Red Hat

OpenShift Container Platform 4.18

OVN-Kubernetes network plugin

In-depth configuration and troubleshooting for the OVN-Kubernetes network plugin in OpenShift Container Platform

Last Updated: 2025-12-17

OpenShift Container Platform 4.18 OVN-Kubernetes network plugin

In-depth configuration and troubleshooting for the OVN-Kubernetes network plugin in OpenShift Container Platform

Legal Notice

Abstract

This document provides information on the architecture, configuration, and troubleshooting of the OVN-Kubernetes network plugin in OpenShift Container Platform.

Table of Contents

CHAPTER 1. ABOUT THE OVN-KUBERNETES NETWORK PLUGIN

The OpenShift Container Platform cluster uses a virtualized network for pod and service networks.

Part of Red Hat OpenShift Networking, the OVN-Kubernetes network plugin is the default network provider for OpenShift Container Platform. OVN-Kubernetes is based on Open Virtual Network (OVN) and provides an overlay-based networking implementation. A cluster that uses the OVN-Kubernetes plugin also runs Open vSwitch (OVS) on each node. OVN configures OVS on each node to implement the declared network configuration.

NOTE

OVN-Kubernetes is the default networking solution for OpenShift Container Platform and single-node OpenShift deployments.

OVN-Kubernetes, which arose from the OVS project, uses many of the same constructs, such as open flow rules, to decide how packets travel through the network. For more information, see the Open Virtual Network website.

OVN-Kubernetes is a series of daemons for OVS that transform virtual network configurations into **OpenFlow** rules. **OpenFlow** is a protocol for communicating with network switches and routers, providing a means for remotely controlling the flow of network traffic on a network device. This means that network administrators can configure, manage, and watch the flow of network traffic.

OVN-Kubernetes provides more of the advanced functionality not available with **OpenFlow**. OVN supports distributed virtual routing, distributed logical switches, access control, Dynamic Host Configuration Protocol (DHCP), and DNS. OVN implements distributed virtual routing within logic flows that equate to open flows. For example, if you have a pod that sends out a DHCP request to the DHCP server on the network, a logic flow rule in the request helps the OVN-Kubernetes handle the packet. This means that the server can respond with gateway, DNS server, IP address, and other information.

OVN-Kubernetes runs a daemon on each node. There are daemon sets for the databases and for the OVN controller that run on every node. The OVN controller programs the Open vSwitch daemon on the nodes to support the following network provider features:

- Egress IPs
- Firewalls
- Hardware offloading
- Hybrid networking
- Internet Protocol Security (IPsec) encryption
- IPv6
- Multicast.
- Network policy and network policy logs
- Routers

1.1. OVN-KUBERNETES PURPOSE

The OVN-Kubernetes network plugin is an open-source, fully-featured Kubernetes CNI plugin that uses Open Virtual Network (OVN) to manage network traffic flows. OVN is a community developed, vendor-agnostic network virtualization solution. The OVN-Kubernetes network plugin uses the following technologies:

- OVN to manage network traffic flows.
- Kubernetes network policy support and logs, including ingress and egress rules.
- The Generic Network Virtualization Encapsulation (Geneve) protocol, rather than Virtual Extensible LAN (VXLAN), to create an overlay network between nodes.

The OVN-Kubernetes network plugin supports the following capabilities:

- Hybrid clusters that can run both Linux and Microsoft Windows workloads. This environment is known as *hybrid networking*.
- Offloading of network data processing from the host central processing unit (CPU) to compatible network cards and data processing units (DPUs). This is known as *hardware offloading*.
- IPv4-primary dual-stack networking on bare-metal, VMware vSphere, IBM Power®, IBM Z®, and Red Hat OpenStack Platform (RHOSP) platforms.
- IPv6 single-stack networking on RHOSP and bare metal platforms.
- IPv6-primary dual-stack networking for a cluster running on a bare-metal, a VMware vSphere, or an RHOSP platform.
- Egress firewall devices and egress IP addresses.
- Egress router devices that operate in redirect mode.
- IPsec encryption of intracluster communications.

Red Hat does not support the following postinstallation configurations that use the OVN-Kubernetes network plugin:

- Configuring the primary network interface, including using the NMState Operator to configure bonding for the interface.
- Configuring a sub-interface or additional network interface on a network device that uses the Open vSwitch (OVS) or an OVN-Kubernetes **br-ex** bridge network.
- Creating additional virtual local area networks (VLANs) on the primary network interface.
- Using the primary network interface, such as **eth0** or **bond0**, that you created for a node during cluster installation to create additional secondary networks.

Red Hat does support the following postinstallation configurations that use the OVN-Kubernetes network plugin:

- Creating additional VLANs from the base physical interface, such as **eth0.100**, where you configured the primary network interface as a VLAN for a node during cluster installation. This works because the Open vSwitch (OVS) bridge attaches to the initial VLAN sub-interface, such as **eth0.100**, leaving the base physical interface available for new configurations.

- Creating an additional OVN secondary network with a **localnet** topology network requires that you define the secondary network in a **NodeNetworkConfigurationPolicy** (NNCP) object. After you create the network, pods or virtual machines (VMs) can then attach to the network. These secondary networks give a dedicated connection to the physical network, which might or might not use VLAN tagging. You cannot access these networks from the host network of a node where the host does not have the required setup, such as the required network settings.

1.2. OVN-KUBERNETES IPV6 AND DUAL-STACK LIMITATIONS

The OVN-Kubernetes network plugin has the following limitations:

- For clusters configured for dual-stack networking, both IPv4 and IPv6 traffic must use the same network interface as the default gateway.
  If this requirement is not met, pods on the host in the **ovnkube-node** daemon set enter the **CrashLoopBackOff** state.

  If you display a pod with a command such as **oc get pod -n openshift-ovn-kubernetes -l app=ovnkube-node -o yaml**, the **status** field has more than one message about the default gateway, as shown in the following output:

  ```
  I1006 16:09:50.985852   60651 helper_linux.go:73] Found default gateway interface br-ex
  192.168.127.1
  I1006 16:09:50.985923   60651 helper_linux.go:73] Found default gateway interface ens4
  fe80::5054:ff:febe:bcd4
  F1006 16:09:50.985939   60651 ovnkube.go:130] multiple gateway interfaces detected: br-ex
  ens4
  ```

  The only resolution is to reconfigure the host networking so that both IP families use the same network interface for the default gateway.

- For clusters configured for dual-stack networking, both the IPv4 and IPv6 routing tables must contain the default gateway.
  If this requirement is not met, pods on the host in the **ovnkube-node** daemon set enter the **CrashLoopBackOff** state.

  If you display a pod with a command such as **oc get pod -n openshift-ovn-kubernetes -l app=ovnkube-node -o yaml**, the **status** field has more than one message about the default gateway, as shown in the following output:

  ```
  I0512 19:07:17.589083  108432 helper_linux.go:74] Found default gateway interface br-ex
  192.168.123.1
  F0512 19:07:17.589141  108432 ovnkube.go:133] failed to get default gateway interface
  ```

  The only resolution is to reconfigure the host networking so that both IP families contain the default gateway.

- If you set the **ipv6.disable** parameter to **1** in the **kernelArgument** section of the **MachineConfig** custom resource (CR) for your cluster, OVN-Kubernetes pods enter a **CrashLoopBackOff** state. Additionally, updating your cluster to a later version of OpenShift Container Platform fails because the Network Operator remains on a **Degraded** state. Red Hat does not support disabling IPv6 adddresses for your cluster so do not set the **ipv6.disable** parameter to **1**.

1.3. SESSION AFFINITY

Session affinity is a feature that applies to Kubernetes **Service** objects. You can use *session affinity* if you want to ensure that each time you connect to a <service_VIP>:<Port>, the traffic is always load balanced to the same back end. For more information, including how to set session affinity based on a client's IP address, see Session affinity.

1.3.1. Stickiness timeout for session affinity

The OVN-Kubernetes network plugin for OpenShift Container Platform calculates the stickiness timeout for a session from a client based on the last packet. For example, if you run a **curl** command 10 times, the sticky session timer starts from the tenth packet not the first. As a result, if the client is continuously contacting the service, then the session never times out. The timeout starts when the service has not received a packet for the amount of time set by the **timeoutSeconds** parameter.

Additional resources

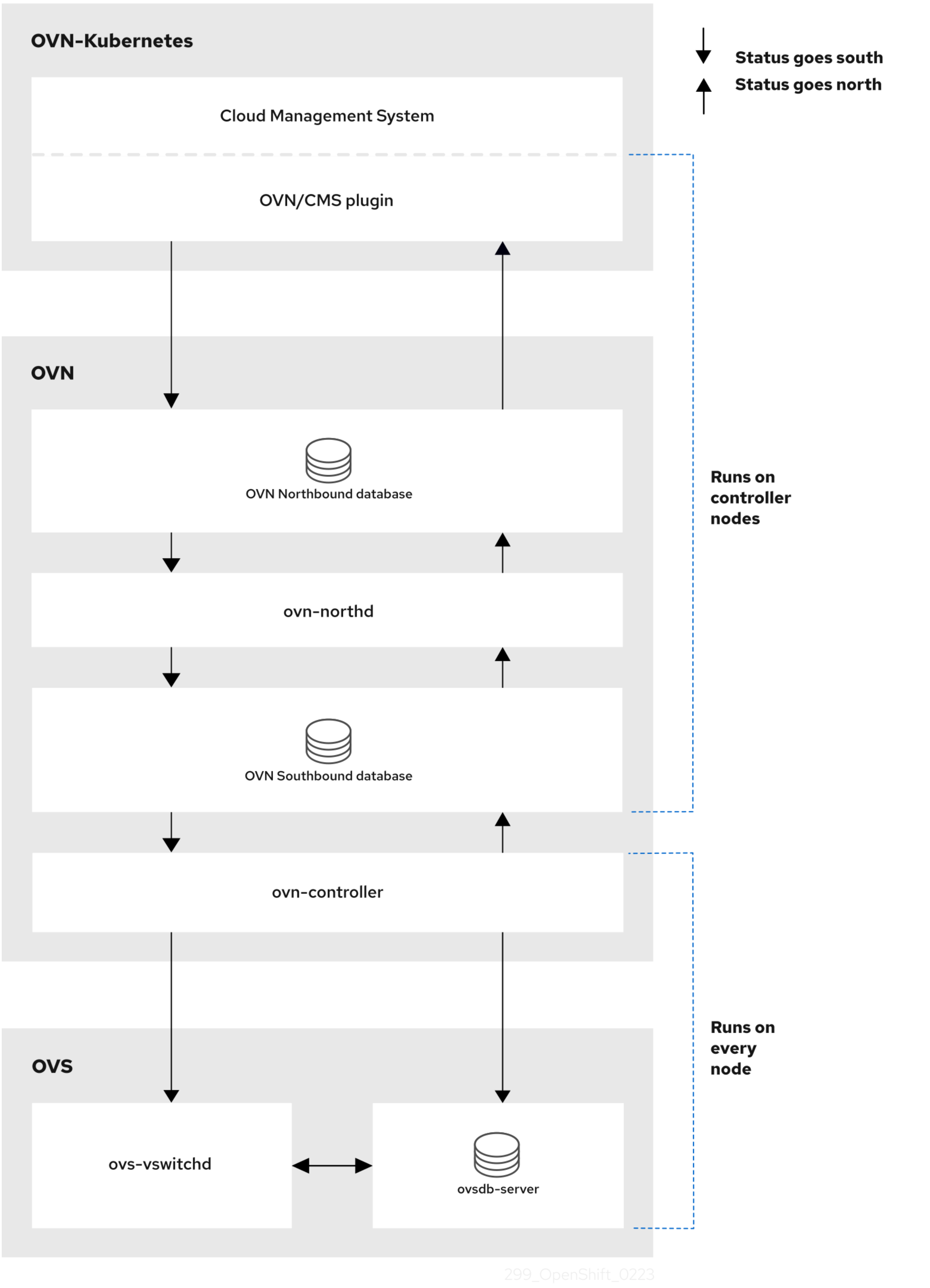
- Configuring an egress firewall for a project
- About network policy
- Logging network policy events
- Enabling multicast for a project
- Configuring IPsec encryption
- Network [operator.openshift.io/v1]

CHAPTER 2. OVN-KUBERNETES ARCHITECTURE

2.1. INTRODUCTION TO OVN-KUBERNETES ARCHITECTURE

The following diagram shows the OVN-Kubernetes architecture.

Figure 2.1. OVK-Kubernetes architecture



The key components are:

- **Cloud Management System (CMS)** - A platform specific client for OVN that provides a CMS specific plugin for OVN integration. The plugin translates the cloud management system's concept of the logical network configuration, stored in the CMS configuration database in a CMS-specific format, into an intermediate representation understood by OVN.
- **OVN Northbound database (nbdb) container** - Stores the logical network configuration passed by the CMS plugin.
- **OVN Southbound database (sbdb) container** - Stores the physical and logical network configuration state for Open vSwitch (OVS) system on each node, including tables that bind them.
- **OVN north daemon (ovn-northd)** - This is the intermediary client between **nbdb** container and **sbdb** container. It translates the logical network configuration in terms of conventional network concepts, taken from the **nbdb** container, into logical data path flows in the **sbdb** container. The container name for **ovn-northd** daemon is **northd** and it runs in the **ovnkube-node** pods.
- **ovn-controller** - This is the OVN agent that interacts with OVS and hypervisors, for any information or update that is needed for **sbdb** container. The **ovn-controller** reads logical flows from the **sbdb** container, translates them into **OpenFlow** flows and sends them to the node's OVS daemon. The container name is **ovn-controller** and it runs in the **ovnkube-node** pods.

The OVN northd, northbound database, and southbound database run on each node in the cluster and mostly contain and process information that is local to that node.

The OVN northbound database has the logical network configuration passed down to it by the cloud management system (CMS). The OVN northbound database contains the current desired state of the network, presented as a collection of logical ports, logical switches, logical routers, and more. The **ovn-northd** (**northd** container) connects to the OVN northbound database and the OVN southbound database. It translates the logical network configuration in terms of conventional network concepts, taken from the OVN northbound database, into logical data path flows in the OVN southbound database.

The OVN southbound database has physical and logical representations of the network and binding tables that link them together. It contains the chassis information of the node and other constructs like remote transit switch ports that are required to connect to the other nodes in the cluster. The OVN southbound database also contains all the logic flows. The logic flows are shared with the **ovn-controller** process that runs on each node and the **ovn-controller** turns those into **OpenFlow** rules to program **Open vSwitch**(OVS).

The Kubernetes control plane nodes contain two **ovnkube-control-plane** pods on separate nodes, which perform the central IP address management (IPAM) allocation for each node in the cluster. At any given time, a single **ovnkube-control-plane** pod is the leader.

2.2. LISTING ALL RESOURCES IN THE OVN-KUBERNETES PROJECT

Finding the resources and containers that run in the OVN-Kubernetes project is important to help you understand the OVN-Kubernetes networking implementation.

Prerequisites

- Access to the cluster as a user with the **cluster-admin** role.
- The OpenShift CLI (**oc**) installed.

Procedure

1. Run the following command to get all resources, endpoints, and **ConfigMaps** in the OVN-Kubernetes project:

```
$ oc get all,ep,cm -n openshift-ovn-kubernetes
```

Example output

```
Warning: apps.openshift.io/v1 DeploymentConfig is deprecated in v4.14+, unavailable in
v4.10000+
NAME                                     READY   STATUS    RESTARTS      AGE
pod/ovnkube-control-plane-65c6f55656-6d55h   2/2     Running   0             114m
pod/ovnkube-control-plane-65c6f55656-fd7vw   2/2     Running   2 (104m ago)   114m
pod/ovnkube-node-bcvts                   8/8     Running   0             113m
pod/ovnkube-node-drgvv                   8/8     Running   0             113m
pod/ovnkube-node-f2pxt                   8/8     Running   0             113m
pod/ovnkube-node-frqsb                   8/8     Running   0             105m
pod/ovnkube-node-lbxkk                   8/8     Running   0             105m
pod/ovnkube-node-tt7bx                   8/8     Running   1 (102m ago)   105m

NAME                             TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)            AGE
service/ovn-kubernetes-control-plane   ClusterIP   None         <none>        9108/TCP
114m
service/ovn-kubernetes-node         ClusterIP   None         <none>        9103/TCP,9105/TCP
114m

NAME                          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE
NODE SELECTOR                 AGE
daemonset.apps/ovnkube-node   6         6         6       6            6
beta.kubernetes.io/os=linux   114m

NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ovnkube-control-plane   3/3     3            3           114m

NAME                                               DESIRED   CURRENT   READY   AGE
replicaset.apps/ovnkube-control-plane-65c6f55656   3         3         3       114m

NAME                                    ENDPOINTS                                          AGE
endpoints/ovn-kubernetes-control-plane   10.0.0.3:9108,10.0.0.4:9108,10.0.0.5:9108
114m
endpoints/ovn-kubernetes-node           10.0.0.3:9105,10.0.0.4:9105,10.0.0.5:9105 + 9 more...
114m

NAME                                 DATA   AGE
configmap/control-plane-status       1      113m
configmap/kube-root-ca.crt           1      114m
configmap/openshift-service-ca.crt   1      114m
configmap/ovn-ca                     1      114m
configmap/ovnkube-config             1      114m
configmap/signer-ca                  1      114m
```

There is one **ovnkube-node** pod for each node in the cluster. The **ovnkube-config** config map has the OpenShift Container Platform OVN-Kubernetes configurations.

2. List all of the containers in the **ovnkube-node** pods by running the following command:

```
$ oc get pods ovnkube-node-bcvts -o jsonpath='{.spec.containers[*].name}' -n openshift-ovn-kubernetes
```

Expected output

```
ovn-controller ovn-acl-logging kube-rbac-proxy-node kube-rbac-proxy-ovn-metrics northd
nbdb sbdb ovnkube-controller
```

The **ovnkube-node** pod is made up of several containers. It is responsible for hosting the northbound database (**nbdb** container), the southbound database (**sbdb** container), the north daemon (**northd** container), **ovn-controller** and the **ovnkube-controller** container. The **ovnkube-controller** container watches for API objects like pods, egress IPs, namespaces, services, endpoints, egress firewall, and network policies. It is also responsible for allocating pod IP from the available subnet pool for that node.

3. List all the containers in the **ovnkube-control-plane** pods by running the following command:

```
$ oc get pods ovnkube-control-plane-65c6f55656-6d55h -o
jsonpath='{.spec.containers[*].name}' -n openshift-ovn-kubernetes
```

Expected output

```
kube-rbac-proxy ovnkube-cluster-manager
```

The **ovnkube-control-plane** pod has a container (**ovnkube-cluster-manager**) that resides on each OpenShift Container Platform node. The **ovnkube-cluster-manager** container allocates pod subnet, transit switch subnet IP and join switch subnet IP to each node in the cluster. The **kube-rbac-proxy** container monitors metrics for the **ovnkube-cluster-manager** container.

2.3. LISTING THE OVN-KUBERNETES NORTHBOUND DATABASE CONTENTS

Each node is controlled by the **ovnkube-controller** container running in the **ovnkube-node** pod on that node. To understand the OVN logical networking entities you need to examine the northbound database that is running as a container inside the **ovnkube-node** pod on that node to see what objects are in the node you wish to see.

Prerequisites

- Access to the cluster as a user with the **cluster-admin** role.
- The OpenShift CLI (**oc**) installed.

