Red Hat OpenStack Services on OpenShift 18.0

Configuring networking services

Configuring the Networking service (neutron) for managing networking traffic in a Red Hat OpenStack Services on OpenShift environment

Last Updated: 2024-08-28

Red Hat OpenStack Services on OpenShift 18.0 Configuring networking services

Configuring the Networking service (neutron) for managing networking traffic in a Red Hat OpenStack Services on OpenShift environment

OpenStack Team
rhos-docs@redhat.com

Legal Notice

Abstract

Configure your Networking service (neutron) in a Red Hat OpenStack Services on OpenShift environment.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your input on our documentation. Tell us how we can make it better.

Providing documentation feedback in Jira

Use the Create Issue form to provide feedback on the documentation for Red Hat OpenStack Services on OpenShift (RHOSO) or earlier releases of Red Hat OpenStack Platform (RHOSP). When you create an issue for RHOSO or RHOSP documents, the issue is recorded in the RHOSO Jira project, where you can track the progress of your feedback.

To complete the Create Issue form, ensure that you are logged in to Jira. If you do not have a Red Hat Jira account, you can create an account at https://issues.redhat.com.

1. Click the following link to open a **Create Issue** page: Create Issue
2. Complete the **Summary** and **Description** fields. In the **Description** field, include the documentation URL, chapter or section number, and a detailed description of the issue. Do not modify any other fields in the form.
3. Click **Create**.

CHAPTER 1. INTRODUCTION TO OPENSTACK NETWORKING

The Networking service (neutron) is the software-defined networking (SDN) component of Red Hat OpenStack Services on OpenShift (RHOSO). The RHOSO Networking service manages internal and external traffic to and from virtual machine instances and provides core services such as routing, segmentation, DHCP, and metadata. It provides the API for virtual networking capabilities and management of switches, routers, ports, and firewalls.

1.1. MANAGING YOUR RHOSO NETWORKS

With the Red Hat OpenStack Services on OpenShift (RHOSO) Networking service (neutron) you can effectively meet your site's networking goals. You can do the following tasks:

- **Provide connectivity to VM instances within a project.**
  Project networks primarily enable general (non-privileged) projects to manage networks without involving administrators. These networks are entirely virtual and require virtual routers to interact with other project networks and external networks such as the Internet. Project networks also usually provide DHCP and metadata services to VM (virtual machine) instances. RHOSO supports the following project network types: flat, VLAN, and GENEVE.

  For more information, see Managing project networks.

- **Secure your network at the port level.**
  Security groups provide a container for virtual firewall rules that control ingress (inbound to instances) and egress (outbound from instances) network traffic at the port level. Security groups use a default deny policy and only contain rules that allow specific traffic. Each port can reference one or more security groups in an additive fashion. ML2/OVN uses the Open vSwitch firewall driver to translate security group rules to a configuration.

  By default, security groups are stateful. In ML2/OVN deployments, you can also create stateless security groups. A stateless security group can provide significant performance benefits. Unlike stateful security groups, stateless security groups do not automatically allow returning traffic, so you must create a complimentary security group rule to allow the return of related traffic.

  For more information, see Configuring shared security groups.

- **Set ingress and egress limits for traffic on VM instances.**
  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.

  For more information, see Configuring Quality of Service (QoS) policies.

- **Optimize your VM instances for Network Functions Virtualization (NFV).**
  Instances can send and receive VLAN-tagged traffic over a single virtual NIC. This is particularly useful for NFV applications (VNFs) that expect VLAN-tagged traffic, allowing a single virtual NIC to serve multiple customers or services.

  In a VLAN transparent network, you set up VLAN tagging in the VM instances. The VLAN tags are transferred over the network and consumed by the VM instances on the same VLAN, and ignored by other instances and devices. VLAN trunks support VLAN-aware instances by combining VLANs into a single trunked port.

  For more information, see VLAN-aware instances.

- **Control which projects can attach instances to a shared network.**
  Using role-based access control (RBAC) policies in the RHOSO Networking service, cloud administrators can remove the ability for some projects to create networks and can instead allow them to attach to pre-existing networks that correspond to their project.

  For more information, see Configuring RBAC policies.

1.2. NETWORKING SERVICE COMPONENTS

The Red Hat OpenStack Services on OpenShift (RHOSO) Networking service (neutron) includes the following components:

- API server
  The RHOSO networking API includes support for Layer 2 networking and IP Address Management (IPAM), as well as an extension for a Layer 3 router construct that enables routing between Layer 2 networks and gateways to external networks. RHOSO networking includes a growing list of plug-ins that enable interoperability with various commercial and open source network technologies, including routers, switches, virtual switches and software-defined networking (SDN) controllers.

- Modular Layer 2 (ML2) plug-in and agents
  ML2 plugs and unplugs ports, creates networks or subnets, and provides IP addressing.

- Messaging queue
  Accepts and routes RPC requests between RHOSO services to complete API operations.

1.3. MODULAR LAYER 2 (ML2) NETWORKING

Modular Layer 2 (ML2) is the Red Hat OpenStack Services on OpenShift (RHOSO) networking core plug-in. The ML2 modular design enables the concurrent operation of mixed network technologies through mechanism drivers. Open Virtual Network (OVN) is the default mechanism driver used with ML2.

The ML2 framework distinguishes between the two kinds of drivers that can be configured:

Type drivers

Define how an RHOSO network is technically realized.
Each available network type is managed by an ML2 type driver, and they maintain any required type-specific network state. Validating the type-specific information for provider networks, type drivers are responsible for the allocation of a free segment in project networks. Examples of type drivers are GENEVE, VLAN, and flat networks.

Mechanism drivers

Define the mechanism to access an RHOSO network of a certain type.
The mechanism driver takes the information established by the type driver and applies it to the networking mechanisms that have been enabled. RHOSO uses the OVN mechanism driver.

Mechanism drivers can employ L2 agents, and by using RPC interact directly with external devices or controllers. You can use multiple mechanism and type drivers simultaneously to access different ports of the same virtual network.

1.4. ML2 NETWORK TYPES

You can operate multiple network segments at the same time. ML2 supports the use and interconnection of multiple network segments. You don't have to bind a port to a network segment because ML2 binds ports to segments with connectivity. Depending on the mechanism driver, ML2 supports the following network segment types:

- Flat
- VLAN
- GENEVE tunnels

Flat

All virtual machine (VM) instances reside on the same network, which can also be shared with the hosts. No VLAN tagging or other network segregation occurs.

VLAN

With RHOSO networking users can create multiple provider or project 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.
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.

GENEVE tunnels

Generic Network Virtualization Encapsulation (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.

1.5. EXTENSION DRIVERS FOR THE RHOSO NETWORKING SERVICE

The Red Hat OpenStack Services on OpenShift (RHOSO) Networking service (neutron) is extensible. Extensions serve two purposes: they allow the introduction of new features in the API without requiring a version change and they allow the introduction of vendor specific niche functionality. Applications can programmatically list available extensions by performing a GET on the /**extensions** URI. Note that this is a versioned request; that is, an extension available in one API version might not be available in another.

The ML2 plug-in also supports extension drivers that allows other pluggable drivers to extend the core resources implemented in the ML2 plug-in for network objects. Examples of extension drivers include support for QoS, port security, and so on.

CHAPTER 2. WORKING WITH ML2/OVN

Red Hat OpenStack Services on OpenShift (RHOSO) networks are managed by the Networking service (neutron). The core of the Networking service is the Modular Layer 2 (ML2) plug-in, and the default mechanism driver for RHOSO ML2 plug-in is the Open Virtual Networking (OVN) mechanism driver.

2.1. OPEN VIRTUAL NETWORK (OVN)

Open Virtual Network (OVN), is a system to support logical network abstraction in virtual machine and container environments. OVN is used as the mechanism driver for the Red Hat OpenStack Services on OpenShift (RHOSO) Networking service (neutron).

Sometimes called open source virtual networking for Open vSwitch (OVS), OVN complements the existing capabilities of OVS to add native support for logical network abstractions, such as logical L2 and L3 overlays, security groups and services such as DHCP.

A physical network comprises physical wires, switches, and routers. A virtual network extends a physical network into a hypervisor or container platform, bridging VMs or containers into the physical network. An OVN logical network is a network implemented in software that is insulated from physical networks by tunnels or other encapsulations. This allows IP and other address spaces used in logical networks to overlap with those used on physical networks without causing conflicts. Logical network topologies can be arranged without regard for the topologies of the physical networks on which they run. Thus, VMs that are part of a logical network can migrate from one physical machine to another without network disruption.

The encapsulation layer prevents VMs and containers connected to a logical network from communicating with nodes on physical networks. For clustering VMs and containers, this can be acceptable or even desirable, but in many cases VMs and containers do need connectivity to physical networks. OVN provides multiple forms of gateways for this purpose.

An OVN deployment consists of several components:

Cloud Management System (CMS)

integrates OVN into a physical network by managing the OVN logical network elements and connecting the OVN logical network infrastructure to physical network elements. Some examples include OpenStack and OpenShift.

OVN databases

stores data representing the OVN logical and physical networks.

Hypervisors

run Open vSwitch and translate the OVN logical network into OpenFlow on a physical or virtual machine.

Gateways

extends a tunnel-based OVN logical network into a physical network by forwarding packets between tunnels and the physical network infrastructure.

2.2. LIST OF COMPONENTS IN THE RHOSO OVN ARCHITECTURE

Open Virtual Network (OVN) provides networking services for Red Hat OpenStack Services on OpenShift (RHOSO) environments. The OVN architecture consists of the following components and services:

Networking service

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 project network, router, and load balancer details, among 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.

ML2 plug-in with OVN mechanism driver

The ML2 plug-in translates the OpenStack-specific networking configuration into the platform-neutral OVN logical networking configuration. It typically runs on the RHOSO control plane on OpenShift worker nodes.

OVN northbound (NB) database (ovn-nb)

This database stores the logical OVN networking configuration from the OVN ML2 plugin. It typically runs on the RHOSO control plane and listens on TCP port **6641**.
The northbound database (**OVN_Northbound**) serves as the interface between OVN and a cloud management system such as RHOSO. RHOSO produces the contents of the northbound database.

The northbound database contains the current desired state of the network, presented as a collection of logical ports, logical switches, logical routers, and more. Every RHOSO Networking service (neutron) object is represented in a table in the northbound database.

