JUNIPER NETWORKS | Engineering Simplicity

Cloud Native Contrail Networking

Cloud-Native Contrail Networking Feature Guide

Published 2023-09-13

Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, California 94089
USA
408-745-2000
www.juniper.net

Juniper Networks, the Juniper Networks logo, Juniper, and Junos are registered trademarks of Juniper Networks, Inc. in the United States and other countries. All other trademarks, service marks, registered marks, or registered service marks are the property of their respective owners.

Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

Cloud Native Contrail Networking Cloud-Native Contrail Networking Feature Guide
Copyright © 2023 Juniper Networks, Inc. All rights reserved.

The information in this document is current as of the date on the title page.

YEAR 2000 NOTICE

Juniper Networks hardware and software products are Year 2000 compliant. Junos OS has no known time-related limitations through the year 2038. However, the NTP application is known to have some difficulty in the year 2036.

END USER LICENSE AGREEMENT

The Juniper Networks product that is the subject of this technical documentation consists of (or is intended for use with) Juniper Networks software. Use of such software is subject to the terms and conditions of the End User License Agreement ("EULA") posted at https://support.juniper.net/support/eula/. By downloading, installing or using such software, you agree to the terms and conditions of that EULA.

Table of Contents

About This Guide | viii

1 **Configure Kubernetes and Contrail**

Enable IP Fabric Forwarding and Fabric Source NAT | 2

Enable Pods with Multiple Network Interfaces | 7

Display Microservice Status | 14

Lens Install with CN2 Extension | 19

- Benefits | 20
- Download and Install Lens | 20
- Download and Install the CN2 Extension for Lens | 21
- Connect Your CN2 Cluster to Lens | 21
- Uninstall the CN2 Extension | 22

IPv4 and IPv6 Dual-Stack Networking | 23

Pod Scheduling | 24

2 **CN2 Apstra Integration**

Extend Virtual Networks to Apstra | 37

- Overview | 37
- Example: CN2 Kubernetes Deployment with SR-IOV Pods | 38
- Prerequisites | 40
  - Considerations | 41
- Installation Workflow | 42
- Install and Configure the CN2 Apstra Plug-In | 44
- Install the CN2 IPAM Plug-In | 48
- Intra-VN and Inter-VN Approaches | 49

Introduction to Configuring Intra-VN Communication | **53**

Configure Intra-VN Communication | **56**

Before You Begin | **56**

Configure Intra-VN Communication Between SR-IOV Pods | **57**

Configure Intra-VN Communication Between SR-IOV Pods and Non-SR-IOV Pods | **58**

Configure Intra-VN Communication Between SR-IOV Pods, Non-SR-IOV Pods, and BMS | **64**

Introduction to Configuring Inter-VN Communication | **64**

Configure Inter-VN Communication | **66**

Before You Begin | **66**

Configure Inter-VN Communication Between SR-IOV Pods | **67**

Configure Inter-VN Communication Between SR-IOV Pods and Non-SR-IOV Pods | **69**

Configure Inter-VN Communication Between SR-IOV Pods, Non-SRIOV Pods and BMS | **74**

3 **CN2 Security**

Kubernetes Network Policies | 77

Security Policies | 83

Namespace Security Policies | **83**

Encrypt Secret Data at Rest | 88

Configure Management and Control Plane Authentication with TLS Encryption | 88

Overview | **88**

Configure TLS Encryption for Contrail Control Plane and vRouter | **89**

4 **Advanced Virtual Networking**

Enable BGP as a Service | 96

Create an Isolated Namespace | 109

Namespace Overview | **109**

Example: Isolated Namespace Configuration | **110**

Isolated Namespace Objects | **113**

Create an Isolated Namespace | **114**

Optional Configuration: IP Fabric Forwarding and Fabric Source NAT | **116**

Enable IP Fabric Forwarding | 116

Enable Fabric Source NAT | 118

Configure Allowed Address Pairs | 120

Enable Packet-Based Forwarding on Virtual Interfaces | 122

Configure Reverse Path Forwarding on Virtual Interfaces | 125

vRouter Interface Health Check | 127

vRouter Interface Health Check Overview | 127

Create a Health-Check Object | 128

Health-Check Process | 133

Kubernetes Ingress Support | 134

Deploy VirtualNetworkRouter in Cloud-Native Contrail Networking | 138

Configure Inter-Virtual Network Routing Through Route Targets | 157

Configure IPAM for Pod Networking | 162

Enable VLAN Subinterface Support on Virtual Interfaces | 166

EVPN Networking Support | 173

Customize Virtual Networks for Pod Deployments, Services, and Namespaces | 177

Deploy Kubevirt DPDK Dataplane Support for VMs | 185

Pull Kubevirt Images and Deploy Kubevirt Using a Local Registry | 197

Static Routes | 200

VPC to CN2 Communication in AWS EKS | 211

Configure a Service Account to Assume an IAM role | 220

5 Configure DPDK

Deploy DPDK vRouter | 223

6 Configure Services

Configure ClusterIP Service by Assigning Endpoints | 232

ClusterIP Service without a Selector and Manually Assigned Endpoints | 232

Configure ClusterIP Service | 233

NodePort Service Support in Cloud-Native Contrail Networking | 236

Create a Load Balancer Service | 246

Load Balancer Service Overview | 246

Create a Load Balancer Service | 247

Dual-Stack Networking Support | 254

Configure Load Balancer Services Without Selectors | 254

FloatingIP/DNAT for IPv6 Addresses | 258

7

Analytics

Contrail Networking Analytics | 263

Contrail Networking Metric List | 269

Kubernetes Metric List | 283

Cluster Node Metric List | 321

Contrail Networking Alert List | 339

vRouter Session Analytics in Contrail Networking | 349

Centralized Logging | 357

Port-Based Mirroring | 360

Overview: Port-Based Mirroring | 360

Example: Configure Port-Based Mirroring | 361

Summary | 364

Configurable Categories of Metrics Collection and Reporting (Tech Preview) | 365

Overview: Configurable Categories of Metrics Collection and Reporting | 366

Install and Upgrade | 367

Manage MetricGroup with Kubectl Commands | 368

Manage Metric Groups with UI | 369

Juniper CN2 Technology Previews (Tech Previews) | 371

About This Guide

This guide provides an understanding of the features and tasks that you can configure for Juniper Cloud-Native Contrail® Networking™ (CN2) Release 23.1. This guide is appropriate for administrators and operators who need to know how to use CN2.

1

CHAPTER

Configure Kubernetes and Contrail

Enable IP Fabric Forwarding and Fabric Source NAT | 2

Enable Pods with Multiple Network Interfaces | 7

Display Microservice Status | 14

Lens Install with CN2 Extension | 19

IPv4 and IPv6 Dual-Stack Networking | 23

Pod Scheduling | 24

Enable IP Fabric Forwarding and Fabric Source NAT

IN THIS SECTION

- Overview: IP Fabric Forwarding | 2
- Overview: Fabric Source NAT | 3
- Example: Configure Fabric Source NAT | 3
- Example: Configure External Networks with IP Fabric Forwarding | 5

Cloud-Native Contrail® Networking™ (CN2) supports IP fabric forwarding and fabric source Network Address Translation (NAT) in Kubernetes-orchestrated environments using Juniper Networks' Cloud-Native Contrail® Networking™ Release 22.1 or later.

Cloud-Native Contrail Networking supports IP fabric forwarding and fabric source NAT. IP fabric forwarding provides clusters running in the overlay network with a path to access the underlay network through the external virtual network. Fabric source NAT enables a gateway device in a fabric to translate the source IP address of data plane node traffic exiting the fabric into a public-side IP address.

You can use IP fabric forwarding and fabric source NAT in cloud-networking environments to provide access to the underlay network. The underlay network access provided by IP fabric forwarding and fabric source NAT enables resources within pods to directly access the Internet or to pull external artifacts from the underlay network. This underlay network access is provided without adding significant network complexity like other underlay network options, such as complex BGP topologies or firewall setups.

Overview: IP Fabric Forwarding

Starting with Release 22.1, Cloud-Native Contrail Networking supports IP fabric forwarding.

You enable IP fabric forwarding within virtual networks that have access to the external network. These virtual networks require direct access to the underlay network.

A virtual network that has access to the external network is named the *default-externalnetwork* by default. You can create a customized user-defined external network name, if you choose. When you enable IP fabric forwarding, the path to the underlay network is directly available to clusters running in the overlay network through this external virtual network. This direct connection between the overlay network and the underlay network gives hosts in the overlay network access to the underlay network.

Because IP fabric forwarding enables a virtual network to span both the overlay network and the underlay network, data packets traversing the two networks are not encapsulated and de-encapsulated. Packet processing, therefore, is more efficient.

IP fabric forwarding is also extremely useful for load balancing network traffic. A LoadBalancer service automatically detects any external virtual network that has enabled IP fabric forwarding when load-balancing external network traffic.

Overview: Fabric Source NAT

Starting in Release 22.1, Cloud-Native Contrail Networking supports fabric source NAT. Fabric source NAT provides a method for traffic from a data plane node in a Kubernetes environment to directly access the Internet without traversing a separate NAT firewall. You can also use source NAT to pull external artifacts into pods when needed.

Traffic from data plane nodes destined for the Internet must traverse a gateway device. This gateway device is a member device in the fabric that also has at least one interface connected to the public network. When fabric source NAT is enabled, the gateway device translates the source IP address of the originating packet from the data plane node into its own public side IP address. This address translation allows traffic from the data plane node to access the Internet.

The IP address translation that source NAT performs also updates the source port in the packet. Multiple data plane nodes can reach the public network through a single gateway public IP address using fabric source NAT.

You need fabric source NAT to translate the IP addresses of traffic exiting the fabric to the Internet. You are not using NAT to translate incoming Internet traffic with this feature.

Example: Configure Fabric Source NAT

Fabric source NAT is disabled by default in user-created virtual networks.

You can enable fabric source NAT manually in any individual virtual network by setting the *fabricSNAT:* variable in the *VirtualNetwork* object to **true**. You can disable fabric source NAT by setting this value to **false**.

The following example shows a virtual network object that has enabled fabric source NAT. This example assumes that a subnet object named *virtual-network-subnet1* is configured in a separate YAML file.

```
apiVersion: core.contrail.juniper.net/v2
kind: VirtualNetwork
metadata:
  name: virtualnetwork1
  namespace: namespace1
  labels:
    vn: virtualnetwork1
  annotations:
    core.juniper.net/display-name: virtualnetwork1
    core.juniper.net/description:
       Virtual Network 1 is a collection of end points that can communicate with each other.
spec:
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v2
    kind: Subnet
    namespace: namespace1
    name: virtual-network-subnet1
  fabricSNAT: true
```

You can also configure your environment to enable fabric source NAT in any user-created virtual network when the virtual network is created. If you want to enable fabric source NAT in any user-created virtual network upon creation, set the *enableSNAT* variable in the *ApiServer* resource to **true** when initially deploying your environment.

You must set this configuration in the *ApiServer* resource during initial deployment. You cannot change this setting in your environment after you apply the deployment YAML file. If you want to change the fabric source NAT setting for an individual virtual network after initial deployment, you must change the configuration manually for that virtual network.

Following is a representative YAML file configuration:

```
    kind: ApiServer
    metadata:
      ...
    spec:
      enableSNAT: true
      common:
```

```
    containers:
    ...
```

Fabric source NAT is enabled in any user-created virtual network upon creation when the *enableSNAT* variable is **true**. You can disable fabric source NAT when user-created virtual networks are created by setting the *enableSNAT* variable to **false**. Fabric source NAT is disabled by default.

Fabric source NAT automatically selects the IP addresses for translation. You do not need to configure address pools for fabric source NAT in most Cloud-Native Contrail Networking use cases. Address pools are configurable, however, using the *portTranslationPools:* hierarchy within the *GlobalVrouterConfig* resource.

Example: Configure External Networks with IP Fabric Forwarding

IP fabric forwarding is disabled by default.

You can enable IP fabric forwarding in any virtual network by setting the *fabricForwarding:* variable to **true**.

The following example shows how to enable IP fabric forwarding in an external virtual network that accesses the Internet through an IPv4 gateway:

```
apiVersion: core.contrail.juniper.net/v2
kind: VirtualNetwork
metadata:
  namespace: contrail
  name: external-vn
  labels:
    service.contrail.juniper.net/externalNetworkSelector: default-external
  annotations:
    core.juniper.net/display-name: Sample Virtual Network
    core.juniper.net/description:
      VirtualNetwork is a collection of end points (interface or ip(s) or MAC(s))
      that can communicate with each other by default. It is a collection of
      subnets whose default gateways are connected by an implicit router.
spec:
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v2
    kind: Subnet
    namespace: contrail
```

```
  name: external-subnet
fabricForwarding: true
```

You can also enable IP fabric forwarding while creating the external virtual network that has a path to the Internet.

You configure a virtual network's path to an external network through the *Kubemanager* resource in environments using CN2.

You enable external access for a virtual network by connecting the virtual network to an IPv4 or IPv6 gateway IP subnet address. You enable IP fabric forwarding for the external traffic in the virtual network using the same *Kubemanager* resource.

NOTE: You must configure the external network subnets and this IP fabric forwarding setting during the initial Cloud-Native Contrail deployment. You cannot configure these parameters after the initial deployment YAML file is applied.

The following example shows a YAML file used to configure a *Kubemanager* resource that creates a virtual network with external network access. The virtual network in this example runs with IP fabric forwarding. You would have to commit this YAML file during initial deployment.

```
kind: Kubemanager
metadata:
  ...
spec:
  externalNetworkV4Subnet: # Fill V4 Subnet of an external network if any
  externalNetworkV6Subnet: # Fill V6 Subnet of an external network if any
  ipFabricFowardingExtSvc: true
  common:
    containers:
    ...
```

You specify the IPv4 subnet or the IPv6 subnet of the external network using the *externalNetworkV4Subnet* or *externalNetworkV6Subnet:* variable in this YAML file. The subnet address is a public-side IP address that is reachable from the Internet through the gateway device. When you configure a Kubemanager resource using this YAML file, a new virtual network to the specified external network is created. This virtual network is named *default-externalnetwork* in the default namespace for CN2.

IP fabric forwarding runs in the virtual network with external network access when the *ipFabricFowardingExtSvc* variable is **true**. You can disable IP fabric forwarding for the external subnet by setting the *ipFabricFowardingExtSvc* variable to **false**.

Enable Pods with Multiple Network Interfaces

IN THIS SECTION

- Multiple Network Interfaces in Cloud-Native Contrail Benefits | 7
- Multiple Network Interfaces in Cloud-Native Contrail Overview | 8
- Cloud-Native Contrail Integration with Multus Overview | 9
- Create a Network Attachment Definition Object | 9
- Configure a Pod to Use Multiple Interfaces | 12
- Disable the Network Attachment Definition Controller | 13

Cloud-Native Contrail® Networking™ (CN2) supports multiple network interfaces for a pod within Kubernetes. Multiple network interface support for a pod in Kubernetes provides a variety of environment-specific functionality, including the ability to segment traffic over multiple interfaces.

Cloud-Native Contrail Networking natively supports multiple network interfaces for a pod. You can also enable multiple network interfaces in Cloud-Native Contrail Networking using Multus. Multus is a container network interface (CNI) plugin for Kubernetes developed by the Kubernetes Network Plumbing Working Group. Cloud-Native Contrail can interoperate with Multus to provide support for multiple interfaces provided by multiple CNIs in a pod.

This document provides the steps to enable multiple interfaces for a pod in environments using CN2. It includes information about when and how to enable multiple networking interfaces. Multiple interface support for a pod was initially released in Contrail Networking Release 22.1.

Multiple Network Interfaces in Cloud-Native Contrail Benefits

Support for multiple network interfaces is useful or required in many cloud-networking environments. This list provides a few common examples:

- Pods routinely require a data interface to carry data traffic and a separate interface for management traffic.
- Virtualized network functions (VNFs) typically need three interfaces—a left, a right, and a management interface—to provide network functions. A VNF often can't provide its function with a single network interface.

- Cloud network topologies routinely need to support two or more network interfaces to isolate management networks from tenant networks.
- In customized or high-scale cloud-networking environments, you often must use a cloud-networking product that supports multiple network interfaces to meet a variety of environment-specific requirements.

A pod in a Kubernetes cluster using the default CNI has a single network interface for sending and receiving network traffic. You can use Cloud-Native Contrail Networking to provide multiple network interfaces. Cloud-Native Contrail Networking also supports Multus integration, allowing environments using Cloud-Native Contrail for networking to support multiple network interfaces using Multus.

Multiple Network Interfaces in Cloud-Native Contrail Overview

You can enable multiple network interfaces in Cloud-Native Contrail using Multus and without using Multus. Multus is a container network interface (CNI) plugin for Kubernetes that enables support for multiple network interfaces on a pod as well as multihoming between pods. Multus can simultaneously support interfaces from multiple delegate CNIs. This multiple delegate CNI support allows for the creation of cloud-networking environments that are interconnected using CNIs from different vendors, including CN2. Multus is often called a "meta-plugin" because of this multi-vendor support.

The following two paragraphs provide information on when to use the two methods of enabling multiple network interfaces.

You should enable multiple network interfaces using the native Cloud-Native Contrail Networking support for multiple network interfaces for the following reasons;

- You do not want the overhead of enabling and maintaining Multus in your environment.
- You are using Cloud-Native Contrail Networking as your only container networking interface (CNI).
- You do not want to create and maintain Network Attachment Definition (NAD) objects to support multiple network interfaces in your environment.

  You must create a NAD object to enable multiple network interfaces with Multus. You do not have to configure a NAD object to enable multiple network interfaces if you are not using Multus.

  Each NAD object notably creates a virtual network and a subnet that you have to monitor and maintain.

You should enable multiple network interfaces using Multus for the following reasons:

- You are using Cloud-Native Contrail in an environment that is already using Multus. Multus is especially common in environments using Openshift orchestration.

- You need the "meta-plugin" capabilities provided by Multus. You are using Cloud-Native Contrail in an environment where a pod is using multiple interfaces and the multiple interfaces are being managed by Cloud-Native Contrail and other CNIs.

- You are using Cloud-Native Contrail in an environment where it is integrated with Juniper Networks Apstra. You must enable Multus in order to enable Cloud-Native Contrail integration with Apstra.

  Cloud-Native Contrail integration with Apstra was introduced in Release 22.4. For more information regarding Cloud-Native Contrail integration with Apstra, see "Extend Virtual Networks to Apstra" on page 37.

- You need some of the other Multus features in your environment.

Cloud-Native Contrail Integration with Multus Overview

A Contrail vRouter is natively Multus-aware. No Cloud-Native Contrail Networking-specific configuration is required to enable Multus interoperability with Cloud-Native Contrail.

This list summarizes Cloud-Native Contrail support interoperability options with Multus:

- Cloud-Native Contrail is compatible with Multus CNI version **0.3.1**.

- Cloud-Native Contrail must be configured as the primary/default CNI with Multus.

- Cloud-Native Contrail can be configured as a delegate CNI with Multus only when Cloud-Native Contrail is also configured as the primary/default CNI. Cloud-Native Contrail is not supported as a delegate CNI when other CNIs are configured as the primary CNI.

- Cloud-Native Contrail supports interoperability with Multus when in vRouter kernel mode or Data Plane Development Kit (DPDK) mode.

Multus is a third-party plugin. You enable and configure Multus within Kubernetes but entirely outside of Cloud-Native Contrail. To enable Multus, you can apply the multus-daemonset.yml files provided by the Kubernetes Network Plumbing Working Group.

For detailed information about Multus, see the Multus CNI Usage Guide from the Kubernetes Network Plumbing Working Group.

Create a Network Attachment Definition Object

You do not need to create a *NetworkAttachmentDefinition* (NAD) object to enable multiple interfaces using the native multiple interfaces support in Cloud-Native Contrail Networking. You can skip this

section if you are not using Multus to enable multiple network interfaces in your environment. If you are not using NAD objects but need to create a virtual network, see "Deploy VirtualNetworkRouter in Cloud-Native Contrail Networking" on page 138.

This section illustrates how to create a NAD object using a YAML file. You configure Cloud-Native Contrail into the NAD object using the *juniper.net/networks* annotation. We provide a representative example of the YAML file that creates the NAD object and a field descriptions table later in this section.

Be sure to include the *juniper.net/networks* annotation when you create the *NetworkAttachmentDefinition* object. If you define the YAML file to create the *NetworkAttachmentDefinition* object without using the *juniper.net/networks* annotation, the *NetworkAttachmentDefinition* object is treated as a third-party object. No Contrail-related objects will be created in the network, including the *VirtualNetwork* object and the *Subnet* object.

You create the *NetworkAttachmentDefinition* object in a Kubernetes environment using the NAD controller. The NAD controller runs in kube-manager and either creates a *VirtualNetwork* object or updates an existing *VirtualNetwork* object when a *NetworkAttachmentDefinition* is successfully created. The NAD controller is enabled by default but you can disable it; see "Disable the Network Attachment Definition Controller" on page 13.

Following is an example of the YAML file used to create a *NetworkAttachmentDefinition* object:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: networkname-1
  namespace: nm1
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "172.16.10.0/24",
      "ipamV6Subnet": "2001:db8::/64",
      "routeTargetList": ["target:23:4561"],
      "importRouteTargetList": ["target:10.2.2.2:561"],
      "exportRouteTargetList": ["target:10.1.1.1:561"],
      "fabricSNAT": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "juniper-network",
  "type": "contrail-k8s-cni"
}'
```

The *NetworkAttachmentDefinition Object Fields* table provides usage details for the variables in the *NetworkAttachmentDefinition* object file.

Table 1: NetworkAttachmentDefinition Object Fields

Variable	Usage
ipamV4Subnet	(Optional) Specifies the IPv4 subnet address for the virtual network.
ipamV6Subnet	(Optional) Specifies the IPv6 subnet address for the virtual network.
routeTargetList	(Optional) Provides a list of route targets that are used as both import and export routes.
importRouteTargetList	(Optional) Provides a list of route targets that are used as import routes.
exportRouteTargetList	(Optional) Provides a list of route targets that are used as export routes.
fabricSNAT	(Optional) Specifies if you'd like to toggle connectivity to the underlay network using the port-mapping capabilities provided by fabric source NAT. Set this parameter to **true** or **false**. It is set to **false** by default. If you want to allow connectivity to the underlay network, set the parameter to **true**.

You should note the following network activities related to the *NetworkAttachmentDefinition* object:

- The network attachment definition controller works in kube-manager and handles processing of all network attachment definition objects.
- You can monitor network attachment definition controller updates in *juniper.net/network-status*.
- IPAM updates are not allowed to the network attachment definition object.

The network attachment definition object creates a virtual network. The Network Attachment Definition Object Impact on Virtual Networks table on page 12 provides an overview of how events related to the network attachment definition object impact virtual networks.

Table 2: Network Attachment Definition Object Impact on Virtual Networks

If	Then
You define a namespace for a network attachment definition object in a single cluster topology	A VirtualNetwork is created in the same namespace as the network attachment definition. This VirtualNetwork will have the same name as the Network Attachment Definition object. The NAD object is named using the **name:** field in the **metadata:** hierarchy.
You define a namespace for a network attachment definition object in a multi-cluster topology	The *VirtualNetwork* namespace is *cluster-name*-**ns**..
A namespace is not defined for a network attachment definition object in a multi-cluster topology	The *VirtualNetwork* namespace is *cluster-name*-**default**.
You delete a network attachment definition resource	The associated *VirtualNetwork* object is also deleted.
You delete a virtual network that was created by the network attachment definition resource	The network attachment definition controller reconciles the issue and recreates the virtual network.

Configure a Pod to Use Multiple Interfaces

You configure multiple interfaces in the pod object. If you are using Multus, you must also configure the NAD object as outlined in "Create a Network Attachment Definition Object" on page 9.

In the following example, you create two interfaces for network traffic in the *juniper-pod-1* pod: *tap1* and *tap2*.

```
apiVersion: v1
kind: Pod
metadata:
        name: juniper-pod-1
        namespace: juniper-ns
        annotations:
                k8s.v1.cni.cncf.io/networks: |-
                        [
```

```
                    {
                            "name":"juniper-network1",
                            "namespace":"juniper-ns",
                            "cni-args":null,
                            "ips":["172.16.20.42"],
                            "mac":"de:ad:00:00:be:ef",
                            "interface":"tap1"
                    },
            [
                    {
                            "name":"juniper-network2",
                            "namespace":"juniper-ns",
                            "cni-args":null,
                            "ips":["172.16.21.42"],
                            "mac":"de:ad:00:00:be:ee",
                            "interface":"tap2"
                    }
            ]
```


Disable the Network Attachment Definition Controller

The NAD controller is part of the kube-manager object. You enable and disable this controller using the **enableNad:** variable within the YAML file that defines the **kubemanager** object. The network attachment definition controller is enabled by default.

If you would prefer to prevent the application of *NetworkAttachmentDefinion* objects, you can disable the network attachment definition controller.

In the following example, the network attachment definition controller is disabled:

```
kind: Kubemanager
metadata:
  name: remote-cluster
  namespace: contrail
spec:
  common:
    nodeSelector:
      node-role.kubernetes.io/master: ""
  enableNad: false
```

Display Microservice Status

IN THIS SECTION

- Overview: Microservice Status in Cloud-Native Contrail Networking | **14**
- Display Microservice Status | **15**
- Display Deployment Status | **15**
- Display Resource Status | **16**

Juniper® Cloud-Native Contrail Networking (CN2) supports microservices in environments using CN2 Release 22.1 or later in a Kubernetes-orchestrated environment.

To display microservice status in CN2 cluster, you need:

- A CLI tool, such as `kubectl` to provide the overall system status of all the services running.
- The `contrailstatus` plugin must be installed along with `kubectl`.
- The use of command `kubectl contrailstatus` to request the status of various services.

Overview: Microservice Status in Cloud-Native Contrail Networking

Microservices exist as small, independent applications deployed without updating the entire Contrail Networking deployment. Microserves provide better ways to manage the life cycle of containers. The containers and their processes are grouped as services and microservices.

ContrailStatus is a `kubectl` plugin used to display the status information of Contrail Networking services in the three different planes (configuration, control, and data). In addition to the usual containers in a specific service, you can also view:

- `init` (initialization) container status within the service.
- The relative software status, such as BGP and XMPP, in `control_controller`.

The `contrailstatus` plug-in is categorized into two sections:

- Deployment status

- Resource status

Display Microservice Status

The following outputs are examples showing deployment status updates and resource status updates to the pods for all planes.

Display Deployment Status

Display deployment status in either short form or default form.

All Planes Deployment Status

To display the deployment status for all of the planes and request the short form:

```
root@helper ~] # kubectl contrailstatus -short


PLANE               STATUS
config              nok
control             ok
data                ok
```

The option `-short` for short form only displays output for the pod name and status. The following example outputs are using the default form.

Configuration Plane Deployment Status

To display the deployment status to the configuration plane:

```
root@helper ~] # kubectl contrailstatus deployment -p config


PODNAME                        STATUS            NODE        IP                 MESSAGE
apiserver-86885bf7d8-q27qk      nok              node        10.1.1.1          process not up,
init cont…..
apiserver-86885bf7d8-sdsdd      ok               node2       10.1.1.2
apiserver-86885bf7d8-sdsss      ok               node3       10.1.1.3
controller-6998bd846f-5cgf7     ok               node1       10.1.1.1
```

```
controller-6998bd846f-5cgf8    ok              node2       10.1.1.2
controller-6998bd846f-5cg10    nok             node3       10.1.1.3         o/1 node is not
allocated.
cluster1-kubemanager-7cff895-sdfsd ok          node2       10.1.1.2
cluster1-kubemanager-7cff895-sdfsa ok          node3       10.1.1.3
```

Data Plane Deployment Status

To display the deployment status to the data plane:

```
root@helper ~] # kubectl contrailstatus deployment -p data


PODNAME                              STATUS            NODE        IP                MESSAGE
vrouter-86885bf7d8-q27qk             nok               node        10.1.1.1           process
not up, init cont.…..
vrouter-86885bf7d8-sdsdd             ok                node2       10.1.1.2
```

Control Plane Deployment Status

To display the deployment status to the control plane:

```
root@helper ~] # kubectl contrailstatus deployment -p control


PODNAME                           STATUS            NODE        IP                MESSAGE
contrail-control-0                nok               node        10.1.1.1           process not
up, init cont.…..
contrail-control-1                ok                node2       10.1.1.2
```

Display Resource Status

The `contrailstatus` plugin also displays status updates for deployment resources, such as XMPP and BGP.

Data Plane Resource Status

To display the resource status of `bgprouter` to the data plane:

```
root@helper ~] kubectl contrailstatus resource bgprouter


PODNAME     STATUS      SERVICE
bgprouter1  nok         xmpp, bgp not working/has error..
bgprouter2  nok
bgprouter2 ok
```

Control Node Resource Status

To display the resource status in the control node, run the following command, which displays the output for the XMPP session:

```
root@helper ~] kubectl contrailstatus resource bgprouter -s xmpp


LOCAL           NEIGHBOR          STATE             POD
bgprouter1      vr1               established (ok)  contrail-control-0
bgprouter1      vr2               active (nok)      contrail-control-0
bgprouter2      vr1                                 contrail-control-1
bgprouter2      vr3                                 contrail-control-1
```

To display the resource status in the control node, run the following command, which gives the output for the BGP session:

```
root@helper ~]  kubectl contrailstatus resource bgprouter -s bgp


LOCAL         NEIGHBOR          STATE             POD
bgprouter1    bgprouter2        established (ok)  contrail-control-0
bgprouter1    bgprouter3        active (nok)      contrail-control-0
bgprouter2    bgprouter1        established (ok)  contrail-control-1
bgprouter2    bgprouter3        established (ok)  contrail-control-1
```

All Planes Resource Status

To display the resource status on all of the planes:

```
[root@helper ~] # kubectl contrailstatus -all


NAME                        STATUS    PLANE     ERRORNOTES
apiserver-86789f7d8-q37qf   Active    Config


NAME                        STATUS    PLANE     ERRORNOTES
control-1                   Active    control
BGP-1                       Active    control
XMPP-1                      Active    control


NAME                        STATUS    PLANE     ERRORNOTES
vrouter-86789f7d8-q37qk     Active    data


[root@helper ~] #
```

Services Status for Multiple Nodes

The following (same) command displays the status of various services running on multiple nodes in a cluster. If the running controller is active without any errors, the status column next to the service is displayed as `Active`. If the controller has any errors, the status column of the controller is displayed as `Not-Active`. The output includes the status of various controllers and containers in the controllers.

To display the status of various services running on multiple nodes in a cluster:

```
[root@helper ~] # kubectl contrailstatus -all

NAME                                 STATUS     ERRORNOTES
apiserver-86885bf7d8-q27qk           Active
apiserver-86885bf7d8-sdsdd           Active
apiserver-86885bf7d8-sdsss           Active
controller-6998bd846f-5cgf7          Active
controller-6998bd846f-5cgf8          Active
controller-6998bd846f-5cg10          Active
cluster1-kubemanager-7cff895-sdfsd   Active
cluster1-kubemanager-7cff895-sdfsa   Active
```

```
NAME           STATUS   ERRORNOTES
control-1      Active
control-2      Active
control-3      Active
BGP-1          Active
BGP-2          Active
XMPP-1         Active
Xmpp-2         Active


NAME                        STATUS   ERRORNOTES
vrouter-86789f7d8-q37qk     Active
vrouter-8905bf7d8-q47qk     Active
vrouter-8688bf7d8-q57qk     Active