PROCEDURE

To run ovn **nbctl** or **sbctl** commands in a cluster you must open a remote shell into the **nbdb** or **sbdb** containers on the relevant node

1. List pods by running the following command:

```
$ oc get po -n openshift-ovn-kubernetes
```

Example output

```
NAME                                     READY   STATUS    RESTARTS     AGE
ovnkube-control-plane-8444dff7f9-4lh9k   2/2     Running   0            27m
ovnkube-control-plane-8444dff7f9-5rjh9   2/2     Running   0            27m
ovnkube-node-55xs2                       8/8     Running   0            26m
ovnkube-node-7r84r                       8/8     Running   0            16m
ovnkube-node-bqq8p                       8/8     Running   0            17m
ovnkube-node-mkj4f                       8/8     Running   0            26m
ovnkube-node-mlr8k                       8/8     Running   0            26m
ovnkube-node-wqn2m                       8/8     Running   0            16m
```

2. Optional: To list the pods with node information, run the following command:

```
$ oc get pods -n openshift-ovn-kubernetes -owide
```

Example output

```
NAME                                     READY   STATUS    RESTARTS     AGE   IP           NODE
NOMINATED NODE   READINESS GATES
ovnkube-control-plane-8444dff7f9-4lh9k   2/2     Running   0            27m   10.0.0.3     ci-ln-
t487nnb-72292-mdcnq-master-1        <none>           <none>
ovnkube-control-plane-8444dff7f9-5rjh9   2/2     Running   0            27m   10.0.0.4     ci-ln-
t487nnb-72292-mdcnq-master-2        <none>           <none>
ovnkube-node-55xs2                       8/8     Running   0            26m   10.0.0.4     ci-ln-t487nnb-
72292-mdcnq-master-2        <none>           <none>
ovnkube-node-7r84r                       8/8     Running   0            17m   10.0.128.3   ci-ln-t487nnb-
72292-mdcnq-worker-b-wbz7z   <none>           <none>
ovnkube-node-bqq8p                       8/8     Running   0            17m   10.0.128.2   ci-ln-
t487nnb-72292-mdcnq-worker-a-lh7ms   <none>           <none>
ovnkube-node-mkj4f                       8/8     Running   0            27m   10.0.0.5     ci-ln-t487nnb-
72292-mdcnq-master-0        <none>           <none>
ovnkube-node-mlr8k                       8/8     Running   0            27m   10.0.0.3     ci-ln-t487nnb-
72292-mdcnq-master-1        <none>           <none>
ovnkube-node-wqn2m                       8/8     Running   0            17m   10.0.128.4   ci-ln-
t487nnb-72292-mdcnq-worker-c-przlm   <none>           <none>
```

3. Navigate into a pod to look at the northbound database by running the following command:

```
$ oc rsh -c nbdb -n openshift-ovn-kubernetes ovnkube-node-55xs2
```

4. Run the following command to show all the objects in the northbound database:

```
$ ovn-nbctl show
```

The output is too long to list here. The list includes the NAT rules, logical switches, load balancers and so on.

You can narrow down and focus on specific components by using some of the following optional commands:

a. Run the following command to show the list of logical routers:

```
$ oc exec -n openshift-ovn-kubernetes -it ovnkube-node-55xs2 \
-c northd -- ovn-nbctl lr-list
```

Example output

```
45339f4f-7d0b-41d0-b5f9-9fca9ce40ce6 (GR_ci-ln-t487nnb-72292-mdcnq-master-2)
96a0a0f0-e7ed-4fec-8393-3195563de1b8 (ovn_cluster_router)
```

NOTE

From this output you can see there is router on each node plus an **ovn_cluster_router**.

b. Run the following command to show the list of logical switches:

```
$ oc exec -n openshift-ovn-kubernetes -it ovnkube-node-55xs2 \
-c nbdb -- ovn-nbctl ls-list
```

Example output

```
bdd7dc3d-d848-4a74-b293-cc15128ea614 (ci-ln-t487nnb-72292-mdcnq-master-2)
b349292d-ee03-4914-935f-1940b6cb91e5 (ext_ci-ln-t487nnb-72292-mdcnq-master-2)
0aac0754-ea32-4e33-b086-35eeabf0a140 (join)
992509d7-2c3f-4432-88db-c179e43592e5 (transit_switch)
```

NOTE

From this output you can see there is an ext switch for each node plus switches with the node name itself and a join switch.

c. Run the following command to show the list of load balancers:

```
$ oc exec -n openshift-ovn-kubernetes -it ovnkube-node-55xs2 \
-c nbdb -- ovn-nbctl lb-list
```

Example output

```
UUID                                    LB                  PROTO      VIP                  IPs
7c84c673-ed2a-4436-9a1f-9bc5dd181eea    Service_default/    tcp        172.30.0.1:443
10.0.0.3:6443,169.254.169.2:6443,10.0.0.5:6443
4d663fd9-ddc8-4271-b333-4c0e279e20bb    Service_default/    tcp        172.30.0.1:443
10.0.0.3:6443,10.0.0.4:6443,10.0.0.5:6443
292eb07f-b82f-4962-868a-4f541d250bca    Service_openshif    tcp
172.30.105.247:443      10.129.0.12:8443
034b5a7f-bb6a-45e9-8e6d-573a82dc5ee3    Service_openshif    tcp
172.30.192.38:443       10.0.0.3:10259,10.0.0.4:10259,10.0.0.5:10259
a68bb53e-be84-48df-bd38-bdd82fcd4026    Service_openshif    tcp
172.30.161.125:8443     10.129.0.32:8443
6cc21b3d-2c54-4c94-8ff5-d8e017269c2e    Service_openshif    tcp        172.30.3.144:443
10.129.0.22:8443
```

```
37996ffd-7268-4862-a27f-61cd62e09c32    Service_openshif    tcp
172.30.181.107:443      10.129.0.18:8443
81d4da3c-f811-411f-ae0c-bc6713d0861d    Service_openshif    tcp
172.30.228.23:443       10.129.0.29:8443
ac5a4f3b-b6ba-4ceb-82d0-d84f2c41306e    Service_openshif    tcp
172.30.14.240:9443      10.129.0.36:9443
c88979fb-1ef5-414b-90ac-43b579351ac9    Service_openshif    tcp
172.30.231.192:9001
10.128.0.5:9001,10.128.2.5:9001,10.129.0.5:9001,10.129.2.4:9001,10.130.0.3:9001,10.13
1.0.3:9001
fcb0a3fb-4a77-4230-a84a-be45dce757e8    Service_openshif    tcp
172.30.189.92:443       10.130.0.17:8440
67ef3e7b-ceb9-4bf0-8d96-b43bde4c9151    Service_openshif    tcp
172.30.67.218:443       10.129.0.9:8443
d0032fba-7d5e-424a-af25-4ab9b5d46e81    Service_openshif    tcp
172.30.102.137:2379     10.0.0.3:2379,10.0.0.4:2379,10.0.0.5:2379
                                                            tcp        172.30.102.137:9979
10.0.0.3:9979,10.0.0.4:9979,10.0.0.5:9979
7361c537-3eec-4e6c-bc0c-0522d182abd4    Service_openshif    tcp
172.30.198.215:9001
10.0.0.3:9001,10.0.0.4:9001,10.0.0.5:9001,10.0.128.2:9001,10.0.128.3:9001,10.0.128.4:9
001
0296c437-1259-410b-a6fd-81c310ad0af5    Service_openshif    tcp
172.30.198.215:9001
10.0.0.3:9001,169.254.169.2:9001,10.0.0.5:9001,10.0.128.2:9001,10.0.128.3:9001,10.0.1
28.4:9001
5d5679f5-45b8-479d-9f7c-08b123c688b8    Service_openshif    tcp
172.30.38.253:17698     10.128.0.52:17698,10.129.0.84:17698,10.130.0.60:17698
2adcbab4-d1c9-447d-9573-b5dc9f2efbfa    Service_openshif    tcp
172.30.148.52:443       10.0.0.4:9202,10.0.0.5:9202
                                                            tcp        172.30.148.52:444
10.0.0.4:9203,10.0.0.5:9203
                                                            tcp        172.30.148.52:445
10.0.0.4:9204,10.0.0.5:9204
                                                            tcp        172.30.148.52:446
10.0.0.4:9205,10.0.0.5:9205
2a33a6d7-af1b-4892-87cc-326a380b809b    Service_openshif    tcp
172.30.67.219:9091      10.129.2.16:9091,10.131.0.16:9091
                                                            tcp        172.30.67.219:9092
10.129.2.16:9092,10.131.0.16:9092
                                                            tcp        172.30.67.219:9093
10.129.2.16:9093,10.131.0.16:9093
                                                            tcp        172.30.67.219:9094
10.129.2.16:9094,10.131.0.16:9094
f56f59d7-231a-4974-99b3-792e2741ec8d    Service_openshif    tcp
172.30.89.212:443       10.128.0.41:8443,10.129.0.68:8443,10.130.0.44:8443
08c2c6d7-d217-4b96-b5d8-c80c4e258116    Service_openshif    tcp
172.30.102.137:2379     10.0.0.3:2379,169.254.169.2:2379,10.0.0.5:2379
                                                            tcp        172.30.102.137:9979
10.0.0.3:9979,169.254.169.2:9979,10.0.0.5:9979
60a69c56-fc6a-4de6-bd88-3f2af5ba5665    Service_openshif    tcp
172.30.10.193:443       10.129.0.25:8443
ab1ef694-0826-4671-a22c-565fc2d282ec    Service_openshif    tcp
172.30.196.123:443      10.128.0.33:8443,10.129.0.64:8443,10.130.0.37:8443
b1fb34d3-0944-4770-9ee3-2683e7a630e2    Service_openshif    tcp
172.30.158.93:8443      10.129.0.13:8443
```

```
95811c11-56e2-4877-be1e-c78ccb3a82a9    Service_openshif    tcp
172.30.46.85:9001       10.130.0.16:9001
4baba1d1-b873-4535-884c-3f6fc07a50fd    Service_openshif    tcp       172.30.28.87:443
10.129.0.26:8443
6c2e1c90-f0ca-484e-8a8e-40e71442110a    Service_openshif    udp       172.30.0.10:53
10.128.0.13:5353,10.128.2.6:5353,10.129.0.39:5353,10.129.2.6:5353,10.130.0.11:5353,1
0.131.0.9:5353
```

NOTE

From this truncated output you can see there are many OVN-Kubernetes load balancers. Load balancers in OVN-Kubernetes are representations of services.

5. Run the following command to display the options available with the command **ovn-nbctl**:

```
$ oc exec -n openshift-ovn-kubernetes -it ovnkube-node-55xs2 \
-c nbdb ovn-nbctl --help
```

2.4. COMMAND-LINE ARGUMENTS FOR OVN-NBCTL TO EXAMINE NORTHBOUND DATABASE CONTENTS

The following table describes the command-line arguments that can be used with **ovn-nbctl** to examine the contents of the northbound database.

NOTE

Open a remote shell in the pod you want to view the contents of and then run the **ovn-nbctl** commands.

Table 2.1. Command-line arguments to examine northbound database contents

Argument	Description
ovn-nbctl show	An overview of the northbound database contents as seen from a specific node.
ovn-nbctl show <switch_or_router>	Show the details associated with the specified switch or router.
ovn-nbctl lr-list	Show the logical routers.
ovn-nbctl lrp-list <router>	Using the router information from **ovn-nbctl lr-list** to show the router ports.
ovn-nbctl lr-nat-list <router>	Show network address translation details for the specified router.
ovn-nbctl ls-list	Show the logical switches

Argument	Description
ovn-nbctl lsp-list <switch>	Using the switch information from **ovn-nbctl ls-list** to show the switch port.
ovn-nbctl lsp-get-type <port>	Get the type for the logical port.
ovn-nbctl lb-list	Show the load balancers.

2.5. LISTING THE OVN-KUBERNETES SOUTHBOUND DATABASE CONTENTS

Each node is controlled by the **ovnkube-controller** container running in the **ovnkube-node** pod on that node. To understand the OVN logical networking entities you need to examine the northbound database that is running as a container inside the **ovnkube-node** pod on that node to see what objects are in the node you wish to see.

Prerequisites

- Access to the cluster as a user with the **cluster-admin** role.
- The OpenShift CLI (**oc**) installed.

PROCEDURE

To run ovn **nbctl** or **sbctl** commands in a cluster you must open a remote shell into the **nbdb** or **sbdb** containers on the relevant node

1. List the pods by running the following command:

```
$ oc get po -n openshift-ovn-kubernetes
```

Example output

```
NAME                                     READY   STATUS    RESTARTS   AGE
ovnkube-control-plane-8444dff7f9-4lh9k   2/2     Running   0          27m
ovnkube-control-plane-8444dff7f9-5rjh9   2/2     Running   0          27m
ovnkube-node-55xs2                       8/8     Running   0          26m
ovnkube-node-7r84r                       8/8     Running   0          16m
ovnkube-node-bqq8p                       8/8     Running   0          17m
ovnkube-node-mkj4f                       8/8     Running   0          26m
ovnkube-node-mlr8k                       8/8     Running   0          26m
ovnkube-node-wqn2m                       8/8     Running   0          16m
```

2. Optional: To list the pods with node information, run the following command:

```
$ oc get pods -n openshift-ovn-kubernetes -owide
```

Example output

```
NAME                                READY   STATUS    RESTARTS     AGE   IP          NODE
NOMINATED NODE   READINESS GATES
ovnkube-control-plane-8444dff7f9-4lh9k   2/2     Running   0            27m   10.0.0.3     ci-ln-
t487nnb-72292-mdcnq-master-1         <none>           <none>
ovnkube-control-plane-8444dff7f9-5rjh9   2/2     Running   0            27m   10.0.0.4     ci-ln-
t487nnb-72292-mdcnq-master-2         <none>           <none>
ovnkube-node-55xs2                      8/8     Running   0            26m   10.0.0.4     ci-ln-t487nnb-
72292-mdcnq-master-2         <none>           <none>
ovnkube-node-7r84r                      8/8     Running   0            17m   10.0.128.3   ci-ln-t487nnb-
72292-mdcnq-worker-b-wbz7z   <none>           <none>
ovnkube-node-bqq8p                      8/8     Running   0            17m   10.0.128.2   ci-ln-
t487nnb-72292-mdcnq-worker-a-lh7ms   <none>           <none>
ovnkube-node-mkj4f                      8/8     Running   0            27m   10.0.0.5     ci-ln-t487nnb-
72292-mdcnq-master-0         <none>           <none>
ovnkube-node-mlr8k                      8/8     Running   0            27m   10.0.0.3     ci-ln-t487nnb-
72292-mdcnq-master-1         <none>           <none>
ovnkube-node-wqn2m                      8/8     Running   0            17m   10.0.128.4   ci-ln-
t487nnb-72292-mdcnq-worker-c-przlm   <none>           <none>
```

3. Navigate into a pod to look at the southbound database:

```
$ oc rsh -c sbdb -n openshift-ovn-kubernetes ovnkube-node-55xs2
```

4. Run the following command to show all the objects in the southbound database:

```
$ ovn-sbctl show
```

Example output

```
Chassis "5db31703-35e9-413b-8cdf-69e7eecb41f7"
    hostname: ci-ln-9gp362t-72292-v2p94-worker-a-8bmwz
    Encap geneve
        ip: "10.0.128.4"
        options: {csum="true"}
    Port_Binding tstor-ci-ln-9gp362t-72292-v2p94-worker-a-8bmwz
Chassis "070debed-99b7-4bce-b17d-17e720b7f8bc"
    hostname: ci-ln-9gp362t-72292-v2p94-worker-b-svmp6
    Encap geneve
        ip: "10.0.128.2"
        options: {csum="true"}
    Port_Binding k8s-ci-ln-9gp362t-72292-v2p94-worker-b-svmp6
    Port_Binding rtoe-GR_ci-ln-9gp362t-72292-v2p94-worker-b-svmp6
    Port_Binding openshift-monitoring_alertmanager-main-1
    Port_Binding rtoj-GR_ci-ln-9gp362t-72292-v2p94-worker-b-svmp6
    Port_Binding etor-GR_ci-ln-9gp362t-72292-v2p94-worker-b-svmp6
    Port_Binding cr-rtos-ci-ln-9gp362t-72292-v2p94-worker-b-svmp6
    Port_Binding openshift-e2e-loki_loki-promtail-qcrcz
    Port_Binding jtor-GR_ci-ln-9gp362t-72292-v2p94-worker-b-svmp6
    Port_Binding openshift-multus_network-metrics-daemon-mkd4t
    Port_Binding openshift-ingress-canary_ingress-canary-xtvj4
    Port_Binding openshift-ingress_router-default-6c76cbc498-pvlqk
    Port_Binding openshift-dns_dns-default-zz582
```

```
        Port_Binding openshift-monitoring_thanos-querier-57585899f5-lbf4f
        Port_Binding openshift-network-diagnostics_network-check-target-tn228
        Port_Binding openshift-monitoring_prometheus-k8s-0
        Port_Binding openshift-image-registry_image-registry-68899bd877-xqxjj
    Chassis "179ba069-0af1-401c-b044-e5ba90f60fea"
        hostname: ci-ln-9gp362t-72292-v2p94-master-0
        Encap geneve
            ip: "10.0.0.5"
            options: {csum="true"}
        Port_Binding tstor-ci-ln-9gp362t-72292-v2p94-master-0
    Chassis "68c954f2-5a76-47be-9e84-1cb13bd9dab9"
        hostname: ci-ln-9gp362t-72292-v2p94-worker-c-mjf9w
        Encap geneve
            ip: "10.0.128.3"
            options: {csum="true"}
        Port_Binding tstor-ci-ln-9gp362t-72292-v2p94-worker-c-mjf9w
    Chassis "2de65d9e-9abf-4b6e-a51d-a1e038b4d8af"
        hostname: ci-ln-9gp362t-72292-v2p94-master-2
        Encap geneve
            ip: "10.0.0.4"
            options: {csum="true"}
        Port_Binding tstor-ci-ln-9gp362t-72292-v2p94-master-2
    Chassis "1d371cb8-5e21-44fd-9025-c4b162cc4247"
        hostname: ci-ln-9gp362t-72292-v2p94-master-1
        Encap geneve
            ip: "10.0.0.3"
            options: {csum="true"}
        Port_Binding tstor-ci-ln-9gp362t-72292-v2p94-master-1
```

This detailed output shows the chassis and the ports that are attached to the chassis which in this case are all of the router ports and anything that runs like host networking. Any pods communicate out to the wider network using source network address translation (SNAT). Their IP address is translated into the IP address of the node that the pod is running on and then sent out into the network.

In addition to the chassis information the southbound database has all the logic flows and those logic flows are then sent to the **ovn-controller** running on each of the nodes. The **ovn-controller** translates the logic flows into open flow rules and ultimately programs **OpenvSwitch** so that your pods can then follow open flow rules and make it out of the network.

5. Run the following command to display the options available with the command **ovn-sbctl**:

```
$ oc exec -n openshift-ovn-kubernetes -it ovnkube-node-55xs2 \
-c sbdb ovn-sbctl --help
```

2.6. COMMAND-LINE ARGUMENTS FOR OVN-SBCTL TO EXAMINE SOUTHBOUND DATABASE CONTENTS

The following table describes the command-line arguments that can be used with **ovn-sbctl** to examine the contents of the southbound database.

NOTE

Open a remote shell in the pod you wish to view the contents of and then run the **ovn-sbctl** commands.

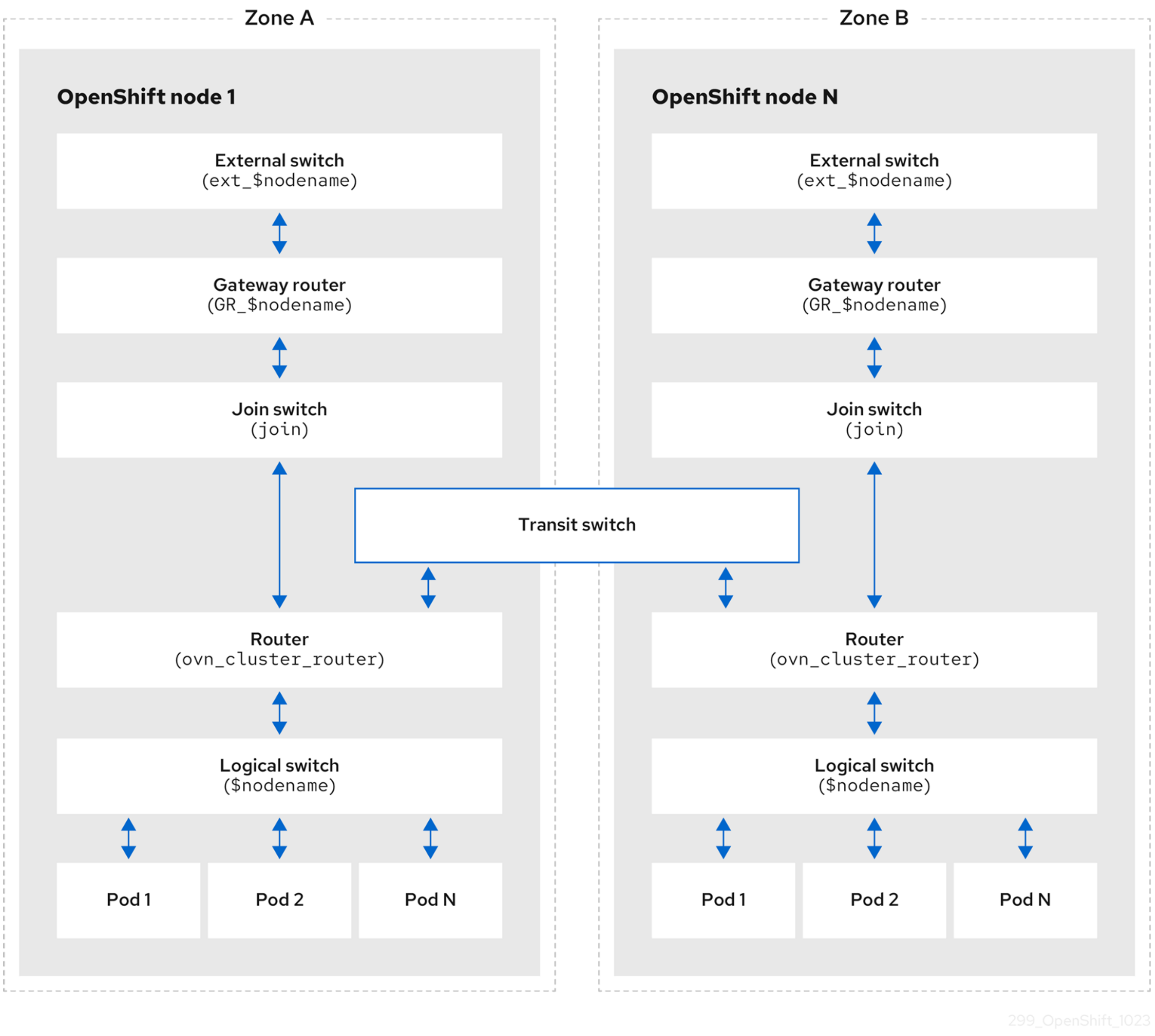
Table 2.2. Command-line arguments to examine southbound database contents

Argument	Description
ovn-sbctl show	An overview of the southbound database contents as seen from a specific node.
ovn-sbctl list Port_Binding <port>	List the contents of southbound database for a the specified port .
ovn-sbctl dump-flows	List the logical flows.

