OpenShift Container Platform 4.8

Networking

Configuring and managing cluster networking

Last Updated: 2021-09-03

OpenShift Container Platform 4.8 Networking

Configuring and managing cluster networking

Legal Notice

Abstract

This document provides instructions for configuring and managing your OpenShift Container Platform cluster network, including DNS, ingress, and the Pod network.

Table of Contents

CHAPTER 1. UNDERSTANDING NETWORKING

Kubernetes ensures that pods are able to network with each other, and allocates each pod an IP address from an internal network. This ensures all containers within the pod behave as if they were on the same host. Giving each pod its own IP address means that pods can be treated like physical hosts or virtual machines in terms of port allocation, networking, naming, service discovery, load balancing, application configuration, and migration.

NOTE

Some cloud platforms offer metadata APIs that listen on the 169.254.169.254 IP address, a link-local IP address in the IPv4 **169.254.0.0/16** CIDR block.

This CIDR block is not reachable from the pod network. Pods that need access to these IP addresses must be given host network access by setting the **spec.hostNetwork** field in the pod spec to **true**.

If you allow a pod host network access, you grant the pod privileged access to the underlying network infrastructure.

1.1. OPENSHIFT CONTAINER PLATFORM DNS

If you are running multiple services, such as front-end and back-end services for use with multiple pods, environment variables are created for user names, service IPs, and more so the front-end pods can communicate with the back-end services. If the service is deleted and recreated, a new IP address can be assigned to the service, and requires the front-end pods to be recreated to pick up the updated values for the service IP environment variable. Additionally, the back-end service must be created before any of the front-end pods to ensure that the service IP is generated properly, and that it can be provided to the front-end pods as an environment variable.

For this reason, OpenShift Container Platform has a built-in DNS so that the services can be reached by the service DNS as well as the service IP/port.

CHAPTER 2. ACCESSING HOSTS

Learn how to create a bastion host to access OpenShift Container Platform instances and access the control plane nodes (also known as the master nodes) with secure shell (SSH) access.

2.1. ACCESSING HOSTS ON AMAZON WEB SERVICES IN AN INSTALLER-PROVISIONED INFRASTRUCTURE CLUSTER

The OpenShift Container Platform installer does not create any public IP addresses for any of the Amazon Elastic Compute Cloud (Amazon EC2) instances that it provisions for your OpenShift Container Platform cluster. To be able to SSH to your OpenShift Container Platform hosts, you must follow this procedure.

Procedure

1. Create a security group that allows SSH access into the virtual private cloud (VPC) created by the **openshift-install** command.

2. Create an Amazon EC2 instance on one of the public subnets the installer created.

3. Associate a public IP address with the Amazon EC2 instance that you created.
   Unlike with the OpenShift Container Platform installation, you should associate the Amazon EC2 instance you created with an SSH keypair. It does not matter what operating system you choose for this instance, as it will simply serve as an SSH bastion to bridge the internet into your OpenShift Container Platform cluster's VPC. The Amazon Machine Image (AMI) you use does matter. With Red Hat Enterprise Linux CoreOS (RHCOS), for example, you can provide keys via Ignition, like the installer does.

4. After you provisioned your Amazon EC2 instance and can SSH into it, you must add the SSH key that you associated with your OpenShift Container Platform installation. This key can be different from the key for the bastion instance, but does not have to be.

   > **NOTE**
   >
   > Direct SSH access is only recommended for disaster recovery. When the Kubernetes API is responsive, run privileged pods instead.

5. Run **oc get nodes**, inspect the output, and choose one of the nodes that is a master. The hostname looks similar to **ip-10-0-1-163.ec2.internal**.

6. From the bastion SSH host you manually deployed into Amazon EC2, SSH into that control plane host (also known as the master host). Ensure that you use the same SSH key you specified during the installation:

   ```
   $ ssh -i <ssh-key-path> core@<master-hostname>
   ```

CHAPTER 3. CLUSTER NETWORK OPERATOR IN OPENSHIFT CONTAINER PLATFORM

The Cluster Network Operator (CNO) deploys and manages the cluster network components on an OpenShift Container Platform cluster, including the Container Network Interface (CNI) default network provider plug-in selected for the cluster during installation.

3.1. CLUSTER NETWORK OPERATOR

The Cluster Network Operator implements the **network** API from the **operator.openshift.io** API group. The Operator deploys the OpenShift SDN default Container Network Interface (CNI) network provider plug-in, or the default network provider plug-in that you selected during cluster installation, by using a daemon set.

Procedure

The Cluster Network Operator is deployed during installation as a Kubernetes **Deployment**.

1. Run the following command to view the Deployment status:

```
$ oc get -n openshift-network-operator deployment/network-operator
```

Example output

```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
network-operator   1/1     1            1           56m
```

2. Run the following command to view the state of the Cluster Network Operator:

```
$ oc get clusteroperator/network
```

Example output

```
NAME      VERSION   AVAILABLE   PROGRESSING   DEGRADED   SINCE
network   4.5.4     True        False         False      50m
```

The following fields provide information about the status of the operator: **AVAILABLE**, **PROGRESSING**, and **DEGRADED**. The **AVAILABLE** field is **True** when the Cluster Network Operator reports an available status condition.

3.2. VIEWING THE CLUSTER NETWORK CONFIGURATION

Every new OpenShift Container Platform installation has a **network.config** object named **cluster**.

Procedure

- Use the **oc describe** command to view the cluster network configuration:

```
$ oc describe network.config/cluster
```

Example output

```
Name:         cluster
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  config.openshift.io/v1
Kind:         Network
Metadata:
  Self Link:           /apis/config.openshift.io/v1/networks/cluster
Spec: 1
  Cluster Network:
    Cidr:         10.128.0.0/14
    Host Prefix:  23
  Network Type:   OpenShiftSDN
  Service Network:
    172.30.0.0/16
Status: 2
  Cluster Network:
    Cidr:               10.128.0.0/14
    Host Prefix:        23
  Cluster Network MTU:  8951
  Network Type:         OpenShiftSDN
  Service Network:
    172.30.0.0/16
Events:  <none>
```

1. The **Spec** field displays the configured state of the cluster network.
2. The **Status** field displays the current state of the cluster network configuration.

3.3. VIEWING CLUSTER NETWORK OPERATOR STATUS

You can inspect the status and view the details of the Cluster Network Operator using the **oc describe** command.

Procedure

- Run the following command to view the status of the Cluster Network Operator:

```
$ oc describe clusteroperators/network
```

3.4. VIEWING CLUSTER NETWORK OPERATOR LOGS

You can view Cluster Network Operator logs by using the **oc logs** command.

Procedure

- Run the following command to view the logs of the Cluster Network Operator:

```
$ oc logs --namespace=openshift-network-operator deployment/network-operator
```

3.5. CLUSTER NETWORK OPERATOR CONFIGURATION

The configuration for the cluster network is specified as part of the Cluster Network Operator (CNO) configuration and stored in a custom resource (CR) object that is named **cluster**. The CR specifies the fields for the **Network** API in the **operator.openshift.io** API group.

The CNO configuration inherits the following fields during cluster installation from the **Network** API in the **Network.config.openshift.io** API group and these fields cannot be changed:

clusterNetwork
: IP address pools from which pod IP addresses are allocated.

serviceNetwork
: IP address pool for services.

defaultNetwork.type
: Cluster network provider, such as OpenShift SDN or OVN-Kubernetes.

> **NOTE**
>
> After cluster installation, you cannot modify the fields listed in the previous section.

You can specify the cluster network provider configuration for your cluster by setting the fields for the **defaultNetwork** object in the CNO object named **cluster**.

3.5.1. Cluster Network Operator configuration object

The fields for the Cluster Network Operator (CNO) are described in the following table:

Table 3.1. Cluster Network Operator configuration object

Field	Type	Description
metadata.name	**string**	The name of the CNO object. This name is always **cluster**.

spec.clusterNetwork	**array**	A list specifying the blocks of IP addresses from which pod IP addresses are allocated and the subnet prefix length assigned to each individual node in the cluster. For example:
		`spec:`
		`  clusterNetwork:`
		`  - cidr: 10.128.0.0/19`
		`    hostPrefix: 23`
		`  - cidr: 10.128.32.0/19`
		`    hostPrefix: 23`
		This value is ready-only and inherited from the **Network.config.openshift.io** object named **cluster** during cluster installation.

Field	Type	Description

spec.serviceNetwork	**array**	A block of IP addresses for services. The OpenShift SDN and OVN-Kubernetes Container Network Interface (CNI) network providers support only a single IP address block for the service network. For example:
		`spec:`
		`  serviceNetwork:`
		`  - 172.30.0.0/14`
		This value is ready-only and inherited from the **Network.config.openshift.io** object named **cluster** during cluster installation.

spec.defaultNetwork	**object**	Configures the Container Network Interface (CNI) cluster network provider for the cluster network.
spec.kubeProxyConfig	**object**	The fields for this object specify the kube-proxy configuration. If you are using the OVN-Kubernetes cluster network provider, the kube-proxy configuration has no effect.

defaultNetwork object configuration

The values for the **defaultNetwork** object are defined in the following table:

Table 3.2. **defaultNetwork** object

Field	Type	Description
type	**string**	Either **OpenShiftSDN** or **OVNKubernetes**. The cluster network provider is selected during installation. This value cannot be changed after cluster installation. NOTE OpenShift Container Platform uses the OpenShift SDN Container Network Interface (CNI) cluster network provider by default.
openshiftSDNConfig	**object**	This object is only valid for the OpenShift SDN cluster network provider.
ovnKubernetesConfig	**object**	This object is only valid for the OVN-Kubernetes cluster network provider.

Configuration for the OpenShift SDN CNI cluster network provider

The following table describes the configuration fields for the OpenShift SDN Container Network Interface (CNI) cluster network provider.

Table 3.3. **openshiftSDNConfig** object

Field	Type	Description
mode	**string**	The network isolation mode for OpenShift SDN.
mtu	**integer**	The maximum transmission unit (MTU) for the VXLAN overlay network. This value is normally configured automatically.
vxlanPort	**integer**	The port to use for all VXLAN packets. The default value is **4789**.

NOTE

You can only change the configuration for your cluster network provider during cluster installation.

Example OpenShift SDN configuration

```
defaultNetwork:
  type: OpenShiftSDN
  openshiftSDNConfig:
    mode: NetworkPolicy
    mtu: 1450
    vxlanPort: 4789
```

Configuration for the OVN-Kubernetes CNI cluster network provider

The following table describes the configuration fields for the OVN-Kubernetes CNI cluster network provider.

Table 3.4. **ovnKubernetesConfig** object

Field	Type	Description
mtu	**integer**	The maximum transmission unit (MTU) for the Geneve (Generic Network Virtualization Encapsulation) overlay network. This value is normally configured automatically.
genevePort	**integer**	The UDP port for the Geneve overlay network.
ipsecConfig	**object**	If the field is present, IPsec is enabled for the cluster.
policyAuditConfig	**object**	Specify a configuration object for customizing network policy audit logging. If unset, the defaults audit log settings are used.

Table 3.5. **policyAuditConfig** object

Field	Type	Description
rateLimit	integer	The maximum number of messages to generate every second per node. The default value is **20** messages per second.
maxFileSize	integer	The maximum size for the audit log in bytes. The default value is **50000000** or 50MB.
destination	string	One of the following additional audit log targets: **libc** The libc **syslog()** function of the journald process on the host. **udp:<host>:<port>** A syslog server. Replace **<host>:<port>** with the host and port of the syslog server. **unix:<file>** A Unix Domain Socket file specified by **<file>**. **null** Do not send the audit logs to any additional target.
syslogFacility	string	The syslog facility, such as **kern**, as defined by RFC5424. The default value is **local0**.

NOTE

You can only change the configuration for your cluster network provider during cluster installation.

Example OVN-Kubernetes configuration

```
defaultNetwork:
  type: OVNKubernetes
  ovnKubernetesConfig:
    mtu: 1400
    genevePort: 6081
    ipsecConfig: {}
```

kubeProxyConfig object configuration

The values for the **kubeProxyConfig** object are defined in the following table:

Table 3.6. kubeProxyConfig object

Field	Type	Description

Field	Type	Description
iptablesSyncPeriod	**string**	The refresh period for **iptables** rules. The default value is **30s**. Valid suffixes include **s**, **m**, and **h** and are described in the Go **time** package documentation. NOTE Because of performance improvements introduced in OpenShift Container Platform 4.3 and greater, adjusting the **iptablesSyncPeriod** parameter is no longer necessary.

proxyArguments.iptables-min-sync-period	**array**	The minimum duration before refreshing **iptables** rules. This field ensures that the refresh does not happen too frequently. Valid suffixes include **s**, **m**, and **h** and are described in the Go **time** package. The default value is:
		
		`kubeProxyConfig:`
		`  proxyArguments:`
		`    iptables-min-sync-period:`
		`    - 0s`

3.5.2. Cluster Network Operator example configuration

A complete CNO configuration is specified in the following example:

Example Cluster Network Operator object

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  clusterNetwork: 1
  - cidr: 10.128.0.0/14
    hostPrefix: 23
  serviceNetwork: 2
  - 172.30.0.0/16
  defaultNetwork: 3
    type: OpenShiftSDN
    openshiftSDNConfig:
      mode: NetworkPolicy
      mtu: 1450
      vxlanPort: 4789
  kubeProxyConfig:
    iptablesSyncPeriod: 30s
```

```
proxyArguments:
  iptables-min-sync-period:
  - 0s
```

1 2 3 Configured only during cluster installation.

3.6. ADDITIONAL RESOURCES

- **Network** API in the **operator.openshift.io** API group

CHAPTER 4. DNS OPERATOR IN OPENSHIFT CONTAINER PLATFORM

The DNS Operator deploys and manages CoreDNS to provide a name resolution service to pods, enabling DNS-based Kubernetes Service discovery in OpenShift Container Platform.

4.1. DNS OPERATOR

The DNS Operator implements the **dns** API from the **operator.openshift.io** API group. The Operator deploys CoreDNS using a daemon set, creates a service for the daemon set, and configures the kubelet to instruct pods to use the CoreDNS service IP address for name resolution.

Procedure

The DNS Operator is deployed during installation with a **Deployment** object.

1. Use the **oc get** command to view the deployment status:

```
$ oc get -n openshift-dns-operator deployment/dns-operator
```

Example output

```
NAME           READY     UP-TO-DATE   AVAILABLE   AGE
dns-operator   1/1       1            1           23h
```

2. Use the **oc get** command to view the state of the DNS Operator:

```
$ oc get clusteroperator/dns
```

Example output

```
NAME      VERSION     AVAILABLE   PROGRESSING   DEGRADED   SINCE
dns       4.1.0-0.11  True        False         False      92m
```

AVAILABLE, **PROGRESSING** and **DEGRADED** provide information about the status of the operator. **AVAILABLE** is **True** when at least 1 pod from the CoreDNS daemon set reports an **Available** status condition.

4.2. CONTROLLING DNS POD PLACEMENT

The DNS Operator has two daemon sets: one for CoreDNS and one for managing the **/etc/hosts** file. The daemon set for **/etc/hosts** must run on every node host to add an entry for the cluster image registry to support pulling images. Security policies can prohibit communication between pairs of nodes, which prevents the daemon set for CoreDNS from running on every node.

As a cluster administrator, you can use a custom node selector to configure the daemon set for CoreDNS to run or not run on certain nodes.

Prerequisites

- You installed the **oc** CLI.

- You are logged in to the cluster with a user with **cluster-admin** privileges.

Procedure

- To prevent communication between certain nodes, configure the **spec.nodePlacement.nodeSelector** API field:
  1. Modify the DNS Operator object named **default**:

     ```
     $ oc edit dns.operator/default
     ```

  2. Specify a node selector that includes only control plane nodes in the **spec.nodePlacement.nodeSelector** API field:

     ```
     spec:
       nodePlacement:
         nodeSelector:
           node-role.kubernetes.io/worker: ""
     ```

- To allow the daemon set for CoreDNS to run on nodes, configure a taint and toleration:
  1. Modify the DNS Operator object named **default**:

     ```
     $ oc edit dns.operator/default
     ```

  2. Specify a taint key and a toleration for the taint:

     ```
     spec:
       nodePlacement:
         tolerations:
         - effect: NoExecute
           key: "dns-only"
           operators: Equal
           value: abc
           tolerationSeconds: 3600 1
     ```

     1 If the taint is **dns-only**, it can be tolerated indefinitely. You can omit **tolerationSeconds**.

4.3. VIEW THE DEFAULT DNS

Every new OpenShift Container Platform installation has a **dns.operator** named **default**.

Procedure

1. Use the **oc describe** command to view the default **dns**:

   ```
   $ oc describe dns.operator/default
   ```

   #### Example output

   ```
   Name:         default
   ```

```
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  operator.openshift.io/v1
Kind:         DNS
...
Status:
  Cluster Domain:  cluster.local 1
  Cluster IP:      172.30.0.10 2
...
```

1 The Cluster Domain field is the base DNS domain used to construct fully qualified pod and service domain names.

2 The Cluster IP is the address pods query for name resolution. The IP is defined as the 10th address in the service CIDR range.

2. To find the service CIDR of your cluster, use the **oc get** command:

```
$ oc get networks.config/cluster -o jsonpath='{$.status.serviceNetwork}'
```

Example output

```
[172.30.0.0/16]
```

4.4. USING DNS FORWARDING

You can use DNS forwarding to override the forwarding configuration identified in **/etc/resolv.conf** on a per-zone basis by specifying which name server should be used for a given zone. If the forwarded zone is the Ingress domain managed by OpenShift Container Platform, then the upstream name server must be authorized for the domain.

Procedure

1. Modify the DNS Operator object named **default**:

```
$ oc edit dns.operator/default
```

This allows the Operator to create and update the ConfigMap named **dns-default** with additional server configuration blocks based on **Server**. If none of the servers has a zone that matches the query, then name resolution falls back to the name servers that are specified in **/etc/resolv.conf**.

Sample DNS

```
apiVersion: operator.openshift.io/v1
kind: DNS
metadata:
  name: default
spec:
  servers:
  - name: foo-server 1
```

```
    zones: 2
    - foo.com
    forwardPlugin:
      upstreams: 3
      - 1.1.1.1
      - 2.2.2.2:5353
  - name: bar-server
    zones:
    - bar.com
    - example.com
    forwardPlugin:
      upstreams:
      - 3.3.3.3
      - 4.4.4.4:5454
```

1. **name** must comply with the **rfc6335** service name syntax.
2. **zones** must conform to the definition of a **subdomain** in **rfc1123**. The cluster domain, **cluster.local**, is an invalid **subdomain** for **zones**.
3. A maximum of 15 **upstreams** is allowed per **forwardPlugin**.

NOTE

If **servers** is undefined or invalid, the ConfigMap only contains the default server.

2. View the ConfigMap:

```
$ oc get configmap/dns-default -n openshift-dns -o yaml
```

Sample DNS ConfigMap based on previous sample DNS

```
apiVersion: v1
data:
  Corefile: |
    foo.com:5353 {
        forward . 1.1.1.1 2.2.2.2:5353
    }
    bar.com:5353 example.com:5353 {
        forward . 3.3.3.3 4.4.4.4:5454 1
    }
    .:5353 {
        errors
        health
        kubernetes cluster.local in-addr.arpa ip6.arpa {
            pods insecure
            upstream
            fallthrough in-addr.arpa ip6.arpa
        }
        prometheus :9153
        forward . /etc/resolv.conf {
            policy sequential
        }
```

```
        cache 30
        reload
    }
kind: ConfigMap
metadata:
  labels:
    dns.operator.openshift.io/owning-dns: default
  name: dns-default
  namespace: openshift-dns
```

1 Changes to the **forwardPlugin** triggers a rolling update of the CoreDNS daemon set.

Additional resources

- For more information on DNS forwarding, see the CoreDNS forward documentation.

4.5. DNS OPERATOR STATUS

You can inspect the status and view the details of the DNS Operator using the **oc describe** command.

Procedure

View the status of the DNS Operator:

```
$ oc describe clusteroperators/dns
```

4.6. DNS OPERATOR LOGS

You can view DNS Operator logs by using the **oc logs** command.

Procedure

View the logs of the DNS Operator:

```
$ oc logs -n openshift-dns-operator deployment/dns-operator -c dns-operator
```

CHAPTER 5. INGRESS OPERATOR IN OPENSHIFT CONTAINER PLATFORM

The Ingress Operator implements the **ingresscontroller** API and is the component responsible for enabling external access to OpenShift Container Platform cluster services. The Operator makes this possible by deploying and managing one or more HAProxy-based Ingress Controllers to handle routing. You can use the Ingress Operator to route traffic by specifying OpenShift Container Platform **Route** and Kubernetes **Ingress** resources.

5.1. THE INGRESS CONFIGURATION ASSET

The installation program generates an asset with an **Ingress** resource in the **config.openshift.io** API group, **cluster-ingress-02-config.yml**.

YAML Definition of the Ingress resource

```
apiVersion: config.openshift.io/v1
kind: Ingress
metadata:
  name: cluster
spec:
  domain: apps.openshiftdemos.com
```

The installation program stores this asset in the **cluster-ingress-02-config.yml** file in the **manifests/** directory. This **Ingress** resource defines the cluster-wide configuration for Ingress. This Ingress configuration is used as follows:

- The Ingress Operator uses the domain from the cluster Ingress configuration as the domain for the default Ingress Controller.
- The OpenShift API Server Operator uses the domain from the cluster Ingress configuration. This domain is also used when generating a default host for a **Route** resource that does not specify an explicit host.

5.2. INGRESS CONTROLLER CONFIGURATION PARAMETERS

The **ingresscontrollers.operator.openshift.io** resource offers the following configuration parameters.

Parameter	Description

Parameter	Description
domain	**domain** is a DNS name serviced by the Ingress Controller and is used to configure multiple features: • For the **LoadBalancerService** endpoint publishing strategy, **domain** is used to configure DNS records. See **endpointPublishingStrategy**. • When using a generated default certificate, the certificate is valid for **domain** and its **subdomains**. See **defaultCertificate**. • The value is published to individual Route statuses so that users know where to target external DNS records. The **domain** value must be unique among all Ingress Controllers and cannot be updated. If empty, the default value is **ingress.config.openshift.io/cluster .spec.domain**.
appsDomain	**appsDomain** is an optional domain for AWS infrastructure to use instead of the one specified in the **domain** field when a Route is created without specifying an explicit host. If a value is entered for **appsDomain**, this value is used to generate default host values for the Route. Unlike **domain**, **appsDomain** can be modified after installation. You can use this parameter only if you set up a new Ingress Controller that uses a wildcard certificate.
replicas	**replicas** is the desired number of Ingress Controller replicas. If not set, the default value is **2**.
endpointPublishingStrategy	**endpointPublishingStrategy** is used to publish the Ingress Controller endpoints to other networks, enable load balancer integrations, and provide access to other systems. If not set, the default value is based on **infrastructure.config.openshift.io/cluster .status.platform**: • AWS: **LoadBalancerService** (with external scope) • Azure: **LoadBalancerService** (with external scope) • GCP: **LoadBalancerService** (with external scope) • Bare metal: **NodePortService** • Other: **HostNetwork** For most platforms, the **endpointPublishingStrategy** value cannot be updated. However, on GCP, you can configure the **loadbalancer.providerParameters.gcp.clientAccess** subfield.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>defaultCertificate</td><td>The defaultCertificate value is a reference to a secret that contains the default certificate that is served by the Ingress Controller. When Routes do not specify their own certificate, defaultCertificate is used.

The secret must contain the following keys and data: * tls.crt: certificate file contents * tls.key: key file contents

If not set, a wildcard certificate is automatically generated and used. The certificate is valid for the Ingress Controller domain and subdomains, and the generated certificate's CA is automatically integrated with the cluster's trust store.

The in-use certificate, whether generated or user-specified, is automatically integrated with OpenShift Container Platform built-in OAuth server.</td></tr>
<tr><td>namespaceSelector</td><td>namespaceSelector is used to filter the set of namespaces serviced by the Ingress Controller. This is useful for implementing shards.</td></tr>
<tr><td>routeSelector</td><td>routeSelector is used to filter the set of Routes serviced by the Ingress Controller. This is useful for implementing shards.</td></tr>
<tr><td>nodePlacement</td><td>nodePlacement enables explicit control over the scheduling of the Ingress Controller.

If not set, the defaults values are used.

NOTE

The nodePlacement parameter includes two parts, nodeSelector and tolerations. For example:

nodePlacement:
 nodeSelector:
   matchLabels:
     beta.kubernetes.io/os: linux
 tolerations:
 - effect: NoSchedule
   operator: Exists</td></tr>
</table>

Parameter	Description
tlsSecurityProfile	**tlsSecurityProfile** specifies settings for TLS connections for Ingress Controllers. If not set, the default value is based on the **apiservers.config.openshift.io/cluster** resource. When using the **Old**, **Intermediate**, and **Modern** profile types, the effective profile configuration is subject to change between releases. For example, given a specification to use the **Intermediate** profile deployed on release **X.Y.Z**, an upgrade to release **X.Y.Z+1** may cause a new profile configuration to be applied to the Ingress Controller, resulting in a rollout. The minimum TLS version for Ingress Controllers is **1.1**, and the maximum TLS version is **1.2**. IMPORTANT The HAProxy Ingress Controller image does not support TLS **1.3** and because the **Modern** profile requires TLS **1.3**, it is not supported. The Ingress Operator converts the **Modern** profile to **Intermediate**. The Ingress Operator also converts the TLS **1.0** of an **Old** or **Custom** profile to **1.1**, and TLS **1.3** of a **Custom** profile to **1.2**. NOTE Ciphers and the minimum TLS version of the configured security profile are reflected in the **TLSProfile** status.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>routeAdmission</td><td> routeAdmission defines a policy for handling new route claims, such as allowing or denying claims across namespaces.
namespaceOwnership describes how hostname claims across namespaces should be handled. The default is Strict.

Strict: does not allow routes to claim the same hostname across namespaces.
InterNamespaceAllowed: allows routes to claim different paths of the same hostname across namespaces.

wildcardPolicy describes how routes with wildcard policies are handled by the Ingress Controller.

WildcardsAllowed: Indicates routes with any wildcard policy are admitted by the Ingress Controller.
WildcardsDisallowed: Indicates only routes with a wildcard policy of None are admitted by the Ingress Controller. Updating wildcardPolicy from WildcardsAllowed to WildcardsDisallowed causes admitted routes with a wildcard policy of Subdomain to stop working. These routes must be recreated to a wildcard policy of None to be readmitted by the Ingress Controller. WildcardsDisallowed is the default setting.
</td></tr>
</table>

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>IngressControllerLogging</td><td>
logging defines parameters for what is logged where. If this field is empty, operational logs are enabled but access logs are disabled.

access describes how client requests are logged. If this field is empty, access logging is disabled.

destination describes a destination for log messages.

type is the type of destination for logs:

Container specifies that logs should go to a sidecar container. The Ingress Operator configures the container, named logs, on the Ingress Controller pod and configures the Ingress Controller to write logs to the container. The expectation is that the administrator configures a custom logging solution that reads logs from this container. Using container logs means that logs may be dropped if the rate of logs exceeds the container runtime capacity or the custom logging solution capacity.
Syslog specifies that logs are sent to a Syslog endpoint. The administrator must specify an endpoint that can receive Syslog messages. The expectation is that the administrator has configured a custom Syslog instance.

container describes parameters for the Container logging destination type. Currently there are no parameters for container logging, so this field must be empty.
syslog describes parameters for the Syslog logging destination type:

address is the IP address of the syslog endpoint that receives log messages.
port is the UDP port number of the syslog endpoint that receives log messages.
facility specifies the syslog facility of log messages. If this field is empty, the facility is local1. Otherwise, it must specify a valid syslog facility: kern, user, mail, daemon, auth, syslog, lpr, news, uucp, cron, auth2, ftp, ntp, audit, alert, cron2, local0, local1, local2, local3. local4, local5, local6, or local7.

httpLogFormat specifies the format of the log message for an HTTP request. If this field is empty, log messages use the implementation’s default HTTP log format. For HAProxy’s default HTTP log format, see the HAProxy documentation.

</td></tr>
</table>

Parameter	Description
httpHeaders	**httpHeaders** defines the policy for HTTP headers. By setting the **forwardedHeaderPolicy** for the **IngressControllerHTTPHeaders**, you specify when and how the Ingress controller sets the **Forwarded**, **X-Forwarded-For**, **X-Forwarded-Host**, **X-Forwarded-Port**, **X-Forwarded-Proto**, and **X-Forwarded-Proto-Version** HTTP headers. By default, the policy is set to **Append**. • **Append** specifies that the Ingress Controller appends the headers, preserving any existing headers. • **Replace** specifies that the Ingress Controller sets the headers, removing any existing headers. • **IfNone** specifies that the Ingress Controller sets the headers if they are not already set. • **Never** specifies that the Ingress Controller never sets the headers, preserving any existing headers. By setting **headerNameCaseAdjustments**, you can specify case adjustments that can be applied to HTTP header names. Each adjustment is specified as an HTTP header name with the desired capitalization. For example, specifying **X-Forwarded-For** indicates that the **x-forwarded-for** HTTP header should be adjusted to have the specified capitalization. These adjustments are only applied to cleartext, edge-terminated, and re-encrypt routes, and only when using HTTP/1. For request headers, these adjustments are applied only for routes that have the **haproxy.router.openshift.io/h1-adjust-case=true** annotation. For response headers, these adjustments are applied to all HTTP responses. If this field is empty, no request headers are adjusted.

Parameter	Description
tuningOptions	**tuningOptions** specifies options for tuning the performance of Ingress Controller pods. • **headerBufferBytes** specifies how much memory is reserved, in bytes, for Ingress Controller connection sessions. This value must be at least **16384** if HTTP/2 is enabled for the Ingress Controller. If not set, the default value is **32768** bytes. Setting this field not recommended because **headerBufferBytes** values that are too small can break the Ingress Controller, and **headerBufferBytes** values that are too large could cause the Ingress Controller to use significantly more memory than necessary. • **headerBufferMaxRewriteBytes** specifies how much memory should be reserved, in bytes, from **headerBufferBytes** for HTTP header rewriting and appending for Ingress Controller connection sessions. The minimum value for **headerBufferMaxRewriteBytes** is **4096**. **headerBufferBytes** must be greater than **headerBufferMaxRewriteBytes** for incoming HTTP requests. If not set, the default value is **8192** bytes. Setting this field not recommended because **headerBufferMaxRewriteBytes** values that are too small can break the Ingress Controller and **headerBufferMaxRewriteBytes** values that are too large could cause the Ingress Controller to use significantly more memory than necessary. • **threadCount** specifies the number of threads to create per HAProxy process. Creating more threads allows each Ingress Controller pod to handle more connections, at the cost of more system resources being used. HAProxy supports up to **64** threads. If this field is empty, the Ingress Controller uses the default value of **4** threads. The default value can change in future releases. Setting this field is not recommended because increasing the number of HAProxy threads allows Ingress Controller pods to use more CPU time under load, and prevent other pods from receiving the CPU resources they need to perform. Reducing the number of threads can cause the Ingress Controller to perform poorly.

NOTE

All parameters are optional.

5.2.1. Ingress Controller TLS security profiles

TLS security profiles provide a way for servers to regulate which ciphers a connecting client can use when connecting to the server.

5.2.1.1. Understanding TLS security profiles

You can use a TLS (Transport Layer Security) security profile to define which TLS ciphers are required by various OpenShift Container Platform components. The OpenShift Container Platform TLS security profiles are based on Mozilla recommended configurations.

You can specify one of the following TLS security profiles for each component:

Table 5.1. TLS security profiles

Profile	Description
Old	This profile is intended for use with legacy clients or libraries. The profile is based on the Old backward compatibility recommended configuration. The **Old** profile requires a minimum TLS version of 1.0. NOTE For the Ingress Controller, the minimum TLS version is converted from 1.0 to 1.1.
Intermediate	This profile is the recommended configuration for the majority of clients. It is the default TLS security profile for the Ingress Controller, kubelet, and control plane. The profile is based on the Intermediate compatibility recommended configuration. The **Intermediate** profile requires a minimum TLS version of 1.2.
Modern	This profile is intended for use with modern clients that have no need for backwards compatibility. This profile is based on the Modern compatibility recommended configuration. The **Modern** profile requires a minimum TLS version of 1.3. IMPORTANT The **Modern** profile is currently not supported.
Custom	This profile allows you to define the TLS version and ciphers to use. WARNING Use caution when using a **Custom** profile, because invalid configurations can cause problems.

NOTE

When using one of the predefined profile types, the effective profile configuration is subject to change between releases. For example, given a specification to use the Intermediate profile deployed on release X.Y.Z, an upgrade to release X.Y.Z+1 might cause a new profile configuration to be applied, resulting in a rollout.

5.2.1.2. Configuring the TLS security profile for the Ingress Controller

To configure a TLS security profile for an Ingress Controller, edit the **IngressController** custom resource (CR) to specify a predefined or custom TLS security profile. If a TLS security profile is not configured, the default value is based on the TLS security profile set for the API server.

Sample IngressController CR that configures the Old TLS security profile

```
apiVersion: config.openshift.io/v1
kind: IngressController
 ...
spec:
  tlsSecurityProfile:
    old: {}
    type: Old
 ...
```

The TLS security profile defines the minimum TLS version and the TLS ciphers for TLS connections for Ingress Controllers.

You can see the ciphers and the minimum TLS version of the configured TLS security profile in the **IngressController** custom resource (CR) under **Status.Tls Profile** and the configured TLS security profile under **Spec.Tls Security Profile**. For the **Custom** TLS security profile, the specific ciphers and minimum TLS version are listed under both parameters.

> **IMPORTANT**
>
> The HAProxy Ingress Controller image does not support TLS **1.3** and because the **Modern** profile requires TLS **1.3**, it is not supported. The Ingress Operator converts the **Modern** profile to **Intermediate**.
>
> The Ingress Operator also converts the TLS **1.0** of an **Old** or **Custom** profile to **1.1**, and TLS **1.3** of a **Custom** profile to **1.2**.

Prerequisites

- You have access to the cluster as a user with the **cluster-admin** role.

Procedure

1. Edit the **IngressController** CR in the **openshift-ingress-operator** project to configure the TLS security profile:

   ```
   $ oc edit IngressController default -n openshift-ingress-operator
   ```

2. Add the **spec.tlsSecurityProfile** field:

   Sample IngressController CR for a Custom profile

   ```
   apiVersion: operator.openshift.io/v1
   kind: IngressController
    ...
   spec:
     tlsSecurityProfile:
       type: Custom 1
       custom: 2
   ```

```
      ciphers: 3
      - ECDHE-ECDSA-CHACHA20-POLY1305
      - ECDHE-RSA-CHACHA20-POLY1305
      - ECDHE-RSA-AES128-GCM-SHA256
      - ECDHE-ECDSA-AES128-GCM-SHA256
      minTLSVersion: VersionTLS11
...
```

1. Specify the TLS security profile type (**Old**, **Intermediate**, or **Custom**). The default is **Intermediate**.

2. Specify the appropriate field for the selected type:
   - **old: {}**
   - **intermediate: {}**
   - **custom:**

3. For the **custom** type, specify a list of TLS ciphers and minimum accepted TLS version.

3. Save the file to apply the changes.

Verification

- Verify that the profile is set in the **IngressController** CR:

```
$ oc describe IngressController default -n openshift-ingress-operator
```

Example output

```
Name:         default
Namespace:    openshift-ingress-operator
Labels:       <none>
Annotations:  <none>
API Version:  operator.openshift.io/v1
Kind:         IngressController
 ...
Spec:
 ...
  Tls Security Profile:
    Custom:
      Ciphers:
        ECDHE-ECDSA-CHACHA20-POLY1305
        ECDHE-RSA-CHACHA20-POLY1305
        ECDHE-RSA-AES128-GCM-SHA256
        ECDHE-ECDSA-AES128-GCM-SHA256
      Min TLS Version:  VersionTLS11
    Type:               Custom
 ...
```

5.2.2. Ingress controller endpoint publishing strategy

NodePortService endpoint publishing strategy

The **NodePortService** endpoint publishing strategy publishes the Ingress Controller using a Kubernetes NodePort service.

In this configuration, the Ingress Controller deployment uses container networking. A **NodePortService** is created to publish the deployment. The specific node ports are dynamically allocated by OpenShift Container Platform; however, to support static port allocations, your changes to the node port field of the managed **NodePortService** are preserved.

NOTE

The Ingress Operator ignores any updates to **.spec.ports[].nodePort** fields of the service.

By default, ports are allocated automatically and you can access the port allocations for integrations. However, sometimes static port allocations are necessary to integrate with existing infrastructure which may not be easily reconfigured in response to dynamic ports. To achieve integrations with static node ports, you can update the managed service resource directly.

For more information, see the Kubernetes Services documentation on **NodePort**.

HostNetwork endpoint publishing strategy

The **HostNetwork** endpoint publishing strategy publishes the Ingress Controller on node ports where the Ingress Controller is deployed.

An Ingress controller with the **HostNetwork** endpoint publishing strategy can have only one pod replica per node. If you want *n* replicas, you must use at least *n* nodes where those replicas can be scheduled. Because each pod replica requests ports **80** and **443** on the node host where it is scheduled, a replica cannot be scheduled to a node if another pod on the same node is using those ports.

5.3. VIEW THE DEFAULT INGRESS CONTROLLER

The Ingress Operator is a core feature of OpenShift Container Platform and is enabled out of the box.

Every new OpenShift Container Platform installation has an **ingresscontroller** named default. It can be supplemented with additional Ingress Controllers. If the default **ingresscontroller** is deleted, the Ingress Operator will automatically recreate it within a minute.

Procedure

- View the default Ingress Controller:

```
$ oc describe --namespace=openshift-ingress-operator ingresscontroller/default
```

5.4. VIEW INGRESS OPERATOR STATUS

You can view and inspect the status of your Ingress Operator.

Procedure

- View your Ingress Operator status:

```
$ oc describe clusteroperators/ingress
```

5.5. VIEW INGRESS CONTROLLER LOGS

You can view your Ingress Controller logs.

Procedure

- View your Ingress Controller logs:

```
$ oc logs --namespace=openshift-ingress-operator deployments/ingress-operator
```

5.6. VIEW INGRESS CONTROLLER STATUS

Your can view the status of a particular Ingress Controller.

Procedure

- View the status of an Ingress Controller:

```
$ oc describe --namespace=openshift-ingress-operator ingresscontroller/<name>
```

5.7. CONFIGURING THE INGRESS CONTROLLER

5.7.1. Setting a custom default certificate

As an administrator, you can configure an Ingress Controller to use a custom certificate by creating a Secret resource and editing the **IngressController** custom resource (CR).

Prerequisites

- You must have a certificate/key pair in PEM-encoded files, where the certificate is signed by a trusted certificate authority or by a private trusted certificate authority that you configured in a custom PKI.
- Your certificate meets the following requirements:
  - The certificate is valid for the ingress domain.
  - The certificate uses the **subjectAltName** extension to specify a wildcard domain, such as ***.apps.ocp4.example.com**.
- You must have an **IngressController** CR. You may use the default one:

```
$ oc --namespace openshift-ingress-operator get ingresscontrollers
```

Example output

```
NAME      AGE
default   10m
```

> **NOTE**
>
> If you have intermediate certificates, they must be included in the **tls.crt** file of the secret containing a custom default certificate. Order matters when specifying a certificate; list your intermediate certificate(s) after any server certificate(s).

Procedure

The following assumes that the custom certificate and key pair are in the **tls.crt** and **tls.key** files in the current working directory. Substitute the actual path names for **tls.crt** and **tls.key**. You also may substitute another name for **custom-certs-default** when creating the Secret resource and referencing it in the IngressController CR.

> **NOTE**
>
> This action will cause the Ingress Controller to be redeployed, using a rolling deployment strategy.

1. Create a Secret resource containing the custom certificate in the **openshift-ingress** namespace using the **tls.crt** and **tls.key** files.

```
$ oc --namespace openshift-ingress create secret tls custom-certs-default --cert=tls.crt --
key=tls.key
```

2. Update the IngressController CR to reference the new certificate secret:

```
$ oc patch --type=merge --namespace openshift-ingress-operator ingresscontrollers/default \
  --patch '{"spec":{"defaultCertificate":{"name":"custom-certs-default"}}}'
```

3. Verify the update was effective:

```
$ echo Q |\
  openssl s_client -connect console-openshift-console.apps.<domain>:443 -showcerts
2>/dev/null |\
  openssl x509 -noout -subject -issuer -enddate
```

where:

<domain>
: Specifies the base domain name for your cluster.

Example output

```
subject=C = US, ST = NC, L = Raleigh, O = RH, OU = OCP4, CN = *.apps.example.com
issuer=C = US, ST = NC, L = Raleigh, O = RH, OU = OCP4, CN = example.com
notAfter=May 10 08:32:45 2022 GM
```

TIP

You can alternatively apply the following YAML to set a custom default certificate:

```
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: default
  namespace: openshift-ingress-operator
spec:
  defaultCertificate:
    name: custom-certs-default
```

The certificate secret name should match the value used to update the CR.

Once the IngressController CR has been modified, the Ingress Operator updates the Ingress Controller's deployment to use the custom certificate.

5.7.2. Removing a custom default certificate

As an administrator, you can remove a custom certificate that you configured an Ingress Controller to use.

Prerequisites

- You have access to the cluster as a user with the **cluster-admin** role.
- You have installed the OpenShift CLI (**oc**).
- You previously configured a custom default certificate for the Ingress Controller.

Procedure

- To remove the custom certificate and restore the certificate that ships with OpenShift Container Platform, enter the following command:

```
$ oc patch -n openshift-ingress-operator ingresscontrollers/default \
  --type json -p $'- op: remove\n  path: /spec/defaultCertificate'
```

There can be a delay while the cluster reconciles the new certificate configuration.

Verification

- To confirm that the original cluster certificate is restored, enter the following command:

```
$ echo Q | \
  openssl s_client -connect console-openshift-console.apps.<domain>:443 -showcerts 2>/dev/null | \
  openssl x509 -noout -subject -issuer -enddate
```

where:

<domain>
Specifies the base domain name for your cluster.

Example output

```
subject=CN = *.apps.<domain>
issuer=CN = ingress-operator@1620633373
notAfter=May 10 10:44:36 2023 GMT
```

5.7.3. Scaling an Ingress Controller

Manually scale an Ingress Controller to meeting routing performance or availability requirements such as the requirement to increase throughput. **oc** commands are used to scale the **IngressController** resource. The following procedure provides an example for scaling up the default **IngressController**.

> **NOTE**
>
> Scaling is not an immediate action, as it takes time to create the desired number of replicas.

Procedure

1. View the current number of available replicas for the default **IngressController**:

   ```
   $ oc get -n openshift-ingress-operator ingresscontrollers/default -o
   jsonpath='{$.status.availableReplicas}'
   ```

   #### Example output

   ```
   2
   ```

2. Scale the default **IngressController** to the desired number of replicas using the **oc patch** command. The following example scales the default **IngressController** to 3 replicas:

   ```
   $ oc patch -n openshift-ingress-operator ingresscontroller/default --patch '{"spec":{"replicas":
   3}}' --type=merge
   ```

   #### Example output

   ```
   ingresscontroller.operator.openshift.io/default patched
   ```

3. Verify that the default **IngressController** scaled to the number of replicas that you specified:

   ```
   $ oc get -n openshift-ingress-operator ingresscontrollers/default -o
   jsonpath='{$.status.availableReplicas}'
   ```

   #### Example output

   ```
   3
   ```

TIP

You can alternatively apply the following YAML to scale an Ingress Controller to three replicas:

```
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: default
  namespace: openshift-ingress-operator
spec:
  replicas: 3 1
```

1 If you need a different amount of replicas, change the **replicas** value.

5.7.4. Configuring Ingress access logging

You can configure the Ingress Controller to enable access logs. If you have clusters that do not receive much traffic, then you can log to a sidecar. If you have high traffic clusters, to avoid exceeding the capacity of the logging stack or to integrate with a logging infrastructure outside of OpenShift Container Platform, you can forward logs to a custom syslog endpoint. You can also specify the format for access logs.

Container logging is useful to enable access logs on low-traffic clusters when there is no existing Syslog logging infrastructure, or for short-term use while diagnosing problems with the Ingress Controller.

Syslog is needed for high-traffic clusters where access logs could exceed the OpenShift Logging stack's capacity, or for environments where any logging solution needs to integrate with an existing Syslog logging infrastructure. The Syslog use-cases can overlap.

Prerequisites

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

Procedure

Configure Ingress access logging to a sidecar.

- To configure Ingress access logging, you must specify a destination using **spec.logging.access.destination**. To specify logging to a sidecar container, you must specify **Container spec.logging.access.destination.type**. The following example is an Ingress Controller definition that logs to a **Container** destination:

```
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: default
  namespace: openshift-ingress-operator
spec:
  replicas: 2
  endpointPublishingStrategy:
    type: NodePortService 1
  logging:
```

```
    access:
      destination:
        type: Container
```

1. **NodePortService** is not required to configure Ingress access logging to a sidecar. Ingress logging is compatible with any **endpointPublishingStrategy**.

- When you configure the Ingress Controller to log to a sidecar, the operator creates a container named **logs** inside the Ingress Controller Pod:

  ```
  $ oc -n openshift-ingress logs deployment.apps/router-default -c logs
  ```

  Example output

  ```
  2020-05-11T19:11:50.135710+00:00 router-default-57dfc6cd95-bpmk6 router-default-
  57dfc6cd95-bpmk6 haproxy[108]: 174.19.21.82:39654 [11/May/2020:19:11:50.133] public
  be_http:hello-openshift:hello-openshift/pod:hello-openshift:hello-openshift:10.128.2.12:8080
  0/0/1/0/1 200 142 - - --NI 1/1/0/0/0 0/0 "GET / HTTP/1.1"
  ```

Configure Ingress access logging to a Syslog endpoint.

- To configure Ingress access logging, you must specify a destination using **spec.logging.access.destination**. To specify logging to a Syslog endpoint destination, you must specify **Syslog** for **spec.logging.access.destination.type**. If the destination type is **Syslog**, you must also specify a destination endpoint using **spec.logging.access.destination.syslog.endpoint** and you can specify a facility using **spec.logging.access.destination.syslog.facility**. The following example is an Ingress Controller definition that logs to a **Syslog** destination:

  ```
  apiVersion: operator.openshift.io/v1
  kind: IngressController
  metadata:
    name: default
    namespace: openshift-ingress-operator
  spec:
    replicas: 2
    endpointPublishingStrategy:
      type: NodePortService
    logging:
      access:
        destination:
          type: Syslog
          syslog:
            address: 1.2.3.4
            port: 10514
  ```

  NOTE

  The **syslog** destination port must be UDP.

Configure Ingress access logging with a specific log format.

- You can specify **spec.logging.access.httpLogFormat** to customize the log format. The following example is an Ingress Controller definition that logs to a **syslog** endpoint with IP address 1.2.3.4 and port 10514:

```
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: default
  namespace: openshift-ingress-operator
spec:
  replicas: 2
  endpointPublishingStrategy:
    type: NodePortService
  logging:
    access:
      destination:
        type: Syslog
        syslog:
          address: 1.2.3.4
          port: 10514
      httpLogFormat: '%ci:%cp [%t] %ft %b/%s %B %bq %HM %HU %HV'
```

Disable Ingress access logging.

- To disable Ingress access logging, leave **spec.logging** or **spec.logging.access** empty:

```
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: default
  namespace: openshift-ingress-operator
spec:
  replicas: 2
  endpointPublishingStrategy:
    type: NodePortService
  logging:
    access: null
```

5.7.5. Setting Ingress Controller thread count

A cluster administrator can set the thread count to increase the amount of incoming connections a cluster can handle. You can patch an existing Ingress Controller to increase the amount of threads.

Prerequisites

- The following assumes that you already created an Ingress Controller.

Procedure

- Update the Ingress Controller to increase the number of threads:

```
$ oc -n openshift-ingress-operator patch ingresscontroller/default --type=merge -p '{"spec":
{"tuningOptions": {"threadCount": 8}}}'
```

NOTE

If you have a node that is capable of running large amounts of resources, you can configure **spec.nodePlacement.nodeSelector** with labels that match the capacity of the intended node, and configure **spec.tuningOptions.threadCount** to an appropriately high value.

5.7.6. Ingress Controller sharding

As the primary mechanism for traffic to enter the cluster, the demands on the Ingress Controller, or router, can be significant. As a cluster administrator, you can shard the routes to:

- Balance Ingress Controllers, or routers, with several routes to speed up responses to changes.
- Allocate certain routes to have different reliability guarantees than other routes.
- Allow certain Ingress Controllers to have different policies defined.
- Allow only specific routes to use additional features.
- Expose different routes on different addresses so that internal and external users can see different routes, for example.

Ingress Controller can use either route labels or namespace labels as a sharding method.

5.7.6.1. Configuring Ingress Controller sharding by using route labels

Ingress Controller sharding by using route labels means that the Ingress Controller serves any route in any namespace that is selected by the route selector.

Ingress Controller sharding is useful when balancing incoming traffic load among a set of Ingress Controllers and when isolating traffic to a specific Ingress Controller. For example, company A goes to one Ingress Controller and company B to another.

Procedure

1. Edit the **router-internal.yaml** file:

```
# cat router-internal.yaml
apiVersion: v1
items:
- apiVersion: operator.openshift.io/v1
  kind: IngressController
  metadata:
    name: sharded
    namespace: openshift-ingress-operator
  spec:
    domain: <apps-sharded.basedomain.example.net>
    nodePlacement:
      nodeSelector:
        matchLabels:
          node-role.kubernetes.io/worker: ""
    routeSelector:
      matchLabels:
        type: sharded
  status: {}
```

```
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

2. Apply the Ingress Controller **router-internal.yaml** file:

```
# oc apply -f router-internal.yaml
```

The Ingress Controller selects routes in any namespace that have the label **type: sharded**.

5.7.6.2. Configuring Ingress Controller sharding by using namespace labels

Ingress Controller sharding by using namespace labels means that the Ingress Controller serves any route in any namespace that is selected by the namespace selector.

Ingress Controller sharding is useful when balancing incoming traffic load among a set of Ingress Controllers and when isolating traffic to a specific Ingress Controller. For example, company A goes to one Ingress Controller and company B to another.

Procedure

1. Edit the **router-internal.yaml** file:

```
# cat router-internal.yaml
```

Example output

```
apiVersion: v1
items:
- apiVersion: operator.openshift.io/v1
  kind: IngressController
  metadata:
    name: sharded
    namespace: openshift-ingress-operator
  spec:
    domain: <apps-sharded.basedomain.example.net>
    nodePlacement:
      nodeSelector:
        matchLabels:
          node-role.kubernetes.io/worker: ""
    namespaceSelector:
      matchLabels:
        type: sharded
  status: {}
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

2. Apply the Ingress Controller **router-internal.yaml** file:

```
# oc apply -f router-internal.yaml
```

The Ingress Controller selects routes in any namespace that is selected by the namespace selector that have the label **type: sharded**.

5.7.7. Configuring an Ingress Controller to use an internal load balancer

When creating an Ingress Controller on cloud platforms, the Ingress Controller is published by a public cloud load balancer by default. As an administrator, you can create an Ingress Controller that uses an internal cloud load balancer.

> **WARNING**
>
> If your cloud provider is Microsoft Azure, you must have at least one public load balancer that points to your nodes. If you do not, all of your nodes will lose egress connectivity to the internet.

> **IMPORTANT**
>
> If you want to change the **scope** for an **IngressController** object, you must delete and then recreate that **IngressController** object. You cannot change the **.spec.endpointPublishingStrategy.loadBalancer.scope** parameter after the custom resource (CR) is created.

See the Kubernetes Services documentation for implementation details.

Prerequisites

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

Procedure

1. Create an **IngressController** custom resource (CR) in a file named **<name>-ingress-controller.yaml**, such as in the following example:

```
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  namespace: openshift-ingress-operator
  name: <name> 1
spec:
  domain: <domain> 2
  endpointPublishingStrategy:
    type: LoadBalancerService
    loadBalancer:
      scope: Internal 3
```

1 Replace **<name>** with a name for the **IngressController** object.

2. Specify the **domain** for the application published by the controller.

3. Specify a value of **Internal** to use an internal load balancer.

2. Create the Ingress Controller defined in the previous step by running the following command:

```
$ oc create -f <name>-ingress-controller.yaml 1
```

1. Replace **<name>** with the name of the **IngressController** object.

3. Optional: Confirm that the Ingress Controller was created by running the following command:

```
$ oc --all-namespaces=true get ingresscontrollers
```

5.7.8. Configuring global access for an Ingress Controller on GCP

An Ingress Controller created on GCP with an internal load balancer generates an internal IP address for the service. A cluster administrator can specify the global access option, which enables clients in any region within the same VPC network and compute region as the load balancer, to reach the workloads running on your cluster.

For more information, see the GCP documentation for global access.

Prerequisites

- You deployed an OpenShift Container Platform cluster on GCP infrastructure.
- You configured an Ingress Controller to use an internal load balancer.
- You installed the OpenShift CLI (**oc**).

Procedure

1. Configure the Ingress Controller resource to allow global access.

> **NOTE**
>
> You can also create an Ingress Controller and specify the global access option.

a. Configure the Ingress Controller resource:

```
$ oc -n openshift-ingress-operator edit ingresscontroller/default
```

b. Edit the YAML file:

Sample clientAccess configuration to Global

```
spec:
  endpointPublishingStrategy:
    loadBalancer:
      providerParameters:
```

```
          gcp:
            clientAccess: Global 1
          type: GCP
        scope: Internal
      type: LoadBalancerService
```

1 Set **gcp.clientAccess** to **Global**.

c. Save the file to apply the changes.

2. Run the following command to verify that the service allows global access:

```
$ oc -n openshift-ingress edit svc/router-default -o yaml
```

The output shows that global access is enabled for GCP with the annotation, **networking.gke.io/internal-load-balancer-allow-global-access**.

5.7.9. Configuring the default Ingress Controller for your cluster to be internal

You can configure the **default** Ingress Controller for your cluster to be internal by deleting and recreating it.

WARNING

If your cloud provider is Microsoft Azure, you must have at least one public load balancer that points to your nodes. If you do not, all of your nodes will lose egress connectivity to the internet.

IMPORTANT

If you want to change the **scope** for an **IngressController** object, you must delete and then recreate that **IngressController** object. You cannot change the **.spec.endpointPublishingStrategy.loadBalancer.scope** parameter after the custom resource (CR) is created.

Prerequisites

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

Procedure

1. Configure the **default** Ingress Controller for your cluster to be internal by deleting and recreating it.