[root@helper ~] #
```

RELATED DOCUMENTATION

- Kubectl Contrailstatus for Upstream Kubernetes
- Kubectl Contrailstatus for OpenShift Container Platform
- Kubectl Contrailstatus for Amazon EKS

Lens Install with CN2 Extension

SUMMARY

This document describes the installation procedure for both Lens and the CN2 extension for Lens, as well as how to connect a Juniper® Cloud-Native Contrail Networking (CN2) cluster to Lens.

IN THIS SECTION

- Benefits | 20
- Download and Install Lens | 20
- Download and Install the CN2 Extension for Lens | 21

- Connect Your CN2 Cluster to Lens | 21
- Uninstall the CN2 Extension | 22

Benefits

Lens is an integrated development environment (IDE) for Kubernetes. The Lens open source tool is implemented as an interface to manage, monitor, and troubleshoot CN2 clusters.

Benefits include:

- Ease of usability and rich end-user experience.
- Unified, secure, multi-cluster management on any platform: support for hundreds of clusters.
- Standalone application: no need to install anything in-cluster.
- Real-time cluster state visualization.
- Resource utilization charts and trends with history powered by built-in Prometheus.
- Smart terminal access to nodes and containers.
- Clusters can be local (for example, minikube) or external (for example, EKS, GKE, or AKS).

Download and Install Lens

To download and install Lens:

1. From your browser, navigate to Lens, and select your OS from the drop-down list to download Lens.

   NOTE: Lens v5.4.4 is the supported and tested version with CN2 extension.

   The download file looks similar to `Lens Setup 5.4.4-latest.20220602.2`.
2. Double-click the file you just downloaded from the Lens website.
3. Follow the setup wizard onscreen prompts.
4. Click **Finish** to complete the installation.

Lens is installed and listed as **Lens** on your machine.

Download and Install the CN2 Extension for Lens

The CN2 extension is a Lens Custom Extension.

- Use Lens extensions to add custom visualizations and functionality to accelerate development workflows for all the technologies and services that integrate with Kubernetes.
- Extensions are a plug-in that you can upload directly to the Lens UI.
- Extensions are developed using the Lens Extensions API.

To download and install the CN2 extension for Lens:

1. From your browser, download the CN2 extension for Lens from Juniper Networks Software Downloads.

   The file name is similar to `cn2_lens_extension-VERSION_TAG.tar`.
2. Click **Launch Lens** to start Lens. You are in the Lens application for the remainder of this section.
3. Sign in with your Lens ID username and password. Follow the prompts to create a Lens ID if you do not have one.
4. From the top menu bar, select **File** > **Extensions**.
5. In the Extensions window, click the folder icon to select the CN2 extension TAR file you downloaded in Step 1. Then, click **Install**.

   Lens begins installing the CN2 extension. This process takes approximately 10 minutes. You then see a message indicating a successful installation. The installed extension status **Enabled** appears on your screen. If Status is Disabled, right-click the "**:**" symbol and select **Enable** to enable the extension.

Connect Your CN2 Cluster to Lens

To connect your CN2 cluster to Lens:

1. Click **Launch Lens** in the Lens application to start Lens. You are working in the Lens application for the remainder of this procedure.
2. Select **File** > **Add Cluster**.
3. In the Add Clusters from Kubeconfig window, copy and paste the contents of your `kubeconfig` file or your YAML file or files for the clusters you want to manage. Click **Add clusters**.

You are now in the Clusters window. Lens automatically populates the Clusters window with all of the valid clusters Lens finds in your folder. You can have multiple CN2 cluster configurations in your folder (multiple clusters).

4. Click the cluster you want to connect to, and Lens automatically connects to that cluster.

   You are now connected to the CN2 cluster.

5. In the left pane, where the Lens options are listed, click **CN2** to interact with your cluster.

   If you don't see the **CN2** menu, select **Lens** > **View** > **Reload**.

   CN2 is your extension for Lens. You can select one of the following: **CN2** > **Infrastructure**, **Networking**, **Workloads**, or **Monitoring**.

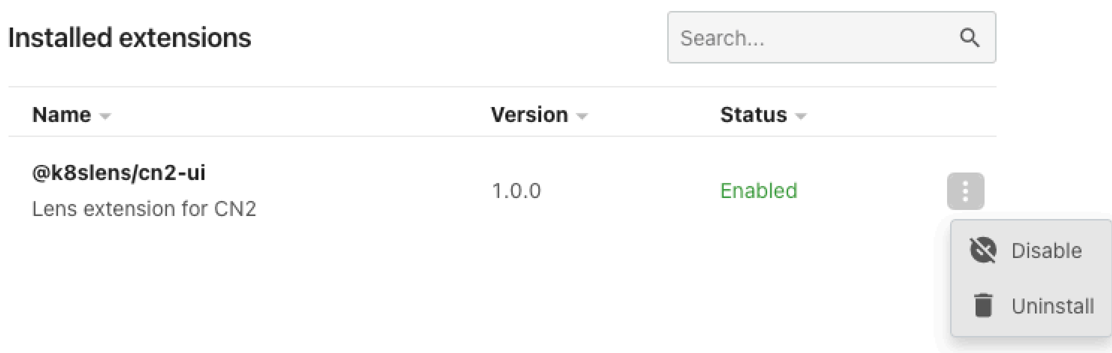
   > **NOTE**: Known Limitation: Lens supports two theme modes, which are Dark and Light. CN2 extension supports Light mode only.

Uninstall the CN2 Extension

To uninstall the CN2 extension:

1. Select **Lens** > **Extensions** in the Lens application.
2. In Installed extensions, click the " **:** " and select **Uninstall**.

Figure 1: Uninstall Lens



IPv4 and IPv6 Dual-Stack Networking

SUMMARY

Cloud-Native Contrail® Networking™ (CN2) release 23.1 supports dual-stack networking for services. Previous releases supported dual-stack networking for pods, but 23.1 enables you to assign IPs to services from an IPv4 or IPv6 network. This article provides an overview of dual-stack and information about configuring dual-stack for pods and services in your CN2 cluster.

IN THIS SECTION

- IPv4 and IPv6 Overview | 23
- Dual-Stack Networking Prerequisites | 23

IPv4 and IPv6 Overview

A dual-stack device has network interfaces that send and receive both IPv4 and IPv6 packets. In the case of CN2 release 23.1, the dual-stack feature of your Kubernetes cluster assigns both IPv4 addresses and IPv6 addresses to pods and services.

Dual-Stack Networking Prerequisites

Dual-stack networking requires the following:

- Kubernetes version 1.20 or later
- Kubernetes nodes configured with dual stack IPv4/IPv6 network interfaces
- A Kubeadm or Kubespray Kubernetes cluster with dual-stack `featureGate` enabled

The CN2 deployer uses the IPv6 CIDR (`podSubnet` and `serviceSubnet` in the deployment) to create an IPv6 subnet for the podNetwork. Subsequent pod networks that you create contain an IPv6 subnet. As a result, pods receive IPv4 and IPv6 addresses.

RELATED DOCUMENTATION

IPv4/IPv6 dual-stack

Dual-stack support with kubeadm

Deploy a Production-Ready Kubernetes Cluster

Pod Scheduling

SUMMARY

Juniper Cloud-Native Contrail Networking (CN2) release 23.1 supports network-aware pod scheduling using `contrail-scheduler`. This feature enhances the Kubernetes pod scheduler with plugins that analyze the network metrics of a node before scheduling pods. This article provides overview, implementation, and deployment information about network-aware pod scheduling.

IN THIS SECTION

- Pod Scheduling in Kubernetes | 24
- Pod Scheduling in CN2 | 25
- Network-Aware Pod Scheduling Overview | 25
- Network-Aware Pod Scheduling Components | 25
- Deploy Network-Aware Pod Scheduling Components | 26
- Metrics Collector Deployment | 26
- Central Collector Deployment | 27
- Contrail Scheduler Deployment | 30
- Use the Contrail Scheduler to Deploy Pods | 35

Pod Scheduling in Kubernetes

In Kubernetes, scheduling refers to the process of matching pods to nodes so that the kubelet is able to run them. A scheduler monitors requests for pod creation and attempts to assign these pods to suitable nodes using a series of extension points during a scheduling and binding cycle. Potential nodes are filtered based on attributes like the resource requirements of a pod. If a node doesn't have the available resources for a pod, that node is filtered out. If more than one node passes the filtering phase, Kubernetes scores and ranks the remaining nodes based on their suitability for a given pod. The scheduler assigns a pod to the node with the highest ranking. If two nodes have the same score, the scheduler picks a node at random.

Pod Scheduling in CN2

CN2 release 22.4 enhanced the default Kubernetes pod scheduler to schedule pods based on the Virtual Machine Interface (VMI) considerations of DPDK nodes. This enhanced scheduler, called `contrail-scheduler`, supports custom plugins that enable the scheduling of pods based on current active VMIs in a DPDK node.

CN2 release 23.1 improves on this feature by supporting two additional plugins. As a result of these plugins, `contrail-scheduler` schedules pods based on the following network metrics:

- Number of active ingress/egress traffic flows
- Bandwidth utilization
- Number of virtual machine interfaces (VMIs)

Network-Aware Pod Scheduling Overview

Many high-performance applications have bandwidth or network interface requirements as well as the typical CPU or VMI requirements. If `contrail-scheduler` assigns a pod to a node with low bandwidth availability, that application cannot run optimally. CN2 release 23.1 addresses this issue with the introduction of a metrics collector, a central collector, and custom scheduler plugins. These components collect, store, and process network metrics so that the `contrail-scheduler` schedules pods based on these metrics.

Network-Aware Pod Scheduling Components

The following main components comprise CN2's network-aware pod scheduling solution:

- Metrics collector: This runs in a container alongside the vRouter pod that runs on each node in the cluster. . The vRouter agent sends metrics data to the metrics collector over `localhost: 6700` specified in the `agent: default: collectors` field of the vRouter CR `Deployment`. The metrics collector then forwards requested data to configured sinks which are specified in the configuration. The central collector is one of the configured sinks and recieves this data from the metrics collector.
- Central collector: This component acts as an aggregator and stores data received from all of the nodes in a cluster via the metrics collector. The central collector exposes gRPC endpoints which consumers use to to request this data for nodes in a cluster. For example, the `contrail-scheduler` uses these gRPC endpoints to retrieve and process network metrics and schedule pods accordingly.

- Contrail scheduler: This custom scheduler introduces the following three custom plugins:
  - `VMICapacity` plugin (available from release 22.4 onwards): Implements Filter, Score, and `NormalizeScore` extension points in the scheduler framework. The `contrail-scheduler` uses these extension points to determine the best node to assign a pod to based on active VMIs.
  - `FlowsCapacity` plugin: Determines the best node to schedule a pod based on the number of active flows in a node. Too many traffic flows on a node means more competition for new pod traffic. Pods and nodes with a lower flow count are ranked higher by the scheduler.
  - `BandwidthUsage` plugin: Determines the best node to assign a pod based on the bandwidth usage of a node. The node with the least bandwidth usage (ingoing and outgoing traffic) per second is ranked highest.

    NOTE: Depending on the configured plugins, each plugin sends out scores to the scheduler. The scheduler takes the weighted scores from from all of the plugins and finds the best node to schedule a pod.

Deploy Network-Aware Pod Scheduling Components

See the following sections for information about deploying the components for network-aware pod scheduling:

"Metrics Collector Deployment" on page 26

"Central Collector Deployment" on page 27

"Contrail Scheduler Deployment" on page 30

Metrics Collector Deployment

CN2 includes the metrics collector in vRouter pod deployments by default. The `agent: default:` field of the vRouter spec contains a `collectors:` field which is configured with the metric collector reciever address. The example below shows the value `collectors: - localhost: 6700`. Since the the metrics collector runs in the same pod as the vRouter agent, it can communicate over the `localhost` port. Note that port 6700 is fixed as the metrics collector reciever address and cannot be changed. The vRouter agent sends metrics data to this address.

The following is a section of a default vRouter deployment with the collector enabled:

```
apiVersion: dataplane.juniper.net/v1
kind: Vrouter
metadata:
  name: contrail-vrouter-nodes
  namespace: contrail
spec:
  agent:
    default:
      collectors:
      - localhost:6700
      xmppAuthEnable: true
    sandesh:
      introspectSslEnable: true
```

Central Collector Deployment

The central collector Deployment object must always have a replica count set to 1. The following `Deployment` section shows an example:

```
spec:
  selector:
    matchLabels:
      component: central-collector
  replicas: 1
  template:
    metadata:
      labels:
        component: central-collector
```

A `configMap` provides key-value configuration data to the pods in your cluster. Create a `configMap` for the central collector configuration. This configuration is mounted in the container.
The following is an example of a central collector config file:

```
http_port: 9090
tls_config:
  key_file: /etc/config/server.key
```

```
  cert_file: /etc/config/server.crt
  ca_file: /etc/config/ca.crt
service_name: central-collector.contrail
metric_configmap:
  name: mc_configmap
  namespace: contrail
  key: config.yaml
```

This config file contains the following fields:

- `http_port`: Specifies the port that the central collector gRPC service runs on.
- `tls_config`: Specifies what `server_name` and `key_file` the central collector service is associated with. This field contains upstream (northbound API) server information.
- `service_name`: Specifies the name of the service the central collector exposes. In this case, `central-collector.contrail` is exposed as a service on top of the central collector `Deployment`. Consumers within the cluster can interact with the central collector using this service name.
- `metric_configmap`: The fields in this section designate the details of the metrics collector `configMap`. Central collector uses this information to configure a metrics-collector sink with the required metrics the sink wants to receive. The following is a sample command to create a `configMap`:

  ```
  kubectl create cm -n contrail central-collector-config -from-file=config.yaml=<path-to-config-
  file>
  ```

The following is an example of a central collector `Deployment`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: central-collector
  namespace: contrail
  labels:
    app: central-collector
spec:
  replicas: 1
  selector:
    matchLabels:
      app: central-collector
  template:
    metadata:
```

```
    labels:
      app: central-collector
  spec:
    securityContext:
      fsGroup: 2000
      runAsGroup: 3000
      runAsNonRoot: true
      runAsUser: 1000
    containers:
    - name: contrail-scheduler
      image: enterprise-hub.juniper.net/contrail-container-prod/central-collector:latest
      command:
      - /central-collector
      - --kubeconfig=/tmp/config/kubeconfig
      - --config=/etc/central-collector/config.yaml
      imagePullPolicy: Always
      volumeMounts:
      - mountPath: /tmp/config
        name: kubeconfig
        readOnly: true
      - mountPath: /etc/central-collector
        name: central-collector-config
        readOnly: true
      - mountPath: /etc/config/tls
        name: tls
        readOnly: true
    volumes:
    - name: kubeconfig
      secret:
        secretName: cc-kubeconfig
    - name: tls
      secret:
        secretName: central-collector-tls
    - name: central-collector-config
      configMap:
        name: central-collector-config
```

NOTE: Verify the `volume` and `volumeMounts` fieds before deploying.

The central collector service is exposed on top of the `Deployment` object. The following YAML file is an example of a central collector service file:

```
apiVersion: v1
kind: Service
metadata:
  name: central-collector
  namespace: contrail
spec:
  selector:
    component: central-collector
  ports:
    - name: grpc
      port: <port-as-per-config>
    - name: json
      protocol: TCP
      port: 10000
```

NOTE: The `name` field must match the service name specified in the central collector configuration. The namespace must match the namespace of the central collector Deployment. For example, `namespace: contrail`.

Contrail Scheduler Deployment

Perform the following steps to deploy the `contrail-scheduler`:

- Create a namespace for the `contrail-scheduler`.

```
kubectl create ns contrail-scheduler
```

- Create a `ServiceAccount` object (required) and configure the cluster roles for the `ServiceAccount`. A `ServiceAccount` assigns a role to a pod or component within a cluster. In this case, the fields `kind: ClusterRole` and `name: system:kube-scheduler` grant the `contrail-scheduler ServiceAccount` the same permissions as the default Kubernetes scheduler (`kube-scheduler`).

- Create a `configMap` for the VMI plugin configuration. You must create the `configMap` within the same namespace as the `contrail-scheduler Deployment`.

```
kubectl create configmap vmi-config -n contrail-scheduler --from-file=vmi-config=<path-to-vmi-config>
```

The following is an example of a VMI plugin config:

```
nodeLabels:
  "test-agent-mode": "dpdk"
maxVMICount: 64
address: "central-collector.contrail:9090"
```

- Create a `Secret` for the `kubeconfig` file. This file is then mounted in the `contrail-scheduler Deployment`. `Secrets` store confidential data as files in a mounted volume or as a container environment variable.

```
kubectl create secret generic kubeconfig -n contrail-scheduler --from-file=kubeconfig=<path-to-kubeconfig-file>
```

- Create a `configMap` for the `contrail-scheduler` config.

```
kubectl create configmap scheduler-config -n contrail-scheduler --from-file=scheduler-config=<path-to-scheduler-config>
```

The following is an example of a scheduler config:

```
apiVersion: kubescheduler.config.k8s.io/v1
clientConnection:
  acceptContentTypes: ""
  burst: 100
  contentType: application/vnd.kubernetes.protobuf
  kubeconfig: /tmp/config/kubeconfig
  qps: 50
enableContentionProfiling: true
enableProfiling: true
kind: KubeSchedulerConfiguration
leaderElection:
  leaderElect: false
```

```
profiles:
  - schedulerName: contrail-scheduler
  pluginConfig:
  - args:
    apiVersion: kubescheduler.config.k8s.io/v1
    kind: VMICapacityArgs
    config: /tmp/vmi/config.yaml
   name: VMICapacity
      - args:
          apiVersion: kubescheduler.config.k8s.io/v1
          kind: FlowsCapacityArgs
          address: central-collector.contrail:9090
        name: FlowsCapacity
      - args:
          apiVersion: kubescheduler.config.k8s.io/v1
          kind: BandwidthUsageArgs
          address: central-collector.contrail:9090
        name: BandwidthUsage
      plugins:
        multiPoint:
          enabled:
          - name: VMICapacity
            weight: 50
          - name: FlowsCapacity
            weight: 1
          - name: BandwidthUsage
            weight: 20
```

Note the following fields:

- `schedulerName`: The name of the scheduler you want to deploy.
- `pluginConfig`: Contains information about the plugins included in the `contrail-scheduler` deployment. The deployment includes the following plugins:
  - `VMICapacity`
  - `FlowsCapacity`
  - `BandwidthUsage`
- `config`: This field contains the filepath where the VMI plugin config is mounted.
- `multiPoint`: You can enable extension points for each of the included plugins. Instead of having to enable specific extension points for a plugin, the `multiPoint` field let's you enable or disable all of

the extension points that are developed for a given plugin. The weights of a plugin decide the priority of a particular score from a plugin. This means that at the end of scoring, all of the plugins send out a weighted score. A pod is scheduled on a node with the highest aggregated score.

- Create a `contrail-scheduler` `Deployment`. The following is an example of a `Deployment`:

```
apiVersion: apps/v1
    kind: Deployment
    metadata:
      name: contrail-scheduler
      namespace: contrail-scheduler
      labels:
        app: scheduler
    spec:
      replicas: 1
      selector:
        matchLabels:
          app: scheduler
      template:
        metadata:
          labels:
            app: scheduler
        spec:
          serviceAccountName: contrail-scheduler
          securityContext:
            fsGroup: 2000
            runAsGroup: 3000
            runAsNonRoot: true
            runAsUser: 1000
          containers:
          - name: contrail-scheduler
            image: <registry>/contrail-scheduler:<tag>
            command:
            - /contrail-scheduler
            - --authentication-kubeconfig=/tmp/config/kubeconfig
            - --authorization-kubeconfig=/tmp/config/kubeconfig
            - --config=/tmp/scheduler/scheduler-config
            - --secure-port=10271
            imagePullPolicy: Always
            livenessProbe:
              failureThreshold: 8
              httpGet:
```

```
            path: /healthz
            port: 10271
            scheme: HTTPS
          initialDelaySeconds: 30
          periodSeconds: 10
          timeoutSeconds: 30
        resources:
          requests:
            cpu: 100m
        startupProbe:
          failureThreshold: 24
          httpGet:
            path: /healthz
            port: 10271
            scheme: HTTPS
          initialDelaySeconds: 30
          periodSeconds: 10
          timeoutSeconds: 30
        volumeMounts:
        - mountPath: /tmp/config
          name: kubeconfig
          readOnly: true
        - mountPath: /tmp/scheduler
          name: scheduler-config
          readOnly: true
        - mountPath: /tmp/vmi
          name: vmi-config
          readOnly: true
      hostPID: false
      volumes:
      - name: kubeconfig
        secret:
          secretName: kubeconfig
      - name: scheduler-config
        configMap:
          name: scheduler-config
      - name: vmi-config
        configMap:
          name: vmi-config
```

After you apply this `Deployment`, the new `contrail-scheduler` is active.

Use the Contrail Scheduler to Deploy Pods

Enter the name of your `contrail-scheduler` to the `schedulerName` field to use the `contrail-scheduler` to schedule (deploy) new pods. The following is an example of a pod manifest with the `schedulerName` defined:

```
apiVersion: v1
 kind: Pod
 metadata:
   name: my-app
   labels:
     app: web
 spec:
   schedulerName: contrail-scheduler
   containers:
     - name: app
       image: busybox
       command:
       - sh
       - -c
       - sleep 500
```

2

CHAPTER

CN2 Apstra Integration

Extend Virtual Networks to Apstra | 37

Extend Virtual Networks to Apstra

SUMMARY

Starting in CN2 Release 23.1 or later, you can extend virtual networks from your Kubernetes cluster to the datacenter fabric managed by Apstra.

IN THIS SECTION

- Overview | 37
- Example: CN2 Kubernetes Deployment with SR-IOV Pods | 38
- Prerequisites | 40
- Installation Workflow | 42
- Install and Configure the CN2 Apstra Plug-In | 44
- Install the CN2 IPAM Plug-In | 48
- Intra-VN and Inter-VN Approaches | 49
- Introduction to Configuring Intra-VN Communication | 53
- Configure Intra-VN Communication | 56
- Introduction to Configuring Inter-VN Communication | 64
- Configure Inter-VN Communication | 66
- Configure Inter-VN Communication Between SR-IOV Pods | 67
- Configure Inter-VN Communication Between SR-IOV Pods and Non-SR-IOV Pods | 69
- Configure Inter-VN Communication Between SR-IOV Pods, Non-SRIOV Pods and BMS | 74

Overview

Data centers typically have a mix of containerized workloads (SR-IOV pods, non-SR-IOV pods) and BMS. SR-IOV servers are being used extensively in data centers as these servers enable efficient I/O virtualization. When you create workloads on SR-IOV servers and attach virtual functions to the pods, the workloads use the fabric underlay directly. However, you might have a scenario where there is a need for communication between SRIOV pods, non-SRIOV pods, and BMS.

The different types of workloads are as follows:

- **SR-IOV pods**: SR-IOV pods use the IP fabric underlay directly for communication. The SR-IOV technology enables the physical NIC to be split into several virtual functions. The pods attach to the virtual functions of the SR-IOV enabled NICs. SR-IOV-enabled NICs on servers are used to deliver efficient I/O virtualization. These virtual NICs or virtual functions can transmit and receive packets directly from the fabric to which the CN2 data network is attached.

- **Non-SR-IOV pods**: Non-SR-IOV pods use the vRouter overlays for communication with other non-SR-IOV pods.

- **BMS**: BMS are physical nodes and are not part of the CN2 cluster. BMS use the fabric underlay for communication with pods. On the BMS, you can run applications directly on the native OS or run applications on containers.

Juniper Apstra is used to provision the fabric to provide the required underlay connectivity for the different workloads. Apstra is Juniper's intent-based networking software that automates and validates the design, deployment, and operation of data center networks. CN2 integrates with the Apstra software to provision the fabric underlay. For more information about Apstra, see the Juniper Apstra User Guide.

NOTE: We refer to *primary nodes* in our documentation. Kubernetes refers to *master nodes*. References in this guide to primary nodes correlate with master nodes in Kubernetes terminology.

Example: CN2 Kubernetes Deployment with SR-IOV Pods

Figure 1 on page 39 shows an example of a CN2 Kubernetes deployment. This deployment uses Apstra to provision the IP fabric underlay for the SR-IOV pods. Table 1 on page 39 describes the different components.

Figure 2: CN2 Kubernetes Deployment

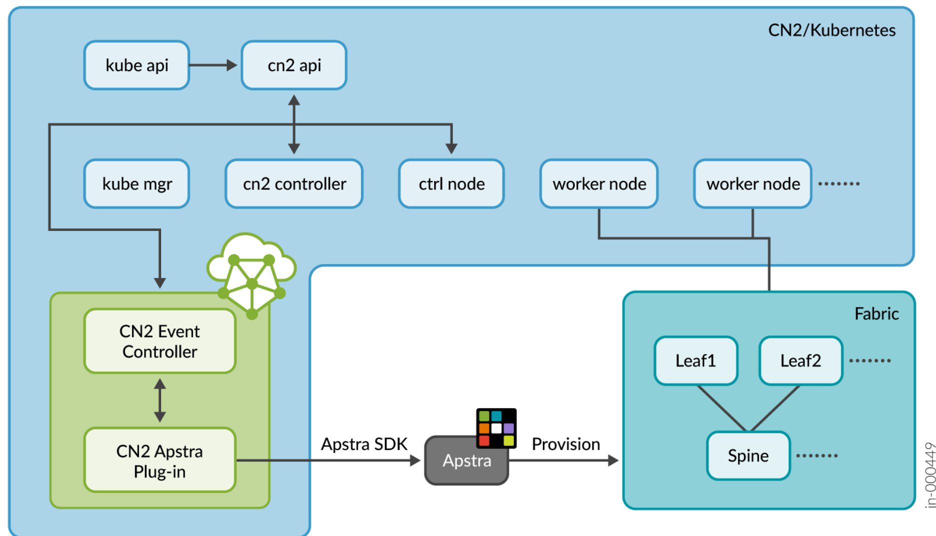


Table 3: CN2 Apstra Components

Component	Description
SR-IOV worker nodes	The SR-IOV worker nodes connect to the leaf devices in the IP fabric. These nodes, which are part of the CN2 cluster, have SRIOV-enabled NICs which can be split into virtual functions. When you create a pod on an SR-IOV worker node, the pod's interface can be attached to a virtual function on the SR-IOV-enabled NIC. The SRIOV-enabled NICs are in turn connected to the leaf devices in the IP fabric.
CN2 Apstra plug-in	The CN2 Apstra plug-in extends the virtual network to the IP fabric. This plug-in listens for CN2 Kubernetes events such as creating a NAD, attaching pods to the virtual network, and creating a Virtual Network Router (VNR). The plug-in then configures the IP fabric for the underlay through Apstra.

Table 3: CN2 Apstra Components *(Continued)*

Component	Description
Apstra	Apstra provisions the IP fabric to provide the required underlay connectivity for SR-IOV pods. Apstra also provides topology information regarding which leaf port is connected to which worker node. The CN2 Apstra plug-in uses this information to configure virtual network membership. The plug-in configures virtual-network membership on relevant fabric ports based on the worker node in which the SR-IOV pod is spawned.

Prerequisites

IN THIS SECTION

- Considerations | 41

To use this feature, you must install the following:

- Juniper Apstra version 4.1.2 or higher
- A CN2 cluster with the following items installed:
  - SR-IOV worker nodes that have SR-IOV-enabled NICs
  - Non-SR-IOV worker nodes
- The following plug-ins:
  - Multus Plug-In
  - SR-IOV Network Device Plug-In
  - CN2 IPAM Plug-In
- Licenses on the switches you are using in your topology

Juniper QFX switches require software licenses for advanced features. To ensure your that your IP fabric has the required licenses, see the Juniper Networks Licensing Guide.

NOTE: Make sure that you onboard the fabric to your Apstra blueprint as described in Step 4 in the "Installation Workflow" on page 42.

Considerations

Read through this list of considerations before you begin the installation:

- This feature assumes:
  - CN2 single-cluster deployments
  - Basic approaches for Intra-VN and Inter-VN communication between SR-IOV pods, non-SR-IOV pods, and BMS. Other forms of routing, such as hub-and-spoke routing, are not supported.
  - A simple spine-leaf topology where the SR-IOV worker node is connected to only one leaf device. If an SR-IOV worker node is connected to multiple leaf ports, ensure that you configure all the leaf ports on all the leaf devices to which this SR-IOV worker node is connected.
- Pods can be SR-IOV pods or non-SR-IOV pods. SR_IOV pods use the fabric underlay whereas Non-SR-IOV pods use the overlay though the vRouter.
- BMS is not part of the CN2 cluster.
- The entire BMS can be used exclusively for running applications directly on the host OS, where the IP is configured on the physical interface.

  Or

  BMS is running multiple VMs or containers, where the IP addresses are configured on the virtual interface.
- In the Intra-VN approach, the same subnet is used by CN2 for allocating IPs to both SR-IOV and non-SRI-OV pods. From the same subnet, you must use the unallocated IPs for configuring IPs in the BMS.
- When you create you a create BMS virtual network in Apstra, you must select all the leaf swtiches in that blueprint even if the BMS is connected to only one switch.
- You cannot edit the fields in the virtual networks and NADs (such as vlanID and VNI) that were created in CN2. To change these fields, you must re-create the virtual networks and NADs.
- Apstra accepts only VNIs greater or equal to 4096. Starting in CN2 release 23.1, on newer installations of CN2, we are allocating VNIs from 4096. Hence, this feature will not work on existing

CN2 setups. If you have an existing CN2 setup upgraded from a previous release, you must run a script to release free VNIs whose value is less that 4096.

- Ensure that the allocated vlanID does not conflict with those vlanID's allocated in Apstra. For example, you might want to use VLANs in the higher range (for example, 2000 and above) in CN2.
- The IP addresses for the pods are automatically allocated by the CN2 IPAM plug-in.
- In CN2, you must manually configure the routes on the pods for Inter-VN routing. For example: you can use the command `ip route add 10.30.30.0/8 via 10.20.20.1` to reach the `10.30.30.0/8` subnet.
- Overlapping IP addresses and bonded interfaces (link from the SR-IOV-enabled NICs to the leaf switches) are not in use.
- Only IPv4 addressing is supported for this feature.

Installation Workflow

Follow the steps in this procedure to install and to configure the CN2 Apstra plug-in and its prerequisites:

1. Install the Apstra Software.

   Install and configure Apstra version 4.1.2 or higher. See the Juniper Apstra Installation and Upgrade Guide.

   If you have an existing data center network, Apstra is already managing the fabric. Make sure that you assign the required resource pools such as ASNs and loopback IP addresses for the blueprint.

2. Install a CN2 Cluster.

   Install and configure a CN2 cluster that contains Kubernetes worker nodes. See the Install sections in the CN2 Installation Guide for Upstream Kubernetes or CN2 Installation Guide for OpenShift Container Platforms for instructions.

3. Install the Plug-Ins.

   a. Multus Plug-In:

      This plug-in enables you to attach multiple network interfaces to pods. See the Multus CNI for Kubernetes or Multus CNI for OpenShift documentation for installation instructions.

   b. SR-IOV Network Device Plug-In:

This plug-in discovers and advertises networking resources for SR-IOV virtual functions on a Kubernetes host. See the SR-IOV Network Device Plugin for Kubernetes or SR-IOV Network Device Plugin for OpenShift documentation for instructions.

c. CN2 Apstra Plug-In:

This plug-in is installed as part of the CN2 deployer. See "Install and Configure the CN2 Apstra Plug-In" on page 44 to install the plug-in.

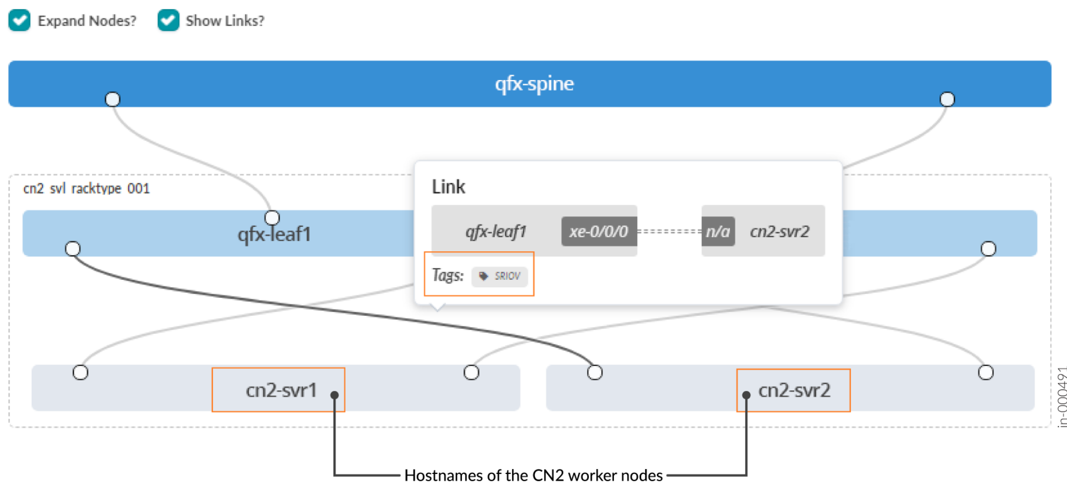
d. CN2 IPAM Plug-In:

This plug-in allocates IP addresses for the pods. You install this plug-in on the SR-IOV nodes. See "Install the CN2 IPAM Plug-In" on page 48 to install the plug-in.

4. Onboard the IP Fabric in Apstra.

You onboard the fabric in Apstra from the Apstra Web GUI. For onboarding instructions, see the Juniper Apstra User Guide..

- Make sure that you assign the required resource pools such as ASNs and loopback IP addresses to the blueprint.
- Make sure that the hostnames of the generic systems (that is, servers) in your Apstra blueprint matches the hostnames of the corresponding CN2 nodes. You must also tag the SR-IOV link that connects the SRIOV-enabled NICs on the worker nodes to the fabric ports. You'll enter this same value in the `sriov_link_tag` in the CN2 Apstra plug-in CRD when you install the plug-in. The following diagram shows an example of a topology in an Apstra blueprint where the hostnames of the generic system were edited to match the corresponding hostnames of the CN2 worker nodes. The diagram also shows the SRIOV tags that were configured for the aforementioned SR-IOV links.



5. Verify Your Installation.

   See "Verify Your Installation" on page 46 for instructions.

Install and Configure the CN2 Apstra Plug-In

IN THIS SECTION

- Verify Your Installation | 46

This section describes how to install and configure the CN2 Apstra plug-in.

The CN2 Apstra plug-in is installed as part of the deployer. The CN2 Apstra plug-in extends the virtual network to the fabric, listens for CN2 Kubernetes events (such as NAD creation), and configures the fabric for the underlay through the Apstra SDK.

Depending on your installation, use the following files to install and configure the plug-in:

- For Kubernetes, use the `single_cluster_deployer_example.yaml` file.
- For OpenShift, copy all the files in the `ocp/plugins` directory to one level up in the directory structure.

To install and configure the CN2 Apstra plug-in:

1. Uncomment the `apstra-plugin-secret` and `contrail-apstra-plugin` in your `single_cluster_deployer_example.yaml` file.

2. Enter your Apstra credentials (username and password) in the `apstra-plugin-secret` section in the corresponding deployer file. Make sure that your credentials are base64 encoded.

   For example:

```
apiVersion: v1
data:
  password: YWRtaW4K
  username: YWRtaW4K
kind: Secret
metadata:
  name: apstra-plugin-secret
```

```
  namespace: contrail
type: Opaque
```

3. Enter the parameters for `blueprint name`, `server_ip`, `sriov_link tag` in the `contrail-apstra-plugin` as shown in the following example. Make sure that the parameter for the `sriov_link tag` is the same parameter that you specified in the Apstra.

   This example also shows the image URL from where it fetches the `contrail-apstra-plugin` image. You can edit the image URL, if needed. For example, you can change the value of the `release_number` in the image to `23.1`.

```
apiVersion: plugins.juniper.net/v1alpha1
kind: ApstraPlugin
metadata:
 name: contrail-apstra-plugin
 namespace: contrail
spec:
  blueprint: ""
  common:
    containers:
    - image: enterprise-hub.juniper.net/contrail-container-prod/contrail-apstra-
plugin:release_number
      name: contrail-apstra-plugin
  server_ip: ""
  sriov_link_tag: ""
```

For help in understanding what each field means, run the `kubectl explain apstraplugin.spec` command.

> **NOTE**: The following example is only for informational purposes. You can run this command only after you deploy the CN2 Apstra plug-in.

```
kubectl explain apstraplugin.spec
KIND:     ApstraPlugin
VERSION:  plugins.juniper.net/v1alpha1

RESOURCE: spec <Object>

DESCRIPTION:
     ApstraPluginSpec defines the desired state of ApstraPlugin
```

```
FIELDS:
   blueprint    <string>
     The BluePrint in Apstra managing the Fabric which acts as underlay for this
     CN2 instance

   common    <Object>
     Common configuration for k8s pods and containers

   log_level    <string>
     The log level of Apstra plugin


   server_ip    <string>
     The Apstra server IP address

   sriov_link_tag    <string>
     Contains the tag value(eg: SRIOV) for the SRIOV links in Apstra BluePrint
```

With the above steps, you have made the required changes in the deployer to install the CN2 Apstra plug-in. You can now proceed with the CN2 installation by following the instructions in the CN2 Installation Guide for Upstream Kubernetes or the CN2 Installation Guide for OpenShift Container Platforms.

NOTE: Even after you have completed the CN2 installation, you can still edit the CN2 Apstra plug-in parameters in the deployer YAML(s) as mentioned in the above steps and then reinstall CN2.