2.7. OVN-KUBERNETES LOGICAL ARCHITECTURE

OVN is a network virtualization solution. It creates logical switches and routers. These switches and routers are interconnected to create any network topologies. When you run **ovnkube-trace** with the log level set to 2 or 5 the OVN-Kubernetes logical components are exposed. The following diagram shows how the routers and switches are connected in OpenShift Container Platform.

Figure 2.2. OVN-Kubernetes router and switch components



The key components involved in packet processing are:

Gateway routers

Gateway routers sometimes called L3 gateway routers, are typically used between the distributed routers and the physical network. Gateway routers including their logical patch ports are bound to a physical location (not distributed), or chassis. The patch ports on this router are known as l3gateway ports in the ovn-southbound database (**ovn-sbdb**).

Distributed logical routers

Distributed logical routers and the logical switches behind them, to which virtual machines and containers attach, effectively reside on each hypervisor.

Join local switch

Join local switches are used to connect the distributed router and gateway routers. It reduces the number of IP addresses needed on the distributed router.

Logical switches with patch ports

Logical switches with patch ports are used to virtualize the network stack. They connect remote logical ports through tunnels.

Logical switches with localnet ports

Logical switches with localnet ports are used to connect OVN to the physical network. They connect remote logical ports by bridging the packets to directly connected physical L2 segments using localnet ports.

Patch ports

Patch ports represent connectivity between logical switches and logical routers and between peer logical routers. A single connection has a pair of patch ports at each such point of connectivity, one on each side.

l3gateway ports

l3gateway ports are the port binding entries in the **ovn-sbdb** for logical patch ports used in the gateway routers. They are called l3gateway ports rather than patch ports just to portray the fact that these ports are bound to a chassis just like the gateway router itself.

localnet ports

localnet ports are present on the bridged logical switches that allows a connection to a locally accessible network from each **ovn-controller** instance. This helps model the direct connectivity to the physical network from the logical switches. A logical switch can only have a single localnet port attached to it.

2.7.1. Installing network-tools on local host

Install **network-tools** on your local host to make a collection of tools available for debugging OpenShift Container Platform cluster network issues.

Procedure

1. Clone the **network-tools** repository onto your workstation with the following command:

```
$ git clone git@github.com:openshift/network-tools.git
```

2. Change into the directory for the repository you just cloned:

```
$ cd network-tools
```

3. Optional: List all available commands:

```
$ ./debug-scripts/network-tools -h
```

2.7.2. Running network-tools

Get information about the logical switches and routers by running **network-tools**.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster as a user with **cluster-admin** privileges.
- You have installed **network-tools** on local host.

Procedure

1. List the routers by running the following command:

```
$ ./debug-scripts/network-tools ovn-db-run-command ovn-nbctl lr-list
```

Example output

```
944a7b53-7948-4ad2-a494-82b55eeccf87 (GR_ci-ln-54932yb-72292-kd676-worker-c-rzj99)
84bd4a4c-4b0b-4a47-b0cf-a2c32709fc53 (ovn_cluster_router)
```

2. List the localnet ports by running the following command:

```
$ ./debug-scripts/network-tools ovn-db-run-command \
ovn-sbctl find Port_Binding type=localnet
```

Example output

```
_uuid               : d05298f5-805b-4838-9224-1211afc2f199
additional_chassis  : []
additional_encap    : []
chassis             : []
datapath            : f3c2c959-743b-4037-854d-26627902597c
encap               : []
external_ids        : {}
gateway_chassis     : []
ha_chassis_group    : []
logical_port        : br-ex_ci-ln-54932yb-72292-kd676-worker-c-rzj99
mac                 : [unknown]
mirror_rules        : []
nat_addresses       : []
options             : {network_name=physnet}
parent_port         : []
port_security       : []
requested_additional_chassis: []
requested_chassis   : []
tag                 : []
tunnel_key          : 2
type                : localnet
up                  : false
virtual_parent      : []

[...]
```

3. List the **l3gateway** ports by running the following command:

```
$ ./debug-scripts/network-tools ovn-db-run-command \
ovn-sbctl find Port_Binding type=l3gateway
```

Example output

```
_uuid               : 5207a1f3-1cf3-42f1-83e9-387bbb06b03c
additional_chassis  : []
additional_encap    : []
chassis             : ca6eb600-3a10-4372-a83e-e0d957c4cd92
datapath            : f3c2c959-743b-4037-854d-26627902597c
encap               : []
```

```
external_ids        : {}
gateway_chassis     : []
ha_chassis_group    : []
logical_port        : etor-GR_ci-ln-54932yb-72292-kd676-worker-c-rzj99
mac                 : ["42:01:0a:00:80:04"]
mirror_rules        : []
nat_addresses       : ["42:01:0a:00:80:04 10.0.128.4"]
options             : {l3gateway-chassis="84737c36-b383-4c83-92c5-2bd5b3c7e772", peer=rtoe-
GR_ci-ln-54932yb-72292-kd676-worker-c-rzj99}
parent_port         : []
port_security       : []
requested_additional_chassis: []
requested_chassis   : []
tag                 : []
tunnel_key          : 1
type                : l3gateway
up                  : true
virtual_parent      : []

_uuid               : 6088d647-84f2-43f2-b53f-c9d379042679
additional_chassis  : []
additional_encap    : []
chassis             : ca6eb600-3a10-4372-a83e-e0d957c4cd92
datapath            : dc9cea00-d94a-41b8-bdb0-89d42d13aa2e
encap               : []
external_ids        : {}
gateway_chassis     : []
ha_chassis_group    : []
logical_port        : jtor-GR_ci-ln-54932yb-72292-kd676-worker-c-rzj99
mac                 : [router]
mirror_rules        : []
nat_addresses       : []
options             : {l3gateway-chassis="84737c36-b383-4c83-92c5-2bd5b3c7e772", peer=rtoj-
GR_ci-ln-54932yb-72292-kd676-worker-c-rzj99}
parent_port         : []
port_security       : []
requested_additional_chassis: []
requested_chassis   : []
tag                 : []
tunnel_key          : 2
type                : l3gateway
up                  : true
virtual_parent      : []

[...]
```

4. List the patch ports by running the following command:

```
$ ./debug-scripts/network-tools ovn-db-run-command \
ovn-sbctl find Port_Binding type=patch
```

Example output

```
_uuid               : 785fb8b6-ee5a-4792-a415-5b1cb855dac2
additional_chassis  : []
```

```
additional_encap    : []
chassis             : []
datapath            : f1ddd1cc-dc0d-43b4-90ca-12651305acec
encap               : []
external_ids        : {}
gateway_chassis     : []
ha_chassis_group    : []
logical_port        : stor-ci-ln-54932yb-72292-kd676-worker-c-rzj99
mac                 : [router]
mirror_rules        : []
nat_addresses       : ["0a:58:0a:80:02:01 10.128.2.1 is_chassis_resident(\"cr-rtos-ci-ln-
54932yb-72292-kd676-worker-c-rzj99\")"]
options             : {peer=rtos-ci-ln-54932yb-72292-kd676-worker-c-rzj99}
parent_port         : []
port_security       : []
requested_additional_chassis: []
requested_chassis   : []
tag                 : []
tunnel_key          : 1
type                : patch
up                  : false
virtual_parent      : []

_uuid               : c01ff587-21a5-40b4-8244-4cd0425e5d9a
additional_chassis  : []
additional_encap    : []
chassis             : []
datapath            : f6795586-bf92-4f84-9222-efe4ac6a7734
encap               : []
external_ids        : {}
gateway_chassis     : []
ha_chassis_group    : []
logical_port        : rtoj-ovn_cluster_router
mac                 : ["0a:58:64:40:00:01 100.64.0.1/16"]
mirror_rules        : []
nat_addresses       : []
options             : {peer=jtor-ovn_cluster_router}
parent_port         : []
port_security       : []
requested_additional_chassis: []
requested_chassis   : []
tag                 : []
tunnel_key          : 1
type                : patch
up                  : false
virtual_parent      : []
[...]
```

2.8. ADDITIONAL RESOURCES

- Tracing Openflow with ovnkube-trace
- OVN architecture
- ovn-nbctl linux manual page

- ovn-sbctl linux manual page

CHAPTER 3. TROUBLESHOOTING OVN-KUBERNETES

OVN-Kubernetes has many sources of built-in health checks and logs. Follow the instructions in these sections to examine your cluster. If a support case is necessary, follow the support guide to collect additional information through a **must-gather**. Only use the **-- gather_network_logs** when instructed by support.

3.1. MONITORING OVN-KUBERNETES HEALTH BY USING READINESS PROBES

The **ovnkube-control-plane** and **ovnkube-node** pods have containers configured with readiness probes.

Prerequisites

- Access to the OpenShift CLI (**oc**).
- You have access to the cluster with **cluster-admin** privileges.
- You have installed **jq**.

Procedure

1. Review the details of the **ovnkube-node** readiness probe by running the following command:

   ```
   $ oc get pods -n openshift-ovn-kubernetes -l app=ovnkube-node \
   -o json | jq '.items[0].spec.containers[] | .name,.readinessProbe'
   ```

   The readiness probe for the northbound and southbound database containers in the **ovnkube-node** pod checks for the health of the databases and the **ovnkube-controller** container.

   The **ovnkube-controller** container in the **ovnkube-node** pod has a readiness probe to verify the presence of the OVN-Kubernetes CNI configuration file, the absence of which would indicate that the pod is not running or is not ready to accept requests to configure pods.

2. Show all events including the probe failures, for the namespace by using the following command:

   ```
   $ oc get events -n openshift-ovn-kubernetes
   ```

3. Show the events for just a specific pod:

   ```
   $ oc describe pod ovnkube-node-9lqfk -n openshift-ovn-kubernetes
   ```

4. Show the messages and statuses from the cluster network operator:

   ```
   $ oc get co/network -o json | jq '.status.conditions[]'
   ```

5. Show the **ready** status of each container in **ovnkube-node** pods by running the following script:

   ```
   $ for p in $(oc get pods --selector app=ovnkube-node -n openshift-ovn-kubernetes \
   -o jsonpath='{range.items[*]}{" "}{.metadata.name}'); do echo === $p ===;  \
   ```

```
oc get pods -n openshift-ovn-kubernetes $p -o json | jq '.status.containerStatuses[] | .name,
.ready'; \
done
```

NOTE

The expectation is all container statuses are reporting as **true**. Failure of a readiness probe sets the status to **false**.

Additional resources

- Monitoring application health by using health checks

3.2. VIEWING OVN-KUBERNETES ALERTS IN THE CONSOLE

The Alerting UI provides detailed information about alerts and their governing alerting rules and silences.

Prerequisites

- You have access to the cluster as a developer or as a user with view permissions for the project that you are viewing metrics for.

Procedure (UI)

1. In the **Administrator** perspective, select **Observe** → **Alerting**. The three main pages in the Alerting UI in this perspective are the **Alerts**, **Silences**, and **Alerting Rules** pages.
2. View the rules for OVN-Kubernetes alerts by selecting **Observe** → **Alerting** → **Alerting Rules**.

3.3. VIEWING OVN-KUBERNETES ALERTS IN THE CLI

You can get information about alerts and their governing alerting rules and silences from the command line.

Prerequisites

- Access to the cluster as a user with the **cluster-admin** role.
- The OpenShift CLI (**oc**) installed.
- You have installed **jq**.

Procedure

1. View active or firing alerts by running the following commands.
   a. Set the alert manager route environment variable by running the following command:

   ```
   $ ALERT_MANAGER=$(oc get route alertmanager-main -n openshift-monitoring \
   -o jsonpath='{@.spec.host}')
   ```

   b. Issue a **curl** request to the alert manager route API by running the following command, replacing **$ALERT_MANAGER** with the URL of your **Alertmanager** instance:

```
$ curl -s -k -H "Authorization: Bearer $(oc create token prometheus-k8s -n openshift-
monitoring)" https://$ALERT_MANAGER/api/v1/alerts | jq '.data[] | "\(.labels.severity) \
(.labels.alertname) \(.labels.pod) \(.labels.container) \(.labels.endpoint) \
(.labels.instance)"'
```

2. View alerting rules by running the following command:

```
$ oc -n openshift-monitoring exec -c prometheus prometheus-k8s-0 -- curl -s
'http://localhost:9090/api/v1/rules' | jq '.data.groups[].rules[] | select(((.name|contains("ovn"))
or (.name|contains("OVN")) or (.name|contains("Ovn")) or (.name|contains("North")) or
(.name|contains("South"))) and .type=="alerting")'
```

3.4. VIEWING THE OVN-KUBERNETES LOGS USING THE CLI

You can view the logs for each of the pods in the **ovnkube-master** and **ovnkube-node** pods using the OpenShift CLI (**oc**).

Prerequisites

- Access to the cluster as a user with the **cluster-admin** role.
- Access to the OpenShift CLI (**oc**).
- You have installed **jq**.

Procedure

1. View the log for a specific pod:

```
$ oc logs -f <pod_name> -c <container_name> -n <namespace>
```

where:

-f

Optional: Specifies that the output follows what is being written into the logs.

<pod_name>

Specifies the name of the pod.

<container_name>

Optional: Specifies the name of a container. When a pod has more than one container, you must specify the container name.

<namespace>

Specify the namespace the pod is running in.

For example:

```
$ oc logs ovnkube-node-5dx44 -n openshift-ovn-kubernetes
```

```
$ oc logs -f ovnkube-node-5dx44 -c ovnkube-controller -n openshift-ovn-kubernetes
```

The contents of log files are printed out.

2. Examine the most recent entries in all the containers in the **ovnkube-node** pods:

```
$ for p in $(oc get pods --selector app=ovnkube-node -n openshift-ovn-kubernetes \
-o jsonpath='{range.items[*]}{" "}{.metadata.name}'); \
do echo === $p ===; for container in $(oc get pods -n openshift-ovn-kubernetes $p \
-o json | jq -r '.status.containerStatuses[] | .name');do echo ---$container---; \
oc logs -c $container $p -n openshift-ovn-kubernetes --tail=5; done; done
```

3. View the last 5 lines of every log in every container in an **ovnkube-node** pod using the following command:

```
$ oc logs -l app=ovnkube-node -n openshift-ovn-kubernetes --all-containers --tail 5
```

3.5. VIEWING THE OVN-KUBERNETES LOGS USING THE WEB CONSOLE

You can view the logs for each of the pods in the **ovnkube-master** and **ovnkube-node** pods in the web console.

Prerequisites

- Access to the OpenShift CLI (**oc**).

Procedure

1. In the OpenShift Container Platform console, navigate to **Workloads → Pods** or navigate to the pod through the resource you want to investigate.
2. Select the **openshift-ovn-kubernetes** project from the drop-down menu.
3. Click the name of the pod you want to investigate.
4. Click **Logs**. By default for the **ovnkube-master** the logs associated with the **northd** container are displayed.
5. Use the down-down menu to select logs for each container in turn.

3.5.1. Changing the OVN-Kubernetes log levels

The default log level for OVN-Kubernetes is 4. To debug OVN-Kubernetes, set the log level to 5. Follow this procedure to increase the log level of the OVN-Kubernetes to help you debug an issue.

Prerequisites

- You have access to the cluster with **cluster-admin** privileges.
- You have access to the OpenShift Container Platform web console.

Procedure

1. Run the following command to get detailed information for all pods in the OVN-Kubernetes project:

```
$ oc get po -o wide -n openshift-ovn-kubernetes
```

Example output

```
NAME                                   READY   STATUS    RESTARTS       AGE    IP           NODE
NOMINATED NODE   READINESS GATES
ovnkube-control-plane-65497d4548-9ptdr   2/2     Running   2 (128m ago)   147m   10.0.0.3
ci-ln-3njdr9b-72292-5nwkp-master-0         <none>           <none>
ovnkube-control-plane-65497d4548-j6zfk   2/2     Running   0              147m   10.0.0.5     ci-
ln-3njdr9b-72292-5nwkp-master-2         <none>           <none>
ovnkube-node-5dx44                       8/8     Running   0              146m   10.0.0.3     ci-ln-
3njdr9b-72292-5nwkp-master-0         <none>           <none>
ovnkube-node-dpfn4                       8/8     Running   0              146m   10.0.0.4     ci-ln-3njdr9b-
72292-5nwkp-master-1         <none>           <none>
ovnkube-node-kwc9l                       8/8     Running   0              134m   10.0.128.2   ci-ln-
3njdr9b-72292-5nwkp-worker-a-2fjcj   <none>           <none>
ovnkube-node-mcrhl                       8/8     Running   0              134m   10.0.128.4   ci-ln-
3njdr9b-72292-5nwkp-worker-c-v9x5v   <none>           <none>
ovnkube-node-nsct4                       8/8     Running   0              146m   10.0.0.5     ci-ln-3njdr9b-
72292-5nwkp-master-2         <none>           <none>
ovnkube-node-zrj9f                       8/8     Running   0              134m   10.0.128.3   ci-ln-3njdr9b-
72292-5nwkp-worker-b-v78h7   <none>           <none>
```

2. Create a **ConfigMap** file similar to the following example and use a filename such as **env-overrides.yaml**:

Example ConfigMap file

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: env-overrides
  namespace: openshift-ovn-kubernetes
data:
  ci-ln-3njdr9b-72292-5nwkp-master-0: | (1)
    # This sets the log level for the ovn-kubernetes node process:
    OVN_KUBE_LOG_LEVEL=5
    # You might also/instead want to enable debug logging for ovn-controller:
    OVN_LOG_LEVEL=dbg
  ci-ln-3njdr9b-72292-5nwkp-master-2: |
    # This sets the log level for the ovn-kubernetes node process:
    OVN_KUBE_LOG_LEVEL=5
    # You might also/instead want to enable debug logging for ovn-controller:
    OVN_LOG_LEVEL=dbg
  _master: | (2)
    # This sets the log level for the ovn-kubernetes master process as well as the ovn-
dbchecker:
    OVN_KUBE_LOG_LEVEL=5
    # You might also/instead want to enable debug logging for northd, nbdb and sbdb on all
masters:
    OVN_LOG_LEVEL=dbg
```

(1) Specify the name of the node you want to set the debug log level on.

2 Specify **_master** to set the log levels of **ovnkube-master** components.

3. Apply the **ConfigMap** file by using the following command:

```
$ oc apply -n openshift-ovn-kubernetes -f env-overrides.yaml
```

Example output

```
configmap/env-overrides.yaml created
```

4. Restart the **ovnkube** pods to apply the new log level by using the following commands:

```
$ oc delete pod -n openshift-ovn-kubernetes \
--field-selector spec.nodeName=ci-ln-3njdr9b-72292-5nwkp-master-0 -l app=ovnkube-node
```

```
$ oc delete pod -n openshift-ovn-kubernetes \
--field-selector spec.nodeName=ci-ln-3njdr9b-72292-5nwkp-master-2 -l app=ovnkube-node
```

```
$ oc delete pod -n openshift-ovn-kubernetes -l app=ovnkube-node
```

5. To verify that the `ConfigMap` file has been applied to all nodes for a specific pod, run the following command:

```
$ oc logs -n openshift-ovn-kubernetes --all-containers --prefix ovnkube-node-<xxxx> | grep -
E -m 10 '(Logging config:|vconsole|DBG)'
```

where:

<XXXX>

Specifies the random sequence of letters for a pod from the previous step.

Example output

```
[pod/ovnkube-node-2cpjc/sbdb] + exec /usr/share/ovn/scripts/ovn-ctl --no-monitor '--ovn-
sb-log=-vconsole:info -vfile:off -vPATTERN:console:%D{%Y-%m-
%dT%H:%M:%S.###Z}|%05N|%c%T|%p|%m' run_sb_ovsdb
[pod/ovnkube-node-2cpjc/ovnkube-controller] I1012 14:39:59.984506   35767
config.go:2247] Logging config: {File: CNIFile:/var/log/ovn-kubernetes/ovn-k8s-cni-
overlay.log LibovsdbFile:/var/log/ovnkube/libovsdb.log Level:5 LogFileMaxSize:100
LogFileMaxBackups:5 LogFileMaxAge:0 ACLLoggingRateLimit:20}
[pod/ovnkube-node-2cpjc/northd] + exec ovn-northd --no-chdir -vconsole:info -vfile:off '-
vPATTERN:console:%D{%Y-%m-%dT%H:%M:%S.###Z}|%05N|%c%T|%p|%m' --pidfile
/var/run/ovn/ovn-northd.pid --n-threads=1
[pod/ovnkube-node-2cpjc/nbdb] + exec /usr/share/ovn/scripts/ovn-ctl --no-monitor '--ovn-
nb-log=-vconsole:info -vfile:off -vPATTERN:console:%D{%Y-%m-
%dT%H:%M:%S.###Z}|%05N|%c%T|%p|%m' run_nb_ovsdb
[pod/ovnkube-node-2cpjc/ovn-controller] 2023-10-
12T14:39:54.552Z|00002|hmap|DBG|lib/shash.c:114: 1 bucket with 6+ nodes, including 1
bucket with 6 nodes (32 nodes total across 32 buckets)
[pod/ovnkube-node-2cpjc/ovn-controller] 2023-10-
12T14:39:54.553Z|00003|hmap|DBG|lib/shash.c:114: 1 bucket with 6+ nodes, including 1
bucket with 6 nodes (64 nodes total across 64 buckets)
```

```
[pod/ovnkube-node-2cpjc/ovn-controller] 2023-10-
12T14:39:54.553Z|00004|hmap|DBG|lib/shash.c:114: 1 bucket with 6+ nodes, including 1
bucket with 7 nodes (32 nodes total across 32 buckets)
[pod/ovnkube-node-2cpjc/ovn-controller] 2023-10-
12T14:39:54.553Z|00005|reconnect|DBG|unix:/var/run/openvswitch/db.sock: entering
BACKOFF
[pod/ovnkube-node-2cpjc/ovn-controller] 2023-10-
12T14:39:54.553Z|00007|reconnect|DBG|unix:/var/run/openvswitch/db.sock: entering
CONNECTING
[pod/ovnkube-node-2cpjc/ovn-controller] 2023-10-
12T14:39:54.553Z|00008|ovsdb_cs|DBG|unix:/var/run/openvswitch/db.sock:
SERVER_SCHEMA_REQUESTED -> SERVER_SCHEMA_REQUESTED at lib/ovsdb-
cs.c:423
```

6. Optional: Check the **ConfigMap** file has been applied by running the following command:

```
for f in $(oc -n openshift-ovn-kubernetes get po -l 'app=ovnkube-node' --no-headers -o
custom-columns=N:.metadata.name) ; do echo "---- $f ----" ; oc -n openshift-ovn-kubernetes
exec -c ovnkube-controller $f --  pgrep -a -f  init-ovnkube-controller | grep -P -o
'^.*loglevel\s+\d' ; done
```

Example output

```
---- ovnkube-node-2dt57 ----
60981 /usr/bin/ovnkube --init-ovnkube-controller xpst8-worker-c-vmh5n.c.openshift-
qe.internal --init-node xpst8-worker-c-vmh5n.c.openshift-qe.internal --config-
file=/run/ovnkube-config/ovnkube.conf --ovn-empty-lb-events --loglevel 4
---- ovnkube-node-4zznh ----
178034 /usr/bin/ovnkube --init-ovnkube-controller xpst8-master-2.c.openshift-qe.internal --
init-node xpst8-master-2.c.openshift-qe.internal --config-file=/run/ovnkube-
config/ovnkube.conf --ovn-empty-lb-events --loglevel 4
---- ovnkube-node-548sx ----
77499 /usr/bin/ovnkube --init-ovnkube-controller xpst8-worker-a-fjtnb.c.openshift-qe.internal -
-init-node xpst8-worker-a-fjtnb.c.openshift-qe.internal --config-file=/run/ovnkube-
config/ovnkube.conf --ovn-empty-lb-events --loglevel 4
---- ovnkube-node-6btrf ----
73781 /usr/bin/ovnkube --init-ovnkube-controller xpst8-worker-b-p8rww.c.openshift-
qe.internal --init-node xpst8-worker-b-p8rww.c.openshift-qe.internal --config-
file=/run/ovnkube-config/ovnkube.conf --ovn-empty-lb-events --loglevel 4
---- ovnkube-node-fkc9r ----
130707 /usr/bin/ovnkube --init-ovnkube-controller xpst8-master-0.c.openshift-qe.internal --
init-node xpst8-master-0.c.openshift-qe.internal --config-file=/run/ovnkube-
config/ovnkube.conf --ovn-empty-lb-events --loglevel 5
---- ovnkube-node-tk9l4 ----
181328 /usr/bin/ovnkube --init-ovnkube-controller xpst8-master-1.c.openshift-qe.internal --
init-node xpst8-master-1.c.openshift-qe.internal --config-file=/run/ovnkube-
config/ovnkube.conf --ovn-empty-lb-events --loglevel 4
```

3.6. CHECKING THE OVN-KUBERNETES POD NETWORK CONNECTIVITY

The connectivity check controller, in OpenShift Container Platform 4.10 and later, orchestrates connection verification checks in your cluster. These include Kubernetes API, OpenShift API and individual nodes. The results for the connection tests are stored in **PodNetworkConnectivity** objects in the **openshift-network-diagnostics** namespace. Connection tests are performed every minute in parallel.

Prerequisites

- Access to the OpenShift CLI (**oc**).
- Access to the cluster as a user with the **cluster-admin** role.
- You have installed **jq**.

Procedure

1. To list the current **PodNetworkConnectivityCheck** objects, enter the following command:

```
$ oc get podnetworkconnectivitychecks -n openshift-network-diagnostics
```

2. View the most recent success for each connection object by using the following command:

```
$ oc get podnetworkconnectivitychecks -n openshift-network-diagnostics \
-o json | jq '.items[]| .spec.targetEndpoint,.status.successes[0]'
```

3. View the most recent failures for each connection object by using the following command:

```
$ oc get podnetworkconnectivitychecks -n openshift-network-diagnostics \
-o json | jq '.items[]| .spec.targetEndpoint,.status.failures[0]'
```

4. View the most recent outages for each connection object by using the following command:

```
$ oc get podnetworkconnectivitychecks -n openshift-network-diagnostics \
-o json | jq '.items[]| .spec.targetEndpoint,.status.outages[0]'
```

The connectivity check controller also logs metrics from these checks into Prometheus.

5. View all the metrics by running the following command:

```
$ oc exec prometheus-k8s-0 -n openshift-monitoring -- \
promtool query instant  http://localhost:9090 \
'{component="openshift-network-diagnostics"}'
```

6. View the latency between the source pod and the openshift api service for the last 5 minutes:

```
$ oc exec prometheus-k8s-0 -n openshift-monitoring -- \
promtool query instant  http://localhost:9090 \
'{component="openshift-network-diagnostics"}'
```

3.7. CHECKING OVN-KUBERNETES NETWORK TRAFFIC WITH OVS SAMPLING USING THE CLI

IMPORTANT

Checking OVN-Kubernetes network traffic with OVS sampling is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

OVN-Kubernetes network traffic can be viewed with OVS sampling via the CLI for the following network APIs:

- **NetworkPolicy**
- **AdminNetworkPolicy**
- **BaselineNetworkPolicy**
- **UserDefinedNetwork** isolation
- **EgressFirewall**
- Multicast ACLs.

Scripts for these networking events are found in the **/usr/bin/ovnkube-observ** path of each OVN-Kubernetes node.

Although both the Network Observability Operator and checking OVN-Kubernetes network traffic with OVS sampling are good for debuggability, the Network Observability Operator is intended for observing network events. Alternatively, checking OVN-Kubernetes network traffic with OVS sampling using the CLI is intended to help with packet tracing; it can also be used while the Network Observability Operator is installed, however that is not a requirement.

Administrators can add the **--add-ovs-collect** option to view network traffic across the node, or pass in additional flags to filter result for specific pods. Additional flags can be found in the "OVN-Kubernetes network traffic with OVS sampling flags" section.

Use the following procedure to view OVN-Kubernetes network traffic using the CLI.

Prerequisites

- You are logged in to the cluster as a user with **cluster-admin** privileges.
- You have created a source pod and a destination pod and ran traffic between them.
- You have created at least one of the following network APIs: **NetworkPolicy**, **AdminNetworkPolicy**, **BaselineNetworkPolicy**, **UserDefinedNetwork** isolation, multicast, or egress firewalls.

Procedure

1. To enable the **OVNObservability** with OVS sampling feature, enable **TechPreviewNoUpgrade** feature set in the **FeatureGate** CR named **cluster** by entering the following command:

```
$ oc patch --type=merge --patch '{"spec": {"featureSet": "TechPreviewNoUpgrade"}}'
featuregate/cluster
```

Example output

```
featuregate.config.openshift.io/cluster patched
```

2. Confirm that the **OVNObservability** feature is enabled by entering the following command:

```
$ oc get featuregate cluster -o yaml
```

Example output

```
  featureGates:
# ...
    enabled:
    - name: OVNObservability
```

3. Obtain a list of the pods inside of the namespace in which you have created one of the relevant network APIs by entering the following command. Note the **NODE** name of the pods, as they are used in the following step.

```
$ oc get pods -n <namespace> -o wide
```

Example output

```
NAME              READY   STATUS    RESTARTS   AGE     IP            NODE
NOMINATED NODE   READINESS GATES
destination-pod   1/1     Running   0          53s     10.131.0.23   ci-ln-1gqp7b2-72292-bb9dv-
worker-a-gtmpc   <none>           <none>
source-pod        1/1     Running   0          56s     10.131.0.22   ci-ln-1gqp7b2-72292-bb9dv-
worker-a-gtmpc   <none>           <none>
```

4. Obtain a list of OVN-Kubernetes pods and locate the pod that shares the same **NODE** as the pods from the previous step by entering the following command:

```
$ oc get pods -n openshift-ovn-kubernetes -o wide
```

Example output

```
NAME
...                          READY   STATUS    RESTARTS      AGE   IP           NODE
NOMINATED NODE
ovnkube-node-jzn5b             8/8     Running   1 (34m ago)   37m   10.0.128.2   ci-ln-1gqp7b2-
72292-bb9dv-worker-a-gtmpc   <none>
...
```

5. Open a bash shell inside of the **ovnkube-node** pod by entering the following command:

```
$ oc exec -it <pod_name> -n openshift-ovn-kubernetes -- bash
```

6. While inside of the **ovnkube-node** pod, you can run the **ovnkube-observ -add-ovs-collector** script to show network events using the OVS collector. For example:

   ```
   # /usr/bin/ovnkube-observ -add-ovs-collector
   ```

   Example output

   ```
   ...
   2024/12/02 19:41:41.327584 OVN-K message: Allowed by default allow from local node
   policy, direction ingress
   2024/12/02 19:41:41.327593 src=10.131.0.2, dst=10.131.0.6

   2024/12/02 19:41:41.327692 OVN-K message: Allowed by default allow from local node
   policy, direction ingress
   2024/12/02 19:41:41.327715 src=10.131.0.6, dst=10.131.0.2
   ...
   ```

7. You can filter the content by type, such as source pods, by entering the following command with the **-filter-src-ip** flag and your pod's IP address. For example:

   ```
   # /usr/bin/ovnkube-observ -add-ovs-collector -filter-src-ip <pod_ip_address>
   ```

   Example output

   ```
   ...
   Found group packets, id 14
   2024/12/10 16:27:12.456473 OVN-K message: Allowed by admin network policy allow-
   egress-group1, direction Egress
   2024/12/10 16:27:12.456570 src=10.131.0.22, dst=10.131.0.23

   2024/12/10 16:27:14.484421 OVN-K message: Allowed by admin network policy allow-
   egress-group1, direction Egress
   2024/12/10 16:27:14.484428 src=10.131.0.22, dst=10.131.0.23

   2024/12/10 16:27:12.457222 OVN-K message: Allowed by network policy test:allow-ingress-
   from-specific-pod, direction Ingress
   2024/12/10 16:27:12.457228 src=10.131.0.22, dst=10.131.0.23

   2024/12/10 16:27:12.457288 OVN-K message: Allowed by network policy test:allow-ingress-
   from-specific-pod, direction Ingress
   2024/12/10 16:27:12.457299 src=10.131.0.22, dst=10.131.0.23
   ...
   ```

   For a full list of flags that can be passed in with **/usr/bin/ovnkube-observ**, see "OVN-Kubernetes network traffic with OVS sampling flags".

3.7.1. OVN-Kubernetes network traffic with OVS sampling flags

The following flags are available to view OVN-Kubernetes network traffic by using the CLI. Append these flags to the following syntax in your terminal after you have opened a bash shell inside of the **ovnkube-node** pod:

Command syntax

```
# /usr/bin/ovnkube-observ <flag>
```

Flag	Description
-h	Returns a complete list flags that can be used with the **usr/bin/ovnkube-observ** command. `
-add-ovs-collector	Add OVS collector to enable sampling. Use with caution. Make sure no one else is using observability.
-enable-enrichment	Enrich samples with NBDB data. Defaults to **true**.
-filter-dst-ip	Filter only packets to a given destination IP.
-filter-src-ip	Filters only packets from a given source IP.
-log-cookie	Print raw sample cookie with psample group_id.
-output-file	Output file to write the samples to.
-print-full-packet	Print full received packet. When false, only source and destination IPs are printed with every sample.

3.8. ADDITIONAL RESOURCES

- Gathering data about your cluster for Red Hat Support
- Implementation of connection health checks
- Verifying network connectivity for an endpoint

CHAPTER 4. TRACING OPENFLOW WITH OVNKUBE-TRACE

OVN and OVS traffic flows can be simulated in a single utility called **ovnkube-trace**. The **ovnkube-trace** utility runs **ovn-trace**, **ovs-appctl ofproto/trace** and **ovn-detrace** and correlates that information in a single output.

You can execute the **ovnkube-trace** binary from a dedicated container. For releases after OpenShift Container Platform 4.7, you can also copy the binary to a local host and execute it from that host.

4.1. INSTALLING THE OVNKUBE-TRACE ON LOCAL HOST

The **ovnkube-trace** tool traces packet simulations for arbitrary UDP or TCP traffic between points in an OVN-Kubernetes driven OpenShift Container Platform cluster. Copy the **ovnkube-trace** binary to your local host making it available to run against the cluster.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.

Procedure

1. Create a pod variable by using the following command:

```
$ POD=$(oc get pods -n openshift-ovn-kubernetes -l app=ovnkube-control-plane -o name |
head -1 | awk -F '/' '{print $NF}')
```

2. Run the following command on your local host to copy the binary from the **ovnkube-control-plane** pods:

```
$ oc cp -n openshift-ovn-kubernetes $POD:/usr/bin/ovnkube-trace -c ovnkube-cluster-
manager ovnkube-trace
```

NOTE

If you are using Red Hat Enterprise Linux (RHEL) 8 to run the **ovnkube-trace** tool, you must copy the file **/usr/lib/rhel8/ovnkube-trace** to your local host.

3. Make **ovnkube-trace** executable by running the following command:

```
$ chmod +x ovnkube-trace
```

4. Display the options available with **ovnkube-trace** by running the following command:

```
$ ./ovnkube-trace -help
```

Expected output

```
Usage of ./ovnkube-trace:
  -addr-family string
```

```
    Address family (ip4 or ip6) to be used for tracing (default "ip4")
  -dst string
    dest: destination pod name
  -dst-ip string
    destination IP address (meant for tests to external targets)
  -dst-namespace string
    k8s namespace of dest pod (default "default")
  -dst-port string
    dst-port: destination port (default "80")
  -kubeconfig string
    absolute path to the kubeconfig file
  -loglevel string
    loglevel: klog level (default "0")
  -ovn-config-namespace string
    namespace used by ovn-config itself
  -service string
    service: destination service name
  -skip-detrace
    skip ovn-detrace command
  -src string
    src: source pod name
  -src-namespace string
    k8s namespace of source pod (default "default")
  -tcp
    use tcp transport protocol
  -udp
    use udp transport protocol
```

The command-line arguments supported are familiar Kubernetes constructs, such as namespaces, pods, services so you do not need to find the MAC address, the IP address of the destination nodes, or the ICMP type.

The log levels are:

- 0 (minimal output)
- 2 (more verbose output showing results of trace commands)
- 5 (debug output)

4.2. RUNNING OVNKUBE-TRACE

Run **ovn-trace** to simulate packet forwarding within an OVN logical network.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.
- You have installed **ovnkube-trace** on local host

Example: Testing that DNS resolution works from a deployed pod

This example illustrates how to test the DNS resolution from a deployed pod to the core DNS pod that runs in the cluster.

Procedure

1. Start a web service in the default namespace by entering the following command:

```
$ oc run web --namespace=default --image=quay.io/openshifttest/nginx --labels="app=web" -
-expose --port=80
```

2. List the pods running in the **openshift-dns** namespace:

```
oc get pods -n openshift-dns
```

Example output

```
NAME                   READY   STATUS    RESTARTS   AGE
dns-default-8s42x      2/2     Running   0          5h8m
dns-default-mdw6r      2/2     Running   0          4h58m
dns-default-p8t5h      2/2     Running   0          4h58m
dns-default-rl6nk      2/2     Running   0          5h8m
dns-default-xbgqx      2/2     Running   0          5h8m
dns-default-zv8f6      2/2     Running   0          4h58m
node-resolver-62jjb    1/1     Running   0          5h8m
node-resolver-8z4cj    1/1     Running   0          4h59m
node-resolver-bq244    1/1     Running   0          5h8m
node-resolver-hc58n    1/1     Running   0          4h59m
node-resolver-lm6z4    1/1     Running   0          5h8m
node-resolver-zfx5k    1/1     Running   0          5h
```

3. Run the following **ovnkube-trace** command to verify DNS resolution is working:

```
$ ./ovnkube-trace \
  -src-namespace default \ (1)
  -src web \ (2)
  -dst-namespace openshift-dns \ (3)
  -dst dns-default-p8t5h \ (4)
  -udp -dst-port 53 \ (5)
  -loglevel 0 (6)
```

(1) Namespace of the source pod

(2) Source pod name

(3) Namespace of destination pod

(4) Destination pod name

(5) Use the **udp** transport protocol. Port 53 is the port the DNS service uses.

(6) Set the log level to 0 (0 is minimal and 5 is debug)

Example output if the src&dst pod lands on the same node

```
ovn-trace source pod to destination pod indicates success from web to dns-default-p8t5h
ovn-trace destination pod to source pod indicates success from dns-default-p8t5h to web
```

```
ovs-appctl ofproto/trace source pod to destination pod indicates success from web to dns-default-p8t5h
ovs-appctl ofproto/trace destination pod to source pod indicates success from dns-default-p8t5h to web
ovn-detrace source pod to destination pod indicates success from web to dns-default-p8t5h
ovn-detrace destination pod to source pod indicates success from dns-default-p8t5h to web
```

Example output if the **src&dst** pod lands on a different node

```
ovn-trace source pod to destination pod indicates success from web to dns-default-8s42x
ovn-trace (remote) source pod to destination pod indicates success from web to dns-default-8s42x
ovn-trace destination pod to source pod indicates success from dns-default-8s42x to web
ovn-trace (remote) destination pod to source pod indicates success from dns-default-8s42x to web
ovs-appctl ofproto/trace source pod to destination pod indicates success from web to dns-default-8s42x
ovs-appctl ofproto/trace destination pod to source pod indicates success from dns-default-8s42x to web
ovn-detrace source pod to destination pod indicates success from web to dns-default-8s42x
ovn-detrace destination pod to source pod indicates success from dns-default-8s42x to web
```

The ouput indicates success from the deployed pod to the DNS port and also indicates that it is successful going back in the other direction. So you know bi-directional traffic is supported on UDP port 53 if my web pod wants to do dns resolution from core DNS.

If for example that did not work and you wanted to get the **ovn-trace**, the **ovs-appctl** of **proto/trace** and **ovn-detrace**, and more debug type information increase the log level to 2 and run the command again as follows:

```
$ ./ovnkube-trace \
  -src-namespace default \
  -src web \
  -dst-namespace openshift-dns \
  -dst dns-default-467qw \
  -udp -dst-port 53 \
  -loglevel 2
```

The output from this increased log level is too much to list here. In a failure situation the output of this command shows which flow is dropping that traffic. For example an egress or ingress network policy may be configured on the cluster that does not allow that traffic.

Example: Verifying by using debug output a configured default deny

This example illustrates how to identify by using the debug output that an ingress default deny policy blocks traffic.

Procedure

1. Create the following YAML that defines a **deny-by-default** policy to deny ingress from all pods in all namespaces. Save the YAML in the **deny-by-default.yaml** file:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-by-default
  namespace: default
spec:
  podSelector: {}
  ingress: []
```

2. Apply the policy by entering the following command:

   ```
   $ oc apply -f deny-by-default.yaml
   ```

   Example output

   ```
   networkpolicy.networking.k8s.io/deny-by-default created
   ```

3. Start a web service in the **default** namespace by entering the following command:

   ```
   $ oc run web --namespace=default --image=quay.io/openshifttest/nginx --labels="app=web" -
   -expose --port=80
   ```

4. Run the following command to create the **prod** namespace:

   ```
   $ oc create namespace prod
   ```

5. Run the following command to label the **prod** namespace:

   ```
   $ oc label namespace/prod purpose=production
   ```

6. Run the following command to deploy an **alpine** image in the **prod** namespace and start a shell:

   ```
   $ oc run test-6459 --namespace=prod --rm -i -t --image=alpine -- sh
   ```

7. Open another terminal session.

8. In this new terminal session run **ovn-trace** to verify the failure in communication between the source pod **test-6459** running in namespace **prod** and destination pod running in the **default** namespace:

   ```
   $ ./ovnkube-trace \
    -src-namespace prod \
    -src test-6459 \
    -dst-namespace default \
    -dst web \
    -tcp -dst-port 80 \
    -loglevel 0
   ```

   Example output

   ```
   ovn-trace source pod to destination pod indicates failure from test-6459 to web
   ```

9. Increase the log level to 2 to expose the reason for the failure by running the following command:

```
$ ./ovnkube-trace \
 -src-namespace prod \
 -src test-6459 \
 -dst-namespace default \
 -dst web \
 -tcp -dst-port 80 \
 -loglevel 2
```

Example output

```
...
------------------------------------------------
 3. ls_out_acl_hint (northd.c:7454): !ct.new && ct.est && !ct.rpl && ct_mark.blocked == 0,
priority 4, uuid 12efc456
    reg0[8] = 1;
    reg0[10] = 1;
    next;
 5. ls_out_acl_action (northd.c:7835): reg8[30..31] == 0, priority 500, uuid 69372c5d
    reg8[30..31] = 1;
    next(4);
 5. ls_out_acl_action (northd.c:7835): reg8[30..31] == 1, priority 500, uuid 2fa0af89
    reg8[30..31] = 2;
    next(4);
 4. ls_out_acl_eval (northd.c:7691): reg8[30..31] == 2 && reg0[10] == 1 && (outport ==
@a16982411286042166782_ingressDefaultDeny), priority 2000, uuid 447d0dab
    reg8[17] = 1;
    ct_commit { ct_mark.blocked = 1; }; (1)
    next;
...
```

1 Ingress traffic is blocked due to the default deny policy being in place.

10. Create a policy that allows traffic from all pods in a particular namespaces with a label **purpose=production**. Save the YAML in the **web-allow-prod.yaml** file:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-allow-prod
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: web
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          purpose: production
```

11. Apply the policy by entering the following command:

```
$ oc apply -f web-allow-prod.yaml
```

12. Run **ovnkube-trace** to verify that traffic is now allowed by entering the following command:

```
$ ./ovnkube-trace \
 -src-namespace prod \
 -src test-6459 \
 -dst-namespace default \
 -dst web \
 -tcp -dst-port 80 \
 -loglevel 0
```

Expected output

```
ovn-trace source pod to destination pod indicates success from test-6459 to web
ovn-trace destination pod to source pod indicates success from web to test-6459
ovs-appctl ofproto/trace source pod to destination pod indicates success from test-6459 to web
ovs-appctl ofproto/trace destination pod to source pod indicates success from web to test-6459
ovn-detrace source pod to destination pod indicates success from test-6459 to web
ovn-detrace destination pod to source pod indicates success from web to test-6459
```

13. Run the following command in the shell that was opened in step six to connect nginx to the web-server:

```
 wget -qO- --timeout=2 http://web.default
```

Expected output

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
```

```
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

4.3. ADDITIONAL RESOURCES

- Tracing Openflow with ovnkube-trace utility
- ovnkube-trace

CHAPTER 5. CONVERTING TO IPV4/IPV6 DUAL-STACK NETWORKING

As a cluster administrator, you can convert your IPv4 single-stack cluster to a dual-network cluster network that supports IPv4 and IPv6 address families. After converting to dual-stack networking, new and existing pods have dual-stack networking enabled.

IMPORTANT

When using dual-stack networking where IPv6 is required, you cannot use IPv4-mapped IPv6 addresses, such as **::FFFF:198.51.100.1**.

Additional resources

- For more information about platform-specific support for dual-stack networking, see OVN-Kubernetes purpose

5.1. CONVERTING TO A DUAL-STACK CLUSTER NETWORK

As a cluster administrator, you can convert your single-stack cluster network to a dual-stack cluster network.

IMPORTANT

After converting your cluster to use dual-stack networking, you must re-create any existing pods for them to receive IPv6 addresses, because only new pods are assigned IPv6 addresses.

Converting a single-stack cluster network to a dual-stack cluster network consists of creating patches and applying them to the network and infrastructure of the cluster. You can convert to a dual-stack cluster network for a cluster that runs on either installer-provisioned infrastructure or user-provisioned infrastructure.

NOTE

Each patch operation that changes **clusterNetwork**, **serviceNetwork**, **apiServerInternalIPs**, and **ingressIP** objects triggers a restart of the cluster. Changing the **MachineNetworks** object does not cause a reboot of the cluster.

On installer-provisioned infrastructure only, if you need to add IPv6 virtual IPs (VIPs) for API and Ingress services to an existing dual-stack-configured cluster, you need to patch only the infrastructure and not the network for the cluster.

IMPORTANT

If you already upgraded your cluster to OpenShift Container Platform 4.16 or later and you need to convert the single-stack cluster network to a dual-stack cluster network, you must specify an existing IPv4 **machineNetwork** network configuration from the **install-config.yaml** file for API and Ingress services in the YAML configuration patch file. This configuration ensures that IPv4 traffic exists in the same network interface as the default gateway.

Example YAML configuration file with an added IPv4 address block for the machineNetwork network

```
- op: add
  path: /spec/platformSpec/baremetal/machineNetworks/- (1)
  value: 192.168.1.0/24
  # ...
```

(1) Ensure that you specify an address block for the **machineNetwork** network where your machines operate. You must select both API and Ingress IP addresses for the machine network.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.
- Your cluster uses the OVN-Kubernetes network plugin.
- The cluster nodes have IPv6 addresses.
- You have configured an IPv6-enabled router based on your infrastructure.

Procedure

1. To specify IPv6 address blocks for cluster and service networks, create a YAML configuration patch file that has a similar configuration to the following example:

```
- op: add
  path: /spec/clusterNetwork/-
  value: (1)
    cidr: fd01::/48
    hostPrefix: 64
- op: add
  path: /spec/serviceNetwork/-
  value: fd02::/112 (2)
```

(1) Specify an object with the **cidr** and **hostPrefix** parameters. The host prefix must be **64** or greater. The IPv6 Classless Inter-Domain Routing (CIDR) prefix must be large enough to accommodate the specified host prefix.

(2) Specify an IPv6 CIDR with a prefix of **112**. Kubernetes uses only the lowest 16 bits. For a prefix of **112**, IP addresses are assigned from **112** to **128** bits.

2. Patch the cluster network configuration by entering the following command in your CLI:

```
$ oc patch network.config.openshift.io cluster \ (1)
  --type='json' --patch-file <file>.yaml
```

(1) Where **file** specifies the name of your created YAML file.

Example output

```
network.config.openshift.io/cluster patched
```

3. On installer-provisioned infrastructure where you added IPv6 VIPs for API and Ingress services, complete the following steps:

   a. Specify IPv6 VIPs for API and Ingress services for your cluster. Create a YAML configuration patch file that has a similar configuration to the following example:

   ```
   - op: add
     path: /spec/platformSpec/baremetal/machineNetworks/- (1)
     value: fd2e:6f44:5dd8::/64
   - op: add
     path: /spec/platformSpec/baremetal/apiServerInternalIPs/- (2)
     value: fd2e:6f44:5dd8::4
   - op: add
     path: /spec/platformSpec/baremetal/ingressIPs/-
     value: fd2e:6f44:5dd8::5
   ```

   (1) Ensure that you specify an address block for the **machineNetwork** network where your machines operate. You must select both API and Ingress IP addresses for the machine network.

   (2) Ensure that you specify each file path according to your platform. The example demonstrates a file path on a bare-metal platform.

   b. Patch the infrastructure by entering the following command in your CLI:

   ```
   $ oc patch infrastructure cluster \
     --type='json' --patch-file <file>.yaml
   ```

   Where:

   <file>
   Specifies the name of your created YAML file.

   #### Example output

   ```
   infrastructure/cluster patched
   ```

Verification

1. Show the cluster network configuration by entering the following command in your CLI:

```
$ oc describe network
```

2. Verify the successful installation of the patch on the network configuration by checking that the cluster network configuration recognizes the IPv6 address blocks that you specified in the YAML file.

   Example output

```
# ...
Status:
  Cluster Network:
    Cidr:              10.128.0.0/14
    Host Prefix:       23
    Cidr:              fd01::/48
    Host Prefix:       64
  Cluster Network MTU:  1400
  Network Type:         OVNKubernetes
  Service Network:
    172.30.0.0/16
    fd02::/112
# ...
```

3. Complete the following additional tasks for a cluster that runs on installer-provisioned infrastructure:

   a. Show the cluster infrastructure configuration by entering the following command in your CLI:

```
$ oc describe infrastructure
```

   b. Verify the successful installation of the patch on the cluster infrastructure by checking that the infrastructure recognizes the IPv6 address blocks that you specified in the YAML file.

   Example output

```
# ...
spec:
# ...
  platformSpec:
    baremetal:
      apiServerInternalIPs:
      - 192.168.123.5
      - fd2e:6f44:5dd8::4
      ingressIPs:
      - 192.168.123.10
      - fd2e:6f44:5dd8::5
status:
# ...
  platformStatus:
    baremetal:
      apiServerInternalIP: 192.168.123.5
      apiServerInternalIPs:
      - 192.168.123.5
      - fd2e:6f44:5dd8::4
      ingressIP: 192.168.123.10
    ingressIPs:
    - 192.168.123.10
    - fd2e:6f44:5dd8::5
# ...
```

5.2. CONVERTING TO A SINGLE-STACK CLUSTER NETWORK

As a cluster administrator, you can convert your dual-stack cluster network to a single-stack cluster network.

IMPORTANT

If you originally converted your IPv4 single-stack cluster network to a dual-stack cluster, you can convert only back to the IPv4 single-stack cluster and not an IPv6 single-stack cluster network. The same restriction applies for converting back to an IPv6 single-stack cluster network.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.
- Your cluster uses the OVN-Kubernetes network plugin.
- The cluster nodes have IPv6 addresses.
- You have enabled dual-stack networking.

Procedure

1. Edit the **networks.config.openshift.io** custom resource (CR) by running the following command:

   ```
   $ oc edit networks.config.openshift.io
   ```

2. Remove the IPv4 or IPv6 configuration that you added to the **cidr** and the **hostPrefix** parameters from completing the "Converting to a dual-stack cluster network " procedure steps.

CHAPTER 6. CONFIGURING OVN-KUBERNETES INTERNAL IP ADDRESS SUBNETS

As a cluster administrator, you can change the IP address ranges that the OVN-Kubernetes network plugin uses for the join and transit subnets.

6.1. CONFIGURING THE OVN-KUBERNETES JOIN SUBNET

You can change the join subnet used by OVN-Kubernetes to avoid conflicting with any existing subnets already in use in your environment.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in to the cluster with a user with **cluster-admin** privileges.
- Ensure that the cluster uses the OVN-Kubernetes network plugin.

Procedure

- To change the OVN-Kubernetes join subnet, enter the following command:

```
$ oc patch network.operator.openshift.io cluster --type='merge' \
  -p='{"spec":{"defaultNetwork":{"ovnKubernetesConfig":
    {"ipv4":{"internalJoinSubnet": "<join_subnet>"},
    "ipv6":{"internalJoinSubnet": "<join_subnet>"}}}}}'
```

where:

<join_subnet>

Specifies an IP address subnet for internal use by OVN-Kubernetes. The subnet must be larger than the number of nodes in the cluster and it must be large enough to accommodate one IP address per node in the cluster. This subnet cannot overlap with any other subnets used by OpenShift Container Platform or on the host itself. The default value for IPv4 is **100.64.0.0/16** and the default value for IPv6 is **fd98::/64**.

Example output

```
network.operator.openshift.io/cluster patched
```

Verification

- To confirm that the configuration is active, enter the following command:

```
$ oc get network.operator.openshift.io \
  -o jsonpath="{.items[0].spec.defaultNetwork}"
```

The command operation can take up to 30 minutes for this change to take effect.

Example output

```
{
  "ovnKubernetesConfig": {
    "ipv4": {
      "internalJoinSubnet": "100.64.1.0/16"
    },
  },
  "type": "OVNKubernetes"
}
```

6.2. CONFIGURING THE OVN-KUBERNETES MASQUERADE SUBNET AS A POST-INSTALLATION OPERATION

You can change the masquerade subnet used by OVN-Kubernetes as a post-installation operation to avoid conflicts with any existing subnets that are already in use in your environment.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in to the cluster as a user with **cluster-admin** privileges.

Procedure

- Change your cluster's masquerade subnet:
  - For dualstack clusters using IPv6, run the following command:

    ```
    $ oc patch networks.operator.openshift.io cluster --type=merge -p '{"spec":
    {"defaultNetwork":{"ovnKubernetesConfig":{"gatewayConfig":{"ipv4":
    {"internalMasqueradeSubnet": "<ipv4_masquerade_subnet>"},"ipv6":
    {"internalMasqueradeSubnet": "<ipv6_masquerade_subnet>"}}}}}}'
    ```

    where:

    ipv4_masquerade_subnet

    Specifies an IP address to be used as the IPv4 masquerade subnet. This range cannot overlap with any other subnets used by OpenShift Container Platform or on the host itself. In versions of OpenShift Container Platform earlier than 4.17, the default value for IPv4 was **169.254.169.0/29**, and clusters that were upgraded to version 4.17 maintain this value. For new clusters starting from version 4.17, the default value is **169.254.0.0/17**.

    ipv6_masquerade_subnet

    Specifies an IP address to be used as the IPv6 masquerade subnet. This range cannot overlap with any other subnets used by OpenShift Container Platform or on the host itself. The default value for IPv6 is **fd69::/125**.

  - For clusters using IPv4, run the following command:

    ```
    $ oc patch networks.operator.openshift.io cluster --type=merge -p '{"spec":
    {"defaultNetwork":{"ovnKubernetesConfig":{"gatewayConfig":{"ipv4":
    {"internalMasqueradeSubnet": "<ipv4_masquerade_subnet>"}}}}}}'
    ```

    where:

ipv4_masquerade_subnet::Specifies an IP address to be used as the IPv4 masquerade subnet. This range cannot overlap with any other subnets used by OpenShift Container Platform or on the host itself. In versions of OpenShift Container Platform earlier than 4.17, the default value for IPv4 was **169.254.169.0/29**, and clusters that were upgraded to version 4.17 maintain this value. For new clusters starting from version 4.17, the default value is **169.254.0.0/17**.

6.3. CONFIGURING THE OVN-KUBERNETES TRANSIT SUBNET

You can change the transit subnet used by OVN-Kubernetes to avoid conflicting with any existing subnets already in use in your environment.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in to the cluster with a user with **cluster-admin** privileges.
- Ensure that the cluster uses the OVN-Kubernetes network plugin.

Procedure

- To change the OVN-Kubernetes transit subnet, enter the following command:

```
$ oc patch network.operator.openshift.io cluster --type='merge' \
  -p='{"spec":{"defaultNetwork":{"ovnKubernetesConfig":
    {"ipv4":{"internalTransitSwitchSubnet": "<transit_subnet>"},
    "ipv6":{"internalTransitSwitchSubnet": "<transit_subnet>"}}}}}'
```

where:

<transit_subnet>

Specifies an IP address subnet for the distributed transit switch that enables east-west traffic. This subnet cannot overlap with any other subnets used by OVN-Kubernetes or on the host itself. The default value for IPv4 is **100.88.0.0/16** and the default value for IPv6 is **fd97::/64**.

Example output

```
network.operator.openshift.io/cluster patched
```

Verification

- To confirm that the configuration is active, enter the following command:

```
$ oc get network.operator.openshift.io \
  -o jsonpath="{.items[0].spec.defaultNetwork}"
```

It can take up to 30 minutes for this change to take effect.

Example output

```
{
  "ovnKubernetesConfig": {
    "ipv4": {
      "internalTransitSwitchSubnet": "100.88.1.0/16"
    },
  },
  "type": "OVNKubernetes"
}
```

CHAPTER 7. CONFIGURING A GATEWAY

As a cluster administrator you can configure the **gatewayConfig** object to manage how external traffic leaves the cluster. You do so by setting the **routingViaHost** parameter to one of the following values:

- **true** means that egress traffic routes through a specific local gateway on the node that hosts the pod. Egress traffic routes through the host and this traffic applies to the routing table of the host.
- **false** means that egress traffic routes through a dedicated node but a group of nodes share the same gateway. Egress traffic does not route through the host. The Open vSwitch (OVS) outputs traffic directly to the node IP interface.

7.1. CONFIGURING EGRESS ROUTING POLICIES

As a cluster administrator you can configure egress routing policies by using the **gatewayConfig** specification in the Cluster Network Operator (CNO). You can use the following procedure to set the **routingViaHost** field to **true** or **false**.

You can follow the optional step in the procedure to enable IP forwarding alongside the **routingViaHost=true** configuration if you need the host network of the node to act as a router for traffic not related to OVN-Kubernetes. For example, possible use cases for combining local gateway with IP forwarding include:

- Configuring all pod egress traffic to be forwarded via the node's IP
- Integrating OVN-Kubernetes CNI with external network address translation (NAT) devices
- Configuring OVN-Kubernetes CNI to use a kernel routing table

Prerequisites

- You are logged in as a user with admin privileges.

Procedure

1. Back up the existing network configuration by running the following command:

```
$ oc get network.operator cluster -o yaml > network-config-backup.yaml
```

2. Set the **routingViaHost** parameter to **true** by entering the following command. Egress traffic then gets routed through a specific gateway according to the routes that you configured on the node.

```
$ oc patch networks.operator.openshift.io cluster --type=merge -p '{"spec":{"defaultNetwork":
{"ovnKubernetesConfig":{"gatewayConfig":{"routingViaHost": true}}}}}'
```

3. Verify the correct application of the **routingViaHost=true** configuration by running the following command:

```
$ oc get networks.operator.openshift.io cluster -o yaml | grep -A 5 "gatewayConfig"
```

Example output

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
# ...
gatewayConfig:
        ipv4: {}
        ipv6: {}
        routingViaHost: true 1
      genevePort: 6081
      ipsecConfig:
# ...
```

1 A value of **true** means that egress traffic gets routed through a specific local gateway on the node that hosts the pod. A value of **false** for the parameter means that a group of nodes share a single gateway so traffic does not get routed through a single host.