```
$ oc replace --force --wait --filename - <<EOF
apiVersion: operator.openshift.io/v1
kind: IngressController
```

```
metadata:
  namespace: openshift-ingress-operator
  name: default
spec:
  endpointPublishingStrategy:
    type: LoadBalancerService
    loadBalancer:
      scope: Internal
EOF
```

5.7.10. Configuring the route admission policy

Administrators and application developers can run applications in multiple namespaces with the same domain name. This is for organizations where multiple teams develop microservices that are exposed on the same hostname.

> **WARNING**
>
> Allowing claims across namespaces should only be enabled for clusters with trust between namespaces, otherwise a malicious user could take over a hostname. For this reason, the default admission policy disallows hostname claims across namespaces.

Prerequisites

- Cluster administrator privileges.

Procedure

- Edit the **.spec.routeAdmission** field of the **ingresscontroller** resource variable using the following command:

```
$ oc -n openshift-ingress-operator patch ingresscontroller/default --patch '{"spec":
{"routeAdmission":{"namespaceOwnership":"InterNamespaceAllowed"}}}' --type=merge
```

Sample Ingress Controller configuration

```
spec:
  routeAdmission:
    namespaceOwnership: InterNamespaceAllowed
...
```

TIP

You can alternatively apply the following YAML to configure the route admission policy:

```
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: default
  namespace: openshift-ingress-operator
spec:
  routeAdmission:
    namespaceOwnership: InterNamespaceAllowed
```

5.7.11. Using wildcard routes

The HAProxy Ingress Controller has support for wildcard routes. The Ingress Operator uses **wildcardPolicy** to configure the **ROUTER_ALLOW_WILDCARD_ROUTES** environment variable of the Ingress Controller.

The default behavior of the Ingress Controller is to admit routes with a wildcard policy of **None**, which is backwards compatible with existing **IngressController** resources.

Procedure

1. Configure the wildcard policy.
   a. Use the following command to edit the **IngressController** resource:

   ```
   $ oc edit IngressController
   ```

   b. Under **spec**, set the **wildcardPolicy** field to **WildcardsDisallowed** or **WildcardsAllowed**:

   ```
   spec:
     routeAdmission:
       wildcardPolicy: WildcardsDisallowed # or WildcardsAllowed
   ```

5.7.12. Using X-Forwarded headers

You configure the HAProxy Ingress Controller to specify a policy for how to handle HTTP headers including **Forwarded** and **X-Forwarded-For**. The Ingress Operator uses the **HTTPHeaders** field to configure the **ROUTER_SET_FORWARDED_HEADERS** environment variable of the Ingress Controller.

Procedure

1. Configure the **HTTPHeaders** field for the Ingress Controller.
   a. Use the following command to edit the **IngressController** resource:

   ```
   $ oc edit IngressController
   ```

   b. Under **spec**, set the **HTTPHeaders** policy field to **Append**, **Replace**, **IfNone**, or **Never**:

```yaml
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: default
  namespace: openshift-ingress-operator
spec:
  httpHeaders:
    forwardedHeaderPolicy: Append
```

Example use cases
As a cluster administrator, you can:

- Configure an external proxy that injects the **X-Forwarded-For** header into each request before forwarding it to an Ingress Controller.
  To configure the Ingress Controller to pass the header through unmodified, you specify the **never** policy. The Ingress Controller then never sets the headers, and applications receive only the headers that the external proxy provides.

- Configure the Ingress Controller to pass the **X-Forwarded-For** header that your external proxy sets on external cluster requests through unmodified.
  To configure the Ingress Controller to set the **X-Forwarded-For** header on internal cluster requests, which do not go through the external proxy, specify the **if-none** policy. If an HTTP request already has the header set through the external proxy, then the Ingress Controller preserves it. If the header is absent because the request did not come through the proxy, then the Ingress Controller adds the header.

As an application developer, you can:

- Configure an application-specific external proxy that injects the **X-Forwarded-For** header.
  To configure an Ingress Controller to pass the header through unmodified for an application's Route, without affecting the policy for other Routes, add an annotation **haproxy.router.openshift.io/set-forwarded-headers: if-none** or **haproxy.router.openshift.io/set-forwarded-headers: never** on the Route for the application.

> **NOTE**
>
> You can set the **haproxy.router.openshift.io/set-forwarded-headers** annotation on a per route basis, independent from the globally set value for the Ingress Controller.

5.7.13. Enabling HTTP/2 Ingress connectivity

You can enable transparent end-to-end HTTP/2 connectivity in HAProxy. It allows application owners to make use of HTTP/2 protocol capabilities, including single connection, header compression, binary streams, and more.

You can enable HTTP/2 connectivity for an individual Ingress Controller or for the entire cluster.

To enable the use of HTTP/2 for the connection from the client to HAProxy, a route must specify a custom certificate. A route that uses the default certificate cannot use HTTP/2. This restriction is necessary to avoid problems from connection coalescing, where the client re-uses a connection for different routes that use the same certificate.

The connection from HAProxy to the application pod can use HTTP/2 only for re-encrypt routes and not for edge-terminated or insecure routes. This restriction is because HAProxy uses Application-Level

Protocol Negotiation (ALPN), which is a TLS extension, to negotiate the use of HTTP/2 with the back-end. The implication is that end-to-end HTTP/2 is possible with passthrough and re-encrypt and not with insecure or edge-terminated routes.

IMPORTANT

For non-passthrough routes, the Ingress Controller negotiates its connection to the application independently of the connection from the client. This means a client may connect to the Ingress Controller and negotiate HTTP/1.1, and the Ingress Controller may then connect to the application, negotiate HTTP/2, and forward the request from the client HTTP/1.1 connection using the HTTP/2 connection to the application. This poses a problem if the client subsequently tries to upgrade its connection from HTTP/1.1 to the WebSocket protocol, because the Ingress Controller cannot forward WebSocket to HTTP/2 and cannot upgrade its HTTP/2 connection to WebSocket. Consequently, if you have an application that is intended to accept WebSocket connections, it must not allow negotiating the HTTP/2 protocol or else clients will fail to upgrade to the WebSocket protocol.

Procedure

Enable HTTP/2 on a single Ingress Controller.

- To enable HTTP/2 on an Ingress Controller, enter the **oc annotate** command:

```
$ oc -n openshift-ingress-operator annotate ingresscontrollers/<ingresscontroller_name>
ingress.operator.openshift.io/default-enable-http2=true
```

  Replace **<ingresscontroller_name>** with the name of the Ingress Controller to annotate.

Enable HTTP/2 on the entire cluster.

- To enable HTTP/2 for the entire cluster, enter the **oc annotate** command:

```
$ oc annotate ingresses.config/cluster ingress.operator.openshift.io/default-enable-http2=true
```

TIP

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

```yaml
apiVersion: v1
kind: IngressController
metadata:
  annotations:
    ingress.operator.openshift.io/default-enable-http2: "true"
```

5.7.14. Configuring the PROXY protocol for an Ingress Controller

A cluster administrator can configure the PROXY protocol when an Ingress Controller uses either the **HostNetwork** or **NodePortService** endpoint publishing strategy types. The PROXY protocol enables the load balancer to preserve the original client addresses for connections that the Ingress Controller receives. The original client addresses are useful for logging, filtering, and injecting HTTP headers. In the default configuration, the connections that the Ingress Controller receives only contain the source address that is associated with the load balancer.

This feature is not supported in cloud deployments. This restriction is because when OpenShift Container Platform runs in a cloud platform, and an IngressController specifies that a service load balancer should be used, the Ingress Operator configures the load balancer service and enables the PROXY protocol based on the platform requirement for preserving source addresses.

> **WARNING**
>
> To prevent connection failure, configure both the Ingress Controller and the load balancer to use the PROXY protocol.

Prerequisites

- You created an Ingress Controller.

Procedure

1. Edit the Ingress Controller resource:

```
$ oc -n openshift-ingress-operator edit ingresscontroller/default
```

2. Set the PROXY configuration:

   - If your Ingress Controller uses the hostNetwork endpoint publishing strategy type, set the **spec.endpointPublishingStrategy.hostNetwork.protocol** subfield to **PROXY**:

     Sample hostNetwork configuration to PROXY

```
spec:
  endpointPublishingStrategy:
    hostNetwork:
      protocol: PROXY
    type: NodePortService
```

   - If your Ingress Controller uses the NodePortService endpoint publishing strategy type, set the **spec.endpointPublishingStrategy.nodePort.protocol** subfield to **PROXY**:

     Sample nodePort configuration to PROXY

```
spec:
  endpointPublishingStrategy:
    nodePort:
      protocol: PROXY
    type: NodePortService
```

5.7.15. Configuring application domain for the Ingress Controller Operator on AWS

As a cluster administrator, you can specify an alternative default domain for user-created routes by configuring **appsDomain** field. If you specify an alternative domain, it overrides the default cluster domain for the purpose of determining the default host for a new route.

For example, you can use the DNS domain for your company as the default domain for routes and ingresses for applications running on your cluster.

Prerequisites

- You deployed an OpenShift Container Platform cluster on AWS infrastructure.
- You installed the **oc** command line interface.

Procedure

1. Configure the **appsDomain** field by specifying an alternative default domain for user-created routes.
   a. Edit the Ingress Controller Operator:

   ```
   $ oc edit ingresses.config/cluster -o yaml
   ```

   b. Edit the YAML file:

   Sample appsDomain configuration to apps.acme.io

   ```
   apiVersion: config.openshift.io/v1
   kind: Ingress
   metadata:
     name: cluster
   spec:
     domain: apps.<domain_url>
     appsDomain: apps.acme.io
   ```

2. Verify that an existing route contains the new domain name by exposing the route and verifying the route domain change:

   NOTE

   Wait for the **openshift-apiserver** finish rolling updates before exposing the route.

   a. Expose the route:

   ```
   $ oc expose service hello-openshift
   route.route.openshift.io/hello-openshift exposed
   ```

   b. Verify route domain change:

   ```
   $ oc get routes
   NAME            HOST/PORT                             PATH   SERVICES          PORT
   TERMINATION   WILDCARD
   hello-openshift   hello-openshift-my-project.apps.acme.io        hello-openshift   8080-tcp
   None
   ```

5.7.16. Converting HTTP header case

HAProxy 2.2 lowercases HTTP header names by default, for example, changing **Host: xyz.com** to **host: xyz.com**. If legacy applications are sensitive to the capitalization of HTTP header names, use the Ingress Controller **spec.httpHeaders.headerNameCaseAdjustments** API field for a solution to accommodate legacy applications until they can be fixed.

IMPORTANT

Because OpenShift Container Platform 4.8 includes HAProxy 2.2, make sure to add the necessary configuration by using **spec.httpHeaders.headerNameCaseAdjustments** before upgrading.

Prerequisites

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

Procedure

As a cluster administrator, you can convert the HTTP header case by entering the **oc patch** command or by setting the **HeaderNameCaseAdjustments** field in the Ingress Controller YAML file.

- Specify an HTTP header to be capitalized by entering the **oc patch** command.
  1. Enter the **oc patch** command to change the HTTP **host** header to **Host**:

     ```
     $ oc -n openshift-ingress-operator patch ingresscontrollers/default --type=merge --
     patch='{"spec":{"httpHeaders":{"headerNameCaseAdjustments":["Host"]}}}'
     ```

  2. Annotate the route of the application:

     ```
     $ oc annotate routes/my-application haproxy.router.openshift.io/h1-adjust-case=true
     ```

     The Ingress Controller then adjusts the **host** request header as specified.

- Specify adjustments using the **HeaderNameCaseAdjustments** field by configuring the Ingress Controller YAML file.
  1. The following example Ingress Controller YAML adjusts the **host** header to **Host** for HTTP/1 requests to appropriately annotated routes:

     #### Example Ingress Controller YAML

     ```
     apiVersion: operator.openshift.io/v1
     kind: IngressController
     metadata:
       name: default
       namespace: openshift-ingress-operator
     spec:
       httpHeaders:
         headerNameCaseAdjustments:
         - Host
     ```

  2. The following example route enables HTTP response header name case adjustments using the **haproxy.router.openshift.io/h1-adjust-case** annotation:

Example route YAML

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/h1-adjust-case: true 1
  name: my-application
  namespace: my-application
spec:
  to:
    kind: Service
    name: my-application
```

1. Set **haproxy.router.openshift.io/h1-adjust-case** to true.

5.8. ADDITIONAL RESOURCES

- Configuring a custom PKI

CHAPTER 6. VERIFYING CONNECTIVITY TO AN ENDPOINT

The Cluster Network Operator (CNO) runs a controller, the connectivity check controller, that performs a connection health check between resources within your cluster. By reviewing the results of the health checks, you can diagnose connection problems or eliminate network connectivity as the cause of an issue that you are investigating.

6.1. CONNECTION HEALTH CHECKS PERFORMED

To verify that cluster resources are reachable, a TCP connection is made to each of the following cluster API services:

- Kubernetes API server service
- Kubernetes API server endpoints
- OpenShift API server service
- OpenShift API server endpoints
- Load balancers

To verify that services and service endpoints are reachable on every node in the cluster, a TCP connection is made to each of the following targets:

- Health check target service
- Health check target endpoints

6.2. IMPLEMENTATION OF CONNECTION HEALTH CHECKS

The connectivity check controller orchestrates connection verification checks in your cluster. 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.

The Cluster Network Operator (CNO) deploys several resources to the cluster to send and receive connectivity health checks:

Health check source

This program deploys in a single pod replica set managed by a **Deployment** object. The program consumes **PodNetworkConnectivity** objects and connects to the **spec.targetEndpoint** specified in each object.

Health check target

A pod deployed as part of a daemon set on every node in the cluster. The pod listens for inbound health checks. The presence of this pod on every node allows for the testing of connectivity to each node.

6.3. PODNETWORKCONNECTIVITYCHECK OBJECT FIELDS

The **PodNetworkConnectivityCheck** object fields are described in the following tables.

Table 6.1. PodNetworkConnectivityCheck object fields

Field	Type	Description
metadata.name	**string**	The name of the object in the following format: **<source>-to-<target>**. The destination described by **<target>** includes one of following strings: • **load-balancer-api-external** • **load-balancer-api-internal** • **kubernetes-apiserver-endpoint** • **kubernetes-apiserver-service-cluster** • **network-check-target** • **openshift-apiserver-endpoint** • **openshift-apiserver-service-cluster**
metadata.namespace	**string**	The namespace that the object is associated with. This value is always **openshift-network-diagnostics**.
spec.sourcePod	**string**	The name of the pod where the connection check originates, such as **network-check-source-596b4c6566-rgh92**.
spec.targetEndpoint	**string**	The target of the connection check, such as **api.devcluster.example.com:6443**.
spec.tlsClientCert	**object**	Configuration for the TLS certificate to use.
spec.tlsClientCert.name	**string**	The name of the TLS certificate used, if any. The default value is an empty string.
status	**object**	An object representing the condition of the connection test and logs of recent connection successes and failures.
status.conditions	**array**	The latest status of the connection check and any previous statuses.
status.failures	**array**	Connection test logs from unsuccessful attempts.
status.outages	**array**	Connect test logs covering the time periods of any outages.
status.successes	**array**	Connection test logs from successful attempts.

The following table describes the fields for objects in the **status.conditions** array:

Table 6.2. status.conditions

Field	Type	Description
lastTransitionTime	**string**	The time that the condition of the connection transitioned from one status to another.
message	**string**	The details about last transition in a human readable format.
reason	**string**	The last status of the transition in a machine readable format.
status	**string**	The status of the condition.
type	**string**	The type of the condition.

The following table describes the fields for objects in the **status.conditions** array:

Table 6.3. status.outages

Field	Type	Description
end	**string**	The timestamp from when the connection failure is resolved.
endLogs	**array**	Connection log entries, including the log entry related to the successful end of the outage.
message	**string**	A summary of outage details in a human readable format.
start	**string**	The timestamp from when the connection failure is first detected.
startLogs	**array**	Connection log entries, including the original failure.

Connection log fields

The fields for a connection log entry are described in the following table. The object is used in the following fields:

- **status.failures[]**
- **status.successes[]**
- **status.outages[].startLogs[]**
- **status.outages[].endLogs[]**

Table 6.4. Connection log object

Field	Type	Description
latency	**string**	Records the duration of the action.
message	**string**	Provides the status in a human readable format.
reason	**string**	Provides the reason for status in a machine readable format. The value is one of **TCPConnect**, **TCPConnectError**, **DNSResolve**, **DNSError**.
success	**boolean**	Indicates if the log entry is a success or failure.
time	**string**	The start time of connection check.

6.4. VERIFYING NETWORK CONNECTIVITY FOR AN ENDPOINT

As a cluster administrator, you can verify the connectivity of an endpoint, such as an API server, load balancer, service, or pod.

Prerequisites

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

Procedure

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

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

Example output

```
NAME                                                                                                                                AGE
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0   75m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-1   73m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-2   75m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-kubernetes-apiserver-service-cluster                               75m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-kubernetes-default-service-cluster                                 75m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-load-balancer-api-external                                         75m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-load-balancer-api-internal                                         75m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-network-check-target-ci-ln-x5sv9rb-f76d1-4rzrp-master-0            75m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-network-check-target-ci-
```

```
ln-x5sv9rb-f76d1-4rzrp-master-1            75m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-network-check-target-ci-
ln-x5sv9rb-f76d1-4rzrp-master-2            75m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-network-check-target-ci-
ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh      74m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-network-check-target-ci-
ln-x5sv9rb-f76d1-4rzrp-worker-c-n8mbf      74m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-network-check-target-ci-
ln-x5sv9rb-f76d1-4rzrp-worker-d-4hnrz      74m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-network-check-target-
service-cluster                            75m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-openshift-apiserver-
endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0    75m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-openshift-apiserver-
endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-1    75m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-openshift-apiserver-
endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-2    74m
network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-openshift-apiserver-
service-cluster                             75m
```

2. View the connection test logs:

   a. From the output of the previous command, identify the endpoint that you want to review the connectivity logs for.

   b. To view the object, enter the following command:

```
$ oc get podnetworkconnectivitycheck <name> \
  -n openshift-network-diagnostics -o yaml
```

   where **<name>** specifies the name of the **PodNetworkConnectivityCheck** object.

Example output

```
apiVersion: controlplane.operator.openshift.io/v1alpha1
kind: PodNetworkConnectivityCheck
metadata:
  name: network-check-source-ci-ln-x5sv9rb-f76d1-4rzrp-worker-b-6xdmh-to-kubernetes-
apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0
  namespace: openshift-network-diagnostics
  ...
spec:
  sourcePod: network-check-source-7c88f6d9f-hmg2f
  targetEndpoint: 10.0.0.4:6443
  tlsClientCert:
    name: ""
status:
  conditions:
  - lastTransitionTime: "2021-01-13T20:11:34Z"
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: tcp
      connection to 10.0.0.4:6443 succeeded'
    reason: TCPConnectSuccess
    status: "True"
    type: Reachable
  failures:
  - latency: 2.241775ms
```

```
  message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: failed
    to establish a TCP connection to 10.0.0.4:6443: dial tcp 10.0.0.4:6443: connect:
    connection refused'
  reason: TCPConnectError
  success: false
  time: "2021-01-13T20:10:34Z"
- latency: 2.582129ms
  message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: failed
    to establish a TCP connection to 10.0.0.4:6443: dial tcp 10.0.0.4:6443: connect:
    connection refused'
  reason: TCPConnectError
  success: false
  time: "2021-01-13T20:09:34Z"
- latency: 3.483578ms
  message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: failed
    to establish a TCP connection to 10.0.0.4:6443: dial tcp 10.0.0.4:6443: connect:
    connection refused'
  reason: TCPConnectError
  success: false
  time: "2021-01-13T20:08:34Z"
outages:
- end: "2021-01-13T20:11:34Z"
  endLogs:
  - latency: 2.032018ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0:
      tcp connection to 10.0.0.4:6443 succeeded'
    reason: TCPConnect
    success: true
    time: "2021-01-13T20:11:34Z"
  - latency: 2.241775ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0:
      failed to establish a TCP connection to 10.0.0.4:6443: dial tcp 10.0.0.4:6443:
      connect: connection refused'
    reason: TCPConnectError
    success: false
    time: "2021-01-13T20:10:34Z"
  - latency: 2.582129ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0:
      failed to establish a TCP connection to 10.0.0.4:6443: dial tcp 10.0.0.4:6443:
      connect: connection refused'
    reason: TCPConnectError
    success: false
    time: "2021-01-13T20:09:34Z"
  - latency: 3.483578ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0:
      failed to establish a TCP connection to 10.0.0.4:6443: dial tcp 10.0.0.4:6443:
      connect: connection refused'
    reason: TCPConnectError
    success: false
    time: "2021-01-13T20:08:34Z"
  message: Connectivity restored after 2m59.999789186s
  start: "2021-01-13T20:08:34Z"
  startLogs:
  - latency: 3.483578ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0:
      failed to establish a TCP connection to 10.0.0.4:6443: dial tcp 10.0.0.4:6443:
```

```
      connect: connection refused'
    reason: TCPConnectError
    success: false
    time: "2021-01-13T20:08:34Z"
  successes:
  - latency: 2.845865ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: tcp
      connection to 10.0.0.4:6443 succeeded'
    reason: TCPConnect
    success: true
    time: "2021-01-13T21:14:34Z"
  - latency: 2.926345ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: tcp
      connection to 10.0.0.4:6443 succeeded'
    reason: TCPConnect
    success: true
    time: "2021-01-13T21:13:34Z"
  - latency: 2.895796ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: tcp
      connection to 10.0.0.4:6443 succeeded'
    reason: TCPConnect
    success: true
    time: "2021-01-13T21:12:34Z"
  - latency: 2.696844ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: tcp
      connection to 10.0.0.4:6443 succeeded'
    reason: TCPConnect
    success: true
    time: "2021-01-13T21:11:34Z"
  - latency: 1.502064ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: tcp
      connection to 10.0.0.4:6443 succeeded'
    reason: TCPConnect
    success: true
    time: "2021-01-13T21:10:34Z"
  - latency: 1.388857ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: tcp
      connection to 10.0.0.4:6443 succeeded'
    reason: TCPConnect
    success: true
    time: "2021-01-13T21:09:34Z"
  - latency: 1.906383ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: tcp
      connection to 10.0.0.4:6443 succeeded'
    reason: TCPConnect
    success: true
    time: "2021-01-13T21:08:34Z"
  - latency: 2.089073ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: tcp
      connection to 10.0.0.4:6443 succeeded'
    reason: TCPConnect
    success: true
    time: "2021-01-13T21:07:34Z"
  - latency: 2.156994ms
    message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: tcp
      connection to 10.0.0.4:6443 succeeded'
```

```
  reason: TCPConnect
  success: true
  time: "2021-01-13T21:06:34Z"
- latency: 1.777043ms
  message: 'kubernetes-apiserver-endpoint-ci-ln-x5sv9rb-f76d1-4rzrp-master-0: tcp
    connection to 10.0.0.4:6443 succeeded'
  reason: TCPConnect
  success: true
  time: "2021-01-13T21:05:34Z"
```

CHAPTER 7. CONFIGURING THE NODE PORT SERVICE RANGE

As a cluster administrator, you can expand the available node port range. If your cluster uses of a large number of node ports, you might need to increase the number of available ports.

The default port range is **30000-32767**. You can never reduce the port range, even if you first expand it beyond the default range.

7.1. PREREQUISITES

- Your cluster infrastructure must allow access to the ports that you specify within the expanded range. For example, if you expand the node port range to **30000-32900**, the inclusive port range of **32768-32900** must be allowed by your firewall or packet filtering configuration.

7.2. EXPANDING THE NODE PORT RANGE

You can expand the node port range for the cluster.

Prerequisites

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

Procedure

1. To expand the node port range, enter the following command. Replace **<port>** with the largest port number in the new range.

```
$ oc patch network.config.openshift.io cluster --type=merge -p \
  '{
    "spec":
      { "serviceNodePortRange": "30000-<port>" }
  }'
```

TIP

You can alternatively apply the following YAML to update the node port range:

```
apiVersion: config.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  serviceNodePortRange: "30000-<port>"
```

Example output

```
network.config.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 configmaps -n openshift-kube-apiserver config \
  -o jsonpath="{.data['config\.yaml']}" | \
  grep -Eo '"service-node-port-range":["[[:digit:]]+-[[:digit:]]+"]'
```

Example output

```
"service-node-port-range":["30000-33000"]
```

7.3. ADDITIONAL RESOURCES

- Configuring ingress cluster traffic using a NodePort
- Network [config.openshift.io/v1]
- Service [core/v1]

CHAPTER 8. CONFIGURING IP FAILOVER

This topic describes configuring IP failover for pods and services on your OpenShift Container Platform cluster.

IP failover manages a pool of Virtual IP (VIP) addresses on a set of nodes. Every VIP in the set is serviced by a node selected from the set. As long a single node is available, the VIPs are served. There is no way to explicitly distribute the VIPs over the nodes, so there can be nodes with no VIPs and other nodes with many VIPs. If there is only one node, all VIPs are on it.

> **NOTE**
>
> The VIPs must be routable from outside the cluster.

IP failover monitors a port on each VIP to determine whether the port is reachable on the node. If the port is not reachable, the VIP is not assigned to the node. If the port is set to **0**, this check is suppressed. The check script does the needed testing.

IP failover uses Keepalived to host a set of externally accessible VIP addresses on a set of hosts. Each VIP is only serviced by a single host at a time. Keepalived uses the Virtual Router Redundancy Protocol (VRRP) to determine which host, from the set of hosts, services which VIP. If a host becomes unavailable, or if the service that Keepalived is watching does not respond, the VIP is switched to another host from the set. This means a VIP is always serviced as long as a host is available.

When a node running Keepalived passes the check script, the VIP on that node can enter the **master** state based on its priority and the priority of the current master and as determined by the preemption strategy.

A cluster administrator can provide a script through the **OPENSHIFT_HA_NOTIFY_SCRIPT** variable, and this script is called whenever the state of the VIP on the node changes. Keepalived uses the **master** state when it is servicing the VIP, the **backup** state when another node is servicing the VIP, or in the **fault** state when the check script fails. The notify script is called with the new state whenever the state changes.

You can create an IP failover deployment configuration on OpenShift Container Platform. The IP failover deployment configuration specifies the set of VIP addresses, and the set of nodes on which to service them. A cluster can have multiple IP failover deployment configurations, with each managing its own set of unique VIP addresses. Each node in the IP failover configuration runs an IP failover pod, and this pod runs Keepalived.

When using VIPs to access a pod with host networking, the application pod runs on all nodes that are running the IP failover pods. This enables any of the IP failover nodes to become the master and service the VIPs when needed. If application pods are not running on all nodes with IP failover, either some IP failover nodes never service the VIPs or some application pods never receive any traffic. Use the same selector and replication count, for both IP failover and the application pods, to avoid this mismatch.

While using VIPs to access a service, any of the nodes can be in the IP failover set of nodes, since the service is reachable on all nodes, no matter where the application pod is running. Any of the IP failover nodes can become master at any time. The service can either use external IPs and a service port or it can use a **NodePort**.

When using external IPs in the service definition, the VIPs are set to the external IPs, and the IP failover monitoring port is set to the service port. When using a node port, the port is open on every node in the cluster, and the service load-balances traffic from whatever node currently services the VIP. In this case, the IP failover monitoring port is set to the **NodePort** in the service definition.

IMPORTANT

Setting up a **NodePort** is a privileged operation.

IMPORTANT

Even though a service VIP is highly available, performance can still be affected. Keepalived makes sure that each of the VIPs is serviced by some node in the configuration, and several VIPs can end up on the same node even when other nodes have none. Strategies that externally load-balance across a set of VIPs can be thwarted when IP failover puts multiple VIPs on the same node.

When you use **ingressIP**, you can set up IP failover to have the same VIP range as the **ingressIP** range. You can also disable the monitoring port. In this case, all the VIPs appear on same node in the cluster. Any user can set up a service with an **ingressIP** and have it highly available.

IMPORTANT

There are a maximum of 254 VIPs in the cluster.

8.1. IP FAILOVER ENVIRONMENT VARIABLES

The following table contains the variables used to configure IP failover.

Table 8.1. IP failover environment variables

Variable Name	Default	Description
OPENSHIFT_HA_MONITOR_PORT	**80**	The IP failover pod tries to open a TCP connection to this port on each Virtual IP (VIP). If connection is established, the service is considered to be running. If this port is set to **0**, the test always passes.
OPENSHIFT_HA_NETWORK_INTERFACE		The interface name that IP failover uses to send Virtual Router Redundancy Protocol (VRRP) traffic. The default value is **eth0**.
OPENSHIFT_HA_REPLICA_COUNT	**2**	The number of replicas to create. This must match **spec.replicas** value in IP failover deployment configuration.
OPENSHIFT_HA_VIRTUAL_IPS		The list of IP address ranges to replicate. This must be provided. For example, **1.2.3.4-6,1.2.3.9**.
OPENSHIFT_HA_VRRP_ID_OFFSET	**0**	The offset value used to set the virtual router IDs. Using different offset values allows multiple IP failover configurations to exist within the same cluster. The default offset is **0**, and the allowed range is **0** through **255**.

Variable Name	Default	Description
OPENSHIFT_HA_VIP_GROUPS		The number of groups to create for VRRP. If not set, a group is created for each virtual IP range specified with the **OPENSHIFT_HA_VIP_GROUPS** variable.
OPENSHIFT_HA_IPTABLES_CHAIN	INPUT	The name of the iptables chain, to automatically add an **iptables** rule to allow the VRRP traffic on. If the value is not set, an **iptables** rule is not added. If the chain does not exist, it is not created.
OPENSHIFT_HA_CHECK_SCRIPT		The full path name in the pod file system of a script that is periodically run to verify the application is operating.
OPENSHIFT_HA_CHECK_INTERVAL	**2**	The period, in seconds, that the check script is run.
OPENSHIFT_HA_NOTIFY_SCRIPT		The full path name in the pod file system of a script that is run whenever the state changes.
OPENSHIFT_HA_PREEMPTION	**preempt_nodelay 300**	The strategy for handling a new higher priority host. The **nopreempt** strategy does not move master from the lower priority host to the higher priority host.

8.2. CONFIGURING IP FAILOVER

As a cluster administrator, you can configure IP failover on an entire cluster, or on a subset of nodes, as defined by the label selector. You can also configure multiple IP failover deployment configurations in your cluster, where each one is independent of the others.

The IP failover deployment configuration ensures that a failover pod runs on each of the nodes matching the constraints or the label used.

This pod runs Keepalived, which can monitor an endpoint and use Virtual Router Redundancy Protocol (VRRP) to fail over the virtual IP (VIP) from one node to another if the first node cannot reach the service or endpoint.

For production use, set a **selector** that selects at least two nodes, and set **replicas** equal to the number of selected nodes.

Prerequisites

- You are logged in to the cluster with a user with **cluster-admin** privileges.
- You created a pull secret.

Procedure

Procedure

1. Create an IP failover service account:

```
$ oc create sa ipfailover
```

2. Update security context constraints (SCC) for **hostNetwork**:

```
$ oc adm policy add-scc-to-user privileged -z ipfailover
$ oc adm policy add-scc-to-user hostnetwork -z ipfailover
```

3. Create a deployment YAML file to configure IP failover:

Example deployment YAML for IP failover configuration

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ipfailover-keepalived 1
  labels:
    ipfailover: hello-openshift
spec:
  strategy:
    type: Recreate
  replicas: 2
  selector:
    matchLabels:
      ipfailover: hello-openshift
  template:
    metadata:
      labels:
        ipfailover: hello-openshift
    spec:
      serviceAccountName: ipfailover
      privileged: true
      hostNetwork: true
      nodeSelector:
        node-role.kubernetes.io/worker: ""
      containers:
      - name: openshift-ipfailover
        image: quay.io/openshift/origin-keepalived-ipfailover
        ports:
        - containerPort: 63000
          hostPort: 63000
        imagePullPolicy: IfNotPresent
        securityContext:
          privileged: true
        volumeMounts:
        - name: lib-modules
          mountPath: /lib/modules
          readOnly: true
        - name: host-slash
          mountPath: /host
          readOnly: true
          mountPropagation: HostToContainer
        - name: etc-sysconfig
```

```
      mountPath: /etc/sysconfig
      readOnly: true
    - name: config-volume
      mountPath: /etc/keepalive
    env:
    - name: OPENSHIFT_HA_CONFIG_NAME
      value: "ipfailover"
    - name: OPENSHIFT_HA_VIRTUAL_IPS 2
      value: "1.1.1.1-2"
    - name: OPENSHIFT_HA_VIP_GROUPS 3
      value: "10"
    - name: OPENSHIFT_HA_NETWORK_INTERFACE 4
      value: "ens3" #The host interface to assign the VIPs
    - name: OPENSHIFT_HA_MONITOR_PORT 5
      value: "30060"
    - name: OPENSHIFT_HA_VRRP_ID_OFFSET 6
      value: "0"
    - name: OPENSHIFT_HA_REPLICA_COUNT 7
      value: "2" #Must match the number of replicas in the deployment
    - name: OPENSHIFT_HA_USE_UNICAST
      value: "false"
    #- name: OPENSHIFT_HA_UNICAST_PEERS
      #value: "10.0.148.40,10.0.160.234,10.0.199.110"
    - name: OPENSHIFT_HA_IPTABLES_CHAIN 8
      value: "INPUT"
    #- name: OPENSHIFT_HA_NOTIFY_SCRIPT 9
    #  value: /etc/keepalive/mynotifyscript.sh
    - name: OPENSHIFT_HA_CHECK_SCRIPT 10
      value: "/etc/keepalive/mycheckscript.sh"
    - name: OPENSHIFT_HA_PREEMPTION 11
      value: "preempt_delay 300"
    - name: OPENSHIFT_HA_CHECK_INTERVAL 12
      value: "2"
    livenessProbe:
      initialDelaySeconds: 10
      exec:
        command:
        - pgrep
        - keepalived
  volumes:
  - name: lib-modules
    hostPath:
      path: /lib/modules
  - name: host-slash
    hostPath:
      path: /
  - name: etc-sysconfig
    hostPath:
      path: /etc/sysconfig
  # config-volume contains the check script
  # created with `oc create configmap keepalived-checkscript --from-file=mycheckscript.sh`
  - configMap:
      defaultMode: 0755
      name: keepalived-checkscript
```

```
    name: config-volume
  imagePullSecrets:
    - name: openshift-pull-secret 13
```

1. The name of the IP failover deployment.
2. The list of IP address ranges to replicate. This must be provided. For example, **1.2.3.4-6,1.2.3.9**.
3. The number of groups to create for VRRP. If not set, a group is created for each virtual IP range specified with the **OPENSHIFT_HA_VIP_GROUPS** variable.
4. The interface name that IP failover uses to send VRRP traffic. By default, **eth0** is used.
5. The IP failover pod tries to open a TCP connection to this port on each VIP. If connection is established, the service is considered to be running. If this port is set to **0**, the test always passes. The default value is **80**.
6. The offset value used to set the virtual router IDs. Using different offset values allows multiple IP failover configurations to exist within the same cluster. The default offset is **0**, and the allowed range is **0** through **255**.
7. The number of replicas to create. This must match **spec.replicas** value in IP failover deployment configuration. The default value is **2**.
8. The name of the **iptables** chain to automatically add an **iptables** rule to allow the VRRP traffic on. If the value is not set, an **iptables** rule is not added. If the chain does not exist, it is not created, and Keepalived operates in unicast mode. The default is **INPUT**.
9. The full path name in the pod file system of a script that is run whenever the state changes.
10. The full path name in the pod file system of a script that is periodically run to verify the application is operating.
11. The strategy for handling a new higher priority host. The default value is **preempt_delay 300**, which causes a Keepalived instance to take over a VIP after 5 minutes if a lower-priority master is holding the VIP.
12. The period, in seconds, that the check script is run. The default value is **2**.
13. Create the pull secret before creating the deployment, otherwise you will get an error when creating the deployment.

8.3. ABOUT VIRTUAL IP ADDRESSES

Keepalived manages a set of virtual IP addresses (VIP). The administrator must make sure that all of these addresses:

- Are accessible on the configured hosts from outside the cluster.
- Are not used for any other purpose within the cluster.

Keepalived on each node determines whether the needed service is running. If it is, VIPs are supported and Keepalived participates in the negotiation to determine which node serves the VIP. For a node to participate, the service must be listening on the watch port on a VIP or the check must be disabled.

NOTE

Each VIP in the set may end up being served by a different node.

8.4. CONFIGURING CHECK AND NOTIFY SCRIPTS

Keepalived monitors the health of the application by periodically running an optional user supplied check script. For example, the script can test a web server by issuing a request and verifying the response.

When a check script is not provided, a simple default script is run that tests the TCP connection. This default test is suppressed when the monitor port is **0**.

Each IP failover pod manages a Keepalived daemon that manages one or more virtual IPs (VIP) on the node where the pod is running. The Keepalived daemon keeps the state of each VIP for that node. A particular VIP on a particular node may be in **master**, **backup**, or **fault** state.

When the check script for that VIP on the node that is in **master** state fails, the VIP on that node enters the **fault** state, which triggers a renegotiation. During renegotiation, all VIPs on a node that are not in the **fault** state participate in deciding which node takes over the VIP. Ultimately, the VIP enters the **master** state on some node, and the VIP stays in the **backup** state on the other nodes.

When a node with a VIP in **backup** state fails, the VIP on that node enters the **fault** state. When the check script passes again for a VIP on a node in the **fault** state, the VIP on that node exits the **fault** state and negotiates to enter the **master** state. The VIP on that node may then enter either the **master** or the **backup** state.

As cluster administrator, you can provide an optional notify script, which is called whenever the state changes. Keepalived passes the following three parameters to the script:

- **$1** - **group** or **instance**
- **$2** - Name of the **group** or **instance**
- **$3** - The new state: **master**, **backup**, or **fault**

The check and notify scripts run in the IP failover pod and use the pod file system, not the host file system. However, the IP failover pod makes the host file system available under the **/hosts** mount path. When configuring a check or notify script, you must provide the full path to the script. The recommended approach for providing the scripts is to use a config map.

The full path names of the check and notify scripts are added to the Keepalived configuration file, **_/etc/keepalived/keepalived.conf**, which is loaded every time Keepalived starts. The scripts can be added to the pod with a config map as follows.

Prerequisites

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

Procedure

1. Create the desired script and create a config map to hold it. The script has no input arguments and must return **0** for **OK** and **1** for **fail**.
   The check script, ***mycheckscript.sh***:

   ```
   #!/bin/bash
       # Whatever tests are needed
       # E.g., send request and verify response
   exit 0
   ```

2. Create the config map:

   ```
   $ oc create configmap mycustomcheck --from-file=mycheckscript.sh
   ```

3. Add the script to the pod. The **defaultMode** for the mounted config map files must able to run by using **oc** commands or by editing the deployment configuration. A value of **0755**, **493** decimal, is typical:

   ```
   $ oc set env deploy/ipfailover-keepalived \
       OPENSHIFT_HA_CHECK_SCRIPT=/etc/keepalive/mycheckscript.sh
   ```

   ```
   $ oc set volume deploy/ipfailover-keepalived --add --overwrite \
       --name=config-volume \
       --mount-path=/etc/keepalive \
       --source='{"configMap": { "name": "mycustomcheck", "defaultMode": 493}}'
   ```

NOTE

The **oc set env** command is whitespace sensitive. There must be no whitespace on either side of the **=** sign.

TIP

You can alternatively edit the **ipfailover-keepalived** deployment configuration:

```
$ oc edit deploy ipfailover-keepalived
```

```
    spec:
      containers:
      - env:
        - name: OPENSHIFT_HA_CHECK_SCRIPT  1
          value: /etc/keepalive/mycheckscript.sh
...
        volumeMounts: 2
        - mountPath: /etc/keepalive
          name: config-volume
      dnsPolicy: ClusterFirst
...
      volumes: 3
      - configMap:
          defaultMode: 0755 4
          name: customrouter
        name: config-volume
...
```

1. In the **spec.container.env** field, add the **OPENSHIFT_HA_CHECK_SCRIPT** environment variable to point to the mounted script file.
2. Add the **spec.container.volumeMounts** field to create the mount point.
3. Add a new **spec.volumes** field to mention the config map.
4. This sets run permission on the files. When read back, it is displayed in decimal, **493**.

Save the changes and exit the editor. This restarts **ipfailover-keepalived**.

8.5. CONFIGURING VRRP PREEMPTION

When a Virtual IP (VIP) on a node leaves the **fault** state by passing the check script, the VIP on the node enters the **backup** state if it has lower priority than the VIP on the node that is currently in the **master** state. However, if the VIP on the node that is leaving **fault** state has a higher priority, the preemption strategy determines its role in the cluster.

The **nopreempt** strategy does not move **master** from the lower priority VIP on the host to the higher priority VIP on the host. With **preempt_delay 300**, the default, Keepalived waits the specified 300 seconds and moves **master** to the higher priority VIP on the host.

Prerequisites

- You installed the OpenShift CLI (**oc**).

Procedure

- To specify preemption enter **oc edit deploy ipfailover-keepalived** to edit the router deployment configuration:

```
$ oc edit deploy ipfailover-keepalived
```

```
...
  spec:
    containers:
    - env:
      - name: OPENSHIFT_HA_PREEMPTION
        value: preempt_delay 300
...
```

1 Set the **OPENSHIFT_HA_PREEMPTION** value:

- **preempt_delay 300**: Keepalived waits the specified 300 seconds and moves **master** to the higher priority VIP on the host. This is the default value.
- **nopreempt**: does not move **master** from the lower priority VIP on the host to the higher priority VIP on the host.

8.6. ABOUT VRRP ID OFFSET

Each IP failover pod managed by the IP failover deployment configuration, **1** pod per node or replica, runs a Keepalived daemon. As more IP failover deployment configurations are configured, more pods are created and more daemons join into the common Virtual Router Redundancy Protocol (VRRP) negotiation. This negotiation is done by all the Keepalived daemons and it determines which nodes service which virtual IPs (VIP).

Internally, Keepalived assigns a unique **vrrp-id** to each VIP. The negotiation uses this set of **vrrp-ids**, when a decision is made, the VIP corresponding to the winning **vrrp-id** is serviced on the winning node.

Therefore, for every VIP defined in the IP failover deployment configuration, the IP failover pod must assign a corresponding **vrrp-id**. This is done by starting at **OPENSHIFT_HA_VRRP_ID_OFFSET** and sequentially assigning the **vrrp-ids** to the list of VIPs. The **vrrp-ids** can have values in the range **1..255**.

When there are multiple IP failover deployment configurations, you must specify **OPENSHIFT_HA_VRRP_ID_OFFSET** so that there is room to increase the number of VIPs in the deployment configuration and none of the **vrrp-id** ranges overlap.

8.7. CONFIGURING IP FAILOVER FOR MORE THAN 254 ADDRESSES

IP failover management is limited to 254 groups of Virtual IP (VIP) addresses. By default OpenShift Container Platform assigns one IP address to each group. You can use the **OPENSHIFT_HA_VIP_GROUPS** variable to change this so multiple IP addresses are in each group and define the number of VIP groups available for each Virtual Router Redundancy Protocol (VRRP) instance when configuring IP failover.

Grouping VIPs creates a wider range of allocation of VIPs per VRRP in the case of VRRP failover events, and is useful when all hosts in the cluster have access to a service locally. For example, when a service is being exposed with an **ExternalIP**.

> **NOTE**
>
> As a rule for failover, do not limit services, such as the router, to one specific host. Instead, services should be replicated to each host so that in the case of IP failover, the services do not have to be recreated on the new host.

NOTE

If you are using OpenShift Container Platform health checks, the nature of IP failover and groups means that all instances in the group are not checked. For that reason, the Kubernetes health checks must be used to ensure that services are live.

Prerequisites

- You are logged in to the cluster with a user with **cluster-admin** privileges.

Procedure

- To change the number of IP addresses assigned to each group, change the value for the **OPENSHIFT_HA_VIP_GROUPS** variable, for example:

  Example Deployment YAML for IP failover configuration

```
...
  spec:
    env:
    - name: OPENSHIFT_HA_VIP_GROUPS (1)
      value: "3"
...
```

(1) If **OPENSHIFT_HA_VIP_GROUPS** is set to **3** in an environment with seven VIPs, it creates three groups, assigning three VIPs to the first group, and two VIPs to the two remaining groups.

NOTE

If the number of groups set by **OPENSHIFT_HA_VIP_GROUPS** is fewer than the number of IP addresses set to fail over, the group contains more than one IP address, and all of the addresses move as a single unit.

8.8. HIGH AVAILABILITY FOR INGRESSIP

In non-cloud clusters, IP failover and **ingressIP** to a service can be combined. The result is high availability services for users that create services using **ingressIP**.

The approach is to specify an **ingressIPNetworkCIDR** range and then use the same range in creating the ipfailover configuration.

Because IP failover can support up to a maximum of 255 VIPs for the entire cluster, the **ingressIPNetworkCIDR** needs to be **/24** or smaller.

CHAPTER 9. USING THE STREAM CONTROL TRANSMISSION PROTOCOL (SCTP) ON A BARE METAL CLUSTER

As a cluster administrator, you can use the Stream Control Transmission Protocol (SCTP) on a cluster.

9.1. SUPPORT FOR STREAM CONTROL TRANSMISSION PROTOCOL (SCTP) ON OPENSHIFT CONTAINER PLATFORM

As a cluster administrator, you can enable SCTP on the hosts in the cluster. On Red Hat Enterprise Linux CoreOS (RHCOS), the SCTP module is disabled by default.

SCTP is a reliable message based protocol that runs on top of an IP network.

When enabled, you can use SCTP as a protocol with pods, services, and network policy. A **Service** object must be defined with the **type** parameter set to either the **ClusterIP** or **NodePort** value.

9.1.1. Example configurations using SCTP protocol

You can configure a pod or service to use SCTP by setting the **protocol** parameter to the **SCTP** value in the pod or service object.

In the following example, a pod is configured to use SCTP:

```
apiVersion: v1
kind: Pod
metadata:
  namespace: project1
  name: example-pod
spec:
  containers:
    - name: example-pod
...
      ports:
        - containerPort: 30100
          name: sctpserver
          protocol: SCTP
```

In the following example, a service is configured to use SCTP:

```
apiVersion: v1
kind: Service
metadata:
  namespace: project1
  name: sctpserver
spec:
...
  ports:
    - name: sctpserver
      protocol: SCTP
      port: 30100
      targetPort: 30100
  type: ClusterIP
```

In the following example, a **NetworkPolicy** object is configured to apply to SCTP network traffic on port **80** from any pods with a specific label:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-sctp-on-http
spec:
  podSelector:
    matchLabels:
      role: web
  ingress:
  - ports:
    - protocol: SCTP
      port: 80
```

9.2. ENABLING STREAM CONTROL TRANSMISSION PROTOCOL (SCTP)

As a cluster administrator, you can load and enable the blacklisted SCTP kernel module on worker nodes in your cluster.

Prerequisites

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

Procedure

1. Create a file named **load-sctp-module.yaml** that contains the following YAML definition:

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  name: load-sctp-module
  labels:
    machineconfiguration.openshift.io/role: worker
spec:
  config:
    ignition:
      version: 3.2.0
    storage:
      files:
        - path: /etc/modprobe.d/sctp-blacklist.conf
          mode: 0644
          overwrite: true
          contents:
            source: data:,
        - path: /etc/modules-load.d/sctp-load.conf
          mode: 0644
          overwrite: true
          contents:
            source: data:,sctp
```

2. To create the **MachineConfig** object, enter the following command:

```
$ oc create -f load-sctp-module.yaml
```

3. Optional: To watch the status of the nodes while the MachineConfig Operator applies the configuration change, enter the following command. When the status of a node transitions to **Ready**, the configuration update is applied.

```
$ oc get nodes
```

9.3. VERIFYING STREAM CONTROL TRANSMISSION PROTOCOL (SCTP) IS ENABLED

You can verify that SCTP is working on a cluster by creating a pod with an application that listens for SCTP traffic, associating it with a service, and then connecting to the exposed service.

Prerequisites

- Access to the internet from the cluster to install the **nc** package.
- Install the OpenShift CLI (**oc**).
- Access to the cluster as a user with the **cluster-admin** role.

Procedure

1. Create a pod starts an SCTP listener:
   a. Create a file named **sctp-server.yaml** that defines a pod with the following YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: sctpserver
  labels:
    app: sctpserver
spec:
  containers:
    - name: sctpserver
      image: registry.access.redhat.com/ubi8/ubi
      command: ["/bin/sh", "-c"]
      args:
        ["dnf install -y nc && sleep inf"]
      ports:
        - containerPort: 30102
          name: sctpserver
          protocol: SCTP
```

   b. Create the pod by entering the following command:

```
$ oc create -f sctp-server.yaml
```

2. Create a service for the SCTP listener pod.

a. Create a file named **sctp-service.yaml** that defines a service with the following YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: sctpservice
  labels:
    app: sctpserver
spec:
  type: NodePort
  selector:
    app: sctpserver
  ports:
    - name: sctpserver
      protocol: SCTP
      port: 30102
      targetPort: 30102
```

b. To create the service, enter the following command:

```
$ oc create -f sctp-service.yaml
```

3. Create a pod for the SCTP client.

a. Create a file named **sctp-client.yaml** with the following YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: sctpclient
  labels:
    app: sctpclient
spec:
  containers:
    - name: sctpclient
      image: registry.access.redhat.com/ubi8/ubi
      command: ["/bin/sh", "-c"]
      args:
        ["dnf install -y nc && sleep inf"]
```

b. To create the **Pod** object, enter the following command:

```
$ oc apply -f sctp-client.yaml
```

4. Run an SCTP listener on the server.

a. To connect to the server pod, enter the following command:

```
$ oc rsh sctpserver
```

b. To start the SCTP listener, enter the following command:

```
$ nc -l 30102 --sctp
```

5. Connect to the SCTP listener on the server.

    a. Open a new terminal window or tab in your terminal program.

    b. Obtain the IP address of the **sctpservice** service. Enter the following command:

    ```
    $ oc get services sctpservice -o go-template='{{.spec.clusterIP}}{{"\n"}}'
    ```

    c. To connect to the client pod, enter the following command:

    ```
    $ oc rsh sctpclient
    ```

    d. To start the SCTP client, enter the following command. Replace **<cluster_IP>** with the cluster IP address of the **sctpservice** service.

    ```
    # nc <cluster_IP> 30102 --sctp
    ```

CHAPTER 10. CONFIGURING PTP HARDWARE

IMPORTANT

Precision Time Protocol (PTP) hardware 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 https://access.redhat.com/support/offerings/techpreview/.

10.1. ABOUT PTP HARDWARE

OpenShift Container Platform includes the capability to use Precision Time Protocol (PTP)hardware on your nodes. You can configure linuxptp services on nodes in your cluster that have PTP-capable hardware.

NOTE

The PTP Operator works with PTP-capable devices on clusters provisioned only on bare metal infrastructure.

You can use the OpenShift Container Platform console to install PTP by deploying the PTP Operator. The PTP Operator creates and manages the linuxptp services. The Operator provides the following features:

- Discovery of the PTP-capable devices in a cluster.
- Management of the configuration of linuxptp services.

10.2. AUTOMATED DISCOVERY OF PTP NETWORK DEVICES

The PTP Operator adds the **NodePtpDevice.ptp.openshift.io** custom resource definition (CRD) to OpenShift Container Platform. The PTP Operator will search your cluster for PTP capable network devices on each node. The Operator creates and updates a **NodePtpDevice** custom resource (CR) object for each node that provides a compatible PTP device.

One CR is created for each node, and shares the same name as the node. The **.status.devices** list provides information about the PTP devices on a node.

The following is an example of a **NodePtpDevice** CR created by the PTP Operator:

```
apiVersion: ptp.openshift.io/v1
kind: NodePtpDevice
metadata:
  creationTimestamp: "2019-11-15T08:57:11Z"
  generation: 1
  name: dev-worker-0 1
  namespace: openshift-ptp 2
  resourceVersion: "487462"
```

```
  selfLink: /apis/ptp.openshift.io/v1/namespaces/openshift-ptp/nodeptpdevices/dev-worker-0
  uid: 08d133f7-aae2-403f-84ad-1fe624e5ab3f
spec: {}
status:
  devices: 3
  - name: eno1
  - name: eno2
  - name: ens787f0
  - name: ens787f1
  - name: ens801f0
  - name: ens801f1
  - name: ens802f0
  - name: ens802f1
  - name: ens803
```

1. The value for the **name** parameter is the same as the name of the node.
2. The CR is created in **openshift-ptp** namespace by PTP Operator.
3. The **devices** collection includes a list of all of the PTP capable devices discovered by the Operator on the node.

10.3. INSTALLING THE PTP OPERATOR

As a cluster administrator, you can install the PTP Operator using the OpenShift Container Platform CLI or the web console.

10.3.1. CLI: Installing the PTP Operator

As a cluster administrator, you can install the Operator using the CLI.

Prerequisites

- A cluster installed on bare-metal hardware with nodes that have hardware that supports PTP.
- Install the OpenShift CLI (**oc**).
- Log in as a user with **cluster-admin** privileges.

Procedure

1. To create a namespace for the PTP Operator, enter the following command:

```
$ cat << EOF| oc create -f -
apiVersion: v1
kind: Namespace
metadata:
  name: openshift-ptp
  annotations:
    workload.openshift.io/allowed: management
  labels:
    name: openshift-ptp
    openshift.io/cluster-monitoring: "true"
EOF
```

–

2. To create an Operator group for the Operator, enter the following command:

```
$ cat << EOF| oc create -f -
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: ptp-operators
  namespace: openshift-ptp
spec:
  targetNamespaces:
  - openshift-ptp
EOF
```

3. Subscribe to the PTP Operator.

   a. Run the following command to set the OpenShift Container Platform major and minor version as an environment variable, which is used as the **channel** value in the next step.

   ```
   $ OC_VERSION=$(oc version -o yaml | grep openshiftVersion | \
       grep -o '[0-9]*[.][0-9]*' | head -1)
   ```

   b. To create a subscription for the PTP Operator, enter the following command:

   ```
   $ cat << EOF| oc create -f -
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
     name: ptp-operator-subscription
     namespace: openshift-ptp
   spec:
     channel: "${OC_VERSION}"
     name: ptp-operator
     source: redhat-operators
     sourceNamespace: openshift-marketplace
   EOF
   ```

4. To verify that the Operator is installed, enter the following command:

```
$ oc get csv -n openshift-ptp \
  -o custom-columns=Name:.metadata.name,Phase:.status.phase
```

Example output

```
Name                                   Phase
ptp-operator.4.4.0-202006160135        Succeeded
```

10.3.2. Web console: Installing the PTP Operator

As a cluster administrator, you can install the Operator using the web console.

> **NOTE**
>
> You have to create the namespace and operator group as mentioned in the previous section.

Procedure

1. Install the PTP Operator using the OpenShift Container Platform web console:
    a. In the OpenShift Container Platform web console, click **Operators → OperatorHub**.
    b. Choose **PTP Operator** from the list of available Operators, and then click **Install**.
    c. On the **Install Operator** page, under **A specific namespace on the cluster** select **openshift-ptp**. Then, click **Install**.
2. Optional: Verify that the PTP Operator installed successfully:
    a. Switch to the **Operators → Installed Operators** page.
    b. Ensure that **PTP Operator** is listed in the **openshift-ptp** project with a **Status** of **InstallSucceeded**.

    > **NOTE**
    >
    > During installation an Operator might display a **Failed** status. If the installation later succeeds with an **InstallSucceeded** message, you can ignore the **Failed** message.

    If the operator does not appear as installed, to troubleshoot further:

    - Go to the **Operators → Installed Operators** page and inspect the **Operator Subscriptions** and **Install Plans** tabs for any failure or errors under **Status**.
    - Go to the **Workloads → Pods** page and check the logs for pods in the **openshift-ptp** project.

10.4. CONFIGURING LINUXPTP SERVICES

The PTP Operator adds the **PtpConfig.ptp.openshift.io** custom resource definition (CRD) to OpenShift Container Platform. You can configure the Linuxptp services (ptp4l, phc2sys) by creating a **PtpConfig** custom resource (CR) object.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in as a user with **cluster-admin** privileges.
- You must have installed the PTP Operator.

Procedure

1. Create the following **PtpConfig** CR, and then save the YAML in the **<name>-ptp-config.yaml** file. Replace **<name>** with the name for this configuration.

```
apiVersion: ptp.openshift.io/v1
kind: PtpConfig
metadata:
  name: <name> 1
  namespace: openshift-ptp 2
spec:
  profile: 3
  - name: "profile1" 4
    interface: "ens787f1" 5
    ptp4lOpts: "-s -2" 6
    phc2sysOpts: "-a -r" 7
  recommend: 8
  - profile: "profile1" 9
    priority: 10 10
    match: 11
    - nodeLabel: "node-role.kubernetes.io/worker" 12
      nodeName: "dev-worker-0" 13
```

1. Specify a name for the **PtpConfig** CR.
2. Specify the namespace where the PTP Operator is installed.
3. Specify an array of one or more **profile** objects.
4. Specify the name of a profile object which is used to uniquely identify a profile object.
5. Specify the network interface name to use by the **ptp4l** service, for example **ens787f1**.
6. Specify system config options for the **ptp4l** service, for example **-s -2**. This should not include the interface name **-i <interface>** and service config file **-f /etc/ptp4l.conf** because these will be automatically appended.
7. Specify system config options for the **phc2sys** service, for example **-a -r**.
8. Specify an array of one or more **recommend** objects which define rules on how the **profile** should be applied to nodes.
9. Specify the **profile** object name defined in the **profile** section.
10. Specify the **priority** with an integer value between **0** and **99**. A larger number gets lower priority, so a priority of **99** is lower than a priority of **10**. If a node can be matched with multiple profiles according to rules defined in the **match** field, the profile with the higher priority will be applied to that node.
11. Specify **match** rules with **nodeLabel** or **nodeName**.
12. Specify **nodeLabel** with the **key** of **node.Labels** from the node object.
13. Specify **nodeName** with **node.Name** from the node object.

2. Create the CR by running the following command:

```
$ oc create -f <filename> 1
```

1. Replace **<filename>** with the name of the file you created in the previous step.

3. Optional: Check that the **PtpConfig** profile is applied to nodes that match with **nodeLabel** or **nodeName**.

```
$ oc get pods -n openshift-ptp -o wide
```

Example output

```
NAME                    READY   STATUS    RESTARTS   AGE   IP             NODE
NOMINATED NODE   READINESS GATES
linuxptp-daemon-4xkbb         1/1     Running   0          43m   192.168.111.15   dev-worker-0
<none>           <none>
linuxptp-daemon-tdspf         1/1     Running   0          43m   192.168.111.11   dev-master-0
<none>           <none>
ptp-operator-657bbb64c8-2f8sj   1/1     Running   0          43m   10.128.0.116     dev-master-0
<none>           <none>

$ oc logs linuxptp-daemon-4xkbb -n openshift-ptp
I1115 09:41:17.117596 4143292 daemon.go:107] in applyNodePTPProfile
I1115 09:41:17.117604 4143292 daemon.go:109] updating NodePTPProfile to:
I1115 09:41:17.117607 4143292 daemon.go:110] ------------------------------------
I1115 09:41:17.117612 4143292 daemon.go:102] Profile Name: profile1 1
I1115 09:41:17.117616 4143292 daemon.go:102] Interface: ens787f1 2
I1115 09:41:17.117620 4143292 daemon.go:102] Ptp4lOpts: -s -2 3
I1115 09:41:17.117623 4143292 daemon.go:102] Phc2sysOpts: -a -r 4
I1115 09:41:17.117626 4143292 daemon.go:116] ------------------------------------
I1115 09:41:18.117934 4143292 daemon.go:186] Starting phc2sys...
I1115 09:41:18.117985 4143292 daemon.go:187] phc2sys cmd: &{Path:/usr/sbin/phc2sys
Args:[/usr/sbin/phc2sys -a -r] Env:[] Dir: Stdin:<nil> Stdout:<nil> Stderr:<nil> ExtraFiles:[]
SysProcAttr:<nil> Process:<nil> ProcessState:<nil> ctx:<nil> lookPathErr:<nil> finished:false
childFiles:[] closeAfterStart:[] closeAfterWait:[] goroutine:[] errch:<nil> waitDone:<nil>}
I1115 09:41:19.118175 4143292 daemon.go:186] Starting ptp4l...
I1115 09:41:19.118209 4143292 daemon.go:187] ptp4l cmd: &{Path:/usr/sbin/ptp4l Args:
[/usr/sbin/ptp4l -m -f /etc/ptp4l.conf -i ens787f1 -s -2] Env:[] Dir: Stdin:<nil> Stdout:<nil>
Stderr:<nil> ExtraFiles:[] SysProcAttr:<nil> Process:<nil> ProcessState:<nil> ctx:<nil>
lookPathErr:<nil> finished:false childFiles:[] closeAfterStart:[] closeAfterWait:[] goroutine:[]
errch:<nil> waitDone:<nil>}
ptp4l[102189.864]: selected /dev/ptp5 as PTP clock
ptp4l[102189.886]: port 1: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[102189.886]: port 0: INITIALIZING to LISTENING on INIT_COMPLETE
```

1. **Profile Name** is the name that is applied to node **dev-worker-0**.
2. **Interface** is the PTP device specified in the **profile1** interface field. The **ptp4l** service runs on this interface.
3. **Ptp4lOpts** are the ptp4l sysconfig options specified in **profile1** Ptp4lOpts field.
4. **Phc2sysOpts** are the phc2sys sysconfig options specified in **profile1** Phc2sysOpts field.

CHAPTER 11. NETWORK POLICY

11.1. ABOUT NETWORK POLICY

As a cluster administrator, you can define network policies that restrict traffic to pods in your cluster.

11.1.1. About network policy

In a cluster using a Kubernetes Container Network Interface (CNI) plug-in that supports Kubernetes network policy, network isolation is controlled entirely by **NetworkPolicy** objects. In OpenShift Container Platform 4.8, OpenShift SDN supports using network policy in its default network isolation mode.

> NOTE
>
> When using the OpenShift SDN cluster network provider, the following limitations apply regarding network policies:
>
> - Egress network policy as specified by the **egress** field is not supported.
> - IPBlock is supported by network policy, but without support for **except** clauses. If you create a policy with an IPBlock section that includes an **except** clause, the SDN pods log warnings and the entire IPBlock section of that policy is ignored.

> WARNING
>
> Network policy does not apply to the host network namespace. Pods with host networking enabled are unaffected by network policy rules.

By default, all pods in a project are accessible from other pods and network endpoints. To isolate one or more pods in a project, you can create **NetworkPolicy** objects in that project to indicate the allowed incoming connections. Project administrators can create and delete **NetworkPolicy** objects within their own project.

If a pod is matched by selectors in one or more **NetworkPolicy** objects, then the pod will accept only connections that are allowed by at least one of those **NetworkPolicy** objects. A pod that is not selected by any **NetworkPolicy** objects is fully accessible.

The following example **NetworkPolicy** objects demonstrate supporting different scenarios:

- Deny all traffic:
  To make a project deny by default, add a **NetworkPolicy** object that matches all pods but accepts no traffic:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-by-default
```

```
spec:
  podSelector:
  ingress: []
```

- Only allow connections from the OpenShift Container Platform Ingress Controller:
  To make a project allow only connections from the OpenShift Container Platform Ingress Controller, add the following **NetworkPolicy** object.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-ingress
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          network.openshift.io/policy-group: ingress
  podSelector: {}
  policyTypes:
  - Ingress
```

- Only accept connections from pods within a project:
  To make pods accept connections from other pods in the same project, but reject all other connections from pods in other projects, add the following **NetworkPolicy** object:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-same-namespace
spec:
  podSelector:
  ingress:
  - from:
    - podSelector: {}
```

- Only allow HTTP and HTTPS traffic based on pod labels:
  To enable only HTTP and HTTPS access to the pods with a specific label (**role=frontend** in following example), add a **NetworkPolicy** object similar to the following:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-http-and-https
spec:
  podSelector:
    matchLabels:
      role: frontend
  ingress:
  - ports:
    - protocol: TCP
      port: 80
    - protocol: TCP
      port: 443
```

- Accept connections by using both namespace and pod selectors:
  To match network traffic by combining namespace and pod selectors, you can use a **NetworkPolicy** object similar to the following:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-pod-and-namespace-both
spec:
  podSelector:
    matchLabels:
      name: test-pods
  ingress:
    - from:
      - namespaceSelector:
          matchLabels:
            project: project_name
        podSelector:
          matchLabels:
            name: test-pods
```

NetworkPolicy objects are additive, which means you can combine multiple **NetworkPolicy** objects together to satisfy complex network requirements.

For example, for the **NetworkPolicy** objects defined in previous samples, you can define both **allow-same-namespace** and **allow-http-and-https** policies within the same project. Thus allowing the pods with the label **role=frontend**, to accept any connection allowed by each policy. That is, connections on any port from pods in the same namespace, and connections on ports **80** and **443** from pods in any namespace.

11.1.2. Optimizations for network policy

Use a network policy to isolate pods that are differentiated from one another by labels within a namespace.

> **NOTE**
>
> The guidelines for efficient use of network policy rules applies to only the OpenShift SDN cluster network provider.

It is inefficient to apply **NetworkPolicy** objects to large numbers of individual pods in a single namespace. Pod labels do not exist at the IP address level, so a network policy generates a separate Open vSwitch (OVS) flow rule for every possible link between every pod selected with a **podSelector**.

For example, if the spec **podSelector** and the ingress **podSelector** within a **NetworkPolicy** object each match 200 pods, then 40,000 (200*200) OVS flow rules are generated. This might slow down a node.

When designing your network policy, refer to the following guidelines:

- Reduce the number of OVS flow rules by using namespaces to contain groups of pods that need to be isolated.
  NetworkPolicy objects that select a whole namespace, by using the **namespaceSelector** or an empty **podSelector**, generate only a single OVS flow rule that matches the VXLAN virtual network ID (VNID) of the namespace.

- Keep the pods that do not need to be isolated in their original namespace, and move the pods that require isolation into one or more different namespaces.
- Create additional targeted cross-namespace network policies to allow the specific traffic that you do want to allow from the isolated pods.

11.1.3. Next steps

- Creating a network policy
- Optional: Defining a default network policy

11.1.4. Additional resources

- Projects and namespaces
- Configuring multitenant network policy
- NetworkPolicy API

11.2. LOGGING NETWORK POLICY EVENTS

As a cluster administrator, you can configure network policy audit logging for your cluster and enable logging for one or more namespaces.

> **NOTE**
>
> Audit logging of network policies is available for only the OVN-Kubernetes cluster network provider.

11.2.1. Network policy audit logging

The OVN-Kubernetes cluster network provider uses Open Virtual Network (OVN) ACLs to manage network policy. Audit logging exposes allow and deny ACL events.

You can configure the destination for network policy audit logs, such as a syslog server or a UNIX domain socket. Regardless of any additional configuration, an audit log is always saved to **/var/log/ovn/acl-audit-log.log** on each OVN-Kubernetes pod in the cluster.

Network policy audit logging is enabled per namespace by annotating the namespace with the **k8s.ovn.org/acl-logging** key as in the following example:

Example namespace annotation

```
kind: Namespace
apiVersion: v1
metadata:
  name: example1
  annotations:
    k8s.ovn.org/acl-logging: |-
      {
        "deny": "info",
        "allow": "info"
      }
```

The logging format is compatible with syslog as defined by RFC5424. The syslog facility is configurable and defaults to **local0**. An example log entry might resemble the following:

Example ACL deny log entry

```
2021-06-13T19:33:11.590Z|00005|acl_log(ovn_pinctrl0)|INFO|name="verify-audit-logging_deny-all", verdict=drop, severity=alert:
icmp,vlan_tci=0x0000,dl_src=0a:58:0a:80:02:39,dl_dst=0a:58:0a:80:02:37,nw_src=10.128.2.57,nw_dst=
10.128.2.55,nw_tos=0,nw_ecn=0,nw_ttl=64,icmp_type=8,icmp_code=0
```

The following table describes namespace annotation values:

Table 11.1. Network policy audit logging namespace annotation

Annotation	Value
k8s.ovn.org/acl-logging	You must specify at least one of **allow**, **deny**, or both to enable network policy audit logging for a namespace. **deny** Optional: Specify **alert**, **warning**, **notice**, **info**, or **debug**. **allow** Optional: Specify **alert**, **warning**, **notice**, **info**, or **debug**.

11.2.2. Network policy audit configuration

The configuration for audit logging is specified as part of the OVN-Kubernetes cluster network provider configuration. The following YAML illustrates default values for network policy audit logging feature.

Audit logging configuration

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  defaultNetwork:
    ovnKubernetesConfig:
      policyAuditConfig:
        destination: "null"
        maxFileSize: 50
        rateLimit: 20
        syslogFacility: local0
```

The following table describes the configuration fields for network policy audit logging.