Verify Your Installation

Run the following `kubectl` commands to verify that your installation is up and running. For example:

```
Check for the multus plug-in.
kubectl get pods -A | grep multus
kube-system      kube-multus-ds-dn5j8                                    1/1     Running
1              26d
kube-system      kube-multus-ds-mnd4j                                    1/1     Running
1              26d
kube-system      kube-multus-ds-xvt5v                                    1/1     Running
```

```
2             26d-
----------------------------------------------------------------------------------------------------
----------------
Check for the sriov-device-plugin.
kubectl get pods -A | grep sriov-device-plugin
kube-system       kube-sriov-device-plugin-amd64-2l792                    1/1     Running
0             6d8h
kube-system       kube-sriov-device-plugin-amd64-n2lxv                    1/1     Running
1             6d8h
kube-system       kube-sriov-device-plugin-amd64-v8tqx                    1/1     Running
1             26d
----------------------------------------------------------------------------------------------------
-----------------
Check if the virtual functions were discovered.
kubectl describe node jfm-qnc-05.lab.juniper.net | grep -A8 Allocatable
Allocatable:
  cpu:                              64
  ephemeral-storage:                189217404206
  hugepages-1Gi:                    0
  hugepages-2Mi:                    0
  intel.com/intel_sriov_netdevice:  7
  memory:                           263710404Ki
  pods:                             110
System Info:
----------------------------------------------------------------------------------------------------
---------
Check for the Apstra plug-in CRDs.
kubectl api-resources | grep apstra
apstraplugins                   plugins.juniper.net/v1alpha1                true
ApstraPlugin

----------------------------------------------------------------------------------------------------
----------
Check for the Apstra secret.
kubectl get secrets -A | grep apstra
contrail          apstra-plugin-secret        Opaque              2      20d

----------------------------------------------------------------------------------------------------
----------
Check for the contrail-apstra-plugin pod.
kubectl get pods -A | grep apstra
```

```
contrail          contrail-apstra-plugin-fd86dd969-5s94s                 1/1      Running    8
(6d7h ago)    12d
```

Install the CN2 IPAM Plug-In

Follow this procedure to install the CN2 IPAM plug-in for both Kubernetes and OpenShift deployments. This procedure assumes that CN2 is already installed on a Kubernetes cluster. In this procedure, we show a single-cluster deployment.

To install and configure the CN2 IPAM plug-in:

1. Run the `kubectl get nodes` command to view the list of available nodes.
2. Add the label `sriov:"true"` for each worker node with SR-IOV-enabled NICs. For example:

   ```
   kubectl edit nodes <node name>

     labels:
       beta.kubernetes.io/arch: amd64
       beta.kubernetes.io/os: linux
       kubernetes.io/arch: amd64
       kubernetes.io/hostname: jfm-qnc-02.englab.juniper.net
       kubernetes.io/os: linux
       server: jfm-qnc-02
       sriov: "true"
   ```

3. Add the `sriovLabelSelector` on the `contrail-vrouters-nodes` CRD.

   ```
   kubectl edit Vrouters/contrail-vrouter-nodes -n contrail
   ```

   In the CRD, under the spec field, add the following information:

   ```
    sriovLabelSelector:
      matchLabels:
        sriov: "true"
   ```

4. Verify the plug-in installation.

Wait for the vRouter pod to restart on the master node. Verify that the `cn2-ipam`and `sriov` binaries are installed, as shown in the following example:

```
ls /opt/cni/bin/
archive  bandwidth  bridge  cn2-ipam  contrail-k8s-cni  dhcp  firewall  host-device  host-
local  ipvlan  loopback  macvlan  multus  portmap  ptp  sbr  sriov  static  tuning  vlan  vrf
```

NOTE: The default location of the binary files depends on whether you use Kubernetes or OpenShift:

- For Kubernetes, the binaries reside in the `/opt/cni/bin/` directory.
- For OpenShift, the binaries reside in the `/var/lib/cni/bin/` directory.

Intra-VN and Inter-VN Approaches

IN THIS SECTION

- Intra-VN Approach | 49
- Inter-VN Approach | 52

This section shows the two approaches to configuring communication between SR-IOV pods, non-SR-IOV pods, and BMS: Intra-VN and Inter-VN.

You can use the Intra-VN approach or the Inter-VN approach depending on your requirement. If you want to see a summary of the configuration workflow for each approach, see Table 4 on page 55 or Table 5 on page 65.

Intra-VN Approach

In the Intra-VN approach, pods are attached to the same virtual network. By default, pods on the same virtual network can communicate with one another, regardless of whether:

The pods are spawned on the same worker nodes or on different worker nodes.

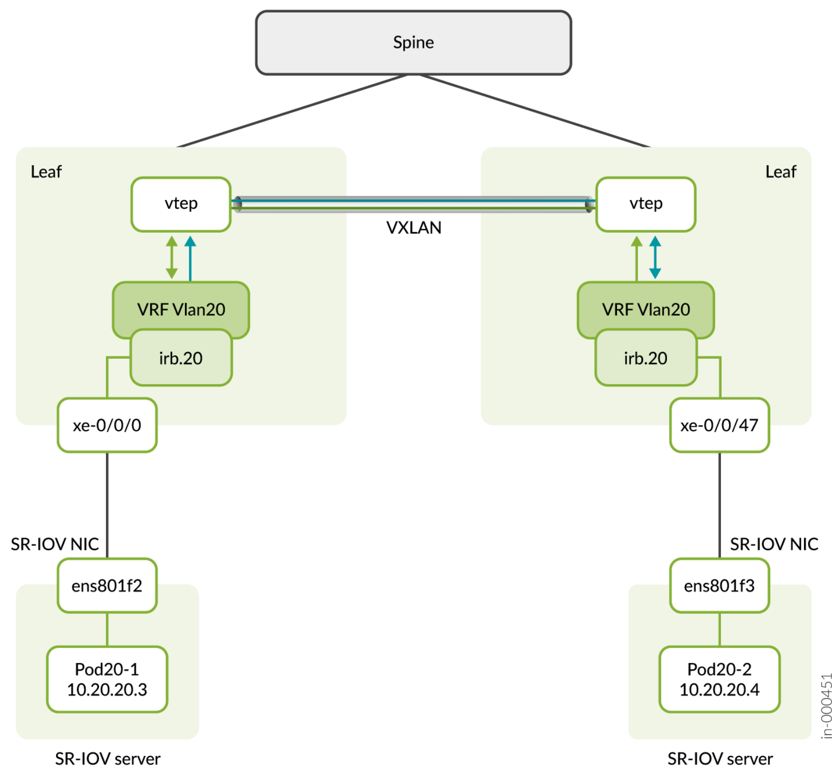
Or

The worker nodes are connected to the same leaf device or to a different leaf.

Intra-VN Approach: Communication Between SR-IOV Pods

Figure 2 on page 50 shows an example of an Intra-VN topology where SR-IOV pods are attached to the same virtual network. This spine-and-leaf topology shows two SR-IOV worker nodes. Each node has a physical NIC with SR-IOV enabled. These physical NICs (ens801f2 and ens801f3) can be split into virtual functions and attached to the pods for direct I/O. When packets travel these virtual functions, the packets are tagged with the appropriate VLAN. In this approach, the packets do not pass through the vRouter but go directly to the IP fabric underlay provisioned by Apstra.

Figure 3: Intra-VN: Communication Between SR-IOV Pods



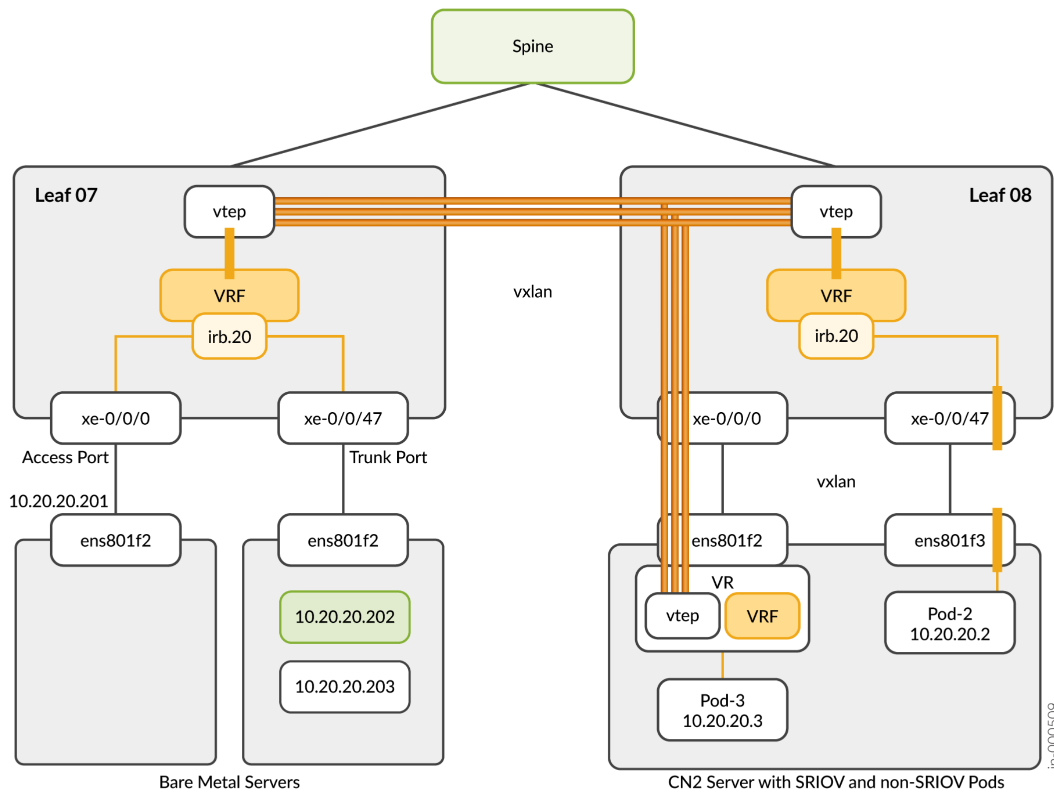
Intra-VN Approach: Communication Between SR-IOV Pods, Non-SR-IOV Pods, and BMS

Figure 3 on page 51 shows an example of SR-IOV pods, non-SR-IOV pods, and BMS attached to the same virtual network. The pods and BMS in this example use the same VNI and subnet.

In this approach, an EVPN session is set up between the CN2 control node and the IP fabric to mutually exchange EVPN Type-2 routes used by the VXLAN protocol. The VTEP interface for the SR-IOV pods and BMS is on the fabric. The VTEP interface for the non-SR-IOV pods resides on the vRouter.

In the following figure, BMS is attached to the same virtual network as the SR-IOV and non-SR-IOV pods, but is not part of the CN2 cluster.

Figure 4: Intra-VN Communication Between SR-IOV Pods, Non-SR-IOV Pods, and BMS



NOTE: For information on configuring Intra-VN communication, see "Introduction to Configuring Intra-VN Communication" on page 53.

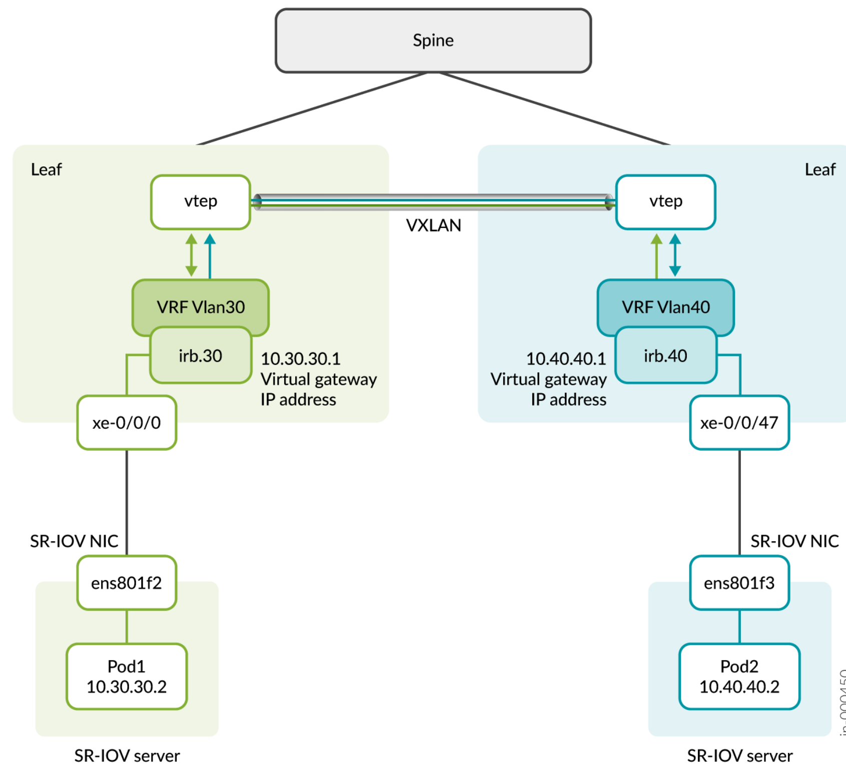
Inter-VN Approach

In the Inter-VN approach, CN2 pods and BMS workloads are attached to different virtual networks. The following sections shows the configuration required for configuring communication between pods and BMS.-

Inter-VN Approach: Communication Between SR-IOV Pods

The following figure shows an example of an Inter-VN topology between SR-IOV pods.

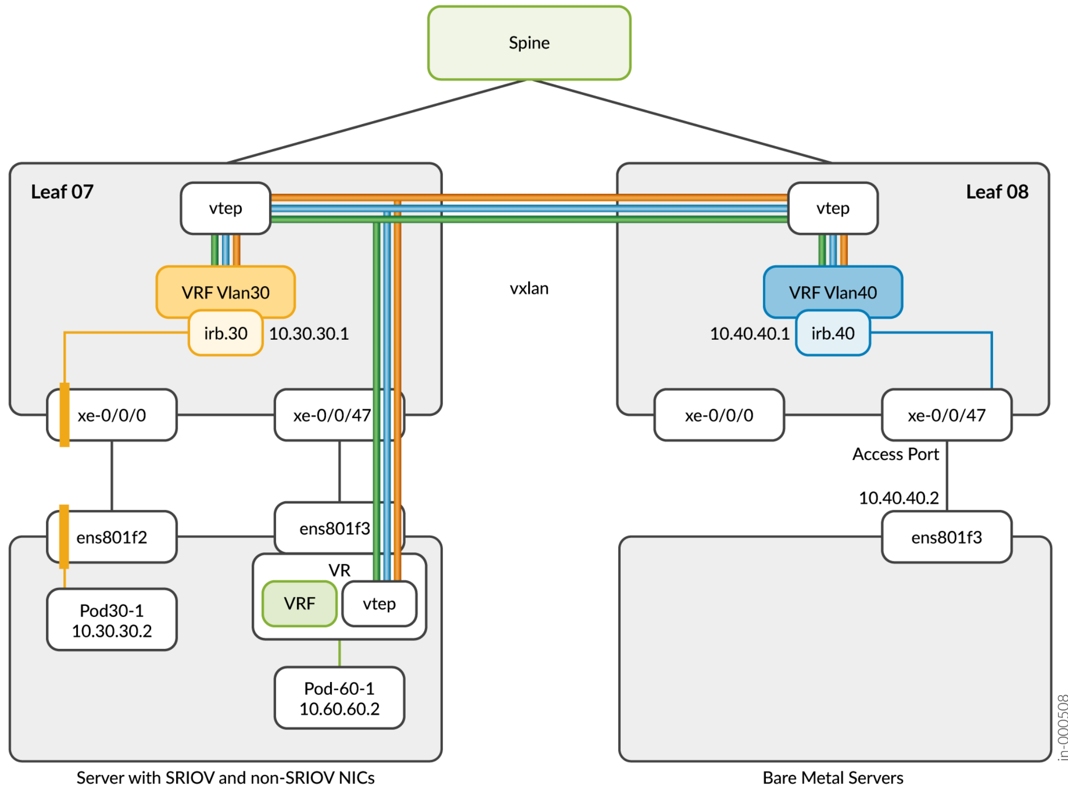
Figure 5: Inter-VN: Communication Between SR-IOV Pods



Inter-VN Approach: Communication Between SR-IOV Pods, Non-SR-IOV Pods, and BMS

In the Inter-VN approach, the pods and BMS belong to different virtual networks. To enable this communication, we are using EVPN Type-5 route exchanges between the CN2 control node and the fabric. For example:

Figure 6: Example: Inter-VN Communication Between SR-IOV Pods, Non-SR-IOV Pods, and BMS



NOTE: For information on configuring Inter-VN communication, see "Introduction to Configuring Inter-VN Communication" on page 64.

Introduction to Configuring Intra-VN Communication

Table 2 on page 55 summarizes the procedures required to configure Intra-VN communication between:

- SR-IOV pods and other SR-IOV pods
- SR-IOV pods and non-SR-IOV pods
- SR-IOV pods, non-SR-IOV pods and BMS

Review "Prerequisites" on page 40 and "Intra-VN Approach" on page 49 before proceeding to the table.

Table 4: Summary of Intra-VN Configuration Workflows

	SR-IOV Pods	Non-SR-IOV Pods	Non-SR-IOV Pods and BMS
SR-IOV Pods	1. Create an SR-IOV NAD (with the Asptra plug-in label) 2. Create an SR-IOV pod and attach the pod to the SR-IOV NAD. For detailed information for each step, see "Configure Intra-VN Communication Between SR-IOV Pods" on page 57.	**Pre-configuration setup** (perform only once) 1. Change the encapsulation priority to `vxlan` in CN2. 2. Create a remote EVPN gateway in Apstra. 3. Configure a `BGPRouter` in CN2. **Configuration steps** 1. Create a common `VirtualNetwork` with the Apstra plug-in label. 2. Create an SR-IOV NAD and reference the common virtual network. 3. Create an SR-IOV pod and attach the pod to the SR-IOV NAD. 4. Create a non-SR-IOV NAD and reference the common `VirtualNetwork` . 5. Create a non-SR-IOV pod and attach the pod to the non-SR-IOV NAD. For detailed information for each step, see "Configure Intra-VN Communication Between	1. Complete the pre-configuration setup and follow the configuration steps in the previous column (SR-IOV pod to non-SR-IOV pods). 2. In Apstra, assign the fabric port (connecting to the BMS) to the virtual network created in Apstra by CN2. For detailed information for each step, see "Configure Intra-VN Communication Between SR-IOV Pods, Non-SR-IOV Pods, and BMS" on page 64.

Table 4: Summary of Intra-VN Configuration Workflows *(Continued)*

	SR-IOV Pods	Non-SR-IOV Pods	Non-SR-IOV Pods and BMS
		SR-IOV Pods and Non-SR-IOV Pods" on page 58.	

Configure Intra-VN Communication

IN THIS SECTION

- Before You Begin | 56
- Configure Intra-VN Communication Between SR-IOV Pods | 57
- Configure Intra-VN Communication Between SR-IOV Pods and Non-SR-IOV Pods | 58
- Configure Intra-VN Communication Between SR-IOV Pods, Non-SR-IOV Pods, and BMS | 64

Follow the procedures in this section to configure Intra-VN communication between SR-IOV pods, non-SR-IOV pods, and BMS.

Before You Begin

Read through this list of considerations before you begin your configuration:

- In the Intra-VN approach, you use the same subnet across the pods and BMS. When configuring IP addresses in BMS, it is important to use the unallocated IP addresses to avoid collision with the IPs allocated by CN2.

  For example: If the subnet is 10.20.20.0/24, CN2 allocates IP addresses to the pod from the lower end, like 10.20.20.2, 10.20.20.3, and so on. For BMS, we suggest that you use IP addresses in the higher end, like 10.20.20.200, 10.20.20.201, 10.20.20.202 to avoid a collision.

  Depending on whether you configure the IP on the physical interface or on the virtual interface, you must use use the appropriate connectivity template (untagged or tagged, respectively) in Apstra. The template is used to configure the ports that connect the BMS to the fabric. See the Juniper Apstra User Guide for more information.

- To configure communication between SR-IOV pods and BMS or communication between non-SR-IOV pods and BMS, see "Configure Intra-VN Communication Between SR-IOV Pods, Non-SR-IOV Pods, and BMS" on page 64.

Configure Intra-VN Communication Between SR-IOV Pods

To configure Intra-VN communication between SR-IOV pods:

1. Create a NAD with the Apstra plug-in label, as shown in the following example:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: sriov-net20
  namespace: telco
  labels:
    juniper.net/plugin: apstra
  annotations:
    k8s.v1.cni.cncf.io/resourceName: intel.com/intel_sriov_netdevice
    juniper.net/networks: '{
      "vlanID": 20,
      "ipamV4Subnet": "10.20.20.0/24",
      "virtualNetworkNetworkId": 10020
    }'

spec:
  config: '{
  "type": "sriov",
  "cniVersion": "0.3.1",
  "name": "sriov-net20",
  "vlan": 20,
  "ipam": {
    "type": "cn2-ipam"
  }
}'
```

   When you create this object, the CN2 Apstra plug-in listens to the NAD and extends the `VirtualNetwork` to the fabric through Apstra.

2. Issue the `kubectl apply -f sriov_net20_nad.yaml` command to create the NAD.

   When the NAD is created, the CN2 Apstra plug-in listens for changes and provisions the fabric through the Apstra SDK.

3. Next, create an SR-IOV pod.

In the following example, we are referencing the NAD (`sriov-net20`) that we created in Step 1, in addition to the resource name (`intel.com/intel_sriov_netdevice`) for the virtual function.

```
apiVersion: v1
kind: Pod
metadata:
  name: sriov-pod-20-1
  namespace: telco
  annotations:
    k8s.v1.cni.cncf.io/networks: sriov-net20

spec:
  containers:
  - name:  sriov-pod-20-1
    image: gcr.io/cos-cloud/toolbox
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", "--" ]
    args: [ "while true; do sleep 300000; done;" ]
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
      privileged: true
    resources:
      requests:
        intel.com/intel_sriov_netdevice: '1'
      limits:
        intel.com/intel_sriov_netdevice: '1'
  nodeSelector:
    server: jfm-qnc-esxi-2
```

When you create the pod, the CN2 Apstra plug-in listens to the pod-creation event and provisions the fabric to assign the relevant fabric ports to the `VirtualNetwork`.

You have now configured Intra-VN communication between SR-IOV pods.

Configure Intra-VN Communication Between SR-IOV Pods and Non-SR-IOV Pods

Configuring Intra-VN communication between SR-IOV pods and non-SR-IOV pods involves two sets of steps: pre-configuration setup and configuration. You perform the pre-configuration setup only once, no matter how many virtual networks and pods you configure.

To complete the pre-configuration setup:

1. Change the encapsulation priority to `vxlan` in CN2 using the `kubectl edit GlobalVrouterConfig default-global-vrouter-config` command.

2. Create a remote EVPN gateway in Apstra. See the "Remote EVPN Gateways (virtual)" chapter in the Juniper Apstra User Guide for instructions.

3. In CN2, configure a `BGPRouter`.

   In the following example, we are referencing the fabric's loopback IP address: (`10.1.1.3`), ASN number (`65003`), and family (`- e-vpn`) used to exchange EVPN Type-2 routes.

```
apiVersion: core.contrail.juniper.net/v3
kind: BGPRouter
metadata:
  namespace: contrail
  name: bgprouter-qfx
  annotations:
    core.juniper.net/display-name: Sample BGP Router
    core.juniper.net/description:
      Represents configuration of BGP peers. All the BGP peers involved in
      Contrail system are under default Routing Instance of the default
      Virtual Network.
spec:
  parent:
    apiVersion: core.contrail.juniper.net/v3
    kind: RoutingInstance
    namespace: contrail
    name: default
  bgpRouterParameters:
    vendor: Juniper
    routerType: router
    address: 10.1.1.3
    identifier: 10.1.1.3
    autonomousSystem: 65003
    addressFamilies:
      family:
        - e-vpn
  bgpRouterReferences:
    - apiVersion: core.contrail.juniper.net/v3
      kind: BGPRouter
      namespace: contrail
      name: jfm-qnc-06.englab.juniper.net
```

To configure Intra-VN communication between SR-IOV pods and non-SR-IOV pods:

1. Create a common `VirtualNetwork` (with Apstra plug-in label) to extend the virtual network to the fabric.

   In the following example, we are creating a `Subnet` and a `VirtualNetwork` that references that subnet.

```
apiVersion: core.contrail.juniper.net/v3
kind: Subnet
metadata:
  namespace: telco
  name: net20-v4
spec:
  cidr: 10.20.20.0/24
---
apiVersion: "core.contrail.juniper.net/v3"
kind: VirtualNetwork
metadata:
    name: net20
    namespace: telco
    labels:

     juniper.net/plugin: apstra
spec:
    vlanID: 20
    virtualNetworkNetworkId: 10020
    routeTargetList:
    - target:10020:1
    v4SubnetReference:
        apiVersion: core.contrail.juniper.net/v3
        kind: Subnet
        name: net20-v4
        namespace: telco
```

2. Create a NAD for the SR-IOV pod (with Apstra plug-in label).

   In the following example, we are referencing the `VirtualNetwork` (`net20`) that we created in Step 1.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: sriov-net20
  namespace: telco
  labels:
```

```
    vn: web
    juniper.net/plugin: apstra
  annotations:
    k8s.v1.cni.cncf.io/resourceName: intel.com/intel_sriov_netdevice
    juniper.net/networks: '{
      "vlanID": 20,
      "virtualNetworkNetworkId": 10020,
      "virtualNetworkReference": {
         "apiVersion": "core.contrail.juniper.net/v3",
         "kind": "VirtualNetwork",
         "name": "net20",
         "namespace": "telco"
      },
      "routeTargetList": ["target:10020:1"]
    }'
spec:
  config: '{
  "type": "sriov",
  "cniVersion": "0.3.1",
  "name": "net20",
  "vlan": 20,
  "ipam": {
    "type": "cn2-ipam"
  }
}'
```

3. Create an SR-IOV pod.

   In the following example, we are referencing the NAD (`sriov-net20`) that we created in Step 2.

```
apiVersion: v1
kind: Pod
metadata:
  name: sriov-pod-20-1
  namespace: telco
  annotations:
    k8s.v1.cni.cncf.io/networks: sriov-net20

spec:
  containers:
  - name:  sriov-pod-20
    image: gcr.io/cos-cloud/toolbox
    imagePullPolicy: IfNotPresent
```

```
    command: [ "/bin/bash", "-c", "--" ]
    args: [ "while true; do sleep 300000; done;" ]
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
      privileged: true
    resources:
      requests:
        intel.com/intel_sriov_netdevice: '1'
      limits:
        intel.com/intel_sriov_netdevice: '1'
  nodeSelector:
    server: jfm-qnc-05
```

When you create the pod, the CN2 Apstra plug-in listens to the pod-creation event and provisions the fabric to assign the relevant fabric ports to the `VirtualNetwork`.

4. Create a NAD for the non-SR-IOV pod.

   In the following example, we are referencing the same virtual network (`net20`) that we created Step 1.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: non-sriov-net20
  namespace: telco
  labels:
    vn: web
  annotations:
    juniper.net/networks: '{
      "virtualNetworkNetworkId": 10020,
      "virtualNetworkReference": {
         "apiVersion": "core.contrail.juniper.net/v3",
         "kind": "VirtualNetwork",
         "name": "net20",
         "namespace": "telco"
      },
      "routeTargetList": ["target:10020:1"]
    }'
spec:
  config: '{
  "type": "contrail-k8s-cni",
  "cniVersion": "0.3.1",
```

```
  "name": "net20"
}'
```

5. Create a non-SR-IOV pod.

   In the following example, were are referencing the NAD (`non-sriov-net20`) we created in Step 4.

```
apiVersion: v1
kind: Pod
metadata:
  name: non-sriov-pod-20-1
  namespace: telco
  annotations:
    k8s.v1.cni.cncf.io/networks: non-sriov-net20

spec:
  containers:
  - name:  non-sriov-pod-20-1
    image: gcr.io/cos-cloud/toolbox
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", "--" ]
    args: [ "while true; do sleep 300000; done;" ]
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
      privileged: true
  nodeSelector:
    server: jfm-qnc-06
```

> **NOTE**: If you are configuring communication between the pods and BMS, proceed with Step 2 in "Configure Intra-VN Communication Between SR-IOV Pods, Non-SR-IOV Pods, and BMS" on page 64 to complete the configuration.

You have now configured Intra-VN communication between SR-IOV pods and non-SR-IOV pods.

Configure Intra-VN Communication Between SR-IOV Pods, Non-SR-IOV Pods, and BMS

> **NOTE**: This procedure describes how to configure Intra-VN communication between SR-IOV pods, non-SR-IOV pods, and BMS.
>
> - For communication between SR-IOV pods and BMS, follow these steps but do not create a non-SR-IOV NAD and non-SR-IOV pod.
> - For communication between non-SR-IOV pods and BMS, follow these steps but do not create an SR-IOV NAD and SR-IOV pod.

To configure Intra-VN communication between SR-IOV pods, non-SR-IOV pods, and BMS:

1. Complete the pre-configuration setup and configuration steps in the the procedure "Configure Intra-VN Communication Between SR-IOV Pods and Non-SR-IOV Pods" on page 58.
2. In Apstra, identify the `VirtualNetwork` that was created by CN2 based on the VNI.
3. Assign the relevant fabric ports (that connect to the BMS) in Apstra to this virtual network.

   Make sure that you use the appropriate connectivity template (untagged or tagged) to assign the ports to the virtual network. See the Juniper Apstra User Guide for instructions.

You have now configured Intra-VN communication between SR-IOV pods, non-SR-IOV pods, and BMS.

Introduction to Configuring Inter-VN Communication

Table 3 on page 65 summarizes the procedures required to configure Inter-VN communication between:

- SR-IOV pods and other SR-IOV pods
- SR-IOV pods and non-SR-IOV pods
- SR-IOV pods, non-SR-IOV pods, and BMS

Review the "Prerequisites" on page 40 and "Inter-VN Approach" on page 52 before proceeding to the table.

Table 5: Summary of Inter-VN Configuration Workflows

	SR-IOV Pods	Non-SR-IOV Pods	Non-SR-IOV Pods and BMS
SR-IOV Pods	1. Create an SR-IOV NAD (with Asptra plug-in label and vnr label). 2. Create an SR-IOV pod and attach the pod to the NAD. 3. Create a VNR. 4. Configure the required routes in the pods. For detailed information for each step, see "Configure Inter-VN Communication Between SR-IOV Pods" on page 67.	**Pre-configuration setup** (perform only once) 1. Change the encapsulation priority to `vxlan` in CN2. 2. Create a remote EVPN gateway in Apstra. 3. Create a `BGPRouter` in CN2. **Configuration steps** 1. Create an SR-IOV NAD (with Apstra plug-in label and vnr label). 2. Create an SR-IOV pod and attach the pod to the SR-IOV NAD. 3. Create a non-SR-IOV NAD (with vnr label). 4. Create a non-SR-IOV pod and attach the pod to the non-SR-IOV NAD. 5. Create a VNR and specify the routing type as `routingType:evpn`. 6. Configure the required routes in the pods. For detailed information for each step, see "Configure Inter-VN	1. Complete the pre-configuration setup and follow the configuration steps (1 through 5) in the previous column (SR-IOV pod to non-SR-IOV pods). 2. Manually create a virtual network for the BMS in Apstra. 3. In CN2, create a reference NAD (with vnr label) to the virtual network that was created in Step 2 above. 4. Configure the required routes in the BMS and also in the CN2 pods. For detailed information for each step, see "Configure Inter-VN Communication Between SR-IOV Pods, Non-SRIOV Pods and BMS" on page 74.

Table 5: Summary of Inter-VN Configuration Workflows *(Continued)*

	SR-IOV Pods	Non-SR-IOV Pods	Non-SR-IOV Pods and BMS
		Communication Between SR-IOV Pods and Non-SR-IOV Pods" on page 69.	

Configure Inter-VN Communication

IN THIS SECTION

- Before You Begin | **66**

Follow the procedures in this section to configure Inter-VN communication between SR-IOV pods, non-SR-IOV pods, and BMS.

Before You Begin

Read through the this list of considerations before you begin your configuration:

- For Inter-VN routing, you must create a `VirtualNetworkRouter` in CN2.
- In the Inter-VN approach, you must manually configure the routes on the pods. For example: you can use the command `ip route add 10.30.30.0/8 via 10.20.20.1` to reach the `10.30.30.0/8` subnet.
- The QFX5200 switch does not support EVPN Type-5 routing and edge-routing bridging (ERB). See Edge-Routing Bridging for QFX Series Switches for more information.

  For a list of supported Juniper devices for use in an Inter-VN topology, see Layer 3 connectivity in an EVPN-VXLAN topology. Also, make sure that the QFX devices are running Junos OS version 20.2R2.11 or above.

Configure Inter-VN Communication Between SR-IOV Pods

To configure Inter-VN communication between SR-IOV pods:

1. Create an SR-IOV NAD (with vn label and Apstra plug-in label), as shown in the following example:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: net30
  namespace: telco
  labels:
    vn: web
    juniper.net/plugin: apstra
  annotations:
    k8s.v1.cni.cncf.io/resourceName: intel.com/intel_sriov_netdevice
    juniper.net/networks: '{
      "vlanID": 30,
      "ipamV4Subnet": "10.30.30.0/24",
      "virtualNetworkNetworkId": 10030
    }'

spec:
  config: '{
  "type": "sriov",
  "cniVersion": "0.3.1",
  "name": "net30",
  "vlan": 30,
  "ipam": {
    "type": "cn2-ipam"
  }
}'
```

2. Issue the `kubectl apply -f sriov_net30_nad.yaml` command to create the NAD.

   The CN2 Apstra plug-in listens to the NAD event and extends the virtual network to the fabric through Apstra. You can create additional NADs as needed, following the same pattern.

3. Create an SR-IOV pod and attach the pod to the SR-IOV NAD.

In the following example, we are referencing the NAD (`sriov-net30`) that we created in Step 1 in addition to the resource name (`intel.com/intel_sriov_netdevice`) for the virtual function.

```
apiVersion: v1
kind: Pod
metadata:
  name: sriov-pod-30-1
  namespace: telco
  annotations:
    k8s.v1.cni.cncf.io/networks: net30

spec:
  containers:
  - name:  sriov-pod-30-1
    image: gcr.io/cos-cloud/toolbox
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", "--" ]
    args: [ "while true; do sleep 300000; done;" ]
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
      privileged: true
    resources:
      requests:
        intel.com/intel_sriov_netdevice: '1'
      limits:
        intel.com/intel_sriov_netdevice: '1'
  nodeSelector:
    server: jfm-qnc-05
```

4. Run the `kubectl apply -f pod.yaml` command to create the pod.

   The CN2 Apstra plug-in listens to the pod-creation event and provisions the fabric to assign the relevant fabric ports to the virtual network.