4. Optional: Enable IP forwarding globally by running the following command:

```
$ oc patch network.operator cluster --type=merge -p '{"spec":{"defaultNetwork":
{"ovnKubernetesConfig":{"gatewayConfig":{"ipForwarding": "Global"}}}}}'
```

   a. Verify that the **ipForwarding** spec has been set to **Global** by running the following command:

```
$ oc get networks.operator.openshift.io cluster -o yaml | grep -A 5 "gatewayConfig"
```

Example output

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
# ...
gatewayConfig:
        ipForwarding: Global
        ipv4: {}
        ipv6: {}
        routingViaHost: true
      genevePort: 6081
# ...
```

CHAPTER 8. CONFIGURE AN EXTERNAL GATEWAY ON THE DEFAULT NETWORK

As a cluster administrator, you can configure an external gateway on the default network.

This feature offers the following benefits:

- Granular control over egress traffic on a per-namespace basis
- Flexible configuration of static and dynamic external gateway IP addresses
- Support for both IPv4 and IPv6 address families

8.1. PREREQUISITES

- Your cluster uses the OVN-Kubernetes network plugin.
- Your infrastructure is configured to route traffic from the secondary external gateway.

8.2. HOW OPENSHIFT CONTAINER PLATFORM DETERMINES THE EXTERNAL GATEWAY IP ADDRESS

You configure a secondary external gateway with the **AdminPolicyBasedExternalRoute** custom resource (CR) from the **k8s.ovn.org** API group. The CR supports static and dynamic approaches to specifying an external gateway's IP address.

Each namespace that a **AdminPolicyBasedExternalRoute** CR targets cannot be selected by any other **AdminPolicyBasedExternalRoute** CR. A namespace cannot have concurrent secondary external gateways.

Changes to policies are isolated in the controller. If a policy fails to apply, changes to other policies do not trigger a retry of other policies. Policies are only re-evaluated, applying any differences that might have occurred by the change, when updates to the policy itself or related objects to the policy such as target namespaces, pod gateways, or namespaces hosting them from dynamic hops are made.

Static assignment

You specify an IP address directly.

Dynamic assignment

You specify an IP address indirectly, with namespace and pod selectors, and an optional network attachment definition.

- If the name of a network attachment definition is provided, the external gateway IP address of the network attachment is used.
- If the name of a network attachment definition is not provided, the external gateway IP address for the pod itself is used. However, this approach works only if the pod is configured with **hostNetwork** set to **true**.

8.3. ADMINPOLICYBASEDEXTERNALROUTE OBJECT CONFIGURATION

You can define an **AdminPolicyBasedExternalRoute** object, which is cluster scoped, with the following properties. A namespace can be selected by only one **AdminPolicyBasedExternalRoute** CR at a time.

Table 8.1. **AdminPolicyBasedExternalRoute** object

Field	Type	Description
metadata.name	**string**	Specifies the name of the **AdminPolicyBasedExternalRoute** object.

spec.from	**string**	Specifies a namespace selector that the routing policies apply to. Only **namespaceSelector** is supported for external traffic. For example:
		 `from:`
		 `  namespaceSelector:`
		 `    matchLabels:`
		 `      kubernetes.io/metadata.name: novxlan-externalgw-ecmp-4059`
		 A namespace can only be targeted by one **AdminPolicyBasedExternalRoute** CR. If a namespace is selected by more than one **AdminPolicyBasedExternalRoute** CR, a **failed** error status occurs on the second and subsequent CRs that target the same namespace. To apply updates, you must change the policy itself or related objects to the policy such as target namespaces, pod gateways, or namespaces hosting them from dynamic hops in order for the policy to be re-evaluated and your changes to be applied.

spec.nextHops	**object**	Specifies the destinations where the packets are forwarded to. Must be either or both of **static** and **dynamic**. You must have at least one next hop defined.

Table 8.2. **nextHops** object

Field	Type	Description
static	**array**	Specifies an array of static IP addresses.
dynamic	**array**	Specifies an array of pod selectors corresponding to pods configured with a network attachment definition to use as the external gateway target.

Table 8.3. **nextHops.static** object

Field	Type	Description
ip	**string**	Specifies either an IPv4 or IPv6 address of the next destination hop.
bfdEnabled	**boolean**	Optional: Specifies whether Bi-Directional Forwarding Detection (BFD) is supported by the network. The default value is **false**.

Table 8.4. **nextHops.dynamic** object

Field	Type	Description
podSelector	**string**	Specifies a [set-based] (https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#set-based-requirement) label selector to filter the pods in the namespace that match this network configuration.
namespaceSelector	**string**	Specifies a **set-based** selector to filter the namespaces that the **podSelector** applies to. You must specify a value for this field.
bfdEnabled	**boolean**	Optional: Specifies whether Bi-Directional Forwarding Detection (BFD) is supported by the network. The default value is **false**.
networkAttachmentName	**string**	Optional: Specifies the name of a network attachment definition. The name must match the list of logical networks associated with the pod. If this field is not specified, the host network of the pod is used. However, the pod must be configure as a host network pod to use the host network.

8.3.1. Example secondary external gateway configurations

In the following example, the **AdminPolicyBasedExternalRoute** object configures two static IP addresses as external gateways for pods in namespaces with the **kubernetes.io/metadata.name: novxlan-externalgw-ecmp-4059** label.

```
apiVersion: k8s.ovn.org/v1
kind: AdminPolicyBasedExternalRoute
metadata:
  name: default-route-policy
spec:
  from:
    namespaceSelector:
      matchLabels:
        kubernetes.io/metadata.name: novxlan-externalgw-ecmp-4059
  nextHops:
    static:
    - ip: "172.18.0.8"
    - ip: "172.18.0.9"
```

In the following example, the **AdminPolicyBasedExternalRoute** object configures a dynamic external gateway. The IP addresses used for the external gateway are derived from the additional network attachments associated with each of the selected pods.

```
apiVersion: k8s.ovn.org/v1
kind: AdminPolicyBasedExternalRoute
metadata:
  name: shadow-traffic-policy
spec:
  from:
    namespaceSelector:
      matchLabels:
        externalTraffic: ""
  nextHops:
    dynamic:
    - podSelector:
        matchLabels:
          gatewayPod: ""
      namespaceSelector:
        matchLabels:
          shadowTraffic: ""
      networkAttachmentName: shadow-gateway
    - podSelector:
        matchLabels:
          gigabyteGW: ""
      namespaceSelector:
        matchLabels:
          gatewayNamespace: ""
      networkAttachmentName: gateway
```

In the following example, the **AdminPolicyBasedExternalRoute** object configures both static and dynamic external gateways.

```
apiVersion: k8s.ovn.org/v1
kind: AdminPolicyBasedExternalRoute
metadata:
  name: multi-hop-policy
spec:
  from:
    namespaceSelector:
      matchLabels:
        trafficType: "egress"
  nextHops:
    static:
    - ip: "172.18.0.8"
    - ip: "172.18.0.9"
    dynamic:
    - podSelector:
        matchLabels:
          gatewayPod: ""
      namespaceSelector:
```

```
    matchLabels:
      egressTraffic: ""
  networkAttachmentName: gigabyte
```

8.4. CONFIGURE A SECONDARY EXTERNAL GATEWAY

You can configure an external gateway on the default network for a namespace in your cluster.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.

Procedure

1. Create a YAML file that contains an **AdminPolicyBasedExternalRoute** object.
2. To create an admin policy based external route, enter the following command:

   ```
   $ oc create -f <file>.yaml
   ```

   where:

   <file>
   Specifies the name of the YAML file that you created in the previous step.

   Example output

   ```
   adminpolicybasedexternalroute.k8s.ovn.org/default-route-policy created
   ```

3. To confirm that the admin policy based external route was created, enter the following command:

   ```
   $ oc describe apbexternalroute <name> | tail -n 6
   ```

   where:

   <name>
   Specifies the name of the **AdminPolicyBasedExternalRoute** object.

   Example output

   ```
   Status:
     Last Transition Time:  2023-04-24T15:09:01Z
     Messages:
     Configured external gateway IPs: 172.18.0.8
     Status:  Success
   Events:  <none>
   ```

8.5. ADDITIONAL RESOURCES

- For more information about additional network attachments, see Understanding multiple networks

CHAPTER 9. CONFIGURING AN EGRESS IP ADDRESS

As a cluster administrator, you can configure the OVN-Kubernetes Container Network Interface (CNI) network plugin to assign one or more egress IP addresses to a namespace, or to specific pods in a namespace.

9.1. EGRESS IP ADDRESS ARCHITECTURAL DESIGN AND IMPLEMENTATION

By using the OpenShift Container Platform egress IP address functionality, you can ensure that the traffic from one or more pods in one or more namespaces has a consistent source IP address for services outside the cluster network.

For example, you might have a pod that periodically queries a database that is hosted on a server outside of your cluster. To enforce access requirements for the server, a packet filtering device is configured to allow traffic only from specific IP addresses. To ensure that you can reliably allow access to the server from only that specific pod, you can configure a specific egress IP address for the pod that makes the requests to the server.

An egress IP address assigned to a namespace is different from an egress router, which is used to send traffic to specific destinations.

In some cluster configurations, application pods and ingress router pods run on the same node. If you configure an egress IP address for an application project in this scenario, the IP address is not used when you send a request to a route from the application project.

IMPORTANT

Egress IP addresses must not be configured in any Linux network configuration files, such as **ifcfg-eth0**.

9.1.1. Platform support

The Egress IP address feature that runs on a primary host network is supported on the following platforms:

Platform	Supported
Bare metal	Yes
VMware vSphere	Yes
Red Hat OpenStack Platform (RHOSP)	Yes
Amazon Web Services (AWS)	Yes
Google Cloud	Yes
Microsoft Azure	Yes
IBM Z® and IBM® LinuxONE	Yes

Platform	Supported
IBM Z® and IBM® LinuxONE for Red Hat Enterprise Linux (RHEL) KVM	Yes
IBM Power®	Yes
Nutanix	Yes

The Egress IP address feature that runs on secondary host networks is supported on the following platform:

Platform	Supported
Bare metal	Yes

IMPORTANT

The assignment of egress IP addresses to control plane nodes with the EgressIP feature is not supported on a cluster provisioned on Amazon Web Services (AWS). (BZ#2039656).

9.1.2. Public cloud platform considerations

Typically, public cloud providers place a limit on egress IP addresses. This means that there is a constraint on the absolute number of assignable IP addresses per node for clusters provisioned on public cloud infrastructure. The maximum number of assignable IP addresses per node, or the *IP capacity*, can be described in the following formula:

```
IP capacity = public cloud default capacity - sum(current IP assignments)
```

While the Egress IP addresses capability manages the IP address capacity per node, it is important to plan for this constraint in your deployments. For example, if a public cloud provider limits IP address capacity to 10 IP addresses per node, and you have 8 nodes, the total number of assignable IP addresses is only 80. To achieve a higher IP address capacity, you would need to allocate additional nodes. For example, if you needed 150 assignable IP addresses, you would need to allocate 7 additional nodes.

To confirm the IP capacity and subnets for any node in your public cloud environment, you can enter the **oc get node <node_name> -o yaml** command. The **cloud.network.openshift.io/egress-ipconfig** annotation includes capacity and subnet information for the node.

The annotation value is an array with a single object with fields that provide the following information for the primary network interface:

- **interface**: Specifies the interface ID on AWS and Azure and the interface name on Google Cloud.
- **ifaddr**: Specifies the subnet mask for one or both IP address families.

- **capacity**: Specifies the IP address capacity for the node. On AWS, the IP address capacity is provided per IP address family. On Azure and Google Cloud, the IP address capacity includes both IPv4 and IPv6 addresses.

Automatic attachment and detachment of egress IP addresses for traffic between nodes are available. This allows for traffic from many pods in namespaces to have a consistent source IP address to locations outside of the cluster. This also supports OpenShift SDN and OVN-Kubernetes, which is the default networking plugin in Red Hat OpenShift Networking in OpenShift Container Platform 4.18.

NOTE

The RHOSP egress IP address feature creates a Neutron reservation port called **egressip-<IP address>**. Using the same RHOSP user as the one used for the OpenShift Container Platform cluster installation, you can assign a floating IP address to this reservation port to have a predictable SNAT address for egress traffic. When an egress IP address on an RHOSP network is moved from one node to another, because of a node failover, for example, the Neutron reservation port is removed and recreated. This means that the floating IP association is lost and you need to manually reassign the floating IP address to the new reservation port.

NOTE

When an RHOSP cluster administrator assigns a floating IP to the reservation port, OpenShift Container Platform cannot delete the reservation port. The **CloudPrivateIPConfig** object cannot perform delete and move operations until an RHOSP cluster administrator unassigns the floating IP from the reservation port.

The following examples illustrate the annotation from nodes on several public cloud providers. The annotations are indented for readability.

Example **cloud.network.openshift.io/egress-ipconfig** annotation on AWS

```
cloud.network.openshift.io/egress-ipconfig: [
  {
    "interface":"eni-078d267045138e436",
    "ifaddr":{"ipv4":"10.0.128.0/18"},
    "capacity":{"ipv4":14,"ipv6":15}
  }
]
```

Example **cloud.network.openshift.io/egress-ipconfig** annotation on Google Cloud

```
cloud.network.openshift.io/egress-ipconfig: [
  {
    "interface":"nic0",
    "ifaddr":{"ipv4":"10.0.128.0/18"},
    "capacity":{"ip":14}
  }
]
```

The following sections describe the IP address capacity for supported public cloud environments for use in your capacity calculation.

9.1.2.1. Amazon Web Services (AWS) IP address capacity limits

On AWS, constraints on IP address assignments depend on the instance type configured. For more information, see IP addresses per network interface per instance type

9.1.2.2. Google Cloud IP address capacity limits

On Google Cloud, the networking model implements additional node IP addresses through IP address aliasing, rather than IP address assignments. However, IP address capacity maps directly to IP aliasing capacity.

The following capacity limits exist for IP aliasing assignment:

- Per node, the maximum number of IP aliases, both IPv4 and IPv6, is 100.
- Per VPC, the maximum number of IP aliases is unspecified, but OpenShift Container Platform scalability testing reveals the maximum to be approximately 15,000.

For more information, see Per instance quotas and Alias IP ranges overview.

9.1.2.3. Microsoft Azure IP address capacity limits

On Azure, the following capacity limits exist for IP address assignment:

- Per NIC, the maximum number of assignable IP addresses, for both IPv4 and IPv6, is 256.
- Per virtual network, the maximum number of assigned IP addresses cannot exceed 65,536.

For more information, see Networking limits.

9.1.3. Architectural diagram of an egress IP address configuration

The following diagram depicts an egress IP address configuration. The diagram describes four pods in two different namespaces running on three nodes in a cluster. The nodes are assigned IP addresses from the **192.168.126.0/18** CIDR block on the host network.

Node 1

```
meta:
 name: node1
 labels:
  k8s.ovn.org/egress-assignable: ""
```

Both Node 1 and Node 3 are labeled with **k8s.ovn.org/egress-assignable: ""** and thus available for the assignment of egress IP addresses.

The dashed lines in the diagram depict the traffic flow from pod1, pod2, and pod3 traveling through the pod network to egress the cluster from Node 1 and Node 3. When an external service receives traffic from any of the pods selected by the example **EgressIP** object, the source IP address is either **192.168.126.10** or **192.168.126.102**. The traffic is balanced roughly equally between these two nodes.

Based on the diagram, the following manifest file defines namespaces:

Namespace objects

```
apiVersion: v1
kind: Namespace
metadata:
  name: namespace1
  labels:
    env: prod
---
apiVersion: v1
kind: Namespace
metadata:
  name: namespace2
  labels:
    env: prod
```

Based on the diagram, the following **EgressIP** object describes a configuration that selects all pods in any namespace with the **env** label set to **prod**. The egress IP addresses for the selected pods are **192.168.126.10** and **192.168.126.102**.

EgressIP object

```
apiVersion: k8s.ovn.org/v1
kind: EgressIP
metadata:
  name: egressips-prod
spec:
  egressIPs:
  - 192.168.126.10
  - 192.168.126.102
  namespaceSelector:
    matchLabels:
      env: prod
status:
  items:
  - node: node1
    egressIP: 192.168.126.10
  - node: node3
    egressIP: 192.168.126.102
```

For the configuration in the previous example, OpenShift Container Platform assigns both egress IP addresses to the available nodes. The **status** field reflects whether and where the egress IP addresses are assigned.

9.1.4. Considerations for using an egress IP address on additional network interfaces

In OpenShift Container Platform, egress IP addresses provide administrators a way to control network traffic. Egress IP addresses can be used with a **br-ex** Open vSwitch (OVS) bridge interface and any physical interface that has IP connectivity enabled.

You can inspect your network interface type by running the following command:

```
$ ip -details link show
```

The primary network interface is assigned a node IP address which also contains a subnet mask. Information for this node IP address can be retrieved from the Kubernetes node object for each node within your cluster by inspecting the **k8s.ovn.org/node-primary-ifaddr** annotation. In an IPv4 cluster, this annotation is similar to the following example: **"k8s.ovn.org/node-primary-ifaddr: {"ipv4":"192.168.111.23/24"}"**.

If the egress IP address is not within the subnet of the primary network interface subnet, you can use an egress IP address on another Linux network interface that is not of the primary network interface type. By doing so, OpenShift Container Platform administrators are provided with a greater level of control over networking aspects such as routing, addressing, segmentation, and security policies. This feature provides users with the option to route workload traffic over specific network interfaces for purposes such as traffic segmentation or meeting specialized requirements.

If the egress IP address is not within the subnet of the primary network interface, then the selection of another network interface for egress traffic might occur if they are present on a node.

You can determine which other network interfaces might support egress IP address addresses by inspecting the **k8s.ovn.org/host-cidrs** Kubernetes node annotation. This annotation contains the addresses and subnet mask found for the primary network interface. It also contains additional network interface addresses and subnet mask information. These addresses and subnet masks are assigned to network interfaces that use the longest prefix match routing mechanism to determine which network interface supports the egress IP address.

NOTE

OVN-Kubernetes provides a mechanism to control and direct outbound network traffic from specific namespaces and pods. This ensures that it exits the cluster through a particular network interface and with a specific egress IP address.

As an administrator who wants an egress IP address and traffic to route over a particular interface that is not the primary network interface, you must meet the following conditions:

- OpenShift Container Platform is installed on a bare-metal cluster. This feature is disabled within a cloud or a hypervisor environment.
- Your OpenShift Container Platform pods are not configured as *host-networked*.
- You understand that if a network interface is removed or if the IP address and subnet mask which allows the egress IP address to be hosted on the interface is removed, reconfiguration of the egress IP address occurs. Consequently, the egress IP address might get assigned to another node and interface.
- If you use an Egress IP address on a secondary network interface card (NIC), you must use the Node Tuning Operator to enable IP forwarding on the secondary NIC.
- You configured a NIC with routes by ensuring a gateway exists in the main routing table. As a postinstallation task, Red Hat does not support configuring a NIC on a cluster that uses OVN-Kubernetes.
- Routes associated with an egress interface get copied from the main routing table to the routing table that was created to support the Egress IP object.

9.2. EGRESSIP OBJECT

View the following YAML files to better understand how you can effectively configure an **EgressIP** object to better meet your needs.

The following YAML describes the API for the **EgressIP** object. The scope of the object is cluster-wide and is not created in a namespace.