Table 11.2. policyAuditConfig object

Field	Type	Description
rateLimit	integer	The maximum number of messages to generate every second per node. The default value is **20** messages per second.

Field	Type	Description
maxFileSize	integer	The maximum size for the audit log in bytes. The default value is **50000000** or 50MB.
destination	string	One of the following additional audit log targets: **libc** The libc **syslog()** function of the journald process on the host. **udp:<host>:<port>** A syslog server. Replace **<host>:<port>** with the host and port of the syslog server. **unix:<file>** A Unix Domain Socket file specified by **<file>**. **null** Do not send the audit logs to any additional target.
syslogFacility	string	The syslog facility, such as **kern**, as defined by RFC5424. The default value is **local0**.

11.2.3. Configuring network policy auditing for a cluster

As a cluster administrator, you can customize network policy audit logging for your cluster.

Prerequisites

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

Procedure

- To customize the network policy audit logging configuration, enter the following command:

```
$ oc edit network.operator.openshift.io/cluster
```

TIP

You can alternatively customize and apply the following YAML to configure audit logging:

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  defaultNetwork:
    ovnKubernetesConfig:
      policyAuditConfig:
        destination: "null"
        maxFileSize: 50
        rateLimit: 20
        syslogFacility: local0
```

Verification

1. To create a namespace with network policies complete the following steps:

   a. Create a namespace for verification:

   ```
   $ cat <<EOF| oc create -f -
   kind: Namespace
   apiVersion: v1
   metadata:
     name: verify-audit-logging
     annotations:
       k8s.ovn.org/acl-logging: '{ "deny": "alert", "allow": "alert" }'
   EOF
   ```

   Example output

   ```
   namespace/verify-audit-logging created
   ```

   b. Enable audit logging:

   ```
   $ oc annotate namespace verify-audit-logging k8s.ovn.org/acl-logging='{ "deny": "alert",
   "allow": "alert" }'
   ```

   ```
   namespace/verify-audit-logging annotated
   ```

   c. Create network policies for the namespace:

   ```
   $ cat <<EOF| oc create -n verify-audit-logging -f -
   apiVersion: networking.k8s.io/v1
   kind: NetworkPolicy
   metadata:
     name: deny-all
   spec:
     podSelector:
       matchLabels:
   ```

```
  policyTypes:
  - Ingress
  - Egress
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-same-namespace
spec:
  podSelector: {}
  policyTypes:
   - Ingress
   - Egress
  ingress:
    - from:
        - podSelector: {}
  egress:
    - to:
       - namespaceSelector:
          matchLabels:
            namespace: verify-audit-logging
EOF
```

Example output

```
networkpolicy.networking.k8s.io/deny-all created
networkpolicy.networking.k8s.io/allow-from-same-namespace created
```

2. Create a pod for source traffic in the **default** namespace:

```
$ cat <<EOF| oc create -n default -f -
apiVersion: v1
kind: Pod
metadata:
  name: client
spec:
  containers:
    - name: client
      image: registry.access.redhat.com/rhel7/rhel-tools
      command: ["/bin/sh", "-c"]
      args:
        ["sleep inf"]
EOF
```

3. Create two pods in the **verify-audit-logging** namespace:

```
$ for name in client server; do
cat <<EOF| oc create -n verify-audit-logging -f -
apiVersion: v1
kind: Pod
metadata:
  name: ${name}
spec:
  containers:
    - name: ${name}
```

```
    image: registry.access.redhat.com/rhel7/rhel-tools
    command: ["/bin/sh", "-c"]
    args:
      ["sleep inf"]
EOF
done
```

Example output

```
pod/client created
pod/server created
```

4. To generate traffic and produce network policy audit log entries, complete the following steps:

   a. Obtain the IP address for pod named **server** in the **verify-audit-logging** namespace:

   ```
   $ POD_IP=$(oc get pods server -n verify-audit-logging -o jsonpath='{.status.podIP}')
   ```

   b. Ping the IP address from the previous command from the pod named **client** in the **default** namespace and confirm that all packets are dropped:

   ```
   $ oc exec -it client -n default -- /bin/ping -c 2 $POD_IP
   ```

   ### Example output

   ```
   PING 10.128.2.55 (10.128.2.55) 56(84) bytes of data.

   --- 10.128.2.55 ping statistics ---
   2 packets transmitted, 0 received, 100% packet loss, time 2041ms
   ```

   c. Ping the IP address saved in the **POD_IP** shell environment variable from the pod named **client** in the **verify-audit-logging** namespace and confirm that all packets are allowed:

   ```
   $ oc exec -it client -n verify-audit-logging -- /bin/ping -c 2 $POD_IP
   ```

   ### Example output

   ```
   PING 10.128.0.86 (10.128.0.86) 56(84) bytes of data.
   64 bytes from 10.128.0.86: icmp_seq=1 ttl=64 time=2.21 ms
   64 bytes from 10.128.0.86: icmp_seq=2 ttl=64 time=0.440 ms

   --- 10.128.0.86 ping statistics ---
   2 packets transmitted, 2 received, 0% packet loss, time 1001ms
   rtt min/avg/max/mdev = 0.440/1.329/2.219/0.890 ms
   ```

5. Display the latest entries in the network policy audit log:

```
$ for pod in $(oc get pods -n openshift-ovn-kubernetes -l app=ovnkube-node --no-
headers=true | awk '{ print $1 }') ; do
    oc exec -it $pod -n openshift-ovn-kubernetes -- tail -4 /var/log/ovn/acl-audit-log.log
  done
```

Example output

```
Defaulting container name to ovn-controller.
Use 'oc describe pod/ovnkube-node-hdb8v -n openshift-ovn-kubernetes' to see all of the
containers in this pod.
2021-06-13T19:33:11.590Z|00005|acl_log(ovn_pinctrl0)|INFO|name="verify-audit-
logging_deny-all", verdict=drop, severity=alert:
icmp,vlan_tci=0x0000,dl_src=0a:58:0a:80:02:39,dl_dst=0a:58:0a:80:02:37,nw_src=10.128.2.57,
nw_dst=10.128.2.55,nw_tos=0,nw_ecn=0,nw_ttl=64,icmp_type=8,icmp_code=0
2021-06-13T19:33:12.614Z|00006|acl_log(ovn_pinctrl0)|INFO|name="verify-audit-
logging_deny-all", verdict=drop, severity=alert:
icmp,vlan_tci=0x0000,dl_src=0a:58:0a:80:02:39,dl_dst=0a:58:0a:80:02:37,nw_src=10.128.2.57,
nw_dst=10.128.2.55,nw_tos=0,nw_ecn=0,nw_ttl=64,icmp_type=8,icmp_code=0
2021-06-13T19:44:10.037Z|00007|acl_log(ovn_pinctrl0)|INFO|name="verify-audit-
logging_allow-from-same-namespace_0", verdict=allow, severity=alert:
icmp,vlan_tci=0x0000,dl_src=0a:58:0a:80:02:3b,dl_dst=0a:58:0a:80:02:3a,nw_src=10.128.2.59,
nw_dst=10.128.2.58,nw_tos=0,nw_ecn=0,nw_ttl=64,icmp_type=8,icmp_code=0
2021-06-13T19:44:11.037Z|00008|acl_log(ovn_pinctrl0)|INFO|name="verify-audit-
logging_allow-from-same-namespace_0", verdict=allow, severity=alert:
icmp,vlan_tci=0x0000,dl_src=0a:58:0a:80:02:3b,dl_dst=0a:58:0a:80:02:3a,nw_src=10.128.2.59,
nw_dst=10.128.2.58,nw_tos=0,nw_ecn=0,nw_ttl=64,icmp_type=8,icmp_code=0
```

11.2.4. Enabling network policy audit logging for a namespace

As a cluster administrator, you can enable network policy audit logging for a namespace.

Prerequisites

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

Procedure

- To enable network policy audit logging for a namespace, enter the following command:

```
$ oc annotate namespace <namespace> \
  k8s.ovn.org/acl-logging='{ "deny": "alert", "allow": "notice" }'
```

where:

<namespace>
: Specifies the name of the namespace.

TIP

You can alternatively apply the following YAML to enable audit logging:

```
kind: Namespace
apiVersion: v1
metadata:
  name: <namespace>
  annotations:
    k8s.ovn.org/acl-logging: |-
      {
        "deny": "alert",
        "allow": "notice"
      }
```

Example output

```
namespace/verify-audit-logging annotated
```

Verification

- Display the latest entries in the network policy audit log:

```
$ for pod in $(oc get pods -n openshift-ovn-kubernetes -l app=ovnkube-node --no-headers=true | awk '{ print $1 }') ; do
    oc exec -it $pod -n openshift-ovn-kubernetes -- tail -4 /var/log/ovn/acl-audit-log.log
  done
```

Example output

```
2021-06-13T19:33:11.590Z|00005|acl_log(ovn_pinctrl0)|INFO|name="verify-audit-logging_deny-all", verdict=drop, severity=alert:
icmp,vlan_tci=0x0000,dl_src=0a:58:0a:80:02:39,dl_dst=0a:58:0a:80:02:37,nw_src=10.128.2.57,
nw_dst=10.128.2.55,nw_tos=0,nw_ecn=0,nw_ttl=64,icmp_type=8,icmp_code=0
```

11.2.5. Disabling network policy audit logging for a namespace

As a cluster administrator, you can disable network policy audit logging for a namespace.

Prerequisites

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

Procedure

- To disable network policy audit logging for a namespace, enter the following command:

```
$ annotate --overwrite namespace <namespace> k8s.ovn.org/acl-logging={}
```

where:

<namespace>

Specifies the name of the namespace.

TIP

You can alternatively apply the following YAML to disable audit logging:

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

Example output

```
namespace/verify-audit-logging annotated
```

11.2.6. Additional resources

- About network policy

11.3. CREATING A NETWORK POLICY

As a user with the **admin** role, you can create a network policy for a namespace.

11.3.1. Creating a network policy

To define granular rules describing ingress or egress network traffic allowed for namespaces in your cluster, you can create a network policy.

NOTE

If you log in with a user with the **cluster-admin** role, then you can create a network policy in any namespace in the cluster.

Prerequisites

- Your cluster uses a cluster network provider that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network provider or the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.
- You are working in the namespace that the network policy applies to.
- Your cluster is using a cluster network provider that supports **NetworkPolicy** objects, such as the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.

Procedure

1. Create a policy rule:

   a. Create a **<policy_name>.yaml** file:

   ```
   $ touch <policy_name>.yaml
   ```

   where:

   <policy_name>
   Specifies the network policy file name.

   b. Define a network policy in the file that you just created, such as in the following examples:

   Deny ingress from all pods in all namespaces

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

   .Allow ingress from all pods in the same namespace

   ```
   kind: NetworkPolicy
   apiVersion: networking.k8s.io/v1
   metadata:
     name: allow-same-namespace
   spec:
     podSelector:
     ingress:
     - from:
       - podSelector: {}
   ```

2. To create the network policy object, enter the following command:

   ```
   $ oc apply -f <policy_name>.yaml -n <namespace>
   ```

   where:

   <policy_name>
   Specifies the network policy file name.

   <namespace>
   Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

   Example output

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

11.3.2. Example NetworkPolicy object

The following annotates an example NetworkPolicy object:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-27107 (1)
spec:
  podSelector: (2)
    matchLabels:
      app: mongodb
  ingress:
  - from:
    - podSelector: (3)
        matchLabels:
          app: app
    ports: (4)
    - protocol: TCP
      port: 27017
```

1. The **name** of the NetworkPolicy object.
2. A selector describing the pods the policy applies to. The policy object can only select pods in the project that the NetworkPolicy object is defined.
3. A selector matching the pods that the policy object allows ingress traffic from. The selector will match pods in any project.
4. A list of one or more destination ports to accept traffic on.

11.4. VIEWING A NETWORK POLICY

As a user with the **admin** role, you can view a network policy for a namespace.

11.4.1. Viewing network policies

You can examine the network policies in a namespace.

> **NOTE**
>
> If you log in with a user with the **cluster-admin** role, then you can view any network policy in the cluster.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.
- You are working in the namespace where the network policy exists.

Procedure

- List network policies in a namespace:
  - To view network policy objects defined in a namespace, enter the following command:

    ```
    $ oc get networkpolicy
    ```

  - Optional: To examine a specific network policy, enter the following command:

    ```
    $ oc describe networkpolicy <policy_name> -n <namespace>
    ```

    where:

    <policy_name>
    Specifies the name of the network policy to inspect.

    <namespace>
    Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

    For example:

    ```
    $ oc describe networkpolicy allow-same-namespace
    ```

    Output for oc describe command

    ```
    Name:         allow-same-namespace
    Namespace:    ns1
    Created on:   2021-05-24 22:28:56 -0400 EDT
    Labels:       <none>
    Annotations:  <none>
    Spec:
      PodSelector:     <none> (Allowing the specific traffic to all pods in this namespace)
      Allowing ingress traffic:
        To Port: <any> (traffic allowed to all ports)
        From:
          PodSelector: <none>
      Not affecting egress traffic
      Policy Types: Ingress
    ```

11.4.2. Example NetworkPolicy object

The following annotates an example NetworkPolicy object:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-27107 1
spec:
  podSelector: 2
    matchLabels:
      app: mongodb
  ingress:
  - from:
```

```
    - podSelector: 3
        matchLabels:
          app: app
    ports: 4
    - protocol: TCP
      port: 27017
```

1. The **name** of the NetworkPolicy object.
2. A selector describing the pods the policy applies to. The policy object can only select pods in the project that the NetworkPolicy object is defined.
3. A selector matching the pods that the policy object allows ingress traffic from. The selector will match pods in any project.
4. A list of one or more destination ports to accept traffic on.

11.5. EDITING A NETWORK POLICY

As a user with the **admin** role, you can edit an existing network policy for a namespace.

11.5.1. Editing a network policy

You can edit a network policy in a namespace.

> **NOTE**
>
> If you log in with a user with the **cluster-admin** role, then you can edit a network policy in any namespace in the cluster.

Prerequisites

- Your cluster uses a cluster network provider that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network provider or the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.
- You are working in the namespace where the network policy exists.

Procedure

1. Optional: To list the network policy objects in a namespace, enter the following command:

   ```
   $ oc get networkpolicy
   ```

   where:

   <namespace>

   Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

2. Edit the network policy object.

   - If you saved the network policy definition in a file, edit the file and make any necessary changes, and then enter the following command.

     ```
     $ oc apply -n <namespace> -f <policy_file>.yaml
     ```

     where:

     <namespace>
     : Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

     <policy_file>
     : Specifies the name of the file containing the network policy.

   - If you need to update the network policy object directly, enter the following command:

     ```
     $ oc edit networkpolicy <policy_name> -n <namespace>
     ```

     where:

     <policy_name>
     : Specifies the name of the network policy.

     <namespace>
     : Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

3. Confirm that the network policy object is updated.

   ```
   $ oc describe networkpolicy <policy_name> -n <namespace>
   ```

   where:

   <policy_name>
   : Specifies the name of the network policy.

   <namespace>
   : Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

11.5.2. Example NetworkPolicy object

The following annotates an example NetworkPolicy object:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-27107 1
spec:
  podSelector: 2
    matchLabels:
      app: mongodb
  ingress:
```

```
- from:
  - podSelector: (3)
      matchLabels:
        app: app
  ports: (4)
  - protocol: TCP
    port: 27017
```

1. The **name** of the NetworkPolicy object.
2. A selector describing the pods the policy applies to. The policy object can only select pods in the project that the NetworkPolicy object is defined.
3. A selector matching the pods that the policy object allows ingress traffic from. The selector will match pods in any project.
4. A list of one or more destination ports to accept traffic on.

11.5.3. Additional resources

- Creating a network policy

11.6. DELETING A NETWORK POLICY

As a user with the **admin** role, you can delete a network policy from a namespace.

11.6.1. Deleting a network policy

You can delete a network policy in a namespace.

> **NOTE**
>
> If you log in with a user with the **cluster-admin** role, then you can delete any network policy in the cluster.

Prerequisites

- Your cluster uses a cluster network provider that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network provider or the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.
- You are working in the namespace where the network policy exists.

Procedure

- To delete a network policy object, enter the following command:

```
$ oc delete networkpolicy <policy_name> -n <namespace>
```

where:

<policy_name>
Specifies the name of the network policy.

<namespace>
Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

Example output

```
networkpolicy.networking.k8s.io/default-deny deleted
```

11.7. DEFINING A DEFAULT NETWORK POLICY FOR PROJECTS

As a cluster administrator, you can modify the new project template to automatically include network policies when you create a new project. If you do not yet have a customized template for new projects, you must first create one.

11.7.1. Modifying the template for new projects

As a cluster administrator, you can modify the default project template so that new projects are created using your custom requirements.

To create your own custom project template:

Procedure

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

2. Generate the default project template:

```
$ oc adm create-bootstrap-project-template -o yaml > template.yaml
```

3. Use a text editor to modify the generated **template.yaml** file by adding objects or modifying existing objects.

4. The project template must be created in the **openshift-config** namespace. Load your modified template:

```
$ oc create -f template.yaml -n openshift-config
```

5. Edit the project configuration resource using the web console or CLI.

   - Using the web console:

     i. Navigate to the **Administration** → **Cluster Settings** page.

     ii. Click **Global Configuration** to view all configuration resources.

     iii. Find the entry for **Project** and click **Edit YAML**.

   - Using the CLI:

i. Edit the **project.config.openshift.io/cluster** resource:

```
$ oc edit project.config.openshift.io/cluster
```

6. Update the **spec** section to include the **projectRequestTemplate** and **name** parameters, and set the name of your uploaded project template. The default name is **project-request**.

Project configuration resource with custom project template

```
apiVersion: config.openshift.io/v1
kind: Project
metadata:
  ...
spec:
  projectRequestTemplate:
    name: <template_name>
```

7. After you save your changes, create a new project to verify that your changes were successfully applied.

11.7.2. Adding network policies to the new project template

As a cluster administrator, you can add network policies to the default template for new projects. OpenShift Container Platform will automatically create all the **NetworkPolicy** objects specified in the template in the project.

Prerequisites

- Your cluster uses a default CNI network provider that supports **NetworkPolicy** objects, such as the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You must log in to the cluster with a user with **cluster-admin** privileges.
- You must have created a custom default project template for new projects.

Procedure

1. Edit the default template for a new project by running the following command:

```
$ oc edit template <project_template> -n openshift-config
```

Replace **<project_template>** with the name of the default template that you configured for your cluster. The default template name is **project-request**.

2. In the template, add each **NetworkPolicy** object as an element to the **objects** parameter. The **objects** parameter accepts a collection of one or more objects.
In the following example, the **objects** parameter collection includes several **NetworkPolicy** objects.

> **IMPORTANT**
>
> For the OVN-Kubernetes network provider plug-in, when the Ingress Controller is configured to use the **HostNetwork** endpoint publishing strategy, there is no supported way to apply network policy so that ingress traffic is allowed and all other traffic is denied.

```
objects:
- apiVersion: networking.k8s.io/v1
  kind: NetworkPolicy
  metadata:
    name: allow-from-same-namespace
  spec:
    podSelector:
    ingress:
    - from:
      - podSelector: {}
- apiVersion: networking.k8s.io/v1
  kind: NetworkPolicy
  metadata:
    name: allow-from-openshift-ingress
  spec:
    ingress:
    - from:
      - namespaceSelector:
          matchLabels:
            network.openshift.io/policy-group: ingress
    podSelector: {}
    policyTypes:
    - Ingress
...
```

3. Optional: Create a new project to confirm that your network policy objects are created successfully by running the following commands:

a. Create a new project:

```
$ oc new-project <project> 1
```

1 Replace **<project>** with the name for the project you are creating.

b. Confirm that the network policy objects in the new project template exist in the new project:

```
$ oc get networkpolicy
NAME                           POD-SELECTOR   AGE
allow-from-openshift-ingress   <none>         7s
allow-from-same-namespace      <none>         7s
```

11.8. CONFIGURING MULTITENANT ISOLATION WITH NETWORK POLICY

As a cluster administrator, you can configure your network policies to provide multitenant network isolation.

> **NOTE**
>
> If you are using the OpenShift SDN cluster network provider, configuring network policies as described in this section provides network isolation similar to multitenant mode but with network policy mode set.

11.8.1. Configuring multitenant isolation by using network policy

You can configure your project to isolate it from pods and services in other project namespaces.

Prerequisites

- Your cluster uses a cluster network provider that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network provider or the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **admin** privileges.

Procedure

1. Create the following **NetworkPolicy** objects:
    a. A policy named **allow-from-openshift-ingress**.

```
$ cat << EOF| oc create -f -
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-ingress
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          network.openshift.io/policy-group: ingress
  podSelector: {}
  policyTypes:
  - Ingress
EOF
```

    b. A policy named **allow-from-openshift-monitoring**:

```
$ cat << EOF| oc create -f -
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-monitoring
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          network.openshift.io/policy-group: monitoring
```

```
  podSelector: {}
  policyTypes:
  - Ingress
EOF
```

c. A policy named **allow-same-namespace**:

```
$ cat << EOF| oc create -f -
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-same-namespace
spec:
  podSelector:
  ingress:
  - from:
    - podSelector: {}
EOF
```

2. Optional: To confirm that the network policies exist in your current project, enter the following command:

```
$ oc describe networkpolicy
```

Example output

```
Name:         allow-from-openshift-ingress
Namespace:    example1
Created on:   2020-06-09 00:28:17 -0400 EDT
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     <none> (Allowing the specific traffic to all pods in this namespace)
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      NamespaceSelector: network.openshift.io/policy-group: ingress
  Not affecting egress traffic
  Policy Types: Ingress


Name:         allow-from-openshift-monitoring
Namespace:    example1
Created on:   2020-06-09 00:29:57 -0400 EDT
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     <none> (Allowing the specific traffic to all pods in this namespace)
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      NamespaceSelector: network.openshift.io/policy-group: monitoring
  Not affecting egress traffic
  Policy Types: Ingress
```

11.8.2. Next steps

- Defining a default network policy

11.8.3. Additional resources

- OpenShift SDN network isolation modes

CHAPTER 12. MULTIPLE NETWORKS

12.1. UNDERSTANDING MULTIPLE NETWORKS

In Kubernetes, container networking is delegated to networking plug-ins that implement the Container Network Interface (CNI).

OpenShift Container Platform uses the Multus CNI plug-in to allow chaining of CNI plug-ins. During cluster installation, you configure your *default* pod network. The default network handles all ordinary network traffic for the cluster. You can define an *additional network* based on the available CNI plug-ins and attach one or more of these networks to your pods. You can define more than one additional network for your cluster, depending on your needs. This gives you flexibility when you configure pods that deliver network functionality, such as switching or routing.

12.1.1. Usage scenarios for an additional network

You can use an additional network in situations where network isolation is needed, including data plane and control plane separation. Isolating network traffic is useful for the following performance and security reasons:

Performance

You can send traffic on two different planes to manage how much traffic is along each plane.

Security

You can send sensitive traffic onto a network plane that is managed specifically for security considerations, and you can separate private data that must not be shared between tenants or customers.

All of the pods in the cluster still use the cluster-wide default network to maintain connectivity across the cluster. Every pod has an **eth0** interface that is attached to the cluster-wide pod network. You can view the interfaces for a pod by using the **oc exec -it <pod_name> -- ip a** command. If you add additional network interfaces that use Multus CNI, they are named **net1**, **net2**, ..., **netN**.

To attach additional network interfaces to a pod, you must create configurations that define how the interfaces are attached. You specify each interface by using a **NetworkAttachmentDefinition** custom resource (CR). A CNI configuration inside each of these CRs defines how that interface is created.

12.1.2. Additional networks in OpenShift Container Platform

OpenShift Container Platform provides the following CNI plug-ins for creating additional networks in your cluster:

- **bridge**: Creating a bridge-based additional network allows pods on the same host to communicate with each other and the host.
- **host-device**: Configuring a host-device additional network allows pods access to a physical Ethernet network device on the host system.
- **ipvlan**: Configuring an ipvlan-based additional network allows pods on a host to communicate with other hosts and pods on those hosts, similar to a macvlan-based additional network. Unlike a macvlan-based additional network, each pod shares the same MAC address as the parent physical network interface.

- **macvlan**: Creating a macvlan-based additional network allows pods on a host to communicate with other hosts and pods on those hosts by using a physical network interface. Each pod that is attached to a macvlan-based additional network is provided a unique MAC address.
  A macvlan additional network can be configured in two ways:
  - Configuring a macvlan-based additional network with basic customizations
  - Configuring a macvlan-based additional network
- **SR-IOV**: Configuring an SR-IOV based additional network allows pods to attach to a virtual function (VF) interface on SR-IOV capable hardware on the host system.

12.2. ABOUT VIRTUAL ROUTING AND FORWARDING

12.2.1. About virtual routing and forwarding

Virtual routing and forwarding (VRF) devices combined with IP rules provide the ability to create virtual routing and forwarding domains. VRF reduces the number of permissions needed by CNF, and provides increased visibility of the network topology of secondary networks. VRF is used to provide multi-tenancy functionality, for example, where each tenant has its own unique routing tables and requires different default gateways.

Processes can bind a socket to the VRF device. Packets through the binded socket use the routing table associated with the VRF device. An important feature of VRF is that it impacts only OSI model layer 3 traffic and above so L2 tools, such as LLDP, are not affected. This allows higher priority IP rules such as policy based routing to take precedence over the VRF device rules directing specific traffic.

12.2.1.1. Benefits of secondary networks for pods for telecommunications operators

In telecommunications use cases, each CNF can potentially be connected to multiple different networks sharing the same address space. These secondary networks can potentially conflict with the cluster's main network CIDR. Using the CNI VRF plug-in, network functions can be connected to different customers' infrastructure using the same IP address, keeping different customers isolated. IP addresses are overlapped with OpenShift Container Platform IP space. The CNI VRF plug-in also reduces the number of permissions needed by CNF and increases the visibility of network topologies of secondary networks.

12.3. CONFIGURING MULTI-NETWORK POLICY

As a cluster administrator, you can configure network policy for additional networks.

> **NOTE**
>
> You can specify multi-network policy for only macvlan additional networks. Other types of additional networks, such as ipvlan, are not supported.

12.3.1. Differences between multi-network policy and network policy

Although the **MultiNetworkPolicy** API implements the **NetworkPolicy** API, there are several important differences:

- You must use the **MultiNetworkPolicy** API:

```
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
```

- You must use the **multi-networkpolicy** resource name when using the CLI to interact with multi-network policies. For example, you can view a multi-network policy object with the **oc get multi-networkpolicy <name>** command where **<name>** is the name of a multi-network policy.

- You must specify an annotation with the name of the network attachment definition that defines the macvlan additional network:

  ```
  apiVersion: k8s.cni.cncf.io/v1beta1
  kind: MultiNetworkPolicy
  metadata:
    annotations:
      k8s.v1.cni.cncf.io/policy-for: <network_name>
  ```

  where:

  <network_name>
  Specifies the name of a network attachment definition.

12.3.2. Enabling multi-network policy for the cluster

As a cluster administrator, you can enable multi-network policy support on your cluster.

Prerequisites

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

- Log in to the cluster with a user with **cluster-admin** privileges.

Procedure

1. Create the **multinetwork-enable-patch.yaml** file with the following YAML:

   ```
   apiVersion: operator.openshift.io/v1
   kind: Network
   metadata:
     name: cluster
   spec:
     useMultiNetworkPolicy: true
   ```

2. Configure the cluster to enable multi-network policy:

   ```
   $ oc patch network.operator.openshift.io cluster --type=merge --patch-file=multinetwork-enable-patch.yaml
   ```

   #### Example output

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

12.3.3. Working with multi-network policy

As a cluster administrator, you can create, edit, view, and delete multi-network policies.

12.3.3.1. Prerequisites

- You have enabled multi-network policy support for your cluster.

12.3.3.2. Creating a multi-network policy

To define granular rules describing ingress or egress network traffic allowed for namespaces in your cluster, you can create a multi-network policy.

Prerequisites

- Your cluster uses a cluster network provider that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network provider or the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.
- You are working in the namespace that the multi-network policy applies to.

Procedure

1. Create a policy rule:

   a. Create a **<policy_name>.yaml** file:

   ```
   $ touch <policy_name>.yaml
   ```

   where:

   <policy_name>
   Specifies the multi-network policy file name.

   b. Define a multi-network policy in the file that you just created, such as in the following examples:

   Deny ingress from all pods in all namespaces

   ```
   apiVersion: k8s.cni.cncf.io/v1beta1
   kind: MultiNetworkPolicy
   metadata:
     name: deny-by-default
     annotations:
       k8s.v1.cni.cncf.io/policy-for: <network_name>
   spec:
     podSelector:
     ingress: []
   ```

   where

   <network_name>

Specifies the name of a network attachment definition.

Allow ingress from all pods in the same namespace

```
apiVersion: k8s.cni.cncf.io/v1beta1
kind: MultiNetworkPolicy
metadata:
  name: allow-same-namespace
  annotations:
    k8s.v1.cni.cncf.io/policy-for: <network_name>
spec:
  podSelector:
  ingress:
  - from:
    - podSelector: {}
```

where

<network_name>

Specifies the name of a network attachment definition.

2. To create the multi-network policy object, enter the following command:

```
$ oc apply -f <policy_name>.yaml -n <namespace>
```

where:

<policy_name>

Specifies the multi-network policy file name.

<namespace>

Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

Example output

```
multinetworkpolicy.k8s.cni.cncf.io/default-deny created
```

12.3.3.3. Editing a multi-network policy

You can edit a multi-network policy in a namespace.

Prerequisites

- Your cluster uses a cluster network provider that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network provider or the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.
- You are working in the namespace where the multi-network policy exists.

Procedure

1. Optional: To list the multi-network policy objects in a namespace, enter the following command:

```
$ oc get multi-networkpolicy
```

where:

<namespace>
Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

2. Edit the multi-network policy object.

- If you saved the multi-network policy definition in a file, edit the file and make any necessary changes, and then enter the following command.

```
$ oc apply -n <namespace> -f <policy_file>.yaml
```

where:

<namespace>
Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

<policy_file>
Specifies the name of the file containing the network policy.

- If you need to update the multi-network policy object directly, enter the following command:

```
$ oc edit multi-networkpolicy <policy_name> -n <namespace>
```

where:

<policy_name>
Specifies the name of the network policy.

<namespace>
Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

3. Confirm that the multi-network policy object is updated.

```
$ oc describe multi-networkpolicy <policy_name> -n <namespace>
```

where:

<policy_name>
Specifies the name of the multi-network policy.

<namespace>
Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

12.3.3.4. Viewing multi-network policies

You can examine the multi-network policies in a namespace.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.
- You are working in the namespace where the multi-network policy exists.

Procedure

- List multi-network policies in a namespace:
  - To view multi-network policy objects defined in a namespace, enter the following command:

    ```
    $ oc get multi-networkpolicy
    ```

  - Optional: To examine a specific multi-network policy, enter the following command:

    ```
    $ oc describe multi-networkpolicy <policy_name> -n <namespace>
    ```

    where:

    <policy_name>
    : Specifies the name of the multi-network policy to inspect.

    <namespace>
    : Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

12.3.3.5. Deleting a multi-network policy

You can delete a multi-network policy in a namespace.

Prerequisites

- Your cluster uses a cluster network provider that supports **NetworkPolicy** objects, such as the OVN-Kubernetes network provider or the OpenShift SDN network provider with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- You installed the OpenShift CLI (**oc**).
- You are logged in to the cluster with a user with **cluster-admin** privileges.
- You are working in the namespace where the multi-network policy exists.

Procedure

- To delete a multi-network policy object, enter the following command:

  ```
  $ oc delete multi-networkpolicy <policy_name> -n <namespace>
  ```

where:

<policy_name>
Specifies the name of the multi-network policy.

<namespace>
Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

Example output

```
multinetworkpolicy.k8s.cni.cncf.io/default-deny deleted
```

12.3.4. Additional resources

- About network policy
- Understanding multiple networks
- Configuring a macvlan network

12.4. ATTACHING A POD TO AN ADDITIONAL NETWORK

As a cluster user you can attach a pod to an additional network.

12.4.1. Adding a pod to an additional network

You can add a pod to an additional network. The pod continues to send normal cluster-related network traffic over the default network.

When a pod is created additional networks are attached to it. However, if a pod already exists, you cannot attach additional networks to it.

The pod must be in the same namespace as the additional network.

Prerequisites

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

Procedure

1. Add an annotation to the **Pod** object. Only one of the following annotation formats can be used:
   a. To attach an additional network without any customization, add an annotation with the following format. Replace **<network>** with the name of the additional network to associate with the pod:

```
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: <network>[,<network>,...] 1
```

1. To specify more than one additional network, separate each network with a comma. Do not include whitespace between the comma. If you specify the same additional

b. To attach an additional network with customizations, add an annotation with the following format:

```
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: |-
      [
        {
          "name": "<network>", 1
          "namespace": "<namespace>", 2
          "default-route": ["<default-route>"] 3
        }
      ]
```

1. Specify the name of the additional network defined by a **NetworkAttachmentDefinition** object.

2. Specify the namespace where the **NetworkAttachmentDefinition** object is defined.

3. Optional: Specify an override for the default route, such as **192.168.17.1**.

2. To create the pod, enter the following command. Replace **<name>** with the name of the pod.

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

3. Optional: To Confirm that the annotation exists in the **Pod** CR, enter the following command, replacing **<name>** with the name of the pod.

```
$ oc get pod <name> -o yaml
```

In the following example, the **example-pod** pod is attached to the **net1** additional network:

```
$ oc get pod example-pod -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: macvlan-bridge
    k8s.v1.cni.cncf.io/networks-status: |- 1
      [{
          "name": "openshift-sdn",
          "interface": "eth0",
          "ips": [
              "10.128.2.14"
          ],
          "default": true,
          "dns": {}
      },{
          "name": "macvlan-bridge",
          "interface": "net1",
```

```
          "ips": [
              "20.2.2.100"
          ],
          "mac": "22:2f:60:a5:f8:00",
          "dns": {}
      }]
    name: example-pod
    namespace: default
  spec:
    ...
  status:
    ...
```

1. The **k8s.v1.cni.cncf.io/networks-status** parameter is a JSON array of objects. Each object describes the status of an additional network attached to the pod. The annotation value is stored as a plain text value.

12.4.1.1. Specifying pod-specific addressing and routing options

When attaching a pod to an additional network, you may want to specify further properties about that network in a particular pod. This allows you to change some aspects of routing, as well as specify static IP addresses and MAC addresses. To accomplish this, you can use the JSON formatted annotations.

Prerequisites

- The pod must be in the same namespace as the additional network.
- Install the OpenShift CLI (**oc**).
- You must log in to the cluster.

Procedure

To add a pod to an additional network while specifying addressing and/or routing options, complete the following steps:

1. Edit the **Pod** resource definition. If you are editing an existing **Pod** resource, run the following command to edit its definition in the default editor. Replace **<name>** with the name of the **Pod** resource to edit.

```
$ oc edit pod <name>
```

2. In the **Pod** resource definition, add the **k8s.v1.cni.cncf.io/networks** parameter to the pod **metadata** mapping. The **k8s.v1.cni.cncf.io/networks** accepts a JSON string of a list of objects that reference the name of **NetworkAttachmentDefinition** custom resource (CR) names in addition to specifying additional properties.

```
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: '[<network>[,<network>,...]]' (1)
```

1. Replace **<network>** with a JSON object as shown in the following examples. The single quotes are required.

3. In the following example the annotation specifies which network attachment will have the default route, using the **default-route** parameter.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  annotations:
    k8s.v1.cni.cncf.io/networks: '
    {
      "name": "net1"
    },
    {
      "name": "net2", 1
      "default-route": ["192.0.2.1"] 2
    }'
spec:
  containers:
  - name: example-pod
    command: ["/bin/bash", "-c", "sleep 2000000000000"]
    image: centos/tools
```

1. The **name** key is the name of the additional network to associate with the pod.

2. The **default-route** key specifies a value of a gateway for traffic to be routed over if no other routing entry is present in the routing table. If more than one **default-route** key is specified, this will cause the pod to fail to become active.

The default route will cause any traffic that is not specified in other routes to be routed to the gateway.

IMPORTANT

Setting the default route to an interface other than the default network interface for OpenShift Container Platform may cause traffic that is anticipated for pod-to-pod traffic to be routed over another interface.

To verify the routing properties of a pod, the **oc** command may be used to execute the **ip** command within a pod.

```
$ oc exec -it <pod_name> -- ip route
```

NOTE

You may also reference the pod's **k8s.v1.cni.cncf.io/networks-status** to see which additional network has been assigned the default route, by the presence of the **default-route** key in the JSON-formatted list of objects.

To set a static IP address or MAC address for a pod you can use the JSON formatted annotations. This requires you create networks that specifically allow for this functionality. This can be specified in a rawCNIConfig for the CNO.

1. Edit the CNO CR by running the following command:

```
$ oc edit networks.operator.openshift.io cluster
```

The following YAML describes the configuration parameters for the CNO:

Cluster Network Operator YAML configuration

```
name: <name> 1
namespace: <namespace> 2
rawCNIConfig: '{ 3
  ...
}'
type: Raw
```

1. Specify a name for the additional network attachment that you are creating. The name must be unique within the specified **namespace**.
2. Specify the namespace to create the network attachment in. If you do not specify a value, then the **default** namespace is used.
3. Specify the CNI plug-in configuration in JSON format, which is based on the following template.

The following object describes the configuration parameters for utilizing static MAC address and IP address using the macvlan CNI plug-in:

macvlan CNI plug-in JSON configuration object using static IP and MAC address

```
{
  "cniVersion": "0.3.1",
  "name": "<name>", 1
  "plugins": [{ 2
     "type": "macvlan",
     "capabilities": { "ips": true }, 3
     "master": "eth0", 4
     "mode": "bridge",
     "ipam": {
       "type": "static"
     }
   }, {
     "capabilities": { "mac": true }, 5
     "type": "tuning"
   }]
}
```

1. Specifies the name for the additional network attachment to create. The name must be unique within the specified **namespace**.
2. Specifies an array of CNI plug-in configurations. The first object specifies a macvlan plug-in configuration and the second object specifies a tuning plug-in configuration.
3. Specifies that a request is made to enable the static IP address functionality of the CNI plug-in runtime configuration capabilities.
4. Specifies the interface that the macvlan plug-in uses.

5 Specifies that a request is made to enable the static MAC address functionality of a CNI plug-in.

The above network attachment can be referenced in a JSON formatted annotation, along with keys to specify which static IP and MAC address will be assigned to a given pod.

Edit the pod with:

```
$ oc edit pod <name>
```

macvlan CNI plug-in JSON configuration object using static IP and MAC address

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
      {
        "name": "<name>", 1
        "ips": [ "192.0.2.205/24" ], 2
        "mac": "CA:FE:C0:FF:EE:00" 3
      }
    ]'
```

1 Use the **<name>** as provided when creating the **rawCNIConfig** above.

2 Provide an IP address including the subnet mask.

3 Provide the MAC address.

> **NOTE**
>
> Static IP addresses and MAC addresses do not have to be used at the same time, you may use them individually, or together.

To verify the IP address and MAC properties of a pod with additional networks, use the **oc** command to execute the ip command within a pod.

```
$ oc exec -it <pod_name> -- ip a
```

12.5. REMOVING A POD FROM AN ADDITIONAL NETWORK

As a cluster user you can remove a pod from an additional network.

12.5.1. Removing a pod from an additional network

You can remove a pod from an additional network only by deleting the pod.

Prerequisites

- An additional network is attached to the pod.
- Install the OpenShift CLI (**oc**).
- Log in to the cluster.

Procedure

- To delete the pod, enter the following command:

```
$ oc delete pod <name> -n <namespace>
```

  - **<name>** is the name of the pod.
  - **<namespace>** is the namespace that contains the pod.

12.6. CONFIGURING A BRIDGE NETWORK

As a cluster administrator, you can configure an additional network for your cluster using the bridge Container Network Interface (CNI) plug-in. When configured, all Pods on a node are connected to a virtual switch. Each pod is assigned an IP address on the additional network.

12.6.1. Creating an additional network attachment with the bridge CNI plug-in

The Cluster Network Operator (CNO) manages additional network definitions. When you specify an additional network to create, the CNO creates the **NetworkAttachmentDefinition** object automatically.

> **IMPORTANT**
>
> Do not edit the **NetworkAttachmentDefinition** objects that the Cluster Network Operator manages. Doing so might disrupt network traffic on your additional network.

Prerequisites

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

Procedure

To create an additional network for your cluster, complete the following steps:

1. Edit the CNO CR by running the following command:

```
$ oc edit networks.operator.openshift.io cluster
```

2. Modify the CR that you are creating by adding the configuration for the additional network you are creating, as in the following example CR.
   The following YAML configures the bridge CNI plug-in:

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
```

```
spec:
  additionalNetworks: (1)
  - name: test-network-1
    namespace: test-1
    type: Raw
    rawCNIConfig: '{
      "cniVersion": "0.3.1",
      "name": "test-network-1",
      "type": "bridge",
      "ipam": {
        "type": "static",
        "addresses": [
          {
            "address": "192.168.1.23/24"
          }
        ]
      }
    }'
```

1 Specify the configuration for the additional network attachment definition.

3. Save your changes and quit the text editor to commit your changes.

4. Confirm that the CNO created the NetworkAttachmentDefinition object by running the following command. Replace **<namespace>** with the namespace that you specified when configuring the network attachment. There might be a delay before the CNO creates the object.

```
$ oc get network-attachment-definitions -n <namespace>
```

Example output

```
NAME             AGE
test-network-1   14m
```

12.6.1.1. Configuration for bridge

The configuration for an additional network attachment that uses the bridge Container Network Interface (CNI) plug-in is provided in two parts:

- Cluster Network Operator (CNO) configuration
- CNI plug-in configuration

The CNO configuration specifies the name for the additional network attachment and the namespace to create the attachment in. The plug-in is configured by a JSON object specified by the **rawCNIConfig** parameter in the CNO configuration.

The following YAML describes the configuration parameters for the CNO:

Cluster Network Operator YAML configuration

```
name: <name> (1)
```

```
namespace: <namespace> 2
rawCNIConfig: '{ 3
  ...
}'
type: Raw
```

1. Specify a name for the additional network attachment that you are creating. The name must be unique within the specified **namespace**.

2. Specify the namespace to create the network attachment in. If you do not specify a value, then the **default** namespace is used.

3. Specify the CNI plug-in configuration in JSON format, which is based on the following template.

The following object describes the configuration parameters for the bridge CNI plug-in:

bridge CNI plug-in JSON configuration object

```
{
  "cniVersion": "0.3.1",
  "name": "<name>", 1
  "type": "bridge",
  "bridge": "<bridge>", 2
  "ipam": { 3
    ...
  },
  "ipMasq": false, 4
  "isGateway": false, 5
  "isDefaultGateway": false, 6
  "forceAddress": false, 7
  "hairpinMode": false, 8
  "promiscMode": false, 9
  "vlan": <vlan>, 10
  "mtu": <mtu> 11
}
```

1. Specify the value for the **name** parameter you provided previously for the CNO configuration.

2. Specify the name of the virtual bridge to use. If the bridge interface does not exist on the host, it is created. The default value is **cni0**.

3. Specify a configuration object for the ipam CNI plug-in. The plug-in manages IP address assignment for the network attachment definition.

4. Set to **true** to enable IP masquerading for traffic that leaves the virtual network. The source IP address for all traffic is rewritten to the bridge's IP address. If the bridge does not have an IP address, this setting has no effect. The default value is **false**.

5. Set to **true** to assign an IP address to the bridge. The default value is **false**.

6. Set to **true** to configure the bridge as the default gateway for the virtual network. The default value is **false**. If **isDefaultGateway** is set to **true**, then **isGateway** is also set to **true** automatically.

7 Set to **true** to allow assignment of a previously assigned IP address to the virtual bridge. When set to **false**, if an IPv4 address or an IPv6 address from overlapping subsets is assigned to the virtual bridge, an error occurs. The default value is **false**.

8 Set to **true** to allow the virtual bridge to send an ethernet frame back through the virtual port it was received on. This mode is also known as *reflective relay*. The default value is **false**.

9 Set to **true** to enable promiscuous mode on the bridge. The default value is **false**.

10 Specify a virtual LAN (VLAN) tag as an integer value. By default, no VLAN tag is assigned.

11 Set the maximum transmission unit (MTU) to the specified value. The default value is automatically set by the kernel.

12.6.1.1.1. bridge configuration example

The following example configures an additional network named **bridge-net**:

```
name: bridge-net
namespace: work-network
type: Raw
rawCNIConfig: '{ 1
  "cniVersion": "0.3.1",
  "name": "work-network",
  "type": "bridge",
  "isGateway": true,
  "vlan": 2,
  "ipam": {
    "type": "dhcp"
    }
}'
```

1 The CNI configuration object is specified as a YAML string.

12.6.1.2. Configuration for ipam CNI plug-in

The ipam Container Network Interface (CNI) plug-in provides IP address management (IPAM) for other CNI plug-ins.

You can use the following methods for IP address assignment:

- Static assignment.
- Dynamic assignment through a DHCP server. The DHCP server you specify must be reachable from the additional network.
- Dynamic assignment through the Whereabouts IPAM CNI plug-in.

12.6.1.2.1. Static IP address assignment configuration

The following JSON describes the configuration for static IP address assignment:

Static assignment configuration

```
{
  "ipam": {
    "type": "static",
    "addresses": [ 1
      {
        "address": "<address>", 2
        "gateway": "<gateway>" 3
      }
    ],
    "routes": [ 4
      {
        "dst": "<dst>", 5
        "gw": "<gw>" 6
      }
    ],
    "dns": { 7
      "nameservers": ["<nameserver>"], 8
      "domain": "<domain>", 9
      "search": ["<search_domain>"] 10
    }
  }
}
```

1 An array describing IP addresses to assign to the virtual interface. Both IPv4 and IPv6 IP addresses are supported.

2 An IP address and network prefix that you specify. For example, if you specify **10.10.21.10/24**, then the additional network is assigned an IP address of **10.10.21.10** and the netmask is **255.255.255.0**.

3 The default gateway to route egress network traffic to.

4 An array describing routes to configure inside the pod.

5 The IP address range in CIDR format, such as **192.168.17.0/24**, or **0.0.0.0/0** for the default route.

6 The gateway where network traffic is routed.

7 Optional: DNS configuration.

8 An of array of one or more IP addresses for to send DNS queries to.

9 The default domain to append to a hostname. For example, if the domain is set to **example.com**, a DNS lookup query for **example-host** is rewritten as **example-host.example.com**.

10 An array of domain names to append to an unqualified hostname, such as **example-host**, during a DNS lookup query.

12.6.1.2.2. Dynamic IP address assignment configuration

The following JSON describes the configuration for dynamic IP address address assignment with DHCP.

RENEWAL OF DHCP LEASES

A pod obtains its original DHCP lease when it is created. The lease must be periodically renewed by a minimal DHCP server deployment running on the cluster.

To trigger the deployment of the DHCP server, you must create a shim network attachment by editing the Cluster Network Operator configuration, as in the following example:

Example shim network attachment definition

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  ...
  additionalNetworks:
  - name: dhcp-shim
    namespace: default
    type: Raw
    rawCNIConfig: |-
      {
        "name": "dhcp-shim",
        "cniVersion": "0.3.1",
        "type": "bridge",
        "ipam": {
          "type": "dhcp"
        }
      }
```

DHCP assignment configuration

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

12.6.1.2.3. Dynamic IP address assignment configuration with Whereabouts

The Whereabouts CNI plug-in allows the dynamic assignment of an IP address to an additional network without the use of a DHCP server.

The following JSON describes the configuration for dynamic IP address assignment with Whereabouts:

Whereabouts assignment configuration

```
{
  "ipam": {
    "type": "whereabouts",
    "range": "<range>", 1
```

```
    "exclude": ["<exclude_part>, ..."], 2
  }
}
```

1. Specify an IP address and range in CIDR notation. IP addresses are assigned from within this range of addresses.

2. Optional: Specify a list of IP addresses and ranges in CIDR notation. IP addresses within an excluded address range are not assigned.

12.6.1.2.4. Static IP address assignment configuration example

You can configure ipam for static IP address assignment:

```
{
  "ipam": {
    "type": "static",
      "addresses": [
        {
          "address": "191.168.1.7"
        }
      ]
  }
}
```

12.6.1.2.5. Dynamic IP address assignment configuration example using DHCP

You can configure ipam for DHCP:

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

12.6.1.2.6. Dynamic IP address assignment configuration example using Whereabouts

You can configure ipam to use Whereabouts:

```
{
  "ipam": {
    "type": "whereabouts",
    "range": "192.0.2.192/27",
    "exclude": [
       "192.0.2.192/30",
       "192.0.2.196/32"
    ]
  }
}
```

12.6.2. Next steps

- Attach a pod to an additional network .

12.7. CONFIGURING A HOST-DEVICE NETWORK

As a cluster administrator, you can configure an additional network for your cluster by using the host-device Container Network Interface (CNI) plug-in. The plug-in moves the specified network device from the network namespace of the host into the network namespace of the pod.

12.7.1. Creating an additional network attachment with the host-device CNI plug-in

The Cluster Network Operator (CNO) manages additional network definitions. When you specify an additional network to create, the CNO creates the **NetworkAttachmentDefinition** object automatically.

> **IMPORTANT**
>
> Do not edit the **NetworkAttachmentDefinition** objects that the Cluster Network Operator manages. Doing so might disrupt network traffic on your additional network.

Prerequisites

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

Procedure

To create an additional network for your cluster, complete the following steps:

1. Edit the CNO CR by running the following command:

```
$ oc edit networks.operator.openshift.io cluster
```

2. Modify the CR that you are creating by adding the configuration for the additional network you are creating, as in the following example CR.
The following YAML configures the host-device CNI plug-in:

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  additionalNetworks: 1
  - name: test-network-1
    namespace: test-1
    type: Raw
    rawCNIConfig: '{
      "cniVersion": "0.3.1",
      "name": "test-network-1",
      "type": "host-device",
      "device": "eth1",
      "ipam": {
        "type": "static",
        "addresses": [
          {
```

```
            "address": "192.168.1.23/24"
          }
        ]
      }
    }'
```

1. Specify the configuration for the additional network attachment definition.

3. Save your changes and quit the text editor to commit your changes.

4. Confirm that the CNO created the NetworkAttachmentDefinition object by running the following command. Replace **<namespace>** with the namespace that you specified when configuring the network attachment. There might be a delay before the CNO creates the object.

```
$ oc get network-attachment-definitions -n <namespace>
```

Example output

```
NAME            AGE
test-network-1   14m
```

12.7.1.1. Configuration for host-device

The configuration for an additional network attachment that uses the host-device Container Network Interface (CNI) plug-in is provided in two parts:

- Cluster Network Operator (CNO) configuration
- CNI plug-in configuration

The CNO configuration specifies the name for the additional network attachment and the namespace to create the attachment in. The plug-in is configured by a JSON object specified by the **rawCNIConfig** parameter in the CNO configuration.

The following YAML describes the configuration parameters for the CNO:

Cluster Network Operator YAML configuration

```
name: <name> 1
namespace: <namespace> 2
rawCNIConfig: '{ 3
  ...
}'
type: Raw
```

1. Specify a name for the additional network attachment that you are creating. The name must be unique within the specified **namespace**.

2. Specify the namespace to create the network attachment in. If you do not specify a value, the **default** namespace is used.

3. Specify the CNI plug-in configuration in JSON format, which is based on the following template.

> **IMPORTANT**
>
> Specify your network device by setting only one of the following parameters: **device**, **hwaddr**, **kernelpath**, or **pciBusID**.

The following object describes the configuration parameters for the host-device CNI plug-in:

host-device CNI plug-in JSON configuration object

```
{
  "cniVersion": "0.3.1",
  "name": "<name>", (1)
  "type": "host-device",
  "device": "<device>", (2)
  "hwaddr": "<hwaddr>", (3)
  "kernelpath": "<kernelpath>", (4)
  "pciBusID": "<pciBusID>", (5)
  "ipam": { (6)
    ...
  }
}
```

1. Specify the value for the **name** parameter you provided previously for the CNO configuration.
2. Specify the name of the device, such as **eth0**.
3. Specify the device hardware MAC address.
4. Specify the Linux kernel device path, such as **/sys/devices/pci0000:00/0000:00:1f.6**.
5. Specify the PCI address of the network device, such as **0000:00:1f.6**.
6. Specify a configuration object for the ipam CNI plug-in. The plug-in manages IP address assignment for the attachment definition.

12.7.1.1.1. host-device configuration example

The following example configures an additional network named **hostdev-net**:

```
name: hostdev-net
namespace: work-network
type: Raw
rawCNIConfig: '{ (1)
  "cniVersion": "0.3.1",
  "name": "work-network",
  "type": "host-device",
  "device": "eth1",
  "ipam": {
    "type": "dhcp"
  }
}'
```

1. The CNI configuration object is specified as a YAML string.

12.7.1.2. Configuration for ipam CNI plug-in

The ipam Container Network Interface (CNI) plug-in provides IP address management (IPAM) for other CNI plug-ins.

You can use the following methods for IP address assignment:

- Static assignment.
- Dynamic assignment through a DHCP server. The DHCP server you specify must be reachable from the additional network.
- Dynamic assignment through the Whereabouts IPAM CNI plug-in.

12.7.1.2.1. Static IP address assignment configuration

The following JSON describes the configuration for static IP address assignment:

Static assignment configuration

```
{
  "ipam": {
    "type": "static",
    "addresses": [ 1
      {
        "address": "<address>", 2
        "gateway": "<gateway>" 3
      }
    ],
    "routes": [ 4
      {
        "dst": "<dst>", 5
        "gw": "<gw>" 6
      }
    ],
    "dns": { 7
      "nameservers": ["<nameserver>"], 8
      "domain": "<domain>", 9
      "search": ["<search_domain>"] 10
    }
  }
}
```

1. An array describing IP addresses to assign to the virtual interface. Both IPv4 and IPv6 IP addresses are supported.
2. An IP address and network prefix that you specify. For example, if you specify **10.10.21.10/24**, then the additional network is assigned an IP address of **10.10.21.10** and the netmask is **255.255.255.0**.
3. The default gateway to route egress network traffic to.
4. An array describing routes to configure inside the pod.
5. The IP address range in CIDR format, such as **192.168.17.0/24**, or **0.0.0.0/0** for the default route.

6. The gateway where network traffic is routed.
7. Optional: DNS configuration.
8. An of array of one or more IP addresses for to send DNS queries to.
9. The default domain to append to a hostname. For example, if the domain is set to **example.com**, a DNS lookup query for **example-host** is rewritten as **example-host.example.com**.
10. An array of domain names to append to an unqualified hostname, such as **example-host**, during a DNS lookup query.

12.7.1.2.2. Dynamic IP address assignment configuration

The following JSON describes the configuration for dynamic IP address address assignment with DHCP.

> **RENEWAL OF DHCP LEASES**
>
> A pod obtains its original DHCP lease when it is created. The lease must be periodically renewed by a minimal DHCP server deployment running on the cluster.
>
> To trigger the deployment of the DHCP server, you must create a shim network attachment by editing the Cluster Network Operator configuration, as in the following example:
>
> **Example shim network attachment definition**
>
> ```
> apiVersion: operator.openshift.io/v1
> kind: Network
> metadata:
>   name: cluster
> spec:
>   ...
>   additionalNetworks:
>   - name: dhcp-shim
>     namespace: default
>     type: Raw
>     rawCNIConfig: |-
>       {
>         "name": "dhcp-shim",
>         "cniVersion": "0.3.1",
>         "type": "bridge",
>         "ipam": {
>           "type": "dhcp"
>         }
>       }
> ```

DHCP assignment configuration

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

12.7.1.2.3. Dynamic IP address assignment configuration with Whereabouts

The Whereabouts CNI plug-in allows the dynamic assignment of an IP address to an additional network without the use of a DHCP server.

The following JSON describes the configuration for dynamic IP address assignment with Whereabouts:

Whereabouts assignment configuration

```
{
  "ipam": {
    "type": "whereabouts",
    "range": "<range>", 1
    "exclude": ["<exclude_part>, ..."], 2
  }
}
```

1. Specify an IP address and range in CIDR notation. IP addresses are assigned from within this range of addresses.

2. Optional: Specify a list of IP addresses and ranges in CIDR notation. IP addresses within an excluded address range are not assigned.

12.7.1.2.4. Static IP address assignment configuration example

You can configure ipam for static IP address assignment:

```
{
  "ipam": {
    "type": "static",
      "addresses": [
        {
          "address": "191.168.1.7"
        }
      ]
  }
}
```

12.7.1.2.5. Dynamic IP address assignment configuration example using DHCP

You can configure ipam for DHCP:

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

12.7.1.2.6. Dynamic IP address assignment configuration example using Whereabouts

You can configure ipam to use Whereabouts:

```
{
  "ipam": {
    "type": "whereabouts",
    "range": "192.0.2.192/27",
    "exclude": [
       "192.0.2.192/30",
       "192.0.2.196/32"
    ]
  }
}
```

12.7.2. Next steps

- Attach a pod to an additional network .

12.8. CONFIGURING AN IPVLAN NETWORK

As a cluster administrator, you can configure an additional network for your cluster by using the ipvlan Container Network Interface (CNI) plug-in. The virtual network created by this plug-in is associated with a physical interface that you specify.

12.8.1. Creating an additional network attachment with the ipvlan CNI plug-in

The Cluster Network Operator (CNO) manages additional network definitions. When you specify an additional network to create, the CNO creates the **NetworkAttachmentDefinition** object automatically.

> **IMPORTANT**
>
> Do not edit the **NetworkAttachmentDefinition** objects that the Cluster Network Operator manages. Doing so might disrupt network traffic on your additional network.

Prerequisites

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

Procedure

To create an additional network for your cluster, complete the following steps:

1. Edit the CNO CR by running the following command:

```
$ oc edit networks.operator.openshift.io cluster
```

2. Modify the CR that you are creating by adding the configuration for the additional network you are creating, as in the following example CR.
The following YAML configures the ipvlan CNI plug-in:

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
```

```
spec:
  additionalNetworks: 1
  - name: test-network-1
    namespace: test-1
    type: Raw
    rawCNIConfig: '{
      "cniVersion": "0.3.1",
      "name": "test-network-1",
      "type": "ipvlan",
      "master": "eth1",
      "mode": "l2",
      "ipam": {
        "type": "static",
        "addresses": [
          {
            "address": "192.168.1.23/24"
          }
        ]
      }
    }'
```

1. Specify the configuration for the additional network attachment definition.

3. Save your changes and quit the text editor to commit your changes.

4. Confirm that the CNO created the NetworkAttachmentDefinition object by running the following command. Replace **<namespace>** with the namespace that you specified when configuring the network attachment. There might be a delay before the CNO creates the object.

```
$ oc get network-attachment-definitions -n <namespace>
```

Example output

```
NAME             AGE
test-network-1   14m
```

12.8.1.1. Configuration for ipvlan

The configuration for an additional network attachment that uses the ipvlan Container Network Interface (CNI) plug-in is provided in two parts:

- Cluster Network Operator (CNO) configuration
- CNI plug-in configuration

The CNO configuration specifies the name for the additional network attachment and the namespace to create the attachment in. The plug-in is configured by a JSON object specified by the **rawCNIConfig** parameter in the CNO configuration.