OVN northbound service (ovn-northd)

This service converts the logical networking configuration from the OVN NB database to the logical data path flows and populates these on the OVN Southbound database. It typically runs on the RHOSO control plane.

OVN southbound (SB) database (ovn-sb)

This database stores the converted logical data path flows. It typically runs on the RHOSO control plane and listens on TCP port **6642**.
The southbound database (**OVN_Southbound**) holds the logical and physical configuration state for OVN system to support virtual network abstraction. The **ovn-controller** uses the information in this database to configure OVS to satisfy Networking service (neutron) requirements.

NOTE

The schema file for the NB database is located in **/usr/share/ovn/ovn-nb.ovsschema**, and the SB database schema file is in **/usr/share/ovn/ovn-sb.ovsschema**.

OVS database server (OVSDB)

Hosts the OVN Northbound and Southbound databases. Also interacts with **ovs-vswitchd** to host the OVS database **conf.db**.

OVN controller (ovn-controller)

This controller connects to the OVN SB database and acts as the open vSwitch controller to control and monitor network traffic. It runs on all Compute and gateway nodes.

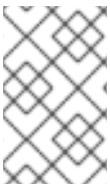
OVN metadata agent (ovn-metadata-agent)

This agent creates the **haproxy** instances for managing the OVS interfaces, network namespaces and HAProxy processes used to proxy metadata API requests. The agent runs on all Compute and gateway nodes.
The OVN Networking service creates a unique network namespace for each virtual network that enables the metadata service. Each network accessed by the instances on the Compute node has a corresponding metadata namespace (ovnmeta-<network_uuid>).

OpenStack guest instances access the Networking metadata service available at the link-local IP address: 169.254.169.254. The **neutron-ovn-metadata-agent** has access to the host networks where the Compute metadata API exists. Each HAProxy is in a network namespace that is not able to reach the appropriate host network. HaProxy adds the necessary headers to the metadata API request and then forwards the request to the **neutron-ovn-metadata-agent** over a UNIX domain socket.

2.3. LAYER 3 HIGH AVAILABILITY WITH OVN

OVN supports Layer 3 high availability (L3 HA) without any special configuration in Red Hat OpenStack Services on OpenShift (RHOSO) environments,



NOTE

When you create a router, do not use **--ha** option because OVN routers are highly available by default. **Openstack router create** commands that include the **--ha** option fail.

OVN automatically schedules the router port to all available gateway nodes that can act as an L3 gateway on the specified external network. OVN L3 HA uses the **gateway_chassis** column in the OVN **Logical_Router_Port** table. Most functionality is managed by OpenFlow rules with bundled active_passive outputs. The **ovn-controller** handles the Address Resolution Protocol (ARP) responder and router enablement and disablement. Gratuitous ARPs for FIPs and router external addresses are also periodically sent by the **ovn-controller**.

NOTE

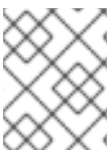
L3HA uses OVN to balance the routers back to the original gateway nodes to avoid any nodes becoming a bottleneck.

BFD monitoring

OVN uses the Bidirectional Forwarding Detection (BFD) protocol to monitor the availability of the gateway nodes. This protocol is encapsulated on top of the GENEVE tunnels established from node to node.

Each gateway node monitors all the other gateway nodes in a star topology in the deployment. Gateway nodes also monitor the compute nodes to let the gateways enable and disable routing of packets and ARP responses and announcements.

Each compute node uses BFD to monitor each gateway node and automatically steers external traffic, such as source and destination Network Address Translation (SNAT and DNAT), through the active gateway node for a given router. Compute nodes do not need to monitor other compute nodes.



NOTE

External network failures are not detected as would happen with an ML2-OVS configuration.

L3 HA for OVN supports the following failure modes:

- The gateway node becomes disconnected from the network (tunneling interface).
- **ovs-vswitchd** stops (**ovs-switchd** is responsible for BFD signaling)

- **ovn-controller** stops (**ovn-controller** removes itself as a registered node).

NOTE

This BFD monitoring mechanism only works for link failures, not for routing failures.

2.4. ACTIVE-ACTIVE CLUSTERED DATABASE SERVICE MODEL

On Red Hat OpenStack Services on OpenShift (RHOSO) environments, OVN uses a clustered database service model that applies the Raft consensus algorithm to enhance performance of OVS database protocol traffic and provide faster, more reliable failover handling.

A clustered database operates on a cluster of at least three database servers on different hosts. Servers use the Raft consensus algorithm to synchronize writes and share network traffic continuously across the cluster. The cluster elects one server as the leader. All servers in the cluster can handle database read operations, which mitigates potential bottlenecks on the control plane. Write operations are handled by the cluster leader.

If a server fails, a new cluster leader is elected and the traffic is redistributed among the remaining operational servers. The clustered database service model handles failovers more efficiently than the pacemaker-based model did. This mitigates related downtime and complications that can occur with longer failover times.

The leader election process requires a majority, so the fault tolerance capacity is limited by the highest odd number in the cluster. For example, a three-server cluster continues to operate if one server fails. A five-server cluster tolerates up to two failures. Increasing the number of servers to an even number does not increase fault tolerance. For example, a four-server cluster cannot tolerate more failures than a three-server cluster.

Most RHOSO deployments use three servers.

Clusters larger than five servers also work, with every two added servers allowing the cluster to tolerate an additional failure, but write performance decreases.

2.5. SR-IOV WITH ML2/OVN AND NATIVE OVN DHCP

You can deploy a custom node set to use SR-IOV in an ML2/OVN deployment with native OVN DHCP in Red Hat OpenStack Services on OpenShift (RHOSO) environments.

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.
- North/south routing on VF(direct) ports on VLAN tenant networks does not work with SR-IOV because the external ports are not colocated with the logical router's gateway ports. See https://bugs.launchpad.net/neutron/+bug/1875852.

CHAPTER 3. CUSTOMIZING DATA PLANE NETWORKS

In a Red Hat OpenStack Services on OpenShift (RHOSO) environment, the network configuration applied by default to the data plane nodes is the single NIC VLANs configuration. However, you can modify the network configuration that the OpenStack Operator applies.

3.1. APPLYING CUSTOM NETWORK CONFIGURATION TO A NODE SET

You can customize the network configuration for each data plane node set in your Red Hat OpenStack Services on OpenShift (RHOSO) environment.

Prerequisites

- You have the **oc** command line tool installed on your workstation.
- You are logged on to a workstation that has access to the RHOSO control plane as a user with **cluster-admin** privileges.

Procedure

1. Open the **OpenStackDataPlaneNodeSet** CR definition file for the node set you want to update, for example, **my_data_plane_node_set.yaml**.
2. Add the required network configuration or modify the existing configuration. Place the configuration in the **edpm_network_config_template** under **ansibleVars**:

```
apiVersion: dataplane.openstack.org/v1beta1
kind: OpenStackDataPlaneNodeSet
metadata:
  name: my-data-plane-node-set
spec:
  ...
  nodeTemplate:
    ...
    ansible:
      ansibleVars:
        edpm_network_config_template: |
          ---
          Network configuration options here
          ...
```

   When modifying your network configuration, refer to Section 3.2, “Network interface configuration options”.
3. Save the **OpenStackDataPlaneNodeSet** CR definition file.
4. Apply the updated **OpenStackDataPlaneNodeSet** CR configuration:

```
$ oc apply -f my_data_plane_node_set.yaml
```

5. Verify that the data plane resource has been updated:

```
$ oc get openstackdataplanenodeset
```

Sample output

```
NAME                    STATUS MESSAGE
my-data-plane-node-set   False  Deployment not started
```

6. Create a file on your workstation to define the **OpenStackDataPlaneDeployment** CR, for example, **my_data_plane_deploy.yaml**:

```
apiVersion: dataplane.openstack.org/v1beta1
kind: OpenStackDataPlaneDeployment
metadata:
  name: my-data-plane-deploy
```

TIP

Give the definition file and the **OpenStackDataPlaneDeployment** CR a unique and descriptive name that indicates the purpose of the modified node set.

7. Add the **OpenStackDataPlaneNodeSet** CR that you modified:

```
spec:
  nodeSets:
    - my-data-plane-node-set
```

8. Save the **OpenStackDataPlaneDeployment** CR deployment file.

9. Deploy the modified **OpenStackDataPlaneNodeSet** CR:

```
$ oc create -f my_data_plane_deploy.yaml -n openstack
```

You can view the Ansible logs while the deployment executes:

```
$ oc get pod -l app=openstackansibleee -n openstack -w

$ oc logs -l app=openstackansibleee -n openstack -f \
--max-log-requests 10
```

10. Verify that the modified **OpenStackDataPlaneNodeSet** CR is deployed:

Example

```
$ oc get openstackdataplanedeployment -n openstack
```

Sample output

```
NAME                    STATUS   MESSAGE
my-data-plane-node-set   True     Setup Complete
```

11. Repeat the **oc get** command until you see the **NodeSet Ready** message:

Example

```
$ oc get openstackdataplanenodeset -n openstack
```

Sample output

```
NAME                  STATUS   MESSAGE
my-data-plane-node-set   True     NodeSet Ready
```

For information on the meaning of the returned status, see Data plane conditions and states in the *Deploying Red Hat OpenStack Services on OpenShift* guide.

Additional resources

- Section 3.2, “Network interface configuration options”
- Section 3.3, “Example custom network interfaces”

3.2. NETWORK INTERFACE CONFIGURATION OPTIONS

Use the following tables to understand the available options for configuring network interfaces for Red Hat OpenStack Services on OpenShift (RHOSO) environments.

- interface
- vlan
- ovs_bridge
- Network interface bonding
- ovs_bond
- LACP with OVS bonding modes
- linux_bond
- routes

3.2.1. interface

Defines a single network interface. The network interface **name** uses either the actual interface name (**eth0**, **eth1**, **enp0s25**) or a set of numbered interfaces (**nic1**, **nic2**, **nic3**). The network interfaces of hosts within a role do not have to be exactly the same when you use numbered interfaces such as **nic1** and **nic2**, instead of named interfaces such as **eth0** and **eno2**. For example, one host might have interfaces **em1** and **em2**, while another has **eno1** and **eno2**, but you can refer to the NICs of both hosts as **nic1** and **nic2**.

