS::TripleO::Services::AuditD
    - OS::TripleO::Services::BootParams
    - OS::TripleO::Services::CACerts
...
    - OS::TripleO::Services::NeutronDhcpAgent
    - OS::TripleO::Services::NeutronMetadataAgent
...
```

In a custom environment file, add the following key-value pair:

```
parameter_defaults:
    ....
    NeutronEnableIsolatedMetadata: 'True'
    ....
```

5. The OpenStackClient plug-in for the Placement service must be installed.

Install this plug-in as root by running this command:

```
# yum install python3-osc-placement
```

Additional resources

- Advantages of routed provider networks
- Fundamentals of routed provider networks
- Limitations of routed provider networks
- Creating a routed provider network
- Migrating a non-routed to a routed provider network

18.5. CREATING A ROUTED PROVIDER NETWORK

Routed provider networks (RPNs) simplify the cloud for end users because they see only one network. For cloud operators, RPNs deliver scalabilty and fault tolerance.

When you perform this procedure, you create a routed provider network with two network segments. Each segment contains one IPv4 subnet and one IPv6 subnet.

Prerequisites

- Complete the steps in Preparing for a routed provider network .

Procedure

1. Create a VLAN provider network that includes a default segment.
   In this example, the VLAN provider network is named **multisegment1** and uses a physical network called **provider1** and a VLAN whose ID is **2019**:

```
$ openstack network create --share --provider-physical-network provider1 \
  --provider-network-type vlan --provider-segment 2019 multisegment1
```

Example output

```
+---------------------------+--------------------------------------+
| Field                     | Value                                |
+---------------------------+--------------------------------------+
| admin_state_up            | UP                                   |
| id                        | 6ab19caa-dda9-4b3d-abc4-5b8f435b98d9 |
| ipv4_address_scope        | None                                 |
| ipv6_address_scope        | None                                 |
| l2_adjacency              | True                                 |
| mtu                       | 1500                                 |
| name                      | multisegment1                        |
| port_security_enabled     | True                                 |
| provider:network_type     | vlan                                 |
| provider:physical_network | provider1                            |
| provider:segmentation_id  | 2019                                 |
| revision_number           | 1                                    |
| router:external         | Internal                        |
| shared                  | True                            |
| status                  | ACTIVE                          |
| subnets                 |                                 |
| tags                    | []                              |
+-------------------------+---------------------------------+
```

2. Rename the default network segment to **segment1**.

    a. Obtain the segment ID:

    ```
    $ openstack network segment list --network multisegment1
    ```

    Example output

    ```
    +--------------------------------------+----------+--------------------------------------+--------------
    +---------+
    | ID                                   | Name     | Network                              | Network Type |
    Segment |
    +--------------------------------------+----------+--------------------------------------+--------------
    +---------+
    | 43e16869-ad31-48e4-87ce-acf756709e18 | None     | 6ab19caa-dda9-4b3d-abc4-
    5b8f435b98d9 | vlan         |    2019 |
    +--------------------------------------+----------+--------------------------------------+--------------
    +---------+
    ```

    b. Using the segment ID, rename the network segment to **segment1**:

    ```
    $ openstack network segment set --name segment1 43e16869-ad31-48e4-87ce-
    acf756709e18
    ```

3. Create a second segment on the provider network.
   In this example, the network segment uses a physical network called **provider2** and a VLAN whose ID is **2020**:

```
$ openstack network segment create --physical-network provider2 \
  --network-type vlan --segment 2020 --network multisegment1 segment2
```

Example output

```
+------------------+--------------------------------------+
| Field            | Value                                |
+------------------+--------------------------------------+
| description      | None                                 |
| headers          |                                      |
| id               | 053b7925-9a89-4489-9992-e164c8cc8763 |
| name             | segment2                             |
| network_id       | 6ab19caa-dda9-4b3d-abc4-5b8f435b98d9 |
| network_type     | vlan                                 |
| physical_network | provider2                            |
| revision_number  | 1                                    |
| segmentation_id  | 2020                                 |
| tags             | []                                   |
+------------------+--------------------------------------+
```

4. Verify that the network contains the **segment1** and **segment2** segments:

```
$ openstack network segment list --network multisegment1
```

Example output

```
+--------------------------------------+----------+--------------------------------------+--------------+-----
----+
| ID                                   | Name     | Network                              | Network Type | Segment |
+--------------------------------------+----------+--------------------------------------+--------------+-----
----+
| 053b7925-9a89-4489-9992-e164c8cc8763 | segment2 | 6ab19caa-dda9-4b3d-abc4-
5b8f435b98d9 | vlan         |    2020 |
| 43e16869-ad31-48e4-87ce-acf756709e18 | segment1 | 6ab19caa-dda9-4b3d-abc4-
5b8f435b98d9 | vlan         |    2019 |
+--------------------------------------+----------+--------------------------------------+--------------+-----
----+
```

5. Create one IPv4 subnet and one IPv6 subnet on the **segment1** segment.
In this example, the IPv4 subnet uses **203.0.113.0/24**:

```
$ openstack subnet create \
  --network multisegment1 --network-segment segment1 \
  --ip-version 4 --subnet-range 203.0.113.0/24 \
  multisegment1-segment1-v4
```

Example output

```
+-------------------+--------------------------------------+
| Field             | Value                                |
+-------------------+--------------------------------------+
| allocation_pools  | 203.0.113.2-203.0.113.254            |
| cidr              | 203.0.113.0/24                       |
| enable_dhcp       | True                                 |
| gateway_ip        | 203.0.113.1                          |
| id                | c428797a-6f8e-4cb1-b394-c404318a2762 |
| ip_version        | 4                                    |
| name              | multisegment1-segment1-v4            |
| network_id        | 6ab19caa-dda9-4b3d-abc4-5b8f435b98d9 |
| revision_number   | 1                                    |
| segment_id        | 43e16869-ad31-48e4-87ce-acf756709e18 |
| tags              | []                                   |
+-------------------+--------------------------------------+
```

In this example, the IPv6 subnet uses **fd00:203:0:113::/64**:

```
$ openstack subnet create \
  --network multisegment1 --network-segment segment1 \
  --ip-version 6 --subnet-range fd00:203:0:113::/64 \
  --ipv6-address-mode slaac multisegment1-segment1-v6
```

Example output

```
+-------------------+------------------------------------------------------+
| Field             | Value                                                |
+-------------------+------------------------------------------------------+
| allocation_pools  | fd00:203:0:113::2-fd00:203:0:113:ffff:ffff:ffff:ffff |
| cidr              | fd00:203:0:113::/64                                  |
| enable_dhcp       | True                                                 |
| gateway_ip        | fd00:203:0:113::1                                    |
| id                | e41cb069-9902-4c01-9e1c-268c8252256a                 |
| ip_version        | 6                                                    |
| ipv6_address_mode | slaac                                                |
| ipv6_ra_mode      | None                                                 |
| name              | multisegment1-segment1-v6                            |
| network_id        | 6ab19caa-dda9-4b3d-abc4-5b8f435b98d9                 |
| revision_number   | 1                                                    |
| segment_id        | 43e16869-ad31-48e4-87ce-acf756709e18                 |
| tags              | []                                                   |
+-------------------+------------------------------------------------------+
```

NOTE

By default, IPv6 subnets on provider networks rely on physical network infrastructure for stateless address autoconfiguration (SLAAC) and router advertisement.

6. Create one IPv4 subnet and one IPv6 subnet on the **segment2** segment.
   In this example, the IPv4 subnet uses **198.51.100.0/24**:

```
$ openstack subnet create \
  --network multisegment1 --network-segment segment2 \
  --ip-version 4 --subnet-range 198.51.100.0/24 \
  multisegment1-segment2-v4
```

Example output

```
+-------------------+--------------------------------------+
| Field             | Value                                |
+-------------------+--------------------------------------+
| allocation_pools  | 198.51.100.2-198.51.100.254          |
| cidr              | 198.51.100.0/24                      |
| enable_dhcp       | True                                 |
| gateway_ip        | 198.51.100.1                         |
| id                | 242755c2-f5fd-4e7d-bd7a-342ca95e50b2 |
| ip_version        | 4                                    |
| name              | multisegment1-segment2-v4            |
| network_id        | 6ab19caa-dda9-4b3d-abc4-5b8f435b98d9 |
| revision_number   | 1                                    |
| segment_id        | 053b7925-9a89-4489-9992-e164c8cc8763 |
| tags              | []                                   |
+-------------------+--------------------------------------+
```

In this example, the IPv6 subnet uses **fd00:198:51:100::/64**:

```
$ openstack subnet create \
  --network multisegment1 --network-segment segment2 \
  --ip-version 6 --subnet-range fd00:198:51:100::/64 \
  --ipv6-address-mode slaac multisegment1-segment2-v6
```

Example output

```
+-------------------+------------------------------------------------------+
| Field             | Value                                                |
+-------------------+------------------------------------------------------+
| allocation_pools  | fd00:198:51:100::2-fd00:198:51:100:ffff:ffff:ffff:ffff |
| cidr              | fd00:198:51:100::/64                                 |
| enable_dhcp       | True                                                 |
| gateway_ip        | fd00:198:51:100::1                                   |
| id                | b884c40e-9cfe-4d1b-a085-0a15488e9441                 |
| ip_version        | 6                                                    |
| ipv6_address_mode | slaac                                                |
| ipv6_ra_mode      | None                                                 |
| name              | multisegment1-segment2-v6                            |
| network_id        | 6ab19caa-dda9-4b3d-abc4-5b8f435b98d9                 |
| revision_number   | 1                                                    |
| segment_id        | 053b7925-9a89-4489-9992-e164c8cc8763                 |
| tags              | []                                                   |
+-------------------+------------------------------------------------------+
```

Verification steps

1. Verify that each IPv4 subnet associates with at least one DHCP agent:

```
$ openstack network agent list --agent-type dhcp --network multisegment1
```

Example output

```
+--------------------------------------+------------+-------------+-------------------+-------+-------+------
--------------+
| ID                                   | Agent Type | Host        | Availability Zone | Alive | State | Binary
|
+--------------------------------------+------------+-------------+-------------------+-------+-------+------
--------------+
| c904ed10-922c-4c1a-84fd-d928abaf8f55 | DHCP agent | compute0001 | nova              | :-)
| UP    | neutron-dhcp-agent |
| e0b22cc0-d2a6-4f1c-b17c-27558e20b454 | DHCP agent | compute0101 | nova              | :-)
| UP    | neutron-dhcp-agent |
+--------------------------------------+------------+-------------+-------------------+-------+-------+------
--------------+
```

2. Verify that inventories were created for each segment IPv4 subnet in the Compute service placement API.
Run this command for all segment IDs:

```
$ SEGMENT_ID=053b7925-9a89-4489-9992-e164c8cc8763
$ openstack resource provider inventory list $SEGMENT_ID
```

Example output

In this example, only one of the segments is shown:

```
+----------------+------------------+----------+----------+-----------+----------+-------+
| resource_class | allocation_ratio | max_unit | reserved | step_size | min_unit | total |
+----------------+------------------+----------+----------+-----------+----------+-------+
| IPV4_ADDRESS   |              1.0 |        1 |        2 |         1 |        1 |    30 |
+----------------+------------------+----------+----------+-----------+----------+-------+
```

3. Verify that host aggregates were created for each segment in the Compute service:

```
$ openstack aggregate list
```

Example output

In this example, only one of the segments is shown:

```
+----+---------------------------------------------------------+-------------------+
| Id | Name                                                    | Availability Zone |
+----+---------------------------------------------------------+-------------------+
| 10 | Neutron segment id 053b7925-9a89-4489-9992-e164c8cc8763 | None              |
+----+---------------------------------------------------------+-------------------+
```

4. Launch one or more instances. Each instance obtains IP addresses according to the segment it uses on the particular compute node.

NOTE

If a fixed IP is specified by the user in the port create request, that particular IP is allocated immediately to the port. However, creating a port and passing it to an instance yields a different behavior than conventional networks. If the fixed IP is not specified on the port create request, the Networking service defers assignment of IP addresses to the port until the particular compute node becomes apparent. For example, when you run this command:

```
$ openstack port create --network multisegment1 port1
```

Example output

```
+-----------------------+--------------------------------------+
| Field                 | Value                                |
+-----------------------+--------------------------------------+
| admin_state_up        | UP                                   |
| binding_vnic_type     | normal                               |
| id                    | 6181fb47-7a74-4add-9b6b-f9837c1c90c4 |
| ip_allocation         | deferred                             |
| mac_address           | fa:16:3e:34:de:9b                    |
| name                  | port1                                |
| network_id            | 6ab19caa-dda9-4b3d-abc4-5b8f435b98d9 |
| port_security_enabled | True                                 |
| revision_number       | 1                                    |
| security_groups       | e4fcef0d-e2c5-40c3-a385-9c33ac9289c5 |
| status                | DOWN                                 |
| tags                  | []                                   |
+-----------------------+--------------------------------------+
```

Additional resources

- Advantages of routed provider networks
- Fundamentals of routed provider networks
- Limitations of routed provider networks
- Preparing for a routed provider network
- Migrating a non-routed to a routed provider network

18.6. MIGRATING A NON-ROUTED NETWORK TO A ROUTED PROVIDER NETWORK

You migrate a non-routed network by associating its subnet with the ID of the network segment.

Prerequisites

- The non-routed network you are migrating must contain *only* one segment and *only* one subnet.

IMPORTANT

In non-routed provider networks that contain multiple subnets or network segments it is not possible to safely migrate to a routed provider network. In non-routed networks, addresses from the subnet allocation pools are assigned to ports without consideration of the network segment to which the port is bound.

Procedure

1. Obtain the ID of the current network segment on the network that is being migrated.
   Here is an example:

   ```
   $ openstack network segment list --network my_network
   ```

   Example output

   ```
   +--------------------------------------+------+--------------------------------------+--------------+---------
   +
   | ID                                   | Name | Network                              | Network Type | Segment |
   +--------------------------------------+------+--------------------------------------+--------------+---------
   +
   | 81e5453d-4c9f-43a5-8ddf-feaf3937e8c7 | None | 45e84575-2918-471c-95c0-
   018b961a2984 | flat         | None    |
   +--------------------------------------+------+--------------------------------------+--------------+---------
   +
   ```

2. Obtain the ID or name of the current subnet on the network that you are migrating.
   Here is an example:

   ```
   $ openstack network segment list --network my_network
   ```

   Example output

   ```
   +--------------------------------------+-----------+--------------------------------------+---------------+
   | ID                                   | Name      | Network                              | Subnet        |
   +--------------------------------------+-----------+--------------------------------------+---------------+
   | 71d931d2-0328-46ae-93bc-126caf794307 | my_subnet | 45e84575-2918-471c-95c0-
   018b961a2984 | 172.24.4.0/24 |
   +--------------------------------------+-----------+--------------------------------------+---------------+
   ```

3. Verify the current **segment_id** of the subnet has a value of **None**.
   Here is an example:

   ```
   $ openstack subnet show my_subnet --c segment_id
   ```

   Example output

   ```
   +------------+-------+
   | Field      | Value |
   +------------+-------+
   | segment_id | None  |
   +------------+-------+
   ```

4. Change the value of the subnet **segment_id** to the network segment ID.
   Here is an example:

   ```
   $ openstack subnet set --network-segment 81e5453d-4c9f-43a5-8ddf-feaf3937e8c7
   my_subnet
   ```

Verification steps

- Verify that the subnet is now associated with the desired network segment.
  Here is an example:

  ```
  $ openstack subnet show my_subnet --c segment_id
  ```

  ### Example output

  ```
  +------------+--------------------------------------+
  | Field      | Value                                |
  +------------+--------------------------------------+
  | segment_id | 81e5453d-4c9f-43a5-8ddf-feaf3937e8c7 |
  +------------+--------------------------------------+
  ```

Additional resources

- Advantages of routed provider networks
- Fundamentals of routed provider networks
- Limitations of routed provider networks
- Preparing for a routed provider network
- Creating a routed provider network

CHAPTER 19. CONFIGURING ALLOWED ADDRESS PAIRS

19.1. OVERVIEW OF ALLOWED ADDRESS PAIRS

An *allowed address pair* is when you identify a specific MAC address, IP address, or both to allow network traffic to pass through a port regardless of the subnet. When you define allowed address pairs, you are able to use protocols like VRRP (Virtual Router Redundancy Protocol) that float an IP address between two VM instances to enable fast data plane failover.

You define allowed address pairs using the Red Hat OpenStack Platform command-line client **openstack port** command.

NOTE

The allowed-address pairs extension is currently supported only by the ML2 and Open vSwitch plug-ins.

Be aware that you should not use the default security group with a wider IP address range in an allowed address pair. Doing so can allow a single port to bypass security groups for all other ports within the same network.

For example, this command impacts all ports in the network and bypasses all security groups:

```
# openstack port set 9e67d44eab334f07bf82fa1b17d824b6 --allowed-address
mac_address=3e:37:09:4b, ip_address=0.0.0.0/0
```

Additional resources

- port command in the *Command Line Interface Reference*
- Creating a port and allowing one address pair
- Adding allowed address pairs

19.2. CREATING A PORT AND ALLOWING ONE ADDRESS PAIR

Creating a port with an allowed address pair enables network traffic to flow through the port regardless of the subnet.

Prerequisites

- You are using an ML2/OVS plug-in.

Procedure

- Use the following command to create a port and allow one address pair:

  ```
  # openstack port create <port-name> --network <network> --allowed-address mac_address=
  <mac-address>,ip_address=<ip-cidr>
  ```

IMPORTANT

Do not use the default security group with a wider IP address range in an allowed address pair. Doing so can allow a single port to bypass security groups for all other ports within the same network.

Additional resources

- port command in the *Command Line Interface Reference*
- Overview of allowed address pairs
- Adding allowed address pairs

19.3. ADDING ALLOWED ADDRESS PAIRS

You can add an allowed address pair to a port to enable network traffic to flow through the port regardless of the subnet.

Prerequisites

- You are using an ML2/OVS plug-in.

Procedure

- Use the following command to add allowed address pairs:

```
# openstack port set <port-uuid> --allowed-address mac_address=
<mac_address>,ip_address=<ip_cidr>
```

NOTE

You cannot set an allowed-address pair that matches the **mac_address** and **ip_address** of a port. This is because such a setting has no effect since traffic matching the **mac_address** and **ip_address** is already allowed to pass through the port.

IMPORTANT

Do not use the default security group with a wider IP address range in an allowed address pair. Doing so can allow a single port to bypass security groups for all other ports within the same network.

Additional resources

- port command in the *Command Line Interface Reference*
- Overview of allowed address pairs
- Creating a port and allowing one address pair

CHAPTER 20. CONFIGURING LAYER 3 HIGH AVAILABILITY (HA)

20.1. OPENSTACK NETWORKING WITHOUT HIGH AVAILABILITY (HA)

OpenStack Networking deployments without any high availability (HA) features are vulnerable to physical node failures.

In a typical deployment, tenants create virtual routers, which are scheduled to run on physical L3 agent nodes. This becomes an issue when you lose a L3 agent node and the dependent virtual machines subsequently lose connectivity to external networks. Any floating IP addresses will also be unavailable. In addition, connectivity is lost between any networks that the router hosts.

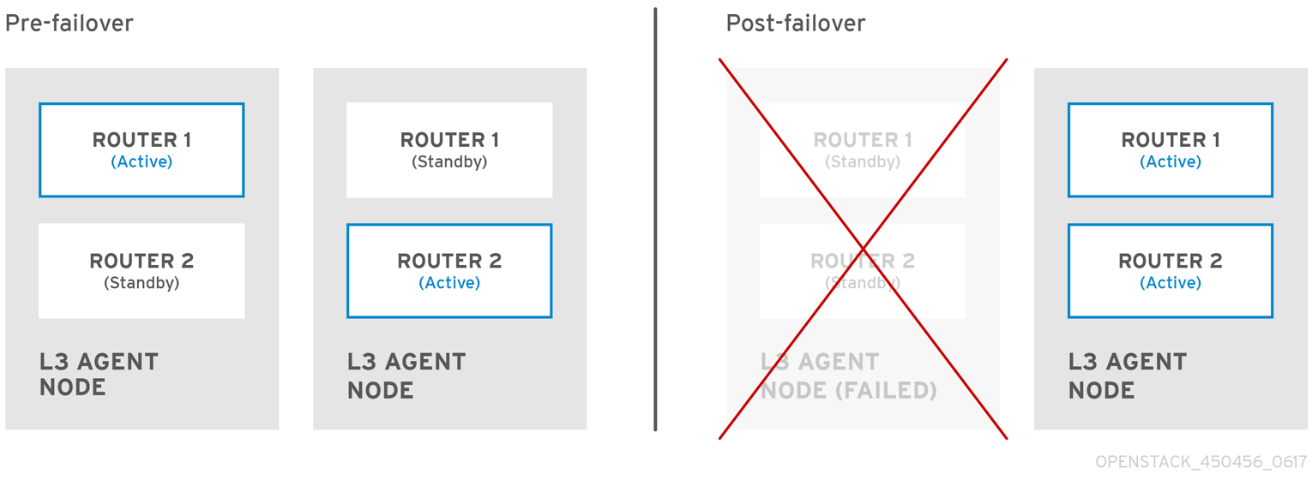
20.2. OVERVIEW OF LAYER 3 HIGH AVAILABILITY (HA)

This active/passive high availability (HA) configuration uses the industry standard VRRP (as defined in RFC 3768) to protect tenant routers and floating IP addresses. A virtual router is randomly scheduled across multiple OpenStack Networking nodes, with one designated as the *active* router, and the remainder serving in a *standby* role.

NOTE

To deploy Layer 3 HA, you must maintain similar configuration on the redundant OpenStack Networking nodes, including floating IP ranges and access to external networks.

In the following diagram, the active Router1 and Router2 routers are running on separate physical L3 agent nodes. Layer 3 HA has scheduled backup virtual routers on the corresponding nodes, ready to resume service in the case of a physical node failure. When the L3 agent node fails, Layer 3 HA reschedules the affected virtual router and floating IP addresses to a working node:



During a failover event, instance TCP sessions through floating IPs remain unaffected, and migrate to the new L3 node without disruption. Only SNAT traffic is affected by failover events.

The L3 agent is further protected when in an active/active HA mode.

20.3. LAYER 3 HIGH AVAILABILITY (HA) FAILOVER CONDITIONS

Layer 3 high availability (HA) automatically reschedules protected resources in the following events:

- The L3 agent node shuts down or otherwise loses power because of a hardware failure.
- The L3 agent node becomes isolated from the physical network and loses connectivity.

NOTE

Manually stopping the L3 agent service does not induce a failover event.

20.4. TENANT CONSIDERATIONS FOR LAYER 3 HIGH AVAILABILITY (HA)

Layer 3 high availability (HA) configuration occurs in the back end and is invisible to the tenant. Tenants can continue to create and manage their virtual routers as usual, however there are some limitations to be aware of when designing your Layer 3 HA implementation:

- Layer 3 HA supports up to 255 virtual routers per tenant.
- Internal VRRP messages are transported within a separate internal network, created automatically for each project. This process occurs transparently to the user.

20.5. HIGH AVAILABILITY (HA) CHANGES TO OPENSTACK NETWORKING

The Neutron API has been updated to allow administrators to set the **--ha=True/False** flag when creating a router, which overrides the default configuration of **l3_ha** in /var/lib/config-data/neutron/etc/neutron/neutron.conf.

- **HA changes to neutron-server:**
  - Layer 3 HA assigns the active role randomly, regardless of the scheduler used by OpenStack Networking (whether random or leastrouter).
  - The database schema has been modified to handle allocation of virtual IP addresses (VIPs) to virtual routers.
  - A transport network is created to direct Layer 3 HA traffic.
- **High availability (HA) changes to L3 agent:**
  - A new keepalived manager has been added, providing load-balancing and HA capabilities.
  - IP addresses are converted to VIPs.

20.6. ENABLING LAYER 3 HIGH AVAILABILITY (HA) ON OPENSTACK NETWORKING NODES

Complete the following steps to enable Layer 3 high availability (HA) on OpenStack Networking and L3 agent nodes.

1. Configure Layer 3 HA in the /var/lib/config-data/neutron/etc/neutron/neutron.conf file by enabling L3 HA and defining the number of L3 agent nodes that you want to protect each virtual router:

```
l3_ha = True
max_l3_agents_per_router = 2
min_l3_agents_per_router = 2
```

L3 HA parameters:

- **l3_ha** - When set to True, all virtual routers created from this point onwards default to HA (and not legacy) routers. Administrators can override the value for each router using the following option in the **openstack router create** command:

  ```
  # openstack router create --ha
  ```

  or

  ```
  # openstack router create --no-ha
  ```

- **max_l3_agents_per_router** - Set this to a value between the minimum and total number of network nodes in your deployment.
  For example, if you deploy four OpenStack Networking nodes but set this parameter to 2, only two L3 agents protect each HA virtual router: one active, and one standby. In addition, each time a new L3 agent node is deployed, additional standby versions of the virtual routers are scheduled until the **max_l3_agents_per_router** limit is reached. As a result, you can scale out the number of standby routers by adding new L3 agents.

  In addition, each time a new L3 agent node is deployed, additional standby versions of the virtual routers are scheduled until the **max_l3_agents_per_router** limit is reached. As a result, you can scale out the number of standby routers by adding new L3 agents.

- **min_l3_agents_per_router** - The minimum setting ensures that the HA rules remain enforced. This setting is validated during the virtual router creation process to ensure a sufficient number of L3 Agent nodes are available to provide HA.
  For example, if you have two network nodes and one becomes unavailable, no new routers can be created during that time, as you need at least min active L3 agents when creating a HA router.

2. Restart the **neutron-server** service to apply the changes:

   ```
   # systemctl restart neutron-server.service
   ```

20.7. REVIEWING HIGH AVAILABILITY (HA) NODE CONFIGURATIONS

- Run the **ip address** command within the virtual router namespace to return a HA device in the result, prefixed with *ha-*.

  ```
  # ip netns exec qrouter-b30064f9-414e-4c98-ab42-646197c74020 ip address
  <snip>
  2794: ha-45249562-ec: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
  noqueue state DOWN group default
  link/ether 12:34:56:78:2b:5d brd ff:ff:ff:ff:ff:ff
  inet 169.254.0.2/24 brd 169.254.0.255 scope global ha-54b92d86-4f
  ```

With Layer 3 HA enabled, virtual routers and floating IP addresses are protected against individual node failure.

CHAPTER 21. IDENTIFYING VIRTUAL DEVICES WITH TAGS

21.1. OVERVIEW OF VIRTUAL DEVICE TAGGING

If you launch an instance with multiple network interfaces or block devices, you can use device tagging to communicate the intended role of each device to the instance operating system. Tags are assigned to devices at instance boot time, and are available to the instance operating system through the metadata API and the configuration drive (if enabled).

The tags are set using the following parameters:

- **--block-device tag=device metadata**
- **--nic tag=device metadata**

21.2. TAGGING VIRTUAL DEVICES

- To tag virtual devices, use the tag parameters, **--block-device** and **--nic**, when creating instances.
  Here is an example:

  ```
  $ nova boot test-vm --flavor m1.tiny --image cirros \
  --nic net-id=55411ca3-83dd-4036-9158-bf4a6b8fb5ce,tag=nfv1 \
  --block-device id=b8c9bef7-aa1d-4bf4-a14d-17674b370e13,bus=virtio,tag=database-server
  NFVappServer
  ```

  The resulting tags are added to the existing instance metadata and are available through both the metadata API, and on the configuration drive.

  In this example, the following devices section populates the metadata:

  Sample contents of the *meta_data.json* file:

  ```
     {
    "devices": [
     {
         "type": "nic",
         "bus": "pci",
         "address": "0030:00:02.0",
         "mac": "aa:00:00:00:01",
         "tags": ["nfv1"]
     },
     {
         "type": "disk",
         "bus": "pci",
         "address": "0030:00:07.0",
         "serial": "disk-vol-227",
         "tags": ["database-server"]
     }
    ]
  }
  ```

  The device tag metadata is available using **GET /openstack/latest/meta_data.json** from the metadata API.

If the configuration drive is enabled, and mounted under */configdrive* in the instance operating system, the metadata is also present in */configdrive/openstack/latest/meta_data.json*.