The following YAML describes the configuration parameters for the CNO:

Cluster Network Operator YAML configuration

```
name: <name> 1
namespace: <namespace> 2
rawCNIConfig: '{ 3
  ...
}'
type: Raw
```

1. Specify a name for the additional network attachment that you are creating. The name must be unique within the specified **namespace**.
2. Specify the namespace to create the network attachment in. If you do not specify a value, then the **default** namespace is used.
3. Specify the CNI plug-in configuration in JSON format, which is based on the following template.

The following object describes the configuration parameters for the ipvlan CNI plug-in:

ipvlan CNI plug-in JSON configuration object

```
{
  "cniVersion": "0.3.1",
  "name": "<name>", 1
  "type": "ipvlan",
  "mode": "<mode>", 2
  "master": "<master>", 3
  "mtu": <mtu>, 4
  "ipam": { 5
    ...
  }
}
```

1. Specify the value for the **name** parameter you provided previously for the CNO configuration.
2. Specify the operating mode for the virtual network. The value must be **l2**, **l3**, or **l3s**. The default value is **l2**.
3. Specify the ethernet interface to associate with the network attachment. If a **master** is not specified, the interface for the default network route is used.
4. Set the maximum transmission unit (MTU) to the specified value. The default value is automatically set by the kernel.
5. Specify a configuration object for the ipam CNI plug-in. The plug-in manages IP address assignment for the attachment definition.

12.8.1.1.1. ipvlan configuration example

The following example configures an additional network named **ipvlan-net**:

```
name: ipvlan-net
namespace: work-network
type: Raw
```

```
rawCNIConfig: '{ (1)
  "cniVersion": "0.3.1",
  "name": "work-network",
  "type": "ipvlan",
  "master": "eth1",
  "mode": "l3",
  "ipam": {
    "type": "dhcp"
    }
}'
```

1. The CNI configuration object is specified as a YAML string.

12.8.1.2. Configuration for ipam CNI plug-in

The ipam Container Network Interface (CNI) plug-in provides IP address management (IPAM) for other CNI plug-ins.

You can use the following methods for IP address assignment:

- Static assignment.
- Dynamic assignment through a DHCP server. The DHCP server you specify must be reachable from the additional network.
- Dynamic assignment through the Whereabouts IPAM CNI plug-in.

12.8.1.2.1. Static IP address assignment configuration

The following JSON describes the configuration for static IP address assignment:

Static assignment configuration

```
{
  "ipam": {
    "type": "static",
    "addresses": [ (1)
      {
        "address": "<address>", (2)
        "gateway": "<gateway>" (3)
      }
    ],
    "routes": [ (4)
      {
        "dst": "<dst>", (5)
        "gw": "<gw>" (6)
      }
    ],
    "dns": { (7)
      "nameservers": ["<nameserver>"], (8)
      "domain": "<domain>", (9)
      "search": ["<search_domain>"] (10)
```

```
      }
    }
  }
```

1. An array describing IP addresses to assign to the virtual interface. Both IPv4 and IPv6 IP addresses are supported.
2. An IP address and network prefix that you specify. For example, if you specify **10.10.21.10/24**, then the additional network is assigned an IP address of **10.10.21.10** and the netmask is **255.255.255.0**.
3. The default gateway to route egress network traffic to.
4. An array describing routes to configure inside the pod.
5. The IP address range in CIDR format, such as **192.168.17.0/24**, or **0.0.0.0/0** for the default route.
6. The gateway where network traffic is routed.
7. Optional: DNS configuration.
8. An of array of one or more IP addresses for to send DNS queries to.
9. The default domain to append to a hostname. For example, if the domain is set to **example.com**, a DNS lookup query for **example-host** is rewritten as **example-host.example.com**.
10. An array of domain names to append to an unqualified hostname, such as **example-host**, during a DNS lookup query.

12.8.1.2.2. Dynamic IP address assignment configuration

The following JSON describes the configuration for dynamic IP address address assignment with DHCP.

RENEWAL OF DHCP LEASES

A pod obtains its original DHCP lease when it is created. The lease must be periodically renewed by a minimal DHCP server deployment running on the cluster.

To trigger the deployment of the DHCP server, you must create a shim network attachment by editing the Cluster Network Operator configuration, as in the following example:

Example shim network attachment definition

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  ...
  additionalNetworks:
  - name: dhcp-shim
    namespace: default
    type: Raw
    rawCNIConfig: |-
      {
        "name": "dhcp-shim",
        "cniVersion": "0.3.1",
        "type": "bridge",
        "ipam": {
          "type": "dhcp"
        }
      }
```

DHCP assignment configuration

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

12.8.1.2.3. Dynamic IP address assignment configuration with Whereabouts

The Whereabouts CNI plug-in allows the dynamic assignment of an IP address to an additional network without the use of a DHCP server.

The following JSON describes the configuration for dynamic IP address assignment with Whereabouts:

Whereabouts assignment configuration

```
{
  "ipam": {
    "type": "whereabouts",
    "range": "<range>", (1)
```

```
    "exclude": ["<exclude_part>, ..."], (2)
  }
}
```

(1) Specify an IP address and range in CIDR notation. IP addresses are assigned from within this range of addresses.

(2) Optional: Specify a list of IP addresses and ranges in CIDR notation. IP addresses within an excluded address range are not assigned.

12.8.1.2.4. Static IP address assignment configuration example

You can configure ipam for static IP address assignment:

```
{
  "ipam": {
    "type": "static",
      "addresses": [
        {
          "address": "191.168.1.7"
        }
      ]
  }
}
```

12.8.1.2.5. Dynamic IP address assignment configuration example using DHCP

You can configure ipam for DHCP:

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

12.8.1.2.6. Dynamic IP address assignment configuration example using Whereabouts

You can configure ipam to use Whereabouts:

```
{
  "ipam": {
    "type": "whereabouts",
    "range": "192.0.2.192/27",
    "exclude": [
       "192.0.2.192/30",
       "192.0.2.196/32"
    ]
  }
}
```

12.8.2. Next steps

- Attach a pod to an additional network .

12.9. CONFIGURING A MACVLAN NETWORK WITH BASIC CUSTOMIZATIONS

As a cluster administrator, you can configure an additional network for your cluster using the macvlan Container Network Interface (CNI) plug-in. When a pod is attached to the network, the plug-in creates a sub-interface from the parent interface on the host. A unique hardware mac address is generated for each sub-device.

> **IMPORTANT**
>
> The unique MAC addresses this plug-in generates for sub-interfaces might not be compatible with the security polices of your cloud provider.

You specify a basic configuration directly in YAML. This approach offers fewer configuration options than by specifying a macvlan configuration by using a CNI object directly in JSON.

12.9.1. Creating an additional network attachment with the macvlan CNI plug-in

The Cluster Network Operator (CNO) manages additional network definitions. When you specify an additional network to create, the CNO creates the **NetworkAttachmentDefinition** object automatically.

> **IMPORTANT**
>
> Do not edit the **NetworkAttachmentDefinition** objects that the Cluster Network Operator manages. Doing so might disrupt network traffic on your additional network.

Prerequisites

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

Procedure

To create an additional network for your cluster, complete the following steps:

1. Edit the CNO CR by running the following command:

```
$ oc edit networks.operator.openshift.io cluster
```

2. Modify the CR that you are creating by adding the configuration for the additional network you are creating, as in the following example CR.
The following YAML configures the macvlan CNI plug-in:

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  additionalNetworks: 1
  - name: test-network-1
```

```
namespace: test-1
type: SimpleMacvlan
simpleMacvlanConfig:
  ipamConfig:
    type: static
    staticIPAMConfig:
      addresses:
      - address: 10.1.1.7/24
```

1. Specify the configuration for the additional network attachment definition.

3. Save your changes and quit the text editor to commit your changes.

4. Confirm that the CNO created the NetworkAttachmentDefinition object by running the following command. Replace **<namespace>** with the namespace that you specified when configuring the network attachment. There might be a delay before the CNO creates the object.

```
$ oc get network-attachment-definitions -n <namespace>
```

Example output

```
NAME            AGE
test-network-1   14m
```

12.9.1.1. Configuration for macvlan CNI plug-in

The following YAML describes the configuration parameters for the macvlan Container Network Interface (CNI) plug-in:

macvlan YAML configuration

```
name: <name> 1
namespace: <namespace> 2
type: SimpleMacvlan
simpleMacvlanConfig:
  master: <master> 3
  mode: <mode> 4
  mtu: <mtu> 5
  ipamConfig: 6
    ...
```

1. Specify a name for the additional network attachment that you are creating. The name must be unique within the specified **namespace**.

2. Specify the namespace to create the network attachment in. If a value is not specified, the **default** namespace is used.

3. The ethernet, bonded, or VLAN interface to associate with the virtual interface. If a value for **master** is not specified, then the host system's primary ethernet interface is used.

4. Configures traffic visibility on the virtual network. Must be either **bridge**, **passthru**, **private**, or **vepa**. If a value for **mode** is not provided, the default value is **bridge**.

5 Set the maximum transmission unit (MTU) to the specified value. The default value is automatically set by the kernel.

6 Specify a configuration object for the ipam CNI plug-in. The plug-in manages IP address assignment for the attachment definition.

12.9.1.1.1. macvlan configuration example

The following example configures an additional network named **macvlan-net**:

```
name: macvlan-net
namespace: work-network
type: SimpleMacvlan
simpleMacvlanConfig:
  ipamConfig:
    type: DHCP
```

12.9.1.2. Configuration for ipam CNI plug-in

The ipam Container Network Interface (CNI) plug-in provides IP address management (IPAM) for other CNI plug-ins.

The following YAML configuration describes the parameters that you can set.

ipam CNI plug-in YAML configuration object

```
ipamConfig:
  type: <type> 1
  ... 2
```

1 Specify **static** to configure the plug-in to manage IP address assignment. Specify **DHCP** to allow a DHCP server to manage IP address assignment. You cannot specify any additional parameters if you specify a value of **DHCP**.

2 If you set the **type** parameter to **static**, then provide the **staticIPAMConfig** parameter.

12.9.1.2.1. Static ipam configuration YAML

The following YAML describes a configuration for static IP address assignment:

Static ipam configuration YAML

```
ipamConfig:
  type: static
  staticIPAMConfig:
    addresses: 1
    - address: <address> 2
      gateway: <gateway> 3
    routes: 4
    - destination: <destination> 5
      gateway: <gateway> 6
```

```
  dns: 7
    nameservers: 8
    - <nameserver>
    domain: <domain> 9
    search: 10
    - <search_domain>
```

1. A collection of mappings that define IP addresses to assign to the virtual interface. Both IPv4 and IPv6 IP addresses are supported.
2. An IP address and network prefix that you specify. For example, if you specify **10.10.21.10/24**, then the additional network is assigned an IP address of **10.10.21.10** and the netmask is **255.255.255.0**.
3. The default gateway to route egress network traffic to.
4. A collection of mappings describing routes to configure inside the pod.
5. The IP address range in CIDR format, such as **192.168.17.0/24**, or **0.0.0.0/0** for the default route.
6. The gateway where network traffic is routed.
7. Optional: The DNS configuration.
8. A collection of one or more IP addresses for to send DNS queries to.
9. The default domain to append to a hostname. For example, if the domain is set to **example.com**, a DNS lookup query for **example-host** is rewritten as **example-host.example.com**.
10. An array of domain names to append to an unqualified hostname, such as **example-host**, during a DNS lookup query.

12.9.1.2.2. Dynamic ipam configuration YAML

The following YAML describes a configuration for static IP address assignment:

Dynamic ipam configuration YAML

```
ipamConfig:
  type: DHCP
```

12.9.1.2.3. Static IP address assignment configuration example

The following example shows an ipam configuration for static IP addresses:

```
ipamConfig:
  type: static
  staticIPAMConfig:
    addresses:
    - address: 198.51.100.11/24
      gateway: 198.51.100.10
    routes:
    - destination: 0.0.0.0/0
      gateway: 198.51.100.1
    dns:
```

```
    nameservers:
    - 198.51.100.1
    - 198.51.100.2
    domain: testDNS.example
    search:
    - testdomain1.example
    - testdomain2.example
```

12.9.1.2.4. Dynamic IP address assignment configuration example

The following example shows an ipam configuration for DHCP:

```
ipamConfig:
  type: DHCP
```

12.9.2. Next steps

- Attach a pod to an additional network .

12.10. CONFIGURING A MACVLAN NETWORK

As a cluster administrator, you can configure an additional network for your cluster using the macvlan Container Network Interface (CNI) plug-in with advanced customization. When a pod is attached to the network, the plug-in creates a sub-interface from the parent interface on the host. A unique hardware mac address is generated for each sub-device.

IMPORTANT

The unique MAC addresses this plug-in generates for sub-interfaces might not be compatible with the security polices of your cloud provider.

You specify a configuration with a CNI object. This approach allows you to specify additional configuration options that are not available when using a YAML configuration.

12.10.1. Creating an additional network attachment with the macvlan CNI plug-in

The Cluster Network Operator (CNO) manages additional network definitions. When you specify an additional network to create, the CNO creates the **NetworkAttachmentDefinition** object automatically.

IMPORTANT

Do not edit the **NetworkAttachmentDefinition** objects that the Cluster Network Operator manages. Doing so might disrupt network traffic on your additional network.

Prerequisites

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

Procedure

To create an additional network for your cluster, complete the following steps:

1. Edit the CNO CR by running the following command:

```
$ oc edit networks.operator.openshift.io cluster
```

2. Modify the CR that you are creating by adding the configuration for the additional network you are creating, as in the following example CR.
   The following YAML configures the macvlan CNI plug-in:

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  additionalNetworks: (1)
  - name: test-network-1
    namespace: test-1
    type: Raw
    rawCNIConfig: '{
      "cniVersion": "0.3.1",
      "name": "test-network-1",
      "type": "macvlan",
      "master": "eth1",
      "ipam": {
        "type": "static",
        "addresses": [
          {
            "address": "192.168.1.23/24"
          }
        ]
      }
    }'
```

(1) Specify the configuration for the additional network attachment definition.

3. Save your changes and quit the text editor to commit your changes.

4. Confirm that the CNO created the NetworkAttachmentDefinition object by running the following command. Replace **<namespace>** with the namespace that you specified when configuring the network attachment. There might be a delay before the CNO creates the object.

```
$ oc get network-attachment-definitions -n <namespace>
```

Example output

```
NAME             AGE
test-network-1   14m
```

12.10.1.1. Configuration for macvlan CNI plug-in

The configuration for an additional network attachment that uses the macvlan Container Network Interface (CNI) plug-in is provided in two parts:

- Cluster Network Operator (CNO) configuration
- CNI plug-in configuration

The CNO configuration specifies the name for the additional network attachment and the namespace to create the attachment in. The plug-in is configured by a JSON object specified by the **rawCNIConfig** parameter in the CNO configuration.

The following YAML describes the configuration parameters for the CNO:

Cluster Network Operator YAML configuration

```
name: <name> 1
namespace: <namespace> 2
rawCNIConfig: '{ 3
  ...
}'
type: Raw
```

1. Specify a name for the additional network attachment that you are creating. The name must be unique within the specified **namespace**.
2. Specify the namespace to create the network attachment in. If you do not specify a value, then the **default** namespace is used.
3. Specify the CNI plug-in configuration in JSON format, which is based on the following template.

The following object describes the configuration parameters for the macvlan CNI plug-in:

macvlan CNI plug-in JSON configuration object

```
{
  "cniVersion": "0.3.1",
  "name": "<name>", 1
  "type": "macvlan",
  "mode": "<mode>", 2
  "master": "<master>", 3
  "mtu": <mtu>, 4
  "ipam": { 5
    ...
  }
}
```

1. Specify a name for the additional network attachment that you are creating. The name must be unique within the specified **namespace**.
2. Configures traffic visibility on the virtual network. Must be either **bridge**, **passthru**, **private**, or **vepa**. If a value is not provided, the default value is **bridge**.
3. The ethernet, bonded, or VLAN interface to associate with the virtual interface. If a value is not specified, then the host system’s primary ethernet interface is used.
4. Set the maximum transmission unit (MTU) to the specified value. The default value is automatically set by the kernel.

5 Specify a configuration object for the ipam CNI plug-in. The plug-in manages IP address assignment for the attachment definition.

12.10.1.1.1. macvlan configuration example

The following example configures an additional network named **macvlan-net**:

```
name: macvlan-net
namespace: work-network
type: Raw
rawCNIConfig: |-
  {
    "cniVersion": "0.3.1",
    "name": "macvlan-net",
    "type": "macvlan",
    "master": "eth1",
    "mode": "bridge",
    "ipam": {
      "type": "dhcp"
      }
  }
```

12.10.1.2. Configuration for ipam CNI plug-in

The ipam Container Network Interface (CNI) plug-in provides IP address management (IPAM) for other CNI plug-ins.

You can use the following methods for IP address assignment:

- Static assignment.
- Dynamic assignment through a DHCP server. The DHCP server you specify must be reachable from the additional network.
- Dynamic assignment through the Whereabouts IPAM CNI plug-in.

12.10.1.2.1. Static IP address assignment configuration

The following JSON describes the configuration for static IP address assignment:

Static assignment configuration

```
{
  "ipam": {
    "type": "static",
    "addresses": [ 1
      {
        "address": "<address>", 2
        "gateway": "<gateway>" 3
      }
    ],
    "routes": [ 4
      {
```

```
      "dst": "<dst>", 5
      "gw": "<gw>" 6
    }
  ],
  "dns": { 7
    "nameservers": ["<nameserver>"], 8
    "domain": "<domain>", 9
    "search": ["<search_domain>"] 10
  }
 }
}
```

1. An array describing IP addresses to assign to the virtual interface. Both IPv4 and IPv6 IP addresses are supported.

2. An IP address and network prefix that you specify. For example, if you specify **10.10.21.10/24**, then the additional network is assigned an IP address of **10.10.21.10** and the netmask is **255.255.255.0**.

3. The default gateway to route egress network traffic to.

4. An array describing routes to configure inside the pod.

5. The IP address range in CIDR format, such as **192.168.17.0/24**, or **0.0.0.0/0** for the default route.

6. The gateway where network traffic is routed.

7. Optional: DNS configuration.

8. An of array of one or more IP addresses for to send DNS queries to.

9. The default domain to append to a hostname. For example, if the domain is set to **example.com**, a DNS lookup query for **example-host** is rewritten as **example-host.example.com**.

10. An array of domain names to append to an unqualified hostname, such as **example-host**, during a DNS lookup query.

12.10.1.2.2. Dynamic IP address assignment configuration

The following JSON describes the configuration for dynamic IP address address assignment with DHCP.

RENEWAL OF DHCP LEASES

A pod obtains its original DHCP lease when it is created. The lease must be periodically renewed by a minimal DHCP server deployment running on the cluster.

To trigger the deployment of the DHCP server, you must create a shim network attachment by editing the Cluster Network Operator configuration, as in the following example:

Example shim network attachment definition

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  ...
  additionalNetworks:
  - name: dhcp-shim
    namespace: default
    type: Raw
    rawCNIConfig: |-
      {
        "name": "dhcp-shim",
        "cniVersion": "0.3.1",
        "type": "bridge",
        "ipam": {
          "type": "dhcp"
        }
      }
```

DHCP assignment configuration

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

12.10.1.2.3. Dynamic IP address assignment configuration with Whereabouts

The Whereabouts CNI plug-in allows the dynamic assignment of an IP address to an additional network without the use of a DHCP server.

The following JSON describes the configuration for dynamic IP address assignment with Whereabouts:

Whereabouts assignment configuration

```
{
  "ipam": {
    "type": "whereabouts",
    "range": "<range>", 1
```

```
    "exclude": ["<exclude_part>, ..."], 2
  }
}
```

1. Specify an IP address and range in CIDR notation. IP addresses are assigned from within this range of addresses.

2. Optional: Specify a list of IP addresses and ranges in CIDR notation. IP addresses within an excluded address range are not assigned.

12.10.1.2.4. Static IP address assignment configuration example

You can configure ipam for static IP address assignment:

```
{
  "ipam": {
    "type": "static",
      "addresses": [
        {
          "address": "191.168.1.7"
        }
      ]
  }
}
```

12.10.1.2.5. Dynamic IP address assignment configuration example using DHCP

You can configure ipam for DHCP:

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

12.10.1.2.6. Dynamic IP address assignment configuration example using Whereabouts

You can configure ipam to use Whereabouts:

```
{
  "ipam": {
    "type": "whereabouts",
    "range": "192.0.2.192/27",
    "exclude": [
       "192.0.2.192/30",
       "192.0.2.196/32"
    ]
  }
}
```

12.10.2. Next steps

- Attach a pod to an additional network .

12.11. EDITING AN ADDITIONAL NETWORK

As a cluster administrator you can modify the configuration for an existing additional network.

12.11.1. Modifying an additional network attachment definition

As a cluster administrator, you can make changes to an existing additional network. Any existing pods attached to the additional network will not be updated.

Prerequisites

- You have configured an additional network for your cluster.
- Install the OpenShift CLI (**oc**).
- Log in as a user with **cluster-admin** privileges.

Procedure

To edit an additional network for your cluster, complete the following steps:

1. Run the following command to edit the Cluster Network Operator (CNO) CR in your default text editor:

```
$ oc edit networks.operator.openshift.io cluster
```

2. In the **additionalNetworks** collection, update the additional network with your changes.

3. Save your changes and quit the text editor to commit your changes.

4. Optional: Confirm that the CNO updated the **NetworkAttachmentDefinition** object by running the following command. Replace **<network-name>** with the name of the additional network to display. There might be a delay before the CNO updates the **NetworkAttachmentDefinition** object to reflect your changes.

```
$ oc get network-attachment-definitions <network-name> -o yaml
```

For example, the following console output displays a **NetworkAttachmentDefinition** object that is named **net1**:

```
$ oc get network-attachment-definitions net1 -o go-template='{{printf "%s\n" .spec.config}}'
{ "cniVersion": "0.3.1", "type": "macvlan",
"master": "ens5",
"mode": "bridge",
"ipam":       {"type":"static","routes":[{"dst":"0.0.0.0/0","gw":"10.128.2.1"}],"addresses":
[{"address":"10.128.2.100/23","gateway":"10.128.2.1"}],"dns":{"nameservers":
["172.30.0.10"],"domain":"us-west-2.compute.internal","search":["us-west-
2.compute.internal"]}} }
```

12.12. REMOVING AN ADDITIONAL NETWORK

As a cluster administrator you can remove an additional network attachment.

12.12.1. Removing an additional network attachment definition

As a cluster administrator, you can remove an additional network from your OpenShift Container Platform cluster. The additional network is not removed from any pods it is attached to.

Prerequisites

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

Procedure

To remove an additional network from your cluster, complete the following steps:

1. Edit the Cluster Network Operator (CNO) in your default text editor by running the following command:

```
$ oc edit networks.operator.openshift.io cluster
```

2. Modify the CR by removing the configuration from the **additionalNetworks** collection for the network attachment definition you are removing.

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  additionalNetworks: [] 1
```

1 If you are removing the configuration mapping for the only additional network attachment definition in the **additionalNetworks** collection, you must specify an empty collection.

3. Save your changes and quit the text editor to commit your changes.
4. Optional: Confirm that the additional network CR was deleted by running the following command:

```
$ oc get network-attachment-definition --all-namespaces
```

12.13. ASSIGNING A SECONDARY NETWORK TO A VRF

IMPORTANT

CNI VRF plug-in 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 https://access.redhat.com/support/offerings/techpreview/.

12.13.1. Assigning a secondary network to a VRF

As a cluster administrator, you can configure an additional network for your VRF domain by using the CNI VRF plug-in. The virtual network created by this plug-in is associated with a physical interface that you specify.

> **NOTE**
>
> Applications that use VRFs need to bind to a specific device. The common usage is to use the **SO_BINDTODEVICE** option for a socket. **SO_BINDTODEVICE** binds the socket to a device that is specified in the passed interface name, for example, **eth1**. To use **SO_BINDTODEVICE**, the application must have **CAP_NET_RAW** capabilities.

12.13.1.1. Creating an additional network attachment with the CNI VRF plug-in

The Cluster Network Operator (CNO) manages additional network definitions. When you specify an additional network to create, the CNO creates the **NetworkAttachmentDefinition** custom resource (CR) automatically.

> **NOTE**
>
> Do not edit the **NetworkAttachmentDefinition** CRs that the Cluster Network Operator manages. Doing so might disrupt network traffic on your additional network.

Prerequisites

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

Procedure

1. Create the CNO CR by running the following command:

```
$ oc edit networks.operator.openshift.io cluster
```

2. Extend the CR that you are creating by adding the **rawCNIConfig** configuration for the additional network, as in the example CR below. The following YAML configures the CNI VRF plug-in:

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
  spec:
  additionalNetworks:
  - name: test-network-1
    namespace: test-1
    type: Raw
    rawCNIConfig: '{
      "cniVersion": "0.3.1",
      "name": "macvlan-vrf",
      "plugins": [  1
      {
```

```
        "type": "macvlan", (2)
        "master": "eth1",
        "ipam": {
            "type": "static",
            "addresses": [
            {
                "address": "191.168.1.23/24"
            }
            ]
        }
      },
      {
        "type": "vrf",
        "vrfname": "example-vrf-name", (3)
        "table": 1001   (4)
      }]
    }'
```

1. **plugins** must be a list. The first item in the list must be secondary network underpinning the VRF network. The second item in the list is the VRF plugin configuration.

2. **type** must be set to **vrf**.

3. **vrfname** is the name of the VRF that the interface is assigned to. If it does not exist in the pod, it is created.

4. **table** is the routing table ID. Optional. By default, the **tableid** parameter is used. If it is not specified, the CNI assigns a free routing table ID to the VRF.

> **NOTE**
>
> VRF will function correctly only when the resource is of type **netdevice**.

3. Save your changes and quit the text editor to commit your changes.

4. Confirm that the CNO created the **NetworkAttachmentDefinition** CR by running the following command. Replace **<namespace>** with the namespace that you specified when configuring the network attachment. There might be a delay before the CNO creates the CR.

```
$ oc get network-attachment-definitions -n <namespace>
```

Example output

```
NAME                 AGE
additional-network-1   14m
```

Verifying that the additional VRF network attachment is successful

To verify that the VRF CNI is correctly configured and the additional network attachment is attached, do the following:

1. Create a network that uses the VRF CNI.

2. Assign the network to a pod.

3. Verify that the pod network attachment is connected to the VRF additional network. SSH into the pod and run the following command:

```
$ ip vrf show
```

Example output

```
Name           Table
-----------------------
red              10
```

4. Confirm the VRF interface is master of the secondary interface:

```
$ ip link
```

Example output

```
5: net1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master red
state UP mode
```

CHAPTER 13. HARDWARE NETWORKS

13.1. ABOUT SINGLE ROOT I/O VIRTUALIZATION (SR-IOV) HARDWARE NETWORKS

The Single Root I/O Virtualization (SR-IOV) specification is a standard for a type of PCI device assignment that can share a single device with multiple pods.

SR-IOV can segment a compliant network device, recognized on the host node as a physical function (PF), into multiple virtual functions (VFs). The VF is used like any other network device. The SR-IOV network device driver for the device determines how the VF is exposed in the container:

- **netdevice** driver: A regular kernel network device in the **netns** of the container
- **vfio-pci** driver: A character device mounted in the container

You can use SR-IOV network devices with additional networks on your OpenShift Container Platform cluster for application that require high bandwidth or low latency.

13.1.1. Components that manage SR-IOV network devices

The SR-IOV Network Operator creates and manages the components of the SR-IOV stack. It performs the following functions:

- Orchestrates discovery and management of SR-IOV network devices
- Generates **NetworkAttachmentDefinition** custom resources for the SR-IOV Container Network Interface (CNI)
- Creates and updates the configuration of the SR-IOV network device plug-in
- Creates node specific **SriovNetworkNodeState** custom resources
- Updates the **spec.interfaces** field in each **SriovNetworkNodeState** custom resource

The Operator provisions the following components:

SR-IOV network configuration daemon

A daemon set that is deployed on worker nodes when the SR-IOV Network Operator starts. The daemon is responsible for discovering and initializing SR-IOV network devices in the cluster.

SR-IOV Network Operator webhook

A dynamic admission controller webhook that validates the Operator custom resource and sets appropriate default values for unset fields.

SR-IOV Network resources injector

A dynamic admission controller webhook that provides functionality for patching Kubernetes pod specifications with requests and limits for custom network resources such as SR-IOV VFs.

SR-IOV network device plug-in

A device plug-in that discovers, advertises, and allocates SR-IOV network virtual function (VF) resources. Device plug-ins are used in Kubernetes to enable the use of limited resources, typically in physical devices. Device plug-ins give the Kubernetes scheduler awareness of resource availability, so that the scheduler can schedule pods on nodes with sufficient resources.

SR-IOV CNI plug-in

A CNI plug-in that attaches VF interfaces allocated from the SR-IOV network device plug-in directly into a pod.

SR-IOV InfiniBand CNI plug-in

A CNI plug-in that attaches InfiniBand (IB) VF interfaces allocated from the SR-IOV network device plug-in directly into a pod.

NOTE

The SR-IOV Network resources injector and SR-IOV Network Operator webhook are enabled by default and can be disabled by editing the **default SriovOperatorConfig** CR.

13.1.1.1. Supported devices

OpenShift Container Platform supports the following network interface controllers:

Table 13.1. Supported network interface controllers

Manufacturer	Model	Vendor ID	Device ID
Intel	X710	8086	1572
Intel	XL710	8086	1583
Intel	XXV710	8086	158b
Mellanox	MT27700 Family [ConnectX-4]	15b3	1013
Mellanox	MT27710 Family [ConnectX-4 Lx]	15b3	1015
Mellanox	MT27800 Family [ConnectX-5]	15b3	1017
Mellanox	MT28880 Family [ConnectX-5 Ex]	15b3	1019
Mellanox	MT28908 Family [ConnectX-6]	15b3	101b

13.1.1.2. Automated discovery of SR-IOV network devices

The SR-IOV Network Operator searches your cluster for SR-IOV capable network devices on worker nodes. The Operator creates and updates a SriovNetworkNodeState custom resource (CR) for each worker node that provides a compatible SR-IOV network device.

The CR is assigned the same name as the worker node. The **status.interfaces** list provides information about the network devices on a node.

IMPORTANT

Do not modify a **SriovNetworkNodeState** object. The Operator creates and manages these resources automatically.

13.1.1.2.1. Example SriovNetworkNodeState object

The following YAML is an example of a **SriovNetworkNodeState** object created by the SR-IOV Network Operator:

An SriovNetworkNodeState object

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodeState
metadata:
  name: node-25 1
  namespace: openshift-sriov-network-operator
  ownerReferences:
  - apiVersion: sriovnetwork.openshift.io/v1
    blockOwnerDeletion: true
    controller: true
    kind: SriovNetworkNodePolicy
    name: default
spec:
  dpConfigVersion: "39824"
status:
  interfaces: 2
  - deviceID: "1017"
    driver: mlx5_core
    mtu: 1500
    name: ens785f0
    pciAddress: "0000:18:00.0"
    totalvfs: 8
    vendor: 15b3
  - deviceID: "1017"
    driver: mlx5_core
    mtu: 1500
    name: ens785f1
    pciAddress: "0000:18:00.1"
    totalvfs: 8
    vendor: 15b3
  - deviceID: 158b
    driver: i40e
    mtu: 1500
    name: ens817f0
    pciAddress: 0000:81:00.0
    totalvfs: 64
    vendor: "8086"
  - deviceID: 158b
    driver: i40e
    mtu: 1500
    name: ens817f1
    pciAddress: 0000:81:00.1
    totalvfs: 64
    vendor: "8086"
  - deviceID: 158b
    driver: i40e
    mtu: 1500
    name: ens803f0
    pciAddress: 0000:86:00.0
    totalvfs: 64
    vendor: "8086"
  syncStatus: Succeeded
```

1. The value of the **name** field is the same as the name of the worker node.
2. The **interfaces** stanza includes a list of all of the SR-IOV devices discovered by the Operator on the worker node.

13.1.1.3. Example use of a virtual function in a pod

You can run a remote direct memory access (RDMA) or a Data Plane Development Kit (DPDK) application in a pod with SR-IOV VF attached.

This example shows a pod using a virtual function (VF) in RDMA mode:

Pod spec that uses RDMA mode

```
apiVersion: v1
kind: Pod
metadata:
  name: rdma-app
  annotations:
    k8s.v1.cni.cncf.io/networks: sriov-rdma-mlnx
spec:
  containers:
  - name: testpmd
    image: <RDMA_image>
    imagePullPolicy: IfNotPresent
    securityContext:
     capabilities:
        add: ["IPC_LOCK"]
    command: ["sleep", "infinity"]
```

The following example shows a pod with a VF in DPDK mode:

Pod spec that uses DPDK mode

```
apiVersion: v1
kind: Pod
metadata:
  name: dpdk-app
  annotations:
    k8s.v1.cni.cncf.io/networks: sriov-dpdk-net
spec:
  containers:
  - name: testpmd
    image: <DPDK_image>
    securityContext:
     capabilities:
        add: ["IPC_LOCK"]
    volumeMounts:
    - mountPath: /dev/hugepages
      name: hugepage
    resources:
      limits:
        memory: "1Gi"
```

```
        cpu: "2"
        hugepages-1Gi: "4Gi"
      requests:
        memory: "1Gi"
        cpu: "2"
        hugepages-1Gi: "4Gi"
    command: ["sleep", "infinity"]
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
```

13.1.1.4. DPDK library for use with container applications

An optional library, **app-netutil**, provides several API methods for gathering network information about a pod from within a container running within that pod.

This library can assist with integrating SR-IOV virtual functions (VFs) in Data Plane Development Kit (DPDK) mode into the container. The library provides both a Golang API and a C API.

Currently there are three API methods implemented:

GetCPUInfo()
: This function determines which CPUs are available to the container and returns the list.

GetHugepages()
: This function determines the amount of huge page memory requested in the **Pod** spec for each container and returns the values.

GetInterfaces()
: This function determines the set of interfaces in the container and returns the list. The return value includes the interface type and type-specific data for each interface.

The repository for the library includes a sample Dockerfile to build a container image, **dpdk-app-centos**. The container image can run one of the following DPDK sample applications, depending on an environment variable in the pod specification: **l2fwd**, **l3wd** or **testpmd**. The container image provides an example of integrating the **app-netutil** library into the container image itself. The library can also integrate into an init container. The init container can collect the required data and pass the data to an existing DPDK workload.

13.1.1.5. Huge pages resource injection for Downward API

When a pod specification includes a resource request or limit for huge pages, the Network Resources Injector automatically adds Downward API fields to the pod specification to provide the huge pages information to the container.

The Network Resources Injector adds a volume that is named **podnetinfo** and is mounted at **/etc/podnetinfo** for each container in the pod. The volume uses the Downward API and includes a file for huge pages requests and limits. The file naming convention is as follows:

- **/etc/podnetinfo/hugepages_1G_request_<container-name>**
- **/etc/podnetinfo/hugepages_1G_limit_<container-name>**
- **/etc/podnetinfo/hugepages_2M_request_<container-name>**

- **/etc/podnetinfo/hugepages_2M_limit_<container-name>**

The paths specified in the previous list are compatible with the **app-netutil** library. By default, the library is configured to search for resource information in the **/etc/podnetinfo** directory. If you choose to specify the Downward API path items yourself manually, the **app-netutil** library searches for the following paths in addition to the paths in the previous list.

- **/etc/podnetinfo/hugepages_request**
- **/etc/podnetinfo/hugepages_limit**
- **/etc/podnetinfo/hugepages_1G_request**
- **/etc/podnetinfo/hugepages_1G_limit**
- **/etc/podnetinfo/hugepages_2M_request**
- **/etc/podnetinfo/hugepages_2M_limit**

As with the paths that the Network Resources Injector can create, the paths in the preceding list can optionally end with a **_<container-name>** suffix.

13.1.2. Next steps

- Installing the SR-IOV Network Operator
- Optional: Configuring the SR-IOV Network Operator
- Configuring an SR-IOV network device
- If you use OpenShift Virtualization: Configuring an SR-IOV network device for virtual machines
- Configuring an SR-IOV network attachment
- Adding a pod to an SR-IOV additional network

13.2. INSTALLING THE SR-IOV NETWORK OPERATOR

You can install the Single Root I/O Virtualization (SR-IOV) Network Operator on your cluster to manage SR-IOV network devices and network attachments.

13.2.1. Installing SR-IOV Network Operator

As a cluster administrator, you can install the SR-IOV Network Operator by using the OpenShift Container Platform CLI or the web console.

13.2.1.1. CLI: Installing the SR-IOV Network Operator

As a cluster administrator, you can install the Operator using the CLI.

Prerequisites

- A cluster installed on bare-metal hardware with nodes that have hardware that supports SR-IOV.

- Install the OpenShift CLI (**oc**).
- An account with **cluster-admin** privileges.

Procedure

1. To create the **openshift-sriov-network-operator** namespace, enter the following command:

```
$ cat << EOF| oc create -f -
apiVersion: v1
kind: Namespace
metadata:
  name: openshift-sriov-network-operator
  annotations:
    workload.openshift.io/allowed: management
EOF
```

2. To create an OperatorGroup CR, enter the following command:

```
$ cat << EOF| oc create -f -
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: sriov-network-operators
  namespace: openshift-sriov-network-operator
spec:
  targetNamespaces:
  - openshift-sriov-network-operator
EOF
```

3. Subscribe to the SR-IOV Network Operator.

   a. Run the following command to get the OpenShift Container Platform major and minor version. It is required for the **channel** value in the next step.

   ```
   $ OC_VERSION=$(oc version -o yaml | grep openshiftVersion | \
       grep -o '[0-9]*[.][0-9]*' | head -1)
   ```

   b. To create a Subscription CR for the SR-IOV Network Operator, enter the following command:

   ```
   $ cat << EOF| oc create -f -
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
     name: sriov-network-operator-subsription
     namespace: openshift-sriov-network-operator
   spec:
     channel: "${OC_VERSION}"
     name: sriov-network-operator
     source: redhat-operators
     sourceNamespace: openshift-marketplace
   EOF
   ```

4. To verify that the Operator is installed, enter the following command:

```
$ oc get csv -n openshift-sriov-network-operator \
  -o custom-columns=Name:.metadata.name,Phase:.status.phase
```

Example output

```
Name                                        Phase
sriov-network-operator.4.4.0-202006160135   Succeeded
```

13.2.1.2. Web console: Installing the SR-IOV Network Operator

As a cluster administrator, you can install the Operator using the web console.

> **NOTE**
>
> You must create the operator group by using the CLI.

Prerequisites

- A cluster installed on bare-metal hardware with nodes that have hardware that supports SR-IOV.
- Install the OpenShift CLI (**oc**).
- An account with **cluster-admin** privileges.

Procedure

1. Create a namespace for the SR-IOV Network Operator:
   a. In the OpenShift Container Platform web console, click **Administration → Namespaces**.
   b. Click **Create Namespace**.
   c. In the **Name** field, enter **openshift-sriov-network-operator**, and then click **Create**.
2. Install the SR-IOV Network Operator:
   a. In the OpenShift Container Platform web console, click **Operators → OperatorHub**.
   b. Select **SR-IOV Network Operator** from the list of available Operators, and then click **Install**.
   c. On the **Install Operator** page, under **A specific namespace on the cluster**, select **openshift-sriov-network-operator**.
   d. Click **Install**.
3. Verify that the SR-IOV Network Operator is installed successfully:
   a. Navigate to the **Operators → Installed Operators** page.
   b. Ensure that **SR-IOV Network Operator** is listed in the **openshift-sriov-network-operator** project with a **Status** of **InstallSucceeded**.

> **NOTE**
>
> During installation an Operator might display a **Failed** status. If the installation later succeeds with an **InstallSucceeded** message, you can ignore the **Failed** message.

If the operator does not appear as installed, to troubleshoot further:

- Inspect the **Operator Subscriptions** and **Install Plans** tabs for any failure or errors under **Status**.
- Navigate to the **Workloads → Pods** page and check the logs for pods in the **openshift-sriov-network-operator** project.

13.2.2. Next steps

- Optional: Configuring the SR-IOV Network Operator

13.3. CONFIGURING THE SR-IOV NETWORK OPERATOR

The Single Root I/O Virtualization (SR-IOV) Network Operator manages the SR-IOV network devices and network attachments in your cluster.

13.3.1. Configuring the SR-IOV Network Operator

> **IMPORTANT**
>
> Modifying the SR-IOV Network Operator configuration is not normally necessary. The default configuration is recommended for most use cases. Complete the steps to modify the relevant configuration only if the default behavior of the Operator is not compatible with your use case.

The SR-IOV Network Operator adds the **SriovOperatorConfig.sriovnetwork.openshift.io** CustomResourceDefinition resource. The operator automatically creates a SriovOperatorConfig custom resource (CR) named **default** in the **openshift-sriov-network-operator** namespace.

> **NOTE**
>
> The **default** CR contains the SR-IOV Network Operator configuration for your cluster. To change the operator configuration, you must modify this CR.

The **SriovOperatorConfig** object provides several fields for configuring the operator:

- **enableInjector** allows project administrators to enable or disable the Network Resources Injector daemon set.
- **enableOperatorWebhook** allows project administrators to enable or disable the Operator Admission Controller webhook daemon set.
- **configDaemonNodeSelector** allows project administrators to schedule the SR-IOV Network Config Daemon on selected nodes.

13.3.1.1. About the Network Resources Injector

The Network Resources Injector is a Kubernetes Dynamic Admission Controller application. It provides the following capabilities:

- Mutation of resource requests and limits in a pod specification to add an SR-IOV resource name according to an SR-IOV network attachment definition annotation.
- Mutation of a pod specification with a Downward API volume to expose pod annotations, labels, and huge pages requests and limits. Containers that run in the pod can access the exposed information as files under the **/etc/podnetinfo** path.

By default, the Network Resources Injector is enabled by the SR-IOV Network Operator and runs as a daemon set on all control plane nodes (also known as the master nodes). The following is an example of Network Resources Injector pods running in a cluster with three control plane nodes:

```
$ oc get pods -n openshift-sriov-network-operator
```

Example output

```
NAME                                  READY   STATUS    RESTARTS   AGE
network-resources-injector-5cz5p      1/1     Running   0          10m
network-resources-injector-dwqpx      1/1     Running   0          10m
network-resources-injector-lktz5      1/1     Running   0          10m
```

13.3.1.2. About the SR-IOV Network Operator admission controller webhook

The SR-IOV Network Operator Admission Controller webhook is a Kubernetes Dynamic Admission Controller application. It provides the following capabilities:

- Validation of the **SriovNetworkNodePolicy** CR when it is created or updated.
- Mutation of the **SriovNetworkNodePolicy** CR by setting the default value for the **priority** and **deviceType** fields when the CR is created or updated.

By default the SR-IOV Network Operator Admission Controller webhook is enabled by the Operator and runs as a daemon set on all control plane nodes. The following is an example of the Operator Admission Controller webhook pods running in a cluster with three control plane nodes:

```
$ oc get pods -n openshift-sriov-network-operator
```

Example output

```
NAME                                  READY   STATUS    RESTARTS   AGE
operator-webhook-9jkw6                1/1     Running   0          16m
operator-webhook-kbr5p                1/1     Running   0          16m
operator-webhook-rpfrl                1/1     Running   0          16m
```

13.3.1.3. About custom node selectors

The SR-IOV Network Config daemon discovers and configures the SR-IOV network devices on cluster nodes. By default, it is deployed to all the **worker** nodes in the cluster. You can use node labels to specify on which nodes the SR-IOV Network Config daemon runs.

13.3.1.4. Disabling or enabling the Network Resources Injector

To disable or enable the Network Resources Injector, which is enabled by default, complete the following procedure.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in as a user with **cluster-admin** privileges.
- You must have installed the SR-IOV Network Operator.

Procedure

- Set the **enableInjector** field. Replace **<value>** with **false** to disable the feature or **true** to enable the feature.

```
$ oc patch sriovoperatorconfig default \
  --type=merge -n openshift-sriov-network-operator \
  --patch '{ "spec": { "enableInjector": <value> } }'
```

TIP

You can alternatively apply the following YAML to update the Operator:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovOperatorConfig
metadata:
  name: default
  namespace: openshift-sriov-network-operator
spec:
  enableInjector: <value>
```

13.3.1.5. Disabling or enabling the SR-IOV Network Operator admission controller webhook

To disable or enable the admission controller webhook, which is enabled by default, complete the following procedure.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in as a user with **cluster-admin** privileges.
- You must have installed the SR-IOV Network Operator.

Procedure

- Set the **enableOperatorWebhook** field. Replace **<value>** with **false** to disable the feature or **true** to enable it:

```
$ oc patch sriovoperatorconfig default --type=merge \
  -n openshift-sriov-network-operator \
  --patch '{ "spec": { "enableOperatorWebhook": <value> } }'
```

TIP

You can alternatively apply the following YAML to update the Operator:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovOperatorConfig
metadata:
  name: default
  namespace: openshift-sriov-network-operator
spec:
  enableOperatorWebhook: <value>
```

13.3.1.6. Configuring a custom NodeSelector for the SR-IOV Network Config daemon

The SR-IOV Network Config daemon discovers and configures the SR-IOV network devices on cluster nodes. By default, it is deployed to all the **worker** nodes in the cluster. You can use node labels to specify on which nodes the SR-IOV Network Config daemon runs.

To specify the nodes where the SR-IOV Network Config daemon is deployed, complete the following procedure.

> IMPORTANT
>
> When you update the **configDaemonNodeSelector** field, the SR-IOV Network Config daemon is recreated on each selected node. While the daemon is recreated, cluster users are unable to apply any new SR-IOV Network node policy or create new SR-IOV pods.

Procedure

- To update the node selector for the operator, enter the following command:

```
$ oc patch sriovoperatorconfig default --type=json \
  -n openshift-sriov-network-operator \
  --patch '[{
      "op": "replace",
      "path": "/spec/configDaemonNodeSelector",
      "value": {<node_label>}
    }]'
```

Replace **<node_label>** with a label to apply as in the following example: **"node-role.kubernetes.io/worker": ""**.

TIP

You can alternatively apply the following YAML to update the Operator:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovOperatorConfig
metadata:
  name: default
  namespace: openshift-sriov-network-operator
spec:
  configDaemonNodeSelector:
    <node_label>
```

13.3.2. Next steps

- Configuring an SR-IOV network device

13.4. CONFIGURING AN SR-IOV NETWORK DEVICE

You can configure a Single Root I/O Virtualization (SR-IOV) device in your cluster.

13.4.1. SR-IOV network node configuration object

You specify the SR-IOV network device configuration for a node by creating an SR-IOV network node policy. The API object for the policy is part of the **sriovnetwork.openshift.io** API group.

The following YAML describes an SR-IOV network node policy:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: <name> 1
  namespace: openshift-sriov-network-operator 2
spec:
  resourceName: <sriov_resource_name> 3
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true" 4
  priority: <priority> 5
  mtu: <mtu> 6
  numVfs: <num> 7
  nicSelector: 8
    vendor: "<vendor_code>" 9
    deviceID: "<device_id>" 10
    pfNames: ["<pf_name>", ...] 11
    rootDevices: ["<pci_bus_id>", ...] 12
    netFilter: "<filter_string>" 13
  deviceType: <device_type> 14
  isRdma: false 15
  linkType: <link_type> 16
```

1. The name for the custom resource object.

2. The namespace where the SR-IOV Network Operator is installed.

3. The resource name of the SR-IOV network device plug-in. You can create multiple SR-IOV network node policies for a resource name.

4. The node selector specifies the nodes to configure. Only SR-IOV network devices on the selected nodes are configured. The SR-IOV Container Network Interface (CNI) plug-in and device plug-in are deployed on selected nodes only.

5. Optional: The priority is an integer value between **0** and **99**. A smaller value receives higher priority. For example, a priority of **10** is a higher priority than **99**. The default value is **99**.

6. Optional: The maximum transmission unit (MTU) of the virtual function. The maximum MTU value can vary for different network interface controller (NIC) models.

7 The number of the virtual functions (VF) to create for the SR-IOV physical network device. For an Intel network interface controller (NIC), the number of VFs cannot be larger than the total VFs

8 The NIC selector identifies the device for the Operator to configure. You do not have to specify values for all the parameters. It is recommended to identify the network device with enough precision to avoid selecting a device unintentionally.

If you specify **rootDevices**, you must also specify a value for **vendor**, **deviceID**, or **pfNames**. If you specify both **pfNames** and **rootDevices** at the same time, ensure that they refer to the same device. If you specify a value for **netFilter**, then you do not need to specify any other parameter because a network ID is unique.

9 Optional: The vendor hexadecimal code of the SR-IOV network device. The only allowed values are **8086** and **15b3**.

10 Optional: The device hexadecimal code of the SR-IOV network device. For example, **101b** is the device ID for a Mellanox ConnectX-6 device.

11 Optional: An array of one or more physical function (PF) names for the device.

12 Optional: An array of one or more PCI bus addresses for the PF of the device. Provide the address in the following format: **0000:02:00.1**.

13 Optional: The platform-specific network filter. The only supported platform is Red Hat OpenStack Platform (RHOSP). Acceptable values use the following format: **openstack/NetworkID:xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx**. Replace **xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx** with the value from the **/var/config/openstack/latest/network_data.json** metadata file.

14 Optional: The driver type for the virtual functions. The only allowed values are **netdevice** and **vfio-pci**. The default value is **netdevice**.

For a Mellanox NIC to work in Data Plane Development Kit (DPDK) mode on bare metal nodes, use the **netdevice** driver type and set **isRdma** to **true**.

15 Optional: Whether to enable remote direct memory access (RDMA) mode. The default value is **false**.

If the **isRDMA** parameter is set to **true**, you can continue to use the RDMA-enabled VF as a normal network device. A device can be used in either mode.

16 Optional: The link type for the VFs. You can specify one of the following values: **eth** or **ib**. Specify **eth** for Ethernet or **ib** for InfiniBand. The default value is **eth**.

When **linkType** is set to **ib**, **isRdma** is automatically set to **true** by the SR-IOV Network Operator webhook. When **linkType** is set to **ib**, **deviceType** should not be set to **vfio-pci**.

13.4.1.1. SR-IOV network node configuration examples

The following example describes the configuration for an InfiniBand device:

Example configuration for an InfiniBand device

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: policy-ib-net-1
```

```yaml
  namespace: openshift-sriov-network-operator
spec:
  resourceName: ibnic1
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  numVfs: 4
  nicSelector:
    vendor: "15b3"
    deviceID: "101b"
    rootDevices:
      - "0000:19:00.0"
  linkType: ib
  isRdma: true
```

The following example describes the configuration for an SR-IOV network device in a RHOSP virtual machine:

Example configuration for an SR-IOV device in a virtual machine

```yaml
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: policy-sriov-net-openstack-1
  namespace: openshift-sriov-network-operator
spec:
  resourceName: sriovnic1
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  numVfs: 1 1
  nicSelector:
    vendor: "15b3"
    deviceID: "101b"
    netFilter: "openstack/NetworkID:ea24bd04-8674-4f69-b0ee-fa0b3bd20509" 2
```

1 The **numVfs** field is always set to **1** when configuring the node network policy for a virtual machine.

2 The **netFilter** field must refer to a network ID when the virtual machine is deployed on RHOSP. Valid values for **netFilter** are available from an **SriovNetworkNodeState** object.

13.4.1.2. Virtual function (VF) partitioning for SR-IOV devices

In some cases, you might want to split virtual functions (VFs) from the same physical function (PF) into multiple resource pools. For example, you might want some of the VFs to load with the default driver and the remaining VFs load with the **vfio-pci** driver. In such a deployment, the **pfNames** selector in your SriovNetworkNodePolicy custom resource (CR) can be used to specify a range of VFs for a pool using the following format: **<pfname>#<first_vf>-<last_vf>**.

For example, the following YAML shows the selector for an interface named **netpf0** with VF **2** through **7**:

```yaml
pfNames: ["netpf0#2-7"]
```

- **netpf0** is the PF interface name.
- **2** is the first VF index (0-based) that is included in the range.

- **7** is the last VF index (0-based) that is included in the range.

You can select VFs from the same PF by using different policy CRs if the following requirements are met:

- The **numVfs** value must be identical for policies that select the same PF.
- The VF index must be in the range of **0** to **<numVfs>-1**. For example, if you have a policy with **numVfs** set to **8**, then the **<first_vf>** value must not be smaller than **0**, and the **<last_vf>** must not be larger than **7**.
- The VFs ranges in different policies must not overlap.
- The **<first_vf>** must not be larger than the **<last_vf>**.

The following example illustrates NIC partitioning for an SR-IOV device.

The policy **policy-net-1** defines a resource pool **net-1** that contains the VF **0** of PF **netpf0** with the default VF driver. The policy **policy-net-1-dpdk** defines a resource pool **net-1-dpdk** that contains the VF **8** to **15** of PF **netpf0** with the **vfio** VF driver.

Policy **policy-net-1**:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: policy-net-1
  namespace: openshift-sriov-network-operator
spec:
  resourceName: net1
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  numVfs: 16
  nicSelector:
    pfNames: ["netpf0#0-0"]
  deviceType: netdevice
```

Policy **policy-net-1-dpdk**:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: policy-net-1-dpdk
  namespace: openshift-sriov-network-operator
spec:
  resourceName: net1dpdk
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  numVfs: 16
  nicSelector:
    pfNames: ["netpf0#8-15"]
  deviceType: vfio-pci
```

13.4.2. Configuring SR-IOV network devices

The SR-IOV Network Operator adds the **SriovNetworkNodePolicy.sriovnetwork.openshift.io** CustomResourceDefinition to OpenShift Container Platform. You can configure an SR-IOV network device by creating a SriovNetworkNodePolicy custom resource (CR).

> **NOTE**
>
> When applying the configuration specified in a **SriovNetworkNodePolicy** object, the SR-IOV Operator might drain the nodes, and in some cases, reboot nodes.
>
> It might take several minutes for a configuration change to apply.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You have access to the cluster as a user with the **cluster-admin** role.
- You have installed the SR-IOV Network Operator.
- You have enough available nodes in your cluster to handle the evicted workload from drained nodes.
- You have not selected any control plane nodes for SR-IOV network device configuration.

Procedure

1. Create an **SriovNetworkNodePolicy** object, and then save the YAML in the **<name>-sriov-node-network.yaml** file. Replace **<name>** with the name for this configuration.
2. Create the **SriovNetworkNodePolicy** object:

   ```
   $ oc create -f <name>-sriov-node-network.yaml
   ```

   where **<name>** specifies the name for this configuration.

   After applying the configuration update, all the pods in **sriov-network-operator** namespace transition to the **Running** status.
3. To verify that the SR-IOV network device is configured, enter the following command. Replace **<node_name>** with the name of a node with the SR-IOV network device that you just configured.

   ```
   $ oc get sriovnetworknodestates -n openshift-sriov-network-operator <node_name> -o
   jsonpath='{.status.syncStatus}'
   ```

13.4.3. Troubleshooting SR-IOV configuration

After following the procedure to configure an SR-IOV network device, the following sections address some error conditions.

To display the state of nodes, run the following command:

```
$ oc get sriovnetworknodestates -n openshift-sriov-network-operator <node_name>
```

where: **<node_name>** specifies the name of a node with an SR-IOV network device.

Error output: Cannot allocate memory

```
"lastSyncError": "write /sys/bus/pci/devices/0000:3b:00.1/sriov_numvfs: cannot allocate memory"
```

When a node indicates that it cannot allocate memory, check the following items:

- Confirm that global SR-IOV settings are enabled in the BIOS for the node.
- Confirm that VT-d is enabled in the BIOS for the node.

> **IMPORTANT**
>
> CNI VRF plug-in 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 https://access.redhat.com/support/offerings/techpreview/.

13.4.4. Assigning an SR-IOV network to a VRF

As a cluster administrator, you can assign an SR-IOV network interface to your VRF domain by using the CNI VRF plug-in.

To do this, add the VRF configuration to the optional **metaPlugins** parameter of the **SriovNetwork** resource.

> **NOTE**
>
> Applications that use VRFs need to bind to a specific device. The common usage is to use the **SO_BINDTODEVICE** option for a socket. **SO_BINDTODEVICE** binds the socket to a device that is specified in the passed interface name, for example, **eth1**. To use **SO_BINDTODEVICE**, the application must have **CAP_NET_RAW** capabilities.

13.4.4.1. Creating an additional SR-IOV network attachment with the CNI VRF plug-in

The SR-IOV Network Operator manages additional network definitions. When you specify an additional SR-IOV network to create, the SR-IOV Network Operator creates the **NetworkAttachmentDefinition** custom resource (CR) automatically.

> **NOTE**
>
> Do not edit **NetworkAttachmentDefinition** custom resources that the SR-IOV Network Operator manages. Doing so might disrupt network traffic on your additional network.

Prerequisites

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

Procedure

1. Create the **SriovNetwork** CR by running the following command:

```
$ oc create sriovnetwork.openshift.io cluster
```

2. Extend the CR that you are creating by adding the **metaPlugins** configuration for the additional network you are creating, as in the following example CR.

3. Save your changes and quit the text editor to commit your changes. The following YAML configures the **SriovNetwork** object:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: example-network
  namespace: additional-sriov-network-1
spec:
  ipam: |
    {
      "type": "host-local",
      "subnet": "10.56.217.0/24",
      "rangeStart": "10.56.217.171",
      "rangeEnd": "10.56.217.181",
      "routes": [{
        "dst": "0.0.0.0/0"
      }],
      "gateway": "10.56.217.1"
    }
  vlan: 0
  resourceName: intelnics
  metaPlugins : |
    {
      "type": "vrf", 1
      "vrfname": "example-vrf-name" 2
    }
```

1. **type** must be set to **vrf**.

2. **vrfname** is the name of the VRF that the interface is assigned to. If it does not exist in the pod, it is created.

Verify the NetworkAttachmentDefinition CR is successfully created

Confirm that the SR-IOV Network Operator created the **NetworkAttachmentDefinition** CR by running the following command. Replace **<namespace>** with the namespace that you specified when configuring the network attachment. There might be a delay before the SR-IOV Network Operator creates the CR.

```
$ oc get network-attachment-definitions -n <namespace>
```

Example output

```
NAME                        AGE
additional-sriov-network-1      14m
```

Verifying that the additional SR-IOV network attachment is successful

To verify that the VRF CNI is correctly configured and the additional SR-IOV network attachment is attached, do the following:

1. Create an SR-IOV network that uses the VRF CNI.
2. Assign the network to a pod.
3. Verify that the pod network attachment is connected to the SR-IOV additional network. SSH into the pod and run the following command:

   ```
   $ ip vrf show
   ```

   Example output

   ```
   Name              Table
   -----------------------
   red                 10
   ```

4. Confirm the VRF interface is master of the secondary interface:

   ```
   $ ip link
   ```

   Example output

   ```
   5: net1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master red
   state UP mode
   ```

13.4.5. Next steps

- Configuring an SR-IOV network attachment

13.5. CONFIGURING AN SR-IOV ETHERNET NETWORK ATTACHMENT

You can configure an Ethernet network attachment for an Single Root I/O Virtualization (SR-IOV) device in the cluster.

13.5.1. Ethernet device configuration object

You can configure an Ethernet network device by defining an **SriovNetwork** object.

The following YAML describes an **SriovNetwork** object:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: <name> 1
  namespace: openshift-sriov-network-operator 2
```

```
spec:
  resourceName: <sriov_resource_name> 3
  networkNamespace: <target_namespace> 4
  vlan: <vlan> 5
  spoofChk: "<spoof_check>" 6
  ipam: |- 7
    {}
  linkState: <link_state> 8
  maxTxRate: <max_tx_rate> 9
  minTxRate: <min_tx_rate> 10
  vlanQoS: <vlan_qos> 11
  trust: "<trust_vf>" 12
  capabilities: <capabilities> 13
```

1. A name for the object. The SR-IOV Network Operator creates a **NetworkAttachmentDefinition** object with same name.

2. The namespace where the SR-IOV Network Operator is installed.

3. The value for the **spec.resourceName** parameter from the **SriovNetworkNodePolicy** object that defines the SR-IOV hardware for this additional network.