The order of numbered interfaces corresponds to the order of named network interface types:

- **ethX** interfaces, such as **eth0**, **eth1**, and so on.
  Names appear in this format when consistent device naming is turned off in **udev**.
- **enoX** and **emX** interfaces, such as **eno0**, **eno1**, **em0**, **em1**, and so on.
  These are usually on-board interfaces.

- **enX** and any other interfaces, sorted alpha numerically, such as **enp3s0**, **enp3s1**, **ens3**, and so on.
  These are usually add-on interfaces.

The numbered NIC scheme includes only live interfaces, for example, if the interfaces have a cable attached to the switch. If you have some hosts with four interfaces and some with six interfaces, use **nic1** to **nic4** and attach only four cables on each host.

Table 3.1. **interface** options

Option	Default	Description
name		Name of the interface. The network interface **name** uses either the actual interface name (**eth0**, **eth1**, **enp0s25**) or a set of numbered interfaces (**nic1**, **nic2**, **nic3**).
use_dhcp	False	Use DHCP to get an IP address.
use_dhcpv6	False	Use DHCP to get a v6 IP address.
addresses		A list of IP addresses assigned to the interface.
routes		A list of routes assigned to the interface. For more information, see Section 3.2.7, “routes”.
mtu	1500	The maximum transmission unit (MTU) of the connection.
primary	False	Defines the interface as the primary interface. Required only when the **interface** is a member of a bond.
persist_mapping	False	Write the device alias configuration instead of the system names.
dhclient_args	None	Arguments that you want to pass to the DHCP client.
dns_servers	None	List of DNS servers that you want to use for the interface.
ethtool_opts		Set this option to **"rx-flow-hash udp4 sdfn"** to improve throughput when you use VXLAN on certain NICs.

Example

```
...
      edpm_network_config_template: |
        ---
        {% set mtu_list = [ctlplane_mtu] %}
        {% for network in nodeset_networks %}
        {{ mtu_list.append(lookup('vars', networks_lower[network] ~ '_mtu')) }}
        {%- endfor %}
        {% set min_viable_mtu = mtu_list | max %}
        network_config:
        - type: interface
          name: nic2
          ...
```

3.2.2. vlan

Defines a VLAN. Use the VLAN ID and subnet passed from the **parameters** section.

Table 3.2. vlan options

Option	Default	Description
vlan_id		The VLAN ID.
device		The parent device to attach the VLAN. Use this parameter when the VLAN is not a member of an OVS bridge. For example, use this parameter to attach the VLAN to a bonded interface device.
use_dhcp	False	Use DHCP to get an IP address.
use_dhcpv6	False	Use DHCP to get a v6 IP address.
addresses		A list of IP addresses assigned to the VLAN.
routes		A list of routes assigned to the VLAN. For more information, see Section 3.2.7, "routes".
mtu	1500	The maximum transmission unit (MTU) of the connection.
primary	False	Defines the VLAN as the primary interface.

Option	Default	Description
persist_mapping	False	Write the device alias configuration instead of the system names.
dhclient_args	None	Arguments that you want to pass to the DHCP client.
dns_servers	None	List of DNS servers that you want to use for the VLAN.

Example

```
...
      edpm_network_config_template: |
        ---
        {% set mtu_list = [ctlplane_mtu] %}
        {% for network in nodeset_networks %}
        {{ mtu_list.append(lookup(vars, networks_lower[network] ~ _mtu)) }}
        {%- endfor %}
        {% set min_viable_mtu = mtu_list | max %}
        network_config:
        ...
          members:
          - type: vlan
            device: nic{{ loop.index + 1 }}
            mtu: {{ lookup(vars, networks_lower[network] ~ _mtu) }}
            vlan_id: {{ lookup(vars, networks_lower[network] ~ _vlan_id) }}
            addresses:
            - ip_netmask:
                {{ lookup(vars, networks_lower[network] ~ _ip) }}/{{ lookup(vars, networks_lower[network] ~ _cidr) }}
            routes: {{ lookup(vars, networks_lower[network] ~ _host_routes) }}
...
```

3.2.3. ovs_bridge

Defines a bridge in Open vSwitch (OVS), which connects multiple **interface**, **ovs_bond**, and **vlan** objects together.

The network interface type, **ovs_bridge**, takes a parameter **name**.

IMPORTANT

The **ovs_bridge** interface is not recommended for control plane network traffic. The OVS bridge connects to the Networking service (neutron) server to obtain configuration data. If the OpenStack control traffic, typically the Control Plane and Internal API networks, is placed on an OVS bridge, then connectivity to the neutron server is lost whenever you upgrade OVS, or the OVS bridge is restarted by the admin user or process. This causes some downtime. If downtime is not acceptable in these circumstances, then you must place the Control group networks on a separate interface or bond rather than on an OVS bridge:

- You can achieve a minimal setting when you put the Internal API network on a VLAN on the provisioning interface and the OVS bridge on a second interface.
- To implement bonding, you need at least two bonds (four network interfaces). Place the control group on a Linux bond. If the switch does not support LACP fallback to a single interface for PXE boot, then this solution requires at least five NICs.

NOTE

If you have multiple bridges, you must use distinct bridge names other than accepting the default name of **bridge_name**. If you do not use distinct names, then during the converge phase, two network bonds are placed on the same bridge.

Table 3.3. ovs_bridge options

Option	Default	Description
name		Name of the bridge.
use_dhcp	False	Use DHCP to get an IP address.
use_dhcpv6	False	Use DHCP to get a v6 IP address.
addresses		A list of IP addresses assigned to the bridge.
routes		A list of routes assigned to the bridge. For more information, see Section 3.2.7, "routes".
mtu	1500	The maximum transmission unit (MTU) of the connection.
members		A sequence of interface, VLAN, and bond objects that you want to use in the bridge.
ovs_options		A set of options to pass to OVS when creating the bridge.

Option	Default	Description
ovs_extra		A set of options to to set as the OVS_EXTRA parameter in the network configuration file of the bridge.
defroute	True	Use a default route provided by the DHCP service. Only applies when you enable **use_dhcp** or **use_dhcpv6**.
persist_mapping	False	Write the device alias configuration instead of the system names.
dhclient_args	None	Arguments that you want to pass to the DHCP client.
dns_servers	None	List of DNS servers that you want to use for the bridge.

Example

```
...
      edpm_network_config_template: |
        ---
        {% set mtu_list = [ctlplane_mtu] %}
        {% for network in nodeset_networks %}
        {{ mtu_list.append(lookup(vars, networks_lower[network] ~ _mtu)) }}
        {%- endfor %}
        {% set min_viable_mtu = mtu_list | max %}
        network_config:
        - type: ovs_bridge
          name: br-bond
          dns_servers: {{ ctlplane_dns_nameservers }}
          domain: {{ dns_search_domains }}
          members:
          - type: ovs_bond
            name: bond1
            mtu: {{ min_viable_mtu }}
            ovs_options: {{ bound_interface_ovs_options }}
            members:
            - type: interface
              name: nic2
              mtu: {{ min_viable_mtu }}
              primary: true
            - type: interface
              name: nic3
              mtu: {{ min_viable_mtu }}
              ...
```

3.2.4. Network interface bonding

You can bundle multiple physical NICs together to form a single logical channel known as a bond. You can configure bonds to provide redundancy for high availability systems or increased throughput.

Red Hat OpenStack Platform supports Open vSwitch (OVS) kernel bonds, OVS-DPDK bonds, and Linux kernel bonds.

Table 3.4. Supported interface bonding types

Bond type	Type value	Allowed bridge types	Allowed members
OVS kernel bonds	**ovs_bond**	**ovs_bridge**	**interface**
OVS-DPDK bonds	**ovs_dpdk_bond**	**ovs_user_bridge**	**ovs_dpdk_port**
Linux kernel bonds	**linux_bond**	**ovs_bridge**	**interface**

IMPORTANT

Do not combine **ovs_bridge** and **ovs_user_bridge** on the same node.

3.2.4.1. ovs_bond

Defines a bond in Open vSwitch (OVS) to join two or more **interfaces** together. This helps with redundancy and increases bandwidth.

Table 3.5. ovs_bond options

Option	Default	Description
name		Name of the bond.
use_dhcp	False	Use DHCP to get an IP address.
use_dhcpv6	False	Use DHCP to get a v6 IP address.
addresses		A list of IP addresses assigned to the bond.
routes		A list of routes assigned to the bond. For more information, see Section 3.2.7, "routes".
mtu	1500	The maximum transmission unit (MTU) of the connection.
primary	False	Defines the interface as the primary interface.

Option	Default	Description
members		A sequence of interface objects that you want to use in the bond.
ovs_options		A set of options to pass to OVS when creating the bond. For more information, see Table 3.6, "**ovs_options** parameters for OVS bonds".
ovs_extra		A set of options to set as the OVS_EXTRA parameter in the network configuration file of the bond.
defroute	True	Use a default route provided by the DHCP service. Only applies when you enable **use_dhcp** or **use_dhcpv6**.
persist_mapping	False	Write the device alias configuration instead of the system names.
dhclient_args	None	Arguments that you want to pass to the DHCP client.
dns_servers	None	List of DNS servers that you want to use for the bond.

Table 3.6. ovs_options parameters for OVS bonds

ovs_option	Description
bond_mode=balance-slb	Source load balancing (slb) balances flows based on source MAC address and output VLAN, with periodic rebalancing as traffic patterns change. When you configure a bond with the **balance-slb** bonding option, there is no configuration required on the remote switch. The Networking service (neutron) assigns each source MAC and VLAN pair to a link and transmits all packets from that MAC and VLAN through that link. A simple hashing algorithm based on source MAC address and VLAN number is used, with periodic rebalancing as traffic patterns change. The **balance-slb** mode is similar to mode 2 bonds used by the Linux bonding driver. You can use this mode to provide load balancing even when the switch is not configured to use LACP.

ovs_option	Description
bond_mode=active-backup	When you configure a bond using **active-backup** bond mode, the Networking service keeps one NIC in standby. The standby NIC resumes network operations when the active connection fails. Only one MAC address is presented to the physical switch. This mode does not require switch configuration, and works when the links are connected to separate switches. This mode does not provide load balancing.
lacp=[active | passive | off]	Controls the Link Aggregation Control Protocol (LACP) behavior. Only certain switches support LACP. If your switch does not support LACP, use **bond_mode=balance-slb** or **bond_mode=active-backup**.
other-config:lacp-fallback-ab=true	Set active-backup as the bond mode if LACP fails.
other_config:lacp-time=[fast | slow]	Set the LACP heartbeat to one second (fast) or 30 seconds (slow). The default is slow.
other_config:bond-detect-mode=[miimon | carrier]	Set the link detection to use miimon heartbeats (miimon) or monitor carrier (carrier). The default is carrier.
other_config:bond-miimon-interval=100	If using miimon, set the heartbeat interval (milliseconds).
bond_updelay=1000	Set the interval (milliseconds) that a link must be up to be activated to prevent flapping.
other_config:bond-rebalance-interval=10000	Set the interval (milliseconds) that flows are rebalancing between bond members. Set this value to zero to disable flow rebalancing between bond members.