5. Create a VNR to route the different virtual networks with a common label.

   In the following example, the common label is: `vn: web`.

```
apiVersion: core.contrail.juniper.net/v3
kind: VirtualNetworkRouter
metadata:
  namespace: telco
```

```
  name: vnr-web
  annotations:
    core.juniper.net/display-name: vnr-web
  labels:
    vnr: web
    ns: telco
spec:
  type: mesh
  routingType: evpn
  l3vxlanNetworkIdentifier: 50000
  virtualNetworkSelector:
    matchLabels:
      vn: web
```

6. Configure the required routes in the pods to reach the other subnet. For example:

```
ip route add 10.30.30.0/24 via 10.20.20.1
```

You have now configured Inter-VN communication between SR-IOV pods.

Configure Inter-VN Communication Between SR-IOV Pods and Non-SR-IOV Pods

Configuring Inter-VN communication between SR-IOV pods and non-SR-IOV pods involves two sets of steps: pre-configuration setup and configuration. You perform the pre-configuration setup only once, no matter how many virtual networks and pods you configure.

To complete the pre-configuration setup:

1. Change the encapsulation priority to `vxlan` in CN2 using the `kubectl edit GlobalVrouterConfig default-global-vrouter-config` command.

2. Create a remote EVPN gateway in Apstra. See the "Remote EVPN Gateways (virtual)" chapter in the Juniper Apstra User Guide for instructions.

3. In CN2, configure a `BGPRouter`.

   In the following example, we are referencing the fabric's loopback IP address: (`10.1.1.3`), ASN number (`65003`), and family (`- e-vpn`) used to exchange EVPN Type-2 routes.

```
apiVersion: core.contrail.juniper.net/v3
kind: BGPRouter
```

```
metadata:
  namespace: contrail
  name: bgprouter-qfx
  annotations:
    core.juniper.net/display-name: Sample BGP Router
    core.juniper.net/description:
      Represents configuration of BGP peers. All the BGP peers involved in
      Contrail system are under default Routing Instance of the default
      Virtual Network.
spec:
  parent:
    apiVersion: core.contrail.juniper.net/v3
    kind: RoutingInstance
    namespace: contrail
    name: default
  bgpRouterParameters:
    vendor: Juniper
    routerType: router
    address: 10.1.1.3
    identifier: 10.1.1.3
    autonomousSystem: 65003
    addressFamilies:
      family:
        - e-vpn
  bgpRouterReferences:
    - apiVersion: core.contrail.juniper.net/v3
      kind: BGPRouter
      namespace: contrail
      name: jfm-qnc-06.englab.juniper.net
```

To configure Inter-VN communication between SR-IOV pods and non-SR-IOV pods:

1. Create an SR-IOV NAD (with vn label and Apstra plug-in label), as shown in the following example:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: net30
  namespace: telco
  labels:
    vn: web
    juniper.net/plugin: apstra
```

```
  annotations:
    k8s.v1.cni.cncf.io/resourceName: intel.com/intel_sriov_netdevice
    juniper.net/networks: '{
      "vlanID": 30,
      "ipamV4Subnet": "10.30.30.0/24",
      "virtualNetworkNetworkId": 10030
    }'

spec:
  config: '{
  "type": "sriov",
  "cniVersion": "0.3.1",
  "name": "net30",
  "vlan": 30,
  "ipam": {
    "type": "cn2-ipam"
  }
}'
```

2. Create an SR-IOV pod and attach the pod to the SR-IOV NAD.

   In this example we are referencing the NAD (`net30`) that we created in Step 1 and the resource name for the SR-IOV pod (`intel.com/intel_sriov_netdevice`) for the virtual function.

```
apiVersion: v1
kind: Pod
metadata:
  name: sriov-pod-30-1
  namespace: telco
  annotations:
    k8s.v1.cni.cncf.io/networks: net30

spec:
  containers:
  - name:  sriov-pod-30-1
    image: gcr.io/cos-cloud/toolbox    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", "--" ]
    args: [ "while true; do sleep 300000; done;" ]
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
      privileged: true
```

```
    resources:
      requests:
        intel.com/intel_sriov_netdevice: '1'
      limits:
        intel.com/intel_sriov_netdevice: '1'
  nodeSelector:
    server: jfm-qnc-05
```

When you create the pod, the CN2 apstra plug-in listens to the pod-creation event and provisions the fabric to assign the relevant fabric ports to the VirtualNetwork.

3. Create a non-SR-IOV NAD (with vn label), as shown in the following example:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: non-sriov-net20
  namespace: telco
  labels:
    vn: web
    juniper.net/plugin: apstra
  annotations:
    juniper.net/networks: '{
      "virtualNetworkNetworkId": 10020,
      "ipamV4Subnet": "10.20.20.0/24",
      "routeTargetList": ["target:10020:1"]
    }'
spec:
  config: '{
  "type": "contrail-k8s-cni",
  "cniVersion": "0.3.1",
  "name": "net20"
}'
```

4. Create a non-SR-IOV pod and attach the pod to the non-SR-IOV NAD.

   Note that we are referencing the NAD (`net20`) that we created in Step 3.

```
apiVersion: v1
kind: Pod
metadata:
  name: non-sriov-pod-20-1
  namespace: telco
```

```
  annotations:
    k8s.v1.cni.cncf.io/networks: net20

spec:
  containers:
  - name:  non-sriov-pod-20-1
    image: gcr.io/cos-cloud/toolbox
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", "--" ]
    args: [ "while true; do sleep 300000; done;" ]
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
      privileged: true
  nodeSelector:
    server: jfm-qnc-06
```

5. Create a VNR to route the different virtual networks with a common label. In the following example, the common label is: `vn: web`.

```
apiVersion: core.contrail.juniper.net/v3
kind: VirtualNetworkRouter
metadata:
  namespace: telco
  name: vnr-web
  annotations:
    core.juniper.net/display-name: vnr-web
  labels:
    vnr: web
    ns: telco
spec:
  type: mesh
  routingType: evpn
  l3vxlanNetworkIdentifier: 50000
  virtualNetworkSelector:
    matchLabels:
      vn: web
```

NOTE: If you are configuring communication between the pods and BMS, can now proceed to Step 2 in "Configure Inter-VN Communication Between SR-IOV Pods, Non-SRIOV Pods and BMS" on page 74 to complete the configuration.

6. Configure the required routes in the pods. For example:

```
ip route add 10.30.30.0/24 via 10.20.20.1
```

You have now configured Inter-VN communication between SR-IOV pods and non-SR-IOV pods.

Configure Inter-VN Communication Between SR-IOV Pods, Non-SRIOV Pods and BMS

To configure Inter-VN communication between SR-IOV pods, non-SR-IOV pods, and BMS:

1. Complete the pre-configuration setup and follow Steps 1 though 5 in the procedure "Configure Inter-VN Communication Between SR-IOV Pods and Non-SR-IOV Pods" on page 69.
2. Manually create a virtual network for the BMS in Apstra. See the Juniper Apstra User Guide for instructions.

NOTE: Although the required `VirtualNetwork` can be created in Apstra using CN2, we are also addressing the use case where the `VirtualNetwork` needed for BMS has already been created in Apstra.

3. In CN2, create a reference NAD (with vn label) with the same name as the BMS virtual network in Apstra. Also, add the label `juniper.net/ssor: apstra` to sync this NAD from Apstra into CN2. For example:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: bms-net40-vn
  namespace: telco
  labels:
    vn: web
    juniper.net/plugin: apstra
    juniper.net/ssor: apstra
```

```
  annotations:
    juniper.net/networks: '{
    }'
```

4. Configure the required routes in the BMS and also in the CN2 pods. For example:

```
ip route add 10.30.30.0/24 via 10.40.40.1
```

You have now configured Inter-VN communication between SR-IOV pods, non-SR-IOV pods, and BMS.

3

CHAPTER

CN2 Security

Kubernetes Network Policies | 77

Security Policies | 83

Encrypt Secret Data at Rest | 88

Configure Management and Control Plane Authentication with TLS Encryption | 88

Kubernetes Network Policies

SUMMARY

Juniper Cloud-Native Contrail® Networking™ (CN2) lets you deploy Kubernetes network policies within the Contrail firewall security policy framework. You must use a Container Network Interface (CNI) that supports `NetworkPolicy`, like Contrail, to deploy a network policy. This topic provides information about how to deploy a Kubernetes network policy in environments running CN2.

IN THIS SECTION

- Kubernetes Network Policy Overview | 77
- Deploy a Kubernetes Network Policy in Cloud-Native Contrail Networking | 80
- Kubernetes Network Policy matchExpressions | 83

Kubernetes Network Policy Overview

Kubernetes network policies let you specify how pods communicate with other pods and network endpoints. A Kubernetes `NetworkPolicy` resource enables a pod to communicate with:

- Other pods in the allowlist (a pod cannot block access to itself).
- Namespaces in the allowlist.
- IP blocks, or Classless Inter-Domain Routing (CIDR).

Kubernetes network policies apply only to pods within a namespace and define ingress (source) and egress (destination) rules. Kubernetes network policies have the following characteristics when applied to a pod:

- Pod specific and apply to a single pod or a group of pods. Network policy rules dictate the traffic to that pod.
- Define traffic rules for a pod for ingress traffic, egress traffic, or both. If you don't specify a direction explicitly, the policy applies to the ingress direction by default.
- Must contain explicit rules that specify traffic from the allowlist in the ingress and egress directions. Traffic that does not match the allowlist rules is denied.
- Permitted traffic includes traffic matching any of the network policies applied to a pod.

Kubernetes network policies have the following additional characteristics:

- When not applied to a pod, that pod accepts traffic from all sources.

- Act on connections rather than individual packets. For example, if traffic from pod A to pod B is allowed by the configured policy, then the packets from pod B to pod A are also allowed, even if the policy in place does not allow pod B to initiate a connection to pod A.

A Kubernetes network policy comprises the following sections:

- `spec`: Describes the desired state of a Kubernetes object. For a network policy, the `podSelector` and `policyTypes` fields within the spec specify the rules for that policy.
- `podSelector`: Selects the groups of pods to which the policy applies. An empty `podSelector` selects all pods in the namespace.
- `policyTypes`: Specifies whether the policy applies to ingress traffic from selected pods or egress traffic to selected pods. If no `policyTypes` are specified, the ingress direction is selected by default.
- `ingress`: Allows ingress traffic that matches the `from` and `ports` sections. In the following example, the ingress rule allows connections to all pods in the `dev` namespace with the label app: `webserver-dev` on TCP port 80 from:
  - Any pod in the default namespace with the label `app: client1-dev`.
  - All IP addresses within the 10.169.25.20/32 range.
  - Any pod in the default namespace with the label `project: jtac`.
- `egress`: Allows egress traffic that matches the `to` and `ports`sections. In Example 1, the egress rule allows connections from any pod in the default namespace with the label `app: dbserver-dev` to port TCP 80.
- `ipBlock`: Selects IP CIDR ranges to allow as ingress sources or egress destinations. The `ipBlock` section of a network policy contains the following two fields:
  - cidr (ipBlock.cidr): The network policy allows egress traffic to, or ingress traffic from, the specified IP range.
  - except (ipBlock.except): Kubernetes expects traffic in the specified IP range not to match the policy. The network policy denies ingress traffic to, or egress traffic from, the IP range specified in `except`.

    NOTE:

    `exceptexceptexceptexcept`

The following `NetworkPolicy` resource example shows `ingress` and `egress` rules:

```
#policy1-do.yaml
apiVersion: networking.k8s.io/v1
```

```
kind: NetworkPolicy
metadata:
  name: policy1
  namespace: dev
spec:
  podSelector:
    matchLabels:
      app: webserver-dev
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 10.169.25.20/32
    - namespaceSelector:
        matchLabels:
          project: jtac
    - podSelector:
        matchLabels:
          app: client1-dev
    ports:
    - protocol: TCP
      port: 80
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: dbserver-dev
    ports:
    - protocol: TCP
      port: 80
```

In this example, ingress TCP traffic from IPs within CIDR `10.169.25.20/32` from `port: 80` is allowed. Egress traffic to pods with `matchLabels` `app: dbserver-dev` to `TCP port: 80` is allowed.

Deploy a Kubernetes Network Policy in Cloud-Native Contrail Networking

In CN2, after you configure and deploy a Kubernetes network policy, that policy is created automatically in Contrail. Here's an example of a Kubernetes network policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

This policy results in the following objects being created in CN2:

Tags on page 81

Address Groups on page 81

Firewall Rules on page 81

Firewall Policy on page 82

Table 6: Tags

Key	Value
role	db
namespace	default
project	myproject
role	frontend

Table 7: Address Groups

Name	Prefix
test-network-policy-except	172.17.1.0/24
test-network-policy	172.17.0.0/16
test-network-policy-egress	10.0.0.0/24

Table 8: Firewall Rules

Rule Name	Action	Service	Endpoint1	Direction	Endpoint2
default-ingress-test-network-policy-0-ipBlock-0-17x.xx.1.0/24-0	deny	tcp:6379	role=db && namespace=default	ingress	Address Group: 172.17.1.0/24

Table 8: Firewall Rules *(Continued)*

Rule Name	Action	Service	Endpoint1	Direction	Endpoint2
default-ingress-test-network-policy-0-ipBlock-0-cidr-17x.xx.0.0/16-0	pass	tcp:6379	role=db && namespace=default	ingress	Address Group: 172.17.0.0/16
default-ingress-test-network-policy-0-namespaceSelector-1-0	pass	tcp:6379	role=db && namespace=default	ingress	project=myproject
default-ingress-test-network-policy-0-podSelector-2-0	pass	tcp:6379	role=db && namespace=default	ingress	namespace=default && role=frontend
default-egress-test-network-policy-ipBlock-0-cidr-10.0.0.0/24-0	pass	tcp:5978	role=db && namespace=default	egress	Address Group: 10.0.0.0/24

Table 9: Firewall Policy

Name	Rules
default-test-network-policy	default-ingress-test-network-policy-0-ipBlock-0-172.17.1.0/24-0, default-ingress-test-network-policy-0-ipBlock-0-cidr-172.17.0.0/16-0 default-ingress-test-network-policy-0-namespaceSelector-1-0 default-ingress-test-network-policy-0-podSelector-2-0, default-egress-test-network-policy-ipBlock-0-cidr-10.0.0.0/24-0

Kubernetes Network Policy matchExpressions

Starting in Cloud-Native Contrail Networking (CN2) version 22.3, CN2 supports Kubernetes Network Policy with `matchExpressions`. For more information about `matchExpressions`, see "Resources that support set-based requirements" in the Kubernetes documentation.

Security Policies

SUMMARY

Starting in CN2 Release 23.1, CN2 supports Namespace Security Polices. Namespace policies allows you to define security polices from endpoints within a namespace, or to an external IP address.

IN THIS SECTION

- Namespace Security Policies | 83

Namespace Security Policies

IN THIS SECTION

- Overview of Namespace Security Policies | 83
- Policy Priority: Namespace Security Polices and Kubernetes Network Policies | 85
- Example of a Namespace Security Policy | 86

Overview of Namespace Security Policies

CN2 supports Kubernetes network policies to control flow of traffic to and from Kubernetes workloads to other Kubernetes workloads or IP addresses. Kubernetes network policies are developer-centric policies where a developer decides which traffic is allowed in and out of their workloads.

Although Kubernetes network policies are useful in restricting the communication between pods in a namespace, you can only control ingress and egress traffic from a selected set of pods. As an administrator, these policies can be complex to use, specifically if you are interested in end-to-end traffic flow based on application or security requirements.

For end-to-end traffic flow, CN2 supports namespace security policies. Namespace security policies are administrative-centric policies. These policies allows you to define polices from a source endpoint to a destination endpoint within a namespace. With Namespace security policies, you can allow or deny traffic between pods in a namespace or to an external IP address.

Namespace security policies have the following characteristics:

- You can only define security polices within a namespace. By default, security policies can only manage traffic within its own namespace.
- When you define a security policy, the policy is applied on all pods in a namespace.
- By default, if a pod does not match any user-defined policies, traffic is denied.
- You define security policies within a namespace using rules. Each rule has a source endpoint and destination endpoint. For ease of use, the security polices use the same endpoints as Kubernetes network policies (`namespace`, `podselector`, `IPblock`).
- Namespace security polices can contain one or multiple rules. For example:

```
Rule:
     Allow
     Src:
             (174.16.10.0/16 except 174.16.10.40/28)
              (174.16.10.40/24 except 174.16.10.70/32})

     Dst:
             (175.16.10.0/24 except 175.16.10.40/32)
             (175.16.10.0/24 except 175.16.10.70/32))

     port:[9000:9010])
```

- Namespace security policies support two actions: `pass` and `deny`. The default action is `deny`. Actions are per-policy, not per rule. This means that any rules you define follows the same action.
- For each rule, you can define the destination ports or destination port range (combination of source, destination, and port). For example, let's say you want particular IP address or source to access the server on a specified destination port or range of ports.

NOTE: Namespace security polices only support IPv4 IP addresses.

A Namespace security policy consists of the following sections:

- **Spec**

  The `spec` field contains the rules and actions you need to define a security policy. The `rules` contain a list of user defined source and destination matches.

  The rules are categorized, as follows:

  - `SrcEP`: List of source endpoints for a rule match criteria (`podSelector` or `ipBlock`).
  - `DstEP`: List of destination endpoints for a rule match criteria (`podSelector` or `ipBlock`).
  - `Ports`: List of destination ports to be matched for this rule (contains a combination of ports and protocol).

- `Action`: Action to be taken if any policy rule matches (`pass` or `deny`). The default action is `deny`.

To see an example of a security policy, see "Example of a Namespace Security Policy" on page 86.

Policy Priority: Namespace Security Polices and Kubernetes Network Policies

Because CN2 uses both Kubernetes network polices and namespace security policies, we use a sequence to determine how rules are processed between the two different policies.

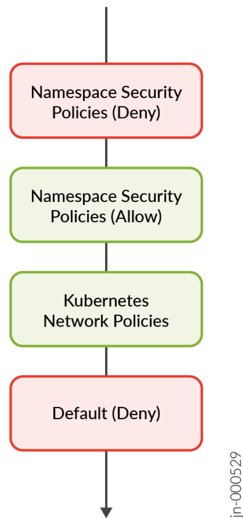
Figure 1 on page 86 shows how Kubernetes network policies along with namespace security policies are prioritized. The rules for namespace security polices are prioritized first, followed by Kubernetes network policies.

> **NOTE**: All Namespace security policies are user-defined.

Each policy is prioritized, as follows:

1. Namespace Security Policies (Deny): By default, if a pod does not match any policies in CN2, traffic is denied. The **deny** policies are given a higher priority over the **allow** policies.
2. Namespace Security Policies (Allow): If you do not define a policy in CN2, traffic is allowed between all pods in the corresponding namespace.
3. Kubernetes Network Polices: By default, a Kubernetes network policy allows traffic from all sources.
4. Default (Deny): If a pod does not match any Kubernetes network policies, traffic is denied.

Figure 7: Policy Priority: Namespace Security Polices and Kubernetes Network Policies



Example of a Namespace Security Policy

Here is an example of a namespace security policy:

NOTE: When you create a namespace security policy, you only need to provide the namespace and the source and destination endpoints (`namespace`, `podSelector`, `ipBlock`).

```
apiVersion: core.contrail.juniper.net/v3
            kind: NamespaceSecurityPolicy
            metadata:
              name: allow-hr11-to-fac12
              namespace: ns-svl
            spec:
              rules:
               - srcEP:
                   endPoints:
                     - podSelector:
                         matchLabels:
                           dept: hr
                         matchExpressions:
                           - {key: tier, operator: In, values: [one]}
                     - ipBlock:
```

```
                cidr: 174.19.12.11/32
          dstEP:
            endPoints:
              - podSelector:
                  matchLabels:
                    dept: fac
              - ipBlock:
                  cidr: 174.19.12.12/32
          ports:
            - protocol: TCP
              port: 3300
              endPort: 3400
      action: pass
```

In this example, we defined a security policy in namespace: `ns-svl`. In this policy, we've created a combination of rules for the source and destination endpoints. Table 1 on page 87 describes the different combinations.

For example, any traffic generated from the source (`podSelector` or `iBlock`) to the destination (`podSelector` or `ipBlock`) is allowed. The remaining traffic in the namespace is denied.

We've also specified the destination ports `3300` and `3400`. This allows TCP traffic between the destination ports and between the source and destination endpoints.

Table 10: Example: Source and Destination Combinations

Source	Destination
Pod with label `dept:hr` to:	• Pod with label `dept:fac` or destination (`dstEP`). Or, • `ipBlock cidr:174.19.12.12/32`.
ipBlock`174.19.12.11/32` to:	• Pod with label `dept:fac` or destination (`dstEP`). Or, • `ipBlock: cidr:174.19.12.12/32`.

SEE ALSO

Kubernetes Network Policies

Encrypt Secret Data at Rest

Starting in CN2 Release 22.1, Juniper supports encryption of secret data at rest. CN2 automatically encrypts secret data at rest in your Kubernetes cluster and encrypts any password that you configure. A secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Data at rest encryption is a cybersecurity practice of encrypting stored data to prevent unauthorized access.

Refer to the Kubernetes documentation titled Encrypting Secret Data at Rest for instructions on how to enable and configure this feature.

Configure Management and Control Plane Authentication with TLS Encryption

SUMMARY

This topic describes how to configure management and control plane authentication with TLS Encryption in CN2 Release 22.4 or later in a Kubernetes-orchestrated environment.

IN THIS SECTION

- Overview | 88
- Configure TLS Encryption for Contrail Control Plane and vRouter | 89

Overview

IN THIS SECTION

- Considerations | 89

CN2 supports management and control plane authentication with TLS encryption. The TLS protocol is used for certificate exchange, mutual authentication, and negotiating ciphers to secure the stream from potential tampering and eavesdropping.

Certificates are part of the deployment used to bring up CN2. In Kubernetes all certificates are translated to secrets. By default, the certificate and secrets for the Contrail control plane and vRouter are automatically generated in the `ContrailCertificateManager` CRD. If desired, you can also create certificates for other components, such as Sandesh and the Contrail API server.

You can generate certificates using one of the following tools:

- cert-manager (default): The cert-manager tool adds certificates as resource types in Kubernetes clusters and simplifies the process of obtaining, renewing and using those certificates. By default, the certificate is valid for 10 years and automatically renews 15 days before its expiration date.
- go-crypto: go-crypto is a cryptographic package you can use to generate certificates. This package is a lightweight generator that does not use containers. By default, the certificate is valid for 10 years, but is not automatically renewed.

Considerations

Read through this list of considerations before you begin the configuration:

- When a certificate is renewed, you must restart the pod.
- You must enable TLS encryption for the Contrail control plane and vRouter, even if a certificate is provided.
- If you are creating your own certificate authority (CA), the secret must contain the keys `tls.crt` and `tls.key`.

Configure TLS Encryption for Contrail Control Plane and vRouter

Follow the steps in this procedure to easily configure TLS encryption for the Contrail control plane and vRouter.

> **NOTE**: By default, TLS encryption is enabled for XMPP and introspect for control and vRouter. TLS-based XMPP is used to secure all XMPP communication that occurs in the networking environment. If you prefer, you can disable TLS encryption by specifying the *false* variable under the spec field in your control and vRouter YAML files. For example:

```
xmppAuthEnable: false
  introspectSslEnable: false
```

To configure TLS encryption for the Contrail control plane and vRouter:

1. Specify the `generatortype`, either `cert-manager` or `go-crypto` in `ContrailCertificateManager`. The default generator is `cert-manager`.

   In Kubernetes all certificates are translated to secrets. When you generate the certificate, the secrets are automatically created when you apply your `deployment.yaml` file.

   The following examples show certificates created in `cert-manager` and `go-crypto` for control and vRouter:

   Example using cert-manager:

```
apiVersion: configplane.juniper.net/v1alpha1
kind: ContrailCertificateManager
metadata:
  name: contrail-certificate-manager
  namespace: contrail
spec:
  common:
    containers:
    - image: REGISTRY/contrail-k8s-cert-manager:TAG
      name: contrail-k8s-cm
    - name: cert-manager
      image: REGISTRY/cert-manager:TAG
    - name: cert-manager-webhook
      image:  REGISTRY/cert-manager-webhook:TAG
    - name: cert-manager-cainjector
      image:  REGISTRY/cert-manager-cainjector:TAG
  generatortype: cert-manager
  casecret:
    name: contrail-ca
    namespace: contrail
  secrets:
    - name: xmpp-tls-vrouter-agent
      namespace: contrail
      components:
      - vrouter-xmpp
```

```
      duration: 87600 [also default and value in hours]
      renewBefore: 360 [also default and value in hours]
```

Example using gocrypto:

```
apiVersion: configplane.juniper.net/v1alpha1
kind: ContrailCertificateManager
metadata:
  name: contrail-go-crypto
  namespace: contrail
spec:
  common:
    containers:
    - image: REGISTRY/contrail-k8s-gocrypto:TAG
      name: contrail-k8s-cm
  generatortype: gocrypto
  secrets:
    - name: xmpp-tls-vrouter-agent
      namespace: contrail
      components:
      - vrouter-xmpp
      duration: 87600 [also default and value in hours]
    - name: xmpp-tls-control
      namespace: contrail
      components:
      - control-xmpp
    - name: contrail-api-tls
      namespace: contrail
      dnsNames:
      - contrail-api.contrail-system.svc
      components:
      - control-sandesh
```

2. Apply your deployment.yaml file to generate the certificate.
3. Verify your configuration.

   Run the following `kubectl` commands to verify that the certificate and secrets were successfully generated.

```
All contrail secrets created by cert-generator in deployer are labeled with contrail=default
and certificate info NotAfter and NotBefore are added in annotation to view secret validity.
```

```
list secrets
kubectl get secret -l contrail=default -A

list secrets with more info
kubectl get secret -l contrail=default -A | tail +2 | awk '{print "kubectl describe  secret
"$2" -n "$1}' | while read line; do $line; done | grep -e "Name:" -e "Namespace:" -e
"NotAfter:" -e "NotBefore:"

Download certificate from secret: [install jq]
kubectl get secret -n contrail contrail-api-tls -o json | jq -r '.data."tls.key"' | base64 -d
> tls.key

View certificate:
openssl x509 -noout -text -in tls.crt
```

By default, certificates and secrets are automatically generated in `ContrailCertificateManager`. You can also create secrets for other components or create your own CA.

To create secrets for other components, specify the component(s) you want to use in either `cert-manager` or `gocrypto`. You can use one secret for multiple components.

The available components are:

- `control-xmpp`
- `vrouter-sandesh`
- `control-sandesh`
- `contrail-api-server`
- `vrouter-xmpp`

If you did not enter a `casecret` in `ContrailCertificateManager`, a self-signed certificate is automatically created. This self-signed certificate is valid for 10 years.

If desired, you can specify your own CA certificate as shown in the following example. The secret must contain the keys `tls.crt` and `tls.key`.

```
apiVersion: v1
kind: Secret
metadata:
  name: ca-key-pair
  namespace: contrail
data:
  tls.crt:
```

```
<output of cat cacert.pem | base64 -w0>
tls.key:
<output of cat cakey.pem | base64 -w0>
```

Run the `kubectl apply -f ca-key-pair.yaml` command to apply the secret.

SEE ALSO

Configuring Transport Layer Security Based XMPP in Contrail

4

CHAPTER

Advanced Virtual Networking

Enable BGP as a Service | 96

Create an Isolated Namespace | 109

Configure Allowed Address Pairs | 120

Enable Packet-Based Forwarding on Virtual Interfaces | 122

Configure Reverse Path Forwarding on Virtual Interfaces | 125

vRouter Interface Health Check | 127

Kubernetes Ingress Support | 134

Deploy VirtualNetworkRouter in Cloud-Native Contrail Networking | 138

Configure Inter-Virtual Network Routing Through Route Targets | 157

Configure IPAM for Pod Networking | 162

Enable VLAN Subinterface Support on Virtual Interfaces | 166

EVPN Networking Support | 173

Customize Virtual Networks for Pod Deployments, Services, and Namespaces | 177

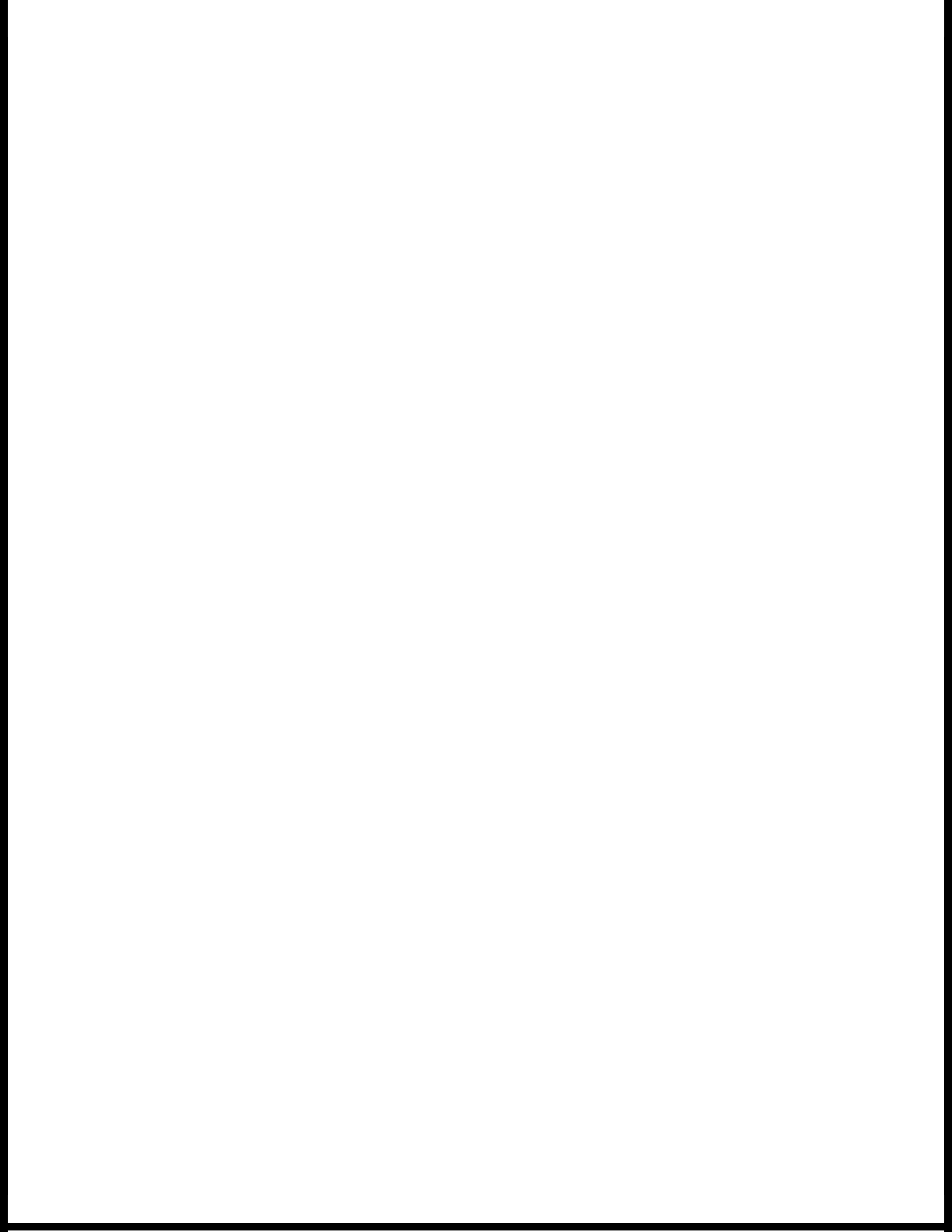
Deploy Kubevirt DPDK Dataplane Support for VMs | 185

Pull Kubevirt Images and Deploy Kubevirt Using a Local Registry | 197

Static Routes | 200

VPC to CN2 Communication in AWS EKS | 211

Configure a Service Account to Assume an IAM role | 220



Enable BGP as a Service

IN THIS SECTION

- Benefits of BGP as a Service in Cloud-Native Contrail | 96
- Prerequisites | 97
- Overview of BGP as a Service in Cloud-Native Contrail Networking | 97
- Enable BGPaaS in a Pod | 98
- Configure the IP Address Allocation Method for BGPasaS | 102
- Configure the BGPasaService Object | 103
- Validate the BGP as a Service Configuration | 107
- Configure BGP in Pod | 108

Cloud-Native Contrail® Networking™ supports BGP as a Service (BGPaaS). This document should be used to enable BGPaaS in environments using Release 22.1 or later.

The BGPaaS feature in Cloud-Native Contrail Networking provides the network support for BGP to operate within a virtual network in cloud networking environments using Cloud-Native Contrail Networking.

Benefits of BGP as a Service in Cloud-Native Contrail

With BGPaaS in Kubernetes environments using Cloud-Native Contrail Networking, you gain the following functionality:

- A BGP protocol service that runs in the virtual network. This BGP service creates BGP neighbor sessions to pods, virtual machines, and other workloads in the virtual network.
- A routing protocol that supports IPv4 neighbors, the IPv4 and IPv6 unicast address family, and IPv6-over-IPv4 next-hop mapping.
- A BGP protocol service that is user-configurable using most well-known BGP configuration parameters.

You can use BGPaaS in any cloud networking environment that needs the functionality provided by a routing protocol. You may find BGPaaS especially useful in the following scenarios:

- If you manage a large cloud networking environment that runs multiple workloads, you may want to use BGPaaS to scale network services.
- If you use tunneling protocols that need network reachability information from a routing protocol to create and maintain tunnels, BGPaaS can help.

Prerequisites

We assume that before you enable BGP as a service:

- You are operating in a working cloud networking environment using Kubernetes orchestration, and Cloud-Native Contrail Networking is operational.
- You have a working knowledge of BGP.

Overview of BGP as a Service in Cloud-Native Contrail Networking

Cloud-Native Contrail Networking provides the networking support for BGPaaS.

You have to find a BGP service to run BGP in your cloud networking environment. This document shows how to enable networking support for BGPaaS with Cloud-Native Contrail Networking using the BGP service provided by the BIRD Internet Routing Daemon (BIRD). This daemon is available as a built-in development tool on many versions of Unix. You can also download it to your environment using a separate image.

In the examples that follow, you see that the BGP daemon from BIRD runs in a pod when BGPaaS is enabled. That daemon then sends BGP messages over the network using the networking capabilities provided by Cloud-Native Contrail Networking. For additional information on BIRD, see the BIRD Internet Routing Daemon homepage.

When BGPaaS is operational, the BGP daemon runs in a pod and manages BGPaaS. The BGP daemon is directly connected to a Contrail vRouter.

The Contrail vRouter has a connection to at least one control plane node and connects the BIRD daemon to the control plane. A BGP peering session between at least one control node and the BIRD daemon is established through this connection with the Contrail vRouter.

After a peering session is created between the control nodes and the BGP daemon, the BGP daemon can manage BGPaaS and send routes to BGP clients over the control plane. The BGPaaS management tasks include assigning IP addresses to workloads, pods, VMs, or other objects.

Enable BGPaaS in a Pod

To enable BGPaaS, you must create a pod to host the BGP service. You must then associate the pod hosting the BGP service with the virtual networks where BGPaaS will run.

You can use either of two methods of associating a pod hosting the BGP service with a virtual network:

- Virtual Machine Interfaces Selector—The pod running the BGP service is directly associated with the virtual network. The pod hosting the BGP service is discovered automatically after the virtual network association is defined.
- Virtual Machine Interface References—The pod running the BGP service is directly associated with the virtual network by explicitly providing the namespace and the name of the virtual machine interface of the pod hosting the BGP service.

The following sections provide the steps for each association method.

Enable BGPaaS in a Pod Using the Virtual Machine Interfaces Selector

You must create a pod to host the BGP service, and then you can enable BGPaaS with the Virtual Machine Interfaces Selector.

The pod must:

- Include at least one IPv4 interface.
- Include annotations using `core.juniper.net/bgpaas-networks` to specify the associated virtual network names. The value in this annotation must include at least one virtual network name. If you are associating the pod hosting the BIRD daemon with multiple virtual networks, enter the virtual network names as a comma-separated list.

In this example YAML file, a pod is created to host the BGP service. The pod is associated with two virtual networks and BGPaaS is enabled to run on both virtual networks. The **image:** variable in the **containers:** hierarchy points to the BIRD image file that will provide the BGP service in this example.