4. The target namespace for the **SriovNetwork** object. Only pods in the target namespace can attach to the additional network.

5. Optional: A Virtual LAN (VLAN) ID for the additional network. The integer value must be from **0** to **4095**. The default value is **0**.

6. Optional: The spoof check mode of the VF. The allowed values are the strings **"on"** and **"off"**.

   IMPORTANT

   You must enclose the value you specify in quotes or the object is rejected by the SR-IOV Network Operator.

7. A configuration object for the IPAM CNI plug-in as a YAML block scalar. The plug-in manages IP address assignment for the attachment definition.

8. Optional: The link state of virtual function (VF). Allowed value are **enable**, **disable** and **auto**.

9. Optional: A maximum transmission rate, in Mbps, for the VF.

10. Optional: A minimum transmission rate, in Mbps, for the VF. This value must be less than or equal to the maximum transmission rate.

    NOTE

    Intel NICs do not support the **minTxRate** parameter. For more information, see BZ#1772847.

11. Optional: An IEEE 802.1p priority level for the VF. The default value is **0**.

12. Optional: The trust mode of the VF. The allowed values are the strings **"on"** and **"off"**.

IMPORTANT

You must enclose the value that you specify in quotes, or the SR-IOV Network Operator rejects the object.

13 Optional: The capabilities to configure for this additional network. You can specify **"{ "ips": true }"** to enable IP address support or **"{ "mac": true }"** to enable MAC address support.

13.5.1.1. Configuration for ipam CNI plug-in

The ipam Container Network Interface (CNI) plug-in provides IP address management (IPAM) for other CNI plug-ins.

You can use the following methods for IP address assignment:

- Static assignment.
- Dynamic assignment through a DHCP server. The DHCP server you specify must be reachable from the additional network.
- Dynamic assignment through the Whereabouts IPAM CNI plug-in.

13.5.1.1.1. Static IP address assignment configuration

The following JSON describes the configuration for static IP address assignment:

Static assignment configuration

```
{
  "ipam": {
    "type": "static",
    "addresses": [ 1
      {
        "address": "<address>", 2
        "gateway": "<gateway>" 3
      }
    ],
    "routes": [ 4
      {
        "dst": "<dst>", 5
        "gw": "<gw>" 6
      }
    ],
    "dns": { 7
      "nameservers": ["<nameserver>"], 8
      "domain": "<domain>", 9
      "search": ["<search_domain>"] 10
    }
  }
}
```

1 An array describing IP addresses to assign to the virtual interface. Both IPv4 and IPv6 IP addresses are supported.

2 An IP address and network prefix that you specify. For example, if you specify **10.10.21.10/24**, then the additional network is assigned an IP address of **10.10.21.10** and the netmask is **255.255.255.0**.

3 The default gateway to route egress network traffic to.

4 An array describing routes to configure inside the pod.

5 The IP address range in CIDR format, such as **192.168.17.0/24**, or **0.0.0.0/0** for the default route.

6 The gateway where network traffic is routed.

7 Optional: DNS configuration.

8 An of array of one or more IP addresses for to send DNS queries to.

9 The default domain to append to a hostname. For example, if the domain is set to **example.com**, a DNS lookup query for **example-host** is rewritten as **example-host.example.com**.

10 An array of domain names to append to an unqualified hostname, such as **example-host**, during a DNS lookup query.

13.5.1.1.2. Dynamic IP address assignment configuration

The following JSON describes the configuration for dynamic IP address address assignment with DHCP.

RENEWAL OF DHCP LEASES

A pod obtains its original DHCP lease when it is created. The lease must be periodically renewed by a minimal DHCP server deployment running on the cluster.

The SR-IOV Network Operator does not create a DHCP server deployment; The Cluster Network Operator is responsible for creating the minimal DHCP server deployment.

To trigger the deployment of the DHCP server, you must create a shim network attachment by editing the Cluster Network Operator configuration, as in the following example:

Example shim network attachment definition

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  ...
  additionalNetworks:
  - name: dhcp-shim
    namespace: default
    type: Raw
    rawCNIConfig: |-
      {
        "name": "dhcp-shim",
        "cniVersion": "0.3.1",
        "type": "bridge",
        "ipam": {
          "type": "dhcp"
        }
      }
```

DHCP assignment configuration

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

13.5.1.1.3. Dynamic IP address assignment configuration with Whereabouts

The Whereabouts CNI plug-in allows the dynamic assignment of an IP address to an additional network without the use of a DHCP server.

The following JSON describes the configuration for dynamic IP address assignment with Whereabouts:

Whereabouts assignment configuration

```
{
  "ipam": {
    "type": "whereabouts",
```

```
      "range": "<range>", 1
      "exclude": ["<exclude_part>, ..."], 2
    }
  }
```

1. Specify an IP address and range in CIDR notation. IP addresses are assigned from within this range of addresses.

2. Optional: Specify a list of IP addresses and ranges in CIDR notation. IP addresses within an excluded address range are not assigned.

13.5.1.1.4. Static IP address assignment configuration example

You can configure ipam for static IP address assignment:

```
{
  "ipam": {
    "type": "static",
      "addresses": [
        {
          "address": "191.168.1.7"
        }
      ]
  }
}
```

13.5.1.1.5. Dynamic IP address assignment configuration example using DHCP

You can configure ipam for DHCP:

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

13.5.1.1.6. Dynamic IP address assignment configuration example using Whereabouts

You can configure ipam to use Whereabouts:

```
{
  "ipam": {
    "type": "whereabouts",
    "range": "192.0.2.192/27",
    "exclude": [
       "192.0.2.192/30",
       "192.0.2.196/32"
    ]
  }
}
```

13.5.2. Configuring SR-IOV additional network

You can configure an additional network that uses SR-IOV hardware by creating a **SriovNetwork** object. When you create a **SriovNetwork** object, the SR-IOV Operator automatically creates a **NetworkAttachmentDefinition** object.

NOTE

Do not modify or delete a **SriovNetwork** object if it is attached to any pods in the **running** state.

Prerequisites

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

Procedure

1. Create a **SriovNetwork** object, and then save the YAML in the **<name>.yaml** file, where **<name>** is a name for this additional network. The object specification might resemble the following example:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: attach1
  namespace: openshift-sriov-network-operator
spec:
  resourceName: net1
  networkNamespace: project2
  ipam: |-
    {
      "type": "host-local",
      "subnet": "10.56.217.0/24",
      "rangeStart": "10.56.217.171",
      "rangeEnd": "10.56.217.181",
      "gateway": "10.56.217.1"
    }
```

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

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

where **<name>** specifies the name of the additional network.

3. Optional: To confirm that the **NetworkAttachmentDefinition** object that is associated with the **SriovNetwork** object that you created in the previous step exists, enter the following command. Replace **<namespace>** with the networkNamespace you specified in the **SriovNetwork** object.

```
$ oc get net-attach-def -n <namespace>
```

13.5.3. Next steps

- Adding a pod to an SR-IOV additional network

13.5.4. Additional resources

- Configuring an SR-IOV network device

13.6. CONFIGURING AN SR-IOV INFINIBAND NETWORK ATTACHMENT

You can configure an InfiniBand (IB) network attachment for an Single Root I/O Virtualization (SR-IOV) device in the cluster.

13.6.1. InfiniBand device configuration object

You can configure an InfiniBand (IB) network device by defining an **SriovIBNetwork** object.

The following YAML describes an **SriovIBNetwork** object:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovIBNetwork
metadata:
  name: <name> 1
  namespace: openshift-sriov-network-operator 2
spec:
  resourceName: <sriov_resource_name> 3
  networkNamespace: <target_namespace> 4
  ipam: |- 5
    {}
  linkState: <link_state> 6
  capabilities: <capabilities> 7
```

1. A name for the object. The SR-IOV Network Operator creates a **NetworkAttachmentDefinition** object with same name.
2. The namespace where the SR-IOV Operator is installed.
3. The value for the **spec.resourceName** parameter from the **SriovNetworkNodePolicy** object that defines the SR-IOV hardware for this additional network.
4. The target namespace for the **SriovIBNetwork** object. Only pods in the target namespace can attach to the network device.
5. Optional: A configuration object for the IPAM CNI plug-in as a YAML block scalar. The plug-in manages IP address assignment for the attachment definition.
6. Optional: The link state of virtual function (VF). Allowed values are **enable**, **disable** and **auto**.
7. Optional: The capabilities to configure for this network. You can specify **"{ "ips": true }"** to enable IP address support or **"{ "infinibandGUID": true }"** to enable IB Global Unique Identifier (GUID) support.

13.6.1.1. Configuration for ipam CNI plug-in

The ipam Container Network Interface (CNI) plug-in provides IP address management (IPAM) for other CNI plug-ins.

You can use the following methods for IP address assignment:

- Static assignment.
- Dynamic assignment through a DHCP server. The DHCP server you specify must be reachable from the additional network.
- Dynamic assignment through the Whereabouts IPAM CNI plug-in.

13.6.1.1.1. Static IP address assignment configuration

The following JSON describes the configuration for static IP address assignment:

Static assignment configuration

```
{
  "ipam": {
    "type": "static",
    "addresses": [ 1
      {
        "address": "<address>", 2
        "gateway": "<gateway>" 3
      }
    ],
    "routes": [ 4
      {
        "dst": "<dst>", 5
        "gw": "<gw>" 6
      }
    ],
    "dns": { 7
      "nameservers": ["<nameserver>"], 8
      "domain": "<domain>", 9
      "search": ["<search_domain>"] 10
    }
  }
}
```

1. An array describing IP addresses to assign to the virtual interface. Both IPv4 and IPv6 IP addresses are supported.
2. An IP address and network prefix that you specify. For example, if you specify **10.10.21.10/24**, then the additional network is assigned an IP address of **10.10.21.10** and the netmask is **255.255.255.0**.
3. The default gateway to route egress network traffic to.
4. An array describing routes to configure inside the pod.
5. The IP address range in CIDR format, such as **192.168.17.0/24**, or **0.0.0.0/0** for the default route.
6. The gateway where network traffic is routed.
7. Optional: DNS configuration.

8 An of array of one or more IP addresses for to send DNS queries to.

9 The default domain to append to a hostname. For example, if the domain is set to **example.com**, a DNS lookup query for **example-host** is rewritten as **example-host.example.com**.

10 An array of domain names to append to an unqualified hostname, such as **example-host**, during a DNS lookup query.

13.6.1.1.2. Dynamic IP address assignment configuration

The following JSON describes the configuration for dynamic IP address address assignment with DHCP.

RENEWAL OF DHCP LEASES

A pod obtains its original DHCP lease when it is created. The lease must be periodically renewed by a minimal DHCP server deployment running on the cluster.

To trigger the deployment of the DHCP server, you must create a shim network attachment by editing the Cluster Network Operator configuration, as in the following example:

Example shim network attachment definition

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  ...
  additionalNetworks:
  - name: dhcp-shim
    namespace: default
    type: Raw
    rawCNIConfig: |-
      {
        "name": "dhcp-shim",
        "cniVersion": "0.3.1",
        "type": "bridge",
        "ipam": {
          "type": "dhcp"
        }
      }
```

DHCP assignment configuration

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

13.6.1.1.3. Dynamic IP address assignment configuration with Whereabouts

The Whereabouts CNI plug-in allows the dynamic assignment of an IP address to an additional network without the use of a DHCP server.

The following JSON describes the configuration for dynamic IP address assignment with Whereabouts:

Whereabouts assignment configuration

```
{
  "ipam": {
    "type": "whereabouts",
    "range": "<range>", (1)
    "exclude": ["<exclude_part>, ..."], (2)
  }
}
```

1. Specify an IP address and range in CIDR notation. IP addresses are assigned from within this range of addresses.

2. Optional: Specify a list of IP addresses and ranges in CIDR notation. IP addresses within an excluded address range are not assigned.

13.6.1.1.4. Static IP address assignment configuration example

You can configure ipam for static IP address assignment:

```
{
  "ipam": {
    "type": "static",
      "addresses": [
        {
          "address": "191.168.1.7"
        }
      ]
  }
}
```

13.6.1.1.5. Dynamic IP address assignment configuration example using DHCP

You can configure ipam for DHCP:

```
{
  "ipam": {
    "type": "dhcp"
  }
}
```

13.6.1.1.6. Dynamic IP address assignment configuration example using Whereabouts

You can configure ipam to use Whereabouts:

```
{
  "ipam": {
```

```
      "type": "whereabouts",
      "range": "192.0.2.192/27",
      "exclude": [
        "192.0.2.192/30",
        "192.0.2.196/32"
      ]
    }
  }
```

13.6.2. Configuring SR-IOV additional network

You can configure an additional network that uses SR-IOV hardware by creating a **SriovIBNetwork** object. When you create a **SriovIBNetwork** object, the SR-IOV Operator automatically creates a **NetworkAttachmentDefinition** object.

> NOTE
>
> Do not modify or delete a **SriovIBNetwork** object if it is attached to any pods in the **running** state.

Prerequisites

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

Procedure

1. Create a **SriovIBNetwork** object, and then save the YAML in the **<name>.yaml** file, where **<name>** is a name for this additional network. The object specification might resemble the following example:

```yaml
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovIBNetwork
metadata:
  name: attach1
  namespace: openshift-sriov-network-operator
spec:
  resourceName: net1
  networkNamespace: project2
  ipam: |-
    {
      "type": "host-local",
      "subnet": "10.56.217.0/24",
      "rangeStart": "10.56.217.171",
      "rangeEnd": "10.56.217.181",
      "gateway": "10.56.217.1"
    }
```

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

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

where **<name>** specifies the name of the additional network.

3. Optional: To confirm that the **NetworkAttachmentDefinition** object that is associated with the **SriovIBNetwork** object that you created in the previous step exists, enter the following command. Replace **<namespace>** with the networkNamespace you specified in the **SriovIBNetwork** object.

```
$ oc get net-attach-def -n <namespace>
```

13.6.3. Next steps

- Adding a pod to an SR-IOV additional network

13.6.4. Additional resources

- Configuring an SR-IOV network device

13.7. ADDING A POD TO AN SR-IOV ADDITIONAL NETWORK

You can add a pod to an existing Single Root I/O Virtualization (SR-IOV) network.

13.7.1. Runtime configuration for a network attachment

When attaching a pod to an additional network, you can specify a runtime configuration to make specific customizations for the pod. For example, you can request a specific MAC hardware address.

You specify the runtime configuration by setting an annotation in the pod specification. The annotation key is **k8s.v1.cni.cncf.io/networks**, and it accepts a JSON object that describes the runtime configuration.

13.7.1.1. Runtime configuration for an Ethernet-based SR-IOV attachment

The following JSON describes the runtime configuration options for an Ethernet-based SR-IOV network attachment.

```
[
  {
    "name": "<name>", 1
    "mac": "<mac_address>", 2
    "ips": ["<cidr_range>"] 3
  }
]
```

1 The name of the SR-IOV network attachment definition CR.

2 Optional: The MAC address for the SR-IOV device that is allocated from the resource type defined in the SR-IOV network attachment definition CR. To use this feature, you also must specify **{ "mac": true }** in the **SriovNetwork** object.

3 Optional: IP addresses for the SR-IOV device that is allocated from the resource type defined in the SR-IOV network attachment definition CR. Both IPv4 and IPv6 addresses are supported. To use this feature, you also must specify **{ "ips": true }** in the **SriovNetwork** object.

Example runtime configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
  annotations:
    k8s.v1.cni.cncf.io/networks: |-
      [
        {
          "name": "net1",
          "mac": "20:04:0f:f1:88:01",
          "ips": ["192.168.10.1/24", "2001::1/64"]
        }
      ]
spec:
  containers:
  - name: sample-container
    image: <image>
    imagePullPolicy: IfNotPresent
    command: ["sleep", "infinity"]
```

13.7.1.2. Runtime configuration for an InfiniBand-based SR-IOV attachment

The following JSON describes the runtime configuration options for an InfiniBand-based SR-IOV network attachment.

```
[
  {
    "name": "<network_attachment>", 1
    "infiniband-guid": "<guid>", 2
    "ips": ["<cidr_range>"] 3
  }
]
```

1. The name of the SR-IOV network attachment definition CR.
2. The InfiniBand GUID for the SR-IOV device. To use this feature, you also must specify **{ "infinibandGUID": true }** in the **SriovIBNetwork** object.
3. The IP addresses for the SR-IOV device that is allocated from the resource type defined in the SR-IOV network attachment definition CR. Both IPv4 and IPv6 addresses are supported. To use this feature, you also must specify **{ "ips": true }** in the **SriovIBNetwork** object.

Example runtime configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
  annotations:
    k8s.v1.cni.cncf.io/networks: |-
      [
        {
          "name": "ib1",
```

```
        "infiniband-guid": "c2:11:22:33:44:55:66:77",
        "ips": ["192.168.10.1/24", "2001::1/64"]
      }
    ]
spec:
  containers:
  - name: sample-container
    image: <image>
    imagePullPolicy: IfNotPresent
    command: ["sleep", "infinity"]
```

13.7.2. Adding a pod to an additional network

You can add a pod to an additional network. The pod continues to send normal cluster-related network traffic over the default network.

When a pod is created additional networks are attached to it. However, if a pod already exists, you cannot attach additional networks to it.

The pod must be in the same namespace as the additional network.

> **NOTE**
>
> If a network attachment is managed by the SR-IOV Network Operator, the SR-IOV Network Resource Injector adds the **resource** field to the **Pod** object automatically.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in to the cluster.
- Install the SR-IOV Operator.
- Create either an **SriovNetwork** object or an **SriovIBNetwork** object to attach the pod to.

Procedure

1. Add an annotation to the **Pod** object. Only one of the following annotation formats can be used:
   a. To attach an additional network without any customization, add an annotation with the following format. Replace **<network>** with the name of the additional network to associate with the pod:

```
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: <network>[,<network>,...] 1
```

1 To specify more than one additional network, separate each network with a comma. Do not include whitespace between the comma. If you specify the same additional network multiple times, that pod will have multiple network interfaces attached to that network.

b. To attach an additional network with customizations, add an annotation with the following format:

```
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: |-
      [
        {
          "name": "<network>", 1
          "namespace": "<namespace>", 2
          "default-route": ["<default-route>"] 3
        }
      ]
```

1 Specify the name of the additional network defined by a **NetworkAttachmentDefinition** object.

2 Specify the namespace where the **NetworkAttachmentDefinition** object is defined.

3 Optional: Specify an override for the default route, such as **192.168.17.1**.

2. To create the pod, enter the following command. Replace **<name>** with the name of the pod.

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

3. Optional: To Confirm that the annotation exists in the **Pod** CR, enter the following command, replacing **<name>** with the name of the pod.

```
$ oc get pod <name> -o yaml
```

In the following example, the **example-pod** pod is attached to the **net1** additional network:

```
$ oc get pod example-pod -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: macvlan-bridge
    k8s.v1.cni.cncf.io/networks-status: |- 1
      [{
          "name": "openshift-sdn",
          "interface": "eth0",
          "ips": [
              "10.128.2.14"
          ],
          "default": true,
          "dns": {}
      },{
          "name": "macvlan-bridge",
          "interface": "net1",
          "ips": [
              "20.2.2.100"
          ],
          "mac": "22:2f:60:a5:f8:00",
```

```
        "dns": {}
    }]
  name: example-pod
  namespace: default
spec:
  ...
status:
  ...
```

1. The **k8s.v1.cni.cncf.io/networks-status** parameter is a JSON array of objects. Each object describes the status of an additional network attached to the pod. The annotation value is stored as a plain text value.

13.7.3. Creating a non-uniform memory access (NUMA) aligned SR-IOV pod

You can create a NUMA aligned SR-IOV pod by restricting SR-IOV and the CPU resources allocated from the same NUMA node with **restricted** or **single-numa-node** Topology Manager polices.

Prerequisites

- You have installed the OpenShift CLI (**oc**).
- You have configured the CPU Manager policy to **static**. For more information on CPU Manager, see the "Additional resources" section.
- You have configured the Topology Manager policy to **single-numa-node**.

> **NOTE**
>
> When **single-numa-node** is unable to satisfy the request, you can configure the Topology Manager policy to **restricted**.

Procedure

1. Create the following SR-IOV pod spec, and then save the YAML in the **<name>-sriov-pod.yaml** file. Replace **<name>** with a name for this pod.
   The following example shows an SR-IOV pod spec:

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
  annotations:
    k8s.v1.cni.cncf.io/networks: <name> (1)
spec:
  containers:
  - name: sample-container
    image: <image> (2)
    command: ["sleep", "infinity"]
    resources:
      limits:
        memory: "1Gi" (3)
        cpu: "2" (4)
```

```
    requests:
      memory: "1Gi"
      cpu: "2"
```

1. Replace **<name>** with the name of the SR-IOV network attachment definition CR.
2. Replace **<image>** with the name of the **sample-pod** image.
3. To create the SR-IOV pod with guaranteed QoS, set **memory limits** equal to **memory requests**.
4. To create the SR-IOV pod with guaranteed QoS, set **cpu limits** equals to **cpu requests**.

2. Create the sample SR-IOV pod by running the following command:

```
$ oc create -f <filename> 1
```

1. Replace **<filename>** with the name of the file you created in the previous step.

3. Confirm that the **sample-pod** is configured with guaranteed QoS.

```
$ oc describe pod sample-pod
```

4. Confirm that the **sample-pod** is allocated with exclusive CPUs.

```
$ oc exec sample-pod -- cat /sys/fs/cgroup/cpuset/cpuset.cpus
```

5. Confirm that the SR-IOV device and CPUs that are allocated for the **sample-pod** are on the same NUMA node.

```
$ oc exec sample-pod -- cat /sys/fs/cgroup/cpuset/cpuset.cpus
```

13.7.4. Additional resources

- Configuring an SR-IOV Ethernet network attachment
- Configuring an SR-IOV InfiniBand network attachment
- Using CPU Manager

13.8. USING HIGH PERFORMANCE MULTICAST

You can use multicast on your Single Root I/O Virtualization (SR-IOV) hardware network.

13.8.1. Configuring high performance multicast

The OpenShift SDN default Container Network Interface (CNI) network provider supports multicast between pods on the default network. This is best used for low-bandwidth coordination or service discovery, and not high-bandwidth applications. For applications such as streaming media, like Internet Protocol television (IPTV) and multipoint videoconferencing, you can utilize Single Root I/O Virtualization (SR-IOV) hardware to provide near-native performance.

When using additional SR-IOV interfaces for multicast:

- Multicast packages must be sent or received by a pod through the additional SR-IOV interface.
- The physical network which connects the SR-IOV interfaces decides the multicast routing and topology, which is not controlled by OpenShift Container Platform.

13.8.2. Using an SR-IOV interface for multicast

The follow procedure creates an example SR-IOV interface for multicast.

Prerequisites

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

Procedure

1. Create a **SriovNetworkNodePolicy** object:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: policy-example
  namespace: openshift-sriov-network-operator
spec:
  resourceName: example
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  numVfs: 4
  nicSelector:
    vendor: "8086"
    pfNames: ['ens803f0']
    rootDevices: ['0000:86:00.0']
```

2. Create a **SriovNetwork** object:

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: net-example
  namespace: openshift-sriov-network-operator
spec:
  networkNamespace: default
  ipam: | 1
    {
      "type": "host-local", 2
      "subnet": "10.56.217.0/24",
      "rangeStart": "10.56.217.171",
      "rangeEnd": "10.56.217.181",
      "routes": [
        {"dst": "224.0.0.0/5"},
        {"dst": "232.0.0.0/5"}
      ],
```

```
    "gateway": "10.56.217.1"
  }
resourceName: example
```

1 2 If you choose to configure DHCP as IPAM, ensure that you provision the following default routes through your DHCP server: **224.0.0.0/5** and **232.0.0.0/5**. This is to override the static multicast route set by the default network provider.

3. Create a pod with multicast application:

```
apiVersion: v1
kind: Pod
metadata:
  name: testpmd
  namespace: default
  annotations:
    k8s.v1.cni.cncf.io/networks: nic1
spec:
  containers:
  - name: example
    image: rhel7:latest
    securityContext:
      capabilities:
        add: ["NET_ADMIN"] 1
    command: [ "sleep", "infinity"]
```

1 The **NET_ADMIN** capability is required only if your application needs to assign the multicast IP address to the SR-IOV interface. Otherwise, it can be omitted.

13.9. USING VIRTUAL FUNCTIONS (VFS) WITH DPDK AND RDMA MODES

You can use Single Root I/O Virtualization (SR-IOV) network hardware with the Data Plane Development Kit (DPDK) and with remote direct memory access (RDMA).

IMPORTANT

The Data Plane Development Kit (DPDK) 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 https://access.redhat.com/support/offerings/techpreview/.

13.9.1. Using a virtual function in DPDK mode with an Intel NIC

Prerequisites

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

- Install the SR-IOV Network Operator.
- Log in as a user with **cluster-admin** privileges.

Procedure

1. Create the following **SriovNetworkNodePolicy** object, and then save the YAML in the **intel-dpdk-node-policy.yaml** file.

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: intel-dpdk-node-policy
  namespace: openshift-sriov-network-operator
spec:
  resourceName: intelnics
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  priority: <priority>
  numVfs: <num>
  nicSelector:
    vendor: "8086"
    deviceID: "158b"
    pfNames: ["<pf_name>", ...]
    rootDevices: ["<pci_bus_id>", "..."]
  deviceType: vfio-pci 1
```

1 Specify the driver type for the virtual functions to **vfio-pci**.

> **NOTE**
>
> See the **Configuring SR-IOV network devices** section for a detailed explanation on each option in **SriovNetworkNodePolicy**.
>
> When applying the configuration specified in a **SriovNetworkNodePolicy** object, the SR-IOV Operator may drain the nodes, and in some cases, reboot nodes. It may take several minutes for a configuration change to apply. Ensure that there are enough available nodes in your cluster to handle the evicted workload beforehand.
>
> After the configuration update is applied, all the pods in **openshift-sriov-network-operator** namespace will change to a **Running** status.

2. Create the **SriovNetworkNodePolicy** object by running the following command:

```
$ oc create -f intel-dpdk-node-policy.yaml
```

3. Create the following **SriovNetwork** object, and then save the YAML in the **intel-dpdk-network.yaml** file.

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
```

```
  name: intel-dpdk-network
  namespace: openshift-sriov-network-operator
spec:
  networkNamespace: <target_namespace>
  ipam: "{}" 1
  vlan: <vlan>
  resourceName: intelnics
```

1. Specify an empty object **"{}"** for the ipam CNI plug-in. DPDK works in userspace mode and does not require an IP address.

> **NOTE**
>
> See the **Configuring SR-IOV additional network** section for a detailed explanation on each option in **SriovNetwork**.

4. Create the **SriovNetwork** object by running the following command:

```
$ oc create -f intel-dpdk-network.yaml
```

5. Create the following **Pod** spec, and then save the YAML in the **intel-dpdk-pod.yaml** file.

```
apiVersion: v1
kind: Pod
metadata:
  name: dpdk-app
  namespace: <target_namespace> 1
  annotations:
    k8s.v1.cni.cncf.io/networks: intel-dpdk-network
spec:
  containers:
  - name: testpmd
    image: <DPDK_image> 2
    securityContext:
     capabilities:
       add: ["IPC_LOCK"] 3
    volumeMounts:
    - mountPath: /dev/hugepages 4
      name: hugepage
    resources:
      limits:
        openshift.io/intelnics: "1" 5
        memory: "1Gi"
        cpu: "4" 6
        hugepages-1Gi: "4Gi" 7
      requests:
        openshift.io/intelnics: "1"
        memory: "1Gi"
        cpu: "4"
        hugepages-1Gi: "4Gi"
    command: ["sleep", "infinity"]
  volumes:
```

```
- name: hugepage
  emptyDir:
    medium: HugePages
```

1. Specify the same **target_namespace** where the **SriovNetwork** object **intel-dpdk-network** is created. If you would like to create the pod in a different namespace, change **target_namespace** in both the **Pod** spec and the **SriovNetowrk** object.
2. Specify the DPDK image which includes your application and the DPDK library used by application.
3. Specify the **IPC_LOCK** capability which is required by the application to allocate hugepage memory inside container.
4. Mount a hugepage volume to the DPDK pod under **/dev/hugepages**. The hugepage volume is backed by the emptyDir volume type with the medium being **Hugepages**.
5. Optional: Specify the number of DPDK devices allocated to DPDK pod. This resource request and limit, if not explicitly specified, will be automatically added by the SR-IOV network resource injector. The SR-IOV network resource injector is an admission controller component managed by the SR-IOV Operator. It is enabled by default and can be disabled by setting **enableInjector** option to **false** in the default **SriovOperatorConfig** CR.
6. Specify the number of CPUs. The DPDK pod usually requires exclusive CPUs to be allocated from the kubelet. This is achieved by setting CPU Manager policy to **static** and creating a pod with **Guaranteed** QoS.
7. Specify hugepage size **hugepages-1Gi** or **hugepages-2Mi** and the quantity of hugepages that will be allocated to the DPDK pod. Configure **2Mi** and **1Gi** hugepages separately. Configuring **1Gi** hugepage requires adding kernel arguments to Nodes. For example, adding kernel arguments **default_hugepagesz=1GB**, **hugepagesz=1G** and **hugepages=16** will result in **16*1Gi** hugepages be allocated during system boot.

6. Create the DPDK pod by running the following command:

```
$ oc create -f intel-dpdk-pod.yaml
```

13.9.2. Using a virtual function in DPDK mode with a Mellanox NIC

Prerequisites

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

Procedure

1. Create the following **SriovNetworkNodePolicy** object, and then save the YAML in the **mlx-dpdk-node-policy.yaml** file.

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
```

```
metadata:
  name: mlx-dpdk-node-policy
  namespace: openshift-sriov-network-operator
spec:
  resourceName: mlxnics
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  priority: <priority>
  numVfs: <num>
  nicSelector:
    vendor: "15b3"
    deviceID: "1015" (1)
    pfNames: ["<pf_name>", ...]
    rootDevices: ["<pci_bus_id>", "..."]
  deviceType: netdevice (2)
  isRdma: true (3)
```

1. Specify the device hex code of the SR-IOV network device. The only allowed values for Mellanox cards are **1015**, **1017**.

2. Specify the driver type for the virtual functions to **netdevice**. Mellanox SR-IOV VF can work in DPDK mode without using the **vfio-pci** device type. VF device appears as a kernel network interface inside a container.

3. Enable RDMA mode. This is required by Mellanox cards to work in DPDK mode.

> **NOTE**
>
> See the **Configuring SR-IOV network devices** section for detailed explanation on each option in **SriovNetworkNodePolicy**.
>
> When applying the configuration specified in a **SriovNetworkNodePolicy** object, the SR-IOV Operator may drain the nodes, and in some cases, reboot nodes. It may take several minutes for a configuration change to apply. Ensure that there are enough available nodes in your cluster to handle the evicted workload beforehand.
>
> After the configuration update is applied, all the pods in the **openshift-sriov-network-operator** namespace will change to a **Running** status.

2. Create the **SriovNetworkNodePolicy** object by running the following command:

```
$ oc create -f mlx-dpdk-node-policy.yaml
```

3. Create the following **SriovNetwork** object, and then save the YAML in the **mlx-dpdk-network.yaml** file.

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: mlx-dpdk-network
  namespace: openshift-sriov-network-operator
spec:
  networkNamespace: <target_namespace>
```

```
ipam: |- (1)
  ...
vlan: <vlan>
resourceName: mlxnics
```

(1) Specify a configuration object for the ipam CNI plug-in as a YAML block scalar. The plug-in manages IP address assignment for the attachment definition.

> NOTE
>
> See the **Configuring SR-IOV additional network** section for detailed explanation on each option in **SriovNetwork**.

4. Create the **SriovNetworkNodePolicy** object by running the following command:

```
$ oc create -f mlx-dpdk-network.yaml
```

5. Create the following **Pod** spec, and then save the YAML in the **mlx-dpdk-pod.yaml** file.

```
apiVersion: v1
kind: Pod
metadata:
  name: dpdk-app
  namespace: <target_namespace> (1)
  annotations:
    k8s.v1.cni.cncf.io/networks: mlx-dpdk-network
spec:
  containers:
  - name: testpmd
    image: <DPDK_image> (2)
    securityContext:
     capabilities:
        add: ["IPC_LOCK","NET_RAW"] (3)
    volumeMounts:
    - mountPath: /dev/hugepages (4)
      name: hugepage
    resources:
      limits:
        openshift.io/mlxnics: "1" (5)
        memory: "1Gi"
        cpu: "4" (6)
        hugepages-1Gi: "4Gi" (7)
      requests:
        openshift.io/mlxnics: "1"
        memory: "1Gi"
        cpu: "4"
        hugepages-1Gi: "4Gi"
    command: ["sleep", "infinity"]
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
```

1. Specify the same **target_namespace** where **SriovNetwork** object **mlx-dpdk-network** is created. If you would like to create the pod in a different namespace, change **target_namespace** in both **Pod** spec and **SriovNetowrk** object.
2. Specify the DPDK image which includes your application and the DPDK library used by application.
3. Specify the **IPC_LOCK** capability which is required by the application to allocate hugepage memory inside the container and **NET_RAW** for the application to access the network interface.
4. Mount the hugepage volume to the DPDK pod under **/dev/hugepages**. The hugepage volume is backed by the emptyDir volume type with the medium being **Hugepages**.
5. Optional: Specify the number of DPDK devices allocated to the DPDK pod. This resource request and limit, if not explicitly specified, will be automatically added by SR-IOV network resource injector. The SR-IOV network resource injector is an admission controller component managed by SR-IOV Operator. It is enabled by default and can be disabled by setting the **enableInjector** option to **false** in the default **SriovOperatorConfig** CR.
6. Specify the number of CPUs. The DPDK pod usually requires exclusive CPUs be allocated from kubelet. This is achieved by setting CPU Manager policy to **static** and creating a pod with **Guaranteed** QoS.
7. Specify hugepage size **hugepages-1Gi** or **hugepages-2Mi** and the quantity of hugepages that will be allocated to DPDK pod. Configure **2Mi** and **1Gi** hugepages separately. Configuring **1Gi** hugepage requires adding kernel arguments to Nodes.

6. Create the DPDK pod by running the following command:

```
$ oc create -f mlx-dpdk-pod.yaml
```

13.9.3. Using a virtual function in RDMA mode with a Mellanox NIC

RDMA over Converged Ethernet (RoCE) is the only supported mode when using RDMA on OpenShift Container Platform.

Prerequisites

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

Procedure

1. Create the following **SriovNetworkNodePolicy** object, and then save the YAML in the **mlx-rdma-node-policy.yaml** file.

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: mlx-rdma-node-policy
```

```
  namespace: openshift-sriov-network-operator
spec:
  resourceName: mlxnics
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  priority: <priority>
  numVfs: <num>
  nicSelector:
    vendor: "15b3"
    deviceID: "1015" 1
    pfNames: ["<pf_name>", ...]
    rootDevices: ["<pci_bus_id>", "..."]
  deviceType: netdevice 2
  isRdma: true 3
```

1. Specify the device hex code of SR-IOV network device. The only allowed values for Mellanox cards are **1015**, **1017**.
2. Specify the driver type for the virtual functions to **netdevice**.
3. Enable RDMA mode.

NOTE

See the **Configuring SR-IOV network devices** section for a detailed explanation on each option in **SriovNetworkNodePolicy**.

When applying the configuration specified in a **SriovNetworkNodePolicy** object, the SR-IOV Operator may drain the nodes, and in some cases, reboot nodes. It may take several minutes for a configuration change to apply. Ensure that there are enough available nodes in your cluster to handle the evicted workload beforehand.

After the configuration update is applied, all the pods in the **openshift-sriov-network-operator** namespace will change to a **Running** status.

2. Create the **SriovNetworkNodePolicy** object by running the following command:

```
$ oc create -f mlx-rdma-node-policy.yaml
```

3. Create the following **SriovNetwork** object, and then save the YAML in the **mlx-rdma-network.yaml** file.

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: mlx-rdma-network
  namespace: openshift-sriov-network-operator
spec:
  networkNamespace: <target_namespace>
  ipam: |- 1
```

```
  ...
vlan: <vlan>
resourceName: mlxnics
```

1. Specify a configuration object for the ipam CNI plug-in as a YAML block scalar. The plug-in manages IP address assignment for the attachment definition.

> **NOTE**
>
> See the **Configuring SR-IOV additional network** section for detailed explanation on each option in **SriovNetwork**.

4. Create the **SriovNetworkNodePolicy** object by running the following command:

```
$ oc create -f mlx-rdma-network.yaml
```

5. Create the following **Pod** spec, and then save the YAML in the **mlx-rdma-pod.yaml** file.

```
apiVersion: v1
kind: Pod
metadata:
  name: rdma-app
  namespace: <target_namespace> 1
  annotations:
    k8s.v1.cni.cncf.io/networks: mlx-rdma-network
spec:
  containers:
  - name: testpmd
    image: <RDMA_image> 2
    securityContext:
     capabilities:
       add: ["IPC_LOCK"] 3
    volumeMounts:
    - mountPath: /dev/hugepages 4
      name: hugepage
    resources:
      limits:
        memory: "1Gi"
        cpu: "4" 5
        hugepages-1Gi: "4Gi" 6
      requests:
        memory: "1Gi"
        cpu: "4"
        hugepages-1Gi: "4Gi"
    command: ["sleep", "infinity"]
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
```

1. Specify the same **target_namespace** where **SriovNetwork** object **mlx-rdma-network** is created. If you would like to create the pod in a different namespace, change **target_namespace** in both **Pod** spec and **SriovNetowrk** object.

2. Specify the RDMA image which includes your application and RDMA library used by application.

3. Specify the **IPC_LOCK** capability which is required by the application to allocate hugepage memory inside the container.

4. Mount the hugepage volume to RDMA pod under **/dev/hugepages**. The hugepage volume is backed by the emptyDir volume type with the medium being **Hugepages**.

5. Specify number of CPUs. The RDMA pod usually requires exclusive CPUs be allocated from the kubelet. This is achieved by setting CPU Manager policy to **static** and create pod with **Guaranteed** QoS.

6. Specify hugepage size **hugepages-1Gi** or **hugepages-2Mi** and the quantity of hugepages that will be allocated to the RDMA pod. Configure **2Mi** and **1Gi** hugepages separately. Configuring **1Gi** hugepage requires adding kernel arguments to Nodes.

6. Create the RDMA pod by running the following command:

```
$ oc create -f mlx-rdma-pod.yaml
```

CHAPTER 14. OPENSHIFT SDN DEFAULT CNI NETWORK PROVIDER

14.1. ABOUT THE OPENSHIFT SDN DEFAULT CNI NETWORK PROVIDER

OpenShift Container Platform uses a software-defined networking (SDN) approach to provide a unified cluster network that enables communication between pods across the OpenShift Container Platform cluster. This pod network is established and maintained by the OpenShift SDN, which configures an overlay network using Open vSwitch (OVS).

14.1.1. OpenShift SDN network isolation modes

OpenShift SDN provides three SDN modes for configuring the pod network:

- *Network policy* mode allows project administrators to configure their own isolation policies using **NetworkPolicy** objects. Network policy is the default mode in OpenShift Container Platform 4.8.
- *Multitenant* mode provides project-level isolation for pods and services. Pods from different projects cannot send packets to or receive packets from pods and services of a different project. You can disable isolation for a project, allowing it to send network traffic to all pods and services in the entire cluster and receive network traffic from those pods and services.
- *Subnet* mode provides a flat pod network where every pod can communicate with every other pod and service. The network policy mode provides the same functionality as subnet mode.

14.1.2. Supported default CNI network provider feature matrix

OpenShift Container Platform offers two supported choices, OpenShift SDN and OVN-Kubernetes, for the default Container Network Interface (CNI) network provider. The following table summarizes the current feature support for both network providers:

Table 14.1. Default CNI network provider feature comparison

Feature	OpenShift SDN	OVN-Kubernetes
Egress IPs	Supported	Supported
Egress firewall [1]	Supported	Supported
Egress router	Supported	Supported [2]
IPsec encryption	Not supported	Supported
IPv6	Not supported	Supported [3]
Kubernetes network policy	Partially supported [4]	Supported
Kubernetes network policy logs	Not supported	Supported

Feature	OpenShift SDN	OVN-Kubernetes
Multicast	Supported	Supported

1. Egress firewall is also known as egress network policy in OpenShift SDN. This is not the same as network policy egress.

2. Egress router for OVN-Kubernetes supports only redirect mode.

3. IPv6 is supported only on bare metal clusters.

4. Network policy for OpenShift SDN does not support egress rules and some **ipBlock** rules.

14.2. CONFIGURING EGRESS IPS FOR A PROJECT

As a cluster administrator, you can configure the OpenShift SDN default Container Network Interface (CNI) network provider to assign one or more egress IP addresses to a project.

14.2.1. Egress IP address assignment for project egress traffic

By configuring an egress IP address for a project, all outgoing external connections from the specified project will share the same, fixed source IP address. External resources can recognize traffic from a particular project based on the egress IP address. An egress IP address assigned to a project is different from the egress router, which is used to send traffic to specific destinations.

Egress IP addresses are implemented as additional IP addresses on the primary network interface of the node and must be in the same subnet as the node's primary IP address.

IMPORTANT

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

Allowing additional IP addresses on the primary network interface might require extra configuration when using some cloud or VM solutions.

You can assign egress IP addresses to namespaces by setting the **egressIPs** parameter of the **NetNamespace** object. After an egress IP is associated with a project, OpenShift SDN allows you to assign egress IPs to hosts in two ways:

- In the *automatically assigned* approach, an egress IP address range is assigned to a node.

- In the *manually assigned* approach, a list of one or more egress IP address is assigned to a node.

Namespaces that request an egress IP address are matched with nodes that can host those egress IP addresses, and then the egress IP addresses are assigned to those nodes. If the **egressIPs** parameter is set on a **NetNamespace** object, but no node hosts that egress IP address, then egress traffic from the namespace will be dropped.

High availability of nodes is automatic. If a node that hosts an egress IP address is unreachable and there are nodes that are able to host that egress IP address, then the egress IP address will move to a new node. When the unreachable node comes back online, the egress IP address automatically moves to balance egress IP addresses across nodes.

IMPORTANT

The following limitations apply when using egress IP addresses with the OpenShift SDN cluster network provider:

- You cannot use manually assigned and automatically assigned egress IP addresses on the same nodes.
- If you manually assign egress IP addresses from an IP address range, you must not make that range available for automatic IP assignment.
- You cannot share egress IP addresses across multiple namespaces using the OpenShift SDN egress IP address implementation. If you need to share IP addresses across namespaces, the OVN-Kubernetes cluster network provider egress IP address implementation allows you to span IP addresses across multiple namespaces.

NOTE

If you use OpenShift SDN in multitenant mode, you cannot use egress IP addresses with any namespace that is joined to another namespace by the projects that are associated with them. For example, if **project1** and **project2** are joined by running the **oc adm pod-network join-projects --to=project1 project2** command, neither project can use an egress IP address. For more information, see BZ#1645577.

14.2.1.1. Considerations when using automatically assigned egress IP addresses

When using the automatic assignment approach for egress IP addresses the following considerations apply:

- You set the **egressCIDRs** parameter of each node's **HostSubnet** resource to indicate the range of egress IP addresses that can be hosted by a node. OpenShift Container Platform sets the **egressIPs** parameter of the **HostSubnet** resource based on the IP address range you specify.

If the node hosting the namespace's egress IP address is unreachable, OpenShift Container Platform will reassign the egress IP address to another node with a compatible egress IP address range. The automatic assignment approach works best for clusters installed in environments with flexibility in associating additional IP addresses with nodes.

14.2.1.2. Considerations when using manually assigned egress IP addresses

This approach is recommended for clusters installed in public cloud environments, where there can be limitations on associating additional IP addresses with nodes.

When using the manual assignment approach for egress IP addresses the following considerations apply:

- You set the **egressIPs** parameter of each node's **HostSubnet** resource to indicate the IP addresses that can be hosted by a node.
- Multiple egress IP addresses per namespace are supported.

If a namespace has multiple egress IP addresses and those addresses are hosted on multiple nodes, the following additional considerations apply:

- If a pod is on a node that is hosting an egress IP address, that pod always uses the egress IP address on the node.
- If a pod is not on a node that is hosting an egress IP address, that pod uses an egress IP address at random.

14.2.2. Configuring automatically assigned egress IP addresses for a namespace

In OpenShift Container Platform you can enable automatic assignment of an egress IP address for a specific namespace across one or more nodes.

Prerequisites

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

Procedure

1. Update the **NetNamespace** object with the egress IP address using the following JSON:

```
$ oc patch netnamespace <project_name> --type=merge -p \
 '{
   "egressIPs": [
     "<ip_address>"
   ]
 }'
```

where:

<project_name>
: Specifies the name of the project.

<ip_address>
: Specifies one or more egress IP addresses for the **egressIPs** array.

For example, to assign **project1** to an IP address of 192.168.1.100 and **project2** to an IP address of 192.168.1.101:

```
$ oc patch netnamespace project1 --type=merge -p \
  '{"egressIPs": ["192.168.1.100"]}'
$ oc patch netnamespace project2 --type=merge -p \
  '{"egressIPs": ["192.168.1.101"]}'
```

> **NOTE**
>
> Because OpenShift SDN manages the **NetNamespace** object, you can make changes only by modifying the existing **NetNamespace** object. Do not create a new **NetNamespace** object.

2. Indicate which nodes can host egress IP addresses by setting the **egressCIDRs** parameter for each host using the following JSON:

```
$ oc patch hostsubnet <node_name> --type=merge -p \
```

```
'{
  "egressCIDRs": [
    "<ip_address_range>", "<ip_address_range>"
  ]
}'
```

where:

<node_name>

Specifies a node name.

<ip_address_range>

Specifies an IP address range in CIDR format. You can specify more than one address range for the **egressCIDRs** array.

For example, to set **node1** and **node2** to host egress IP addresses in the range 192.168.1.0 to 192.168.1.255:

```
$ oc patch hostsubnet node1 --type=merge -p \
  '{"egressCIDRs": ["192.168.1.0/24"]}'
$ oc patch hostsubnet node2 --type=merge -p \
  '{"egressCIDRs": ["192.168.1.0/24"]}'
```

OpenShift Container Platform automatically assigns specific egress IP addresses to available nodes in a balanced way. In this case, it assigns the egress IP address 192.168.1.100 to **node1** and the egress IP address 192.168.1.101 to **node2** or vice versa.

14.2.3. Configuring manually assigned egress IP addresses for a namespace

In OpenShift Container Platform you can associate one or more egress IP addresses with a namespace.

Prerequisites

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

Procedure

1. Update the **NetNamespace** object by specifying the following JSON object with the desired IP addresses:

```
$ oc patch netnamespace <project_name> --type=merge -p \
  '{
    "egressIPs": [
      "<ip_address>"
    ]
  }'
```

where:

<project_name>

Specifies the name of the project.

<ip_address>

Specifies one or more egress IP addresses for the **egressIPs** array.

For example, to assign the **project1** project to the IP addresses **192.168.1.100** and **192.168.1.101**:

```
$ oc patch netnamespace project1 --type=merge \
  -p '{"egressIPs": ["192.168.1.100","192.168.1.101"]}'
```

To provide high availability, set the **egressIPs** value to two or more IP addresses on different nodes. If multiple egress IP addresses are set, then pods use all egress IP addresses roughly equally.

> **NOTE**
>
> Because OpenShift SDN manages the **NetNamespace** object, you can make changes only by modifying the existing **NetNamespace** object. Do not create a new **NetNamespace** object.

2. Manually assign the egress IP to the node hosts. Set the **egressIPs** parameter on the **HostSubnet** object on the node host. Using the following JSON, include as many IP addresses as you want to assign to that node host:

```
$ oc patch hostsubnet <node_name> --type=merge -p \
  '{
    "egressIPs": [
      "<ip_address>",
      "<ip_address>"
      ]
  }'
```

where:

<node_name>
Specifies a node name.

<ip_address>
Specifies an IP address. You can specify more than one IP address for the **egressIPs** array.

For example, to specify that **node1** should have the egress IPs **192.168.1.100**, **192.168.1.101**, and **192.168.1.102**:

```
$ oc patch hostsubnet node1 --type=merge -p \
  '{"egressIPs": ["192.168.1.100", "192.168.1.101", "192.168.1.102"]}'
```

In the previous example, all egress traffic for **project1** will be routed to the node hosting the specified egress IP, and then connected through Network Address Translation (NAT) to that IP address.

14.3. CONFIGURING AN EGRESS FIREWALL FOR A PROJECT

As a cluster administrator, you can create an egress firewall for a project that restricts egress traffic leaving your OpenShift Container Platform cluster.

14.3.1. How an egress firewall works in a project

As a cluster administrator, you can use an *egress firewall* to limit the external hosts that some or all pods can access from within the cluster. An egress firewall supports the following scenarios:

- A pod can only connect to internal hosts and cannot initiate connections to the public internet.
- A pod can only connect to the public internet and cannot initiate connections to internal hosts that are outside the OpenShift Container Platform cluster.
- A pod cannot reach specified internal subnets or hosts outside the OpenShift Container Platform cluster.
- A pod can connect to only specific external hosts.

For example, you can allow one project access to a specified IP range but deny the same access to a different project. Or you can restrict application developers from updating from Python pip mirrors, and force updates to come only from approved sources.

You configure an egress firewall policy by creating an EgressNetworkPolicy custom resource (CR) object. The egress firewall matches network traffic that meets any of the following criteria:

- An IP address range in CIDR format
- A DNS name that resolves to an IP address

IMPORTANT

You must have OpenShift SDN configured to use either the network policy or multitenant mode to configure an egress firewall.

If you use network policy mode, an egress firewall is compatible with only one policy per namespace and will not work with projects that share a network, such as global projects.

WARNING

Egress firewall rules do not apply to traffic that goes through routers. Any user with permission to create a Route CR object can bypass egress firewall policy rules by creating a route that points to a forbidden destination.

14.3.1.1. Limitations of an egress firewall

An egress firewall has the following limitations:

- No project can have more than one EgressNetworkPolicy object.
- A maximum of one EgressNetworkPolicy object with a maximum of 1,000 rules can be defined per project.
- The **default** project cannot use an egress firewall.

- When using the OpenShift SDN default Container Network Interface (CNI) network provider in multitenant mode, the following limitations apply:
    - Global projects cannot use an egress firewall. You can make a project global by using the **oc adm pod-network make-projects-global** command.
    - Projects merged by using the **oc adm pod-network join-projects** command cannot use an egress firewall in any of the joined projects.

Violating any of these restrictions results in a broken egress firewall for the project, and might cause all external network traffic to be dropped.

14.3.1.2. Matching order for egress firewall policy rules

The egress firewall policy rules are evaluated in the order that they are defined, from first to last. The first rule that matches an egress connection from a pod applies. Any subsequent rules are ignored for that connection.

14.3.1.3. How Domain Name Server (DNS) resolution works

If you use DNS names in any of your egress firewall policy rules, proper resolution of the domain names is subject to the following restrictions:

- Domain name updates are polled based on a time-to-live (TTL) duration. By default, the duration is 30 seconds. When the egress firewall controller queries the local name servers for a domain name, if the response includes a TTL that is less than 30 seconds, the controller sets the duration to the returned value. If the TTL in the response is greater than 30 minutes, the controller sets the duration to 30 minutes. If the TTL is between 30 seconds and 30 minutes, the controller ignores the value and sets the duration to 30 seconds.
- The pod must resolve the domain from the same local name servers when necessary. Otherwise the IP addresses for the domain known by the egress firewall controller and the pod can be different. If the IP addresses for a hostname differ, the egress firewall might not be enforced consistently.
- Because the egress firewall controller and pods asynchronously poll the same local name server, the pod might obtain the updated IP address before the egress controller does, which causes a race condition. Due to this current limitation, domain name usage in EgressNetworkPolicy objects is only recommended for domains with infrequent IP address changes.

> **NOTE**
>
> The egress firewall always allows pods access to the external interface of the node that the pod is on for DNS resolution.
>
> If you use domain names in your egress firewall policy and your DNS resolution is not handled by a DNS server on the local node, then you must add egress firewall rules that allow access to your DNS server's IP addresses. if you are using domain names in your pods.

14.3.2. EgressNetworkPolicy custom resource (CR) object

You can define one or more rules for an egress firewall. A rule is either an **Allow** rule or a **Deny** rule, with a specification for the traffic that the rule applies to.

The following YAML describes an EgressNetworkPolicy CR object:

EgressNetworkPolicy object

```
apiVersion: network.openshift.io/v1
kind: EgressNetworkPolicy
metadata:
  name: <name> 1
spec:
  egress: 2
    ...
```

1. A name for your egress firewall policy.
2. A collection of one or more egress network policy rules as described in the following section.

14.3.2.1. EgressNetworkPolicy rules

The following YAML describes an egress firewall rule object. The **egress** stanza expects an array of one or more objects.

Egress policy rule stanza

```
egress:
- type: <type> 1
  to: 2
    cidrSelector: <cidr> 3
    dnsName: <dns_name> 4
```

1. The type of rule. The value must be either **Allow** or **Deny**.
2. A stanza describing an egress traffic match rule. A value for either the **cidrSelector** field or the **dnsName** field for the rule. You cannot use both fields in the same rule.
3. An IP address range in CIDR format.
4. A domain name.

14.3.2.2. Example EgressNetworkPolicy CR objects

The following example defines several egress firewall policy rules:

```
apiVersion: network.openshift.io/v1
kind: EgressNetworkPolicy
metadata:
  name: default
spec:
  egress: 1
  - type: Allow
    to:
      cidrSelector: 1.2.3.0/24
  - type: Allow
    to:
```

```
    dnsName: www.example.com
- type: Deny
  to:
    cidrSelector: 0.0.0.0/0
```

1 A collection of egress firewall policy rule objects.

14.3.3. Creating an egress firewall policy object

As a cluster administrator, you can create an egress firewall policy object for a project.

> **IMPORTANT**
>
> If the project already has an EgressNetworkPolicy object defined, you must edit the existing policy to make changes to the egress firewall rules.

Prerequisites

- A cluster that uses the OpenShift SDN default Container Network Interface (CNI) network provider plug-in.
- Install the OpenShift CLI (**oc**).
- You must log in to the cluster as a cluster administrator.

Procedure

1. Create a policy rule:
   a. Create a **<policy_name>.yaml** file where **<policy_name>** describes the egress policy rules.
   b. In the file you created, define an egress policy object.
2. Enter the following command to create the policy object. Replace **<policy_name>** with the name of the policy and **<project>** with the project that the rule applies to.

   ```
   $ oc create -f <policy_name>.yaml -n <project>
   ```

   In the following example, a new EgressNetworkPolicy object is created in a project named **project1**:

   ```
   $ oc create -f default.yaml -n project1
   ```

   Example output

   ```
   egressnetworkpolicy.network.openshift.io/v1 created
   ```

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

14.4. EDITING AN EGRESS FIREWALL FOR A PROJECT

As a cluster administrator, you can modify network traffic rules for an existing egress firewall.

14.4.1. Viewing an EgressNetworkPolicy object

You can view an EgressNetworkPolicy object in your cluster.

Prerequisites

- A cluster using the OpenShift SDN default Container Network Interface (CNI) network provider plug-in.
- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster.

Procedure

1. Optional: To view the names of the EgressNetworkPolicy objects defined in your cluster, enter the following command:

```
$ oc get egressnetworkpolicy --all-namespaces
```

2. To inspect a policy, enter the following command. Replace **<policy_name>** with the name of the policy to inspect.

```
$ oc describe egressnetworkpolicy <policy_name>
```

Example output

```
Name:  default
Namespace: project1
Created: 20 minutes ago
Labels:  <none>
Annotations: <none>
Rule:  Allow to 1.2.3.0/24
Rule:  Allow to www.example.com
Rule:  Deny to 0.0.0.0/0
```

14.5. EDITING AN EGRESS FIREWALL FOR A PROJECT

As a cluster administrator, you can modify network traffic rules for an existing egress firewall.

14.5.1. Editing an EgressNetworkPolicy object

As a cluster administrator, you can update the egress firewall for a project.

Prerequisites

- A cluster using the OpenShift SDN default Container Network Interface (CNI) network provider plug-in.
- Install the OpenShift CLI (**oc**).
- You must log in to the cluster as a cluster administrator.

Procedure

1. Find the name of the EgressNetworkPolicy object for the project. Replace **<project>** with the name of the project.

```
$ oc get -n <project> egressnetworkpolicy
```

2. Optional: If you did not save a copy of the EgressNetworkPolicy object when you created the egress network firewall, enter the following command to create a copy.

```
$ oc get -n <project> egressnetworkpolicy <name> -o yaml > <filename>.yaml
```

Replace **<project>** with the name of the project. Replace **<name>** with the name of the object. Replace **<filename>** with the name of the file to save the YAML to.

3. After making changes to the policy rules, enter the following command to replace the EgressNetworkPolicy object. Replace **<filename>** with the name of the file containing the updated EgressNetworkPolicy object.

```
$ oc replace -f <filename>.yaml
```

14.6. REMOVING AN EGRESS FIREWALL FROM A PROJECT

As a cluster administrator, you can remove an egress firewall from a project to remove all restrictions on network traffic from the project that leaves the OpenShift Container Platform cluster.

14.6.1. Removing an EgressNetworkPolicy object

As a cluster administrator, you can remove an egress firewall from a project.

Prerequisites

- A cluster using the OpenShift SDN default Container Network Interface (CNI) network provider plug-in.
- Install the OpenShift CLI (**oc**).
- You must log in to the cluster as a cluster administrator.

Procedure

1. Find the name of the EgressNetworkPolicy object for the project. Replace **<project>** with the name of the project.

```
$ oc get -n <project> egressnetworkpolicy
```

2. Enter the following command to delete the EgressNetworkPolicy object. Replace **<project>** with the name of the project and **<name>** with the name of the object.

```
$ oc delete -n <project> egressnetworkpolicy <name>
```

14.7. CONSIDERATIONS FOR THE USE OF AN EGRESS ROUTER POD

14.7.1. 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.

14.7.1.1. 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 modified to connect to the egress router rather than connecting directly to the destination IP.

In *HTTP proxy mode*, an egress router pod runs as an HTTP proxy on port **8080**. This mode only works for clients that are connecting to HTTP-based or HTTPS-based services, but usually requires fewer changes to the client pods to get them to work. Many programs can be told to use an HTTP proxy by setting an environment variable.

In *DNS proxy mode*, an egress router pod runs as a DNS proxy for TCP-based services from its own IP address to one or more destination IP addresses. To make use of the reserved, source IP address, client pods must be modified to connect to the egress router pod rather than connecting directly to the destination IP address. This modification ensures that external destinations treat traffic as though it were coming from a known source.

Redirect mode works for all services except for HTTP and HTTPS. For HTTP and HTTPS services, use HTTP proxy mode. For TCP-based services with IP addresses or domain names, use DNS proxy mode.

14.7.1.2. Egress router pod implementation

The egress router pod setup is performed by an initialization container. That container runs in a privileged context so that it can configure the macvlan interface and set up **iptables** rules. After the initialization container finishes setting up the **iptables** rules, it exits. Next the egress router pod executes the container to handle the egress router traffic. The image used varies depending on the egress router mode.

The environment variables determine which addresses the egress-router image uses. The image configures the macvlan interface to use **EGRESS_SOURCE** as its IP address, with **EGRESS_GATEWAY** as the IP address for the gateway.

Network Address Translation (NAT) rules are set up so that connections to the cluster IP address of the pod on any TCP or UDP port are redirected to the same port on IP address specified by the **EGRESS_DESTINATION** variable.

If only some of the nodes in your cluster are capable of claiming the specified source IP address and using the specified gateway, you can specify a **nodeName** or **nodeSelector** to identify which nodes are acceptable.

14.7.1.3. 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>
```

Red Hat Virtualization (RHV)

If you are using RHV, you must select **No Network Filter** for the Virtual Network Interface Card (vNIC).