Example - OVS bond

```
...
      edpm_network_config_template: |
        ---
        {% set mtu_list = [ctlplane_mtu] %}
        {% for network in nodeset_networks %}
        {{ mtu_list.append(lookup(vars, networks_lower[network] ~ _mtu)) }}
        {%- endfor %}
        {% set min_viable_mtu = mtu_list | max %}
        network_config:
        ...
          members:
            - type: ovs_bond
```

```
        name: bond1
        mtu: {{ min_viable_mtu }}
        ovs_options: {{ bond_interface_ovs_options }}
        members:
        - type: interface
          name: nic2
          mtu: {{ min_viable_mtu }}
          primary: true
        - type: interface
          name: nic3
          mtu: {{ min_viable_mtu }}
```

Example - OVS DPDK bond

In this example, a bond is created as part of an OVS user space bridge:

```
edpm_network_config_template: |
  ---
  {% set mtu_list = [ctlplane_mtu] %}
  {% for network in nodeset_networks %}
  {{ mtu_list.append(lookup(vars, networks_lower[network] ~ _mtu)) }}
  {%- endfor %}
  {% set min_viable_mtu = mtu_list | max %}
  network_config:
  ...
    members:
    - type: ovs_user_bridge
      name: br-dpdk0
      members:
      - type: ovs_dpdk_bond
        name: dpdkbond0
        rx_queue: {{ num_dpdk_interface_rx_queues }}
        members:
        - type: ovs_dpdk_port
          name: dpdk0
          members:
          - type: interface
            name: nic4
        - type: ovs_dpdk_port
          name: dpdk1
          members:
          - type: interface
            name: nic5
```

3.2.5. LACP with OVS bonding modes

You can use Open vSwitch (OVS) bonds with the optional Link Aggregation Control Protocol (LACP). LACP is a negotiation protocol that creates a dynamic bond for load balancing and fault tolerance.

Use the following table to understand support compatibility for OVS kernel and OVS-DPDK bonded interfaces in conjunction with LACP options.

IMPORTANT

On control and storage networks, Red Hat recommends that you use Linux bonds with VLAN and LACP, because OVS bonds carry the potential for control plane disruption that can occur when OVS or the neutron agent is restarted for updates, hot fixes, and other events. The Linux bond-LACP-VLAN configuration provides NIC management without the OVS disruption potential.

Table 3.7. LACP options for OVS kernel and OVS-DPDK bond modes

Objective	OVS bond mode	Compatible LACP options	Notes
High availability (active-passive)	**active-backup**	**active**, **passive**, or **off**	
Increased throughput (active-active)	**balance-slb**	**active**, **passive**, or **off**	• Performance is affected by extra parsing per packet. • There is a potential for vhost-user lock contention.
	balance-tcp	**active** or **passive**	• As with balance-slb, performance is affected by extra parsing per packet and there is a potential for vhost-user lock contention. • LACP must be configured and enabled. • Set **lb-output-action=true**. For example: `ovs-vsctl set port <bond port> other_config:lb-output-action=true`

3.2.6. linux_bond

Defines a Linux bond that joins two or more **interfaces** together. This helps with redundancy and increases bandwidth. Ensure that you include the kernel-based bonding options in the **bonding_options** parameter.

Table 3.8. linux_bond options

Option	Default	Description
name		Name of the bond.
use_dhcp	False	Use DHCP to get an IP address.
use_dhcpv6	False	Use DHCP to get a v6 IP address.
addresses		A list of IP addresses assigned to the bond.
routes		A list of routes assigned to the bond. See Section 3.2.7, “routes”.
mtu	1500	The maximum transmission unit (MTU) of the connection.
members		A sequence of interface objects that you want to use in the bond.
bonding_options		A set of options when creating the bond. See **bonding_options** parameters for Linux bonds.
defroute	True	Use a default route provided by the DHCP service. Only applies when you enable **use_dhcp** or **use_dhcpv6**.
persist_mapping	False	Write the device alias configuration instead of the system names.
dhclient_args	None	Arguments that you want to pass to the DHCP client.
dns_servers	None	List of DNS servers that you want to use for the bond.

bonding_options parameters for Linux bonds

The **bonding_options** parameter sets the specific bonding options for the Linux bond. See the Linux bonding examples that follow this table:

bonding_options	Description
mode	Sets the bonding mode, which in the example is **802.3ad** or LACP mode. For more information about Linux bonding modes, see Configuring a network bond in Red Hat Enterprise Linux 9,*Configuring and managing networking*.
lacp_rate	Defines whether LACP packets are sent every 1 second, or every 30 seconds.
updelay	Defines the minimum amount of time that an interface must be active before it is used for traffic. This minimum configuration helps to mitigate port flapping outages.
miimon	The interval in milliseconds that is used for monitoring the port state using the MIIMON functionality of the driver.

Example - Linux bond

```
...
      edpm_network_config_template: |
        ---
        {% set mtu_list = [ctlplane_mtu] %}
        {% for network in nodeset_networks %}
        {{ mtu_list.append(lookup(vars, networks_lower[network] ~ _mtu)) }}
        {%- endfor %}
        {% set min_viable_mtu = mtu_list | max %}
        network_config:
        - type: linux_bond
          name: bond1
          mtu: {{ min_viable_mtu }}
          bonding_options: "mode=802.3ad lacp_rate=fast updelay=1000 miimon=100
xmit_hash_policy=layer3+4"
          members:
            type: interface
            name: ens1f0
            mtu: {{ min_viable_mtu }}
            primary: true
          type: interface
            name: ens1f1
            mtu: {{ min_viable_mtu }}
            ...
```

Example - Linux bond: bonding two interfaces

```
...
```

```
    edpm_network_config_template: |
      ---
      {% set mtu_list = [ctlplane_mtu] %}
      {% for network in nodeset_networks %}
      {{ mtu_list.append(lookup(vars, networks_lower[network] ~ _mtu)) }}
      {%- endfor %}
      {% set min_viable_mtu = mtu_list | max %}
      network_config:
      - type: linux_bond
        name: bond1
        members:
        - type: interface
          name: nic2
        - type: interface
          name: nic3
        bonding_options: "mode=802.3ad lacp_rate=[fast|slow] updelay=1000 miimon=100"
        ...
```

Example - Linux bond set to **active-backup** mode with one VLAN

```
....
    edpm_network_config_template: |
      ---
      {% set mtu_list = [ctlplane_mtu] %}
      {% for network in nodeset_networks %}
      {{ mtu_list.append(lookup(vars, networks_lower[network] ~ _mtu)) }}
      {%- endfor %}
      {% set min_viable_mtu = mtu_list | max %}
      network_config:
      - type: linux_bond
        name: bond_api
        bonding_options: "mode=active-backup"
        use_dhcp: false
        dns_servers:
          get_param: DnsServers
        members:
        - type: interface
          name: nic3
          primary: true
        - type: interface
          name: nic4

        - type: vlan
          vlan_id:
            get_param: InternalApiNetworkVlanID
          device: bond_api
          addresses:
          - ip_netmask:
              get_param: InternalApiIpSubnet
```

Example - Linux bond on OVS bridge

In this example, the bond is set to **802.3ad** with LACP mode and one VLAN:

```
...
```

```
edpm_network_config_template: |
  ---
  {% set mtu_list = [ctlplane_mtu] %}
  {% for network in nodeset_networks %}
  {{ mtu_list.append(lookup(vars, networks_lower[network] ~ _mtu)) }}
  {%- endfor %}
  {% set min_viable_mtu = mtu_list | max %}
  network_config:
  -  type: ovs_bridge
     name: br-tenant
     use_dhcp: false
     mtu: 9000
     members:
       - type: linux_bond
         name: bond_tenant
         bonding_options: "mode=802.3ad updelay=1000 miimon=100"
         use_dhcp: false
         dns_servers:
           get_param: DnsServers
         members:
         - type: interface
           name: p1p1
           primary: true
         - type: interface
           name: p1p2
       - type: vlan
         device: bond_tenant
         vlan_id: {get_param: TenantNetworkVlanID}
         addresses:
           - ip_netmask: {get_param: TenantIpSubnet}
           ...
```

3.2.7. routes

Defines a list of routes to apply to a network interface, VLAN, bridge, or bond.

Table 3.9. routes options

Option	Default	Description
ip_netmask	None	IP and netmask of the destination network.
default	False	Sets this route to a default route. Equivalent to setting **ip_netmask: 0.0.0.0/0**.
next_hop	None	The IP address of the router used to reach the destination network.

Example - routes

```
...
```

```
edpm_network_config_template: |
  ---
  {% set mtu_list = [ctlplane_mtu] %}
  {% for network in nodeset_networks %}
  {{ mtu_list.append(lookup(vars, networks_lower[network] ~ _mtu)) }}
  {%- endfor %}
  {% set min_viable_mtu = mtu_list | max %}
  network_config:
  -  type: ovs_bridge
     name: br-tenant
     ...
     routes: {{ [ctlplane_host_routes] | flatten | unique }}
     ...
```

Additional resources

- Section 3.3, “Example custom network interfaces”

3.3. EXAMPLE CUSTOM NETWORK INTERFACES

The following example illustrates how you can use a template to customize network interfaces for Red Hat OpenStack Services on OpenShift (RHOSO) environments.