```
apiVersion: v1
kind: Pod
metadata:
  name: bird-pod-shared-1
  namespace: bgpaas-ns
  annotations:
    k8s.v1.cni.cncf.io/networks: |
        [{
          "name": "bgpaas-vn-1",
```

```
          "namespace": "bgpaas-ns",
          "cni-args": null
          "interface": "eth1"
        },{
          "name": "bgpaas-vn-2",
          "namespace": "bgpaas-ns",
          "cni-args": null
          "interface": "eth2"
        }]
    core.juniper.net/bgpaas-networks: bgpaas-vn-1,bgapss-vn-2
spec:
  containers:
    - name: bird-pod-c
      image: somewhere.juniper.net/cn2/bazel-build/dev/bird-sut:1.0
      command: ["bash","-c","while true; do sleep 60s; done"]
      securityContext:
        privileged: true
```

Enter the **kubectl get vmi -n** ***virtual-network-name*** to confirm that the pod and its associated virtual machine interfaces have been created. You can also enter the **kubectl describe** command to ensure that the virtual machine interfaces exist.

You can confirm the virtual network was created by reviewing the **bgpaasVN=** output in the label section of the **kubectl describe** command.

```
kubectl get vmi -n bgpaas-ns
  CLUSTERNAME                          NAME                 NETWORK              PODNAME
IFCNAME   STATE     AGE
  contrail-k8s-kubemanager-kubernetes   bird-pod-1-abb881a8   bgpaas-vn-1       bird-pod-1
eth1      Success   13s
  contrail-k8s-kubemanager-kubernetes   bird-pod-1-e3f93f05   default-podnetwork   bird-pod-1
eth0      Success   13s

  kubectl describe vmi bird-pod-1-abb881a8 -n bgpaas-ns
  Name:         bird-pod-1-abb881a8
  Namespace:    bgpaas-ns
  Labels:       core.juniper.net/bgpaasVN=bgpaas-vn-1
                namespace=bgpaas-ns
```

You must then create a BGPaaS object to configure BGPaaS. The BGPaaS object references the virtual networks in the **virtualMachineInterfacesSelector:** section.

```
  apiVersion: core.contrail.juniper.net/v1alpha1
  kind: BGPAsAService
  metadata:
    namespace: bgpaas-ns
    name: bgpaas-test
  spec:
    shared: false
    autonomousSystem: 10
    bgpAsAServiceSessionAttributes:
      loopCount: 2
      routeOriginOverride:
      origin: EGP
      addressFamilies:
        family:
          - inet
          - inet6
    virtualMachineInterfacesSelector:
      - matchLabels:
          core.juniper.net/bgpaasVN: bgpaas-vn-1
      - matchLabels:
          core.juniper.net/bgpaasVN: bgpaas-vn-2
```

Enable BGPaaS in a Pod Using Virtual Machine Interface References

You must first create a pod to host the BIRD daemon to enable BGPaaS with Virtual Machine Interface references. The pod must include at least one IPv4 interface.

In the following example, a pod is created in the **bgpaas-ns** namespace. The annotation associates the pod with the **bgpaas-vn-1** virtual network. The **image:** variable in the **containers:** hierarchy points to the BIRD image file that will provide the BGP service in this example.

```
apiVersion: v1
kind: Pod
metadata:
  name: bird-pod-1
  namespace: bgpaas-ns
  annotations:
    k8s.v1.cni.cncf.io/networks: bgpaas-vn-1
```

```
spec:
  containers:
    - name: bird-pod-c
      image: somewhere.juniper.net/cn2/bazel-build/dev/bird-sut:1.0
      command: ["bash","-c","while true; do sleep 60s; done"]
      securityContext:
        privileged: true
```

Confirm that the pod was created after committing the pod object configuration file by entering the **kubectl get vmi -n bgpaas-ns** command.

Note the name of the virtual machine interface for the pod in this command output. You will need to specify the virtual machine interface name later in this procedure when configuring the BGPaaS object.

```
kubectl get vmi -n bgpaas-ns
  CLUSTERNAME                          NAME                NETWORK             PODNAME
IFCNAME   STATE     AGE
  contrail-k8s-kubemanager-kubernetes  bird-pod-1-abb881a8  bgpaas-vn-1         bird-pod-1
eth1      Success   13s
  contrail-k8s-kubemanager-kubernetes  bird-pod-1-e3f93f05  default-podnetwork  bird-pod-1
eth0      Success   13s
```

The Virtual Machine interface references are defined while creating the BGPaaS object using the **virtualMachineInterfaceReferences:** hierarchy. The **namespace:** is the pod namespace and the **name:** is the virtual machine interface name that you retrieved using the **kubectl get vmi -n bgpaas-ns** command.

```
apiVersion: core.contrail.juniper.net/v1alpha1
kind: BGPAsAService
metadata:
  namespace: bgpaas-ns
  name: bgpaas-sample
spec:
  shared: false
  autonomousSystem: 100
  bgpAsAServiceSessionAttributes:
    localAutonomousSystem: 10
    loopCount: 2
    routeOriginOverride:
      origin: EGP
    addressFamilies:
      family:
        - inet
```

```
      - inet6
  virtualMachineInterfaceReferences:
    - apiVersion: core.contrail.juniper.net/v1alpha1
      kind: VirtualMachineInterface
      namespace: bgpaas-ns
      name: bird-pod-1-abb881a8
```

Configure the IP Address Allocation Method for BGPasaS

You can configure BGPaaService with one of the following IP address allocation methods:

- automatic IP address allocation—The BGP service assigns IP addresses.
- user-specified IP address allocation—You assign the IP address.

You configure the IP address allocation method in the `Subnet` object.

Automatic IP address allocation is enabled by default. If you enable BGPaaS without manually disabling automatic IP address allocation, BGPaaS uses automatic IP address allocation.

You disable automatic IP address allocation by setting the *disableBGPaaSIPAutoAllocation:* variable in the `Subnet` object to *true*. If the *disableBGPaaSIPAutoAllocation:* variable is not present in the **Subnet** object file, automatic IP address allocation is enabled.

In the following configuration sample, automatic IP address allocation is enabled because the *disableBGPaaSIPAutoAllocation:* variable isn't present in the **Subnet** object configuration file.

```
apiVersion: core.contrail.juniper.net/v1alpha1
kind: Subnet
metadata:
  namespace: bgpaas-ns
  name: bgpaas-subnet-1
spec:
  cidr: "172.20.10.0/24"
```

In this configuration sample, automatic IP address allocation is enabled because the *disableBGPaaSIPAutoAllocation:* variable is set to **false**.

```
apiVersion: core.contrail.juniper.net/v1alpha1
kind: Subnet
metadata:
```

```
  namespace: bgpaas-ns
  name: bgpaas-subnet-2
spec:
  cidr: "172.20.20.0/24"
 disableBGPaaSIPAutoAllocation: false
```

To enable user-specified IP address allocation, set the *disableBGPaaSIPAutoAllocation:* variable to `true`. When user-specified IP address allocation is enabled, you must also configure the BGP addresses that BGPaaS can assign to endpoints within the subnet. You must set a primary IP address using the *bgpaasPrimaryIP:* variable. You can also set an optional secondary IP address, which you can see in this example with the *bgpaasSecondaryIP:* variable.

```
apiVersion: core.contrail.juniper.net/v1alpha1
kind: Subnet
metadata:
  namespace: bgpaas-ns
  name: bgpaas-subnet-2
spec:
  cidr: "172.20.20.0/24"
 disableBGPaaSIPAutoAllocation: true
  bgpaasPrimaryIP: 172.20.20.3
  bgpaasSecondaryIP: 172.20.20.4
```

Configure the BGPasaService Object

You enable BGPaaS in a cluster by creating a *BGPAsAService* object.

Create the *BGPAsAService* object by creating a YAML file that uses `BGPAsAService` in the **kind:** field:

```
  apiVersion: core.contrail.juniper.net/v1alpha1
  kind: BGPAsAService
  metadata:
    namespace: bgpaas-ns
    name: bgpaas-test
  spec:
    shared: false
    autonomousSystem: 10
    bgpAsAServiceSessionAttributes:
      loopCount: 2
```

```
    routeOriginOverride:
    origin: EGP
    addressFamilies:
      family:
        - inet
        - inet6
  virtualMachineInterfacesSelector:
    - matchLabels:
        core.juniper.net/bgpaasVN: bgpaas-vn-1
    - matchLabels:
        core.juniper.net/bgpaasVN: bgpaas-vn-2
```

Table 11: Spec Field Variables for BGPaaS

This table provides a description of each Spec field variable in the BGPaaS object file.

Field Variable	Description
shared:	Specifies whether a common BGP router object can be shared with multiple virtual machine interfaces in the same virtual network. When this field is set to **true**, one BGP client router can be shared with multiple virtual machine interfaces in the same virtual network. This setting limits the total number of BGP client routers that have to be created in a virtual network for a VMI. When this field is set to **false**, one BGP client router is created for each virtual machine interface.
autonomousSystem:	Specifies the global Autonomous System number for the BGP instance. The Autonomous System number can be any whole number between 1 and 4294967295.
bgpAsAServiceSessionAttributes:	Defines the BGP session attributes for BGPaaS. See Table 12 on page 105.

Table 11: Spec Field Variables for BGPaaS *(Continued)*

Field Variable	Description
virtualMachineInterfaceReferences:	Defines the virtual machine interface parameters to associate with BGPaaS when using virtual machine interface references. See Table 13 on page 106.
virtualMachineInterfacesSelector:	Defines the virtual networks where BGPaaS runs when using the virtual machine interfaces selector. See Table 14 on page 106.

Table 12: BgpAsAServiceSessionAttributes Fields for BGPaaS

The `bgpAsAServiceSessionAttributes:` in the `spec:` hierarchy are used in all BGPaaS setups. The `bgpAsAServiceSessionAttributes:` hierarchy includes these fields:

Field	Description
localAutonomousSystem:	Specifies the local Autonomous System number for BGP.
LoopCount:	Specifies the number of times that the same ASN can be seen in a route update before the route is discarded. The **LoopCount:** can be any whole number up to 16.
routeOriginOverride:	Overrides the original setting and sets the origin attribute to **Incomplete** when forwarding routes. If you set this field to **false**, routes are advertised into BGP based on the origin setting. The origin is either IGP or EGP and is set using the **origin:** field in this file. If you set this field to **true**, the origin is set to **Incomplete** for advertised routes. Use this field if you want to change how BGP networks prioritize routes received from the BGP service. By default, BGP networks prioritize routes based on origin, and routes with an **Incomplete** origin receive lower priority than routes received from IGP or EGP.

Table 12: BgpAsAServiceSessionAttributes Fields for BGPaaS *(Continued)*

Field	Description
Origin:	Specifies if BGP operates as an interior gateway protocol (**igp**) or exterior gateway protocol (**egp**). The default route origin is **igp**.
AddressFamilies:	Specifies the address family. You can specify the family as **inet** for IPv4 or **inet6** for IPv6. You can specify both address families simultaneously.

Table 13: virtualMachineInterfaceReferences: in BGPaaS

The `virtualMachineInterfaceReferences:` in the `spec:` hierarchy include the following fields:

Field	Description
apiVersion:	Specifies the API version for the virtual machine interface reference.
kind:	Always set this field to **VirtualMachineInterface**.
namespace:	Specifies the namespace associated with the virtual machine interface reference. You define this namespace while creating the **Pod** object. See "Enable BGPaaS in a Pod Using Virtual Machine Interface References" on page 100.
name:	Specifies the name of the pod associated with the virtual machine interface reference. You can retrieve the pod name by entering the **kubectl get vmi -n bgpaas-ns** command. See "Enable BGPaaS in a Pod Using Virtual Machine Interface References" on page 100.

Table 14: The virtualMachineInterfacesSelector: Fields in BGPaaS

The `virtualMachineInterfacesSelector:` in the `spec:` hierarchy includes the following fields:

Field	Description
matchLabels:	Define the match labels for the Virtual Machine Interfaces selector. The match labels in this context are always used to reference the virtual networks where the Virtual Machine interfaces selector is running. Always enter the match label values in this section as **core.juniper.net/bgpaasVN:***virtual-network-name*. See "Enable BGPaaS in a Pod Using the Virtual Machine Interfaces Selector" on page 98.

Validate the BGP as a Service Configuration

You should confirm that the BGPaaS object is successfully running after you commit the BGPAsAService object file.

Enter the **kubectl get BGPAsAService** command after you create the BGPAsAService object to confirm the object state. The object is successfully created when the **State** field indicates **Success**.

```
kubectl get BGPAsAService -n bgpaas-ns
    NAME            AS     IPADDRESS    SHARED    STATE      AGE
    bgpaas-sample   100                 false     Success    33s
```

You should also ensure the BGPaaS server and the BGPaaS client are created and are in the **Success** state.

Enter the **kubectl get BGPRouter** command to confirm the presence and operational state of the BGPaaS servers and clients.

```
kubectl get BGPRouter -n bgpaas-ns
    NAME                                     TYPE            IDENTIFIER    STATE      AGE
    bgpaas-ns-bgpaas-vn-1-bgpaas-server      bgpaas-server                 Success    2m57s
    bgpaas-ns-bgpaas-vn-1-bird-pod-1-abb881a8   bgpaas-client   172.20.10.2   Success    2m57s
```

Configure BGP in Pod

You must also configure the networking parameters for the BGP service running in the pod. The configuration for each individual BGP service is unique. Documenting the required networking configuration parameters is beyond the scope of this document. Please check the documentation that accompanies your BGP service.

In this example, we show you how to configure the BGP network configuration using BIRD.

You configure BGP using the BIRD CLI in this example. The parameters of the BGP configuration that need to match the BGPaaS objects defined in Cloud-Native Contrail Networking are noted. Although not shown in this example, you should know that the default location to access the BIRD configuration file in most deployments is /etc/bird.conf or /etc/bird/bird.conf.

```
  # Change the router id to your BIRD router ID. It's a world-wide unique identification
  # of your router, usually one of router's IPv4 addresses.
  router id 172.20.10.2;

  protocol direct {
        interface "eth1*";  -> interface on which BGPAsAService needs to be configured
  }

  protocol bgp bgp1_1 {
        import all;
        export all;
        local as 10;                     -> AS configured in BGPAsAService
        neighbor 172.20.10.3 as 64512;   -> neighbor for primary BGP session, use
BGPaaSPrimaryIP from subnet
        neighbor 172.20.10.3 as 64512;    -> neighbor for secondary BGP session, use
BGPaaSSecondaryIP from subnet
```

From your BGP service, verify that the BGP protocol is running.

In this example from BIRD, you enter the **show protocol** command to verify that the BGP protocol is established in BIRD.

```
birdc show protocol bgp1_1
   BIRD 1.6.8 ready.
   name     proto    table    state  since       info
   bgp1_1   BGP      master   up     10:31:27    Established
```

Create an Isolated Namespace

SUMMARY

This topic describes how to create an isolated namespace in CN2 Release 22.1 or later in a Kubernetes-orchestrated environment.

IN THIS SECTION

- Namespace Overview | 109
- Example: Isolated Namespace Configuration | 110
- Isolated Namespace Objects | 113
- Create an Isolated Namespace | 114
- Optional Configuration: IP Fabric Forwarding and Fabric Source NAT | 116
- Enable IP Fabric Forwarding | 116
- Enable Fabric Source NAT | 118

Namespace Overview

NOTE: In this document, we use the term "isolated" and "non-isolated" in the context of Contrail networking only.

Non-isolated Namespaces.

Namespaces, or non-isolated namespaces, provide a mechanism for isolating a group of resources within a single cluster. By default, namespaces are not isolated.

Non-isolated namespaces are intended for use in environments with many users spread across multiple teams, or projects. Non-isolated namespaces enable each team to exist in their own virtual cluster without impacting each other's work. Let's say you created all your resources in the default namespace that Kubernetes provides. If you have a complex application with multiple deployments, the default namespace can be hard to maintain. An easier way to manage this deployment is to group all your resources into different namespaces within the cluster. For example, the cluster can contain separate namespaces, such as a database namespace or a monitoring database. Names of resources must be unique within a namespace, but not across namespaces.

Pods in a non-isolated namespace exhibit the following network behavior:

- Pods in non-isolated namespaces can communicate with other pods in the cluster without using NAT.
- Pods and services in non-isolated namespaces share the same `default-podnetwork` and `default-servicenetwork`.

Isolated Namespaces.

An isolated namespace enables you to run customer-specific applications that you want to keep private. You can create an isolated namespace to isolate a pod from other pods, without explicitly configuring a network policy.

Isolated namespaces are similar to non-isolated namespaces, except that each isolated namespace has its own pod network and service network. This means that pods in isolated namespaces cannot reach pods or services in other isolated or non isolated namespaces.

Pods in isolated namespace can only communicate with pods in the same namespace. The only exception is when a pod in an isolated namespace needs access to a Kubernetes service, such as Core DNS. In this case, the pod uses the cluster's `default-servicenetwork` to access the services.

Pods in an isolated namespace exhibit the following network behavior:

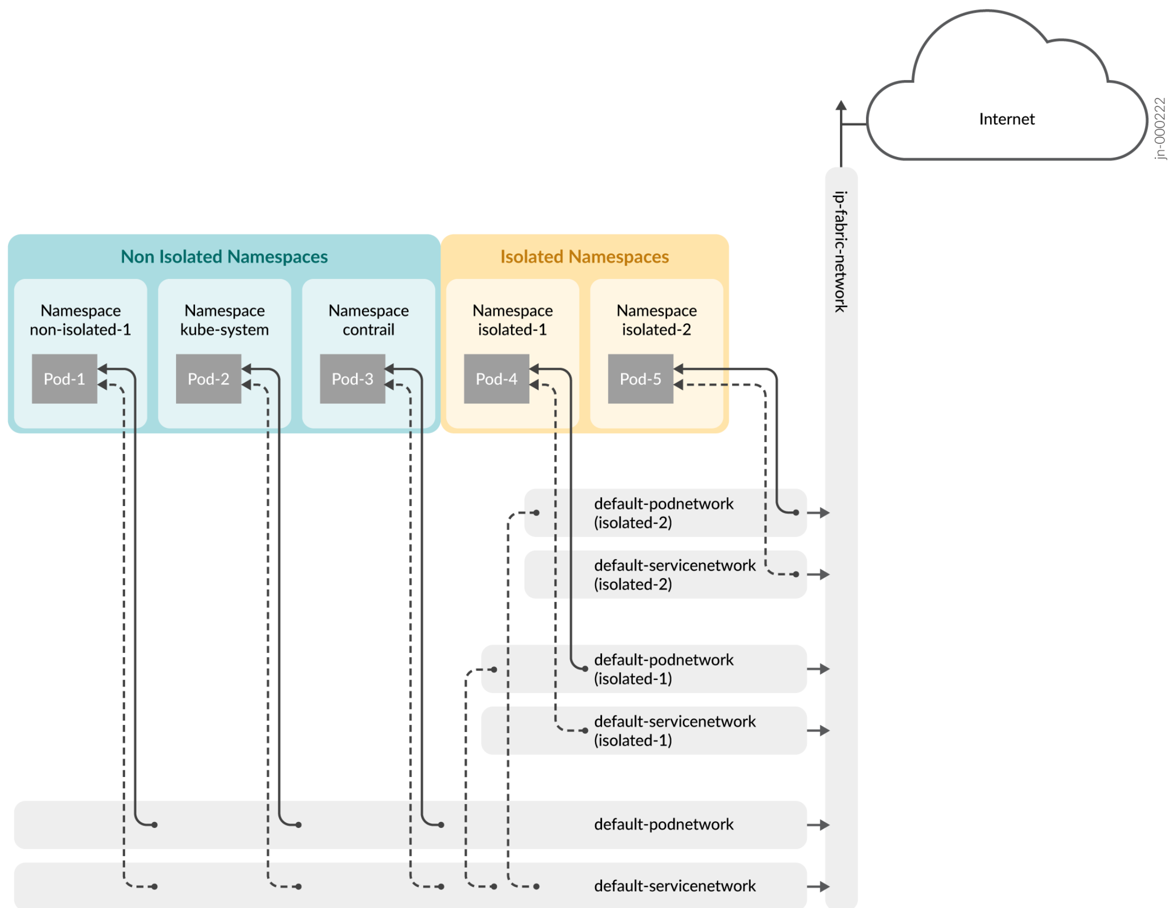
- Pods in isolated namespaces can only communicate with pods in the same namespace.
- Pods in isolated namespaces can reach services in non-isolated namespaces.
- The IP addresses and service IP addresses of pods in isolated namespaces are allocated from the same subnet as the cluster's pod and service subnet.
- Pods in an isolated namespace can access the underlay network, or IP fabric network, though IP fabric forwarding and fabric source NAT.

NOTE: You cannot covert a non-isolated namespace to an isolated namespace, and vice versa.

Example: Isolated Namespace Configuration

This sample configuration demonstrates an isolated namespace configuration in CN2.

Figure 8: Isolated Namespace Configuration



In the isolated namespace configuration:

- Pod-1 (non-isolated-1) is in a non-isolated namespace created by the user.
- Pod-2 (kube-system) and Pod-3 (contrail) are in non-isolated namespaces created by the controller.
- Pod-4 (isolated-1) and Pod-5 (isolated-2) are in isolated namespaces created by the user.
- The interfaces for Pod-1, Pod-2, Pod-3 are created from the cluster's `default-podnetwork` and `default-servicenetwork`.
- The interfaces for Pod-4 and Pod-5 are created on the `default-podnetwork` and `default-servicenetwork` in their own isolated namespaces. Both Pod-4 and Pod-5 interfaces share the same subnet as the cluster's `default-podnetwork` and `default-servicenetwork`.

- Pods in isolated namespaces cannot communicate with pods in non-isolated namespaces. In this example, Pod-4 and Pod-5 in isolated namespaces cannot communicate with Pod-1, Pod-2, Pod-3 in non-isolated namespaces.
- Pods in isolated namespaces (Pod-4, Pod 5) can access any service through the cluster's `default-servicenetwork`.
- Pods in all namespaces (non-isolated and isolated) are able to connect to the fabric through the cluster's `ip-fabric-network`.

Notes

- Isolated namespaces affect only the pod's default interface. This is because the default interface of pods in an isolated namespace are created on the `default-podnetwork` of the isolated namespace. However, interfaces from user-defined `VirtualNetworks` behave the same way in both isolated and non-isolated namespaces.
- You can create network policies on isolated namespaces to adjust the isolation of pods. The network policy behaves the same for both isolated and non-isolated namespaces.
- Two or more isolated namespaces can be interconnected through the `VirtualNetworkRouter` (VNR). See "VirtualNetworkRouter Overview" on page 139.

Here is an example of a VNR configuration used to interconnect the `default-podnetworks` of two isolated namespaces (`ns-isolated-1` and `ns-isolated-2`). In this configuration, the `VirtualNetworkRouter` connects to ns-isolated-1 and ns-isolated-2. This means that pods in these isolated namespaces can communicate with each other.

```
apiVersion: core.contrail.juniper.net/v3
kind: VirtualNetworkRouter
metadata:
  namespace: ns-isolated-1
  name: vnr-1
  annotations:
    core.juniper.net/display-name: vnr-1
  labels:
      vnr: vnr-1
spec:
  type: mesh
  virtualNetworkSelector:
    matchExpressions:
    - key: core.juniper.net/virtualnetwork
      operator: In
      values:
```

```
      - isolated-namespace-pod-virtualnetwork
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: vnr-2
        namespaceSelector:
          matchLabels:
            kubernetes.io/metadata.name: ns-isolated-2
---
apiVersion: core.contrail.juniper.net/v3
kind: VirtualNetworkRouter
metadata:
  namespace: ns-isolated-2
  name: vnr-2
  annotations:
    core.juniper.net/display-name: vnr-2
  labels:
      vnr: vnr-2
spec:
  type: mesh
  virtualNetworkSelector:
    matchExpressions:
    - key: core.juniper.net/virtualnetwork
      operator: In
      values:
      - isolated-namespace-pod-virtualnetwork
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: vnr-1
        namespaceSelector:
          matchLabels:
            kubernetes.io/metadata.name: ns-isolated-1
```

Isolated Namespace Objects

This table describes the namespace objects (API resources) the controller creates when you create an isolated namespace.

Table 15: Isolated Namespace Objects

Isolated Namespace Object	Description
`default-podnetwork (VirtualNetwork)`	The default interfaces of pods in an isolated namespace are created in this `default-podnetwork`, instead of the cluster's default-network.
`default-servicenetwork (VirtualNetwork)`	The cluster IP of services in isolated namespaces are created in this `default-servicenetwork`, instead of the cluster's `default-servicenetwork`.
`IsolatedNamespacePodServiceNetwork (VirtualNetworkRouter)`	This object establishes connectivity between the isolated namespace's `default-podnetwork` and `default-servicenetwork`.
`IsolatedNamespaceIPFabricNetwork (VirtualNetworkRouter)`	This object establishes connectivity between the isolated namespace's `default-podnetwork` and `default-servicenetwork`to the cluster's `ip-fabricnetwork`.
`IsolatedNamespacePodToDefaultService (VirtualNetworkRouter)`	This object establishes connectivity between the isolated namespace's `default-podnetwork` to the cluster's `default-servicenetwork`.

Create an Isolated Namespace

Follow these steps to create an isolated namespace:

1. Create a YAML file called `ns-isolated.yaml`.
2. Add the label `core.juniper.net/isolated-namespace` to the namespace metadata and set the variable to `"true"`.

```
apiVersion: v1
kind: Namespace
metadata:
  name: ns-isolated
  labels:
    core.juniper.net/isolated-namespace: "true"
```

3. Issue the `kubectl apply` command to apply the configuration.

```
kubectl apply -f ns.yaml
```

4. To verify your configuration, issue the `kubectl get ns ns-isolated -o yaml` command.

```
apiVersion: v1
kind: Namespace
metadata: `\
  annotations:
    core.juniper.net/forwarding-mode: "false"
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Namespace","metadata":{"annotations":{"core.juniper.net/
forwarding-mode":"false"},"labels":{"core.juniper.net/isolated-namespace":"true"},"name":"ns-
isolated"}}
  creationTimestamp: "2021-10-04T21:47:40Z"
  finalizers:
  - finalizers.core.juniper.net/isns-virtualnetworks-delete
  - finalizers.core.juniper.net/isns-virtualnetworkrouters-delete
  labels:
    core.juniper.net/isolated-namespace: "true"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    …
    …
    …
  name: ns-isolated
  resourceVersion: "4183"
  uid: d25d2b71-2051-4ac5-a738-e9b344235818
spec:
  finalizers:
  - kubernetes
status:
  phase: Active
```

Success! You created an isolated namespace.

Optional Configuration: IP Fabric Forwarding and Fabric Source NAT

Optionally, you can enable IP fabric forwarding and fabric source NAT on an isolated namespace.

IP fabric forwarding enables virtual networks to be created as part of the underlay network and eliminates the need for encapsulation and de-encapsulation of data. Fabric source NAT allows pods in the overlay to reach the Internet without floating IPs or a logical system.

When you create an isolated namespace, two virtual networks are created, a `default-podnetwork` and a `default-servicenetwork`. By default, IP fabric forwarding and fabric source NAT in these two virtual networks are disabled. You enable IP fabric forwarding or fabric source NAT in the virtual networks by adding “forwarding-mode” annotations for each feature in your isolated namespace YAML file.

Here is an example of the `default-podnetwork` for an isolated namespace with forwarding-mode set to `fabricSNAT`.

```
apiVersion: core.contrail.juniper.net/v3
kind: VirtualNetwork
metadata:
  annotations:
    core.juniper.net/description: Default Pod Network for IsolatedNamespace (ns-isolated)
    core.juniper.net/display-name: default-podnetwork
    ...
spec:
    ...
    fabricSNAT: true
    ...
```

Enable IP Fabric Forwarding

Follow these steps to enable IP fabric forwarding on an isolated namespace:

1. Add the annotation `core.juniper.net/forwarding-mode: "ip-fabric"` to the namespace metadata.
2. Set the label for the isolated namespace to `"true"`.

```
apiVersion: v1
kind: Namespace
metadata:
 name: ns-isolated
```

```
 annotations:
    core.juniper.net/forwarding-mode: "ip-fabric"
    labels:
      "core.juniper.net/isolated-namespace": "true"
```

3. Issue the `kubectl apply` command to enable IP fabric forwarding.

```
kubectl apply -f ns-isolated.yaml
```

4. Verify your configuration.

```
get vn -n ns-isolated default-podnetwork -o yaml
```

```
spec:
  fabricForwarding: true
  fabricSNAT: false
  fqName:
  - default-domain
  - ns-isolated
  - default-podnetwork
  providerNetworkReference:
    apiVersion: core.contrail.juniper.net/v3
    fqName:
    - default-domain
    - contrail
    - ip-fabric
    kind: VirtualNetwork
    name: ip-fabric
    namespace: contrail
    resourceVersion: "5629"
    uid: bdb0ae55-d5e5-49b2-803d-d93eea206df0
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v3
    fqName:
    - default-domain
    - contrail-k8s-kubemanager-mycluster-contrail
    - default-podnetwork-pod-v4-subnet
    kind: Subnet
    name: default-podnetwork-pod-v4-subnet
    namespace: contrail-k8s-kubemanager-mycluster-contrail
```

```
    resourceVersion: "4999"
    uid: fc9b9471-3b3e-4a57-80ac-5b9ed806fe94
  virtualNetworkProperties:
    forwardingMode: l3
    rpf: enable
status:
  observation: ""
  state: Success
  virtualNetworkNetworkId: 5
```

Success! You enabled IP fabric forwarding on the isolated namespace.

Enable Fabric Source NAT

NOTE: You can only enable fabric source NAT on the `default-podnetwork`.

Follow these steps to enable fabric source NAT on an isolated namespace:

1. Add the annotation `core.juniper.net/forwarding-mode: "fabric-snat"` to the namespace metadata.
2. Set the label for the isolated namespace to `"true"`.

```
apiVersion: v1
kind: Namespace
metadata:
 name: ns-isolated
 annotations:
    core.juniper.net/forwarding-mode: "fabric-snat"
    labels:
      "core.juniper.net/isolated-namespace": "true"
```

3. Issue the `kubectl apply` command to enable fabric source NAT.

```
kubectl apply -f ns-isolated.yaml
```

4. Verify your configuration.

```
kubectl get vn -n <isolated-namespace-name> default-podnetwork
```

Success! You enabled fabric source NAT on the isolated namespace.

```
spec:
  fabricSNAT: true
  fqName:
  - default-domain
  - ns-isolated-snat
  - default-podnetwork
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v3
    fqName:
    - default-domain
    - contrail-k8s-kubemanager-mycluster-contrail
    - default-podnetwork-pod-v4-subnet
    kind: Subnet
    name: default-podnetwork-pod-v4-subnet
    namespace: contrail-k8s-kubemanager-mycluster-contrail
    resourceVersion: "4999"
    uid: fc9b9471-3b3e-4a57-80ac-5b9ed806fe94
  virtualNetworkProperties:
    forwardingMode: l3
    rpf: enable
status:
  observation: ""
  state: Success
  virtualNetworkNetworkId: 7
```

SEE ALSO

Enable IP Fabric Forwarding and Fabric Source NAT | 2

Configure Allowed Address Pairs

Starting in CN2 Release 22.1 or later, Juniper Networks supports allowed address pairs (AAPs). Allowed address pairs enables you to add IP/MAC (CIDR) addresses to the guest interface (`VirtualMachineInterface`) by using a secondary IP address.

When you create a pod in a cluster, each pod automatically obtains its IP address from the virtual machine (VM) interface. If your pod is *not* on the same virtual network, you can add an AAP to allow traffic to flow through the port regardless of the subnet. For example, let's say that your pod's IP address is 192.168.2.0. If you define an AAP with subnet 192.168.2.0/24, the AAP allows the pods to communicate with the guest interface. The vRouter then forwards the traffic and advertises reachability to the pod.

To configure an allowed address pair, insert the following attributes into your pod YAML file. For example:

```
kind: Pod
metadata:
  name: my-pod
  namespace: my-namespace
  annotations:
    k8s.v1.cni.cncf.op/networks: |
      [
        {
          "name": "net-a",
          "cni-args": {
              "net.juniper.contrail.allowedAddressPairs": [{
                "ip": 192.168.2.0/24
                "mac": "02:3f:66:ad:00:e9",
                "addressMode": "active-active"
              }],}
          ...
        },
        {
          "name": "net-b",
          ...
        },
```

The `AllowedAddressPairs` attribute contains a list of allowed address pair definitions, as described in the following table:

Table 16: Allowed Address Pair Definitions

Definition	Description
ip	Specify the external pod IP address through which you want to allow traffic to pass.
mac	(Optional) Specify the MAC address of the external pod.
addressMode	(Optional) Specify a high availability (HA) address mode. Choose from *active/active* or *active/standby*. Active/standby is the default. The addressMode default value is an empty string. *Active/standby* is used for VRRP addresses. *Active/active* is used for ECMP.

In Kubemanager, the `PodController` watches for Pod events and reads the interface definitions for each new AAP. The controller then generates an `AllowedAddressPair` and adds it to the list of interfaces in the `VirtualMachineInterface`.

Alternative Configuration

Alternatively, you can configure AAP interfaces directly from the `VirtualMachineInterface`. For example:

```
kubectl patch --namespace project-kubemanager VirtualMachineInterface $VMINAME -p "$(cat ./
aap.yaml)"
```

The preceding command updates the existing `VirtualMachineInterface` with the AAP configuration, as follows:

```
spec:
  allowedAddressPairs:
    allowedAddressPair:
      - ip:
          ipPrefix: 192.0.2.0
          ipPrefixLen: 24
```

Enable Packet-Based Forwarding on Virtual Interfaces

IN THIS SECTION

- Overview | 122
- Configure Packet Mode on a Virtual Interface | 122

Juniper Networks supports packet-based forwarding on virtual interfaces using CN2 Release 22.1 or later in a Kubernetes-orchestrated environment.

Overview

By default, Contrail compute nodes use flow mode for packet forwarding on a virtual interface. This means that every vRouter has a flow table to keep track of all flows that pass through it. In flow mode, the virtual interface processes all traffic by analyzing the state or session of traffic. However, in some instances you might want to switch from flow mode to packet mode. For example, you might want to achieve higher traffic-forwarding performance or get around certain limitations of flow mode.

In packet mode, the virtual interface processes the traffic on a per-packet basis and ignores all flow information. The main advantage of packet mode is that the processing type is stateless. Stateless mode means that the virtual interface does not keep track of session information or goes through traffic analysis to determine how a session is established.

NOTE: Features that require a network policy (such as ACLs, security groups, and floating IPs) are unable to work in packet mode.

Configure Packet Mode on a Virtual Interface

To configure packet mode on a virtual interface:

1. Verify that you are running flow mode. For example:

Generate some traffic by pinging another pod in the same network. In this example, the pod's IP address is 25.26.27.2.