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

14.7.1.4. Failover configuration

To avoid downtime, you can deploy an egress router pod with a **Deployment** resource, as in the following example. To create a new **Service** object for the example deployment, use the **oc expose deployment/egress-demo-controller** command.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: egress-demo-controller
spec:
  replicas: 1 (1)
  selector:
    matchLabels:
      name: egress-router
  template:
```

```
metadata:
  name: egress-router
  labels:
    name: egress-router
  annotations:
    pod.network.openshift.io/assign-macvlan: "true"
spec: (2)
  initContainers:
    ...
  containers:
    ...
```

1. Ensure that replicas is set to **1**, because only one pod can use a given egress source IP address at any time. This means that only a single copy of the router runs on a node.
2. Specify the **Pod** object template for the egress router pod.

14.7.2. Additional resources

- Deploying an egress router in redirection mode
- Deploying an egress router in HTTP proxy mode
- Deploying an egress router in DNS proxy mode

14.8. DEPLOYING AN EGRESS ROUTER POD IN REDIRECT MODE

As a cluster administrator, you can deploy an egress router pod that is configured to redirect traffic to specified destination IP addresses.

14.8.1. Egress router pod specification for redirect mode

Define the configuration for an egress router pod in the **Pod** object. The following YAML describes the fields for the configuration of an egress router pod in redirect mode:

```
apiVersion: v1
kind: Pod
metadata:
  name: egress-1
  labels:
    name: egress-1
  annotations:
    pod.network.openshift.io/assign-macvlan: "true" (1)
spec:
  initContainers:
  - name: egress-router
    image: registry.redhat.io/openshift4/ose-egress-router
    securityContext:
      privileged: true
    env:
    - name: EGRESS_SOURCE (2)
      value: <egress_router>
    - name: EGRESS_GATEWAY (3)
```

```
      value: <egress_gateway>
    - name: EGRESS_DESTINATION (4)
      value: <egress_destination>
    - name: EGRESS_ROUTER_MODE
      value: init
  containers:
  - name: egress-router-wait
    image: registry.redhat.io/openshift4/ose-pod
```

1. Before starting the **egress-router** container, create a macvlan network interface on the primary network interface and move that interface into the pod network namespace. You must include the quotation marks around the **"true"** value. To create the macvlan interface on a network interface other than the primary one, set the annotation value to the name of that interface. For example, **eth1**.

2. IP address from the physical network that the node is on that is reserved for use by the egress router pod. Optional: You can include the subnet length, the **/24** suffix, so that a proper route to the local subnet is set. If you do not specify a subnet length, then the egress router can access only the host specified with the **EGRESS_GATEWAY** variable and no other hosts on the subnet.

3. Same value as the default gateway used by the node.

4. External server to direct traffic to. Using this example, connections to the pod are redirected to **203.0.113.25**, with a source IP address of **192.168.12.99**.

Example egress router Pod specification

```
apiVersion: v1
kind: Pod
metadata:
  name: egress-multi
  labels:
    name: egress-multi
  annotations:
    pod.network.openshift.io/assign-macvlan: "true"
spec:
  initContainers:
  - name: egress-router
    image: registry.redhat.io/openshift4/ose-egress-router
    securityContext:
      privileged: true
    env:
    - name: EGRESS_SOURCE
      value: 192.168.12.99/24
    - name: EGRESS_GATEWAY
      value: 192.168.12.1
    - name: EGRESS_DESTINATION
      value: |
        80   tcp 203.0.113.25
        8080 tcp 203.0.113.26 80
        8443 tcp 203.0.113.26 443
        203.0.113.27
    - name: EGRESS_ROUTER_MODE
      value: init
```

```
containers:
- name: egress-router-wait
  image: registry.redhat.io/openshift4/ose-pod
```

14.8.2. Egress destination configuration format

When an egress router pod is deployed in redirect mode, you can specify redirection rules by using one or more of the following formats:

- **<port> <protocol> <ip_address>** - Incoming connections to the given **<port>** should be redirected to the same port on the given **<ip_address>**. **<protocol>** is either **tcp** or **udp**.

- **<port> <protocol> <ip_address> <remote_port>** - As above, except that the connection is redirected to a different **<remote_port>** on **<ip_address>**.

- **<ip_address>** - If the last line is a single IP address, then any connections on any other port will be redirected to the corresponding port on that IP address. If there is no fallback IP address then connections on other ports are rejected.

In the example that follows several rules are defined:

- The first line redirects traffic from local port **80** to port **80** on **203.0.113.25**.

- The second and third lines redirect local ports **8080** and **8443** to remote ports **80** and **443** on **203.0.113.26**.

- The last line matches traffic for any ports not specified in the previous rules.

Example configuration

```
80   tcp 203.0.113.25
8080 tcp 203.0.113.26 80
8443 tcp 203.0.113.26 443
203.0.113.27
```

14.8.3. Deploying an egress router pod in redirect mode

In *redirect mode*, an egress router pod sets up 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 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 pod.

2. To ensure that other pods can find the IP address of the egress router pod, create a service to point to the egress router pod, as in the following example:

   ```
   apiVersion: v1
   ```

```yaml
kind: Service
metadata:
  name: egress-1
spec:
  ports:
  - name: http
    port: 80
  - name: https
    port: 443
  type: ClusterIP
  selector:
    name: egress-1
```

Your pods can now connect to this service. Their connections are redirected to the corresponding ports on the external server, using the reserved egress IP address.

14.8.4. Additional resources

- Configuring an egress router destination mappings with a ConfigMap

14.9. DEPLOYING AN EGRESS ROUTER POD IN HTTP PROXY MODE

As a cluster administrator, you can deploy an egress router pod configured to proxy traffic to specified HTTP and HTTPS-based services.

14.9.1. Egress router pod specification for HTTP mode

Define the configuration for an egress router pod in the **Pod** object. The following YAML describes the fields for the configuration of an egress router pod in HTTP mode:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: egress-1
  labels:
    name: egress-1
  annotations:
    pod.network.openshift.io/assign-macvlan: "true" (1)
spec:
  initContainers:
  - name: egress-router
    image: registry.redhat.io/openshift4/ose-egress-router
    securityContext:
      privileged: true
    env:
    - name: EGRESS_SOURCE (2)
      value: <egress-router>
    - name: EGRESS_GATEWAY (3)
      value: <egress-gateway>
    - name: EGRESS_ROUTER_MODE
      value: http-proxy
  containers:
  - name: egress-router-pod
    image: registry.redhat.io/openshift4/ose-egress-http-proxy
```

```
env:
- name: EGRESS_HTTP_PROXY_DESTINATION 4
  value: |-
    ...
...
```

1. Before starting the **egress-router** container, create a macvlan network interface on the primary network interface and move that interface into the pod network namespace. You must include the quotation marks around the **"true"** value. To create the macvlan interface on a network interface other than the primary one, set the annotation value to the name of that interface. For example, **eth1**.

2. IP address from the physical network that the node is on that is reserved for use by the egress router pod. Optional: You can include the subnet length, the **/24** suffix, so that a proper route to the local subnet is set. If you do not specify a subnet length, then the egress router can access only the host specified with the **EGRESS_GATEWAY** variable and no other hosts on the subnet.

3. Same value as the default gateway used by the node.

4. A string or YAML multi-line string specifying how to configure the proxy. Note that this is specified as an environment variable in the HTTP proxy container, not with the other environment variables in the init container.

14.9.2. Egress destination configuration format

When an egress router pod is deployed in HTTP proxy mode, you can specify redirection rules by using one or more of the following formats. Each line in the configuration specifies one group of connections to allow or deny:

- An IP address allows connections to that IP address, such as **192.168.1.1**.
- A CIDR range allows connections to that CIDR range, such as **192.168.1.0/24**.
- A hostname allows proxying to that host, such as **www.example.com**.
- A domain name preceded by ***.** allows proxying to that domain and all of its subdomains, such as ***.example.com**.
- A **!** followed by any of the previous match expressions denies the connection instead.
- If the last line is *****, then anything that is not explicitly denied is allowed. Otherwise, anything that is not allowed is denied.

You can also use * to allow connections to all remote destinations.

Example configuration

```
!*.example.com
!192.168.1.0/24
192.168.2.1
*
```

14.9.3. Deploying an egress router pod in HTTP proxy mode

In *HTTP proxy mode*, an egress router pod runs as an HTTP proxy on port **8080**. This mode only works for clients that are connecting to HTTP-based or HTTPS-based services, but usually requires fewer changes to the client pods to get them to work. Many programs can be told to use an HTTP proxy by setting an environment variable.

Prerequisites

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

Procedure

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

```
apiVersion: v1
kind: Service
metadata:
  name: egress-1
spec:
  ports:
  - name: http-proxy
    port: 8080 (1)
  type: ClusterIP
  selector:
    name: egress-1
```

(1) Ensure the **http** port is set to **8080**.

3. To configure the client pod (not the egress proxy pod) to use the HTTP proxy, set the **http_proxy** or **https_proxy** variables:

```
apiVersion: v1
kind: Pod
metadata:
  name: app-1
  labels:
    name: app-1
spec:
  containers:
    env:
    - name: http_proxy
      value: http://egress-1:8080/ (1)
    - name: https_proxy
      value: http://egress-1:8080/
    ...
```

(1) The service created in the previous step.

NOTE

Using the **http_proxy** and **https_proxy** environment variables is not necessary for all setups. If the above does not create a working setup, then consult the documentation for the tool or software you are running in the pod.

14.9.4. Additional resources

- Configuring an egress router destination mappings with a ConfigMap

14.10. DEPLOYING AN EGRESS ROUTER POD IN DNS PROXY MODE

As a cluster administrator, you can deploy an egress router pod configured to proxy traffic to specified DNS names and IP addresses.

14.10.1. Egress router pod specification for DNS mode

Define the configuration for an egress router pod in the **Pod** object. The following YAML describes the fields for the configuration of an egress router pod in DNS mode:

```
apiVersion: v1
kind: Pod
metadata:
  name: egress-1
  labels:
    name: egress-1
  annotations:
    pod.network.openshift.io/assign-macvlan: "true" (1)
spec:
  initContainers:
  - name: egress-router
    image: registry.redhat.io/openshift4/ose-egress-router
    securityContext:
      privileged: true
    env:
    - name: EGRESS_SOURCE (2)
      value: <egress-router>
    - name: EGRESS_GATEWAY (3)
      value: <egress-gateway>
    - name: EGRESS_ROUTER_MODE
      value: dns-proxy
  containers:
  - name: egress-router-pod
    image: registry.redhat.io/openshift4/ose-egress-dns-proxy
    securityContext:
      privileged: true
    env:
    - name: EGRESS_DNS_PROXY_DESTINATION (4)
      value: |-
        ...
    - name: EGRESS_DNS_PROXY_DEBUG (5)
      value: "1"
    ...
```

1. Before starting the **egress-router** container, create a macvlan network interface on the primary network interface and move that interface into the pod network namespace. You must include the

2. IP address from the physical network that the node is on that is reserved for use by the egress router pod. Optional: You can include the subnet length, the **/24** suffix, so that a proper route to the local subnet is set. If you do not specify a subnet length, then the egress router can access only the host specified with the **EGRESS_GATEWAY** variable and no other hosts on the subnet.

3. Same value as the default gateway used by the node.

4. Specify a list of one or more proxy destinations.

5. Optional: Specify to output the DNS proxy log output to **stdout**.

14.10.2. Egress destination configuration format

When the router is deployed in DNS proxy mode, you specify a list of port and destination mappings. A destination may be either an IP address or a DNS name.

An egress router pod supports the following formats for specifying port and destination mappings:

Port and remote address

You can specify a source port and a destination host by using the two field format: **<port> <remote_address>**.

The host can be an IP address or a DNS name. If a DNS name is provided, DNS resolution occurs at runtime. For a given host, the proxy connects to the specified source port on the destination host when connecting to the destination host IP address.

Port and remote address pair example

```
80 172.16.12.11
100 example.com
```

Port, remote address, and remote port

You can specify a source port, a destination host, and a destination port by using the three field format: **<port> <remote_address> <remote_port>**.

The three field format behaves identically to the two field version, with the exception that the destination port can be different than the source port.

Port, remote address, and remote port example

```
8080 192.168.60.252 80
8443 web.example.com 443
```

14.10.3. Deploying an egress router pod in DNS proxy mode

In *DNS proxy mode*, an egress router pod acts as a DNS proxy for TCP-based services from its own IP address to one or more destination IP addresses.

Prerequisites

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

Procedure

1. Create an egress router pod.
2. Create a service for the egress router pod:
   a. Create a file named **egress-router-service.yaml** that contains the following YAML. Set **spec.ports** to the list of ports that you defined previously for the **EGRESS_DNS_PROXY_DESTINATION** environment variable.

```
apiVersion: v1
kind: Service
metadata:
  name: egress-dns-svc
spec:
  ports:
    ...
  type: ClusterIP
  selector:
    name: egress-dns-proxy
```

   For example:

```
apiVersion: v1
kind: Service
metadata:
  name: egress-dns-svc
spec:
  ports:
  - name: con1
    protocol: TCP
    port: 80
    targetPort: 80
  - name: con2
    protocol: TCP
    port: 100
    targetPort: 100
  type: ClusterIP
  selector:
    name: egress-dns-proxy
```

   b. To create the service, enter the following command:

```
$ oc create -f egress-router-service.yaml
```

   Pods can now connect to this service. The connections are proxied to the corresponding ports on the external server, using the reserved egress IP address.

14.10.4. Additional resources

- Configuring an egress router destination mappings with a ConfigMap

14.11. CONFIGURING AN EGRESS ROUTER POD DESTINATION LIST FROM A CONFIG MAP

As a cluster administrator, you can define a **ConfigMap** object that specifies destination mappings for an egress router pod. The specific format of the configuration depends on the type of egress router pod. For details on the format, refer to the documentation for the specific egress router pod.

14.11.1. Configuring an egress router destination mappings with a config map

For a large or frequently-changing set of destination mappings, you can use a config map to externally maintain the list. An advantage of this approach is that permission to edit the config map can be delegated to users without **cluster-admin** privileges. Because the egress router pod requires a privileged container, it is not possible for users without **cluster-admin** privileges to edit the pod definition directly.

NOTE

The egress router pod does not automatically update when the config map changes. You must restart the egress router pod to get updates.

Prerequisites

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

Procedure

1. Create a file containing the mapping data for the egress router pod, as in the following example:

```
# Egress routes for Project "Test", version 3

80   tcp 203.0.113.25

8080 tcp 203.0.113.26 80
8443 tcp 203.0.113.26 443

# Fallback
203.0.113.27
```

   You can put blank lines and comments into this file.

2. Create a **ConfigMap** object from the file:

```
$ oc delete configmap egress-routes --ignore-not-found
```

```
$ oc create configmap egress-routes \
  --from-file=destination=my-egress-destination.txt
```

   In the previous command, the **egress-routes** value is the name of the **ConfigMap** object to create and **my-egress-destination.txt** is the name of the file that the data is read from.

TIP

You can alternatively apply the following YAML to create the config map:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: egress-routes
data:
  destination: |
    # Egress routes for Project "Test", version 3

    80   tcp 203.0.113.25

    8080 tcp 203.0.113.26 80
    8443 tcp 203.0.113.26 443

    # Fallback
    203.0.113.27
```

3. Create an egress router pod definition and specify the **configMapKeyRef** stanza for the **EGRESS_DESTINATION** field in the environment stanza:

```
...
env:
- name: EGRESS_DESTINATION
  valueFrom:
    configMapKeyRef:
      name: egress-routes
      key: destination
...
```

14.11.2. Additional resources

- Redirect mode
- HTTP proxy mode
- DNS proxy mode

14.12. ENABLING MULTICAST FOR A PROJECT

14.12.1. 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.

Multicast traffic between OpenShift Container Platform pods is disabled by default. If you are using the OpenShift SDN default Container Network Interface (CNI) network provider, you can enable multicast on a per-project basis.

When using the OpenShift SDN network plug-in in **networkpolicy** isolation mode:

- Multicast packets sent by a pod will be delivered to all other pods in the project, regardless of **NetworkPolicy** objects. Pods might be able to communicate over multicast even when they cannot communicate over unicast.
- Multicast packets sent by a pod in one project will never be delivered to pods in any other project, even if there are **NetworkPolicy** objects that allow communication between the projects.

When using the OpenShift SDN network plug-in in **multitenant** isolation mode:

- Multicast packets sent by a pod will be delivered to all other pods in the project.
- Multicast packets sent by a pod in one project will be delivered to pods in other projects only if each project is joined together and multicast is enabled in each joined project.

14.12.2. 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 netnamespace <namespace> \
    netnamespace.network.openshift.io/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/ubi8
      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/ubi8
      command: ["/bin/sh", "-c"]
      args:
        ["dnf -y install socat && sleep inf"]
EOF
```

4. Start the multicast listener.

   a. Get the IP address for the Pod:

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

   b. To start the multicast listener, in a new terminal window or tab, enter 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
```

14.13. DISABLING MULTICAST FOR A PROJECT

14.13.1. 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 netnamespace <namespace> \ 1
    netnamespace.network.openshift.io/multicast-enabled-
```

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

14.14. CONFIGURING NETWORK ISOLATION USING OPENSHIFT SDN

When your cluster is configured to use the multitenant isolation mode for the OpenShift SDN CNI plug-in, each project is isolated by default. Network traffic is not allowed between pods or services in different projects in multitenant isolation mode.

You can change the behavior of multitenant isolation for a project in two ways:

- You can join one or more projects, allowing network traffic between pods and services in different projects.
- You can disable network isolation for a project. It will be globally accessible, accepting network traffic from pods and services in all other projects. A globally accessible project can access pods and services in all other projects.

14.14.1. Prerequisites

- You must have a cluster configured to use the OpenShift SDN Container Network Interface (CNI) plug-in in multitenant isolation mode.

14.14.2. Joining projects

You can join two or more projects to allow network traffic between pods and services in different projects.

Prerequisites

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

Procedure

1. Use the following command to join projects to an existing project network:

```
$ oc adm pod-network join-projects --to=<project1> <project2> <project3>
```

   Alternatively, instead of specifying specific project names, you can use the **--selector=<project_selector>** option to specify projects based upon an associated label.

2. Optional: Run the following command to view the pod networks that you have joined together:

```
$ oc get netnamespaces
```

   Projects in the same pod-network have the same network ID in the **NETID** column.

14.14.3. Isolating a project

You can isolate a project so that pods and services in other projects cannot access its pods and services.

Prerequisites

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

Procedure

- To isolate the projects in the cluster, run the following command:

```
$ oc adm pod-network isolate-projects <project1> <project2>
```

  Alternatively, instead of specifying specific project names, you can use the **--selector=<project_selector>** option to specify projects based upon an associated label.

14.14.4. Disabling network isolation for a project

You can disable network isolation for a 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 for the project:

```
$ oc adm pod-network make-projects-global <project1> <project2>
```

Alternatively, instead of specifying specific project names, you can use the **--selector=<project_selector>** option to specify projects based upon an associated label.

14.15. CONFIGURING KUBE-PROXY

The Kubernetes network proxy (kube-proxy) runs on each node and is managed by the Cluster Network Operator (CNO). kube-proxy maintains network rules for forwarding connections for endpoints associated with services.

14.15.1. About iptables rules synchronization

The synchronization period determines how frequently the Kubernetes network proxy (kube-proxy) syncs the iptables rules on a node.

A sync begins when either of the following events occurs:

- An event occurs, such as service or endpoint is added to or removed from the cluster.
- The time since the last sync exceeds the sync period defined for kube-proxy.

14.15.2. kube-proxy configuration parameters

You can modify the following **kubeProxyConfig** parameters.

> **IMPORTANT**
>
> Because of performance improvements introduced in OpenShift Container Platform 4.3 and greater, adjusting the **iptablesSyncPeriod** parameter is no longer necessary.

Table 14.2. Parameters

Parameter	Description	Values	Default
iptablesSyncPeriod	The refresh period for **iptables** rules.	A time interval, such as **30s** or **2m**. Valid suffixes include **s**, **m**, and **h** and are described in the Go time package documentation.	**30s**

Parameter	Description	Values	Default
proxyArguments.iptables-min-sync-period	The minimum duration before refreshing **iptables** rules. This parameter ensures that the refresh does not happen too frequently. By default, a refresh starts as soon as a change that affects **iptables** rules occurs.	A time interval, such as **30s** or **2m**. Valid suffixes include **s**, **m**, and **h** and are described in the Go time package	**0s**

14.15.3. Modifying the kube-proxy configuration

You can modify the Kubernetes network proxy configuration for your cluster.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Log in to a running cluster with the **cluster-admin** role.

Procedure

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

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

2. Modify the **kubeProxyConfig** parameter in the CR with your changes to the kube-proxy configuration, such as in the following example CR:

```
apiVersion: operator.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  kubeProxyConfig:
    iptablesSyncPeriod: 30s
    proxyArguments:
      iptables-min-sync-period: ["30s"]
```

3. Save the file and exit the text editor.
   The syntax is validated by the **oc** command when you save the file and exit the editor. If your modifications contain a syntax error, the editor opens the file and displays an error message.

4. Enter the following command to confirm the configuration update:

```
$ oc get networks.operator.openshift.io -o yaml
```

Example output

```
apiVersion: v1
items:
- apiVersion: operator.openshift.io/v1
  kind: Network
  metadata:
    name: cluster
  spec:
    clusterNetwork:
    - cidr: 10.128.0.0/14
      hostPrefix: 23
    defaultNetwork:
      type: OpenShiftSDN
    kubeProxyConfig:
      iptablesSyncPeriod: 30s
      proxyArguments:
        iptables-min-sync-period:
        - 30s
    serviceNetwork:
    - 172.30.0.0/16
  status: {}
kind: List
```

5. Optional: Enter the following command to confirm that the Cluster Network Operator accepted the configuration change:

```
$ oc get clusteroperator network
```

Example output

```
NAME      VERSION     AVAILABLE   PROGRESSING   DEGRADED   SINCE
network   4.1.0-0.9   True        False         False      1m
```

The **AVAILABLE** field is **True** when the configuration update is applied successfully.

CHAPTER 15. OVN-KUBERNETES DEFAULT CNI NETWORK PROVIDER

15.1. ABOUT THE OVN-KUBERNETES DEFAULT CONTAINER NETWORK INTERFACE (CNI) NETWORK PROVIDER

The OpenShift Container Platform cluster uses a virtualized network for pod and service networks. The OVN-Kubernetes Container Network Interface (CNI) plug-in is a network provider for the default cluster network. OVN-Kubernetes is based on Open Virtual Network (OVN) and provides an overlay-based networking implementation. A cluster that uses the OVN-Kubernetes network provider also runs Open vSwitch (OVS) on each node. OVN configures OVS on each node to implement the declared network configuration.

15.1.1. OVN-Kubernetes features

The OVN-Kubernetes default Container Network Interface (CNI) network provider implements the following features:

- Uses OVN (Open Virtual Network) to manage network traffic flows. OVN is a community developed, vendor agnostic network virtualization solution.

- Implements Kubernetes network policy support, including ingress and egress rules.

- Uses the Geneve (Generic Network Virtualization Encapsulation) protocol rather than VXLAN to create an overlay network between nodes.

15.1.2. Supported default CNI network provider feature matrix

OpenShift Container Platform offers two supported choices, OpenShift SDN and OVN-Kubernetes, for the default Container Network Interface (CNI) network provider. The following table summarizes the current feature support for both network providers:

Table 15.1. Default CNI network provider feature comparison

Feature	OVN-Kubernetes	OpenShift SDN
Egress IPs	Supported	Supported
Egress firewall [1]	Supported	Supported
Egress router	Supported [2]	Supported
IPsec encryption	Supported	Not supported
IPv6	Supported [3]	Not supported
Kubernetes network policy	Supported	Partially supported [4]
Kubernetes network policy logs	Supported	Not supported

Feature	OVN-Kubernetes	OpenShift SDN
Multicast	Supported	Supported

1. Egress firewall is also known as egress network policy in OpenShift SDN. This is not the same as network policy egress.
2. Egress router for OVN-Kubernetes supports only redirect mode.
3. IPv6 is supported only on bare metal clusters.
4. Network policy for OpenShift SDN does not support egress rules and some **ipBlock** rules.

Additional resources

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

15.2. MIGRATE FROM THE OPENSHIFT SDN CLUSTER NETWORK PROVIDER

As a cluster administrator, you can migrate to the OVN-Kubernetes default Container Network Interface (CNI) network provider from the OpenShift SDN default CNI network provider.

To learn more about OVN-Kubernetes, read About the OVN-Kubernetes network provider .

15.2.1. Migration to the OVN-Kubernetes network provider

Migrating to the OVN-Kubernetes Container Network Interface (CNI) cluster network provider is a manual process that includes some downtime during which your cluster is unreachable. Although a rollback procedure is provided, the migration is intended to be a one-way process.

A migration to the OVN-Kubernetes cluster network provider is supported on the following platforms:

- Bare metal hardware
- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure
- Red Hat OpenStack Platform (RHOSP)
- Red Hat Virtualization (RHV)

- VMware vSphere

15.2.1.1. Considerations for migrating to the OVN-Kubernetes network provider

The subnets assigned to nodes and the IP addresses assigned to individual pods are not preserved during the migration.

While the OVN-Kubernetes network provider implements many of the capabilities present in the OpenShift SDN network provider, the configuration is not the same.

- If your cluster uses any of the following OpenShift SDN capabilities, you must manually configure the same capability in OVN-Kubernetes:
  - Namespace isolation
  - Egress IP addresses
  - Egress network policies
  - Egress router pods
  - Multicast
- If your cluster uses any part of the **100.64.0.0/16** IP address range, you cannot migrate to OVN-Kubernetes because it uses this IP address range internally.

The following sections highlight the differences in configuration between the aforementioned capabilities in OVN-Kubernetes and OpenShift SDN.

Namespace isolation
OVN-Kubernetes supports only the network policy isolation mode.

> **IMPORTANT**
>
> If your cluster uses OpenShift SDN configured in either the multitenant or subnet isolation modes, you cannot migrate to the OVN-Kubernetes network provider.

Egress IP addresses
The differences in configuring an egress IP address between OVN-Kubernetes and OpenShift SDN is described in the following table:

Table 15.2. Differences in egress IP address configuration

OVN-Kubernetes	OpenShift SDN
• Create an **EgressIPs** object • Add an annotation on a **Node** object	• Patch a **NetNamespace** object • Patch a **HostSubnet** object

For more information on using egress IP addresses in OVN-Kubernetes, see "Configuring an egress IP address".

Egress network policies

The difference in configuring an egress network policy, also known as an egress firewall, between OVN-Kubernetes and OpenShift SDN is described in the following table:

Table 15.3. Differences in egress network policy configuration

OVN-Kubernetes	OpenShift SDN
• Create an **EgressFirewall** object in a namespace	• Create an **EgressNetworkPolicy** object in a namespace

For more information on using an egress firewall in OVN-Kubernetes, see "Configuring an egress firewall for a project".

Egress router pods

OVN-Kubernetes supports egress router pods in redirect mode. OVN-Kubernetes does not support egress router pods in HTTP proxy mode or DNS proxy mode.

When you deploy an egress router with the Cluster Network Operator, you cannot specify a node selector to control which node is used to host the egress router pod.

Multicast

The difference between enabling multicast traffic on OVN-Kubernetes and OpenShift SDN is described in the following table:

Table 15.4. Differences in multicast configuration

OVN-Kubernetes	OpenShift SDN
• Add an annotation on a **Namespace** object	• Add an annotation on a **NetNamespace** object

For more information on using multicast in OVN-Kubernetes, see "Enabling multicast for a project".

Network policies

OVN-Kubernetes fully supports the Kubernetes **NetworkPolicy** API in the **networking.k8s.io/v1** API group. No changes are necessary in your network policies when migrating from OpenShift SDN.

15.2.1.2. How the migration process works

The following table summarizes the migration process by segmenting between the user-initiated steps in the process and the actions that the migration performs in response.

Table 15.5. Migrating to OVN-Kubernetes from OpenShift SDN

User-initiated steps	Migration activity
Set the **migration** field of the **Network.operator.openshift.io** custom resource (CR) named **cluster** to **OVNKubernetes**. Make sure the **migration** field is **null** before setting it to a value.	**Cluster Network Operator (CNO)** Updates the status of the **Network.config.openshift.io** CR named **cluster** accordingly. **Machine Config Operator (MCO)** Rolls out an update to the systemd configuration necessary for OVN-Kubernetes; The MCO updates a single machine per pool at a time by default, causing the total time the migration takes to increase with the size of the cluster.
Update the **networkType** field of the **Network.config.openshift.io** CR.	**CNO** Performs the following actions: • Destroys the OpenShift SDN control plane pods. • Deploys the OVN-Kubernetes control plane pods. • Updates the Multus objects to reflect the new cluster network provider.
Reboot each node in the cluster.	**Cluster** As nodes reboot, the cluster assigns IP addresses to pods on the OVN-Kubernetes cluster network.

If a rollback to OpenShift SDN is required, the following table describes the process.

Table 15.6. Performing a rollback to OpenShift SDN

User-initiated steps	Migration activity
Suspend the MCO to ensure that it does not interrupt the migration.	The MCO stops.
Set the **migration** field of the **Network.operator.openshift.io** custom resource (CR) named **cluster** to **OVNKubernetes**. Make sure the **migration** field is **null** before setting it to a value.	**CNO** Updates the status of the **Network.config.openshift.io** CR named **cluster** accordingly.

User-initiated steps	Migration activity
Update the **networkType** field.	**CNO** Performs the following actions: • Destroys the OpenShift SDN control plane pods. • Deploys the OVN-Kubernetes control plane pods. • Updates the Multus objects to reflect the new cluster network provider.
Reboot each node in the cluster.	**Cluster** As nodes reboot, the cluster assigns IP addresses to pods on the OVN-Kubernetes cluster network.
Enable the MCO after all nodes in the cluster reboot.	**MCO** Rolls out an update to the systemd configuration necessary for OpenShift SDN; The MCO updates a single machine per pool at a time by default, so the total time the migration takes increases with the size of the cluster.

15.2.2. Migrating to the OVN-Kubernetes default CNI network provider

As a cluster administrator, you can change the default Container Network Interface (CNI) network provider for your cluster to OVN-Kubernetes. During the migration, you must reboot every node in your cluster.

> **IMPORTANT**
>
> While performing the migration, your cluster is unavailable and workloads might be interrupted. Perform the migration only when an interruption in service is acceptable.

Prerequisites

- A cluster configured with the OpenShift SDN CNI cluster network provider in the network policy isolation mode.
- Install the OpenShift CLI (**oc**).
- Access to the cluster as a user with the **cluster-admin** role.
- A recent backup of the etcd database is available.
- A reboot can be triggered manually for each node.
- The cluster is in a known good state, without any errors.

Procedure

1. To backup the configuration for the cluster network, enter the following command:

```
$ oc get Network.config.openshift.io cluster -o yaml > cluster-openshift-sdn.yaml
```

2. To prepare all the nodes for the migration, set the **migration** field on the Cluster Network Operator configuration object by entering the following command:

```
$ oc patch Network.operator.openshift.io cluster --type='merge' \
  --patch '{ "spec": { "migration": {"networkType": "OVNKubernetes" } } }'
```

> **NOTE**
>
> This step does not deploy OVN-Kubernetes immediately. Instead, specifying the **migration** field triggers the Machine Config Operator (MCO) to apply new machine configs to all the nodes in the cluster in preparation for the OVN-Kubernetes deployment.

3. Optional: You can customize the following settings for OVN-Kubernetes to meet your network infrastructure requirements:

   - Maximum transmission unit (MTU)
   - Geneve (Generic Network Virtualization Encapsulation) overlay network port

   To customize either of the previously noted settings, enter and customize the following command. If you do not need to change the default value, omit the key from the patch.

```
$ oc patch Network.operator.openshift.io cluster --type=merge \
  --patch '{
    "spec":{
      "defaultNetwork":{
        "ovnKubernetesConfig":{
          "mtu":<mtu>,
          "genevePort":<port>
    }}}}'
```

mtu

The MTU for the Geneve overlay network. This value is normally configured automatically, but if the nodes in your cluster do not all use the same MTU, then you must set this explicitly to **100** less than the smallest node MTU value.

port

The UDP port for the Geneve overlay network. If a value is not specified, the default is **6081**. The port cannot be the same as the VXLAN port that is used by OpenShift SDN. The default value for the VXLAN port is **4789**.

Example patch command to update mtu field

```
$ oc patch Network.operator.openshift.io cluster --type=merge \
  --patch '{
    "spec":{
      "defaultNetwork":{
```

```
    "ovnKubernetesConfig":{
      "mtu":1200
}}}}'
```

4. As the MCO updates machines in each machine config pool, it reboots each node one by one. You must wait until all the nodes are updated. Check the machine config pool status by entering the following command:

```
$ oc get mcp
```

A successfully updated node has the following status: **UPDATED=true**, **UPDATING=false**, **DEGRADED=false**.

> NOTE
>
> By default, the MCO updates one machine per pool at a time, causing the total time the migration takes to increase with the size of the cluster.

5. Confirm the status of the new machine configuration on the hosts:

    a. To list the machine configuration state and the name of the applied machine configuration, enter the following command:

    ```
    $ oc describe node | egrep "hostname|machineconfig"
    ```

    Example output

    ```
    kubernetes.io/hostname=master-0
    machineconfiguration.openshift.io/currentConfig: rendered-master-c53e221d9d24e1c8bb6ee89dd3d8ad7b
    machineconfiguration.openshift.io/desiredConfig: rendered-master-c53e221d9d24e1c8bb6ee89dd3d8ad7b
    machineconfiguration.openshift.io/reason:
    machineconfiguration.openshift.io/state: Done
    ```

    Verify that the following statements are true:

    - The value of **machineconfiguration.openshift.io/state** field is **Done**.
    - The value of the **machineconfiguration.openshift.io/currentConfig** field is equal to the value of the **machineconfiguration.openshift.io/desiredConfig** field.

    b. To confirm that the machine config is correct, enter the following command:

    ```
    $ oc get machineconfig <config_name> -o yaml | grep ExecStart
    ```

    where **<config_name>** is the name of the machine config from the **machineconfiguration.openshift.io/currentConfig** field.

    The machine config must include the following update to the systemd configuration:

    ```
    ExecStart=/usr/local/bin/configure-ovs.sh OVNKubernetes
    ```

c. If a node is stuck in the **NotReady** state, investigate the machine config daemon pod logs and resolve any errors.

i. To list the pods, enter the following command:

```
$ oc get pod -n openshift-machine-config-operator
```

Example output

```
NAME                                         READY   STATUS    RESTARTS   AGE
machine-config-controller-75f756f89d-sjp8b   1/1     Running   0          37m
machine-config-daemon-5cf4b                  2/2     Running   0          43h
machine-config-daemon-7wzcd                  2/2     Running   0          43h
machine-config-daemon-fc946                  2/2     Running   0          43h
machine-config-daemon-g2v28                  2/2     Running   0          43h
machine-config-daemon-gcl4f                  2/2     Running   0          43h
machine-config-daemon-l5tnv                  2/2     Running   0          43h
machine-config-operator-79d9c55d5-hth92      1/1     Running   0          37m
machine-config-server-bsc8h                  1/1     Running   0          43h
machine-config-server-hklrm                  1/1     Running   0          43h
machine-config-server-k9rtx                  1/1     Running   0          43h
```

The names for the config daemon pods are in the following format: **machine-config-daemon-<seq>**. The **<seq>** value is a random five character alphanumeric sequence.

ii. Display the pod log for the first machine config daemon pod shown in the previous output by enter the following command:

```
$ oc logs <pod> -n openshift-machine-config-operator
```

where **pod** is the name of a machine config daemon pod.

iii. Resolve any errors in the logs shown by the output from the previous command.

6. To start the migration, configure the OVN-Kubernetes cluster network provider by using one of the following commands:

- To specify the network provider without changing the cluster network IP address block, enter the following command:

```
$ oc patch Network.config.openshift.io cluster \
  --type='merge' --patch '{ "spec": { "networkType": "OVNKubernetes" } }'
```

- To specify a different cluster network IP address block, enter the following command:

```
$ oc patch Network.config.openshift.io cluster \
  --type='merge' --patch '{
    "spec": {
      "clusterNetwork": [
        {
          "cidr": "<cidr>",
          "hostPrefix": "<prefix>"
        }
      ]
```

```
    "networkType": "OVNKubernetes"
  }
}'
```

where **cidr** is a CIDR block and **prefix** is the slice of the CIDR block apportioned to each node in your cluster. You cannot use any CIDR block that overlaps with the **100.64.0.0/16** CIDR block because the OVN-Kubernetes network provider uses this block internally.

> **IMPORTANT**
>
> You cannot change the service network address block during the migration.

7. Verify that the Multus daemon set rollout is complete before continuing with subsequent steps:

```
$ oc -n openshift-multus rollout status daemonset/multus
```

The name of the Multus pods is in the form of **multus-<xxxxx>** where **<xxxxx>** is a random sequence of letters. It might take several moments for the pods to restart.

Example output

```
Waiting for daemon set "multus" rollout to finish: 1 out of 6 new pods have been updated...
...
Waiting for daemon set "multus" rollout to finish: 5 of 6 updated pods are available...
daemon set "multus" successfully rolled out
```

8. To complete the migration, reboot each node in your cluster. For example, you can use a bash script similar to the following example. The script assumes that you can connect to each host by using **ssh** and that you have configured **sudo** to not prompt for a password.

```
#!/bin/bash

for ip in $(oc get nodes  -o jsonpath='{.items[*].status.addresses[?
(@.type=="InternalIP")].address}')
do
   echo "reboot node $ip"
   ssh -o StrictHostKeyChecking=no core@$ip sudo shutdown -r -t 3
done
```

If ssh access is not available, you might be able to reboot each node through the management portal for your infrastructure provider.

9. Confirm that the migration succeeded:

   a. To confirm that the CNI cluster network provider is OVN-Kubernetes, enter the following command. The value of **status.networkType** must be **OVNKubernetes**.

   ```
   $ oc get network.config/cluster -o jsonpath='{.status.networkType}{"\n"}'
   ```

   b. To confirm that the cluster nodes are in the **Ready** state, enter the following command:

   ```
   $ oc get nodes
   ```

   c. To confirm that your pods are not in an error state, enter the following command:

```
$ oc get pods --all-namespaces -o wide --sort-by='{.spec.nodeName}'
```

If pods on a node are in an error state, reboot that node.

d. To confirm that all of the cluster Operators are not in an abnormal state, enter the following command:

```
$ oc get co
```

The status of every cluster Operator must be the following: **AVAILABLE="True"**, **PROGRESSING="False"**, **DEGRADED="False"**. If a cluster Operator is not available or degraded, check the logs for the cluster Operator for more information.

10. Complete the following steps only if the migration succeeds and your cluster is in a good state:

a. To remove the migration configuration from the CNO configuration object, enter the following command:

```
$ oc patch Network.operator.openshift.io cluster --type='merge' \
  --patch '{ "spec": { "migration": null } }'
```

b. To remove custom configuration for the OpenShift SDN network provider, enter the following command:

```
$ oc patch Network.operator.openshift.io cluster --type='merge' \
  --patch '{ "spec": { "defaultNetwork": { "openshiftSDNConfig": null } } }'
```

c. To remove the OpenShift SDN network provider namespace, enter the following command:

```
$ oc delete namespace openshift-sdn
```

15.2.3. Additional resources

- Configuration parameters for the OVN-Kubernetes default CNI network provider
- Backing up etcd
- About network policy
- OVN-Kubernetes capabilities
  - Configuring an egress IP address
  - Configuring an egress firewall for a project
  - Enabling multicast for a project
- OpenShift SDN capabilities
  - Configuring egress IPs for a project
  - Configuring an egress firewall for a project
  - Enabling multicast for a project

- Network [operator.openshift.io/v1]

15.3. ROLLBACK TO THE OPENSHIFT SDN NETWORK PROVIDER

As a cluster administrator, you can rollback to the OpenShift SDN cluster default Container Network Interface (CNI) provider from the OVN-Kubernetes default CNI network provider if the migration to OVN-Kubernetes unsuccessful.

15.3.1. Rolling back the default CNI network provider to OpenShift SDN

As a cluster administrator, you can rollback your cluster to the OpenShift SDN Container Network Interface (CNI) cluster network provider. During the rollback, you must reboot every node in your cluster.

> **IMPORTANT**
>
> Only rollback to OpenShift SDN if the migration to OVN-Kubernetes fails.

Prerequisites

- Install the OpenShift CLI (**oc**).
- Access to the cluster as a user with the **cluster-admin** role.
- A cluster installed on infrastructure configured with the OVN-Kubernetes CNI cluster network provider.

Procedure

1. Stop all of the machine configuration pools managed by the Machine Config Operator (MCO):
   - Stop the master configuration pool:

     ```
     $ oc patch MachineConfigPool master --type='merge' --patch \
       '{ "spec": { "paused": true } }'
     ```

   - Stop the worker machine configuration pool:

     ```
     $ oc patch MachineConfigPool worker --type='merge' --patch \
       '{ "spec":{ "paused" :true } }'
     ```

2. To start the migration, set the cluster network provider back to OpenShift SDN by entering the following commands:

   ```
   $ oc patch Network.operator.openshift.io cluster --type='merge' \
     --patch '{ "spec": { "migration": { "networkType": "OpenShiftSDN" } } }'

   $ oc patch Network.config.openshift.io cluster --type='merge' \
     --patch '{ "spec": { "networkType": "OpenShiftSDN" } }'
   ```

3. Optional: You can customize the following settings for OpenShift SDN to meet your network infrastructure requirements:
   - Maximum transmission unit (MTU)

- VXLAN port

To customize either or both of the previously noted settings, customize and enter the following command. If you do not need to change the default value, omit the key from the patch.

```
$ oc patch Network.operator.openshift.io cluster --type=merge \
  --patch '{
    "spec":{
      "defaultNetwork":{
        "openshiftSDNConfig":{
          "mtu":<mtu>,
          "vxlanPort":<port>
    }}}}'
```

mtu

The MTU for the VXLAN overlay network. This value is normally configured automatically, but if the nodes in your cluster do not all use the same MTU, then you must set this explicitly to **50** less than the smallest node MTU value.

port

The UDP port for the VXLAN overlay network. If a value is not specified, the default is **4789**. The port cannot be the same as the Geneve port that is used by OVN-Kubernetes. The default value for the Geneve port is **6081**.

Example patch command

```
$ oc patch Network.operator.openshift.io cluster --type=merge \
  --patch '{
    "spec":{
      "defaultNetwork":{
        "openshiftSDNConfig":{
          "mtu":1200
    }}}}'
```

4. Wait until the Multus daemon set rollout completes.

```
$ oc -n openshift-multus rollout status daemonset/multus
```

The name of the Multus pods is in form of **multus-<xxxxx>** where **<xxxxx>** is a random sequence of letters. It might take several moments for the pods to restart.

Example output

```
Waiting for daemon set "multus" rollout to finish: 1 out of 6 new pods have been updated...
...
Waiting for daemon set "multus" rollout to finish: 5 of 6 updated pods are available...
daemon set "multus" successfully rolled out
```

5. To complete the rollback, reboot each node in your cluster. For example, you could use a bash script similar to the following. The script assumes that you can connect to each host by using **ssh** and that you have configured **sudo** to not prompt for a password.

```
#!/bin/bash
```

```
for ip in $(oc get nodes  -o jsonpath='{.items[*].status.addresses[?
(@.type=="InternalIP")].address}')
do
  echo "reboot node $ip"
  ssh -o StrictHostKeyChecking=no core@$ip sudo shutdown -r -t 3
done
```

If ssh access is not available, you might be able to reboot each node through the management portal for your infrastructure provider.

6. After the nodes in your cluster have rebooted, start all of the machine configuration pools:
   - Start the master configuration pool:

     ```
     $ oc patch MachineConfigPool master --type='merge' --patch \
       '{ "spec": { "paused": false } }'
     ```

   - Start the worker configuration pool:

     ```
     $ oc patch MachineConfigPool worker --type='merge' --patch \
       '{ "spec": { "paused": false } }'
     ```

   As the MCO updates machines in each config pool, it reboots each node.

   By default the MCO updates a single machine per pool at a time, so the time that the migration requires to complete grows with the size of the cluster.

7. Confirm the status of the new machine configuration on the hosts:
   a. To list the machine configuration state and the name of the applied machine configuration, enter the following command:

      ```
      $ oc describe node | egrep "hostname|machineconfig"
      ```

      Example output

      ```
      kubernetes.io/hostname=master-0
      machineconfiguration.openshift.io/currentConfig: rendered-master-
      c53e221d9d24e1c8bb6ee89dd3d8ad7b
      machineconfiguration.openshift.io/desiredConfig: rendered-master-
      c53e221d9d24e1c8bb6ee89dd3d8ad7b
      machineconfiguration.openshift.io/reason:
      machineconfiguration.openshift.io/state: Done
      ```

      Verify that the following statements are true:

      - The value of **machineconfiguration.openshift.io/state** field is **Done**.
      - The value of the **machineconfiguration.openshift.io/currentConfig** field is equal to the value of the **machineconfiguration.openshift.io/desiredConfig** field.

   b. To confirm that the machine config is correct, enter the following command:

      ```
      $ oc get machineconfig <config_name> -o yaml
      ```

where **<config_name>** is the name of the machine config from the **machineconfiguration.openshift.io/currentConfig** field.

8. Confirm that the migration succeeded:

   a. To confirm that the default CNI network provider is OVN-Kubernetes, enter the following command. The value of **status.networkType** must be **OpenShiftSDN**.

   ```
   $ oc get network.config/cluster -o jsonpath='{.status.networkType}{"\n"}'
   ```

   b. To confirm that the cluster nodes are in the **Ready** state, enter the following command:

   ```
   $ oc get nodes
   ```

   c. If a node is stuck in the **NotReady** state, investigate the machine config daemon pod logs and resolve any errors.

      i. To list the pods, enter the following command:

      ```
      $ oc get pod -n openshift-machine-config-operator
      ```

      ### Example output

      ```
      NAME                                         READY   STATUS    RESTARTS   AGE
      machine-config-controller-75f756f89d-sjp8b   1/1     Running   0          37m
      machine-config-daemon-5cf4b                  2/2     Running   0          43h
      machine-config-daemon-7wzcd                  2/2     Running   0          43h
      machine-config-daemon-fc946                  2/2     Running   0          43h
      machine-config-daemon-g2v28                  2/2     Running   0          43h
      machine-config-daemon-gcl4f                  2/2     Running   0          43h
      machine-config-daemon-l5tnv                  2/2     Running   0          43h
      machine-config-operator-79d9c55d5-hth92      1/1     Running   0          37m
      machine-config-server-bsc8h                  1/1     Running   0          43h
      machine-config-server-hklrm                  1/1     Running   0          43h
      machine-config-server-k9rtx                  1/1     Running   0          43h
      ```

      The names for the config daemon pods are in the following format: **machine-config-daemon-<seq>**. The **<seq>** value is a random five character alphanumeric sequence.

      ii. To display the pod log for each machine config daemon pod shown in the previous output, enter the following command:

      ```
      $ oc logs <pod> -n openshift-machine-config-operator
      ```

      where **pod** is the name of a machine config daemon pod.

      iii. Resolve any errors in the logs shown by the output from the previous command.

   d. To confirm that your pods are not in an error state, enter the following command:

   ```
   $ oc get pods --all-namespaces -o wide --sort-by='{.spec.nodeName}'
   ```

   If pods on a node are in an error state, reboot that node.

9. Complete the following steps only if the migration succeeds and your cluster is in a good state:

   a. To remove the migration configuration from the Cluster Network Operator configuration object, enter the following command:

   ```
   $ oc patch Network.operator.openshift.io cluster --type='merge' \
     --patch '{ "spec": { "migration": null } }'
   ```

   b. To remove the OVN-Kubernetes configuration, enter the following command:

   ```
   $ oc patch Network.operator.openshift.io cluster --type='merge' \
     --patch '{ "spec": { "defaultNetwork": { "ovnKubernetesConfig":null } } }'
   ```

   c. To remove the OVN-Kubernetes network provider namespace, enter the following command:

   ```
   $ oc delete namespace openshift-ovn-kubernetes
   ```

15.4. 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, all newly created pods are dual-stack enabled.

NOTE

A dual-stack network is supported on clusters provisioned on only bare metal infrastructure.

15.4.1. 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.

NOTE

After converting to dual-stack networking only newly created pods are assigned IPv6 addresses. Any pods created before the conversion must be recreated to receive an IPv6 address.

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 cluster network provider.

Procedure

1. To specify IPv6 address blocks for the cluster and service networks, create a file containing the following YAML:

```
- 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** fields. The host prefix must be **64** or greater. The IPv6 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. To patch the cluster network configuration, enter the following command:

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

where:

file

Specifies the name of the file you created in the previous step.

Example output

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

Verification

Complete the following step to verify that the cluster network recognizes the IPv6 address blocks that you specified in the previous procedure.

1. Display the network configuration:

```
$ oc describe network
```

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
```

15.5. IPSEC ENCRYPTION CONFIGURATION

With IPsec enabled, all network traffic between nodes on the OVN-Kubernetes Container Network Interface (CNI) cluster network travels through an encrypted tunnel.

IPsec is disabled by default.

> **NOTE**
>
> IPsec encryption can be enabled only during cluster installation and cannot be disabled after it is enabled. For installation documentation, refer to Selecting a cluster installation method and preparing it for users.

15.5.1. Types of network traffic flows encrypted by IPsec

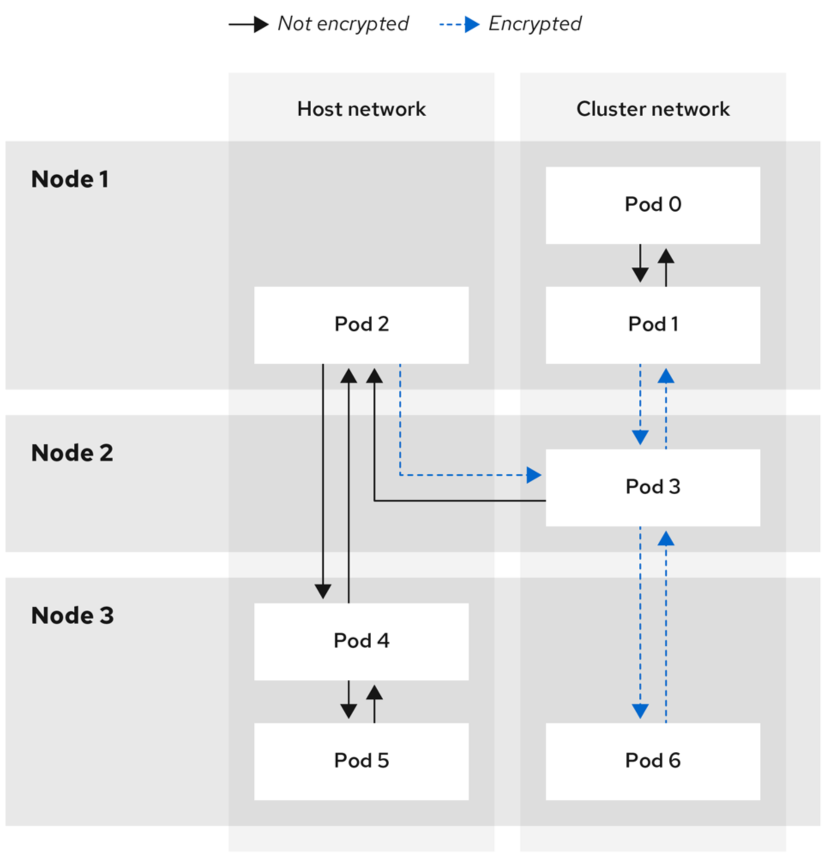
With IPsec enabled, only the following network traffic flows between pods are encrypted:

- Traffic between pods on different nodes on the cluster network
- Traffic from a pod on the host network to a pod on the cluster network

The following traffic flows are not encrypted:

- Traffic between pods on the same node on the cluster network
- Traffic between pods on the host network
- Traffic from a pod on the cluster network to a pod on the host network

The encrypted and unencrypted flows are illustrated in the following diagram:



15.5.2. Encryption protocol and tunnel mode for IPsec

The encrypt cipher used is **AES-GCM-16-256**. The integrity check value (ICV) is **16** bytes. The key length is **256** bits.

The IPsec tunnel mode used is *Transport mode*, a mode that encrypts end-to-end communication.

15.5.3. Security certificate generation and rotation

The Cluster Network Operator (CNO) generates a self-signed X.509 certificate authority (CA) that is used by IPsec for encryption. Certificate signing requests (CSRs) from each node are automatically fulfilled by the CNO.

The CA is valid for 10 years. The individual node certificates are valid for 5 years and are automatically rotated after 4 1/2 years elapse.

15.6. CONFIGURING AN EGRESS FIREWALL FOR A PROJECT

As a cluster administrator, you can create an egress firewall for a project that restricts egress traffic leaving your OpenShift Container Platform cluster.

15.6.1. How an egress firewall works in a project

As a cluster administrator, you can use an *egress firewall* to limit the external hosts that some or all pods can access from within the cluster. An egress firewall supports the following scenarios:

- A pod can only connect to internal hosts and cannot initiate connections to the public internet.
- A pod can only connect to the public internet and cannot initiate connections to internal hosts that are outside the OpenShift Container Platform cluster.
- A pod cannot reach specified internal subnets or hosts outside the OpenShift Container Platform cluster.
- A pod can connect to only specific external hosts.

For example, you can allow one project access to a specified IP range but deny the same access to a different project. Or you can restrict application developers from updating from Python pip mirrors, and force updates to come only from approved sources.

You configure an egress firewall policy by creating an EgressFirewall custom resource (CR) object. The egress firewall matches network traffic that meets any of the following criteria:

- An IP address range in CIDR format
- A DNS name that resolves to an IP address
- A port number
- A protocol that is one of the following protocols: TCP, UDP, and SCTP

> **WARNING**
>
> Egress firewall rules do not apply to traffic that goes through routers. Any user with permission to create a Route CR object can bypass egress firewall policy rules by creating a route that points to a forbidden destination.

15.6.1.1. Limitations of an egress firewall

An egress firewall has the following limitations:

- No project can have more than one EgressFirewall object.

Violating any of these restrictions results in a broken egress firewall for the project, and might cause all external network traffic to be dropped.

15.6.1.2. Matching order for egress firewall policy rules

The egress firewall policy rules are evaluated in the order that they are defined, from first to last. The first rule that matches an egress connection from a pod applies. Any subsequent rules are ignored for that connection.

15.6.1.3. How Domain Name Server (DNS) resolution works

If you use DNS names in any of your egress firewall policy rules, proper resolution of the domain names is subject to the following restrictions:

- Domain name updates are polled based on a time-to-live (TTL) duration. By default, the duration is 30 minutes. When the egress firewall controller queries the local name servers for a domain name, if the response includes a TTL and the TTL is less than 30 minutes, the controller sets the duration for that DNS name to the returned value. Each DNS name is queried after the TTL for the DNS record expires.

- The pod must resolve the domain from the same local name servers when necessary. Otherwise the IP addresses for the domain known by the egress firewall controller and the pod can be different. If the IP addresses for a hostname differ, the egress firewall might not be enforced consistently.

- Because the egress firewall controller and pods asynchronously poll the same local name server, the pod might obtain the updated IP address before the egress controller does, which causes a race condition. Due to this current limitation, domain name usage in EgressFirewall objects is only recommended for domains with infrequent IP address changes.

> **NOTE**
>
> The egress firewall always allows pods access to the external interface of the node that the pod is on for DNS resolution.
>
> If you use domain names in your egress firewall policy and your DNS resolution is not handled by a DNS server on the local node, then you must add egress firewall rules that allow access to your DNS server’s IP addresses. if you are using domain names in your pods.

15.6.2. EgressFirewall custom resource (CR) object

You can define one or more rules for an egress firewall. A rule is either an **Allow** rule or a **Deny** rule, with a specification for the traffic that the rule applies to.

The following YAML describes an EgressFirewall CR object:

EgressFirewall object

```
apiVersion: k8s.ovn.org/v1
kind: EgressFirewall
metadata:
  name: <name> 1
spec:
  egress: 2
    ...
```

1. The name for the object must be **default**.
2. A collection of one or more egress network policy rules as described in the following section.

15.6.2.1. EgressFirewall rules

The following YAML describes an egress firewall rule object. The **egress** stanza expects an array of one or more objects.

Egress policy rule stanza

```
egress:
- type: <type> 1
  to: 2
    cidrSelector: <cidr> 3
    dnsName: <dns_name> 4
  ports: 5
      ...
```

1. The type of rule. The value must be either **Allow** or **Deny**.
2. A stanza describing an egress traffic match rule that specifies the **cidrSelector** field or the **dnsName** field. You cannot use both fields in the same rule.
3. An IP address range in CIDR format.
4. A DNS domain name.
5. Optional: A stanza describing a collection of network ports and protocols for the rule.

Ports stanza

```
ports:
- port: <port> 1
  protocol: <protocol> 2
```

–

1. A network port, such as **80** or **443**. If you specify a value for this field, you must also specify a value for **protocol**.

2. A network protocol. The value must be either **TCP**, **UDP**, or **SCTP**.

15.6.2.2. Example EgressFirewall CR objects

The following example defines several egress firewall policy rules:

```
apiVersion: k8s.ovn.org/v1
kind: EgressFirewall
metadata:
  name: default
spec:
  egress: 1
  - type: Allow
    to:
      cidrSelector: 1.2.3.0/24
  - type: Deny
    to:
      cidrSelector: 0.0.0.0/0
```

1. A collection of egress firewall policy rule objects.

The following example defines a policy rule that denies traffic to the host at the **172.16.1.1** IP address, if the traffic is using either the TCP protocol and destination port **80** or any protocol and destination port **443**.

```
apiVersion: k8s.ovn.org/v1
kind: EgressFirewall
metadata:
  name: default
spec:
  egress:
  - type: Deny
    to:
      cidrSelector: 172.16.1.1
    ports:
    - port: 80
      protocol: TCP
    - port: 443
```

15.6.3. Creating an egress firewall policy object

As a cluster administrator, you can create an egress firewall policy object for a project.

> **IMPORTANT**
>
> If the project already has an EgressFirewall object defined, you must edit the existing policy to make changes to the egress firewall rules.

Prerequisites

- A cluster that uses the OVN-Kubernetes default Container Network Interface (CNI) network provider plug-in.
- Install the OpenShift CLI (**oc**).
- You must log in to the cluster as a cluster administrator.

Procedure

1. Create a policy rule:
   a. Create a **<policy_name>.yaml** file where **<policy_name>** describes the egress policy rules.
   b. In the file you created, define an egress policy object.
2. Enter the following command to create the policy object. Replace **<policy_name>** with the name of the policy and **<project>** with the project that the rule applies to.

```
$ oc create -f <policy_name>.yaml -n <project>
```

   In the following example, a new EgressFirewall object is created in a project named **project1**:

```
$ oc create -f default.yaml -n project1
```

   Example output

```
egressfirewall.k8s.ovn.org/v1 created
```

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

15.7. VIEWING AN EGRESS FIREWALL FOR A PROJECT

As a cluster administrator, you can list the names of any existing egress firewalls and view the traffic rules for a specific egress firewall.

15.7.1. Viewing an EgressFirewall object

You can view an EgressFirewall object in your cluster.

Prerequisites

- A cluster using the OVN-Kubernetes default Container Network Interface (CNI) network provider plug-in.
- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster.

Procedure

1. Optional: To view the names of the EgressFirewall objects defined in your cluster, enter the following command:

```
$ oc get egressfirewall --all-namespaces
```

2. To inspect a policy, enter the following command. Replace **<policy_name>** with the name of the policy to inspect.

```
$ oc describe egressfirewall <policy_name>
```

Example output

```
Name:  default
Namespace: project1
Created: 20 minutes ago
Labels:  <none>
Annotations: <none>
Rule:  Allow to 1.2.3.0/24
Rule:  Allow to www.example.com
Rule:  Deny to 0.0.0.0/0
```

15.8. EDITING AN EGRESS FIREWALL FOR A PROJECT

As a cluster administrator, you can modify network traffic rules for an existing egress firewall.

15.8.1. Editing an EgressFirewall object

As a cluster administrator, you can update the egress firewall for a project.

Prerequisites

- A cluster using the OVN-Kubernetes default Container Network Interface (CNI) network provider plug-in.
- Install the OpenShift CLI (**oc**).
- You must log in to the cluster as a cluster administrator.

Procedure

1. Find the name of the EgressFirewall object for the project. Replace **<project>** with the name of the project.

```
$ oc get -n <project> egressfirewall
```

2. Optional: If you did not save a copy of the EgressFirewall object when you created the egress network firewall, enter the following command to create a copy.

```
$ oc get -n <project> egressfirewall <name> -o yaml > <filename>.yaml
```

Replace **<project>** with the name of the project. Replace **<name>** with the name of the object. Replace **<filename>** with the name of the file to save the YAML to.

3. After making changes to the policy rules, enter the following command to replace the EgressFirewall object. Replace **<filename>** with the name of the file containing the updated EgressFirewall object.

```
$ oc replace -f <filename>.yaml
```

15.9. REMOVING AN EGRESS FIREWALL FROM A PROJECT

As a cluster administrator, you can remove an egress firewall from a project to remove all restrictions on network traffic from the project that leaves the OpenShift Container Platform cluster.