Example

This template example configures the control group separate from the OVS bridge. The template uses five network interfaces and assigns a number of tagged VLAN devices to the numbered interfaces. On **nic2** and **nic3** the template creates a linux bond for control plane traffic. The template creates OVS bridges for the RHOSO data plane on **nic4** and **nic5**.

```
edpm_network_config_os_net_config_mappings:
  edpm-compute-0:
    dmiString: system-serial-number
    id: 3V3J4V3
    nic1: ec:2a:72:40:ca:2e
    nic2: 6c:fe:54:3f:8a:00
    nic3: 6c:fe:54:3f:8a:01
    nic4: 6c:fe:54:3f:8a:02
    nic5: 6c:fe:54:3f:8a:03
    nic6: e8:eb:d3:33:39:12
    nic7: e8:eb:d3:33:39:13

edpm_network_config_template: |
  ---
  {% set mtu_list = [ctlplane_mtu] %}
  {% for network in nodeset_networks %}
  {{ mtu_list.append(lookup('vars', networks_lower[network] ~ '_mtu')) }}
  {%- endfor %}
  {% set min_viable_mtu = mtu_list | max %}
  - type: interface
    name: nic1
    use_dhcp: false
    use_dhcpv6: false
    type: linux_bond
    name: bond_api
```

```
            use_dhcp: false
            use_dhcpv6: false
            bonding_options: "mode=active-backup"
            dns_servers: {{ ctlplane_dns_nameservers }}
            addresses:
            ip_netmask: {{ ctlplane_ip }}/{{ ctlplane_cidr }}
            routes:
            default: true
            next_hop: 192.168.122.1
            members:
              - type: interface
                name: nic2
                primary: true
              - type: interface
                name: nic3
          {% for network in nodeset_networks if network not in ['external', 'tenant'] %}
          - type: vlan
            mtu: {{ lookup('vars', networks_lower[network] ~ '_mtu') }}
            vlan_id: {{ lookup('vars', networks_lower[network] ~ '_vlan_id') }}
            device: bond_api
            addresses:
            - ip_netmask: {{ lookup('vars', networks_lower[network] ~ '_ip') }}/{{ lookup('vars',
networks_lower[network] ~ '_cidr') }}
          {% endfor %}
          - type: ovs_bridge
            name: br-access
            use_dhcp: false
            use_dhcpv6: false
            members:
            - type: linux_bond
              name: bond_data
              mtu: {{ min_viable_mtu }}
              bonding_options: "mode=active-backup"
              members:
              - type: interface
                name: nic4
              - type: interface
                name: nic5
            - type: vlan
              vlan_id: {{ lookup('vars', networks_lower['tenant'] ~ '_vlan_id') }}
              mtu: {{ lookup('vars', networks_lower['tenant'] ~ '_mtu') }}
              device: bond_data
              addresses:
              - ip_netmask:
                  {{ lookup('vars', networks_lower['tenant'] ~ '_ip') }}/{{ lookup('vars', networks_lower['tenant']
~ '_cidr') }}
              routes: {{ lookup('vars', networks_lower['tenant'] ~ '_host_routes') }}
```

Additional resources

- Section 3.2, “Network interface configuration options”

CHAPTER 4. MANAGING PROJECT NETWORKS

Project networks help you to isolate network traffic for cloud computing. Steps to create a project network include planning and creating the network, and adding subnets and routers.

4.1. VLAN PLANNING

When you plan for VLANs in your Red Hat OpenStack Services on OpenShift (RHOSO) environment, 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.

4.2. DEFAULT RED HAT OPENSTACK SERVICES ON OPENSHIFT NETWORKS

The following physical data center networks are typically implemented for a Red Hat OpenStack Services on OpenShift (RHOSO) deployment:

- Control plane network: This network is used by the OpenStack Operator for Ansible SSH access to deploy and connect to the data plane nodes from the Red Hat OpenShift Container Platform (RHOCP) environment. This network is also used by data plane nodes for live migration of instances.
- External network: (Optional) You can configure an external network if one is required for your environment. For example, you might create an external network for any of the following purposes:
  - To provide virtual machine instances with Internet access.
  - To create flat provider networks that are separate from the control plane.
  - To configure VLAN provider networks on a separate bridge from the control plane.
  - To provide access to virtual machine instances with floating IPs on a network other than the control plane network.
- Internal API network: This network is used for internal communication between RHOSO components.
- Storage network: This network is used for block storage, RBD, NFS, FC, and iSCSI.
- Tenant (project) network: This network is used for data communication between virtual machine instances within the cloud deployment.
- Storage Management network: (Optional) This network is used by storage components. For example, Red Hat Ceph Storage uses the Storage Management network in a hyperconverged infrastructure (HCI) environment as the **cluster_network** to replicate data.

NOTE

For more information on Red Hat Ceph Storage network configuration, see Ceph network configuration in the *Red Hat Ceph Storage Configuration Guide* .

The following table details the default networks used in a RHOSO deployment. If required, you can update the networks for your environment.

NOTE

By default, the control plane and external networks do not use VLANs. Networks that do not use VLANs must be placed on separate NICs. You can use a VLAN for the control plane network on new RHOSO deployments. You can also use the Native VLAN on a trunked interface as the non-VLAN network. For example, you can have the control plane and the internal API on one NIC, and the external network with no VLAN on a separate NIC.

Table 4.1. Default RHOSO networks

Network name	VLAN	CIDR	NetConfig **allocationRange**	MetalLB **IPAddressPool** range	net-attach-def **ipam** range	OCP worker **nncp** range
ctlplane	n/a	192.168.122.0/24	192.168.122.100 - 192.168.122.250	192.168.122.80 - 192.168.122.90	192.168.122.30 - 192.168.122.70	192.168.122.10 - 192.168.122.20
external	n/a	10.0.0.0/24	10.0.0.100 - 10.0.0.250	n/a	n/a	
internalapi	20	172.17.0.0/24	172.17.0.100 - 172.17.0.250	172.17.0.80 - 172.17.0.90	172.17.0.30 - 172.17.0.70	172.17.0.10 - 172.17.0.20
storage	21	172.18.0.0/24	172.18.0.100 - 172.18.0.250	n/a	172.18.0.30 - 172.18.0.70	172.18.0.10 - 172.18.0.20
tenant	22	172.19.0.0/24	172.19.0.100 - 172.19.0.250	n/a	172.19.0.30 - 172.19.0.70	172.19.0.10 - 172.19.0.20
storageMgmt	23	172.20.0.0/24	172.20.0.100 - 172.20.0.250	n/a	172.20.0.30 - 172.20.0.70	172.20.0.10 - 172.20.0.20

4.3. IP ADDRESS CONSUMPTION

In Red Hat OpenStack Services on OpenShift (RHOSO) environments 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 in Red Hat OpenStack Services on OpenShift (RHOSO) environments. These resources are considered local to the cloud infrastructure, and do not need to be reachable by systems in the external physical network:

- **Project networks** - Each project 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.
- **Instances** - Each instance requires an address from the project 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 in a Red Hat OpenStack Services on OpenShift (RHOSO) environment that accommodate multiple subnets, with each subnet being assigned a range of IP addresses:

Table 4.2. 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
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

Subnet name	Address range	Number of addresses	Subnet Mask
Provider network (infrastructure)	10.10.3.10 - 10.10.3.250	241	255.255.252.0

4.6. WORKING WITH SUBNETS

In Red Hat OpenStack Services on OpenShift (RHOSO) environments use subnets to grant network connectivity to instances. A subnet is a pool of IP addresses. Instances are assigned to a Networking service (neutron) network. One network can have multiple subnets, and you can also add IP addresses from multiple subnets to the port.

You can create subnets only in pre-existing networks. Remember that project networks in the Networking service 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.

You can lessen network latency and load by grouping systems in the same subnet that require a high volume of traffic between each other.

4.7. CONFIGURING FLOATING IP PORT FORWARDING

In Red Hat OpenStack Services on OpenShift (RHOSO) environments, to enable users to set up port forwarding for floating IPs, you must enable the Networking service (neutron) **port_forwarding** service plug-in.

Prerequisites

- You have the **oc** command line tool installed on your workstation.
- You are logged on to a workstation that has access to the RHOSO control plane as a user with **cluster-admin** privileges.
- The **port_forwarding** service plug-in requires that you also set the **ovn-router** service plug-in.

Procedure

- Update the control plane:

```
$ oc patch -n openstack openstackcontrolplane openstack-galera-network-isolation --
type=merge --patch "
---
spec:
  neutron:
    template:
      customServiceConfig: |
        [default]
        service_plugins=ovn-router,port_forwarding
"
```

NOTE

The **port_forwarding** service plug-in requires that you also set the **router** service plug-in.

RHOSO users can now set up port forwarding for floating IPs.

Verification

1. Access the remote shell for the OpenStackClient pod from your workstation:

```
$ oc rsh -n openstack openstackclient
```

2. Ensure that the Networking service has successfully loaded the **port_forwarding** and **router** service plug-ins:

```
$ openstack extension list --network -c Name -c Alias --max-width 74 | \
grep -i -e 'Neutron L3 Router' -i -e floating-ip-port-forwarding \
--os-cloud <cloud_name>
```

- Replace <cloud_name> with the name of the cloud on which you are running the command.

Sample output

A successful verification produces output similar to the following:

```
| Floating IP Port Forwarding       | floating-ip-port-forwarding       |
| Neutron L3 Router                 | router                            |
```

3. Exit the **openstackclient** pod:

```
$ exit
```

4.8. BRIDGING THE PHYSICAL NETWORK

In Red Hat OpenStack Services on OpenShift (RHOSO) environments you can 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.

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

Procedure

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:

```
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:

```
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 5. USING QUALITY OF SERVICE (QOS) POLICIES TO MANAGE DATA TRAFFIC

You can offer varying service levels for VM instances by using quality of service (QoS) policies to apply rate limits to egress and ingress traffic in Red Hat OpenStack Services on OpenShift (RHOSO) environments.

You can apply QoS policies to individual ports, or 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.

5.1. QOS RULES

You can configure the following rule types to define a quality of service (QoS) policy in the Red Hat OpenStack Services on OpenShift (RHOSO) Networking service (neutron):

Minimum bandwidth (`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.

Bandwidth limit (`bandwidth_limit`)

Provides bandwidth limitations on networks, ports, floating IPs (FIPs), and router gateway IPs. If implemented, any traffic that exceeds the specified rate is dropped.

DSCP marking (`dscp_marking`)

Marks network traffic with a Differentiated Services Code Point (DSCP) value.

QoS policies can be enforced in various contexts, including virtual machine instance placements, floating IP assignments, and gateway IP assignments.