```
root@pod-vn-1:/# ping -q -c5 25.26.27.2
PING 25.26.27.2 (25.26.27.2) 56(84) bytes of data.

--- 25.26.27.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4057ms
rtt min/avg/max/mdev = 0.059/1.721/7.620/2.955 ms
```

2. Use the flow command-line tool to check for flows. The following example indicates that the `virtualMachineInterface` is in flow mode.

```
root@minikube:/# flow -l --match 25.26.27.3
...
...
Listing flows matching ([25.26.27.3]:*)

    Index                Source:Port/Destination:Port                      Proto(V)
-----------------------------------------------------------------------------------
   159692<=>400664       25.26.27.2:28                                       1 (3)
                         25.26.27.3:0
(Gen: 1, K(nh):39, Action:F, Flags:, QOS:-1, S(nh):39,  Stats:5/490,  SPort 64222,
 TTL 0, UnderlayEcmpIdx:0, Sinfo 7.0.0.0)

   400664<=>159692       25.26.27.3:28                                       1 (3)
                         25.26.27.2:0
(Gen: 1, K(nh):33, Action:F, Flags:, QOS:-1, S(nh):33,  Stats:5/490,  SPort 56567,
 TTL 0, UnderlayEcmpIdx:0, Sinfo 5.0.0.0)
```

3. Enable packet mode on the `virtualMachineInterface`.

   Create a patch file named `packet-mode-patch.yaml`, and set the VMI policy to `true`.

```
spec:
  virtualMachineInterfaceDisablePolicy:true
```

4. Apply the patch.

```
[user@machine:~]$ kubectl -n vmi-disablepolicy patch vmi pod-vn-1-7d622c4d --patch "$(cat
packet-mode-patch.yaml)"
virtualmachineinterface.core.contrail.juniper.net/pod-vn-1-7d622c4d patched
```

After you apply the patch flow mode switches to packet mode, as shown in the following example:

```
[user@machine:~]$ kubectl -n vmi-disablepolicy get vmi pod-vn-1-7d622c4d -oyaml |
yq .spec.virtualMachineInterfaceDisablePolicy
true
```

5. Verify that packet mode is active.

Generate traffic by pinging another pod in the same network that you pinged in Step 1.

```
root@pod-vn-1:/# ping -q -c5 25.26.27.2
PING 25.26.27.2 (25.26.27.2) 56(84) bytes of data.

--- 25.26.27.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4105ms
rtt min/avg/max/mdev = 0.051/2.725/13.388/5.331 ms
```

6. Use the flow command-line tool to check for flows.

```
root@minikube:/# flow -l --match 25.26.27.3
...
...
Listing flows matching ([25.26.27.3]:*)

    Index                Source:Port/Destination:Port                      Proto(V)
```

Success! No flows exist, which indicates that you are in packet mode.

Configure Reverse Path Forwarding on Virtual Interfaces

IN THIS SECTION

- Overview | 125
- Enable RPF on a Virtual Interface | 126

Starting in CN2 Release 22.1 or later, Juniper Networks supports reverse path forwarding (RPF) on virtual interfaces in a Kubernetes-orchestrated environment.

Overview

Unicast reverse-path-forwarding (RPF) verifies that a packet is sent from a valid source address by performing an RPF check. RPF check is a validation tool that uses the IP routing table to verify whether the source IP address of an incoming packet is arriving from a valid path. RPF helps reduce forwarding of IP packets that might be spoofing an IP address.

When a packet arrives on an interface, RPF performs a forwarding table lookup on the packet's source IP address and checks the incoming interface. The incoming interface must match the interface on which the packet arrived. If the interface doesn't match, the vRouter drops the packet. If the packet is from a valid path, the vRouter forwards the packet to the destination address.

You can enable or disable source RPF on a per-virtual network basis. By default, RPF is disabled.

- RPF enable: Whenever a packet reaches the interface, RPF performs a check on the packet's source IP address. All packets are dropped if the route is not learned by the vRouter. Only packets received from the MAC/IP address allocated to the workload are permitted on an interface.
- RPF disable: Packets from any source are accepted on the interface. A forwarding table lookup is not performed on the incoming packet source IP address.

Enable RPF on a Virtual Interface

To enable RPF on a virtual interface, set the `rpf` variable under `virtualNetworkProperties` to `enable`. For example:

```
apiVersion: v1
kind: Namespace
metadata:
  name: rpf-ns
---
apiVersion: core.contrail.juniper.net/v3
kind: Subnet
metadata:
  namespace: rpf-ns
  name: rpf-subnet-1
  annotations:
    core.juniper.net/display-name: Sample Subnet
    core.juniper.net/description:
      Subnet represents a block of IP addresses and its configuration.
      IPAM allocates and releases IP address from that block on demand.
      It can be used by different VirtualNetwork in the mean time.
spec:
  cidr: "172.20.10.0/24"
---
apiVersion: core.contrail.juniper.net/v3
kind: VirtualNetwork
metadata:
  namespace: rpf-ns
  name: rpf-vn-1
  annotations:
    core.juniper.net/display-name: Sample Virtual Network
    core.juniper.net/description:
      VirtualNetwork is a collection of end points (interface or ip(s) or MAC(s))
      that can communicate with each other by default. It is a collection of
      subnets whose default gateways are connected by an implicit router
spec:
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v3
    kind: Subnet
    namespace: rpf-ns
    name: rpf-subnet-1
```

```
fabricSNAT: true
virtualNetworkProperties:
  rpf: enable
```

RELATED DOCUMENTATION

Understanding How Unicast Reverse Path Forwarding Prevents Spoofed IP Packet Forwarding

vRouter Interface Health Check

SUMMARY

In Juniper® Cloud-Native Contrail Networking (CN2) Release 22.3, a new health-check custom resource object is introduced that associates the virtual machine interface (VMI) to the pod creation and update workflow. The health-check resource is a namespace-scoped resource.

IN THIS SECTION

- vRouter Interface Health Check Overview | 127
- Create a Health-Check Object | 128
- Health-Check Process | 133

vRouter Interface Health Check Overview

The Contrail vRouter agent provides the health-check functionality. You can associate a ping or HTTP health check to an interface. If the health check fails, the interface is set as administratively down and associated routes are withdrawn. Those settings are based on the timers and intervals configured in the health-check object. Health check traffic continues to be transmitted in an administratively down state to allow for an interface to recover.

Create a Health-Check Object

NOTE: These two attributes (`targetIpList` and `targetIpAll`) related to VMI health check are not supported in CN2 Release 22.3. These two attributes will be supported in a future release.

To create a health-check object:

1. In the deployment manifests from the Contrail Networking download page, use the `hc.yaml` file (shown below) for the YAML definition for health-check objects. The same folder also includes the `hc_pod.yaml`, which has the YAML definition to associate the health-check object with VMI by means of pod definitions.

   Sample `hc.yaml` file:

```
apiVersion: v1
kind: Namespace
metadata:
  name: healthcheck
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: ServiceHealthCheck
metadata:
  name: ping-hc
  namespace: healthcheck
spec:
  serviceHealthCheckProperties:
    delay: 2
    enabled: true
    healthCheckType: end-to-end
    maxRetries: 5
    monitorType: PING
    timeout: 5
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: ServiceHealthCheck
metadata:
  name: bfd-hc
  namespace: healthcheck
spec:
  serviceHealthCheckProperties:
```

```
    delay: 2
    enabled: true
    healthCheckType: link-local
    maxRetries: 5
    monitorType: BFD
    timeout: 5
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: ServiceHealthCheck
metadata:
  name: http-hc
  namespace: healthcheck
spec:
  serviceHealthCheckProperties:
    delay: 2
    enabled: true
    healthCheckType: end-to-end
    maxRetries: 5
    monitorType: HTTP
    timeout: 5
    httpMethod: GET
    expectedCodes: 200
    urlPath: /health
```

2. Complete the parameters to define the health check. Table 17 on page 129 lists and explains the parameters.

Table 17: Health-Check Configurable Parameters

Field	Description
`Delay`	The delay, in seconds, to repeat the health check.
`DelayUsecs`	Time in micro seconds at which the health check is repeated.
`Enabled`	Indicates that the health check is enabled. The default is `False`.
`ExpectedCodes`	When the monitor protocol is HTTP, the expected return code for HTTP operations must be in the range of 200-299.

Table 17: Health-Check Configurable Parameters *(Continued)*

Field	Description
`HealthCheckType`	Indicates the health-check type: `link-local`, `end-to-end`, `segment`, `vn-ip-list`, and `end2end`. The default is `link-local`. In both `link-local` and `end-to-end` modes, the health check is executed for the pod on the vRouter where the VMI is running.
`HttpMethod`	When the monitor protocol is HTTP, the type of HTTP method used is GET.
`MaxRetries`	The number of retries to attempt before declaring a health down instance .
`MonitorType`	The protocol type to be used is PING, BFD, or TCP.
`targetIpList`	**NOTE**: Attribute is configurable but not supported in CN2 Release 22.3. This attribute will be supported in a future release.
`targetIpAll`	**NOTE**: Attribute is configurable but not supported in CN2 Release 22.3. This attribute will be supported in a future release.
`Timeout`	The number of seconds to wait for a response.
`TimeoutUsecs`	Time in micro seconds to wait for response.
`UrlPath`	Must be a valid URL, such as `http://172.16.0.1/`<*path*>. The IP address can be a placeholder that will be replaced with the pod link-local IP address or metadata IP address.

Following is an abstract Golang schema for the health-check resource:

```
type ServiceHealthCheckProperties struct {
    Delay              *int
    DelayUsecs         *int
```

```
    Enabled            boolean
    ExpectedCodes      int // Only for http
    HealthCheckType    (link-local | end-to-end |segment | vn-ip-list) //end2end
    HttpMethod          *string
    MaxRetries          int
    MonitorType         (ping | BFD |TCP)
    Timeout             int
    TimeoutUsecs        int

}

type ServiceHealthCheckSpec struct {
    ServiceHealthCheckProperties *ServiceHealthCheckProperties
}

type ServiceHealthCheckStatus struct {
    uuid *string
}

type ServiceHealthCheck struct {
    <kube_specific_objetcs>
    Spec     ServiceHealthCheckSpec
    Status   ServiceHealthCheckStatus
}
```

The YML representation for the Golang schema is:

````
```
apiVersion: core.contrail.juniper.net/v1alpha1
kind: ServiceHealthCheck
metadata:
  name: ping-hc
  namespace: healthcheck
spec:
  serviceHealthCheckProperties:
    delay: 2
    enabled: true
    healthCheckType: end-to-end  #valid values are link-local|end-to-end|segment|vn-ip-list
    maxRetries: 5
    monitorType: PING  #valid are PING|HTTP|BFD
    timeout: 5
````
````

~~~
```
~~~

3. Link the health-check object to the VMI by means of the pod annotation reference value `core.juniper.net/health-check`. The default behavior is to associate the health check with the primary interface.

```
apiVersion: v1
kind: Pod
metadata:
  name: hc_pod
  namespace: hc_ns
  annotations:
     core.juniper.net/health-check: '[{"name": "ping-hc", "namespace": "healthcheck"}]'
spec:
  <>
```

4. (Optional) To link the health check with multiple interfaces, attached to a different Network Attachment Definition (NAD) or virtual network (VN), you can refer the health check object within the `cni-args` section. Following is an example of configured `cni-args` in annotations.

```
apiVersion: v1
kind: Pod
metadata:
  name: hc_pod
  namespace: hc_ns
  annotations:
          k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "hc-vn",
          "namespace": "healthcheck",
          "cni-args": {
            "core.juniper.net/health-check": "[{\"name\": \"ping-hc\", \"namespace\":
\"healthcheck\"}]"
          }
        }
      ]
spec:
  <>
```

Existing VMI objects will have a new field to reference the `HealthCheck` object.

```
type VirtualMachineInterfaceStatus struct {
    <existing_vmi_status_attributes>
    ServiceHealthCheckReference *ResourceReference
}


type VirtualMachineInterface struct {
    <other VMI attributes>
    Status   VirtualMachineInterfaceStatus
}
```

For the PING or HTTP monitoring-based health check, the minimum interval is **1second**. If you need a sub-second level health check for critical applications, you can opt for the BFD-based monitoring type.

Health-Check Process

The Contrail vRouter agent is responsible for providing the health-check service. The agent spawns a health-check probe process to monitor the status of a service hosted on the same compute node. Then the process updates the status to the vRouter agent.

The vRouter agent acts on the status provided by the script to withdraw or restore the exported interface routes. The agent is responsible for providing a link-local metadata IP address for allowing the script to communicate with the destination IP address from the underlay network, using appropriate NAT translations. In a running system, this information is displayed in the vRouter agent introspect at:

```
http://<compute-node-ip>:8085/Snh_HealthCheckSandeshReq?uuid=
```

Kubernetes Ingress Support

SUMMARY

Cloud-Native Contrail® Networking™ supports the Container Network Interface (CNI) for integration with Kubernetes. This topic provides an overview of Kubernetes ingress service implementation in Cloud-Native Contrail Networking. This topic also contains a list of validated Kubernetes ingress controllers and their installation instructions.

IN THIS SECTION

- Ingress Controller Overview | 134
- Validated Ingress Controllers | 136
- NGINX Ingress Controller | 136
- HAProxy Ingress Controller | 137
- Contour Ingress Controller | 137

Ingress Controller Overview

You must have a Kubernetes ingress controller for an ingress to function properly. An ingress controller receives traffic from outside a Kubernetes cluster and routes and load-balances that traffic to containers within a cluster. Ingress controllers also manage egress traffic between services within a cluster and external services. Controllers automatically route traffic to containers depending on service requirements.

Ingress controllers deploy ingress resources. Ingress resources comprise rules that specify which inbound traffic connections reach which services (pods). Ingress resources, combined with ingress controllers, route Layer 7 traffic to containers in a cluster.

Here's an example of an NGINX ingress resource:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apple-app-echo
  namespace: ingress-nginx-test
spec:
  selector:
    matchLabels:
      app: apple-echo
  replicas: 1
  template:
    metadata:
```

```
    labels:
      app: apple-echo
  spec:
    containers:
    - name: apple-echo
      image: <nginx>:<latest>
      args:
        - "-text=apple-echo"
```

Ingress resources contain `spec` fields such as `type: NodePort` and `type: LoadBalancer`. These service types determine a controller's traffic routing and forwarding behavior. For example, if you enter a `NodePort` in the `type` field, the control plane allocates a port from a range (default 30000–32767) of ports to your service.

Consider the following example:

```
apiVersion: v1
kind: Service
metadata:
  name: envoy
  namespace: projectcontour
  annotations:
spec:
  #externalTrafficPolicy: Local
  ports:
  - port: 80
    name: http
    protocol: TCP
    nodePort: 30080
    targetPort: 8080
  - port: 443
    name: https
    protocol: TCP
    targetPort: 8443
    nodePort: 30443
  selector:
    app: envoy
  type: NodePort
```

Some highlights from the example above include:

- `selector`: The label selector that determines which set of pods this service targets. In this example, this service selects any pod with the label `app: envoy`.

- `port`: The service port (80).
- `targetPort`: The actual port used by the application in the container (8080).
- `nodePort`: The port on the host of each node in the cluster that your service is exposed to (30080).

Different ingress controllers require different configurations. Review the documentation of your ingress controller for annotation, specification, and configuration information.

Validated Ingress Controllers

Cloud-Native Contrail Networking supports many ingress controllers. We've validated the following three popular third-party controllers for use with Cloud-Native Contrail Networking:

- NGINX
- HAProxy
- Contour

NGINX Ingress Controller

NGINX is an open-source HTTP server that also functions as a reverse proxy, load balancer, and IMAP or POP3 proxy server. The NGINX ingress controller is a Kubernetes controller that deploys an NGINX configuration using a ConfigMap resource. Other than endpoint-only changes, you must reload NGINX after any change to the configuration file occurs. This reload mechanism is powered by a lua-nginx-module. NGINX requires Kubernetes v1.22 or later.

NOTE: We support the NGINX ingress controller in environments using Cloud-Native Contrail Networking as the software-defined networking (SND) solution starting in Contrail Networking Release 21.4.

See the NGINX Ingress Controller installation guide for installation instructions. This guide contains instructions for installing NGINX using several different methods (Docker, minikube, Helm).

HAProxy Ingress Controller

The HAProxy ingress controller provides TCP and HTTP routing and high availability (HA) load balancing. HAProxy offers features such as Runtime API, Data Plane API, and hitless reloads. These features excel in dynamic, high-traffic environments where users constantly deploy, configure, and terminate pods, services, and microservices. The HAProxy ingress controller v0.13 requires Kubernetes v1.19 or later.

NOTE: Starting in Contrail Networking Release 21.4, we support the HAProxy ingress controller in environments using Cloud-Native Contrail Networking as the SDN solution.

See the HAProxy Getting Started guide for installation instructions.

NOTE: You must use Helm to install and to configure the HAProxy ingress controller. See Installing Helm for more information.

Contour Ingress Controller

Contour ingress controller deploys an Envoy proxy as a reverse proxy and load balancer. Envoy is a Layer 7 bus network for proxy services and communication. Envoy is deployed as a self-contained proxy instead of a library. As a result, any application can access Envoy's load-balancing features. This implementation is suitable for a distributed system such as a Kubernetes cluster. Other benefits of Contour include:

- Easy installation and integration of Envoy.
- Stable ingress support in multi-team Kubernetes clusters.
- Dynamic updates and ingress configuration without interruptions or dropped connections.

Contour requires Kubernetes 1.16 or later. You must enable role-based access control (RBAC) in your cluster for Contour to function properly.

NOTE: Starting in Contrail Networking Release 21.4, we support the Contour ingress controller in environments using Cloud-Native Contrail Networking as the SDN solution.

See the Getting Started guide for instructions about how to install the Contour ingress controller. This guide contains instructions about how to install and configure Contour with either kind or Docker. Install Contour after installing kind or Docker to run your ingress controller.

Deploy VirtualNetworkRouter in Cloud-Native Contrail Networking

SUMMARY

Cloud-Native Contrail® Networking™ supports the `VirtualNetworkRouter` (VNR) construct. This construct provides connectivity between `VirtualNetworks`.

IN THIS SECTION

- VirtualNetworkRouter Overview | 139
- VirtualNetworkRouter Use Cases | 139
- Mesh Use Cases | 139
- Hub-spoke Use Cases | 140
- Mesh VNR That Connects Two or More Virtual Networks in the Same Namespace | 140
- Add New Virtual Networks Within the Same Namespace to an Existing Mesh-Type VNR | 141
- Two Mesh VNRs in the Same Namespace | 142
- Two Mesh VNRs with Different Namespaces | 143
- Hub and Spoke VNRs in the Same Namespace | 144
- Hub and Spoke VNRs in Different Namespaces | 145
- Same Virtual Networks Under Multiple VNRs | 146
- Use Case Explanation | 146
- Standard Use Case: Single VNR Connecting Two Virtual Networks | 147

- Update Use Case: Single VNR Connecting Two Additional Virtual Networks | 150
- VirtualNetworkRouter Configuration | 154
- API Type (Schema) | 154
- Mesh VNR | 155
- Spoke VNR | 156
- Hub VNR | 156

VirtualNetworkRouter Overview

Typically, `VirtualNetwork` (VN) traffic is isolated to maintain tenant separation. In Cloud-Native Contrail Networking (CN2), `VirtualNetworkRouter` (VNR) performs route leaking. Route leaking establishes connectivity between `VirtualNetworks` by importing routing instances (RI) and the routing tables associated with these instances. As a result, devices in one routing table can access resources from devices in another routing table.

The VNR provides connectivity for the following two common network models:

- **Mesh**: Pods in all connected `VirtualNetworks` communicate with each other.
- **Hub-spoke**: `VirtualNetworks` connect to two different VNR types (spoke, hub). `VirtualNetworks` connected to spoke-type VNRs communicate with `VirtualNetworks` connected to hub-type VNRs and vice versa. `VirtualNetworks` connected to spoke VNRs cannot communicate with other `VirtualNetworks` attached to spoke VNRs.

VNR is a Kubernetes construct deployed within CN2.

VirtualNetworkRouter Use Cases

The following examples are common use cases that demonstrate the functionality of VNR in CN2.

Mesh Use Cases

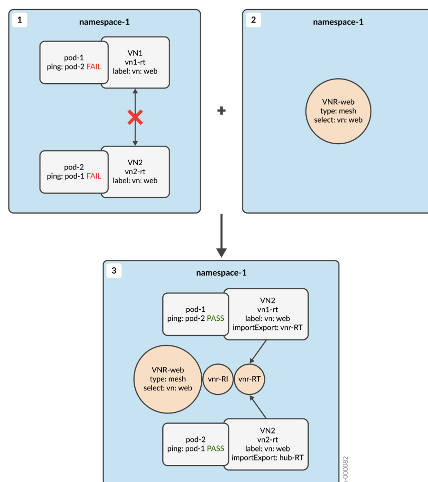
- "Mesh VNR That Connects Two or More Virtual Networks in the Same Namespace" on page 140

- "Add New Virtual Networks Within the Same Namespace to an Existing Mesh-Type VNR" on page 141
- "Two Mesh VNRs in the Same Namespace" on page 142
- "Two Mesh VNRs with Different Namespaces" on page 143

Hub-spoke Use Cases

- "Hub and Spoke VNRs in the Same Namespace" on page 144
- "Hub and Spoke VNRs in Different Namespaces" on page 145
- "Same Virtual Networks Under Multiple VNRs" on page 146

Mesh VNR That Connects Two or More Virtual Networks in the Same Namespace



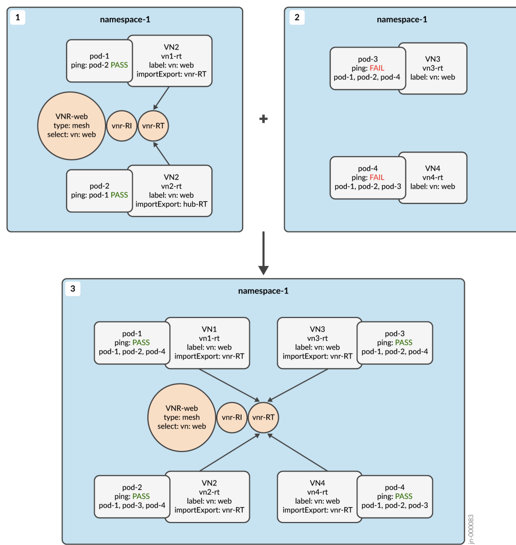
1. Figure-1: The user creates VN1 and VN2 in namespace-1. Pods in VN1 cannot connect to pods in VN2. This is the default behavior of `VirtualNetworks` in CN2.

2. Figure-2: The user defines a VNR of type mesh that selects VN1 and VN2. This VNR allows Pods in VN1 to communicate with Pods in VN2 and vice-versa.

3. Figure-3: Pods in VN1 connect to Pods in VN2. The route-target of VNR is `importExported` to both `VirtualNetworks`.

"Back to VirtualNetworkRouter Use Cases" on page 139

Add New Virtual Networks Within the Same Namespace to an Existing Mesh-Type VNR



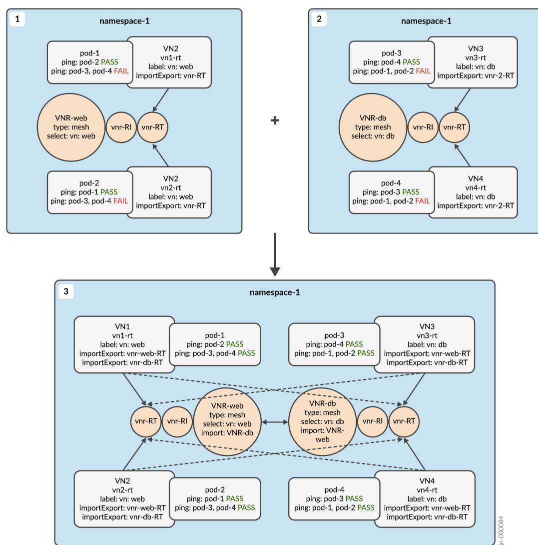
1. Figure-1: Two `VirtualNetworks` (VN1, VN2) connect to VNR in namespace-1.

2. Figure-2: The user creates two new `VirtualNetworks` (VN3, VN4).

3. Figure-3: VN3 and VN4 connect to VNR. As a result, all `VirtualNetworks`connected to the VNR receive connectivity.

"Back to VirtualNetworkRouter Use Cases" on page 139

Two Mesh VNRs in the Same Namespace



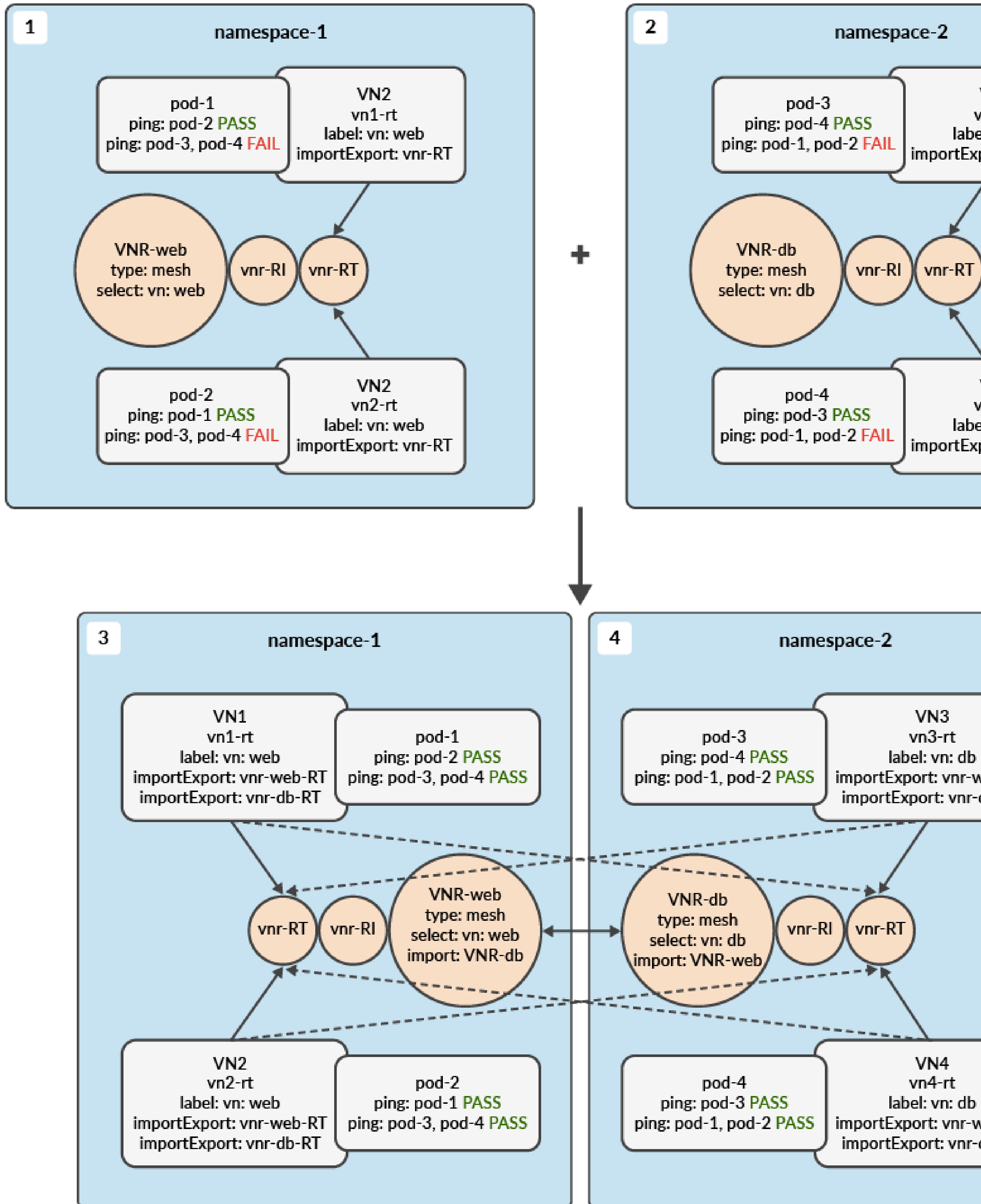
1. Figure 1 and Figure 2: VNR-web and VNR-db of type mesh already exist in namespace-1. Only VNRs connected to respective VNRs communicate with each other.

2. Figure 1 and Figure 2: VNR-web and VNR-db communicate with each other.

3. Figure 3: All `VirtualNetworks` connected to both VNR-web and VNR-db communicate with each other.

"Back to VirtualNetworkRouter Use Cases" on page 139

Two Mesh VNRs with Different Namespaces

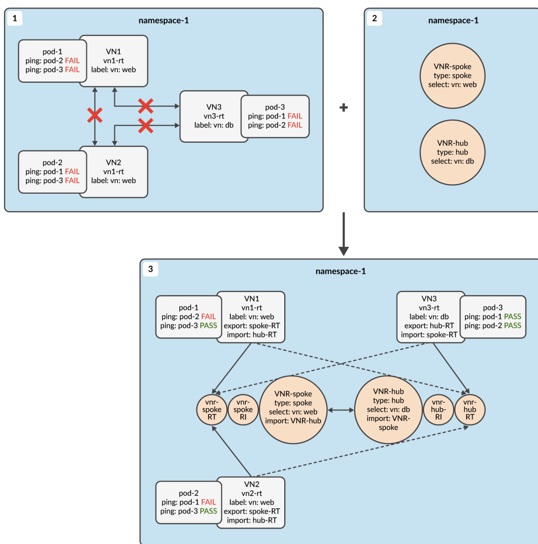


1. Figure 1: VNR-web selects VN1 and VN2. Pods in VN1 and VN2 communicate with each other. VN1 and VN2 cannot communicate with VN3 or VN4.
2. Figure 2: VNR-db selects VN3 and VN4. Pods in VN3 and VN4 communicate with each other. VN3 and VN4 cannot communicate with VN1 or VN2.
3. Figure 3: The user updates VNR-web to select VNR-db.
4. Figure 3: The user updates VNR-db to select VNR-web.
5. Figure 3: Since two VNRs select each other, VNR-web's RT (route target) is added to VN3 and VN4. VNR-db's RT is added to VN1 and VN2. Pods in VN1, VN2, VN3, and VN4 communicate with each other.

> **NOTE**: VNRs select VNs based on `matchExpression` labels in a `virtualNetworkSelector` spec. For example, in the illustration above, VNR-web in namespace-1 selects VN1 and VN2 based on the label `vn: web` from namespace-1. A `virtualNetworkSelector` only looks for matching labels within it's own namespace.

"Back to VirtualNetworkRouter Use Cases" on page 139

Hub and Spoke VNRs in the Same Namespace

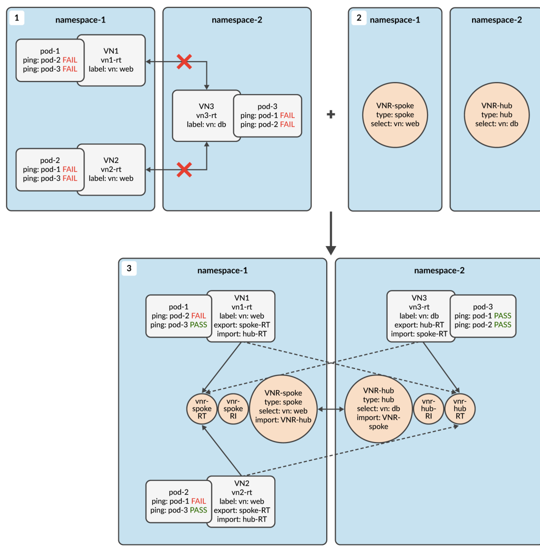


- Figure-1: Pods in VN1 cannot communicate with pods in VN2. VN1 and VN2 cannot communicate with VN3.

- Figure-2: The user creates a VNR of type "spoke" and "hub." VNR-spoke and VNR-hub import each other's RTs.
- Figure-3: VNR-spoke and VNR-hub's RTs are added to VN1, VN2, and VN3 because they import each others' RTs. As a result, pods in VN1 and VN2 communicate with VN3. Pods in VN1 and VN2 cannot communicate with each other.

"Back to VirtualNetworkRouter Use Cases" on page 139

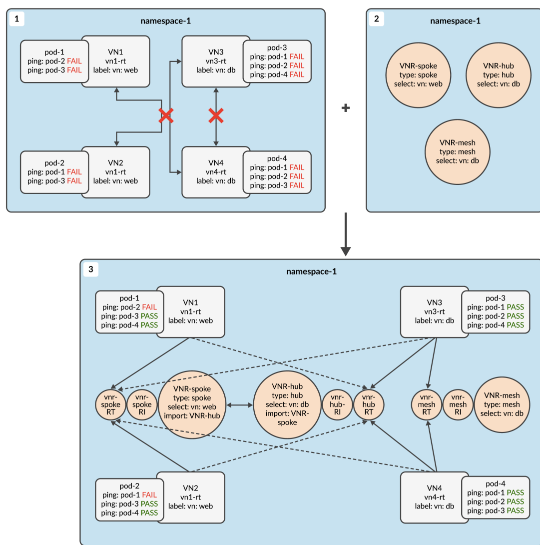
Hub and Spoke VNRs in Different Namespaces



- Figure 1 through Figure 3 are the same as "Hub and Spoke VNRs in the Same Namespace" on page 144, except that VNR-spoke and VNR-hub operate in different namespaces.

"Back to VirtualNetworkRouter Use Cases" on page 139

Same Virtual Networks Under Multiple VNRs



- Figure 1: Pods in VN1 and VN2 cannot communicate with each other. Also resources on VN3, VN4 can communicate with each other.
- Figure 2: You create a VNR-spoke by selecting VN1 and VN2. You create a VNR-hub by selecting VN3 and VN4. You create a VNR-mesh by selecting VN3 and VN4.
- Figure 3: VNR-spoke ensures that VN1 and VN2 cannot communicate each other, VNR-hub lets VN1 and VN2 reach VN3 and VN4, and VNR-mesh enables communication between VN3 and VN4.

"Back to VirtualNetworkRouter Use Cases" on page 139

Use Case Explanation

This section comprises the following two VNR use cases along with end-to-end explanations of each use case:

- "Standard Use Case: Single VNR Connecting Two Virtual Networks" on page 147
- "Update Use Case: Single VNR Connecting Two Additional Virtual Networks" on page 150

Standard Use Case: Single VNR Connecting Two Virtual Networks