15.9.1. Removing an EgressFirewall object

As a cluster administrator, you can remove an egress firewall from a project.

Prerequisites

- A cluster using the OVN-Kubernetes default Container Network Interface (CNI) network provider plug-in.
- Install the OpenShift CLI (**oc**).
- You must log in to the cluster as a cluster administrator.

Procedure

1. Find the name of the EgressFirewall object for the project. Replace **<project>** with the name of the project.

```
$ oc get -n <project> egressfirewall
```

2. Enter the following command to delete the EgressFirewall object. Replace **<project>** with the name of the project and **<name>** with the name of the object.

```
$ oc delete -n <project> egressfirewall <name>
```

15.10. CONFIGURING AN EGRESS IP ADDRESS

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

15.10.1. Egress IP address architectural design and implementation

The OpenShift Container Platform egress IP address functionality allows you to 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 is implemented as an additional IP address on the primary network interface of a node and must be in the same subnet as the primary IP address of the node. The additional IP address must not be assigned to any other node in the cluster.

15.10.1.1. Platform support

Support for the egress IP address functionality on various platforms is summarized in the following table:

> **IMPORTANT**
>
> The egress IP address implementation is not compatible with Amazon Web Services (AWS), Azure Cloud, or any other public cloud platform incompatible with the automatic layer 2 network manipulation required by the egress IP feature.

Platform	Supported
Bare metal	Yes
vSphere	Yes
Red Hat OpenStack Platform (RHOSP)	No
Public cloud	No

15.10.1.2. Assignment of egress IPs to 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.

15.10.1.3. Assignment of egress IPs to nodes

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, no node will ever host more than one of the specified addresses.
- 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.

15.10.1.4. 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.

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 following resources from the diagram are illustrated in detail:

Namespace objects

The namespaces are defined in the following manifest:

Namespace objects

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

EgressIP object

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:
  assignments:
  - 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.

15.10.2. EgressIP object

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

```
apiVersion: k8s.ovn.org/v1
kind: EgressIP
metadata:
  name: <name> 1
spec:
  egressIPs: 2
  - <ip_address>
  namespaceSelector: 3
    ...
  podSelector: 4
    ...
```

1. The name for the **EgressIPs** object.
2. An array of one or more IP addresses.
3. One or more selectors for the namespaces to associate the egress IP addresses with.
4. Optional: 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

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: 1
  matchLabels:
    <label_name>: <label_value>
```

1 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: 1
  matchLabels:
    <label_name>: <label_value>
```

1 Optional: 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

```yaml
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
```

15.10.3. 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:

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

15.10.4. Next steps

- Assigning egress IPs

15.10.5. Additional resources

- LabelSelector meta/v1
- LabelSelectorRequirement meta/v1

15.11. ASSIGNING AN EGRESS IP ADDRESS

As a cluster administrator, you can assign an egress IP address for traffic leaving the cluster from a namespace or from specific pods in a namespace.

15.11.1. 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 **EgressIPs** 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 1
```

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

Example output

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

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

15.11.2. Additional resources

- Configuring egress IP addresses

15.12. CONSIDERATIONS FOR THE USE OF AN EGRESS ROUTER POD

15.12.1. 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.

15.12.1.1. 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 modified to connect to the egress router rather than connecting directly to the destination IP.

NOTE

The egress router CNI plug-in supports redirect mode only. This is a difference with the egress router implementation that you can deploy with OpenShift SDN. Unlike the egress router for OpenShift SDN, the egress router CNI plug-in does not support HTTP proxy mode or DNS proxy mode.

15.12.1.2. Egress router pod implementation

The egress router implementation uses the egress router Container Network Interface (CNI) plug-in. The plug-in 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.

15.12.1.3. 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>
```

Red Hat Virtualization (RHV)

If you are using RHV, you must select **No Network Filter** for the Virtual Network Interface Card (vNIC).

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

15.12.1.4. 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
```

15.12.2. Additional resources

- Deploying an egress router in redirection mode

15.13. 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) plug-in.

15.13.1. 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>  <.>
spec:
  addresses: [  <.>
    {
      ip: "<egress_router>",  <.>
      gateway: "<egress_gateway>"  <.>
    }
  ]
  mode: Redirect
  redirect: {
    redirectRules: [  <.>
      {
        destinationIP: "<egress_destination>",
        port: <egress_router_port>,
        targetPort: <target_port>,  <.>
        protocol: <network_protocol>  <.>
      },
      ...
```

```
  ],
  fallbackIP: "<egress_destination>" <.>
}
```

<.> 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.

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

<.> 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.

<.> The **gateway** field specifies the IP address of the network gateway.

<.> 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.

<.> 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.

<.> The **protocol** field supports TCP, UDP, or SCTP.

<.> 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
      }
    ]
  }
```

15.13.2. 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 <.>
```

<.> 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
```

15.14. ENABLING MULTICAST FOR A PROJECT

15.14.1. 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.

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

15.14.2. 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/ubi8
      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/ubi8
```

```
    command: ["/bin/sh", "-c"]
    args:
      ["dnf -y install socat && sleep inf"]
EOF
```

4. Start the multicast listener.

   a. Get the IP address for the Pod:

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

   b. To start the multicast listener, in a new terminal window or tab, enter 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
   ```

15.15. DISABLING MULTICAST FOR A PROJECT

15.15.1. 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:

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

15.16. 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.

15.16.1. 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 15.7. 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
```

15.16.2. Adding destinations for network flows collectors

As a cluster administator, 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-node $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"]-

...
```

15.16.3. Deleting all destinations for network flows collectors

As a cluster administator, 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
```

15.16.4. Additional resources

- Network [operator.openshift.io/v1]

15.17. CONFIGURING HYBRID NETWORKING

As a cluster administrator, you can configure the OVN-Kubernetes Container Network Interface (CNI) cluster network provider to allow Linux and Windows nodes to host Linux and Windows workloads, respectively.

15.17.1. Configuring hybrid networking with OVN-Kubernetes

You can configure your cluster to use hybrid networking with OVN-Kubernetes. This allows a hybrid cluster that supports different node networking configurations. For example, this is necessary to run both Linux and Windows nodes in a cluster.

> **IMPORTANT**
>
> You must configure hybrid networking with OVN-Kubernetes during the installation of your cluster. You cannot switch to hybrid networking after the installation process.

Prerequisites

- You defined **OVNKubernetes** for the **networking.networkType** parameter in the **install-config.yaml** file. See the installation documentation for configuring OpenShift Container Platform network customizations on your chosen cloud provider for more information.

Procedure

1. Change to the directory that contains the installation program and create the manifests:

   ```
   $ ./openshift-install create manifests --dir=<installation_directory>
   ```

   where:

   <installation_directory>
   Specifies the name of the directory that contains the **install-config.yaml** file for your cluster.

2. Create a stub manifest file for the advanced network configuration that is named **cluster-network-03-config.yml** in the **<installation_directory>/manifests/** directory:

   ```
   $ cat <<EOF > <installation_directory>/manifests/cluster-network-03-config.yml
   apiVersion: operator.openshift.io/v1
   kind: Network
   metadata:
     name: cluster
   spec:
   EOF
   ```

   where:

   <installation_directory>
   Specifies the directory name that contains the **manifests/** directory for your cluster.

3. Open the **cluster-network-03-config.yml** file in an editor and configure OVN-Kubernetes with hybrid networking, such as in the following example:

   #### Specify a hybrid networking configuration

   ```
   apiVersion: operator.openshift.io/v1
   kind: Network
   metadata:
     name: cluster
   spec:
     defaultNetwork:
       ovnKubernetesConfig:
         hybridOverlayConfig:
           hybridClusterNetwork: 1
   ```

```
    - cidr: 10.132.0.0/14
      hostPrefix: 23
    hybridOverlayVXLANPort: 9898 2
```

1. Specify the CIDR configuration used for nodes on the additional overlay network. The **hybridClusterNetwork** CIDR cannot overlap with the **clusterNetwork** CIDR.

2. 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 **4789** port. For more information on this requirement, see the Microsoft documentation on Pod-to-pod connectivity between hosts is broken.

> **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.

4. Save the **cluster-network-03-config.yml** file and quit the text editor.

5. Optional: Back up the **manifests/cluster-network-03-config.yml** file. The installation program deletes the **manifests/** directory when creating the cluster.

Complete any further installation configurations, and then create your cluster. Hybrid networking is enabled when the installation process is finished.

15.17.2. 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

CHAPTER 16. CONFIGURING ROUTES

16.1. ROUTE CONFIGURATION

16.1.1. Configuring route timeouts

You can configure the default timeouts for an existing route when you have services in need of a low timeout, which is required for Service Level Availability (SLA) purposes, or a high timeout, for cases with a slow back end.

Prerequisites

- You need a deployed Ingress Controller on a running cluster.

Procedure

1. Using the **oc annotate** command, add the timeout to the route:

```
$ oc annotate route <route_name> \
    --overwrite haproxy.router.openshift.io/timeout=<timeout><time_unit> 1
```

1 Supported time units are microseconds (us), milliseconds (ms), seconds (s), minutes (m), hours (h), or days (d).

The following example sets a timeout of two seconds on a route named **myroute**:

```
$ oc annotate route myroute --overwrite haproxy.router.openshift.io/timeout=2s
```

16.1.2. Enabling HTTP strict transport security

HTTP Strict Transport Security (HSTS) policy is a security enhancement, which ensures that only HTTPS traffic is allowed on the host. Any HTTP requests are dropped by default. This is useful for ensuring secure interactions with websites, or to offer a secure application for the user's benefit.

When HSTS is enabled, HSTS adds a Strict Transport Security header to HTTPS responses from the site. You can use the **insecureEdgeTerminationPolicy** value in a route to redirect to send HTTP to HTTPS. However, when HSTS is enabled, the client changes all requests from the HTTP URL to HTTPS before the request is sent, eliminating the need for a redirect. This is not required to be supported by the client, and can be disabled by setting **max-age=0**.

> **IMPORTANT**
>
> HSTS works only with secure routes (either edge terminated or re-encrypt). The configuration is ineffective on HTTP or passthrough routes.

Procedure

- To enable HSTS on a route, add the **haproxy.router.openshift.io/hsts_header** value to the edge terminated or re-encrypt route:

```
apiVersion: v1
```

```
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/hsts_header: max-age=31536000;includeSubDomains;preload
1 2 3
```

1. **max-age** is the only required parameter. It measures the length of time, in seconds, that the HSTS policy is in effect. The client updates **max-age** whenever a response with a HSTS header is received from the host. When **max-age** times out, the client discards the policy.

2. **includeSubDomains** is optional. When included, it tells the client that all subdomains of the host are to be treated the same as the host.

3. **preload** is optional. When **max-age** is greater than 0, then including **preload** in **haproxy.router.openshift.io/hsts_header** allows external services to include this site in their HSTS preload lists. For example, sites such as Google can construct a list of sites that have **preload** set. Browsers can then use these lists to determine which sites they can communicate with over HTTPS, before they have interacted with the site. Without **preload** set, browsers must have interacted with the site over HTTPS to get the header.

16.1.3. Troubleshooting throughput issues

Sometimes applications deployed through OpenShift Container Platform can cause network throughput issues such as unusually high latency between specific services.

Use the following methods to analyze performance issues if pod logs do not reveal any cause of the problem:

- Use a packet analyzer, such as ping or tcpdump to analyze traffic between a pod and its node. For example, run the tcpdump tool on each pod while reproducing the behavior that led to the issue. Review the captures on both sides to compare send and receive timestamps to analyze the latency of traffic to and from a pod. Latency can occur in OpenShift Container Platform if a node interface is overloaded with traffic from other pods, storage devices, or the data plane.

  ```
  $ tcpdump -s 0 -i any -w /tmp/dump.pcap host <podip 1> && host <podip 2> 1
  ```

  1. **podip** is the IP address for the pod. Run the **oc get pod <pod_name> -o wide** command to get the IP address of a pod.

  tcpdump generates a file at **/tmp/dump.pcap** containing all traffic between these two pods. Ideally, run the analyzer shortly before the issue is reproduced and stop the analyzer shortly after the issue is finished reproducing to minimize the size of the file. You can also run a packet analyzer between the nodes (eliminating the SDN from the equation) with:

  ```
  $ tcpdump -s 0 -i any -w /tmp/dump.pcap port 4789
  ```

- Use a bandwidth measuring tool, such as iperf, to measure streaming throughput and UDP throughput. Run the tool from the pods first, then from the nodes, to locate any bottlenecks.
  - For information on installing and using iperf, see this Red Hat Solution.

16.1.4. Using cookies to keep route statefulness

OpenShift Container Platform provides sticky sessions, which enables stateful application traffic by ensuring all traffic hits the same endpoint. However, if the endpoint pod terminates, whether through restart, scaling, or a change in configuration, this statefulness can disappear.

OpenShift Container Platform can use cookies to configure session persistence. The Ingress controller selects an endpoint to handle any user requests, and creates a cookie for the session. The cookie is passed back in the response to the request and the user sends the cookie back with the next request in the session. The cookie tells the Ingress Controller which endpoint is handling the session, ensuring that client requests use the cookie so that they are routed to the same pod.

16.1.4.1. Annotating a route with a cookie

You can set a cookie name to overwrite the default, auto-generated one for the route. This allows the application receiving route traffic to know the cookie name. By deleting the cookie it can force the next request to re-choose an endpoint. So, if a server was overloaded it tries to remove the requests from the client and redistribute them.

Procedure

1. Annotate the route with the specified cookie name:

   ```
   $ oc annotate route <route_name> router.openshift.io/cookie_name="<cookie_name>"
   ```

   where:

   <route_name>
   Specifies the name of the route.

   <cookie_name>
   Specifies the name for the cookie.

   For example, to annotate the route **my_route** with the cookie name **my_cookie**:

   ```
   $ oc annotate route my_route router.openshift.io/cookie_name="my_cookie"
   ```

2. Capture the route hostname in a variable:

   ```
   $ ROUTE_NAME=$(oc get route <route_name> -o jsonpath='{.spec.host}')
   ```

   where:

   <route_name>
   Specifies the name of the route.

3. Save the cookie, and then access the route:

   ```
   $ curl $ROUTE_NAME -k -c /tmp/cookie_jar
   ```

   Use the cookie saved by the previous command when connecting to the route:

   ```
   $ curl $ROUTE_NAME -k -b /tmp/cookie_jar
   ```

16.1.5. Path-based routes

Path-based routes specify a path component that can be compared against a URL, which requires that the traffic for the route be HTTP based. Thus, multiple routes can be served using the same hostname, each with a different path. Routers should match routes based on the most specific path to the least. However, this depends on the router implementation.

The following table shows example routes and their accessibility:

Table 16.1. Route availability

Route	When Compared to	Accessible
www.example.com/test	*www.example.com/test*	Yes
	www.example.com	No
www.example.com/test and *www.example.com*	*www.example.com/test*	Yes
	www.example.com	Yes
www.example.com	*www.example.com/text*	Yes (Matched by the host, not the route)
	www.example.com	Yes

An unsecured route with a path

```
apiVersion: v1
kind: Route
metadata:
  name: route-unsecured
spec:
  host: www.example.com
  path: "/test" (1)
  to:
    kind: Service
    name: service-name
```

1. The path is the only added attribute for a path-based route.

NOTE

Path-based routing is not available when using passthrough TLS, as the router does not terminate TLS in that case and cannot read the contents of the request.

16.1.6. Route-specific annotations

The Ingress Controller can set the default options for all the routes it exposes. An individual route can override some of these defaults by providing specific configurations in its annotations. Red Hat does not support adding a route annotation to an operator-managed route.

IMPORTANT

To create a whitelist with multiple source IPs or subnets, use a space-delimited list. Any other delimiter type causes the list to be ignored without a warning or error message.

Table 16.2. Route annotations

Variable	Description	Environment variable used as default
haproxy.router.openshift.io/balance	Sets the load-balancing algorithm. Available options are **random**, **source**, **roundrobin**, and **leastconn**.	**ROUTER_TCP_BALANCE_SCHEME** for passthrough routes. Otherwise, use **ROUTER_LOAD_BALANCE_ALGORITHM**.
haproxy.router.openshift.io/disable_cookies	Disables the use of cookies to track related connections. If set to **true** or **TRUE**, the balance algorithm is used to choose which back-end serves connections for each incoming HTTP request.	
router.openshift.io/cookie_name	Specifies an optional cookie to use for this route. The name must consist of any combination of upper and lower case letters, digits, "_", and "-". The default is the hashed internal key name for the route.	
haproxy.router.openshift.io/pod-concurrent-connections	Sets the maximum number of connections that are allowed to a backing pod from a router. Note: If there are multiple pods, each can have this many connections. If you have multiple routers, there is no coordination among them, each may connect this many times. If not set, or set to 0, there is no limit.	
haproxy.router.openshift.io/rate-limit-connections	Setting **true** or **TRUE** enables rate limiting functionality which is implemented through stick-tables on the specific backend per route. Note: Using this annotation provides basic protection against distributed denial-of-service (DDoS) attacks.	

Variable	Description	Environment variable used as default
haproxy.router.openshift.io/rate-limit-connections.concurrent-tcp	Limits the number of concurrent TCP connections made through the same source IP address. It accepts a numeric value. Note: Using this annotation provides basic protection against distributed denial-of-service (DDoS) attacks.	
haproxy.router.openshift.io/rate-limit-connections.rate-http	Limits the rate at which a client with the same source IP address can make HTTP requests. It accepts a numeric value. Note: Using this annotation provides basic protection against distributed denial-of-service (DDoS) attacks.	
haproxy.router.openshift.io/rate-limit-connections.rate-tcp	Limits the rate at which a client with the same source IP address can make TCP connections. It accepts a numeric value. Note: Using this annotation provides basic protection against distributed denial-of-service (DDoS) attacks.	
haproxy.router.openshift.io/timeout	Sets a server-side timeout for the route. (TimeUnits)	**ROUTER_DEFAULT_SERVER_TIMEOUT**
router.openshift.io/haproxy.health.check.interval	Sets the interval for the back-end health checks. (TimeUnits)	**ROUTER_BACKEND_CHECK_INTERVAL**
haproxy.router.openshift.io/ip_whitelist	Sets a whitelist for the route. The whitelist is a space-separated list of IP addresses and CIDR ranges for the approved source addresses. Requests from IP addresses that are not in the whitelist are dropped. The maximum number of IP addresses and CIDR ranges allowed in a whitelist is 61.	
haproxy.router.openshift.io/hsts_header	Sets a Strict-Transport-Security header for the edge terminated or re-encrypt route.	

Variable	Description	Environment variable used as default
haproxy.router.openshift.io/log-send-hostname	Sets the **hostname** field in the Syslog header. Uses the hostname of the system. **log-send-hostname** is enabled by default if any Ingress API logging method, such as sidecar or Syslog facility, is enabled for the router.	
haproxy.router.openshift.io/rewrite-target	Sets the rewrite path of the request on the backend.	
router.openshift.io/cookie-same-site	Sets a value to restrict cookies. The values are: **Lax**: cookies are transferred between the visited site and third-party sites. **Strict**: cookies are restricted to the visited site. **None**: cookies are restricted to the visited site. This value is applicable to re-encrypt and edge routes only. For more information, see the SameSite cookies documentation.	
haproxy.router.openshift.io/set-forwarded-headers	Sets the policy for handling the **Forwarded** and **X-Forwarded-For** HTTP headers per route. The values are: **append**: appends the header, preserving any existing header. This is the default value. **replace**: sets the header, removing any existing header. **never**: never sets the header, but preserves any existing header. **if-none**: sets the header if it is not already set.	**ROUTER_SET_FORWARDED_HEADERS**

NOTE

Environment variables cannot be edited.

A route setting custom timeout

```
apiVersion: v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/timeout: 5500ms 1
...
```

1 Specifies the new timeout with HAProxy supported units (**us**, **ms**, **s**, **m**, **h**, **d**). If the unit is not provided, **ms** is the default.

> **NOTE**
>
> Setting a server-side timeout value for passthrough routes too low can cause WebSocket connections to timeout frequently on that route.

A route that allows only one specific IP address

```
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 192.168.1.10
```

A route that allows several IP addresses

```
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 192.168.1.10 192.168.1.11 192.168.1.12
```

A route that allows an IP address CIDR network

```
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 192.168.1.0/24
```

A route that allows both IP an address and IP address CIDR networks

```
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 180.5.61.153 192.168.1.0/24 10.0.0.0/8
```

A route specifying a rewrite target

```
apiVersion: v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/rewrite-target: / 1
...
```

1 Sets / as rewrite path of the request on the backend.

Setting the **haproxy.router.openshift.io/rewrite-target** annotation on a route specifies that the Ingress Controller should rewrite paths in HTTP requests using this route before forwarding the requests to the backend application. The part of the request path that matches the path specified in **spec.path** is replaced with the rewrite target specified in the annotation.

The following table provides examples of the path rewriting behavior for various combinations of **spec.path**, request path, and rewrite target.

Table 16.3. rewrite-target examples:

Route.spec.path	Request path	Rewrite target	Forwarded request path
/foo	/foo	/	/
/foo	/foo/	/	/
/foo	/foo/bar	/	/bar
/foo	/foo/bar/	/	/bar/
/foo	/foo	/bar	/bar
/foo	/foo/	/bar	/bar/
/foo	/foo/bar	/baz	/baz/bar
/foo	/foo/bar/	/baz	/baz/bar/
/foo/	/foo	/	N/A (request path does not match route path)
/foo/	/foo/	/	/
/foo/	/foo/bar	/	/bar

16.1.7. Configuring the route admission policy

Administrators and application developers can run applications in multiple namespaces with the same domain name. This is for organizations where multiple teams develop microservices that are exposed on the same hostname.

> **WARNING**
>
> Allowing claims across namespaces should only be enabled for clusters with trust between namespaces, otherwise a malicious user could take over a hostname. For this reason, the default admission policy disallows hostname claims across namespaces.

Prerequisites

- Cluster administrator privileges.

Procedure

- Edit the **.spec.routeAdmission** field of the **ingresscontroller** resource variable using the following command:

```
$ oc -n openshift-ingress-operator patch ingresscontroller/default --patch '{"spec":
{"routeAdmission":{"namespaceOwnership":"InterNamespaceAllowed"}}}' --type=merge
```

Sample Ingress Controller configuration

```
spec:
  routeAdmission:
    namespaceOwnership: InterNamespaceAllowed
...
```

TIP

You can alternatively apply the following YAML to configure the route admission policy:

```
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: default
  namespace: openshift-ingress-operator
spec:
  routeAdmission:
    namespaceOwnership: InterNamespaceAllowed
```

16.1.8. Creating a route through an Ingress object

Some ecosystem components have an integration with Ingress resources but not with route resources. To cover this case, OpenShift Container Platform automatically creates managed route objects when an Ingress object is created. These route objects are deleted when the corresponding Ingress objects are deleted.

Procedure

1. Define an Ingress object in the OpenShift Container Platform console or by entering the oc **create** command:

YAML Definition of an Ingress

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend
  annotations:
    route.openshift.io/termination: "reencrypt" 1
spec:
  rules:
  - host: www.example.com
    http:
      paths:
      - backend:
          serviceName: frontend
          servicePort: 443
  tls:
  - hosts:
    - www.example.com
    secretName: example-com-tls-certificate
```

1 The **route.openshift.io/termination** annotation can be used to configure the **spec.tls.termination** field of the **Route** as **Ingress** has no field for this. The accepted values are **edge**, **passthrough** and **reencrypt**. All other values are silently ignored. When unset, **edge** is used.

   a. If you specify the **passthrough** value in the **route.openshift.io/termination** annotation, set **path** to **''** and **pathType** to **ImplementationSpecific** in the spec:

```yaml
spec:
  rules:
  - host: www.example.com
    http:
      paths:
      - path: ''
        pathType: ImplementationSpecific
      - backend:
          serviceName: frontend
          servicePort: 443
```

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

2. List your routes:

```
$ oc get routes
```

The result includes an autogenerated route whose name starts with **frontend-**:

```
NAME           HOST/PORT        PATH    SERVICES    PORT    TERMINATION
WILDCARD
frontend-gnztq   www.example.com          frontend    443     reencrypt/Redirect   None
```

If you inspect this route, it looks this:

YAML Definition of an autogenerated route

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: frontend-gnztq
  ownerReferences:
  - apiVersion: networking.k8s.io/v1
    controller: true
    kind: Ingress
    name: frontend
    uid: 4e6c59cc-704d-4f44-b390-617d879033b6
spec:
  host: www.example.com
  to:
    kind: Service
    name: frontend
  tls:
    certificate: |
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    insecureEdgeTerminationPolicy: Redirect
    key: |
      -----BEGIN RSA PRIVATE KEY-----
      [...]
      -----END RSA PRIVATE KEY-----
    termination: reencrypt
```

16.1.9. Configuring the OpenShift Container Platform Ingress Controller for dual-stack networking

If your OpenShift Container Platform cluster is configured for IPv4 and IPv6 dual-stack networking, your cluster is externally reachable by OpenShift Container Platform routes.

The Ingress Controller automatically serves services that have both IPv4 and IPv6 endpoints, but you can configure the Ingress Controller for single-stack or dual-stack services.

Prerequisites

- You deployed an OpenShift Container Platform cluster on bare metal.
- You installed the OpenShift CLI (**oc**).

Procedure

1. To have the Ingress Controller serve traffic over IPv4/IPv6 to a workload, you can create a service YAML file or modify an existing service YAML file by setting the **ipFamilies** and **ipFamilyPolicy** fields. For example:

Sample service YAML file

```
apiVersion: v1
 kind: Service
 metadata:
  creationTimestamp: yyyy-mm-ddT00:00:00Z
  labels:
   name: <service_name>
   manager: kubectl-create
   operation: Update
   time: yyyy-mm-ddT00:00:00Z
  name: <service_name>
  namespace: <namespace_name>
  resourceVersion: "<resource_version_number>"
  selfLink: "/api/v1/namespaces/<namespace_name>/services/<service_name>"
  uid: <uid_number>
 spec:
  clusterIP: 172.30.0.0/16
  clusterIPs: 1
  - 172.30.0.0/16
  - <second_IP_address>
  ipFamilies: 2
  - IPv4
  - IPv6
  ipFamilyPolicy: RequireDualStack 3
  ports:
  - port: 8080
   protocol: TCP
   targetport: 8080
  selector:
   name: <namespace_name>
  sessionAffinity: None
  type: ClusterIP
 status:
  loadbalancer: {}
```

1. In a dual-stack instance, there are two different **clusterIPs** provided.
2. For a single-stack instance, enter **IPv4** or **IPv6**. For a dual-stack instance, enter both **IPv4** and **IPv6**.
3. For a single-stack instance, enter **SingleStack**. For a dual-stack instance, enter **RequireDualStack**.

These resources generate corresponding **endpoints**. The Ingress Controller now watches **endpointslices**.

2. To view **endpoints**, enter the following command:

```
$ oc get endpoints
```

3. To view **endpointslices**, enter the following command:

```
$ oc get endpointslices
```

16.2. SECURED ROUTES

Secure routes provide the ability to use several types of TLS termination to serve certificates to the client. The following sections describe how to create re-encrypt, edge, and passthrough routes with custom certificates.

> **IMPORTANT**
>
> If you create routes in Microsoft Azure through public endpoints, the resource names are subject to restriction. You cannot create resources that use certain terms. For a list of terms that Azure restricts, see Resolve reserved resource name errors in the Azure documentation.

16.2.1. Creating a re-encrypt route with a custom certificate

You can configure a secure route using reencrypt TLS termination with a custom certificate by using the **oc create route** command.

Prerequisites

- You must have a certificate/key pair in PEM-encoded files, where the certificate is valid for the route host.
- You may have a separate CA certificate in a PEM-encoded file that completes the certificate chain.
- You must have a separate destination CA certificate in a PEM-encoded file.
- You must have a service that you want to expose.

> **NOTE**
>
> Password protected key files are not supported. To remove a passphrase from a key file, use the following command:
>
> ```
> $ openssl rsa -in password_protected_tls.key -out tls.key
> ```

Procedure

This procedure creates a **Route** resource with a custom certificate and reencrypt TLS termination. The following assumes that the certificate/key pair are in the **tls.crt** and **tls.key** files in the current working directory. You must also specify a destination CA certificate to enable the Ingress Controller to trust the service's certificate. You may also specify a CA certificate if needed to complete the certificate chain. Substitute the actual path names for **tls.crt**, **tls.key**, **cacert.crt**, and (optionally) **ca.crt**. Substitute the name of the **Service** resource that you want to expose for **frontend**. Substitute the appropriate hostname for **www.example.com**.

- Create a secure **Route** resource using reencrypt TLS termination and a custom certificate:

```
$ oc create route reencrypt --service=frontend --cert=tls.crt --key=tls.key --dest-ca-cert=destca.crt --ca-cert=ca.crt --hostname=www.example.com
```

If you examine the resulting **Route** resource, it should look similar to the following:

YAML Definition of the Secure Route

```yaml
apiVersion: v1
kind: Route
metadata:
  name: frontend
spec:
  host: www.example.com
  to:
    kind: Service
    name: frontend
  tls:
    termination: reencrypt
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    caCertificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    destinationCACertificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

See **oc create route reencrypt --help** for more options.

16.2.2. Creating an edge route with a custom certificate

You can configure a secure route using edge TLS termination with a custom certificate by using the **oc create route** command. With an edge route, the Ingress Controller terminates TLS encryption before forwarding traffic to the destination pod. The route specifies the TLS certificate and key that the Ingress Controller uses for the route.

Prerequisites

- You must have a certificate/key pair in PEM-encoded files, where the certificate is valid for the route host.
- You may have a separate CA certificate in a PEM-encoded file that completes the certificate chain.
- You must have a service that you want to expose.

> **NOTE**
>
> Password protected key files are not supported. To remove a passphrase from a key file, use the following command:
>
> ```
> $ openssl rsa -in password_protected_tls.key -out tls.key
> ```

Procedure

This procedure creates a **Route** resource with a custom certificate and edge TLS termination. The following assumes that the certificate/key pair are in the **tls.crt** and **tls.key** files in the current working directory. You may also specify a CA certificate if needed to complete the certificate chain. Substitute the actual path names for **tls.crt**, **tls.key**, and (optionally) **ca.crt**. Substitute the name of the service that you want to expose for **frontend**. Substitute the appropriate hostname for **www.example.com**.

- Create a secure **Route** resource using edge TLS termination and a custom certificate.

  ```
  $ oc create route edge --service=frontend --cert=tls.crt --key=tls.key --ca-cert=ca.crt --
  hostname=www.example.com
  ```

  If you examine the resulting **Route** resource, it should look similar to the following:

  #### YAML Definition of the Secure Route

  ```
  apiVersion: v1
  kind: Route
  metadata:
    name: frontend
  spec:
    host: www.example.com
    to:
      kind: Service
      name: frontend
    tls:
      termination: edge
      key: |-
        -----BEGIN PRIVATE KEY-----
        [...]
        -----END PRIVATE KEY-----
      certificate: |-
        -----BEGIN CERTIFICATE-----
        [...]
        -----END CERTIFICATE-----
      caCertificate: |-
        -----BEGIN CERTIFICATE-----
        [...]
        -----END CERTIFICATE-----
  ```

  See **oc create route edge --help** for more options.

16.2.3. Creating a passthrough route

You can configure a secure route using passthrough termination by using the **oc create route** command. With passthrough termination, encrypted traffic is sent straight to the destination without the router providing TLS termination. Therefore no key or certificate is required on the route.

Prerequisites

- You must have a service that you want to expose.

Procedure

- Create a **Route** resource:

```
$ oc create route passthrough route-passthrough-secured --service=frontend --port=8080
```

If you examine the resulting **Route** resource, it should look similar to the following:

A Secured Route Using Passthrough Termination

```
apiVersion: v1
kind: Route
metadata:
  name: route-passthrough-secured (1)
spec:
  host: www.example.com
  port:
    targetPort: 8080
  tls:
    termination: passthrough (2)
    insecureEdgeTerminationPolicy: None (3)
  to:
    kind: Service
    name: frontend
```

(1) The name of the object, which is limited to 63 characters.

(2) The **termination** field is set to **passthrough**. This is the only required **tls** field.

(3) Optional **insecureEdgeTerminationPolicy**. The only valid values are **None**, **Redirect**, or empty for disabled.

The destination pod is responsible for serving certificates for the traffic at the endpoint. This is currently the only method that can support requiring client certificates, also known as two-way authentication.

CHAPTER 17. CONFIGURING INGRESS CLUSTER TRAFFIC

17.1. CONFIGURING INGRESS CLUSTER TRAFFIC OVERVIEW

OpenShift Container Platform provides the following methods for communicating from outside the cluster with services running in the cluster.

The methods are recommended, in order or preference:

- If you have HTTP/HTTPS, use an Ingress Controller.
- If you have a TLS-encrypted protocol other than HTTPS. For example, for TLS with the SNI header, use an Ingress Controller.
- Otherwise, use a Load Balancer, an External IP, or a **NodePort**.

Method	Purpose
Use an Ingress Controller	Allows access to HTTP/HTTPS traffic and TLS-encrypted protocols other than HTTPS (for example, TLS with the SNI header).
Automatically assign an external IP using a load balancer service	Allows traffic to non-standard ports through an IP address assigned from a pool.
Manually assign an external IP to a service	Allows traffic to non-standard ports through a specific IP address.
Configure a **NodePort**	Expose a service on all nodes in the cluster.

17.2. CONFIGURING EXTERNALIPS FOR SERVICES

As a cluster administrator, you can designate an IP address block that is external to the cluster that can send traffic to services in the cluster.

This functionality is generally most useful for clusters installed on bare-metal hardware.

17.2.1. Prerequisites

- Your network infrastructure must route traffic for the external IP addresses to your cluster.

17.2.2. About ExternalIP

For non-cloud environments, OpenShift Container Platform supports the assignment of external IP addresses to a **Service** object **spec.externalIPs[]** field through the **ExternalIP** facility. By setting this field, OpenShift Container Platform assigns an additional virtual IP address to the service. The IP address can be outside the service network defined for the cluster. A service configured with an ExternalIP functions similarly to a service with **type=NodePort**, allowing you to direct traffic to a local node for load balancing.

You must configure your networking infrastructure to ensure that the external IP address blocks that you define are routed to the cluster.

OpenShift Container Platform extends the ExternalIP functionality in Kubernetes by adding the following capabilities:

- Restrictions on the use of external IP addresses by users through a configurable policy
- Allocation of an external IP address automatically to a service upon request

> **WARNING**
>
> Disabled by default, use of ExternalIP functionality can be a security risk, because in-cluster traffic to an external IP address is directed to that service. This could allow cluster users to intercept sensitive traffic destined for external resources.

> **IMPORTANT**
>
> This feature is supported only in non-cloud deployments. For cloud deployments, use the load balancer services for automatic deployment of a cloud load balancer to target the endpoints of a service.

You can assign an external IP address in the following ways:

Automatic assignment of an external IP

OpenShift Container Platform automatically assigns an IP address from the **autoAssignCIDRs** CIDR block to the **spec.externalIPs[]** array when you create a **Service** object with **spec.type=LoadBalancer** set. In this case, OpenShift Container Platform implements a non-cloud version of the load balancer service type and assigns IP addresses to the services. Automatic assignment is disabled by default and must be configured by a cluster administrator as described in the following section.

Manual assignment of an external IP

OpenShift Container Platform uses the IP addresses assigned to the **spec.externalIPs[]** array when you create a **Service** object. You cannot specify an IP address that is already in use by another service.

17.2.2.1. Configuration for ExternalIP

Use of an external IP address in OpenShift Container Platform is governed by the following fields in the **Network.config.openshift.io** CR named **cluster**:

- **spec.externalIP.autoAssignCIDRs** defines an IP address block used by the load balancer when choosing an external IP address for the service. OpenShift Container Platform supports only a single IP address block for automatic assignment. This can be simpler than having to manage the port space of a limited number of shared IP addresses when manually assigning ExternalIPs to services. If automatic assignment is enabled, a **Service** object with **spec.type=LoadBalancer** is allocated an external IP address.

- **spec.externalIP.policy** defines the permissible IP address blocks when manually specifying an IP address. OpenShift Container Platform does not apply policy rules to IP address blocks defined by **spec.externalIP.autoAssignCIDRs**.

If routed correctly, external traffic from the configured external IP address block can reach service endpoints through any TCP or UDP port that the service exposes.

> **IMPORTANT**
>
> You must ensure that the IP address block you assign terminates at one or more nodes in your cluster.

OpenShift Container Platform supports both the automatic and manual assignment of IP addresses, and each address is guaranteed to be assigned to a maximum of one service. This ensures that each service can expose its chosen ports regardless of the ports exposed by other services.

> **NOTE**
>
> To use IP address blocks defined by **autoAssignCIDRs** in OpenShift Container Platform, you must configure the necessary IP address assignment and routing for your host network.

The following YAML describes a service with an external IP address configured:

Example Service object with spec.externalIPs[] set

```yaml
apiVersion: v1
kind: Service
metadata:
  name: http-service
spec:
  clusterIP: 172.30.163.110
  externalIPs:
  - 192.168.132.253
  externalTrafficPolicy: Cluster
  ports:
  - name: highport
    nodePort: 31903
    port: 30102
    protocol: TCP
    targetPort: 30102
  selector:
    app: web
  sessionAffinity: None
  type: LoadBalancer
status:
  loadBalancer:
    ingress:
    - ip: 192.168.132.253
```

17.2.2.2. Restrictions on the assignment of an external IP address

As a cluster administrator, you can specify IP address blocks to allow and to reject.

Restrictions apply only to users without **cluster-admin** privileges. A cluster administrator can always set the service **spec.externalIPs[]** field to any IP address.

You configure IP address policy with a **policy** object defined by specifying the **spec.ExternalIP.policy** field. The policy object has the following shape:

```
{
  "policy": {
    "allowedCIDRs": [],
    "rejectedCIDRs": []
  }
}
```

When configuring policy restrictions, the following rules apply:

- If **policy={}** is set, then creating a **Service** object with **spec.ExternalIPs[]** set will fail. This is the default for OpenShift Container Platform. The behavior when **policy=null** is set is identical.
- If **policy** is set and either **policy.allowedCIDRs[]** or **policy.rejectedCIDRs[]** is set, the following rules apply:
  - If **allowedCIDRs[]** and **rejectedCIDRs[]** are both set, then **rejectedCIDRs[]** has precedence over **allowedCIDRs[]**.
  - If **allowedCIDRs[]** is set, creating a **Service** object with **spec.ExternalIPs[]** will succeed only if the specified IP addresses are allowed.
  - If **rejectedCIDRs[]** is set, creating a **Service** object with **spec.ExternalIPs[]** will succeed only if the specified IP addresses are not rejected.

17.2.2.3. Example policy objects

The examples that follow demonstrate several different policy configurations.

- In the following example, the policy prevents OpenShift Container Platform from creating any service with an external IP address specified:

  Example policy to reject any value specified for Service object spec.externalIPs[]

  ```
  apiVersion: config.openshift.io/v1
  kind: Network
  metadata:
    name: cluster
  spec:
    externalIP:
      policy: {}
    ...
  ```

- In the following example, both the **allowedCIDRs** and **rejectedCIDRs** fields are set.

  Example policy that includes both allowed and rejected CIDR blocks

  ```
  apiVersion: config.openshift.io/v1
  kind: Network
  metadata:
  ```

```
  name: cluster
spec:
  externalIP:
    policy:
      allowedCIDRs:
      - 172.16.66.10/23
      rejectedCIDRs:
      - 172.16.66.10/24
  ...
```

- In the following example, **policy** is set to **null**. If set to **null**, when inspecting the configuration object by entering **oc get networks.config.openshift.io -o yaml**, the **policy** field will not appear in the output.

  Example policy to allow any value specified for Service object spec.externalIPs[]

```
apiVersion: config.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  externalIP:
    policy: null
  ...
```

17.2.3. ExternalIP address block configuration

The configuration for ExternalIP address blocks is defined by a Network custom resource (CR) named **cluster**. The Network CR is part of the **config.openshift.io** API group.

IMPORTANT

During cluster installation, the Cluster Version Operator (CVO) automatically creates a Network CR named **cluster**. Creating any other CR objects of this type is not supported.

The following YAML describes the ExternalIP configuration:

Network.config.openshift.io CR named cluster

```
apiVersion: config.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  externalIP:
    autoAssignCIDRs: [] 1
    policy: 2
      ...
```

1 Defines the IP address block in CIDR format that is available for automatic assignment of external IP addresses to a service. Only a single IP address range is allowed.

2

Defines restrictions on manual assignment of an IP address to a service. If no restrictions are defined, specifying the **spec.externalIP** field in a **Service** object is not allowed. By default, no

The following YAML describes the fields for the **policy** stanza:

Network.config.openshift.io policy stanza

```
policy:
  allowedCIDRs: [] 1
  rejectedCIDRs: [] 2
```

1 A list of allowed IP address ranges in CIDR format.

2 A list of rejected IP address ranges in CIDR format.

Example external IP configurations

Several possible configurations for external IP address pools are displayed in the following examples:

- The following YAML describes a configuration that enables automatically assigned external IP addresses:

  #### Example configuration with spec.externalIP.autoAssignCIDRs set

  ```
  apiVersion: config.openshift.io/v1
  kind: Network
  metadata:
    name: cluster
  spec:
    ...
    externalIP:
      autoAssignCIDRs:
      - 192.168.132.254/29
  ```

- The following YAML configures policy rules for the allowed and rejected CIDR ranges:

  #### Example configuration with spec.externalIP.policy set

  ```
  apiVersion: config.openshift.io/v1
  kind: Network
  metadata:
    name: cluster
  spec:
    ...
    externalIP:
      policy:
        allowedCIDRs:
        - 192.168.132.0/29
        - 192.168.132.8/29
        rejectedCIDRs:
        - 192.168.132.7/32
  ```

17.2.4. Configure external IP address blocks for your cluster

As a cluster administrator, you can configure the following ExternalIP settings:

- An ExternalIP address block used by OpenShift Container Platform to automatically populate the **spec.clusterIP** field for a **Service** object.
- A policy object to restrict what IP addresses may be manually assigned to the **spec.clusterIP** array of a **Service** object.

Prerequisites

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

Procedure

1. Optional: To display the current external IP configuration, enter the following command:

```
$ oc describe networks.config cluster
```

2. To edit the configuration, enter the following command:

```
$ oc edit networks.config cluster
```

3. Modify the ExternalIP configuration, as in the following example:

```
apiVersion: config.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  ...
  externalIP: 1
  ...
```

1 Specify the configuration for the **externalIP** stanza.

4. To confirm the updated ExternalIP configuration, enter the following command:

```
$ oc get networks.config cluster -o go-template='{{.spec.externalIP}}{{"\n"}}'
```

17.2.5. Next steps

- Configuring ingress cluster traffic for a service external IP

17.3. CONFIGURING INGRESS CLUSTER TRAFFIC USING AN INGRESS CONTROLLER

OpenShift Container Platform provides methods for communicating from outside the cluster with services running in the cluster. This method uses an Ingress Controller.

17.3.1. Using Ingress Controllers and routes

The Ingress Operator manages Ingress Controllers and wildcard DNS.

Using an Ingress Controller is the most common way to allow external access to an OpenShift Container Platform cluster.

An Ingress Controller is configured to accept external requests and proxy them based on the configured routes. This is limited to HTTP, HTTPS using SNI, and TLS using SNI, which is sufficient for web applications and services that work over TLS with SNI.

Work with your administrator to configure an Ingress Controller to accept external requests and proxy them based on the configured routes.

The administrator can create a wildcard DNS entry and then set up an Ingress Controller. Then, you can work with the edge Ingress Controller without having to contact the administrators.

By default, every ingress controller in the cluster can admit any route created in any project in the cluster.

The Ingress Controller:

- Has two replicas by default, which means it should be running on two worker nodes.
- Can be scaled up to have more replicas on more nodes.

> **NOTE**
>
> The procedures in this section require prerequisites performed by the cluster administrator.

17.3.2. Prerequisites

Before starting the following procedures, the administrator must:

- Set up the external port to the cluster networking environment so that requests can reach the cluster.
- Make sure there is at least one user with cluster admin role. To add this role to a user, run the following command:

  ```
  $ oc adm policy add-cluster-role-to-user cluster-admin username
  ```

- Have an OpenShift Container Platform cluster with at least one master and at least one node and a system outside the cluster that has network access to the cluster. This procedure assumes that the external system is on the same subnet as the cluster. The additional networking required for external systems on a different subnet is out-of-scope for this topic.

17.3.3. Creating a project and service

If the project and service that you want to expose do not exist, first create the project, then the service.

If the project and service already exist, skip to the procedure on exposing the service to create a route.

Prerequisites

- Install the **oc** CLI and log in as a cluster administrator.

Procedure

1. Create a new project for your service:

```
$ oc new-project <project_name>
```

For example:

```
$ oc new-project myproject
```

2. Use the **oc new-app** command to create a service. For example:

```
$ oc new-app \
    -e MYSQL_USER=admin \
    -e MYSQL_PASSWORD=redhat \
    -e MYSQL_DATABASE=mysqldb \
    registry.redhat.io/rhscl/mysql-80-rhel7
```

3. Run the following command to see that the new service is created:

```
$ oc get svc -n myproject
```

Example output

```
NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
mysql-80-rhel7   ClusterIP   172.30.63.31   <none>        3306/TCP   4m55s
```

By default, the new service does not have an external IP address.

17.3.4. Exposing the service by creating a route

You can expose the service as a route by using the **oc expose** command.

Procedure

To expose the service:

1. Log in to OpenShift Container Platform.

2. Log in to the project where the service you want to expose is located:

```
$ oc project myproject
```

3. Run the following command to expose the route:

```
$ oc expose service <service_name>
```

For example:

```
$ oc expose service mysql-80-rhel7
```

Example output

```
route "mysql-80-rhel7" exposed
```

4. Use a tool, such as cURL, to make sure you can reach the service using the cluster IP address for the service:

```
$ curl <pod_ip>:<port>
```

For example:

```
$ curl 172.30.131.89:3306
```

The examples in this section uses a MySQL service, which requires a client application. If you get a string of characters with the **Got packets out of order** message, you are connected to the service.

If you have a MySQL client, log in with the standard CLI command:

```
$ mysql -h 172.30.131.89 -u admin -p
```

Example output

```
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.

MySQL [(none)]>
```

17.3.5. Configuring Ingress Controller sharding by using route labels

Ingress Controller sharding by using route labels means that the Ingress Controller serves any route in any namespace that is selected by the route selector.

Ingress Controller sharding is useful when balancing incoming traffic load among a set of Ingress Controllers and when isolating traffic to a specific Ingress Controller. For example, company A goes to one Ingress Controller and company B to another.

Procedure

1. Edit the **router-internal.yaml** file:

```
# cat router-internal.yaml
apiVersion: v1
items:
- apiVersion: operator.openshift.io/v1
  kind: IngressController
  metadata:
    name: sharded
    namespace: openshift-ingress-operator
  spec:
    domain: <apps-sharded.basedomain.example.net>
    nodePlacement:
      nodeSelector:
```

```
        matchLabels:
          node-role.kubernetes.io/worker: ""
    routeSelector:
      matchLabels:
        type: sharded
  status: {}
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

2. Apply the Ingress Controller **router-internal.yaml** file:

```
# oc apply -f router-internal.yaml
```

The Ingress Controller selects routes in any namespace that have the label **type: sharded**.

17.3.6. Configuring Ingress Controller sharding by using namespace labels

Ingress Controller sharding by using namespace labels means that the Ingress Controller serves any route in any namespace that is selected by the namespace selector.

Ingress Controller sharding is useful when balancing incoming traffic load among a set of Ingress Controllers and when isolating traffic to a specific Ingress Controller. For example, company A goes to one Ingress Controller and company B to another.

Procedure

1. Edit the **router-internal.yaml** file:

```
# cat router-internal.yaml
```

Example output

```
apiVersion: v1
items:
- apiVersion: operator.openshift.io/v1
  kind: IngressController
  metadata:
    name: sharded
    namespace: openshift-ingress-operator
  spec:
    domain: <apps-sharded.basedomain.example.net>
    nodePlacement:
      nodeSelector:
        matchLabels:
          node-role.kubernetes.io/worker: ""
    namespaceSelector:
      matchLabels:
        type: sharded
  status: {}
kind: List
```

```
metadata:
  resourceVersion: ""
  selfLink: ""
```

2. Apply the Ingress Controller **router-internal.yaml** file:

```
# oc apply -f router-internal.yaml
```

The Ingress Controller selects routes in any namespace that is selected by the namespace selector that have the label **type: sharded**.

17.3.7. Additional resources

- The Ingress Operator manages wildcard DNS. For more information, see Ingress Operator in OpenShift Container Platform, Installing a cluster on bare metal , and Installing a cluster on vSphere.

17.4. CONFIGURING INGRESS CLUSTER TRAFFIC USING A LOAD BALANCER

OpenShift Container Platform provides methods for communicating from outside the cluster with services running in the cluster. This method uses a load balancer.

17.4.1. Using a load balancer to get traffic into the cluster

If you do not need a specific external IP address, you can configure a load balancer service to allow external access to an OpenShift Container Platform cluster.

A load balancer service allocates a unique IP. The load balancer has a single edge router IP, which can be a virtual IP (VIP), but is still a single machine for initial load balancing.

> **NOTE**
>
> If a pool is configured, it is done at the infrastructure level, not by a cluster administrator.

> **NOTE**
>
> The procedures in this section require prerequisites performed by the cluster administrator.

17.4.2. Prerequisites

Before starting the following procedures, the administrator must:

- Set up the external port to the cluster networking environment so that requests can reach the cluster.
- Make sure there is at least one user with cluster admin role. To add this role to a user, run the following command:

```
$ oc adm policy add-cluster-role-to-user cluster-admin username
```

- Have an OpenShift Container Platform cluster with at least one master and at least one node and a system outside the cluster that has network access to the cluster. This procedure assumes that the external system is on the same subnet as the cluster. The additional networking required for external systems on a different subnet is out-of-scope for this topic.

17.4.3. Creating a project and service

If the project and service that you want to expose do not exist, first create the project, then the service.

If the project and service already exist, skip to the procedure on exposing the service to create a route.

Prerequisites

- Install the **oc** CLI and log in as a cluster administrator.

Procedure

1. Create a new project for your service:

```
$ oc new-project <project_name>
```

   For example:

```
$ oc new-project myproject
```

2. Use the **oc new-app** command to create a service. For example:

```
$ oc new-app \
    -e MYSQL_USER=admin \
    -e MYSQL_PASSWORD=redhat \
    -e MYSQL_DATABASE=mysqldb \
    registry.redhat.io/rhscl/mysql-80-rhel7
```

3. Run the following command to see that the new service is created:

```
$ oc get svc -n myproject
```

   Example output

```
NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
mysql-80-rhel7   ClusterIP   172.30.63.31   <none>        3306/TCP   4m55s
```

   By default, the new service does not have an external IP address.

17.4.4. Exposing the service by creating a route

You can expose the service as a route by using the **oc expose** command.

Procedure

To expose the service:

1. Log in to OpenShift Container Platform.

2. Log in to the project where the service you want to expose is located:

```
$ oc project myproject
```

3. Run the following command to expose the route:

```
$ oc expose service <service_name>
```

For example:

```
$ oc expose service mysql-80-rhel7
```

Example output

```
route "mysql-80-rhel7" exposed
```

4. Use a tool, such as cURL, to make sure you can reach the service using the cluster IP address for the service:

```
$ curl <pod_ip>:<port>
```

For example:

```
$ curl 172.30.131.89:3306
```

The examples in this section uses a MySQL service, which requires a client application. If you get a string of characters with the **Got packets out of order** message, you are connected to the service.

If you have a MySQL client, log in with the standard CLI command:

```
$ mysql -h 172.30.131.89 -u admin -p
```

Example output

```
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.

MySQL [(none)]>
```

17.4.5. Creating a load balancer service

Use the following procedure to create a load balancer service.

Prerequisites

- Make sure that the project and service you want to expose exist.

Procedure

To create a load balancer service:

1. Log in to OpenShift Container Platform.

2. Load the project where the service you want to expose is located.

```
$ oc project project1
```

3. Open a text file on the control plane node (also known as the master node) and paste the following text, editing the file as needed:

Sample load balancer configuration file

```
apiVersion: v1
kind: Service
metadata:
  name: egress-2 1
spec:
  ports:
  - name: db
    port: 3306 2
  loadBalancerIP:
  loadBalancerSourceRanges: 3
  - 10.0.0.0/8
  - 192.168.0.0/16
  type: LoadBalancer 4
  selector:
    name: mysql 5
```

1. Enter a descriptive name for the load balancer service.
2. Enter the same port that the service you want to expose is listening on.
3. Enter a list of specific IP addresses to restrict traffic through the load balancer. This field is ignored if the cloud-provider does not support the feature.
4. Enter **Loadbalancer** as the type.
5. Enter the name of the service.

> **NOTE**
>
> To restrict traffic through the load balancer to specific IP addresses, it is recommended to use the **service.beta.kubernetes.io/load-balancer-source-ranges** annotation rather than setting the **loadBalancerSourceRanges** field. With the annotation, you can more easily migrate to the OpenShift API, which will be implemented in a future release.

4. Save and exit the file.

5. Run the following command to create the service:

```
$ oc create -f <file-name>
```

For example:

```
$ oc create -f mysql-lb.yaml
```

6. Execute the following command to view the new service:

```
$ oc get svc
```

Example output

```
NAME       TYPE           CLUSTER-IP      EXTERNAL-IP                              PORT(S)
AGE
egress-2   LoadBalancer   172.30.22.226   ad42f5d8b303045-487804948.example.com
3306:30357/TCP   15m
```

The service has an external IP address automatically assigned if there is a cloud provider enabled.

7. On the master, use a tool, such as cURL, to make sure you can reach the service using the public IP address:

```
$ curl <public-ip>:<port>
```

For example:

```
$ curl 172.29.121.74:3306
```

The examples in this section use a MySQL service, which requires a client application. If you get a string of characters with the **Got packets out of order** message, you are connecting with the service:

If you have a MySQL client, log in with the standard CLI command:

```
$ mysql -h 172.30.131.89 -u admin -p
```

Example output

```
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.

MySQL [(none)]>
```

17.5. CONFIGURING INGRESS CLUSTER TRAFFIC ON AWS USING A NETWORK LOAD BALANCER

OpenShift Container Platform provides methods for communicating from outside the cluster with services running in the cluster. This method uses a Network Load Balancer (NLB), which forwards the client’s IP address to the node. You can configure an NLB on a new or existing AWS cluster.

17.5.1. Configuring an Ingress Controller Network Load Balancer on an existing AWS cluster

You can create an Ingress Controller backed by an AWS Network Load Balancer (NLB) on an existing cluster.

Prerequisites

- You must have an installed AWS cluster.
- **PlatformStatus** of the infrastructure resource must be AWS.
  - To verify that the **PlatformStatus** is AWS, run:

```
$ oc get infrastructure/cluster -o jsonpath='{.status.platformStatus.type}'
AWS
```

Procedure

Create an Ingress Controller backed by an AWS NLB on an existing cluster.

1. Create the Ingress Controller manifest:

```
$ cat ingresscontroller-aws-nlb.yaml
```

Example output

```
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: $my_ingress_controller 1
  namespace: openshift-ingress-operator
spec:
  domain: $my_unique_ingress_domain 2
  endpointPublishingStrategy:
    type: LoadBalancerService
    loadBalancer:
      scope: External 3
      providerParameters:
        type: AWS
        aws:
          type: NLB
```

1 Replace **$my_ingress_controller** with a unique name for the Ingress Controller.

2 Replace **$my_unique_ingress_domain** with a domain name that is unique among all Ingress Controllers in the cluster.

3 You can replace **External** with **Internal** to use an internal NLB.

2. Create the resource in the cluster:

```
$ oc create -f ingresscontroller-aws-nlb.yaml
```

IMPORTANT

Before you can configure an Ingress Controller NLB on a new AWS cluster, you must complete the Creating the installation configuration file procedure.

17.5.2. Configuring an Ingress Controller Network Load Balancer on a new AWS cluster

You can create an Ingress Controller backed by an AWS Network Load Balancer (NLB) on a new cluster.

Prerequisites

- Create the **install-config.yaml** file and complete any modifications to it.

Procedure

Create an Ingress Controller backed by an AWS NLB on a new cluster.

1. Change to the directory that contains the installation program and create the manifests:

```
$ ./openshift-install create manifests --dir=<installation_directory> 1
```

1 For **<installation_directory>**, specify the name of the directory that contains the **install-config.yaml** file for your cluster.

2. Create a file that is named **cluster-ingress-default-ingresscontroller.yaml** in the **<installation_directory>/manifests/** directory:

```
$ touch <installation_directory>/manifests/cluster-ingress-default-ingresscontroller.yaml 1
```

1 For **<installation_directory>**, specify the directory name that contains the **manifests/** directory for your cluster.

After creating the file, several network configuration files are in the **manifests/** directory, as shown:

```
$ ls <installation_directory>/manifests/cluster-ingress-default-ingresscontroller.yaml
```

Example output

```
cluster-ingress-default-ingresscontroller.yaml
```

3. Open the **cluster-ingress-default-ingresscontroller.yaml** file in an editor and enter a custom resource (CR) that describes the Operator configuration you want:

```
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  creationTimestamp: null
  name: default
  namespace: openshift-ingress-operator
spec:
```

```
endpointPublishingStrategy:
  loadBalancer:
    scope: External
    providerParameters:
      type: AWS
      aws:
        type: NLB
  type: LoadBalancerService
```

4. Save the **cluster-ingress-default-ingresscontroller.yaml** file and quit the text editor.

5. Optional: Back up the **manifests/cluster-ingress-default-ingresscontroller.yaml** file. The installation program deletes the **manifests/** directory when creating the cluster.

17.5.3. Additional resources

- Installing a cluster on AWS with network customizations .

- For more information, see Network Load Balancer support on AWS.

17.6. CONFIGURING INGRESS CLUSTER TRAFFIC FOR A SERVICE EXTERNAL IP

You can attach an external IP address to a service so that it is available to traffic outside the cluster. This is generally useful only for a cluster installed on bare metal hardware. The external network infrastructure must be configured correctly to route traffic to the service.

17.6.1. Prerequisites

- Your cluster is configured with ExternalIPs enabled. For more information, read Configuring ExternalIPs for services.

17.6.2. Attaching an ExternalIP to a service

You can attach an ExternalIP to a service. If your cluster is configured to allocate an ExternalIP automatically, you might not need to manually attach an ExternalIP to the service.

Procedure

1. Optional: To confirm what IP address ranges are configured for use with ExternalIP, enter the following command:

   ```
   $ oc get networks.config cluster -o jsonpath='{.spec.externalIP}{"\n"}'
   ```

   If **autoAssignCIDRs** is set, OpenShift Container Platform automatically assigns an ExternalIP to a new **Service** object if the **spec.externalIPs** field is not specified.

2. Attach an ExternalIP to the service.

   a. If you are creating a new service, specify the **spec.externalIPs** field and provide an array of one or more valid IP addresses. For example:

   ```
   apiVersion: v1
   kind: Service
   ```

```
metadata:
  name: svc-with-externalip
spec:
  ...
  externalIPs:
  - 192.174.120.10
```

b. If you are attaching an ExternalIP to an existing service, enter the following command. Replace **<name>** with the service name. Replace **<ip_address>** with a valid ExternalIP address. You can provide multiple IP addresses separated by commas.

```
$ oc patch svc <name> -p \
  '{
    "spec": {
      "externalIPs": [ "<ip_address>" ]
    }
  }'
```

For example:

```
$ oc patch svc mysql-55-rhel7 -p '{"spec":{"externalIPs":["192.174.120.10"]}}'
```

Example output

```
"mysql-55-rhel7" patched
```

3. To confirm that an ExternalIP address is attached to the service, enter the following command. If you specified an ExternalIP for a new service, you must create the service first.

```
$ oc get svc
```

Example output

```
NAME             CLUSTER-IP      EXTERNAL-IP     PORT(S)    AGE
mysql-55-rhel7   172.30.131.89   192.174.120.10  3306/TCP   13m
```

17.6.3. Additional resources

- Configuring ExternalIPs for services

17.7. CONFIGURING INGRESS CLUSTER TRAFFIC USING A NODEPORT

OpenShift Container Platform provides methods for communicating from outside the cluster with services running in the cluster. This method uses a **NodePort**.