Depending on the enforcement context and on the mechanism driver you use, a QoS rule affects egress traffic (upload from instance), ingress traffic (download to instance), or both.

NOTE

In ML2/OVN deployments, you can enable minimum bandwidth and bandwidth limit egress policies for hardware offloaded ports. You cannot enable ingress policies for hardware offloaded ports. For more information, see Configuring the Networking service for QoS policies.

Table 5.1. Supported traffic direction by driver (all QoS rule types)

Rule [1]	Supported traffic direction by mechanism driver

	ML2/SR-IOV	ML2/OVN
Minimum bandwidth	Egress only	Currently, no support [2]
Bandwidth limit	Egress only [3]	Egress and ingress
DSCP marking	N/A	Egress only [4]

[1] RHOSO does not support QoS for trunk ports.

[2] https://bugzilla.redhat.com/show_bug.cgi?id=2060310

[3] The mechanism drivers ignore the **max-burst-kbits** parameter because they do not support it.

[4] ML2/OVN does not support DSCP marking on tunneled protocols.

Table 5.2. Supported traffic direction by driver for placement reporting and scheduling (minimum bandwidth only)

Enforcement type	Supported traffic by direction mechanism driver	
	ML2/SR-IOV	ML2/OVN
Placement	Egress and ingress	Currently, no support

Table 5.3. Supported traffic direction by driver for enforcement types (bandwidth limit only)

Enforcement type	Supported traffic direction by mechanism driver
	ML2/OVN
Floating IP	Egress and ingress
Gateway IP	Egress and ingress [1]

[1] Technology preview in RHOSP 17.1. See BZ 2088291.

5.2. CONFIGURING THE NETWORKING SERVICE FOR QOS POLICIES

The quality of service feature in the Red Hat OpenStack Services on OpenShift (RHOSO) Networking service (neutron) is provided through the **qos** service plug-in. With the ML2/OVN mechanism driver, **qos** is loaded by default. However, this is not true for ML2/SR-IOV.

When using the **qos** service plug-in with the ML2/SR-IOV mechanism driver, you must also load the **qos** extension on their respective agents.

The following list summarizes the tasks that you must perform to configure the Networking service for QoS. The task details follow this list:

- For all types of QoS policies:
  - Add the **qos** service plug-in.
  - Add **qos** extension for the agents (SR-IOV only).
- In ML2/OVN deployments, you can enable minimum bandwidth and bandwidth limit egress policies for hardware offloaded ports. You cannot enable ingress policies for hardware offloaded ports.
- Additional tasks for scheduling VM instances using minimum bandwidth policies only:
  - Specify the hypervisor name if it differs from the name that the Compute service (nova) uses.
  - Configure the resource provider ingress and egress bandwidths for the relevant agents on each Compute node.
  - (Optional) Mark **vnic_types** as not supported.
- Additional task for DSCP marking policies on systems that use ML/OVS with tunneling only:
  - Set **dscp_inherit** to **true**.

Prerequisites

- You have the **oc** command line tool installed on your workstation.
- You are logged on to a workstation that has access to the RHOSO control plane as a user with **cluster-admin** privileges.

Procedure

1. If you are using the ML2/SR-IOV mechanism driver, you must enable the **qos** agent extension on the Compute nodes, also referred to as the RHOSO data plane.
   For more information, see Configuring the Networking service for QoS policies for SR-IOV .
2. In ML2/OVN deployments, you can enable egress minimum and maximum bandwidth policies for hardware offloaded ports, which is managed by the **neutron-ovn-agent**. The OVN agent runs on the Compute nodes and is configured through the **OpenStackDataPlaneNodeSet** CR definition. Open the **OpenStackDataPlaneNodeSet** CR definition file for the node set you want to update, for example, **my_data_plane_node_set.yaml**.
3. Add the required QoS configuration. Place the configuration in the **edpm_network_config_template** under **ansibleVars**:

```
apiVersion: dataplane.openstack.org/v1beta1
kind: OpenStackDataPlaneNodeSet
metadata:
  name: my-data-plane-node-set
spec:
  ...
  nodeTemplate:
    ...
    ansible:
      ansibleVars:
        edpm_network_config_template: |
```

```
---
OvnHardwareOffloadedQos: true
...
```

4. Save the **OpenStackDataPlaneNodeSet** CR definition file.
5. Apply the updated **OpenStackDataPlaneNodeSet** CR configuration:

```
$ oc apply -f my_data_plane_node_set.yaml
```

6. Verify that the data plane resource has been updated:

```
$ oc get openstackdataplanenodeset
```

Sample output

```
NAME                    STATUS MESSAGE
my-data-plane-node-set   False  Deployment not started
```

7. Create a file on your workstation to define the **OpenStackDataPlaneDeployment** CR, for example, **my_data_plane_deploy.yaml**:

```
apiVersion: dataplane.openstack.org/v1beta1
kind: OpenStackDataPlaneDeployment
metadata:
  name: my-data-plane-deploy
```

TIP

Give the definition file and the **OpenStackDataPlaneDeployment** CR a unique and descriptive name that indicates the purpose of the modified node set.

8. Add the **OpenStackDataPlaneNodeSet** CR that you modified:

```
spec:
  nodeSets:
    - my-data-plane-node-set
```

9. Save the **OpenStackDataPlaneDeployment** CR deployment file.
10. Deploy the modified **OpenStackDataPlaneNodeSet** CR:

```
$ oc create -f my_data_plane_deploy.yaml -n openstack
```

You can view the Ansible logs while the deployment executes:

```
$ oc get pod -l app=openstackansibleee -n openstack -w

$ oc logs -l app=openstackansibleee -n openstack -f \
--max-log-requests 10
```

11. Verify that the modified **OpenStackDataPlaneNodeSet** CR is deployed:

Example

```
$ oc get openstackdataplanedeployment -n openstack
```

Sample output

```
NAME                    STATUS   MESSAGE
my-data-plane-node-set   True     Setup Complete
```

12. Repeat the **oc get** command until you see the **NodeSet Ready** message:

Example

```
$ oc get openstackdataplanenodeset -n openstack
```

Sample output

```
NAME                    STATUS   MESSAGE
my-data-plane-node-set   True     NodeSet Ready
```

For information on the meaning of the returned status, see Data plane conditions and states in the *Deploying Red Hat OpenStack Services on OpenShift* guide.

Verification

- Confirm that the **qos** service plug-in is loaded:

```
$ openstack network qos policy list
```

If the **qos** service plug-in is loaded, then you do not receive a **ResourceNotFound** error.

5.3. CONFIGURING THE NETWORKING SERVICE FOR QOS POLICIES FOR SR-IOV

The quality of service feature in the Red Hat OpenStack Services on OpenShift (RHOSO) Networking service (neutron) is provided through the **qos** service plug-in. If your Networking service ML2 mechanism driver is SR-IOV, then you must also load the **qos** extension driver for the NIC switch agent, **neutron-sriov-nic-agent**, which runs on the Compute nodes, also referred to as the RHOSO data plane.

Prerequisites

- You have the **oc** command line tool installed on your workstation.
- You are logged on to a workstation that has access to the RHOSO control plane as a user with **cluster-admin** privileges.

Procedure

1. Open the **OpenStackDataPlaneNodeSet** CR definition file for the node set you want to update, for example, **my_data_plane_node_set.yaml**.
2. Add the required QoS configuration, **NeutronSriovAgentExtensions: "qos"**.

Place the configuration in the **edpm_network_config_template** under **ansibleVars**:

```
apiVersion: dataplane.openstack.org/v1beta1
kind: OpenStackDataPlaneNodeSet
metadata:
  name: my-data-plane-node-set
spec:
  ...
  nodeTemplate:
    ...
    ansible:
      ansibleVars:
        edpm_network_config_template: |
          ---
          NeutronSriovAgentExtensions: "qos"
          ...
```

3. Save the **OpenStackDataPlaneNodeSet** CR definition file.
4. Apply the updated **OpenStackDataPlaneNodeSet** CR configuration:

```
$ oc apply -f my_data_plane_node_set.yaml
```

5. Verify that the data plane resource has been updated:

```
$ oc get openstackdataplanenodeset
```

Sample output

```
NAME                    STATUS MESSAGE
my-data-plane-node-set   False  Deployment not started
```

6. Create a file on your workstation to define the **OpenStackDataPlaneDeployment** CR, for example, **my_data_plane_deploy.yaml**:

```
apiVersion: dataplane.openstack.org/v1beta1
kind: OpenStackDataPlaneDeployment
metadata:
  name: my-data-plane-deploy
```

TIP

Give the definition file and the **OpenStackDataPlaneDeployment** CR a unique and descriptive name that indicates the purpose of the modified node set.

7. Add the **OpenStackDataPlaneNodeSet** CR that you modified:

```
spec:
  nodeSets:
    - my-data-plane-node-set
```

8. Save the **OpenStackDataPlaneDeployment** CR deployment file.

9. Deploy the modified **OpenStackDataPlaneNodeSet** CR:

```
$ oc create -f my_data_plane_deploy.yaml -n openstack
```

You can view the Ansible logs while the deployment executes:

```
$ oc get pod -l app=openstackansibleee -n openstack -w

$ oc logs -l app=openstackansibleee -n openstack -f \
--max-log-requests 10
```

10. Verify that the modified **OpenStackDataPlaneNodeSet** CR is deployed:

Example

```
$ oc get openstackdataplanedeployment -n openstack
```

Sample output

```
NAME                  STATUS   MESSAGE
my-data-plane-node-set   True     Setup Complete
```

11. Repeat the **oc get** command until you see the **NodeSet Ready** message:

Example

```
$ oc get openstackdataplanenodeset -n openstack
```

Sample output

```
NAME                  STATUS   MESSAGE
my-data-plane-node-set   True     NodeSet Ready
```

For information on the meaning of the returned status, see Data plane conditions and states in the *Deploying Red Hat OpenStack Services on OpenShift* guide.

Verification

Confirm that the NIC switch agent, **neutron-sriov-nic-agent**, has loaded the **qos** extension.