```
apiVersion: v1
kind: Namespace
metadata:
  name: ns-single-mesh
  labels:
    ns: ns-single-mesh
spec:
  finalizers:
  - kubernetes
---
apiVersion: core.contrail.juniper.net/v1
kind: Subnet
metadata:
  namespace: ns-single-mesh
  name: subnet-1
  annotations:
    core.juniper.net/display-name: subnet_vn_1
spec:
  cidr: "10.10.1.0/24"
  defaultGateway: 10.10.1.254
---
apiVersion: core.contrail.juniper.net/v1
kind: Subnet
metadata:
  namespace: ns-single-mesh
  name: subnet-2
  annotations:
    core.juniper.net/display-name: subnet_vn_2
spec:
  cidr: "10.10.2.0/24"
  defaultGateway: 10.10.2.254
---
apiVersion: core.contrail.juniper.net/v1
kind: VirtualNetwork
metadata:
  namespace: ns-single-mesh
  name: vn-1
  annotations:
    core.juniper.net/display-name: vn-1
```

```
  labels:
    vn: web
spec:
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v1
    kind: Subnet
    namespace: ns-single-mesh
    name: subnet-1
---
apiVersion: core.contrail.juniper.net/v1
kind: VirtualNetwork
metadata:
  namespace: ns-single-mesh
  name: vn-2
  annotations:
    core.juniper.net/display-name: vn-2
  labels:
    vn: web
spec:
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v1
    kind: Subnet
    namespace: ns-single-mesh
    name: subnet-2
---
apiVersion: core.contrail.juniper.net/v1
kind: VirtualNetworkRouter
metadata:
  namespace: ns-single-mesh
  name: vnr-1
  annotations:
    core.juniper.net/display-name: vnr-1
  labels:
      vnr: web
spec:
  type: mesh
  virtualNetworkSelector:
    matchExpressions:
      - key: vn
            operator: In
            values:
            - web
---
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-vn-1
  namespace: ns-single-mesh
  annotations:
    k8s.v1.cni.cncf.io/networks: vn-1
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: agent-mode
            operator: NotIn
            values:
            - dpdk
  containers:
  - name: pod-vn-1
    image: gcr.io/cos-cloud/toolbox
    command: ["bash","-c","while true; do sleep 60s; done"]
    securityContext:
      privileged: true
    imagePullPolicy: IfNotPresent
  restartPolicy: OnFailure
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-vn-2
  namespace: ns-single-mesh
  annotations:
    k8s.v1.cni.cncf.io/networks: vn-2
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: agent-mode
            operator: NotIn
            values:
            - dpdk
```

```
  containers:
  - name: pod-vn-2
    image: gcr.io/cos-cloud/toolbox
    command: ["bash","-c","while true; do sleep 60s; done"]
    securityContext:
      privileged: true
    imagePullPolicy: IfNotPresent
  restartPolicy: OnFailure
```

This use case comprises the following:

- Two `VirtualNetworks` (`vn-1` and `vn-2`) in namespace `ns-single-mesh`. Both virtual networks have the `label vn: web`. Each `VirtualNetwork` contains a single pod. The `VirtualNetwork vn-1` contains `pod-vn-1`. The `VirtualNetwork vn-2` contains `pod-vn-2`.
- A `type: mesh` VNR with the name `vnr-1`. This VNR establishes connectivity between the two `VirtualNetworks` using `matchExpressions` and `vn: web`. he VNR imports the RI and routing table of `vn-1` to `vn-2` and vice versa. Since `vnr-1` is a mesh-type VNR, all pods in connected `VirtualNetworks` communicate with each other.

Update Use Case: Single VNR Connecting Two Additional Virtual Networks

```
apiVersion: v1
kind: Namespace
metadata:
  name: ns-single-mesh
  labels:
    ns: ns-single-mesh
spec:
  finalizers:
  - kubernetes
---
apiVersion: core.contrail.juniper.net/v1
kind: Subnet
metadata:
  namespace: ns-single-mesh
  name: subnet-2
  annotations:
```

```
    core.juniper.net/display-name: subnet_vn_1
spec:
  cidr: "10.10.3.0/24"
  defaultGateway: 10.10.3.254
---
apiVersion: core.contrail.juniper.net/v1
kind: Subnet
metadata:
  namespace: ns-single-mesh
  name: subnet-4
  annotations:
    core.juniper.net/display-name: subnet_vn_2
spec:
  cidr: "10.10.4.0/24"
  defaultGateway: 10.10.4.254
---
apiVersion: core.contrail.juniper.net/v1
kind: VirtualNetwork
metadata:
  namespace: ns-single-mesh
  name: vn-3
  annotations:
    core.juniper.net/display-name: vn-1
  labels:
    vn: db
spec:
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v1
    kind: Subnet
    namespace: ns-single-mesh
    name: subnet-3
---
apiVersion: core.contrail.juniper.net/v1
kind: VirtualNetwork
metadata:
  namespace: ns-single-mesh
  name: vn-4
  annotations:
    core.juniper.net/display-name: vn-2
  labels:
    vn: middleware
spec:
  v4SubnetReference:
```

```
    apiVersion: core.contrail.juniper.net/v1
    kind: Subnet
    namespace: ns-single-mesh
    name: subnet-4
---
apiVersion: core.contrail.juniper.net/v1
kind: VirtualNetworkRouter
metadata:
  namespace: ns-single-mesh
  name: vnr-2
  annotations:
    core.juniper.net/display-name: vnr-1
  labels:
      vnr: db
      vnr: middleware
spec:
  type: mesh
  virtualNetworkSelector:
    matchExpressions:
      - key: vn
            operator: In
            values:
            - db, middlware
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-vn-3
  namespace: ns-single-mesh
  annotations:
    k8s.v1.cni.cncf.io/networks: vn-1
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: agent-mode
            operator: NotIn
            values:
            - dpdk
  containers:
  - name: pod-vn-3
```

```
    image: gcr.io/cos-cloud/toolbox
    command: ["bash","-c","while true; do sleep 60s; done"]
    securityContext:
      privileged: true
    imagePullPolicy: IfNotPresent
  restartPolicy: OnFailure
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-vn-4
  namespace: ns-single-mesh
  annotations:
    k8s.v1.cni.cncf.io/networks: vn-2
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: agent-mode
            operator: NotIn
            values:
            - dpdk
  containers:
  - name: pod-vn-4
    image: gcr.io/cos-cloud/toolbox
    command: ["bash","-c","while true; do sleep 60s; done"]
    securityContext:
      privileged: true
    imagePullPolicy: IfNotPresent
  restartPolicy: OnFailure
```

This use case is similar to the standard use case, except that in this use case the user updates the YAML file with an additional `type: mesh` VNR to connect two new `VirtualNetworks` (`vn-3` and `vn-4`) in namespace `ns-single-mesh`. Note the following:

- The VNR shown has the name `vnr-2` in namespace `ns-single-mesh` with `matchExpressions: db, middlware`.
- The `VirtualNetwork` `vn-3` has the label `vn: db`, and `vn-4` has the label `vn: middleware`.

As a result, `vnr-2` imports the RI and routing table of `vn-3` to `vn-4` and vice versa.

VirtualNetworkRouter Configuration

The following section provides YAML configuration information for the following resources:

- "API Type (Schema)" on page 154
- "Mesh VNR" on page 155
- "Spoke VNR" on page 156
- "Hub VNR" on page 156

API Type (Schema)

```
type VirtualNetworkRouterSpec struct {
    // Common spec fields
    CommonSpec `json:",inline" protobuf:"bytes,1,opt,name=commonSpec"`

    // Type of VirtualNetworkRouter. valid types - mesh, spoke, hub
    Type VirtualNetworkRouterType `json:"type,omitempty" protobuf:"bytes,2,opt,name=type"`

    // Select VirtualNetworks to which this VNR's RT be shared
    VirtualNetworkSelector *metav1.LabelSelector `json:"virtualNetworkSelector,omitempty"
protobuf:"bytes,3,opt,name=virtualNetworkSelector"`

    // Import Router targets from other virtualnetworkrouters
    Import ImportVirtualNetworkRouter `json:"import,omitempty"
protobuf:"bytes,4,opt,name=import"`
}

type ImportVirtualNetworkRouter struct {
    VirtualNetworkRouters []VirtualNetworkRouterEntry `json:"virtualNetworkRouters,omitempty"
protobuf:"bytes,1,opt,name=virtualNetworkRouters"`
}

type VirtualNetworkRouterEntry struct {
    VirtualNetworkRouterSelector *metav1.LabelSelector
`json:"virtualNetworkRouterSelector,omitempty"
protobuf:"bytes,1,opt,name=virtualNetworkRouterSelector"`
    NamespaceSelector            *metav1.LabelSelector `json:"namespaceSelector,omitempty"
```

```
protobuf:"bytes,2,opt,name=namespaceSelector"`
}
```

Mesh VNR

```
apiVersion: core.contrail.juniper.net/v1
kind: VirtualNetworkRouter
metadata:
  namespace: frontend
  name: vnr-1
  annotations:
    core.juniper.net/display-name: vnr-1
  labels:
    vnr: web
    ns: frontend
spec:
  type: mesh
  virtualNetworkSelector:
    matchLabels:
      vn: web
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: db
        namespaceSelector:
          matchLabels:
            ns: backend
```

The preceding YAML file is an example of a mesh VNR with the name `vnr-1` in namespace `frontend`, with the `labels` `vnr: web` and `ns: frontend`. This VNR imports its route-target to any VNR in the namespace `backend` with `matchLabel` `vnr: db`.

Spoke VNR

```
kind: VirtualNetworkRouter
metadata:
  namespace: frontend
  name: vnr-1
  annotations:
    core.juniper.net/display-name: vnr-1
  labels:
    vnrgroup: spokes
    ns: frontend
spec:
  type: spoke
  virtualNetworkSelector:
    matchLabels:
      vngroup: spokes
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnrgroup: hubs
        namespaceSelector:
          matchLabels:
            ns: backend
```

The preceding YAML file is an example of a spoke VNR with the name `vnr-1` in namespace `frontend` with the `labels` `vnrgroup: spokes` and `ns: frontend`. This VNR imports its route-targets to any VNR in the namespace `backend` with `matchLabel` `vnrgroup: hubs`.

Hub VNR

```
apiVersion: core.contrail.juniper.net/v1
kind: VirtualNetworkRouter
metadata:
  namespace: backend
  name: vnr-2
  annotations:
    core.juniper.net/display-name: vnr-2
  labels:
```

```
    vnrgroup: hubs
    ns: backend
spec:
  type: hub
  virtualNetworkSelector:
    matchLabels:
      vngroup: hubs
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnrgroup: spokes
        namespaceSelector:
          matchLabels:
            ns: frontend
```

The preceding YAML file is an example of a hub VNR with the name `vnr-2` in the namespace `backend` with `labels vnrgroup: hubs` and `ns: backend`. This VNR imports its route-targets to any VNR in the namespace `frontend` with `matchLabels vnrgroup: spokes`.

Configure Inter-Virtual Network Routing Through Route Targets

SUMMARY

Cloud-Native Contrail® Networking™ (CN2) supports inter-virtual network routing using route targets. Specify common route targets to route traffic between your virtual networks.

IN THIS SECTION

- Virtual Networks and Routing Instances Overview | 158
- Route Target Overview | 158
- Enable Inter-Virtual Network Routing Through Route Targets with NAD | 159

Virtual Networks and Routing Instances Overview

A routing instance is a collection of routing tables, interfaces, and routing protocol parameters. The set of interfaces in a routing instance belongs to the routing tables, and the routing protocol parameters control the information in the routing tables. A single routing instance might have multiple routing tables —for example, unicast IPv4, unicast IPv6, and multicast IPv4 routing tables can exist in a single routing instance.

In virtual networking, a physical networking device might be split into multiple virtual routers, each with its own interfaces, routing instances, and associated virtual networks. Routing instances isolate traffic within a `VirtualNetwork`. If you want to route traffic between your virtual networks, you can define common route targets for those networks.

Route Target Overview

Route targets enable your virtual networks (namespaces) to exchange virtual routing and forwarding (VRF) routing tables in a Multiprotocol Label Switching (MPLS) configuration. A route target is a BGP Extended Communities Attribute that defines VPN membership. In other words, members of that VPN share all routes defined within an Extended Communities Attribute. You define the following two route targets in your VRF policy:

- Route-target import list: Defines a list of acceptable route targets for a VRF to import. When a provider edge (PE) router receives a route from another PE router, it compares the route targets to the route-target import list. Specifically, the PE router compares the route targets attached to each route against the route-target import list defined for each of its VRFs. If no new route target matches the route targets defined in the import list, the VRF rejects the route.

- Route-target export list: Defines a list of route targets attached to every route advertised to other PE routers in your VPN.

Depending on your network configuration, the import and export lists might be identical. Typically, you do the following:

- Allocate one route target extended-community value per VPN.

- Configure the import list and the export list to include the same information: the set of VPNs comprising the sites associated with the VRF.

For more complicated configurations like hub-and-spoke VPNs, the route-target import list and the route-target export list might not be identical.

Enable Inter-Virtual Network Routing Through Route Targets with NAD

Establish route-target communities by defining matching route targets in your `VirtualNetwork` resource. This enables you to route traffic between your virtual networks (namespaces). Add route targets to a `VirtualNetwork` resource object using the Network Attachment Definition (`NAD`).

The Network Attachment Definition (NAD) is a Custom Resource Definition (CRD) specified by the Kubernetes Network Plumbing Working Group. This CRD, `NAD`, defines how a pod attaches to a logical (virtual) or physical network that the `NAD` object refers to. In other words, the `NAD` object contains networking information (namespace, subnet, routing, interface) for a pod in relation to a network. You can define the following options for your `VirtualNetwork` resource in the `annotations` of a `NAD` YAML file:

- `ipamV4Subnet` (optional): Specifies an IPv4 CIDR subnet for your `VirtualNetwork`.
- `ipamV6Subnet` (optional): Specifies an IPv6 CIDR subnet for your `VirtualNetwork`.
- `routeTargetList` (optional): Lists import and export route targets.
- `importRouteTargetList` (optional): Lists route targets used as imports.
- `exportRouteTargetList` (optional): Lists route targets used as exports.
- `fabricSNAT` (optional): Toggles connectivity to the underlay network by port mapping. The default setting is `false`.

Additionally, the `NAD-Controller` monitors `NAD` object-creation events and creates and updates a `VirtualNetwork` accordingly. The `juniper.net/networks-status` annotation of the `NAD` updates success or error events during `VirtualNetwork` creation.

NOTE: If you do not specify a `juniper.net/networks` annotation, then Cloud-Native Contrail Networking treats the `NAD` resource as a third-party resource. Cloud-Native Contrail Networking does not create Contrail resources (such as `VirtualNetwork` and `Subnet`).

The following example shows a `NAD` YAML file with several `annotations` defined:

Example 1:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: nasa-network
  namespace: nm1
  annotations:
```

```
    juniper.net/networks: '{
      "ipamV4Subnet": "172.16.10.0/24",
      "ipamV6Subnet": "2001:db8::/64",
      "routeTargetList": ["target:23:4561"],
      "importRouteTargetList": ["target:10.2.2.2:561"],
      "exportRouteTargetList": ["target:10.1.1.1:561"],
      "fabricSNAT": true
    }'
    juniper.net/networks-status: # should be updated by Kube-Manager to status of NAD object.
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "nasa-network",
  "type": "contrail-k8s-cni"
}'
```

The `NAD-Controller` automatically updates the `VirtualNetwork` resource after you apply your `NAD` YAML file.

The following example shows a `VirtualNetwork` resource with several route-target options defined:

Example 2:

```
apiVersion: core.contrail.juniper.net/v1
kind: VirtualNetwork
metadata:
  namespace: project-sample
  name: virtualnetwork-sample
spec:
  routeTargetList:
    - target:23:4561
    - target:21L:7000
    - target:871:6540
  importRouteTargetList:
    - target:10.2.2.2:561
    - target:97:651
  exportRouteTargetList:
    - target:10.1.1.1:561
    - target:97:651
```

After establishing your desired network annotations, you can create a pod with custom interfaces that are attached to networks with shared route targets. These networks route traffic between one another as a result of the shared route targets defined in the `NAD` and `VirtualNetwork` objects.

The following example shows a pod YAML file with custom interfaces derived from the annotations in Example 1.

Example 3:

```
apiVersion: v1
  kind: Pod
  metadata:
    name: nasa-pod-1
    annotations:
      k8s.v1.cni.cncf.io/networks: |-
       [
         {
            "name":"nasa-network1",
            "namespace":"nasa-ns",
            "cni-args":null,
            "ips":["172.16.20.42"],
            "mac":"de:ad:00:00:be:ef",
            "interface":"tap1"
         },
         {
            "name":"nasa-network2",
            "namespace":"nasa-ns",
            "cni-args":null,
            "ips":["172.16.21.42"],
            "mac":"de:ad:00:00:be:ee",
            "interface":"tap2"
          }
        ]
```

Note that the two interfaces shown in the preceding code example (`nasa-network1` and `nasa-network2`) attach to different networks. As a result of `NAD` functionality, you can route traffic between these networks.

RELATED DOCUMENTATION

Understanding Route Targets

Routing Instances Overview

Virtual Routing Instances

Configure IPAM for Pod Networking

SUMMARY

Cloud-Native Contrail® Networking™ release 22.1 supports IP address management (IPAM) for pods and services. Configure a `Subnet` resource to facilitate IP address allocation.

IN THIS SECTION

- IPAM in Cloud-Native Contrail Networking | 162
- SubnetPool Overview | 162
- Subnet Overview | 163
- VirtualNetwork Overview | 164
- BGP as a Service Session IP Addresses Overview | 165

IPAM in Cloud-Native Contrail Networking

Cloud-Native Contrail Networking introduces the `Subnet` and `SubnetPool` resources for the purpose of IPAM for pods and services. Each `Subnet` has an associated `SubnetPool`. These resources enable you to configure IPv4 and IPv6 address allocation in your cluster. A `VirtualNetwork` references a `Subnet` resource to determine available subnets for new pods and services. Multiple `VirtualNetworks` can reference the same `Subnet`. The `Subnet` resource is translated into IPAM and consumed by the control node and vRouter agent.

SubnetPool Overview

The `SubnetPool` manages a pool of addresses from which `Subnets` are allocated. When a request for an IP address occurs, that IP address is allocated from a virtual network's associated `SubnetPool`. CIDR parameters (prefix length, capacity, range) for IP address allocation are determined when a `SubnetPool` is created. You can allocate additional prefixes if you exhaust a `SubnetPool`.

Consider the following `SubnetPool` example:

```
apiVersion: idallocator.contrail.juniper.net/v1
capacity: 262144
count: 157
kind: Pool
```

```
max: 262143
metadata:
  creationTimestamp: null
  name: subnet-id-pool-Subnet-contrail-k8s-kubemanager-ocp-rdang-q8roaw-contrail-default-
podnetwork-pod-v4-subnet
reserved:
- 0
- 262143
- 1
```

The `capacity` parameter denotes the total number of possible IDs in the pool. The `count` parameter denotes the number of used IDs in the pool. The `max` parameter denotes the maximum number of IDs available to be allocated from the pool. A given ID maps to an IP address in the `Subnet` pool.

Subnet Overview

The `Subnet` is a block of IP addresses and the configurations associated with those addresses. A `Subnet` is based on a single address family (IPv4, IPv6) at a time. You must create separate IPv4 and IPv6 `Subnets`. If you do not specify a `SubnetPool`, the `Subnet` functions as Contrail Classic IPAM. This means that the `Subnet` is isolated to a single namespace.

Consider the following `Subnet` spec example:

```
apiVersion: v13
kind: Subnet
metadata:
  name: default-servicenetwork-pod-v4-subnet
  namespace: contrail-k8s-kubemanager-ocp
spec:
  cidr: 10.128.0.0/16
  defaultGateway: 10.128.0.1
  ranges:
  - ipRanges:
    - from: 10.128.0.0
      to: 10.128.0.255
    key: contrail-k8s-kubemanager-ocp-user-4yu0qk-ocp-user-4yu0qk-ctrl-1
```

The `cidr` and `defaultGateway` parameters are the main parameters that define a `Subnet` resource. The `cidr` parameter determines the range of IPs available for allocation in that `Subnet`. The `defaultGateway` parameter

defines the IP address of the `defaultGateway` for the `Subnet`. Specifying a `defaultGateway` address is optional. If you do not specify a `defaultGateway` address, it is automatically set as the first IP address in the `Subnet`.

A Kubernetes node configuration can have a `podCIDR` configuration parameter. The `podCIDR` is a subset of the `default-podnetwork-subnet`. When the `podCIDR` is present, the IP address of any pod created on that node will have an IP address allocated from the `podCIDR`. If no `podCIDR` is present, all of the IP addresses in the CIDR of the `Subnet` can be allocated for the node. The `podCIDR` can also reference a wildcard key. In the example, IP address allocation requests choose from IPs `10.128.0.0` to `10.128.0.255` as long as the requesting pod is created on the node with the key `contrail-k8s-kubemanager-ocp-kparmar-4yu0qk-ocp-kparmar-4yu0qk-ctrl-1` .

Alternatively, you can define a `ranges` parameter . The `ranges` parameter defines a list of IPs available for allocation. The `ranges` parameter overrides the CIDR parameter when it is present in a spec. The `ranges` parameter does not override the `podCIDR` parameter.

VirtualNetwork Overview

Cloud-Native Contrail Networking updates the `VirtualNetwork` resource to be compatible with IPAM implementation. Consider the following example:

```
apiVersion: v3
kind: VirtualNetwork
metadata:
  namespace: contrail
  name: virtualnetwork-sample
spec:
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v1
    kind: Subnet
    namespace: contrail
    name: v4subnet
 v6SubnetReference:
    apiVersion: core.contrail.juniper.net/v1
    kind: Subnet
    namespace: contrail
    name: v6subnet
```

Note the separate `Subnet` references for the IPv4 address family and the IPv6 address family. You cannot update the `Subnet` reference of a `VirtualNetwork` through the entire lifecycle of that `VirtualNetwork`.

BGP as a Service Session IP Addresses Overview

BGP as a Service (BGPaaS) enables the establishment of a BGP session between a control node to a workload or pod's IP address. You can create a `Subnet` with the `DisableBGPaaSIPAutoAllocation` flag set to **false** or **true**. When you set the `DisableBGPaaSIPAutoAllocation` to **false**, the following occurs:

- No IP address is allocated for `BGPaaSPrimaryIP` or `BGPaaSSecondaryIP` immediately. These IPs are only allocated (within `Subnet` CIDR range) when the first `BGPAsAService` is configured within the network of this `Subnet`.
- When you delete all of the `BGPAsAService` resources associated with a `Subnet`, the IP addresses assigned to `BGPaaSPrimaryIP` and `BGPaaSSecondaryIP` are released from the pool and set to empty values. These addresses are re-allocated from the pool when a `BGPAsAService` is configured again.

When you set the `DisableBGPaaSIPAutoAllocation` flag to **true**, the following occurs:

- You are able to use user-defined values for the `BGPaaSPrimaryIP` and `BGPaaSSecondaryIP` fields. These IP fields are mandatory and cannot be left empty. User-defined values for these fields are also reserved in the `Subnet` pool.
- The IP addresses used for `BGPaaSPrimaryIP` and `BGPaaSSecondaryIP` still remain reserved in the `Subnet` pool even if no `BGPAsAService` is configured or if all `BGPAsAService` resources are deleted.

When you change the `DisableBGPaaSIPAutoAllocation` field from **false** to **true**, `BGPaaSPrimaryIP` and `BGPaaSSecondaryIP` become mandatory fields. If the IPs were auto allocated before changing this flag from **false** to **true**, then those IPs are released from the pool and new user-provided IPs are reserved in the pool.

When you change `DisableBGPaaSIPAutoAllocation` from **true** to **false** the following occurs:

- If no `BGPAsAService` is configured within the `Subnet`, `BGPaaSPrimaryIP` and `BGPaaSSecondaryIP` values are released from the pool and these fields become empty.
- If at least one `BGPAsAService` is configured, no change happens to the existing values of `BGPaaSPrimaryIP` and `BGPaaSSecondaryIP`.

For more information about BGP as a Service (BGPaaS), see the "Enable BGP as a Service" on page 96 section.

Enable VLAN Subinterface Support on Virtual Interfaces

SUMMARY

Virtualized Network Function (VNF) and Containerized Network Function (CNF) workloads often require multiple virtual network services on a single interface. Cloud-Native Contrail® Networking™ supports VLAN subinterfaces on virtual interfaces.

IN THIS SECTION

- VLAN Subinterface Overview | 166
- API Changes | 166
- Network Definition Changes | 167
- Configuration Use Cases | 168
- Valid Configuration 1: One Parent, One Subinterface: | 169
- Valid Configuration 2: One Parent, Multiple Subinterfaces: | 169
- Valid Configuration 3: Multiple Parents, Multiple Subinterfaces: | 170
- Invalid Configuration 1: Multiple Interfaces on Same Network: | 172
- Invalid Configuration 2: Two Interfaces with Same `interfacegroup` but no VLAN | 172

VLAN Subinterface Overview

A VLAN subinterface is a logical division of a virtual (or physical) interface at the network level. VLAN subinterfaces are Layer 3 interfaces that receive and forward 802.1Q VLAN tags. You can assign multiple VLAN tags to a single virtual interface. When a packet arrives at that interface, the packet's associated VLAN tags designate which VLAN the packet routes to. You can use VLAN subinterfaces to route traffic to multiple VLANs for your services.

API Changes

This section provides information about API calls that occur when configuring a VLAN subinterface.

When configuring VLAN subinterfaces in Cloud-Native Contrail Networking, Kubernetes updates the `VirtualMachineInterface` field with new `properties`, or VLAN tags. After an update occurs, the `VirtualMachineInterface` object references other `VirtualMachineInterface` objects based on existing VLAN tags.

> **NOTE**: Cloud-Native Contrail Networking defines the `properties` field from Contrail Classic as `virtualMachineInterfaceProperties`.

Network Definition Changes

This section provides information about the network definition enhancements necessary when creating a subinterface for a virtual interface within a pod.

In `kube-manager`, the `PodController` watching for pod events reads the network definition applied to it. `Kube-manager` parses each network selection element and creates an associated VMI (virtual machine interface). Parent VMIs are the network elements with only the `net.juniper.contrail.interfacegroup` tag attached in the YAML file. Subinterfaces are the network elements with the `net.juniper.contrail.interfacegroup` and `net.juniper.contrail.vlan` tags attached in the YAML file.

The following two tags enhance the network definition in the `cni-args` section:

- `net.juniper.contrail.interfacegroup`
  - Interface Group groups two or more interfaces.
  - The parent interface is the network selection element associated with only this tag.
  - The subinterface is the network selection element associated with this tag and a VLAN tag.
- `net.juniper.contrail.vlan`

  Specifies the VLANID on the subinterface.

  .

A VLAN subinterface belongs to its parent interface. Users must specify the namespace to which the subinterface attaches. Consider the following example:

Example

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: my-pod
  namespace: my-namespace
  annotations:
    k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "parent-vn",
          "namespace": "vn-ns",
          "cni-args": {
            "net.juniper.contrail.interfacegroup": "eth1"}
          ...
        },
        {
          "name": "subitf-vn",
          "namespace": "vn-ns",
          "cni-args": {
            "net.juniper.contrail.vlan": 100,
            "net.juniper.contrail.interfacegroup": "eth1"},
          ...
        },
  ...
```

The preceding example shows specified pod annotations for `cni-args`. This example configuration creates the following three VMIs and three interface IPs (IIPs) within the pod:

- VMI, IIP for eth0 on default pod network
- VMI, IIP for eth1 on `parent-vn` (parent interface)
- VMI, IIP for eth1.100 on `subitf-vn` (subinterface)

Configuration Use Cases

This section provides examples of different valid and invalid parent and subinterface configurations.

Valid Configurations

- "Valid Configuration 1: One Parent, One Subinterface:" on page 169
- "Valid Configuration 2: One Parent, Multiple Subinterfaces:" on page 169
- "Valid Configuration 3: Multiple Parents, Multiple Subinterfaces:" on page 170

Invalid Configurations

- "Invalid Configuration 1: Multiple Interfaces on Same Network:" on page 172
- "Invalid Configuration 2: Two Interfaces with Same interfacegroup but no VLAN" on page 172

Valid Configuration 1: One Parent, One Subinterface:

```
apiVersion: v1
kind: Pod
metadata:
  name: vlan100-0
  namespace: vlan-project
  annotations:
    k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "vlan-parent-vn",
          "namespace": "vlan-project",
          "cni-args": {
              "net.juniper.contrail.interfacegroup": "eth1"
          }
        },
        {
          "name": "vlan-subintf-vn",
          "namespace": "vlan-project",
          "cni-args": {
              "net.juniper.contrail.vlan": "100",
              "net.juniper.contrail.interfacegroup": "eth1"
          }
        },
  ...
```

Valid Configuration 2: One Parent, Multiple Subinterfaces:

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: vlan100-0
  namespace: vlan-project
  annotations:
    k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "vlan-parent-vn",
          "namespace": "vlan-project",
          "cni-args": {
              "net.juniper.contrail.interfacegroup": "eth1"
          }
        },
        {
          "name": "vlan-subintf-vn2",
          "namespace": "vlan-project",
          "cni-args": {
              "net.juniper.contrail.vlan": "200",
              "net.juniper.contrail.interfacegroup": "eth1"
          }
        },
        {
          "name": "vlan-subintf-vn",
          "namespace": "vlan-project",
          "cni-args": {
              "net.juniper.contrail.vlan": "100",
              "net.juniper.contrail.interfacegroup": "eth1"
          }
        }
      ]
```

Valid Configuration 3: Multiple Parents, Multiple Subinterfaces:

```
apiVersion: v1
kind: Pod
metadata:
  name: vlan100-0
  namespace: vlan-project
  annotations:
```

```
    k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "vlan-parent-vn",
          "namespace": "vlan-project",
          "cni-args": {
              "net.juniper.contrail.interfacegroup": "eth1"
          }
        },
        {
          "name": "vlan-subintf-vn2",
          "namespace": "vlan-project",
          "cni-args": {
              "net.juniper.contrail.vlan": "200",
              "net.juniper.contrail.interfacegroup": "eth1"
          }
        },
        {
          "name": "vlan-subintf-vn",
          "namespace": "vlan-project",
          "cni-args": {
              "net.juniper.contrail.vlan": "100",
              "net.juniper.contrail.interfacegroup": "eth1"
          }
        },
        {
          "name": "vlan-subintf-vn4",
          "namespace": "vlan-project",
          "cni-args": {
              "net.juniper.contrail.vlan": "100",
              "net.juniper.contrail.interfacegroup": "eth2"
          }
        },
        {
          "name": "vlan-subintf-vn3",
          "namespace": "vlan-project",
          "cni-args": {
              "net.juniper.contrail.interfacegroup": "eth2"
          }
        }
      ]
```

Invalid Configuration 1: Multiple Interfaces on Same Network:

```
apiVersion: v1
kind: Pod
metadata:
  name: vlan100-0
  namespace: vlan-project
  annotations:
    k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "vn1",
          "namespace": "vlan-project",
          "cni-args": {
              "net.juniper.contrail.interfacegroup": "eth1"
          }
        },
        {
          "name": "vn1",
          "namespace": "vlan-project",
          "cni-args": {
              "net.juniper.contrail.vlan": "200",
              "net.juniper.contrail.interfacegroup": "eth1"
          }
        },
      ]
```

Invalid Configuration 2: Two Interfaces with Same `interfacegroup` but no VLAN

```
apiVersion: v1
kind: Pod
metadata:
  name: vlan100-0
  namespace: vlan-project
  annotations:
    k8s.v1.cni.cncf.io/networks: |
      [
        {
```

```
            "name": "vn1",
            "namespace": "vlan-project",
            "cni-args": {
                "net.juniper.contrail.interfacegroup": "eth1"
            }
          },
          {
            "name": "vn2",
            "namespace": "vlan-project",
            "cni-args": {
                "net.juniper.contrail.interfacegroup": "eth1"
            }
          },
        ]
```

EVPN Networking Support

SUMMARY

Juniper Cloud-Native Contrail Networking release R22.4 supports EVPN/VXLAN forwarding using Type2 prefixes with Virtual Networks utilizing forwarding mode L2 and L2_L3. With EVPN for Layer 2 connectivity and VXLAN for data encapsulation, CN2 enables you to establish connectivity between a CN2 virtual network and an EVPN-VXLAN-signalled service.

IN THIS SECTION

- EVPN Overview | **173**
- VXLAN Overview | **174**
- EVPN-VXLAN Overview | **174**
- Configuring VXLAN Encapsulation Priority | **174**
- Manually Set a VXLAN VNI for Virtual Networks | **176**

EVPN Overview

EVPN is an extension to Border Gateway Protocol (BGP) that allows the network to carry endpoint reachability information such as Layer 2 MAC addresses and Layer 3 IP addresses. This control plane technology uses Multiprotocol BP (MP-BGP) for MAC and IP address endpoint distribution, where MAC addresses are treated as routes.