17.7.1. Using a NodePort to get traffic into the cluster

Use a **NodePort**-type **Service** resource to expose a service on a specific port on all nodes in the cluster. The port is specified in the **Service** resource's **.spec.ports[*].nodePort** field.

IMPORTANT

Using a node port requires additional port resources.

A **NodePort** exposes the service on a static port on the node's IP address. **NodePort**s are in the **30000** to **32767** range by default, which means a **NodePort** is unlikely to match a service's intended port. For example, port **8080** may be exposed as port **31020** on the node.

The administrator must ensure the external IP addresses are routed to the nodes.

NodePorts and external IPs are independent and both can be used concurrently.

NOTE

The procedures in this section require prerequisites performed by the cluster administrator.

17.7.2. Prerequisites

Before starting the following procedures, the administrator must:

- Set up the external port to the cluster networking environment so that requests can reach the cluster.
- Make sure there is at least one user with cluster admin role. To add this role to a user, run the following command:

```
$ oc adm policy add-cluster-role-to-user cluster-admin <user_name>
```

- Have an OpenShift Container Platform cluster with at least one master and at least one node and a system outside the cluster that has network access to the cluster. This procedure assumes that the external system is on the same subnet as the cluster. The additional networking required for external systems on a different subnet is out-of-scope for this topic.

17.7.3. Creating a project and service

If the project and service that you want to expose do not exist, first create the project, then the service.

If the project and service already exist, skip to the procedure on exposing the service to create a route.

Prerequisites

- Install the **oc** CLI and log in as a cluster administrator.

Procedure

1. Create a new project for your service:

```
$ oc new-project <project_name>
```

For example:

```
$ oc new-project myproject
```

2. Use the **oc new-app** command to create a service. For example:

```
$ oc new-app \
    -e MYSQL_USER=admin \
    -e MYSQL_PASSWORD=redhat \
    -e MYSQL_DATABASE=mysqldb \
    registry.redhat.io/rhscl/mysql-80-rhel7
```

3. Run the following command to see that the new service is created:

```
$ oc get svc -n myproject
```

Example output

```
NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
mysql-80-rhel7   ClusterIP   172.30.63.31   <none>        3306/TCP   4m55s
```

By default, the new service does not have an external IP address.

17.7.4. Exposing the service by creating a route

You can expose the service as a route by using the **oc expose** command.

Procedure

To expose the service:

1. Log in to OpenShift Container Platform.

2. Log in to the project where the service you want to expose is located:

```
$ oc project myproject
```

3. To expose a node port for the application, enter the following command. OpenShift Container Platform automatically selects an available port in the **30000-32767** range.

```
$ oc expose dc mysql-80-rhel7 --type=NodePort --name=mysql-ingress
```

4. Optional: To confirm the service is available with a node port exposed, enter the following command:

```
$ oc get svc -n myproject
```

Example output

```
NAME             TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
mysql-80-rhel7   ClusterIP   172.30.217.127   <none>        3306/TCP         9m44s
mysql-ingress    NodePort    172.30.107.72    <none>        3306:31345/TCP   39s
```

5. Optional: To remove the service created automatically by the **oc new-app** command, enter the following command:

```
$ oc delete svc mysql-80-rhel7
```

–

17.7.5. Additional resources

- Configuring the node port service range

CHAPTER 18. KUBERNETES NMSTATE

18.1. ABOUT THE KUBERNETES NMSTATE OPERATOR

The Kubernetes NMState Operator provides a Kubernetes API for performing state-driven network configuration across the OpenShift Container Platform cluster's nodes with NMState. The Kubernetes NMState Operator provides users with functionality to configure various network interface types, DNS, and routing on cluster nodes. Additionally, the daemons on the cluster nodes periodically report on the state of each node's network interfaces to the API server.

> **IMPORTANT**
>
> Kubernetes NMState Operator 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 https://access.redhat.com/support/offerings/techpreview/.

Before you can use NMState with OpenShift Container Platform, you must install the Kubernetes NMState Operator.

18.1.1. Installing the Kubernetes NMState Operator

You must install the Kubernetes NMState Operator from the web console while logged in with administrator privileges. After it is installed, the Operator can deploy the NMState State Controller as a daemon set across all of the cluster nodes.

Procedure

1. Select **Operators** → **OperatorHub**.
2. In the search field below **All Items**, enter **nmstate** and click **Enter** to search for the Kubernetes NMState Operator.
3. Click on the Kubernetes NMState Operator search result.
4. Click on **Install** to open the **Install Operator** window.
5. Under **Installed Namespace**, ensure the namespace is **openshift-nmstate**. If **openshift-nmstate** does not exist in the combo box, click on **Create Namespace** and enter **openshift-nmstate** in the **Name** field of the dialog box and press **Create**.
6. Click **Install** to install the Operator.
7. After the Operator finishes installing, click **View Operator**.
8. Under **Provided APIs**, click **Create Instance** to open the dialog box for creating an instance of **kubernetes-nmstate**.
9. In the **Name** field of the dialog box, ensure the name of the instance is **nmstate.**

> **NOTE**
>
> The name restriction is a known issue. The instance is a singleton for the entire cluster.

10. Accept the default settings and click **Create** to create the instance.

Summary

Once complete, the Operator has deployed the NMState State Controller as a daemon set across all of the cluster nodes.

18.2. OBSERVING NODE NETWORK STATE

Node network state is the network configuration for all nodes in the cluster.

18.2.1. About nmstate

OpenShift Container Platform uses **nmstate** to report on and configure the state of the node network. This makes it possible to modify network policy configuration, such as by creating a Linux bridge on all nodes, by applying a single configuration manifest to the cluster.

Node networking is monitored and updated by the following objects:

NodeNetworkState
: Reports the state of the network on that node.

NodeNetworkConfigurationPolicy
: Describes the requested network configuration on nodes. You update the node network configuration, including adding and removing interfaces, by applying a **NodeNetworkConfigurationPolicy** manifest to the cluster.

NodeNetworkConfigurationEnactment
: Reports the network policies enacted upon each node.

OpenShift Container Platform supports the use of the following nmstate interface types:

- Linux Bridge
- VLAN
- Bond
- Ethernet

> **NOTE**
>
> If your OpenShift Container Platform cluster uses OVN-Kubernetes as the default Container Network Interface (CNI) provider, you cannot attach a Linux bridge or bonding to the default interface of a host because of a change in the host network topology of OVN-Kubernetes. As a workaround, you can use a secondary network interface connected to your host, or switch to the OpenShift SDN default CNI provider.

18.2.2. Viewing the network state of a node

A **NodeNetworkState** object exists on every node in the cluster. This object is periodically updated and captures the state of the network for that node.

Procedure

1. List all the **NodeNetworkState** objects in the cluster:

```
$ oc get nns
```

2. Inspect a **NodeNetworkState** object to view the network on that node. The output in this example has been redacted for clarity:

```
$ oc get nns node01 -o yaml
```

Example output

```
apiVersion: nmstate.io/v1beta1
kind: NodeNetworkState
metadata:
  name: node01 1
status:
  currentState: 2
    dns-resolver:
...
    interfaces:
...
    route-rules:
...
    routes:
...
  lastSuccessfulUpdateTime: "2020-01-31T12:14:00Z" 3
```

1. The name of the **NodeNetworkState** object is taken from the node.

2. The **currentState** contains the complete network configuration for the node, including DNS, interfaces, and routes.

3. Timestamp of the last successful update. This is updated periodically as long as the node is reachable and can be used to evalute the freshness of the report.

18.3. UPDATING NODE NETWORK CONFIGURATION

You can update the node network configuration, such as adding or removing interfaces from nodes, by applying **NodeNetworkConfigurationPolicy** manifests to the cluster.

18.3.1. About nmstate

OpenShift Container Platform uses **nmstate** to report on and configure the state of the node network. This makes it possible to modify network policy configuration, such as by creating a Linux bridge on all nodes, by applying a single configuration manifest to the cluster.

Node networking is monitored and updated by the following objects:

NodeNetworkState

Reports the state of the network on that node.

NodeNetworkConfigurationPolicy

Describes the requested network configuration on nodes. You update the node network configuration, including adding and removing interfaces, by applying a **NodeNetworkConfigurationPolicy** manifest to the cluster.

NodeNetworkConfigurationEnactment

Reports the network policies enacted upon each node.

OpenShift Container Platform supports the use of the following nmstate interface types:

- Linux Bridge
- VLAN
- Bond
- Ethernet

> **NOTE**
>
> If your OpenShift Container Platform cluster uses OVN-Kubernetes as the default Container Network Interface (CNI) provider, you cannot attach a Linux bridge or bonding to the default interface of a host because of a change in the host network topology of OVN-Kubernetes. As a workaround, you can use a secondary network interface connected to your host, or switch to the OpenShift SDN default CNI provider.

18.3.2. Creating an interface on nodes

Create an interface on nodes in the cluster by applying a **NodeNetworkConfigurationPolicy** manifest to the cluster. The manifest details the requested configuration for the interface.

By default, the manifest applies to all nodes in the cluster. To add the interface to specific nodes, add the **spec: nodeSelector** parameter and the appropriate **<key>:<value>** for your node selector.

Procedure

1. Create the **NodeNetworkConfigurationPolicy** manifest. The following example configures a Linux bridge on all worker nodes:

```
apiVersion: nmstate.io/v1beta1
kind: NodeNetworkConfigurationPolicy
metadata:
  name: <br1-eth1-policy> 1
spec:
  nodeSelector: 2
    node-role.kubernetes.io/worker: "" 3
  desiredState:
    interfaces:
      - name: br1
        description: Linux bridge with eth1 as a port 4
        type: linux-bridge
        state: up
```

```
    ipv4:
      dhcp: true
      enabled: true
    bridge:
      options:
        stp:
          enabled: false
      port:
        - name: eth1
```

1. Name of the policy.
2. Optional: If you do not include the **nodeSelector** parameter, the policy applies to all nodes in the cluster.
3. This example uses the **node-role.kubernetes.io/worker: ""** node selector to select all worker nodes in the cluster.
4. Optional: Human-readable description for the interface.

2. Create the node network policy:

```
$ oc apply -f <br1-eth1-policy.yaml> 1
```

1. File name of the node network configuration policy manifest.

Additional resources

- Example for creating multiple interfaces in the same policy
- Examples of different IP management methods in policies

18.3.3. Confirming node network policy updates on nodes

A **NodeNetworkConfigurationPolicy** manifest describes your requested network configuration for nodes in the cluster. The node network policy includes your requested network configuration and the status of execution of the policy on the cluster as a whole.

When you apply a node network policy, a **NodeNetworkConfigurationEnactment** object is created for every node in the cluster. The node network configuration enactment is a read-only object that represents the status of execution of the policy on that node. If the policy fails to be applied on the node, the enactment for that node includes a traceback for troubleshooting.

Procedure

1. To confirm that a policy has been applied to the cluster, list the policies and their status:

```
$ oc get nncp
```

2. Optional: If a policy is taking longer than expected to successfully configure, you can inspect the requested state and status conditions of a particular policy:

```
$ oc get nncp <policy> -o yaml
```

3. Optional: If a policy is taking longer than expected to successfully configure on all nodes, you can list the status of the enactments on the cluster:

```
$ oc get nnce
```

4. Optional: To view the configuration of a particular enactment, including any error reporting for a failed configuration:

```
$ oc get nnce <node>.<policy> -o yaml
```

18.3.4. Removing an interface from nodes

You can remove an interface from one or more nodes in the cluster by editing the **NodeNetworkConfigurationPolicy** object and setting the **state** of the interface to **absent**.

Removing an interface from a node does not automatically restore the node network configuration to a previous state. If you want to restore the previous state, you will need to define that node network configuration in the policy.

If you remove a bridge or bonding interface, any node NICs in the cluster that were previously attached or subordinate to that bridge or bonding interface are placed in a **down** state and become unreachable. To avoid losing connectivity, configure the node NIC in the same policy so that it has a status of **up** and either DHCP or a static IP address.

> **NOTE**
>
> Deleting the node network policy that added an interface does not change the configuration of the policy on the node. Although a **NodeNetworkConfigurationPolicy** is an object in the cluster, it only represents the requested configuration.
> Similarly, removing an interface does not delete the policy.

Procedure

1. Update the **NodeNetworkConfigurationPolicy** manifest used to create the interface. The following example removes a Linux bridge and configures the **eth1** NIC with DHCP to avoid losing connectivity:

```
apiVersion: nmstate.io/v1beta1
kind: NodeNetworkConfigurationPolicy
metadata:
  name: <br1-eth1-policy> 1
spec:
  nodeSelector: 2
    node-role.kubernetes.io/worker: "" 3
  desiredState:
    interfaces:
    - name: br1
      type: linux-bridge
      state: absent 4
    - name: eth1 5
      type: ethernet 6
      state: up 7
```

```
      ipv4:
        dhcp: true 8
        enabled: true 9
```

1. Name of the policy.
2. Optional: If you do not include the **nodeSelector** parameter, the policy applies to all nodes in the cluster.
3. This example uses the **node-role.kubernetes.io/worker: ""** node selector to select all worker nodes in the cluster.
4. Changing the state to **absent** removes the interface.
5. The name of the interface that is to be unattached from the bridge interface.
6. The type of interface. This example creates an Ethernet networking interface.
7. The requested state for the interface.
8. Optional: If you do not use **dhcp**, you can either set a static IP or leave the interface without an IP address.
9. Enables **ipv4** in this example.

2. Update the policy on the node and remove the interface:

```
$ oc apply -f <br1-eth1-policy.yaml> 1
```

1. File name of the policy manifest.

18.3.5. Example policy configurations for different interfaces

18.3.5.1. Example: Linux bridge interface node network configuration policy

Create a Linux bridge interface on nodes in the cluster by applying a **NodeNetworkConfigurationPolicy** manifest to the cluster.

The following YAML file is an example of a manifest for a Linux bridge interface. It includes samples values that you must replace with your own information.

```
apiVersion: nmstate.io/v1beta1
kind: NodeNetworkConfigurationPolicy
metadata:
  name: br1-eth1-policy 1
spec:
  nodeSelector: 2
    kubernetes.io/hostname: <node01> 3
  desiredState:
    interfaces:
      - name: br1 4
        description: Linux bridge with eth1 as a port 5
```

```
      type: linux-bridge 6
      state: up 7
      ipv4:
        dhcp: true 8
        enabled: true 9
      bridge:
        options:
          stp:
            enabled: false 10
        port:
          - name: eth1 11
```

1. Name of the policy.
2. Optional: If you do not include the **nodeSelector** parameter, the policy applies to all nodes in the cluster.
3. This example uses a **hostname** node selector.
4. Name of the interface.
5. Optional: Human-readable description of the interface.
6. The type of interface. This example creates a bridge.
7. The requested state for the interface after creation.
8. Optional: If you do not use **dhcp**, you can either set a static IP or leave the interface without an IP address.
9. Enables **ipv4** in this example.
10. Disables **stp** in this example.
11. The node NIC to which the bridge attaches.

18.3.5.2. Example: VLAN interface node network configuration policy

Create a VLAN interface on nodes in the cluster by applying a **NodeNetworkConfigurationPolicy** manifest to the cluster.

The following YAML file is an example of a manifest for a VLAN interface. It includes samples values that you must replace with your own information.

```
apiVersion: nmstate.io/v1beta1
kind: NodeNetworkConfigurationPolicy
metadata:
  name: vlan-eth1-policy 1
spec:
  nodeSelector: 2
    kubernetes.io/hostname: <node01> 3
  desiredState:
    interfaces:
    - name: eth1.102 4
```

```
    description: VLAN using eth1 5
    type: vlan 6
    state: up 7
    vlan:
      base-iface: eth1 8
      id: 102 9
```

1. Name of the policy.
2. Optional: If you do not include the **nodeSelector** parameter, the policy applies to all nodes in the cluster.
3. This example uses a **hostname** node selector.
4. Name of the interface.
5. Optional: Human-readable description of the interface.
6. The type of interface. This example creates a VLAN.
7. The requested state for the interface after creation.
8. The node NIC to which the VLAN is attached.
9. The VLAN tag.

18.3.5.3. Example: Bond interface node network configuration policy

Create a bond interface on nodes in the cluster by applying a **NodeNetworkConfigurationPolicy** manifest to the cluster.

> **NOTE**
>
> OpenShift Container Platform only supports the following bond modes:
>
> - mode=1 active-backup
> - mode=2 balance-xor
> - mode=4 802.3ad
> - mode=5 balance-tlb
> - mode=6 balance-alb

The following YAML file is an example of a manifest for a bond interface. It includes samples values that you must replace with your own information.

```
apiVersion: nmstate.io/v1beta1
kind: NodeNetworkConfigurationPolicy
metadata:
  name: bond0-eth1-eth2-policy 1
spec:
  nodeSelector: 2
```

```
    kubernetes.io/hostname: <node01> (3)
  desiredState:
    interfaces:
    - name: bond0 (4)
      description: Bond enslaving eth1 and eth2 (5)
      type: bond (6)
      state: up (7)
      ipv4:
        dhcp: true (8)
        enabled: true (9)
      link-aggregation:
        mode: active-backup (10)
        options:
          miimon: '140' (11)
        slaves: (12)
        - eth1
        - eth2
      mtu: 1450 (13)
```

1. Name of the policy.
2. Optional: If you do not include the **nodeSelector** parameter, the policy applies to all nodes in the cluster.
3. This example uses a **hostname** node selector.
4. Name of the interface.
5. Optional: Human-readable description of the interface.
6. The type of interface. This example creates a bond.
7. The requested state for the interface after creation.
8. Optional: If you do not use **dhcp**, you can either set a static IP or leave the interface without an IP address.
9. Enables **ipv4** in this example.
10. The driver mode for the bond. This example uses an active backup mode.
11. Optional: This example uses miimon to inspect the bond link every 140ms.
12. The subordinate node NICs in the bond.
13. Optional: The maximum transmission unit (MTU) for the bond. If not specified, this value is set to **1500** by default.

18.3.5.4. Example: Ethernet interface node network configuration policy

Configure an Ethernet interface on nodes in the cluster by applying a **NodeNetworkConfigurationPolicy** manifest to the cluster.

The following YAML file is an example of a manifest for an Ethernet interface. It includes sample values that you must replace with your own information.

```
apiVersion: nmstate.io/v1beta1
kind: NodeNetworkConfigurationPolicy
metadata:
  name: eth1-policy 1
spec:
  nodeSelector: 2
    kubernetes.io/hostname: <node01> 3
  desiredState:
    interfaces:
    - name: eth1 4
      description: Configuring eth1 on node01 5
      type: ethernet 6
      state: up 7
      ipv4:
        dhcp: true 8
        enabled: true 9
```

1. Name of the policy.
2. Optional: If you do not include the **nodeSelector** parameter, the policy applies to all nodes in the cluster.
3. This example uses a **hostname** node selector.
4. Name of the interface.
5. Optional: Human-readable description of the interface.
6. The type of interface. This example creates an Ethernet networking interface.
7. The requested state for the interface after creation.
8. Optional: If you do not use **dhcp**, you can either set a static IP or leave the interface without an IP address.
9. Enables **ipv4** in this example.

18.3.5.5. Example: Multiple interfaces in the same node network configuration policy

You can create multiple interfaces in the same node network configuration policy. These interfaces can reference each other, allowing you to build and deploy a network configuration by using a single policy manifest.

The following example snippet creates a bond that is named **bond10** across two NICs and a Linux bridge that is named **br1** that connects to the bond.

```
...
    interfaces:
    - name: bond10
      description: Bonding eth2 and eth3 for Linux bridge
      type: bond
```

```
      state: up
      link-aggregation:
        slaves:
        - eth2
        - eth3
    - name: br1
      description: Linux bridge on bond
      type: linux-bridge
      state: up
      bridge:
        port:
        - name: bond10
...
```

18.3.6. Examples: IP management

The following example configuration snippets demonstrate different methods of IP management.

These examples use the **ethernet** interface type to simplify the example while showing the related context in the policy configuration. These IP management examples can be used with the other interface types.

18.3.6.1. Static

The following snippet statically configures an IP address on the Ethernet interface:

```
...
    interfaces:
    - name: eth1
      description: static IP on eth1
      type: ethernet
      state: up
      ipv4:
        address:
        - ip: 192.168.122.250 1
          prefix-length: 24
        enabled: true
...
```

1 Replace this value with the static IP address for the interface.

18.3.6.2. No IP address

The following snippet ensures that the interface has no IP address:

```
...
    interfaces:
    - name: eth1
      description: No IP on eth1
      type: ethernet
      state: up
```

```
    ipv4:
      enabled: false
...
```

18.3.6.3. Dynamic host configuration

The following snippet configures an Ethernet interface that uses a dynamic IP address, gateway address, and DNS:

```
...
  interfaces:
  - name: eth1
    description: DHCP on eth1
    type: ethernet
    state: up
    ipv4:
      dhcp: true
      enabled: true
...
```

The following snippet configures an Ethernet interface that uses a dynamic IP address but does not use a dynamic gateway address or DNS:

```
...
  interfaces:
  - name: eth1
    description: DHCP without gateway or DNS on eth1
    type: ethernet
    state: up
    ipv4:
      dhcp: true
      auto-gateway: false
      auto-dns: false
      enabled: true
...
```

18.3.6.4. DNS

The following snippet sets DNS configuration on the host.

```
...
  interfaces:
     ...
  dns-resolver:
    config:
      search:
      - example.com
      - example.org
      server:
      - 8.8.8.8
...
```

18.3.6.5. Static routing

The following snippet configures a static route and a static IP on interface **eth1**.

```
...
  interfaces:
  - name: eth1
    description: Static routing on eth1
    type: ethernet
    state: up
    ipv4:
      address:
      - ip: 192.0.2.251 (1)
        prefix-length: 24
      enabled: true
  routes:
    config:
    - destination: 198.51.100.0/24
      metric: 150
      next-hop-address: 192.0.2.1 (2)
      next-hop-interface: eth1
      table-id: 254
...
```

1. The static IP address for the Ethernet interface.
2. Next hop address for the node traffic. This must be in the same subnet as the IP address set for the Ethernet interface.

18.4. TROUBLESHOOTING NODE NETWORK CONFIGURATION

If the node network configuration encounters an issue, the policy is automatically rolled back and the enactments report failure. This includes issues such as:

- The configuration fails to be applied on the host.
- The host loses connection to the default gateway.
- The host loses connection to the API server.

18.4.1. Troubleshooting an incorrect node network configuration policy configuration

You can apply changes to the node network configuration across your entire cluster by applying a node network configuration policy. If you apply an incorrect configuration, you can use the following example to troubleshoot and correct the failed node network policy.

In this example, a Linux bridge policy is applied to an example cluster that has three control plane nodes (master) and three compute (worker) nodes. The policy fails to be applied because it references an incorrect interface. To find the error, investigate the available NMState resources. You can then update the policy with the correct configuration.

Procedure

1. Create a policy and apply it to your cluster. The following example creates a simple bridge on the **ens01** interface:

```yaml
apiVersion: nmstate.io/v1beta1
kind: NodeNetworkConfigurationPolicy
metadata:
  name: ens01-bridge-testfail
spec:
  desiredState:
    interfaces:
      - name: br1
        description: Linux bridge with the wrong port
        type: linux-bridge
        state: up
        ipv4:
          dhcp: true
          enabled: true
        bridge:
          options:
            stp:
              enabled: false
          port:
            - name: ens01
```

```
$ oc apply -f ens01-bridge-testfail.yaml
```

Example output

```
nodenetworkconfigurationpolicy.nmstate.io/ens01-bridge-testfail created
```

2. Verify the status of the policy by running the following command:

```
$ oc get nncp
```

The output shows that the policy failed:

Example output

```
NAME                  STATUS
ens01-bridge-testfail   FailedToConfigure
```

However, the policy status alone does not indicate if it failed on all nodes or a subset of nodes.

3. List the node network configuration enactments to see if the policy was successful on any of the nodes. If the policy failed for only a subset of nodes, it suggests that the problem is with a specific node configuration. If the policy failed on all nodes, it suggests that the problem is with the policy.

```
$ oc get nnce
```

The output shows that the policy failed on all nodes:

Example output

```
NAME                                 STATUS
control-plane-1.ens01-bridge-testfail        FailedToConfigure
```

```
control-plane-2.ens01-bridge-testfail        FailedToConfigure
control-plane-3.ens01-bridge-testfail        FailedToConfigure
compute-1.ens01-bridge-testfail              FailedToConfigure
compute-2.ens01-bridge-testfail              FailedToConfigure
compute-3.ens01-bridge-testfail              FailedToConfigure
```

4. View one of the failed enactments and look at the traceback. The following command uses the output tool **jsonpath** to filter the output:

```
$ oc get nnce compute-1.ens01-bridge-testfail -o jsonpath='{.status.conditions[?
(@.type=="Failing")].message}'
```

This command returns a large traceback that has been edited for brevity:

Example output

```
error reconciling NodeNetworkConfigurationPolicy at desired state apply: , failed to execute
nmstatectl set --no-commit --timeout 480: 'exit status 1' ''
...
libnmstate.error.NmstateVerificationError:
desired
=======
---
name: br1
type: linux-bridge
state: up
bridge:
  options:
    group-forward-mask: 0
    mac-ageing-time: 300
    multicast-snooping: true
    stp:
      enabled: false
      forward-delay: 15
      hello-time: 2
      max-age: 20
      priority: 32768
  port:
  - name: ens01
description: Linux bridge with the wrong port
ipv4:
  address: []
  auto-dns: true
  auto-gateway: true
  auto-routes: true
  dhcp: true
  enabled: true
ipv6:
  enabled: false
mac-address: 01-23-45-67-89-AB
mtu: 1500

current
=======
---
```

```
name: br1
type: linux-bridge
state: up
bridge:
  options:
    group-forward-mask: 0
    mac-ageing-time: 300
    multicast-snooping: true
    stp:
      enabled: false
      forward-delay: 15
      hello-time: 2
      max-age: 20
      priority: 32768
  port: []
description: Linux bridge with the wrong port
ipv4:
  address: []
  auto-dns: true
  auto-gateway: true
  auto-routes: true
  dhcp: true
  enabled: true
ipv6:
  enabled: false
mac-address: 01-23-45-67-89-AB
mtu: 1500

difference
==========
--- desired
+++ current
@@ -13,8 +13,7 @@
       hello-time: 2
       max-age: 20
       priority: 32768
-  port:
-  - name: ens01
+  port: []
 description: Linux bridge with the wrong port
 ipv4:
   address: []
  line 651, in _assert_interfaces_equal\n
current_state.interfaces[ifname],\nlibnmstate.error.NmstateVerificationError:
```

The **NmstateVerificationError** lists the **desired** policy configuration, the **current** configuration of the policy on the node, and the **difference** highlighting the parameters that do not match. In this example, the **port** is included in the **difference**, which suggests that the problem is the port configuration in the policy.

5. To ensure that the policy is configured properly, view the network configuration for one or all of the nodes by requesting the **NodeNetworkState** object. The following command returns the network configuration for the **control-plane-1** node:

```
$ oc get nns control-plane-1 -o yaml
```

The output shows that the interface name on the nodes is **ens1** but the failed policy incorrectly uses **ens01**:

Example output

```
   - ipv4:
  ...
      name: ens1
      state: up
      type: ethernet
```

6. Correct the error by editing the existing policy:

```
$ oc edit nncp ens01-bridge-testfail
```

```
...
          port:
            - name: ens1
```

Save the policy to apply the correction.

7. Check the status of the policy to ensure it updated successfully:

```
$ oc get nncp
```

Example output

```
NAME                    STATUS
ens01-bridge-testfail   SuccessfullyConfigured
```

The updated policy is successfully configured on all nodes in the cluster.

CHAPTER 19. CONFIGURING THE CLUSTER-WIDE PROXY

Production environments can deny direct access to the internet and instead have an HTTP or HTTPS proxy available. You can configure OpenShift Container Platform to use a proxy by modifying the Proxy object for existing clusters or by configuring the proxy settings in the **install-config.yaml** file for new clusters.

IMPORTANT

The cluster-wide proxy is only supported if you used a user-provisioned infrastructure installation or provide your own networking, such as a virtual private cloud or virtual network, for a supported provider.

19.1. PREREQUISITES

- Review the sites that your cluster requires access to and determine whether any of them must bypass the proxy. By default, all cluster system egress traffic is proxied, including calls to the cloud provider API for the cloud that hosts your cluster. System-wide proxy affects system components only, not user workloads. Add sites to the Proxy object's **spec.noProxy** field to bypass the proxy if necessary.

NOTE

The Proxy object **status.noProxy** field is populated with the values of the **networking.machineNetwork[].cidr**, **networking.clusterNetwork[].cidr**, and **networking.serviceNetwork[]** fields from your installation configuration.

For installations on Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure, and Red Hat OpenStack Platform (RHOSP), the **Proxy** object **status.noProxy** field is also populated with the instance metadata endpoint (**169.254.169.254**).

19.2. ENABLING THE CLUSTER-WIDE PROXY

The Proxy object is used to manage the cluster-wide egress proxy. When a cluster is installed or upgraded without the proxy configured, a Proxy object is still generated but it will have a nil **spec**. For example:

```
apiVersion: config.openshift.io/v1
kind: Proxy
metadata:
  name: cluster
spec:
  trustedCA:
    name: ""
status:
```

A cluster administrator can configure the proxy for OpenShift Container Platform by modifying this **cluster** Proxy object.

> **NOTE**
>
> Only the Proxy object named **cluster** is supported, and no additional proxies can be created.

Prerequisites

- Cluster administrator permissions
- OpenShift Container Platform **oc** CLI tool installed

Procedure

1. Create a ConfigMap that contains any additional CA certificates required for proxying HTTPS connections.

   > **NOTE**
   >
   > You can skip this step if the proxy's identity certificate is signed by an authority from the RHCOS trust bundle.

   a. Create a file called **user-ca-bundle.yaml** with the following contents, and provide the values of your PEM-encoded certificates:

   ```
   apiVersion: v1
   data:
     ca-bundle.crt: | (1)
       <MY_PEM_ENCODED_CERTS> (2)
   kind: ConfigMap
   metadata:
     name: user-ca-bundle (3)
     namespace: openshift-config (4)
   ```

   (1) This data key must be named **ca-bundle.crt**.

   (2) One or more PEM-encoded X.509 certificates used to sign the proxy's identity certificate.

   (3) The ConfigMap name that will be referenced from the Proxy object.

   (4) The ConfigMap must be in the **openshift-config** namespace.

   b. Create the ConfigMap from this file:

   ```
   $ oc create -f user-ca-bundle.yaml
   ```

2. Use the **oc edit** command to modify the Proxy object:

   ```
   $ oc edit proxy/cluster
   ```

3. Configure the necessary fields for the proxy:

   ```
   apiVersion: config.openshift.io/v1
   ```

```
kind: Proxy
metadata:
  name: cluster
spec:
  httpProxy: http://<username>:<pswd>@<ip>:<port> 1
  httpsProxy: http://<username>:<pswd>@<ip>:<port> 2
  noProxy: example.com 3
  readinessEndpoints:
  - http://www.google.com 4
  - https://www.google.com
  trustedCA:
    name: user-ca-bundle 5
```

1. A proxy URL to use for creating HTTP connections outside the cluster. The URL scheme must be **http**.

2. A proxy URL to use for creating HTTPS connections outside the cluster. If this is not specified, then **httpProxy** is used for both HTTP and HTTPS connections.

3. A comma-separated list of destination domain names, domains, IP addresses or other network CIDRs to exclude proxying.

   Preface a domain with **.** to match subdomains only. For example, **.y.com** matches **x.y.com**, but not **y.com**. Use ***** to bypass proxy for all destinations. If you scale up workers that are not included in the network defined by the **networking.machineNetwork[].cidr** field from the installation configuration, you must add them to this list to prevent connection issues.

   This field is ignored if neither the **httpProxy** or **httpsProxy** fields are set.

4. One or more URLs external to the cluster to use to perform a readiness check before writing the **httpProxy** and **httpsProxy** values to status.

5. A reference to the ConfigMap in the **openshift-config** namespace that contains additional CA certificates required for proxying HTTPS connections. Note that the ConfigMap must already exist before referencing it here. This field is required unless the proxy's identity certificate is signed by an authority from the RHCOS trust bundle.

4. Save the file to apply the changes.

NOTE

The URL scheme must be **http**. The **https** scheme is currently not supported.

19.3. REMOVING THE CLUSTER-WIDE PROXY

The **cluster** Proxy object cannot be deleted. To remove the proxy from a cluster, remove all **spec** fields from the Proxy object.

Prerequisites

- Cluster administrator permissions
- OpenShift Container Platform **oc** CLI tool installed

Procedure

1. Use the **oc edit** command to modify the proxy:

```
$ oc edit proxy/cluster
```

2. Remove all **spec** fields from the Proxy object. For example:

```
apiVersion: config.openshift.io/v1
kind: Proxy
metadata:
  name: cluster
spec: {}
status: {}
```

3. Save the file to apply the changes.

CHAPTER 20. CONFIGURING A CUSTOM PKI

Some platform components, such as the web console, use Routes for communication and must trust other components' certificates to interact with them. If you are using a custom public key infrastructure (PKI), you must configure it so its privately signed CA certificates are recognized across the cluster.

You can leverage the Proxy API to add cluster-wide trusted CA certificates. You must do this either during installation or at runtime.

- During *installation*, configure the cluster-wide proxy. You must define your privately signed CA certificates in the **install-config.yaml** file's **additionalTrustBundle** setting.
  The installation program generates a ConfigMap that is named **user-ca-bundle** that contains the additional CA certificates you defined. The Cluster Network Operator then creates a **trusted-ca-bundle** ConfigMap that merges these CA certificates with the Red Hat Enterprise Linux CoreOS (RHCOS) trust bundle; this ConfigMap is referenced in the Proxy object's **trustedCA** field.

- At *runtime*, modify the default Proxy object to include your privately signed CA certificates (part of cluster's proxy enablement workflow). This involves creating a ConfigMap that contains the privately signed CA certificates that should be trusted by the cluster, and then modifying the proxy resource with the **trustedCA** referencing the privately signed certificates' ConfigMap.

NOTE

The installer configuration's **additionalTrustBundle** field and the proxy resource's **trustedCA** field are used to manage the cluster-wide trust bundle; **additionalTrustBundle** is used at install time and the proxy's **trustedCA** is used at runtime.

The **trustedCA** field is a reference to a **ConfigMap** containing the custom certificate and key pair used by the cluster component.

20.1. CONFIGURING THE CLUSTER-WIDE PROXY DURING INSTALLATION

Production environments can deny direct access to the internet and instead have an HTTP or HTTPS proxy available. You can configure a new OpenShift Container Platform cluster to use a proxy by configuring the proxy settings in the **install-config.yaml** file.

Prerequisites

- You have an existing **install-config.yaml** file.
- You reviewed the sites that your cluster requires access to and determined whether any of them need to bypass the proxy. By default, all cluster egress traffic is proxied, including calls to hosting cloud provider APIs. You added sites to the **Proxy** object's **spec.noProxy** field to bypass the proxy if necessary.

> **NOTE**
>
> The **Proxy** object **status.noProxy** field is populated with the values of the **networking.machineNetwork[].cidr**, **networking.clusterNetwork[].cidr**, and **networking.serviceNetwork[]** fields from your installation configuration.
>
> For installations on Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure, and Red Hat OpenStack Platform (RHOSP), the **Proxy** object **status.noProxy** field is also populated with the instance metadata endpoint (**169.254.169.254**).

- If your cluster is on AWS, you added the **ec2.<region>.amazonaws.com**, **elasticloadbalancing.<region>.amazonaws.com**, and **s3.<region>.amazonaws.com** endpoints to your VPC endpoint. These endpoints are required to complete requests from the nodes to the AWS EC2 API. Because the proxy works on the container level, not the node level, you must route these requests to the AWS EC2 API through the AWS private network. Adding the public IP address of the EC2 API to your allowlist in your proxy server is not sufficient.

Procedure

1. Edit your **install-config.yaml** file and add the proxy settings. For example:

```
apiVersion: v1
baseDomain: my.domain.com
proxy:
  httpProxy: http://<username>:<pswd>@<ip>:<port> 1
  httpsProxy: https://<username>:<pswd>@<ip>:<port> 2
  noProxy: example.com 3
additionalTrustBundle: | 4
    -----BEGIN CERTIFICATE-----
    <MY_TRUSTED_CA_CERT>
    -----END CERTIFICATE-----
...
```

1. A proxy URL to use for creating HTTP connections outside the cluster. The URL scheme must be **http**. If you use an MITM transparent proxy network that does not require additional proxy configuration but requires additional CAs, you must not specify an **httpProxy** value.

2. A proxy URL to use for creating HTTPS connections outside the cluster. If this field is not specified, then **httpProxy** is used for both HTTP and HTTPS connections. If you use an MITM transparent proxy network that does not require additional proxy configuration but requires additional CAs, you must not specify an **httpsProxy** value.

3. A comma-separated list of destination domain names, IP addresses, or other network CIDRs to exclude from proxying. Preface a domain with **.** to match subdomains only. For example, **.y.com** matches **x.y.com**, but not **y.com**. Use * to bypass the proxy for all destinations.

4. If provided, the installation program generates a config map that is named **user-ca-bundle** in the **openshift-config** namespace that contains one or more additional CA certificates that are required for proxying HTTPS connections. The Cluster Network Operator then creates a **trusted-ca-bundle** config map that merges these contents with the Red Hat Enterprise Linux CoreOS (RHCOS) trust bundle, and this config map is referenced in the **trustedCA** field of the **Proxy** object. The **additionalTrustBundle** field is required unless

the proxy's identity certificate is signed by an authority from the RHCOS trust bundle. If you use an MITM transparent proxy network that does not require additional proxy configuration but requires additional CAs, you must provide the MITM CA certificate.

> NOTE
>
> The installation program does not support the proxy **readinessEndpoints** field.

2. Save the file and reference it when installing OpenShift Container Platform.

The installation program creates a cluster-wide proxy that is named **cluster** that uses the proxy settings in the provided **install-config.yaml** file. If no proxy settings are provided, a **cluster Proxy** object is still created, but it will have a nil **spec**.

> NOTE
>
> Only the **Proxy** object named **cluster** is supported, and no additional proxies can be created.

20.2. ENABLING THE CLUSTER-WIDE PROXY

The Proxy object is used to manage the cluster-wide egress proxy. When a cluster is installed or upgraded without the proxy configured, a Proxy object is still generated but it will have a nil **spec**. For example:

```
apiVersion: config.openshift.io/v1
kind: Proxy
metadata:
  name: cluster
spec:
  trustedCA:
    name: ""
status:
```

A cluster administrator can configure the proxy for OpenShift Container Platform by modifying this **cluster** Proxy object.

> NOTE
>
> Only the Proxy object named **cluster** is supported, and no additional proxies can be created.

Prerequisites

- Cluster administrator permissions
- OpenShift Container Platform **oc** CLI tool installed

Procedure

1. Create a ConfigMap that contains any additional CA certificates required for proxying HTTPS connections.

> **NOTE**
>
> You can skip this step if the proxy's identity certificate is signed by an authority from the RHCOS trust bundle.

a. Create a file called **user-ca-bundle.yaml** with the following contents, and provide the values of your PEM-encoded certificates:

```
apiVersion: v1
data:
  ca-bundle.crt: | (1)
    <MY_PEM_ENCODED_CERTS> (2)
kind: ConfigMap
metadata:
  name: user-ca-bundle (3)
  namespace: openshift-config (4)
```

(1) This data key must be named **ca-bundle.crt**.

(2) One or more PEM-encoded X.509 certificates used to sign the proxy's identity certificate.

(3) The ConfigMap name that will be referenced from the Proxy object.

(4) The ConfigMap must be in the **openshift-config** namespace.

b. Create the ConfigMap from this file:

```
$ oc create -f user-ca-bundle.yaml
```

2. Use the **oc edit** command to modify the Proxy object:

```
$ oc edit proxy/cluster
```

3. Configure the necessary fields for the proxy:

```
apiVersion: config.openshift.io/v1
kind: Proxy
metadata:
  name: cluster
spec:
  httpProxy: http://<username>:<pswd>@<ip>:<port> (1)
  httpsProxy: http://<username>:<pswd>@<ip>:<port> (2)
  noProxy: example.com (3)
  readinessEndpoints:
  - http://www.google.com (4)
  - https://www.google.com
  trustedCA:
    name: user-ca-bundle (5)
```

(1) A proxy URL to use for creating HTTP connections outside the cluster. The URL scheme must be **http**.

2. A proxy URL to use for creating HTTPS connections outside the cluster. If this is not specified, then **httpProxy** is used for both HTTP and HTTPS connections.

3. A comma-separated list of destination domain names, domains, IP addresses or other network CIDRs to exclude proxying.

   Preface a domain with **.** to match subdomains only. For example, **.y.com** matches **x.y.com**, but not **y.com**. Use * to bypass proxy for all destinations. If you scale up workers that are not included in the network defined by the **networking.machineNetwork[].cidr** field from the installation configuration, you must add them to this list to prevent connection issues.

   This field is ignored if neither the **httpProxy** or **httpsProxy** fields are set.

4. One or more URLs external to the cluster to use to perform a readiness check before writing the **httpProxy** and **httpsProxy** values to status.

5. A reference to the ConfigMap in the **openshift-config** namespace that contains additional CA certificates required for proxying HTTPS connections. Note that the ConfigMap must already exist before referencing it here. This field is required unless the proxy's identity certificate is signed by an authority from the RHCOS trust bundle.

4. Save the file to apply the changes.

> **NOTE**
>
> The URL scheme must be **http**. The **https** scheme is currently not supported.

20.3. CERTIFICATE INJECTION USING OPERATORS

Once your custom CA certificate is added to the cluster via ConfigMap, the Cluster Network Operator merges the user-provided and system CA certificates into a single bundle and injects the merged bundle into the Operator requesting the trust bundle injection.

Operators request this injection by creating an empty ConfigMap with the following label:

```
config.openshift.io/inject-trusted-cabundle="true"
```

The Operator mounts this ConfigMap into the container's local trust store.

> **NOTE**
>
> Adding a trusted CA certificate is only needed if the certificate is not included in the Red Hat Enterprise Linux CoreOS (RHCOS) trust bundle.

Certificate injection is not limited to Operators. The Cluster Network Operator injects certificates across any namespace when an empty ConfigMap is created with the **config.openshift.io/inject-trusted-cabundle=true** label.

The ConfigMap can reside in any namespace, but the ConfigMap must be mounted as a volume to each container within a Pod that requires a custom CA. For example:

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: my-example-custom-ca-deployment
  namespace: my-example-custom-ca-ns
spec:
  ...
    spec:
      ...
      containers:
        - name: my-container-that-needs-custom-ca
          volumeMounts:
          - name: trusted-ca
            mountPath: /etc/pki/ca-trust/extracted/pem
            readOnly: true
      volumes:
      - name: trusted-ca
        configMap:
          name: trusted-ca
          items:
            - key: ca-bundle.crt (1)
              path: tls-ca-bundle.pem (2)
```

1. **ca-bundle.crt** is required as the ConfigMap key.
2. **tls-ca-bundle.pem** is required as the ConfigMap path.

CHAPTER 21. LOAD BALANCING ON RHOSP

21.1. USING THE OCTAVIA OVN LOAD BALANCER PROVIDER DRIVER WITH KURYR SDN

If your OpenShift Container Platform cluster uses Kuryr and was installed on a Red Hat OpenStack Platform (RHOSP) 13 cloud that was later upgraded to RHOSP 16, you can configure it to use the Octavia OVN provider driver.

IMPORTANT

Kuryr replaces existing load balancers after you change provider drivers. This process results in some downtime.

Prerequisites

- Install the RHOSP CLI, **openstack**.
- Install the OpenShift Container Platform CLI, **oc**.
- Verify that the Octavia OVN driver on RHOSP is enabled.

  TIP

  To view a list of available Octavia drivers, on a command line, enter **openstack loadbalancer provider list**.

  The **ovn** driver is displayed in the command's output.

Procedure

To change from the Octavia Amphora provider driver to Octavia OVN:

1. Open the **kuryr-config** ConfigMap. On a command line, enter:

```
$ oc -n openshift-kuryr edit cm kuryr-config
```

2. In the ConfigMap, delete the line that contains **kuryr-octavia-provider: default**. For example:

```
...
kind: ConfigMap
metadata:
  annotations:
    networkoperator.openshift.io/kuryr-octavia-provider: default (1)
...
```

(1) Delete this line. The cluster will regenerate it with **ovn** as the value.

Wait for the Cluster Network Operator to detect the modification and to redeploy the **kuryr-controller** and **kuryr-cni** pods. This process might take several minutes.

3. Verify that the **kuryr-config** ConfigMap annotation is present with **ovn** as its value. On a command line, enter:

```
$ oc -n openshift-kuryr edit cm kuryr-config
```

The **ovn** provider value is displayed in the output:

```
...
kind: ConfigMap
metadata:
  annotations:
    networkoperator.openshift.io/kuryr-octavia-provider: ovn
...
```

4. Verify that RHOSP recreated its load balancers.

   a. On a command line, enter:

   ```
   $ openstack loadbalancer list | grep amphora
   ```

   A single Amphora load balancer is displayed. For example:

   ```
   a4db683b-2b7b-4988-a582-c39daaad7981 | ostest-7mbj6-kuryr-api-loadbalancer  |
   84c99c906edd475ba19478a9a6690efd | 172.30.0.1      | ACTIVE              | amphora
   ```

   b. Search for **ovn** load balancers by entering:

   ```
   $ openstack loadbalancer list | grep ovn
   ```

   The remaining load balancers of the **ovn** type are displayed. For example:

   ```
   2dffe783-98ae-4048-98d0-32aa684664cc | openshift-apiserver-operator/metrics |
   84c99c906edd475ba19478a9a6690efd | 172.30.167.119 | ACTIVE              | ovn
   0b1b2193-251f-4243-af39-2f99b29d18c5 | openshift-etcd/etcd                  |
   84c99c906edd475ba19478a9a6690efd | 172.30.143.226 | ACTIVE              | ovn
   f05b07fc-01b7-4673-bd4d-adaa4391458e | openshift-dns-operator/metrics       |
   84c99c906edd475ba19478a9a6690efd | 172.30.152.27  | ACTIVE              | ovn
   ```

21.2. SCALING CLUSTERS FOR APPLICATION TRAFFIC BY USING OCTAVIA

OpenShift Container Platform clusters that run on Red Hat OpenStack Platform (RHOSP) can use the Octavia load balancing service to distribute traffic across multiple virtual machines (VMs) or floating IP addresses. This feature mitigates the bottleneck that single machines or addresses create.

If your cluster uses Kuryr, the Cluster Network Operator created an internal Octavia load balancer at deployment. You can use this load balancer for application network scaling.

If your cluster does not use Kuryr, you must create your own Octavia load balancer to use it for application network scaling.

21.2.1. Scaling clusters by using Octavia

If you want to use multiple API load balancers, or if your cluster does not use Kuryr, create an Octavia load balancer and then configure your cluster to use it.

Prerequisites

- Octavia is available on your Red Hat OpenStack Platform (RHOSP) deployment.

Procedure

1. From a command line, create an Octavia load balancer that uses the Amphora driver:

   ```
   $ openstack loadbalancer create --name API_OCP_CLUSTER --vip-subnet-id <id_of_worker_vms_subnet>
   ```

   You can use a name of your choice instead of **API_OCP_CLUSTER**.

2. After the load balancer becomes active, create listeners:

   ```
   $ openstack loadbalancer listener create --name API_OCP_CLUSTER_6443 --protocol HTTPS--protocol-port 6443 API_OCP_CLUSTER
   ```

   NOTE

   To view the status of the load balancer, enter **openstack loadbalancer list**.

3. Create a pool that uses the round robin algorithm and has session persistence enabled:

   ```
   $ openstack loadbalancer pool create --name API_OCP_CLUSTER_pool_6443 --lb-algorithm ROUND_ROBIN --session-persistence type=<source_IP_address> --listener API_OCP_CLUSTER_6443 --protocol HTTPS
   ```

4. To ensure that control plane machines are available, create a health monitor:

   ```
   $ openstack loadbalancer healthmonitor create --delay 5 --max-retries 4 --timeout 10 --type TCP API_OCP_CLUSTER_pool_6443
   ```

5. Add the control plane machines as members of the load balancer pool:

   ```
   $ for SERVER in $(MASTER-0-IP MASTER-1-IP MASTER-2-IP)
   do
     openstack loadbalancer member create --address $SERVER  --protocol-port 6443 API_OCP_CLUSTER_pool_6443
   done
   ```

6. Optional: To reuse the cluster API floating IP address, unset it:

   ```
   $ openstack floating ip unset $API_FIP
   ```

7. Add either the unset **API_FIP** or a new address to the created load balancer VIP:

   ```
   $ openstack floating ip set  --port $(openstack loadbalancer show -c <vip_port_id> -f value API_OCP_CLUSTER) $API_FIP
   ```

Your cluster now uses Octavia for load balancing.

> **NOTE**
>
> If Kuryr uses the Octavia Amphora driver, all traffic is routed through a single Amphora virtual machine (VM).
>
> You can repeat this procedure to create additional load balancers, which can alleviate the bottleneck.

21.2.2. Scaling clusters that use Kuryr by using Octavia

If your cluster uses Kuryr, associate the API floating IP address of your cluster with the pre-existing Octavia load balancer.

Prerequisites

- Your OpenShift Container Platform cluster uses Kuryr.
- Octavia is available on your Red Hat OpenStack Platform (RHOSP) deployment.

Procedure

1. Optional: From a command line, to reuse the cluster API floating IP address, unset it:

```
$ openstack floating ip unset $API_FIP
```

2. Add either the unset **API_FIP** or a new address to the created load balancer VIP:

```
$ openstack floating ip set --port $(openstack loadbalancer show -c <vip_port_id> -f value
${OCP_CLUSTER}-kuryr-api-loadbalancer) $API_FIP
```

Your cluster now uses Octavia for load balancing.

> **NOTE**
>
> If Kuryr uses the Octavia Amphora driver, all traffic is routed through a single Amphora virtual machine (VM).
>
> You can repeat this procedure to create additional load balancers, which can alleviate the bottleneck.

21.3. SCALING FOR INGRESS TRAFFIC BY USING RHOSP OCTAVIA

You can use Octavia load balancers to scale Ingress controllers on clusters that use Kuryr.

Prerequisites

- Your OpenShift Container Platform cluster uses Kuryr.
- Octavia is available on your RHOSP deployment.

Procedure

1. To copy the current internal router service, on a command line, enter:

```
$ oc -n openshift-ingress get svc router-internal-default -o yaml > external_router.yaml
```

2. In the file **external_router.yaml**, change the values of **metadata.name** and **spec.type** to **LoadBalancer**.

Example router file

```
apiVersion: v1
kind: Service
metadata:
  labels:
    ingresscontroller.operator.openshift.io/owning-ingresscontroller: default
  name: router-external-default 1
  namespace: openshift-ingress
spec:
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: http
  - name: https
    port: 443
    protocol: TCP
    targetPort: https
  - name: metrics
    port: 1936
    protocol: TCP
    targetPort: 1936
  selector:
    ingresscontroller.operator.openshift.io/deployment-ingresscontroller: default
  sessionAffinity: None
  type: LoadBalancer 2
```

1 Ensure that this value is descriptive, like **router-external-default**.

2 Ensure that this value is **LoadBalancer**.

NOTE

You can delete timestamps and other information that is irrelevant to load balancing.

1. From a command line, create a service from the **external_router.yaml** file:

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

2. Verify that the external IP address of the service is the same as the one that is associated with the load balancer:

   a. On a command line, retrieve the external IP address of the service:

   ```
   $ oc -n openshift-ingress get svc
   ```

Example output

```
NAME                    TYPE          CLUSTER-IP       EXTERNAL-IP    PORT(S)
AGE
router-external-default   LoadBalancer   172.30.235.33    10.46.22.161
80:30112/TCP,443:32359/TCP,1936:30317/TCP   3m38s
router-internal-default   ClusterIP      172.30.115.123   <none>
80/TCP,443/TCP,1936/TCP                      22h
```

b. Retrieve the IP address of the load balancer:

```
$ openstack loadbalancer list | grep router-external
```

Example output

```
| 21bf6afe-b498-4a16-a958-3229e83c002c | openshift-ingress/router-external-default |
66f3816acf1b431691b8d132cc9d793c | 172.30.235.33  | ACTIVE | octavia |
```

c. Verify that the addresses you retrieved in the previous steps are associated with each other in the floating IP list:

```
$ openstack floating ip list | grep 172.30.235.33
```

Example output

```
| e2f80e97-8266-4b69-8636-e58bacf1879e | 10.46.22.161 | 172.30.235.33 | 655e7122-
806a-4e0a-a104-220c6e17bda6 | a565e55a-99e7-4d15-b4df-f9d7ee8c9deb |
66f3816acf1b431691b8d132cc9d793c |
```

You can now use the value of **EXTERNAL-IP** as the new Ingress address.

> **NOTE**
>
> If Kuryr uses the Octavia Amphora driver, all traffic is routed through a single Amphora virtual machine (VM).
>
> You can repeat this procedure to create additional load balancers, which can alleviate the bottleneck.

21.4. CONFIGURING AN EXTERNAL LOAD BALANCER

You can configure an OpenShift Container Platform cluster on Red Hat OpenStack Platform (RHOSP) to use an external load balancer in place of the default load balancer.

Prerequisites

- On your load balancer, TCP over ports 6443, 443, and 80 must be available to any users of your system.
- Load balance the API port, 6443, between each of the control plane nodes.
- Load balance the application ports, 443 and 80, between all of the compute nodes.

- On your load balancer, port 22623, which is used to serve ignition start-up configurations to nodes, is not exposed outside of the cluster.
- Your load balancer must be able to access every machine in your cluster. Methods to allow this access include:
  - Attaching the load balancer to the cluster's machine subnet.
  - Attaching floating IP addresses to machines that use the load balancer.

IMPORTANT

External load balancing services and the control plane nodes must run on the same L2 network, and on the same VLAN when using VLANs to route traffic between the load balancing services and the control plane nodes.

Procedure

1. Enable access to the cluster from your load balancer on ports 6443, 443, and 80.
   As an example, note this HAProxy configuration:

   A section of a sample HAProxy configuration

```
...
listen my-cluster-api-6443
  bind 0.0.0.0:6443
  mode tcp
  balance roundrobin
  server my-cluster-master-2 192.0.2.2:6443 check
  server my-cluster-master-0 192.0.2.3:6443 check
  server my-cluster-master-1 192.0.2.1:6443 check
listenmy-cluster-apps-443
    bind 0.0.0.0:443
    mode tcp
    balance roundrobin
    server my-cluster-worker-0 192.0.2.6:443 check
    server my-cluster-worker-1 192.0.2.5:443 check
    server my-cluster-worker-2 192.0.2.4:443 check
listenmy-cluster-apps-80
    bind 0.0.0.0:80
    mode tcp
    balance roundrobin
    server my-cluster-worker-0 192.0.2.7:80 check
    server my-cluster-worker-1 192.0.2.9:80 check
    server my-cluster-worker-2 192.0.2.8:80 check
```

2. Add records to your DNS server for the cluster API and apps over the load balancer. For example:

```
<load_balancer_ip_address> api.<cluster_name>.<base_domain>
<load_balancer_ip_address> apps.<cluster_name>.<base_domain>
```

3. From a command line, use **curl** to verify that the external load balancer and DNS configuration are operational.

   a. Verify that the cluster API is accessible:

a. Verify that the cluster API is accessible.

```
$ curl https://<loadbalancer_ip_address>:6443/version --insecure
```

If the configuration is correct, you receive a JSON object in response:

```
{
  "major": "1",
  "minor": "11+",
  "gitVersion": "v1.11.0+ad103ed",
  "gitCommit": "ad103ed",
  "gitTreeState": "clean",
  "buildDate": "2019-01-09T06:44:10Z",
  "goVersion": "go1.10.3",
  "compiler": "gc",
  "platform": "linux/amd64"
}
```

b. Verify that cluster applications are accessible:

NOTE

You can also verify application accessibility by opening the OpenShift Container Platform console in a web browser.

```
$ curl http://console-openshift-console.apps.<cluster_name>.<base_domain> -I -L --
insecure
```

If the configuration is correct, you receive an HTTP response:

```
HTTP/1.1 302 Found
content-length: 0
location: https://console-openshift-console.apps.<cluster-name>.<base domain>/
cache-control: no-cacheHTTP/1.1 200 OK
referrer-policy: strict-origin-when-cross-origin
set-cookie: csrf-
token=39HoZgztDnzjJkq/JuLJMeoKNXlfiVv2YgZc09c3TBOBU4NI6kDXaJH1LdicNhN1UsQ
Wzon4Dor9GWGfopaTEQ==; Path=/; Secure
x-content-type-options: nosniff
x-dns-prefetch-control: off
x-frame-options: DENY
x-xss-protection: 1; mode=block
date: Tue, 17 Nov 2020 08:42:10 GMT
content-type: text/html; charset=utf-8
set-cookie:
1e2670d92730b515ce3a1bb65da45062=9b714eb87e93cf34853e87a92d6894be; path=/;
HttpOnly; Secure; SameSite=None
cache-control: private
```

CHAPTER 22. ASSOCIATING SECONDARY INTERFACES METRICS TO NETWORK ATTACHMENTS

22.1. ASSOCIATING SECONDARY INTERFACES METRICS TO NETWORK ATTACHMENTS

Secondary devices, or interfaces, are used for different purposes. It is important to have a way to classify them to be able to aggregate the metrics for secondary devices with the same classification.

Exposed metrics contain the interface but do not specify where the interface originates. This is workable when there are no additional interfaces, but if a secondary interface is added, it is difficult to make use of the metrics since it is hard to identify the interfaces using only the interface name as an identifier.

When adding secondary interfaces, their names depend on the order in which they are added, and different secondary interfaces might belong to different networks and can be used for different purposes.

With **pod_network_name_info** it is possible to extend the current metrics with the additional information that identifies the interface type. In this way, it is possible to aggregate the metrics and to add specific alarms to specific interface types.

The network type is generated using the name of the related **NetworkAttachementDefinition**, that in turn is used to differentiate different classes of secondary networks. For example, different interfaces belonging to different networks or using different CNIs use different network attachment definition names.

22.1.1. Network Metrics Daemon

The Network Metrics Daemon is a daemon component that collects and publishes network related metrics.

The kubelet is already publishing network related metrics you can observe. These metrics are:

- **container_network_receive_bytes_total**
- **container_network_receive_errors_total**
- **container_network_receive_packets_total**
- **container_network_receive_packets_dropped_total**
- **container_network_transmit_bytes_total**
- **container_network_transmit_errors_total**
- **container_network_transmit_packets_total**
- **container_network_transmit_packets_dropped_total**

The labels in these metrics contain, among others:

- Pod name
- Pod namespace

- Interface name (such as **eth0**)

These metrics work well until new interfaces are added to the pod, for example via Multus, as it is not clear what the interface names refer to.

The interface label refers to the interface name, but it is not clear what that interface is meant for. In case of many different interfaces, it would be impossible to understand what network the metrics you are monitoring refer to.

This is addressed by introducing the new **pod_network_name_info** described in the following section.

22.1.2. Metrics with network name

This daemonset publishes a **pod_network_name_info** gauge metric, with a fixed value of **0**:

```
pod_network_name_info{interface="net0",namespace="namespacename",network_name="nadname
space/firstNAD",pod="podname"} 0
```

The network name label is produced using the annotation added by Multus. It ia the concatenation of the namespace the network attachment definition belongs to, plus the name of the network attachment definition.

The new metric alone does not provide much value, but combined with the network related **container_network_*** metrics, it offers better support for monitoring secondary networks.

Using a **promql** query like the following ones, it is possible to get a new metric containing the value and the network name retrieved from the **k8s.v1.cni.cncf.io/networks-status** annotation:

```
(container_network_receive_bytes_total) + on(namespace,pod,interface) group_left(network_name) (
pod_network_name_info )
(container_network_receive_errors_total) + on(namespace,pod,interface) group_left(network_name) (
pod_network_name_info )
(container_network_receive_packets_total) + on(namespace,pod,interface)
group_left(network_name) ( pod_network_name_info )
(container_network_receive_packets_dropped_total) + on(namespace,pod,interface)
group_left(network_name) ( pod_network_name_info )
(container_network_transmit_bytes_total) + on(namespace,pod,interface) group_left(network_name)
( pod_network_name_info )
(container_network_transmit_errors_total) + on(namespace,pod,interface) group_left(network_name)
( pod_network_name_info )
(container_network_transmit_packets_total) + on(namespace,pod,interface)
group_left(network_name) ( pod_network_name_info )
(container_network_transmit_packets_dropped_total) + on(namespace,pod,interface)
group_left(network_name)
```