```
apiVersion: k8s.ovn.org/v1
kind: EgressIP
metadata:
  name: <name>
spec:
  egressIPs:
  - <ip_address>
  namespaceSelector:
    ...
  podSelector:
    ...
```

where:

<name>

The name for the **EgressIPs** object.

<egressIPs>

An array of one or more IP addresses.

<namespaceSelector>

One or more selectors for the namespaces to associate the egress IP addresses with.

<podSelector>

Optional parameter. One or more selectors for pods in the specified namespaces to associate egress IP addresses with. Applying these selectors allows for the selection of a subset of pods within a namespace.

The following YAML describes the stanza for the namespace selector:

Namespace selector stanza

```
namespaceSelector:
  matchLabels:
    <label_name>: <label_value>
```

where:

<namespaceSelector>

One or more matching rules for namespaces. If more than one match rule is provided, all matching namespaces are selected.

The following YAML describes the optional stanza for the pod selector:

Pod selector stanza

```
podSelector:
  matchLabels:
    <label_name>: <label_value>
```

where:

<podSelector>

Optional parameter. One or more matching rules for pods in the namespaces that match the specified **namespaceSelector** rules. If specified, only pods that match are selected. Others pods in the namespace are not selected.

In the following example, the **EgressIP** object associates the **192.168.126.11** and **192.168.126.102** egress IP addresses with pods that have the **app** label set to **web** and are in the namespaces that have the **env** label set to **prod**:

Example EgressIP object

```
apiVersion: k8s.ovn.org/v1
kind: EgressIP
metadata:
  name: egress-group1
spec:
  egressIPs:
  - 192.168.126.11
  - 192.168.126.102
  podSelector:
    matchLabels:
      app: web
  namespaceSelector:
    matchLabels:
      env: prod
```

In the following example, the **EgressIP** object associates the **192.168.127.30** and **192.168.127.40** egress IP addresses with any pods that do not have the **environment** label set to **development**:

Example EgressIP object

```
apiVersion: k8s.ovn.org/v1
kind: EgressIP
metadata:
  name: egress-group2
spec:
  egressIPs:
  - 192.168.127.30
  - 192.168.127.40
  namespaceSelector:
    matchExpressions:
    - key: environment
      operator: NotIn
      values:
      - development
```

9.3. ASSIGNMENT OF EGRESS IPS TO A NAMESPACE, NODES, AND PODS

To assign one or more egress IPs to a namespace or specific pods in a namespace, the following conditions must be satisfied:

- At least one node in your cluster must have the **k8s.ovn.org/egress-assignable: ""** label.
- An **EgressIP** object exists that defines one or more egress IP addresses to use as the source IP address for traffic leaving the cluster from pods in a namespace.

IMPORTANT

If you create **EgressIP** objects prior to labeling any nodes in your cluster for egress IP assignment, OpenShift Container Platform might assign every egress IP address to the first node with the **k8s.ovn.org/egress-assignable: ""** label.

To ensure that egress IP addresses are widely distributed across nodes in the cluster, always apply the label to the nodes you intent to host the egress IP addresses before creating any **EgressIP** objects.

When creating an **EgressIP** object, the following conditions apply to nodes that are labeled with the **k8s.ovn.org/egress-assignable: ""** label:

- An egress IP address is never assigned to more than one node at a time.
- An egress IP address is equally balanced between available nodes that can host the egress IP address.
- If the **spec.EgressIPs** array in an **EgressIP** object specifies more than one IP address, the following conditions apply:
  - No node will ever host more than one of the specified IP addresses.
  - Traffic is balanced roughly equally between the specified IP addresses for a given namespace.
- If a node becomes unavailable, any egress IP addresses assigned to it are automatically reassigned, subject to the previously described conditions.

When a pod matches the selector for multiple **EgressIP** objects, there is no guarantee which of the egress IP addresses that are specified in the **EgressIP** objects is assigned as the egress IP address for the pod.

Additionally, if an **EgressIP** object specifies multiple egress IP addresses, there is no guarantee which of the egress IP addresses might be used. For example, if a pod matches a selector for an **EgressIP** object with two egress IP addresses, **10.10.20.1** and **10.10.20.2**, either might be used for each TCP connection or UDP conversation.

9.4. ASSIGNING AN EGRESS IP ADDRESS TO A NAMESPACE

You can assign one or more egress IP addresses to a namespace or to specific pods in a namespace.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in to the cluster as a cluster administrator.
- Configure at least one node to host an egress IP address.

Procedure

1. Create an **EgressIP** object.

    a. Create a **<egressips_name>.yaml** file where **<egressips_name>** is the name of the object.

    b. In the file that you created, define an **EgressIP** object, as in the following example:

    ```
    apiVersion: k8s.ovn.org/v1
    kind: EgressIP
    metadata:
      name: egress-project1
    spec:
      egressIPs:
      - 192.168.127.10
      - 192.168.127.11
      namespaceSelector:
        matchLabels:
          env: qa
    ```

2. To create the object, enter the following command.

    ```
    $ oc apply -f <egressips_name>.yaml
    ```

    where:

    <egressips_name>

    Replace **<egressips_name>** with the name of the object.

    #### Example output

    ```
    egressips.k8s.ovn.org/<egressips_name> created
    ```

3. Optional: Store the **<egressips_name>.yaml** file so that you can make changes later.

4. Add labels to the namespace that requires egress IP addresses. To add a label to the namespace of an **EgressIP** object defined in step 1, run the following command:

    ```
    $ oc label ns <namespace> env=qa
    ```

    where:

    <namespace>

    Replace **<namespace>** with the namespace that requires egress IP addresses.

Verification

- To show all egress IP addresses that are in use in your cluster, enter the following command:

```
$ oc get egressip -o yaml
```

NOTE

The command **oc get egressip** only returns one egress IP address regardless of how many are configured. This is not a bug and is a limitation of Kubernetes. As a workaround, you can pass in the **-o yaml** or **-o json** flags to return all egress IPs addresses in use.

Example output

```
# ...
spec:
  egressIPs:
  - 192.168.127.10
  - 192.168.127.11
# ...
```

9.5. LABELING A NODE TO HOST EGRESS IP ADDRESSES

You can apply the **k8s.ovn.org/egress-assignable=""** label to a node in your cluster so that OpenShift Container Platform can assign one or more egress IP addresses to the node.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in to the cluster as a cluster administrator.

Procedure

- To label a node so that it can host one or more egress IP addresses, enter the following command:

```
$ oc label nodes <node_name> k8s.ovn.org/egress-assignable="" 1
```

1 The name of the node to label.

TIP

You can alternatively apply the following YAML to add the label to a node:

```
apiVersion: v1
kind: Node
metadata:
  labels:
    k8s.ovn.org/egress-assignable: ""
  name: <node_name>
```

9.6. CONFIGURING DUAL-STACK NETWORKING FOR AN EGRESSIP OBJECT

For a cluster configured for dual-stack networking, you can apply dual-stack networking to a single **EgressIP** object. The **EgressIP** object can then extend dual-stack networking capabilities to a pod.

IMPORTANT

Red Hat does not support creating two **EgressIP** objects to represent dual-stack networking capabilities. For example, specifying IPv4 addresses with one object and using another object to specify IPv6 addresses. This configuration limit impacts address-type assignments to pods.

Prerequisites

- You created two egress nodes so that an **EgressIP** object can allocate IPv4 addresses to one node and IPv6 addresses to the other node. For more information, see "Assignment of egress IP addresses to nodes".

Procedure

- Create an **EgressIP** object and configure IPv4 and IPv6 addresses for the object. The following example **EgressIP** object uses selectors to identify which pods use the specified egress IP addresses for their outbound traffic:

```
kind: EgressIP
metadata:
  name: egressip-dual
spec:
  egressIPs:
    - 192.168.118.30
    - 2600:52:7:94::30
  namespaceSelector:
    matchLabels:
      env: qa
  podSelector:
    matchLabels:
      egressip: ds
# ...
```

Verification

1. Create a **Pod** manifest file to test and validate your **EgressIP** object. The pod serves as a client workload that sends outbound traffic to verify that your **EgressIP** policy works as expected.

```
apiVersion: v1
kind: Pod
metadata:
  name: ubi-egressip-pod
  namespace: test
  labels:
    egressip: ds
spec:
  containers:
```

```
  - name: fedora-curl
    image: registry.redhat.io/ubi9/ubi
    command: ["/bin/bash", "-c", "sleep infinity"]
# ...
```

where:

<labels>

Sets custom identifiers so that the **EgressIP** object can use these labels to apply egress IP address to target pods.

2. Run a **curl** request from inside a pod to an external server. This action verifies that outbound traffic correctly uses an address that you specified in the **EgressIP** object.

```
$ curl <ipv_address>
```

where:

<ipv_address>

Depending on the **EgressIP** object, enter an IPv4 or IPv6 address.

9.7. UNDERSTANDING EGRESSIP FAILOVER CONTROL

The **reachabilityTotalTimeoutSeconds** parameter controls how quickly the system detects a failing **egressIP** node and initiates a failover. This parameter directly determines the maximum time the platform waits before declaring a node unreachable.

IMPORTANT

When you configure **egressIP** with multiple egress nodes, the complete failover time from node failure to recovery on a new node is expected to be on the order of seconds or longer. This is because the new IP assignment can only begin after the **reachabilityTotalTimeoutSeconds** period has fully elapsed without a successful check.

To ensure traffic uses the correct external path, **egressIP** traffic on a node will always egress through the network interface on which the **egressIP** address has been assigned.

9.7.1. Configuring the EgressIP failover time limit

Follow this procedure to configure the **reachabilityTotalTimeoutSeconds** parameter and control how quickly the system detects a failing **egressIP** node and initiates a failover.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in to the cluster as a cluster administrator.

Procedure

1. Edit the **Network** custom resource by running the following command:

```
$ oc edit network.operator cluster
```

2. Navigate to the **egressIPConfig: {}** section under **spec:defaultNetwork:ovnKubernetesConfig:**

3. Modify the block to include the **reachabilityTotalTimeoutSeconds** parameter with your chosen value, 5 seconds for example. Make sure to use the correct indentation:

```
defaultNetwork:
  ovnKubernetesConfig:
    egressIPConfig:
      reachabilityTotalTimeoutSeconds: 5
```

NOTE

The value must be an integer between 0 and 60. For details on possible values, see the "EgressIP failover settings" section.

4. Save and exit the editor. The operator automatically applies the changes.

Verification

1. Verify that the system correctly accepted the **reachabilityTotalTimeoutSeconds** parameter by running the following command:

```
$ oc get network.operator cluster -o yaml
```

2. Inspect the output and confirm that the **reachabilityTotalTimeoutSeconds** parameter is correctly nested under **spec:defaultNetwork:ovnKubernetesConfig:egressIPConfig:** with your intended value:

```
# ...
spec:
  # ...
  defaultNetwork:
    ovnKubernetesConfig:
      egressIPConfig:
        reachabilityTotalTimeoutSeconds: 5
      gatewayConfig:
# ...
```

9.7.2. EgressIP failover settings

The **reachabilityTotalTimeoutSeconds** parameter defines the total time limit in seconds for the platform health check process before a node is declared down.

The following table summarizes the acceptable values and their implications:

Parameter Value (Seconds)	Effect on reachability check	Failover impact and use case
0	Disables the reachability check.	No automatic failover: Use only if an external system handles node health monitoring and failover. The platform will not automatically react to node failures.
1 - 60	Sets the total time limit for reachability probing.	Directly controls detection time: This value defines the lower limit for your overall failover time. A smaller value leads to faster failover but might increase network traffic. Default: 1 second. The maximum accepted integer value is 60.

9.8. ADDITIONAL RESOURCES

- LabelSelector meta/v1
- LabelSelectorRequirement meta/v1

CHAPTER 10. CONFIGURING AN EGRESS SERVICE

As a cluster administrator, you can configure egress traffic for pods behind a load balancer service by using an egress service.

IMPORTANT

Egress service is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

You can use the **EgressService** custom resource (CR) to manage egress traffic in the following ways:

- Assign a load balancer service IP address as the source IP address for egress traffic for pods behind the load balancer service.
  Assigning the load balancer IP address as the source IP address in this context is useful to present a single point of egress and ingress. For example, in some scenarios, an external system communicating with an application behind a load balancer service can expect the source and destination IP address for the application to be the same.

  NOTE

  When you assign the load balancer service IP address to egress traffic for pods behind the service, OVN-Kubernetes restricts the ingress and egress point to a single node. This limits the load balancing of traffic that MetalLB typically provides.

- Assign the egress traffic for pods behind a load balancer to a different network than the default node network.
  This is useful to assign the egress traffic for applications behind a load balancer to a different network than the default network. Typically, the different network is implemented by using a VRF instance associated with a network interface.

10.1. EGRESS SERVICE CUSTOM RESOURCE

Define the configuration for an egress service in an **EgressService** custom resource. The following YAML describes the fields for the configuration of an egress service:

```
apiVersion: k8s.ovn.org/v1
kind: EgressService
metadata:
  name: <egress_service_name> 1
  namespace: <namespace> 2
spec:
  sourceIPBy: <egress_traffic_ip> 3
  nodeSelector: 4
```

```
    matchLabels:
      node-role.kubernetes.io/<role>: ""
  network: <egress_traffic_network> 5
```

1. Specify the name for the egress service. The name of the **EgressService** resource must match the name of the load-balancer service that you want to modify.

2. Specify the namespace for the egress service. The namespace for the **EgressService** must match the namespace of the load-balancer service that you want to modify. The egress service is namespace-scoped.

3. Specify the source IP address of egress traffic for pods behind a service. Valid values are **LoadBalancerIP** or **Network**. Use the **LoadBalancerIP** value to assign the **LoadBalancer** service ingress IP address as the source IP address for egress traffic. Specify **Network** to assign the network interface IP address as the source IP address for egress traffic.

4. Optional: If you use the **LoadBalancerIP** value for the **sourceIPBy** specification, a single node handles the **LoadBalancer** service traffic. Use the **nodeSelector** field to limit which node can be assigned this task. When a node is selected to handle the service traffic, OVN-Kubernetes labels the node in the following format: **egress-service.k8s.ovn.org/<svc-namespace>-<svc-name>: ""**. When the **nodeSelector** field is not specified, any node can manage the **LoadBalancer** service traffic.

5. Optional: Specify the routing table ID for egress traffic. Ensure that the value matches the **route-table-id** ID defined in the **NodeNetworkConfigurationPolicy** resource. If you do not include the **network** specification, the egress service uses the default host network.

Example egress service specification

```
apiVersion: k8s.ovn.org/v1
kind: EgressService
metadata:
  name: test-egress-service
  namespace: test-namespace
spec:
  sourceIPBy: "LoadBalancerIP"
  nodeSelector:
    matchLabels:
      vrf: "true"
  network: "2"
```

10.2. DEPLOYING AN EGRESS SERVICE

You can deploy an egress service to manage egress traffic for pods behind a **LoadBalancer** service.

The following example configures the egress traffic to have the same source IP address as the ingress IP address of the **LoadBalancer** service.

Prerequisites

- Install the OpenShift CLI (**oc**).

- Log in as a user with **cluster-admin** privileges.

- You configured MetalLB **BGPPeer** resources.

Procedure

1. Create an **IPAddressPool** CR with the desired IP for the service:

   a. Create a file, such as **ip-addr-pool.yaml**, with content like the following example:

   ```
   apiVersion: metallb.io/v1beta1
   kind: IPAddressPool
   metadata:
     name: example-pool
     namespace: metallb-system
   spec:
     addresses:
     - 172.19.0.100/32
   ```

   b. Apply the configuration for the IP address pool by running the following command:

   ```
   $ oc apply -f ip-addr-pool.yaml
   ```

2. Create **Service** and **EgressService** CRs:

   a. Create a file, such as **service-egress-service.yaml**, with content like the following example:

   ```
   apiVersion: v1
   kind: Service
   metadata:
     name: example-service
     namespace: example-namespace
     annotations:
       metallb.io/address-pool: example-pool 1
   spec:
     selector:
       app: example
     ports:
       - name: http
         protocol: TCP
         port: 8080
         targetPort: 8080
     type: LoadBalancer
   ---
   apiVersion: k8s.ovn.org/v1
   kind: EgressService
   metadata:
     name: example-service
     namespace: example-namespace
   spec:
     sourceIPBy: "LoadBalancerIP" 2
     nodeSelector: 3
       matchLabels:
         node-role.kubernetes.io/worker: ""
   ```

1 The **LoadBalancer** service uses the IP address assigned by MetalLB from the **example-pool** IP address pool.

2 This example uses the **LoadBalancerIP** value to assign the ingress IP address of the **LoadBalancer** service as the source IP address of egress traffic.

3 When you specify the **LoadBalancerIP** value, a single node handles the **LoadBalancer** service's traffic. In this example, only nodes with the **worker** label can be selected to handle the traffic. When a node is selected, OVN-Kubernetes labels the node in the following format **egress-service.k8s.ovn.org/<svc-namespace>-<svc-name>: ""**.

NOTE

If you use the **sourceIPBy: "LoadBalancerIP"** setting, you must specify the load-balancer node in the **BGPAdvertisement** custom resource (CR).

b. Apply the configuration for the service and egress service by running the following command:

```
$ oc apply -f service-egress-service.yaml
```

3. Create a **BGPAdvertisement** CR to advertise the service:

a. Create a file, such as **service-bgp-advertisement.yaml**, with content like the following example:

```
apiVersion: metallb.io/v1beta1
kind: BGPAdvertisement
metadata:
  name: example-bgp-adv
  namespace: metallb-system
spec:
  ipAddressPools:
  - example-pool
  nodeSelectors:
  - matchLabels:
      egress-service.k8s.ovn.org/example-namespace-example-service: "" 1
```

1 In this example, the **EgressService** CR configures the source IP address for egress traffic to use the load-balancer service IP address. Therefore, you must specify the load-balancer node for return traffic to use the same return path for the traffic originating from the pod.

Verification

1. Verify that you can access the application endpoint of the pods running behind the MetalLB service by running the following command:

```
$ curl <external_ip_address>:<port_number> 1
```

1 Update the external IP address and port number to suit your application endpoint.

2. If you assigned the **LoadBalancer** service's ingress IP address as the source IP address for egress traffic, verify this configuration by using tools such as **tcpdump** to analyze packets received at the external client.

Additional resources

- Exposing a service through a network VRF
- Example: Network interface with a VRF instance node network configuration policy
- Managing symmetric routing with MetalLB
- About virtual routing and forwarding

CHAPTER 11. CONSIDERATIONS FOR THE USE OF AN EGRESS ROUTER POD

11.1. ABOUT AN EGRESS ROUTER POD

The OpenShift Container Platform egress router pod redirects traffic to a specified remote server from a private source IP address that is not used for any other purpose. An egress router pod can send network traffic to servers that are set up to allow access only from specific IP addresses.

NOTE

The egress router pod is not intended for every outgoing connection. Creating large numbers of egress router pods can exceed the limits of your network hardware. For example, creating an egress router pod for every project or application could exceed the number of local MAC addresses that the network interface can handle before reverting to filtering MAC addresses in software.

IMPORTANT

The egress router image is not compatible with Amazon AWS, Azure Cloud, or any other cloud platform that does not support layer 2 manipulations due to their incompatibility with macvlan traffic.

11.1.1. Egress router modes

In *redirect mode*, an egress router pod configures **iptables** rules to redirect traffic from its own IP address to one or more destination IP addresses. Client pods that need to use the reserved source IP address must be configured to access the service for the egress router rather than connecting directly to the destination IP. You can access the destination service and port from the application pod by using the **curl** command. For example:

```
$ curl <router_service_IP> <port>
```

NOTE

The egress router CNI plugin supports redirect mode only. The egress router CNI plugin does not support HTTP proxy mode or DNS proxy mode.

11.1.2. Egress router pod implementation

The egress router implementation uses the egress router Container Network Interface (CNI) plugin. The plugin adds a secondary network interface to a pod.

An egress router is a pod that has two network interfaces. For example, the pod can have **eth0** and **net1** network interfaces. The **eth0** interface is on the cluster network and the pod continues to use the interface for ordinary cluster-related network traffic. The **net1** interface is on a secondary network and has an IP address and gateway for that network. Other pods in the OpenShift Container Platform cluster can access the egress router service and the service enables the pods to access external services. The egress router acts as a bridge between pods and an external system.

Traffic that leaves the egress router exits through a node, but the packets have the MAC address of the **net1** interface from the egress router pod.

When you add an egress router custom resource, the Cluster Network Operator creates the following objects:

- The network attachment definition for the **net1** secondary network interface of the pod.
- A deployment for the egress router.

If you delete an egress router custom resource, the Operator deletes the two objects in the preceding list that are associated with the egress router.

11.1.3. Deployment considerations

An egress router pod adds an additional IP address and MAC address to the primary network interface of the node. As a result, you might need to configure your hypervisor or cloud provider to allow the additional address.

Red Hat OpenStack Platform (RHOSP)

If you deploy OpenShift Container Platform on RHOSP, you must allow traffic from the IP and MAC addresses of the egress router pod on your OpenStack environment. If you do not allow the traffic, then communication will fail:

```
$ openstack port set --allowed-address \
  ip_address=<ip_address>,mac_address=<mac_address> <neutron_port_uuid>
```

VMware vSphere

If you are using VMware vSphere, see the VMware documentation for securing vSphere standard switches. View and change VMware vSphere default settings by selecting the host virtual switch from the vSphere Web Client.

Specifically, ensure that the following are enabled:

- MAC Address Changes
- Forged Transits
- Promiscuous Mode Operation

11.1.4. Failover configuration

To avoid downtime, the Cluster Network Operator deploys the egress router pod as a deployment resource. The deployment name is **egress-router-cni-deployment**. The pod that corresponds to the deployment has a label of **app=egress-router-cni**.

To create a new service for the deployment, use the **oc expose deployment/egress-router-cni-deployment --port <port_number>** command or create a file like the following example:

```
apiVersion: v1
kind: Service
metadata:
  name: app-egress
spec:
  ports:
  - name: tcp-8080
    protocol: TCP
    port: 8080
  - name: tcp-8443
    protocol: TCP
    port: 8443
  - name: udp-80
    protocol: UDP
    port: 80
  type: ClusterIP
  selector:
    app: egress-router-cni
```

11.2. ADDITIONAL RESOURCES

- Deploying an egress router in redirection mode

CHAPTER 12. DEPLOYING AN EGRESS ROUTER POD IN REDIRECT MODE

As a cluster administrator, you can deploy an egress router pod to redirect traffic to specified destination IP addresses from a reserved source IP address.

The egress router implementation uses the egress router Container Network Interface (CNI) plugin.

12.1. EGRESS ROUTER CUSTOM RESOURCE

Define the configuration for an egress router pod in an egress router custom resource. The following YAML describes the fields for the configuration of an egress router in redirect mode:

```
apiVersion: network.operator.openshift.io/v1
kind: EgressRouter
metadata:
  name: <egress_router_name>
  namespace: <namespace>  (1)
spec:
  addresses: [  (2)
    {
      ip: "<egress_router>",  (3)
      gateway: "<egress_gateway>"  (4)
    }
  ]
  mode: Redirect
  redirect: {
    redirectRules: [  (5)
      {
        destinationIP: "<egress_destination>",
        port: <egress_router_port>,
        targetPort: <target_port>,  (6)
        protocol: <network_protocol>  (7)
      },
      ...
    ],
    fallbackIP: "<egress_destination>" (8)
  }
```

1. Optional: The **namespace** field specifies the namespace to create the egress router in. If you do not specify a value in the file or on the command line, the **default** namespace is used.

2. The **addresses** field specifies the IP addresses to configure on the secondary network interface.

3. The **ip** field specifies the reserved source IP address and netmask from the physical network that the node is on to use with egress router pod. Use CIDR notation to specify the IP address and netmask.

4. The **gateway** field specifies the IP address of the network gateway.

5. Optional: The **redirectRules** field specifies a combination of egress destination IP address, egress router port, and protocol. Incoming connections to the egress router on the specified port and protocol are routed to the destination IP address.

6 Optional: The **targetPort** field specifies the network port on the destination IP address. If this field is not specified, traffic is routed to the same network port that it arrived on.

7 The **protocol** field supports TCP, UDP, or SCTP.

8 Optional: The **fallbackIP** field specifies a destination IP address. If you do not specify any redirect rules, the egress router sends all traffic to this fallback IP address. If you specify redirect rules, any connections to network ports that are not defined in the rules are sent by the egress router to this fallback IP address. If you do not specify this field, the egress router rejects connections to network ports that are not defined in the rules.

Example egress router specification

```
apiVersion: network.operator.openshift.io/v1
kind: EgressRouter
metadata:
  name: egress-router-redirect
spec:
  networkInterface: {
    macvlan: {
      mode: "Bridge"
    }
  }
  addresses: [
    {
      ip: "192.168.12.99/24",
      gateway: "192.168.12.1"
    }
  ]
  mode: Redirect
  redirect: {
    redirectRules: [
      {
        destinationIP: "10.0.0.99",
        port: 80,
        protocol: UDP
      },
      {
        destinationIP: "203.0.113.26",
        port: 8080,
        targetPort: 80,
        protocol: TCP
      },
      {
        destinationIP: "203.0.113.27",
        port: 8443,
        targetPort: 443,
        protocol: TCP
      }
    ]
  }
```

12.2. DEPLOYING AN EGRESS ROUTER IN REDIRECT MODE

You can deploy an egress router to redirect traffic from its own reserved source IP address to one or more destination IP addresses.

After you add an egress router, the client pods that need to use the reserved source IP address must be modified to connect to the egress router rather than connecting directly to the destination IP.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in as a user with **cluster-admin** privileges.

Procedure

1. Create an egress router definition.
2. To ensure that other pods can find the IP address of the egress router pod, create a service that uses the egress router, as in the following example:

```
apiVersion: v1
kind: Service
metadata:
  name: egress-1
spec:
  ports:
  - name: web-app
    protocol: TCP
    port: 8080
  type: ClusterIP
  selector:
    app: egress-router-cni (1)
```

(1) Specify the label for the egress router. The value shown is added by the Cluster Network Operator and is not configurable.

After you create the service, your pods can connect to the service. The egress router pod redirects traffic to the corresponding port on the destination IP address. The connections originate from the reserved source IP address.

Verification

To verify that the Cluster Network Operator started the egress router, complete the following procedure:

1. View the network attachment definition that the Operator created for the egress router:

```
$ oc get network-attachment-definition egress-router-cni-nad
```

The name of the network attachment definition is not configurable.

Example output

```
NAME                    AGE
egress-router-cni-nad   18m
```

2. View the deployment for the egress router pod:

```
$ oc get deployment egress-router-cni-deployment
```

The name of the deployment is not configurable.

Example output

```
NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
egress-router-cni-deployment   1/1     1            1           18m
```

3. View the status of the egress router pod:

```
$ oc get pods -l app=egress-router-cni
```

Example output

```
NAME                                            READY   STATUS    RESTARTS   AGE
egress-router-cni-deployment-575465c75c-qkq6m   1/1     Running   0          18m
```

4. View the logs and the routing table for the egress router pod.

a. Get the node name for the egress router pod:

```
$ POD_NODENAME=$(oc get pod -l app=egress-router-cni -o jsonpath="
{.items[0].spec.nodeName}")
```

b. Enter into a debug session on the target node. This step instantiates a debug pod called **<node_name>-debug**:

```
$ oc debug node/$POD_NODENAME
```

c. Set **/host** as the root directory within the debug shell. The debug pod mounts the root file system of the host in **/host** within the pod. By changing the root directory to **/host**, you can run binaries from the executable paths of the host:

```
# chroot /host
```

d. From within the **chroot** environment console, display the egress router logs:

```
# cat /tmp/egress-router-log
```

Example output

```
2021-04-26T12:27:20Z [debug] Called CNI ADD
2021-04-26T12:27:20Z [debug] Gateway: 192.168.12.1
2021-04-26T12:27:20Z [debug] IP Source Addresses: [192.168.12.99/24]
2021-04-26T12:27:20Z [debug] IP Destinations: [80 UDP 10.0.0.99/30 8080 TCP
203.0.113.26/30 80 8443 TCP 203.0.113.27/30 443]
2021-04-26T12:27:20Z [debug] Created macvlan interface
2021-04-26T12:27:20Z [debug] Renamed macvlan to "net1"
2021-04-26T12:27:20Z [debug] Adding route to gateway 192.168.12.1 on macvlan interface
```

```
2021-04-26T12:27:20Z [debug] deleted default route {Ifindex: 3 Dst: <nil> Src: <nil> Gw:
10.128.10.1 Flags: [] Table: 254}
2021-04-26T12:27:20Z [debug] Added new default route with gateway 192.168.12.1
2021-04-26T12:27:20Z [debug] Added iptables rule: iptables -t nat PREROUTING -i eth0 -p
UDP --dport 80 -j DNAT --to-destination 10.0.0.99
2021-04-26T12:27:20Z [debug] Added iptables rule: iptables -t nat PREROUTING -i eth0 -p
TCP --dport 8080 -j DNAT --to-destination 203.0.113.26:80
2021-04-26T12:27:20Z [debug] Added iptables rule: iptables -t nat PREROUTING -i eth0 -p
TCP --dport 8443 -j DNAT --to-destination 203.0.113.27:443
2021-04-26T12:27:20Z [debug] Added iptables rule: iptables -t nat -o net1 -j SNAT --to-
source 192.168.12.99
```

The logging file location and logging level are not configurable when you start the egress router by creating an **EgressRouter** object as described in this procedure.

e. From within the **chroot** environment console, get the container ID:

```
# crictl ps --name egress-router-cni-pod | awk '{print $1}'
```

Example output

```
CONTAINER
bac9fae69ddb6
```

f. Determine the process ID of the container. In this example, the container ID is **bac9fae69ddb6**:

```
# crictl inspect -o yaml bac9fae69ddb6 | grep 'pid:' | awk '{print $2}'
```

Example output

```
68857
```

g. Enter the network namespace of the container:

```
# nsenter -n -t 68857
```

h. Display the routing table:

```
# ip route
```

In the following example output, the **net1** network interface is the default route. Traffic for the cluster network uses the **eth0** network interface. Traffic for the **192.168.12.0/24** network uses the **net1** network interface and originates from the reserved source IP address **192.168.12.99**. The pod routes all other traffic to the gateway at IP address **192.168.12.1**. Routing for the service network is not shown.

Example output

```
default via 192.168.12.1 dev net1
10.128.10.0/23 dev eth0 proto kernel scope link src 10.128.10.18
192.168.12.0/24 dev net1 proto kernel scope link src 192.168.12.99
192.168.12.1 dev net1
```

CHAPTER 13. ENABLING MULTICAST FOR A PROJECT

13.1. ABOUT MULTICAST

With IP multicast, data is broadcast to many IP addresses simultaneously.

IMPORTANT

- At this time, multicast is best used for low-bandwidth coordination or service discovery and not a high-bandwidth solution.
- By default, network policies affect all connections in a namespace. However, multicast is unaffected by network policies. If multicast is enabled in the same namespace as your network policies, it is always allowed, even if there is a **deny-all** network policy.
- Cluster administrators must consider the implications of the exemption of multicast from network policies before enabling it.

Multicast traffic between OpenShift Container Platform pods is disabled by default. If you are using the OVN-Kubernetes network plugin, you can enable multicast on a per-project basis.

13.2. ENABLING MULTICAST BETWEEN PODS

You can enable multicast between pods for your project.

Prerequisites

- Install the OpenShift CLI (**oc**).
- You must log in to the cluster with a user that has the **cluster-admin** role.

Procedure

- Run the following command to enable multicast for a project. Replace **<namespace>** with the namespace for the project you want to enable multicast for.

```
$ oc annotate namespace <namespace> \
    k8s.ovn.org/multicast-enabled=true
```

TIP

You can alternatively apply the following YAML to add the annotation:

```
apiVersion: v1
kind: Namespace
metadata:
  name: <namespace>
  annotations:
    k8s.ovn.org/multicast-enabled: "true"
```

Verification

To verify that multicast is enabled for a project, complete the following procedure:

1. Change your current project to the project that you enabled multicast for. Replace **<project>** with the project name.

   ```
   $ oc project <project>
   ```

2. Create a pod to act as a multicast receiver:

   ```
   $ cat <<EOF| oc create -f -
   apiVersion: v1
   kind: Pod
   metadata:
     name: mlistener
     labels:
       app: multicast-verify
   spec:
     containers:
       - name: mlistener
         image: registry.access.redhat.com/ubi9
         command: ["/bin/sh", "-c"]
         args:
           ["dnf -y install socat hostname && sleep inf"]
         ports:
           - containerPort: 30102
             name: mlistener
             protocol: UDP
   EOF
   ```

3. Create a pod to act as a multicast sender:

   ```
   $ cat <<EOF| oc create -f -
   apiVersion: v1
   kind: Pod
   metadata:
     name: msender
     labels:
       app: multicast-verify
   spec:
     containers:
       - name: msender
         image: registry.access.redhat.com/ubi9
         command: ["/bin/sh", "-c"]
         args:
           ["dnf -y install socat && sleep inf"]
   EOF
   ```

4. In a new terminal window or tab, start the multicast listener.

   a. Get the IP address for the Pod:

      ```
      $ POD_IP=$(oc get pods mlistener -o jsonpath='{.status.podIP}')
      ```

b. Start the multicast listener by entering the following command:

```
$ oc exec mlistener -i -t -- \
    socat UDP4-RECVFROM:30102,ip-add-membership=224.1.0.1:$POD_IP,fork
EXEC:hostname
```

5. Start the multicast transmitter.

a. Get the pod network IP address range:

```
$ CIDR=$(oc get Network.config.openshift.io cluster \
    -o jsonpath='{.status.clusterNetwork[0].cidr}')
```

b. To send a multicast message, enter the following command:

```
$ oc exec msender -i -t -- \
    /bin/bash -c "echo | socat STDIO UDP4-
DATAGRAM:224.1.0.1:30102,range=$CIDR,ip-multicast-ttl=64"
```

If multicast is working, the previous command returns the following output:

```
mlistener
```

CHAPTER 14. DISABLING MULTICAST FOR A PROJECT

14.1. DISABLING MULTICAST BETWEEN PODS

You can disable multicast between pods for your project.

Prerequisites

- Install the OpenShift CLI (**oc**).
- You must log in to the cluster with a user that has the **cluster-admin** role.

Procedure

- Disable multicast by running the following command:

```
$ oc annotate namespace <namespace> \ (1)
    k8s.ovn.org/multicast-enabled-
```

(1) The **namespace** for the project you want to disable multicast for.

TIP

You can alternatively apply the following YAML to delete the annotation:

```
apiVersion: v1
kind: Namespace
metadata:
  name: <namespace>
  annotations:
    k8s.ovn.org/multicast-enabled: null
```

CHAPTER 15. TRACKING NETWORK FLOWS

As a cluster administrator, you can collect information about pod network flows from your cluster to assist with the following areas:

- Monitor ingress and egress traffic on the pod network.
- Troubleshoot performance issues.
- Gather data for capacity planning and security audits.

When you enable the collection of the network flows, only the metadata about the traffic is collected. For example, packet data is not collected, but the protocol, source address, destination address, port numbers, number of bytes, and other packet-level information is collected.

The data is collected in one or more of the following record formats:

- NetFlow
- sFlow
- IPFIX

When you configure the Cluster Network Operator (CNO) with one or more collector IP addresses and port numbers, the Operator configures Open vSwitch (OVS) on each node to send the network flows records to each collector.

You can configure the Operator to send records to more than one type of network flow collector. For example, you can send records to NetFlow collectors and also send records to sFlow collectors.

When OVS sends data to the collectors, each type of collector receives identical records. For example, if you configure two NetFlow collectors, OVS on a node sends identical records to the two collectors. If you also configure two sFlow collectors, the two sFlow collectors receive identical records. However, each collector type has a unique record format.

Collecting the network flows data and sending the records to collectors affects performance. Nodes process packets at a slower rate. If the performance impact is too great, you can delete the destinations for collectors to disable collecting network flows data and restore performance.

NOTE

Enabling network flow collectors might have an impact on the overall performance of the cluster network.

15.1. NETWORK OBJECT CONFIGURATION FOR TRACKING NETWORK FLOWS

The fields for configuring network flows collectors in the Cluster Network Operator (CNO) are shown in the following table:

Table 15.1. Network flows configuration

Field	Type	Description
metadata.name	**string**	The name of the CNO object. This name is always **cluster**.
spec.exportNetworkFlows	**object**	One or more of **netFlow**, **sFlow**, or **ipfix**.
spec.exportNetworkFlows.netFlow.collectors	**array**	A list of IP address and network port pairs for up to 10 collectors.
spec.exportNetworkFlows.sFlow.collectors	**array**	A list of IP address and network port pairs for up to 10 collectors.
spec.exportNetworkFlows.ipfix.collectors	**array**	A list of IP address and network port pairs for up to 10 collectors.

After applying the following manifest to the CNO, the Operator configures Open vSwitch (OVS) on each node in the cluster to send network flows records to the NetFlow collector that is listening at **192.168.1.99:2056**.

Example configuration for tracking network flows

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  exportNetworkFlows:
    netFlow:
      collectors:
        - 192.168.1.99:2056
```

15.2. ADDING DESTINATIONS FOR NETWORK FLOWS COLLECTORS

As a cluster administrator, you can configure the Cluster Network Operator (CNO) to send network flows metadata about the pod network to a network flows collector.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.
- You have a network flows collector and know the IP address and port that it listens on.

Procedure

1. Create a patch file that specifies the network flows collector type and the IP address and port information of the collectors:

   ```
   spec:
     exportNetworkFlows:
       netFlow:
         collectors:
           - 192.168.1.99:2056
   ```

2. Configure the CNO with the network flows collectors:

   ```
   $ oc patch network.operator cluster --type merge -p "$(cat <file_name>.yaml)"
   ```

   Example output

   ```
   network.operator.openshift.io/cluster patched
   ```

Verification

Verification is not typically necessary. You can run the following command to confirm that Open vSwitch (OVS) on each node is configured to send network flows records to one or more collectors.

1. View the Operator configuration to confirm that the **exportNetworkFlows** field is configured:

   ```
   $ oc get network.operator cluster -o jsonpath="{.spec.exportNetworkFlows}"
   ```

   Example output

   ```
   {"netFlow":{"collectors":["192.168.1.99:2056"]}}
   ```

2. View the network flows configuration in OVS from each node:

   ```
   $ for pod in $(oc get pods -n openshift-ovn-kubernetes -l app=ovnkube-node -o
   jsonpath='{range@.items[*]}{.metadata.name}{"\n"}{end}');
     do ;
       echo;
       echo $pod;
       oc -n openshift-ovn-kubernetes exec -c ovnkube-controller $pod \
         -- bash -c 'for type in ipfix sflow netflow ; do ovs-vsctl find $type ; done';
   done
   ```

   Example output

   ```
   ovnkube-node-xrn4p
   _uuid               : a4d2aaca-5023-4f3d-9400-7275f92611f9
   active_timeout      : 60
   add_id_to_interface : false
   engine_id           : []
   engine_type         : []
   external_ids        : {}
   targets             : ["192.168.1.99:2056"]

   ovnkube-node-z4vq9
   ```

```
_uuid               : 61d02fdb-9228-4993-8ff5-b27f01a29bd6
active_timeout      : 60
add_id_to_interface : false
engine_id           : []
engine_type         : []
external_ids        : {}
targets             : ["192.168.1.99:2056"]-

...
```

15.3. DELETING ALL DESTINATIONS FOR NETWORK FLOWS COLLECTORS

As a cluster administrator, you can configure the Cluster Network Operator (CNO) to stop sending network flows metadata to a network flows collector.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.

Procedure

1. Remove all network flows collectors:

   ```
   $ oc patch network.operator cluster --type='json' \
       -p='[{"op":"remove", "path":"/spec/exportNetworkFlows"}]'
   ```

   Example output

   ```
   network.operator.openshift.io/cluster patched
   ```

15.4. ADDITIONAL RESOURCES

- Network [operator.openshift.io/v1]

CHAPTER 16. CONFIGURING HYBRID NETWORKING

As a cluster administrator, you can configure the Red Hat OpenShift Networking OVN-Kubernetes network plugin to allow Linux and Windows nodes to host Linux and Windows workloads, respectively.

16.1. CONFIGURING HYBRID NETWORKING WITH OVN-KUBERNETES

You can configure your cluster to use hybrid networking with the OVN-Kubernetes network plugin. This allows a hybrid cluster that supports different node networking configurations.

NOTE

This configuration is necessary to run both Linux and Windows nodes in the same cluster.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in to the cluster as a user with **cluster-admin** privileges.
- Ensure that the cluster uses the OVN-Kubernetes network plugin.

Procedure

1. To configure the OVN-Kubernetes hybrid network overlay, enter the following command:

```
$ oc patch networks.operator.openshift.io cluster --type=merge \
  -p '{
    "spec":{
      "defaultNetwork":{
        "ovnKubernetesConfig":{
          "hybridOverlayConfig":{
            "hybridClusterNetwork":[
              {
                "cidr": "<cidr>",
                "hostPrefix": <prefix>
              }
            ],
            "hybridOverlayVXLANPort": <overlay_port>
          }
        }
      }
    }
  }'
```

where:

cidr

Specify the CIDR configuration used for nodes on the additional overlay network. This CIDR must not overlap with the cluster network CIDR.

hostPrefix

Specifies the subnet prefix length to assign to each individual node. For example, if **hostPrefix** is set to **23**, then each node is assigned a **/23** subnet out of the given **cidr**, which allows for 510 (2^(32 - 23) - 2) pod IP addresses. If you are required to provide access to nodes from an external network, configure load balancers and routers to manage the traffic.

hybridOverlayVXLANPort

Specify a custom VXLAN port for the additional overlay network. This is required for running Windows nodes in a cluster installed on vSphere, and must not be configured for any other cloud provider. The custom port can be any open port excluding the default **6081** port. For more information on this requirement, see Pod-to-pod connectivity between hosts is broken in the Microsoft documentation.

NOTE

Windows Server Long-Term Servicing Channel (LTSC): Windows Server 2019 is not supported on clusters with a custom **hybridOverlayVXLANPort** value because this Windows server version does not support selecting a custom VXLAN port.

Example output

```
network.operator.openshift.io/cluster patched
```

2. To confirm that the configuration is active, enter the following command. It can take several minutes for the update to apply.

```
$ oc get network.operator.openshift.io -o jsonpath="
{.items[0].spec.defaultNetwork.ovnKubernetesConfig}"
```

16.2. ADDITIONAL RESOURCES

- Understanding Windows container workloads
- Enabling Windows container workloads
- Installing a cluster on AWS with network customizations
- Installing a cluster on Azure with network customizations