1. Obtain the UUID for the NIC switch agent:

```
$ openstack network agent list
```

2. With the **neutron-sriov-nic-agent** UUID, run the following command:

```
$ openstack network agent show <uuid>
```

Example

```
$ openstack network agent show 8676ccb3-1de0-4ca6-8fb7-b814015d9e5f \
--max-width 70
```

Sample output

You should see an agent object with a field called **configuration**. When the **qos** extension is loaded, the **extensions** field should contain **qos** in its list.

```
----------------------------------------------------------+
| Field             | Value                                |
----------------------------------------------------------+
| admin_state_up    | UP                                   |
| agent_type        | NIC Switch agent                     |
| alive             | :-)                                  |
| availability_zone | None                                 |
| binary            | neutron-sriov-nic-agent              |
| configuration     | {device_mappings: {}, devices: 0, extensi | | | ons: [qos],
resource_provider_bandwidths: |
|                   | {}, resource_provider_hypervisors: {}, reso | | |
urce_provider_inventory_defaults: {allocatio | | | n_ratio: 1.0, min_unit: 1, step_size: 1,  |
|                   | reserved: 0}}                        |
| created_at        | 2024-08-08 08:22:57                  |
| description       | None                                 |
| ha_state          | None                                 |
| host              | edpm-compute-0.ctlplane.example.com  |
| id                | 8676ccb3-1de0-4ca6-8fb7-b814015d9e5f |
| last_heartbeat_at | 2024-08-08 08:24:27                  |
| resources_synced  | None                                 |
| started_at        | 2024-08-08 08:22:57                  |
| topic             | N/A                                  |
----------------------------------------------------------+
```

CHAPTER 6. VLAN-AWARE INSTANCES

In Red Hat OpenStack Services on OpenShift (RHOSO) environments, VM instances can send and receive VLAN-tagged traffic over a single virtual NIC. This is particularly useful for NFV applications (VNFs) that expect VLAN-tagged traffic, allowing a single virtual NIC to serve multiple customers or services. You can support VLAN-aware instances using VLAN transparent networks. As an alternative, you can support VLAN-aware instances using trunks.

6.1. VLAN TRUNKS AND VLAN TRANSPARENT NETWORKS

In Red Hat OpenStack Services on OpenShift (RHOSO) environments that use a VLAN transparent network, you set up VLAN tagging in the VM instances. The VLAN tags are transferred over the network and consumed by the instances on the same VLAN, and ignored by other instances and devices. In a VLAN transparent network, the VLANs are managed in the instance. You do not need to set up the VLAN in the OpenStack Networking Service (neutron).

VLAN trunks support VLAN-aware instances by combining VLANs into a single trunked port. For example, a project data network can use VLANs or tunneling (VXLAN, GRE, or GENEVE) segmentation, while the instances see the traffic tagged with VLAN IDs. Network packets are tagged immediately before they are injected to the instance and do not need to be tagged throughout the entire network.

The following table compares certain features of VLAN transparent networks and VLAN trunks:

	Transparent	Trunk
Mechanism driver support	ML2/OVN	ML2/OVN
VLAN setup managed by	VM instance	OpenStack Networking Service (neutron)
IP assignment	Configured in VM instance	Assigned by DHCP
VLAN ID	Flexible. You can set the VLAN ID in the instance	Fixed. Instances must use the VLAN ID configured in the trunk

6.2. ENABLING VLAN TRANSPARENCY

Enable VLAN transparency if you need to send VLAN tagged traffic between virtual machine (VM) instances. In a VLAN transparent network you can configure the VLANS directly in the VMs without configuring them in neutron.

Prerequisites

- You have the **oc** command line tool installed on your workstation.
- You are logged on to a workstation that has access to the RHOSO control plane as a user with **cluster-admin** privileges.
- Provider network of type VLAN or GENEVE. Do not use VLAN transparency in deployments with flat type provider networks.

- Ensure that the external switch supports 802.1q VLAN stacking using ethertype 0x8100 on both VLANs. OVN VLAN transparency does not support 802.1ad QinQ with outer provider VLAN ethertype set to 0x88A8 or 0x9100.

Procedure

1. Update the control plane by adding the **vlan_transparent = true** key value pair:

```
$ oc patch -n openstack openstackcontrolplane openstack-galera-network-isolation --
type=merge --patch "
---
spec:
  neutron:
    template:
      customServiceConfig: |
        [DEFAULT]
        vlan_transparent = true
"
```

2. Access the remote shell for the OpenStackClient pod from your workstation:

```
$ oc rsh -n openstack openstackclient
```

3. Create the network using the **--transparent-vlan** argument.

   Example

```
$ openstack network create <network-name> --transparent-vlan
```

   - Replace <network-name> with the name of the network that you are creating.

4. Exit the **openstackclient** pod:

```
$ exit
```

5. Set up a VLAN interface on each participating VM.
   Set the interface MTU to 4 bytes less than the MTU of the underlay network to accommodate the extra tagging required by VLAN transparency. For example, if the underlay network MTU is 1500, set the interface MTU to 1496.

   The following example command adds a VLAN interface on **eth0** with an MTU of 1496. The VLAN is 50 and the interface name is **vlan50**:

   Example

```
$ ip link add link eth0 name vlan50 type vlan id 50 mtu 1496
$ ip link set vlan50 up
$ ip addr add 192.128.111.3/24 dev vlan50
```

6. Access the remote shell for the OpenStackClient pod from your workstation:

```
$ oc rsh -n openstack openstackclient
```

7. Set **--allowed-address** on the VM port.
   Set the allowed address to the IP address you created on the VLAN interface inside the VM in step 4. Optionally, you can also set the VLAN interface MAC address:

   Example

   The following example sets the IP address to **192.128.111.3** with the optional MAC address **00:40:96:a8:45:c4** on port **fv82gwk3-qq2e-yu93-go31-56w7sf476mm0**:

   ```
   $ openstack port set --allowed-address ip-address=192.128.111.3,mac-
   address=00:40:96:a8:45:c4 fv82gwk3-qq2e-yu93-go31-56w7sf476mm0
   ```

8. Exit the **openstackclient** pod:

   ```
   $ exit
   ```

Verification

1. Ping between two VMs on the VLAN using the vlan50 IP address.
2. Use **tcpdump** on **eth0** to see if the packets arrive with the VLAN tag intact.

CHAPTER 7. CONFIGURING RBAC POLICIES

In Red Hat OpenStack Services on OpenShift (RHOSO) environments, administrators can use role-based access control (RBAC) policies in the Networking service (neutron) to control which projects are granted permission to attach instances to a network, and also access to other resources like QoS policies, security groups, address scopes, subnet pools, and address groups.

IMPORTANT

Networking service RBAC is separate from secure role-based access control (SRBAC) that the Identity service (keystone) uses in RHOSO.

7.1. CREATING RBAC POLICIES

This example procedure demonstrates how to use a Networking service (neutron) role-based access control (RBAC) policy to grant a project access to a shared network in a Red Hat OpenStack Services on OpenShift (RHOSO) environment.

Prerequisites

- The administrator has created a project for you and has provided you with a **clouds.yaml** file for you to access the cloud.
- The **python-openstackclient** package resides on your workstation.

```
$ dnf list installed python-openstackclient
```

Procedure

1. Confirm that the system **OS_CLOUD** variable is set for your cloud:

```
$ echo OS_CLOUD
my_cloud
```

Reset the variable if necessary:

```
$ export OS_CLOUD=my_other_cloud
```

As an alternative, you can specify the cloud name by adding the **--os-cloud <cloud_name>** option each time you run an **openstack** command.

2. 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  |
+--------------------------------------+-------------+-----------------------------------------------------+
```

3. View the list of projects:

```
$ openstack project list
```

```
+----------------------------------+----------+
| ID                               | Name     |
+----------------------------------+----------+
| 4b0b98f8c6c040f38ba4f7146e8680f5 | auditors |
| 519e6344f82e4c079c8e2eabb690023b | services |
| 80bf5732752a41128e612fe615c886c6 | demo     |
| 98a2f53c20ce4d50a40dac4a38016c69 | admin    |
+----------------------------------+----------+
```

4. 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
```

Sample output

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

7.2. REVIEWING RBAC POLICIES

This example procedure demonstrates how to obtain information about a Networking service (neutron) role-based access control (RBAC) policy used to grant a project access to a shared network in a Red Hat OpenStack Services on OpenShift (RHOSO) environment.

Prerequisites

- The administrator has created a project for you and has provided you with a **clouds.yaml** file for you to access the cloud.
- The **python-openstackclient** package resides on your workstation.

```
$ dnf list installed python-openstackclient
```

Procedure

1. Confirm that the system **OS_CLOUD** variable is set for your cloud:

```
$ echo OS_CLOUD
my_cloud
```

   Reset the variable if necessary:

```
$ export OS_CLOUD=my_other_cloud
```

   As an alternative, you can specify the cloud name by adding the **--os-cloud <cloud_name>** option each time you run an **openstack** command.

2. Run the **openstack network rbac list** command to retrieve the ID of your existing role-based access control (RBAC) policies:

```
$ openstack network rbac list
```

   Sample output

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

3. 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
```

   Sample output

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

7.3. DELETING RBAC POLICIES

This example procedure demonstrates how to remove a Networking service (neutron) role-based access control (RBAC) policy that grants a project access to a shared network in a Red Hat OpenStack Services on OpenShift (RHOSO) environment.

Prerequisites

- The administrator has created a project for you and has provided you with a **clouds.yaml** file for you to access the cloud.
- The **python-openstackclient** package resides on your workstation.

```
$ dnf list installed python-openstackclient
```

Procedure

1. Confirm that the system **OS_CLOUD** variable is set for your cloud:

```
$ echo OS_CLOUD
my_cloud
```

Reset the variable if necessary:

```
$ export OS_CLOUD=my_other_cloud
```

As an alternative, you can specify the cloud name by adding the **--os-cloud <cloud_name>** option each time you run an **openstack** command.

2. 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 |
+--------------------------------------+-------------+--------------------------------------+
```

3. 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
```

7.4. GRANTING RBAC POLICY ACCESS FOR EXTERNAL NETWORKS

In a Red Hat OpenStack Services on OpenShift (RHOSO) environment, you can use a Networking service (neutron) role-based access control (RBAC) policy to grant a project access to an external networks—networks with gateway interfaces attached.

In the following example, a RBAC policy is created for the **web-servers** network and access is granted to the **engineering** project, **c717f263785d4679b16a122516247deb**:

Prerequisites

- You have the **oc** command line tool installed on your workstation.
- You are logged on to a workstation that has access to the RHOSO control plane as a user with **cluster-admin** privileges.

Procedure

1. Access the remote shell for the OpenStackClient pod from your workstation:

```
$ oc rsh -n openstack openstackclient
```

2. 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
```

Sample output

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

3. Exit the **openstackclient** pod:

```
$ exit
```

CHAPTER 8. COMMON ADMINISTRATIVE NETWORKING TASKS

Sometimes you might need to perform administration tasks on the Red Hat OpenStack Services on OpenShift (RHOSO) Networking service (neutron) such as specifying the name assigned to ports by the internal DNS.

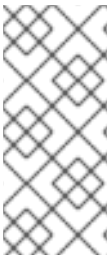
This section includes the following topics:

- Section 8.1, “Configuring shared security groups”
- Section 8.2, “Specifying the name that DNS assigns to ports”
- Section 8.3, “Enabling NUMA affinity on ports”

8.1. CONFIGURING SHARED SECURITY GROUPS

When you want one or more projects to be able to share data in a Red Hat OpenStack Services on OpenShift (RHOSO) environment, you can use the Networking service (neutron) RBAC policy feature to share a security group. You create security groups and Networking service role-based access control (RBAC) policies using the OpenStack Client.

You can apply a security group directly to an instance during instance creation, or to a port on the running instance.



NOTE

You cannot apply a role-based access control (RBAC)-shared security group directly to an instance during instance creation. To apply an RBAC-shared security group to an instance you must first create the port, apply the shared security group to that port, and then assign that port to the instance. See Adding a security group to a port in *Creating and managing instances*.

Prerequisites

- You have at least two RHOSO projects that you want to share.
- In one of the projects, the *current project*, you have created a security group that you want to share with another project, the *target project*.
  In this example, the **ping_ssh** security group is created:

  Example

  ```
  $ openstack security group create ping_ssh
  ```

- You have the **oc** command line tool installed on your workstation.
- You are logged on to a workstation that has access to the RHOSO control plane as a user with **cluster-admin** privileges.

Procedure

1. Access the remote shell for the OpenStackClient pod from your workstation:

```
$ oc rsh -n openstack openstackclient
```

2. Obtain the names or IDs of the project that contains the security group and the target project.

```
$ openstack project list
```

3. Obtain the name or ID of the security group that you want to share between RHOSO projects.

```
$ openstack security group list
```

4. Using the identifiers from the previous steps, create an RBAC policy using the **openstack network rbac create** command.
In this example, the ID of the target project is **32016615de5d43bb88de99e7f2e26a1e**. The ID of the security group is **5ba835b7-22b0-4be6-bdbe-e0722d1b5f24**:

Example

```
$ openstack network rbac create --target-project \
32016615de5d43bb88de99e7f2e26a1e --action access_as_shared \
--type security_group 5ba835b7-22b0-4be6-bdbe-e0722d1b5f24
```

--target-project

specifies the project that requires access to the security group.

TIP

You can share data between *all* projects by using the **--target-all-projects** argument instead of **--target-project <target_project>**. By default, only the admin user has this privilege.

--action access_as_shared

specifies what the project is allowed to do.

--type

indicates that the target object is a security group.

5ba835b7-22b0-4be6-bdbe-e0722d1b5f24

is the ID of the particular security group which is being granted access to.
The target project is able to access the security group when running the OpenStack Client **security group** commands, in addition to being able to bind to its ports. No other users (other than administrators and the owner) are able to access the security group.

TIP

To remove access for the target project, delete the RBAC policy that allows it using the **openstack network rbac delete** command.

5. Exit the **openstackclient** pod:

```
$ exit
```

8.2. SPECIFYING THE NAME THAT DNS ASSIGNS TO PORTS

In Red Hat OpenStack Services on OpenShift (RHOSO) environments, you can specify the name assigned to ports by the internal DNS. You enable this functionality in the Networking service (neutron), by loading the ML2 extension driver, DNS domain for ports, **dns_domain_ports**.

After loading the driver, you can use the OpenStack Client port commands, **port set** or **port create**, with **--dns-name** to assign a port name.

IMPORTANT

You must enable the DNS domain for ports extension (**dns_domain_ports**) for DNS to internally resolve names for ports in your RHOSO environment. Using the **NeutronDnsDomain** default value, **openstacklocal**, means that the Networking service does not internally resolve port names for DNS.

Also, when the DNS domain for ports extension is enabled, the Compute service automatically populates the **dns_name** attribute with the **hostname** attribute of the instance during the boot of VM instances. At the end of the boot process, **dnsmasq** recognizes the allocated ports by their instance hostname.

Prerequisites

- You have the **oc** command line tool installed on your workstation.
- You are logged on to a workstation that has access to the RHOSO control plane as a user with **cluster-admin** privileges.

Procedure

- Update the control plane with the key value pair, **service_plugins=dns_domain_ports**:

```
$ oc patch -n openstack openstackcontrolplane openstack-galera-network-isolation --
type=merge --patch "
---
spec:
  neutron:
    template:
      customServiceConfig: |
        [ml2]
        extension_drivers=dns_domain_ports
"
```

NOTE

If you set **dns_domain_ports**, ensure that the deployment does not also use **dns_domain**, the DNS Integration extension. These extensions are incompatible, and both extensions cannot be defined simultaneously.

RHOSO users can now set up port forwarding for floating IPs.

Verification

1. Access the remote shell for the OpenStackClient pod from your workstation:

```
$ oc rsh -n openstack openstackclient
```

2. Confirm that the Networking service has successfully loaded the **dns_domain_ports** ML2 extension driver:

```
$ openstack extension list --network --max-width 75 | \
grep dns-domain-ports --os-cloud <cloud_name>
```

- Replace <cloud_name> with the name of the cloud on which you are running the command.

Sample output

A successful verification produces output similar to the following:

```
| dns_domain for ports
| dns-domain-ports    | Allows the DNS domain to be specified for a network
port.
```

3. Create a new port (**new_port**) on a network (**public**). Assign a DNS name (**my_port**) to the port.

Example

```
$ openstack port create --network public --dns-name my_port new_port
```

4. Display the details for your port (**new_port**).

Example

```
$ openstack port show -c dns_assignment -c dns_domain -c dns_name -c name new_port
```

Sample output

```
+-------------------------+----------------------------------------------+
| Field                   | Value                                        |
+-------------------------+----------------------------------------------+
| dns_assignment          | fqdn='my_port.example.com',                  |
|                         | hostname='my_port',                          |
|                         | ip_address='10.65.176.113'                   |
| dns_domain              | example.com                                  |
| dns_name                | my_port                                      |
| name                    | new_port                                     |
+-------------------------+----------------------------------------------+
```

Under **dns_assignment**, the fully qualified domain name (**fqdn**) value for the port contains a concatenation of the DNS name (**my_port**) and the domain name (**example.com**) that you set earlier with **NeutronDnsDomain**.

5. Create a new VM instance (**my_vm**) using the port (**new_port**) that you just created.

Example

```
$ openstack server create --image rhel --flavor m1.small --port new_port my_vm
```

6. Display the details for your port (**new_port**).

   Example

   ```
   $ openstack port show -c dns_assignment -c dns_domain -c dns_name -c name new_port
   ```

   Sample output

   ```
   +-------------------------+---------------------------------------------+
   | Field                   | Value                                       |
   +-------------------------+---------------------------------------------+
   | dns_assignment          | fqdn='my_vm.example.com',                   |
   |                         | hostname='my_vm',                           |
   |                         | ip_address='10.65.176.113'                  |
   | dns_domain              | example.com                                 |
   | dns_name                | my_vm                                       |
   | name                    | new_port                                    |
   +-------------------------+---------------------------------------------+
   ```

   Note that the Compute service changes the **dns_name** attribute from its original value (**my_port**) to the name of the instance with which the port is associated (**my_vm**).

7. Exit the **openstackclient** pod:

   ```
   $ exit
   ```

8.3. ENABLING NUMA AFFINITY ON PORTS

In Red Hat OpenStack Services on OpenShift (RHOSO) environments, to enable users to create instances with NUMA affinity on the port, you must load the Networking service (neutron) ML2 extension driver, NUMA port affinity policy, **port_numa_affinity_policy**.

Prerequisites

- You have the **oc** command line tool installed on your workstation.
- You are logged on to a workstation that has access to the RHOSO control plane as a user with **cluster-admin** privileges.

Procedure

- Update the control plane with the key value pair, **service_plugins=port_numa_affinity_policy**:

  ```
  $ oc patch -n openstack openstackcontrolplane openstack-galera-network-isolation --
  type=merge --patch "
  ---
  spec:
    neutron:
      template:
        customServiceConfig: |
          [ml2]
          extension_drivers=port_numa_affinity_policy
  "
  ```

Verification

1. Access the remote shell for the OpenStackClient pod from your workstation:

```
$ oc rsh -n openstack openstackclient
```

2. Confirm that the Networking service has successfully loaded the **port_numa_affinity_policy** ML2 extension driver:

```
$ openstack extension list --network --max-width 74 | \
grep port-numa-affinity-policy --os-cloud <cloud_name>
```

- Replace <cloud_name> with the name of the cloud on which you are running the command.

Sample output

A successful verification produces output similar to the following:

```
| Port NUMA affinity policy
| port-numa-affinity-policy              | Expose the port NUMA affinity
policy
```

3. Create a new port.
When you create a port, use one of the following options to specify the NUMA affinity policy to apply to the port:

- **--numa-policy-required** - NUMA affinity policy required to schedule this port.
- **--numa-policy-preferred** - NUMA affinity policy preferred to schedule this port.
- **--numa-policy-legacy** - NUMA affinity policy using legacy mode to schedule this port.

Example

```
$ openstack port create --network public \
  --numa-policy-legacy  myNUMAAffinityPort
```

4. Display the details for your port.

Example

```
$ openstack port show myNUMAAffinityPort -c numa_affinity_policy
```

Sample output

When the extension is loaded, the **Value** column should read, **legacy**, **preferred** or **required**. If the extension has failed to load, **Value** reads **None**:

```
+----------------------+--------+
| Field                | Value  |
+----------------------+--------+
| numa_affinity_policy | legacy |
+----------------------+--------+
```

5. Exit the **openstackclient** pod:

```
$ exit
```

Additional resources

- "Instance PCI NUMA affinity policy" in Flavor metadata in *Configuring the Compute service for instance creation*