EVPN also provides multipath forwarding and redundancy through an all-active multihoming model. An endpoint or device connects to two or more upstream devices and forwards traffic using all the links. If a link or device fails, traffic continues to flow using the remaining active links.

With EVPN, MAC learning is handled in the control plane. This avoids the data plane flooding of unknown IPs typical with Layer 2 networks. EVPN also supports different data-plane encapsulation technologies between EVPN-VXLAN-enabled switches. In EVPN-VXLAN architecture, VXLAN provides the overlay data-plane encapsulation.

VXLAN Overview

VXLAN is an overlay tunneling protocol. The VXLAN tunneling protocol encapsulates MAC frames into UDP headers at Layer 2. This means that VXLAN encapsulates Ethernet frames into UDP packets with the physical network headers (IP header, Ethernet header) as outer headers. As a result, VXLAN enables devices to route UDP packets across networks, independent of the physical underlay.

The entity that performs the encapsulation and de-encapsulation is called a VXLAN tunnel endpoint (VTEP). After CN2 establishes a VXLAN segment, a VTEP encapsulates VM or pod traffic into a VXLAN header and sends that traffic to another VTEP. The receiving VTEP removes or strips off the encapsulation and forwards the data. VMs or pods must belong to the same VXLAN segment to communicate using VXLAN.

EVPN-VXLAN Overview

In Juniper networks, EVPN performs control plane functionality and VXLAN performs data plane functionality. EVPN handles MAC address learning in the control plane. VXLAN defines a tunneling scheme to overlay Layer 2 networks on top of Layer 3 networks. This tunneling scheme allows for optimal forwarding of Ethernet frames with support for an all-active multipathing of unicast and multicast traffic with the use of UDP/IP encapsulation for tunneling.

Configuring VXLAN Encapsulation Priority

The default encapsulation protocol for EVPN is MPLS over UDP. In order to enable EVPN-VXLAN for your cluster, you must change the encapsulation priority order in the `GlobalVrouterConfig` of your cluster. Set VXLAN as the top encapsulation priority in the `encapsulation` section of `encapsulationPriorities` in the `default-global-vrouter-config` object of `GlobalVrouterConfig`.

The following example `GlobalVrouterConfig` shows VXLAN as the top encapsulation priority.

```
apiVersion: core.contrail.juniper.net/v1
kind: GlobalVrouterConfig
metadata:
  creationTimestamp: "2022-08-23T10:45:52Z"
  generation: 6
  labels:
    core.juniper.net/parent: 9b83eddfaaa5778ad6b99cb81c803529cf911d492b9e7ec6d63d029d
  name: default-global-vrouter-config
  resourceVersion: "35583"
  uid: b2b57f5c-8dd0-4cd1-848f-0ec23f2819df
spec:
  encapsulationPriorities:
    encapsulation:
    - VXLAN
    - MPLSoGRE
    - MPLSoUDP
  fqName:
  - default-global-system-config
  - default-global-vrouter-config
  linklocalServices:
    linklocalServiceEntry:
    - ipFabricServiceIP:
      - 10.87.76.29
      - 10.87.76.31
      - 10.87.76.32
      ipFabricServicePort: 6443
      linklocalServiceIP: 10.200.0.1
      linklocalServiceName: kubernetes
      linklocalServicePort: 443
  parent:
    apiVersion: core.contrail.juniper.net/v1
    kind: GlobalSystemConfig
    name: default-global-system-config
    uid: 2f9ff5cf-4d40-4e8f-b7d3-2e403624c572
  portTranslationPools:
    pools:
    - portRange:
        endPort: 57023
        startPort: 56000
      protocol: tcp
```

```
    - portRange:
        endPort: 58047
        startPort: 57024
      protocol: udp
status:
  observation: ""
  state: Success
```

Manually Set a VXLAN VNI for Virtual Networks

A VXLAN network identifier (VNI) uniquely identifies the VXLAN segment. VNIs enable CN2 to use the same MAC frames across multiple VXLAN segments without traffic crossover. This means that CN2 can establish multiple VXLAN segments between the same VMs or pods with traffic isolation. VMs or pods on the same VNI communicate with each other and VMs or pods on separate VNIs need a router to communicate. VLAN tunnel endpoints (VTEPs) perform data encapsulation and reference VNI and MAC address information when looking up the forwarding table of another VM or pod. Once the VTEP determines a forwarding table, one VTEP endpoint sends a UDP packet to another endpoint over a network.

CN2 assigns a unique VNI to a virtual network (VN) upon VN creation. In CN2 release 22.4, you can manually set a VNI for your virtual networks. You can also define a VNI using a Network Attachment Definition (NAD).

The following YAML example shows a user-defined VNI in a VN.

```
apiVersion: core.contrail.juniper.net/v1
kind: VirtualNetwork
metadata:
  namespace: evpnl2test
  name: vn1
spec:
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v1
    kind: Subnet
    namespace: evpnl2test
    name: sn1
  virtualNetworkProperties:
    forwardingMode: l2
  virtualNetworkNetworkId: 5000
```

The following YAML example shows a user-defined VNI in a NAD.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: evpnl2
  namespace: evpnl2test
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "172.26.10.0/24",
      "virtualNetworkNetworkID": 5000
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "evpnl2",
  "type": "contrail-k8s-cni"
}'
```

RELATED DOCUMENTATION

EVPN User Guide

Customize Virtual Networks for Pod Deployments, Services, and Namespaces

SUMMARY

Starting in release 23.1, Juniper Cloud-Native Contrail Networking (CN2) supports the ability to apply a custom default network for namespaces, Deployments, and Services. This feature also supports environments with Multus CNI enabled. With Multus as the CNI of your CN2 environment,

IN THIS SECTION

- Custom Namespace Network Overview | **178**
- Apply a Custom Namespace Network to a Namespace | **178**

you can deploy pods that have multiple interfaces for different use cases.

- Apply a Custom Service Network to a Service | 179
- Designate a Virtual Network as a Custom Default Pod Network | 180
- Use a NAD to Create a Custom Default Pod Network | 180
- Deploy a Custom Pod Network per Pod | 181
- Custom Default Namespace Network Interactions | 183
- Multi-NIC Pod | 184

Custom Namespace Network Overview

In traditional Kubernetes, the default pod network is a single CIDR used by all pods in the cluster, regardless of namespace. This approach doesn't allow for network layer segmentation between pods because Kubernetes assigns IPs from a shared CIDR. CN2 addresses this drawback with isolated namespaces. CN2 isolated namespaces enable Kubernetes to create custom default namespace networks on a per-namespace basis. This means that when you configure a `Deployment` in a namespace with a custom namespace network, new pods and services use the custom network within an individual isolated namespace. Isolated namespaces ensure network isolation between pods and services without the need for a Kubernetes network policy.

CN2 improves on this feature by enabling Kubernetes to create pods that use custom namespace networks on a per-pod basis. Custom namespace networks provide their own `VirtualNetworks` (VNs) and Subnets. CN2 assigns pod IPs based on the Subnet parameters of a given custom namespace network. In other words, you can create pods with their own networks in a given namespace. This means that CN2 supports network isolation at the namespace level and pod level.

Apply a Custom Namespace Network to a Namespace

You can specify a custom default namespace network per namespace. Designate a custom namespace network on a per-namespace basis and all pods and services created within that namespace use that network as the pod or service network. The `Namespace` annotation `net.juniper.contrail.podnetwork: network-namespace/network-name` designates the desired network as the custom namespace network.

The following YAML shows an example of namespace with a custom namespace network annotation.

```
apiVersion: v1
kind: Namespace
metadata:
  name: custom-podnet
  annotations:
    net.juniper.contrail.podnetwork: custom-podnet/ns-level-custom-podnets
```

NOTE: The annotation of a namespace must be present when you create the namespace. You cannot update the annotation on a namespace to change its network. You must recreate the namespace to change its network. If `kubemanager` detects an update to the Custom Namespace Network annotation within a namespace, `kubemanager` flags that namespace and any pod created within that namespace after the update does not start. Reverting the update removes the flag and the pods launch normally.

Apply a Custom Service Network to a Service

You can specify a custom default network for services in the `annotations` of a `Service` object. As a result, the service can select pods that use the custom service network. Services that select pods with a custom network are isolated from other networks. The annotation format is: `network-namespace/network-name`.

The following is an example of a `Service` with a custom service network.

```
apiVersion: v1
kind: Service
metadata:
  name: custom-podnet-svc
  namespace: custom-podnet
  annotations:
    net.juniper.contrail.podnetwork: custom-podnet/pod-level-custom-podnet
```

Designate a Virtual Network as a Custom Default Pod Network

You can designate a VN as a custom default pod network. If you manually create a `VirtualNetwork` object, you must set the field "`podNetwork: true`" on the VirtualNetwork's spec. This field designates the new VN as the custom default pod network. CN2 assigns IPs to pods from this network.

The following is an example of a `VirtualNetwork` designated as a custom default pod network.

```
apiVersion: core.contrail.juniper.net/v2
kind: VirtualNetwork
metadata:
  namespace: custom-podnet
  name: vn-network
spec:
  podNetwork: true
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v2
    kind: Subnet
    namespace: custom-podnet
    name: vn-network-subnet
```

Use a NAD to Create a Custom Default Pod Network

Install a Network Attachment Definiton (NAD) with `podNetwork: true` set at the `juniper.net/networks` annotation to create a custom default pod network. After you create this NAD, the NAD controller automatically creates a VN and sets the "`podNetwork: true`" field during `VirtualNetwork` creation.

The following is an example of a NAD designated as a custom default pod network.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: nad-network
  namespace: custom-podnet
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "5.5.5.0/24",
```

```
    "podNetwork": true
  }'
```

- **NOTE**: You don't need to specify a NAD annotation for pods within a namespace if this namespace uses the same network as the primary interface (eth0). In other words, since the namespace specifies a network with an annotation, pods within that namespace don't need to specify that same network.

Deploy a Custom Pod Network per Pod

You can also specify a custom default pod network per pod. In order to designate a custom pod network on a per-pod basis, you must specify the following key value pair in the `annotations` of a pod: `"net.juniper.contrail.podnetwork": network-namespace/network-name`. This feature provides pod isolation because pods that use a custom pod network are isolated from other networks.

The following example shows a namespace, `VirtualNetwork`, and pod with custom default networks defined.

```
apiVersion: v1
kind: Namespace
metadata:
  name: cpn-intra-network-test-vn
  annotations:
    net.juniper.contrail.podnetwork: cpn-intra-network-test-vn/vn
---
apiVersion: core.contrail.juniper.net/v1
kind: Subnet
metadata:
  namespace: cpn-intra-network-test-vn
  name: vn-sn
spec:
  cidr: 15.15.15.0/24
---
apiVersion: core.contrail.juniper.net/v1
kind: VirtualNetwork
metadata:
  namespace: cpn-intra-network-test-vn
  name: vn
```

```
spec:
  podNetwork: true
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v1
    kind: Subnet
    namespace: cpn-intra-network-test-vn
    name: vn-sn
---
apiVersion: v1
kind: Pod
metadata:
  name: ns-level-cpn-pod
  namespace: cpn-intra-network-test-vn
spec:
  containers:
    - name: toolbox
      image: <repository>:<tag>
      imagePullPolicy: IfNotPresent
      command: ["bash", "-c", "while true; do sleep 60s; done"]
      securityContext:
        capabilities:
          add:
            - NET_ADMIN
        privileged: true
  tolerations:
    - key: "key"
      operator: "Equal"
      value: "value"
      effect: "NoSchedule"
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-level-cpn-pod
  namespace: cpn-intra-network-test-vn
  annotations:
    net.juniper.contrail.podnetwork: cpn-intra-network-test-vn/vn # namespace/name of the network
    net.juniper.contrail.podnetwork.ip: 15.15.15.3 # ip request for the interface
    net.juniper.contrail.podnetwork.cni-args: | # any cni-args needed for this interface
spec:
  containers:
    - name: toolbox
      image: <repository>:<tag>
```

```
      imagePullPolicy: IfNotPresent
      command: ["bash", "-c", "while true; do sleep 60s; done"]
      securityContext:
        capabilities:
          add:
            - NET_ADMIN
        privileged: true
  tolerations:
    - key: "key"
      operator: "Equal"
      value: "value"
      effect: "NoSchedule"
```

Note the following pod `annotations`:

```
  annotations:
    net.juniper.contrail.podnetwork: cpn-mc-cc/cpn1 # namespace/name of the network
    net.juniper.contrail.podnetwork.ip: 100.100.100.3 # ip request for the interface
    net.juniper.contrail.podnetwork.cni-args: | # any cni-args needed for this interface
      {
        "net.juniper.contrail.interfacegroup": "eth0"
      }
```

You define a custom pod network in the annotations of both the pod (`pod-level-cpn-pod`) and namespace (`cpn-intra-network-test-vn`). As part of the 23.1 release, you must configure a pod to use a custom pod network in the `annotations` at the pod level. If you specify an annotation at both the pod and namespace level, the pod-level annotation takes priority.

Custom Default Namespace Network Interactions

See the following sections for information about common custom namespace network interactions.

Multi-NIC Pod

A pod with a custom default namespace network may still contain multiple interfaces. The following YAML is an example of a `NetworkSelectionElement` of a pod with a custom network and multiple interfaces.

```
apiVersion: v1
kind: Namespace
metadata:
  name: custom-podnet
  annotations:
    net.juniper.contrail.podnetwork: cpn-intra-network-test/vn
---
# Pod
  annotations:
    k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "vn1",
          "namespace": "vn1-ns",
        },
        {
          "name": "vn2",
          "namespace": "vn2-ns",
        }
      ]
```

A pod with the annotations defined above would utilize eth0 from `custom-podnet-vn`, eth1 from `vn1`, and eth2 from `vn2`. You can also create a namespace with a custom namespace network and multiple interfaces. The following YAML shows the configuration above, replicated at the namespace level.

```
apiVersion: v1
kind: Namespace
metadata:
  name: vn0-ns
  annotations:
    net.juniper.contrail.podnetwork: vn0-ns/vn0
---
# Pod
  annotations:
    k8s.v1.cni.cncf.io/networks: |
      [
```

```
    {
      "name": "vn1",
      "namespace": "vn1-ns",
    },
    {
      "name": "vn2",
      "namespace": "vn2-ns",
    }
  ]
```

Deploy Kubevirt DPDK Dataplane Support for VMs

SUMMARY

Cloud-Native Contrail® Networking™ supports the deployment of the vRouter DPDK dataplane (Kubevirt) for high-performance VM and container networking in Kubernetes.

IN THIS SECTION

- Kubevirt Overview | 185
- Kubevirt DPDK Implementation | 186
- Deploy Kubevirt | 186
- Prerequisites | 187
- Pull Kubevirt Images and Deploy Kubevirt Using a Local Registry | 187
- Launch a VM Alongside a Container | 187
- Launch a VM | 187
- Create a Virtual Network | 192
- Launch a VM | 193

Kubevirt Overview

Kubevirt is an open-source Kubernetes project that enables the management (scheduling) of virtual machine (VM) workloads alongside container workloads within a Kubernetes cluster. Kubevirt provides a unified development platform where developers build, modify, and deploy applications residing in both application containers and VMs within a common, shared environment.

Kubevirt provides the following additional functions to your Kubernetes cluster by adding:

- Other types of pods, or Custom Resource Definitions (CRDs), to the Kubernetes API server.
- Controllers for cluster-wide logic to support the new types of pods.
- Daemons for node-specific logic to support the new types of pods.

As a result of this new functionality, Kubevirt creates and manages `VirtualMachineInstance` (VMI) objects. VMIs contain a workload controller called a `VirtualMachine` (VM). The VM maintains the persistent state of its VMI. This process enables users to terminate and initiate VMs at another time with no change in data or state. Additionally, you can deploy Kubevirt on top of a Kubernetes cluster, which lets you manage traditional container workloads along with VMIs managed by Kubevirt. VMs have access to Kubernetes cluster features with no additional permissions required.

Kubevirt DPDK Implementation

Kubevirt does not typically support user space networking for fast packet processing. In Cloud-Native Contrail Networking however, enhancements enable Kubevirt to support `vhostuser` interface types for VMs. These interfaces perform user space networking with the Data Plane Development Kit (DPDK) vRouter and give pods access to the increased performance and packet processing the DPDK vRouter provides.

Following are some of the benefits of the DPDK vRouter application:

- Packet processing occurs in user space and bypasses kernel space. This bypass increases packet-processing efficiency.
- Kernel interrupts and context switches do not occur because packets bypass kernel space. This bypass results in less CPU overhead and increased data throughput.
- DPDK enhances the forwarding plane of the vRouter in user space, increasing performance.
- DPDK Lcores run in poll mode. This mode enables the Lcores to receive and process packets immediately upon receiving them.

Deploy Kubevirt

Prerequisites

You must have an active Kubernetes cluster and the ability to use the `kubectl` client in order to deploy Kubevirt.

Pull Kubevirt Images and Deploy Kubevirt Using a Local Registry

See the following topic for information about how to deploy Kubevirt: "Pull Kubevirt Images and Deploy Kubevirt Using a Local Registry" on page 197.

NOTE: These instructions are for the following Kubevirt releases:

- v0.58.0 (current)
- v0.48.0

Launch a VM Alongside a Container

With Kubevirt, launching and managing a VM in Kubernetes is similar to deploying a pod. You can create a VM object using `kubectl`. After creating a VM object, that VM is active and running in your cluster.

Use the following high-level steps to launch a VM alongside a container:

1. Create a `VirtualNetwork`.
2. Launch a VM.

Launch a VM

The following `VirtualMachine` specs are examples of `VirtualMachine` instances with a varying number of interfaces.

- Single `vhostuser` interface VM:

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
```

```
metadata:
  name: vm-single-virtio
  namespace: contrail
spec:
  running: true
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: vm-single-virtio
        app: vm-single-virtio-app
    spec:
      nodeSelector:
        master: master
      terminationGracePeriodSeconds: 30
      domain:
        cpu:
          sockets: 1
          cores: 8
          threads: 2
          #dedicatedCpuPlacement: true
        memory:
          hugepages:
            pageSize: "2Mi"
        resources:
          requests:
            memory: "512Mi"
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
          interfaces:
          - name: default
            bridge: {}
          - name: vhost-user-vn-blue
            vhostuser: {}
          useVirtioTransitional: true
      networks:
      - name: default
```

```
        pod: {}
      - name: vhost-user-vn-blue
        multus:
          networkName: vn-blue
      volumes:
        - name: containerdisk
          containerDisk:
            image: svl-artifactory.juniper.net/atom-docker/dpdk-pktgen/vmdisks/dpdk-pktgen-
auto:latest
        - name: cloudinitdisk
          cloudInitNoCloud:
            userDataBase64: SGkuXG4=
```

- Multi `vhostuser` interface:

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: vm-multi-virtio
  namespace: contrail
spec:
  running: true
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: vm-multi-virtio
        app: vm-multi-virtio-app
    spec:
      nodeSelector:
        worker: worker
      terminationGracePeriodSeconds: 30
      domain:
        cpu:
          sockets: 1
          cores: 8
          threads: 2
          #dedicatedCpuPlacement: true
        memory:
          hugepages:
            pageSize: "2Mi"
        resources:
```

```
          requests:
            memory: "512Mi"
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
          interfaces:
          - name: default
            bridge: {}
          - name: vhost-user-vn-blue
            vhostuser: {}
          - name: vhost-user-vn-green
            vhostuser: {}
          useVirtioTransitional: true
      networks:
      - name: default
        pod: {}
      - name: vhost-user-vn-blue
        multus:
          networkName: vn-blue
      - name: vhost-user-vn-green
        multus:
          networkName: vn-green
      volumes:
        - name: containerdisk
          containerDisk:
            image: svl-artifactory.juniper.net/atom-docker/dpdk-pktgen/vmdisks/dpdk-pktgen-
auto:latest
        - name: cloudinitdisk
          cloudInitNoCloud:
            userDataBase64: SGkuXG4=
```

- Bridge/`vhostuser` interface VM:

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: vm-virtio-veth
```

```
  namespace: contrail
spec:
  running: true
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: vm-virtio-veth
        app: vm-virtio-veth-app
    spec:
      nodeSelector:
        master: master
      terminationGracePeriodSeconds: 30
      domain:
        cpu:
          sockets: 1
          cores: 8
          threads: 2
          #dedicatedCpuPlacement: true
        memory:
          hugepages:
            pageSize: "2Mi"
        resources:
          requests:
            memory: "512Mi"
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
          interfaces:
          - name: default
            bridge: {}
          - name: vhost-user-vn-blue
            vhostuser: {}
          - name: vhost-user-vn-green
            bridge: {}
          useVirtioTransitional: true
      networks:
      - name: default
```

```
        pod: {}
      - name: vhost-user-vn-blue
        multus:
          networkName: vn-blue
      - name: vhost-user-vn-green
        multus:
          networkName: vn-green
      volumes:
        - name: containerdisk
          containerDisk:
            image: svl-artifactory.juniper.net/atom-docker/dpdk-pktgen/vmdisks/dpdk-pktgen-
auto:latest
        - name: cloudinitdisk
          cloudInitNoCloud:
            userDataBase64: SGkuXG4=
```

Create a Virtual Network

The following `net-attach-def` object is an example of a virtual network:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vn-blue
  namespace: contrail
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "19.1.1.0/24"
    }'
  labels:
    vn: vn-blue-vn-green
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "nad-blue",
  "type": "contrail-k8s-cni"
}'
```

Launch a VM

The following `VirtualMachine` specs are examples of `VirtualMachine` instances with a varying number of interfaces:

- Single `vhostuser` interface VM:

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: vm-single-virtio
  namespace: contrail
spec:
  running: true
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: vm-single-virtio
        app: vm-single-virtio-app
    spec:
      nodeSelector:
        master: master
      terminationGracePeriodSeconds: 30
      domain:
        cpu:
          sockets: 1
          cores: 8
          threads: 2
          #dedicatedCpuPlacement: true
        memory:
          hugepages:
            pageSize: "2Mi"
        resources:
          requests:
            memory: "512Mi"
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: cloudinitdisk
```

```
            disk:
              bus: virtio
          interfaces:
          - name: default
            bridge: {}
          - name: vhost-user-vn-blue
            vhostuser: {}
          useVirtioTransitional: true
      networks:
      - name: default
        pod: {}
      - name: vhost-user-vn-blue
        multus:
          networkName: vn-blue
      volumes:
        - name: containerdisk
          containerDisk:
            image: svl-artifactory.juniper.net/atom-docker/dpdk-pktgen/vmdisks/dpdk-pktgen-
auto:latest
        - name: cloudinitdisk
          cloudInitNoCloud:
            userDataBase64: SGkuXG4=
```

- Multi `vhostuser` interface:

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: vm-multi-virtio
  namespace: contrail
spec:
  running: true
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: vm-multi-virtio
        app: vm-multi-virtio-app
    spec:
      nodeSelector:
        worker: worker
      terminationGracePeriodSeconds: 30
```

```
      domain:
        cpu:
          sockets: 1
          cores: 8
          threads: 2
          #dedicatedCpuPlacement: true
        memory:
          hugepages:
            pageSize: "2Mi"
        resources:
          requests:
            memory: "512Mi"
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
          interfaces:
          - name: default
            bridge: {}
          - name: vhost-user-vn-blue
            vhostuser: {}
          - name: vhost-user-vn-green
            vhostuser: {}
          useVirtioTransitional: true
      networks:
      - name: default
        pod: {}
      - name: vhost-user-vn-blue
        multus:
          networkName: vn-blue
      - name: vhost-user-vn-green
        multus:
          networkName: vn-green
      volumes:
        - name: containerdisk
          containerDisk:
            image: svl-artifactory.juniper.net/atom-docker/dpdk-pktgen/vmdisks/dpdk-pktgen-
auto:latest
        - name: cloudinitdisk
```

```
          cloudInitNoCloud:
            userDataBase64: SGkuXG4=
```

- Bridge/`vhostuser` interface VM:

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: vm-virtio-veth
  namespace: contrail
spec:
  running: true
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: vm-virtio-veth
        app: vm-virtio-veth-app
    spec:
      nodeSelector:
        master: master
      terminationGracePeriodSeconds: 30
      domain:
        cpu:
          sockets: 1
          cores: 8
          threads: 2
          #dedicatedCpuPlacement: true
        memory:
          hugepages:
            pageSize: "2Mi"
        resources:
          requests:
            memory: "512Mi"
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
```

```
          interfaces:
          - name: default
            bridge: {}
          - name: vhost-user-vn-blue
            vhostuser: {}
          - name: vhost-user-vn-green
            bridge: {}
          useVirtioTransitional: true
      networks:
      - name: default
        pod: {}
      - name: vhost-user-vn-blue
        multus:
          networkName: vn-blue
      - name: vhost-user-vn-green
        multus:
          networkName: vn-green
      volumes:
        - name: containerdisk
          containerDisk:
            image: svl-artifactory.juniper.net/atom-docker/dpdk-pktgen/vmdisks/dpdk-pktgen-
auto:latest
        - name: cloudinitdisk
          cloudInitNoCloud:
            userDataBase64: SGkuXG4=
```

Pull Kubevirt Images and Deploy Kubevirt Using a Local Registry

The current Kubevirt release (v0.58.0) doesn't support `imagePullSecret`. This field-located in a Kubernetes configuration file-tells Kubernetes where to pull credentials from in order to pull images from a secure registry (`enterprise-hub.juniper.net/contrail-container-prod/`). Juniper Cloud-Native Contrail® Networking™ (CN2) release 22.4 addresses this issue with a workaround for the older Kubevirt release (v048.0) and current release (v0.58.0).

See the following information about how to perform this workaround to pull images and deploy Kubevirt using a local registry.

1. Install Docker.

```
sudo curl -fsSL https://get.docker.com -o /tmp/get-docker.sh
sudo sh /tmp/get-docker.sh
```

2. Create a local registry.

```
sudo docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

3. Edit `/etc/docker/daemon.json` to add your local registry's `ip:port`.

```
# add the following to /etc/docker/daemon.json
#
#{
#  "local-registries" : ["your-computer-hostname:5000"]
#}
```

4. Restart Docker.

```
sudo service docker restart
```

5. Download the required containers. These containers are located at Release Userspace CNI - dpdk vhostuser interface support Juniper/kubevirt. The `kubevirt-operator.yaml` and `kubevirt-cr.yaml` are also located at this repository.

```
sudo wget -O /tmp/virt-api_v0.58.0-jnpr.tar.gz https://github.com/Juniper/kubevirt/releases/
download/v0.58.0-jnpr/virt-api_v0.58.0-jnpr.tar.gz
sudo wget -O /tmp/virt-controller_v0.58.0-jnpr.tar.gz https://github.com/Juniper/kubevirt/
releases/download/v0.58.0-jnpr/virt-controller_v0.58.0-jnpr.tar.gz
sudo wget -O /tmp/virt-handler_v0.58.0-jnpr.tar.gz https://github.com/Juniper/kubevirt/
releases/download/v0.58.0-jnpr/virt-handler_v0.58.0-jnpr.tar.gz
sudo wget -O /tmp/virt-launcher_v0.58.0-jnpr.tar.gz https://github.com/Juniper/kubevirt/
releases/download/v0.58.0-jnpr/virt-launcher_v0.58.0-jnpr.tar.gz
sudo wget -O /tmp/virt-operator_v0.58.0-jnpr.tar.gz https://github.com/Juniper/kubevirt/
releases/download/v0.58.0-jnpr/virt-operator_v0.58.0-jnpr.tar.gz
```

6. Load the required containers.

```
sudo docker load < /tmp/virt-api_v0.58.0-jnpr.tar.gz
sudo docker load < /tmp/virt-controller_v0.58.0-jnpr.tar.gz
sudo docker load < /tmp/virt-handler_v0.58.0-jnpr.tar.gz
sudo docker load < /tmp/virt-launcher_v0.58.0-jnpr.tar.gz
sudo docker load < /tmp/virt-operator_v0.58.0-jnpr.tar.gz
```

7. Tag and push the containers to your local registry.

   Replace `<LOCAL_REGISTRY>` with your local registry. For example, if the containers are hosted at `10.84.13.52:5000/kubevirt`, replace `<LOCAL_REGISTRY>` with `10.84.13.52:5000/kubevirt`.

```
sudo docker tag svl-artifactory.juniper.net/atom-docker/kubevirt/virt-api:v0.58.0-jnpr
<LOCAL_REGISTRY>/virt-api:v0.58.0-jnpr
sudo docker push <LOCAL_REGISTRY>/virt-api:v0.58.0-jnpr
sudo docker tag svl-artifactory.juniper.net/atom-docker/kubevirt/virt-controller:v0.58.0-
jnpr <LOCAL_REGISTRY>/virt-controller:v0.58.0-jnpr
sudo docker push <LOCAL_REGISTRY>/virt-controller:v0.58.0-jnpr
sudo docker tag svl-artifactory.juniper.net/atom-docker/kubevirt/virt-handler:v0.58.0-jnpr
<LOCAL_REGISTRY>/virt-handler:v0.58.0-jnpr
sudo docker push <LOCAL_REGISTRY>/virt-handler:v0.58.0-jnpr
sudo docker tag svl-artifactory.juniper.net/atom-docker/kubevirt/virt-launcher:v0.58.0-jnpr
<LOCAL_REGISTRY>/virt-launcher:v0.58.0-jnpr
sudo docker push <LOCAL_REGISTRY>/virt-launcher:v0.58.0-jnpr
sudo docker tag svl-artifactory.juniper.net/atom-docker/kubevirt/virt-operator:v0.58.0-jnpr
<LOCAL_REGISTRY>/virt-operator:v0.58.0-jnpr
sudo docker push <LOCAL_REGISTRY>/virt-operator:v0.58.0-jnpr
```

8. Download the `kubevirt-operator.yaml` and `kubevirt-cr.yaml`.

```
wget https://github.com/Juniper/kubevirt/releases/download/v0.58.0-jnpr/kubevirt-
operator.yaml
wget https://github.com/Juniper/kubevirt/releases/download/v0.58.0-jnpr/kubevirt-cr.yaml
```

9. Modify the `kubevirt-operator.yaml`.

   Replace `<LOCAL_REGISTRY>` with your local registry. For example, if the containers are hosted at `10.84.13.52:5000/kubevirt`, replace `<LOCAL_REGISTRY>` with `10.84.13.52:5000/kubevirt`.

10. Modify `/etc/crio/crio.conf` in all Kubernetes nodes in the cluster. Add the following to the `crio.conf` in all of the Kubernetes nodes in the cluster. These commands allow cri-o (Container Runtime Interface-Open Container Initiative) to pull images from your local registry.

    NOTE: The cri-o service is a version of the Kubernetes container runtime interface (CRI) that enables the use of Open Container Initiative (OCI) compatible runtimes.

    ```
    insecure_registries = ["10.92.81.91/22"]
                registries = ["10.92.81.91:5000"]

                Where 10.92.81.91 is the ip of <LOCAL_REGISTRY>
    ```

11. After modifying the `crio.conf`, restart the service.

    ```
    service crio restart
    ```

RELATED DOCUMENTATION

Deploy Kubevirt DPDK Dataplane Support for VMs | 185

Static Routes

SUMMARY

Juniper Cloud-Native Contrail Networking (CN2) release 23.1 supports static routes for your cluster. This article provides information about how to configure static routes for your CN2 cluster.

IN THIS SECTION

- Understanding Static Routes | 201
- Static Routes in CN2 | 201
- Configure Static Routes for a Virtual Network | 203
- Configure Static Routes for a VMI | 203

- Configure Static Routes on Pod Interfaces | 204
- Configure Static Routes for a Virtual Network with a NAD | 206
- Multiple Static Routes on Pod Interfaces | 206
- Troubleshooting RouteTable and InterfaceRouteTable | 209
- Config Plane Verification | 209
- Dataplane Verification | 210

Understanding Static Routes

You can use static routes when a network doesn't require the complexity of a dynamic routing protocol. Routes that are permanent fixtures in routing and forwarding tables are often configured as static routes. The internal traffic from stub networks benefits from static routes.

The route consists of a destination prefix and a next-hop forwarding address. The static route is activated in the routing table and inserted into the forwarding table when the next-hop address is reachable. Traffic that matches the static route is forwarded to the specified next-hop address.

Static Routes in CN2

CN2 implements static routes through the following two custom resources (CRs):

- `RouteTable`: Contains a user-defined next hop destination (`nextHop`), along with a destination prefix to identify next hop traffic. The `nextHop` IP address must be an IP address of another VMI object. A prefix defines the destination network which acts as the next hop for matching traffic. A `RouteTable` lets you define a static route. You can associate a `RouteTable` with a virtual network (VN). The following is an example of a `RouteTable` CR:

```
apiVersion: core.contrail.juniper.net/v3
kind: RouteTable
metadata:
  name: static-rt
  namespace: static-route
```

```
spec:
    routes:
      route:
        - nextHop: 10.20.30.2
          nextHopType: ip-address
          prefix: 10.20.30.0/24
          communityAttributes:
            communityAttribute:
                - accept-own
                - no-advertise
```

Note that the field `nextHopType` must have the value `ip-address`. Any other value results in a user input error. The `communityAttributes` field enables you to control route learning via BGP.

- `InterfaceRouteTable`: The `InterfaceRouteTable` configures static routing for a virtual machine interface (VMI). An `InterfaceRouteTable` contains the destination prefix without the need for a next hop entry. As with a `RouteTable`, the prefix defines the destination network, or next hop. Unlike a `RouteTable`, you do not need to define a `nextHop` IP address because when you associate an `InterfaceRouteTable` with a VMI, the associated VMI acts as the next hop for this prefix.

  The following is an example of an `InterfaceRouteTable` CR:

```
apiVersion: core.contrail.juniper.net/v3
kind: InterfaceRouteTable
metadata:
  name: static-rt
  namespace: static-route
spec:
    interfaceRouteTableRoutes:
       route:
         - nextHopType: ip-address
           prefix: 10.20.30.0/24
           communityAttributes:
              communityAttribute:
                - accept-own
```

  Note that the field `nextHopType` must have the value `ip-address`. Any other value results in a user input error.

These CRs are scoped to their respective namespaces and enable you to configure required attributes for static routes.

Configure Static Routes for a Virtual Network

Configure the `RouteTable` CR to apply static routes to a VN. A VN references a `RouteTable` in it's spec. As a result, the `RouteTable` is associated with that VN and the static route is configured. The following is a VN object with an associated `RouteTable`:

```
apiVersion: core.contrail.juniper.net/v3
kind: VirtualNetwork
metadata:
  namespace: static-route
  name: vn-route
spec:
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v1
    kind: Subnet
    namespace: static-route
    name: vn-subnet
  routeTableReferences:
    - apiVersion: core.contrail.juniper.net/v3
      kind: RouteTable
      namespace: static-route
      name: static-rt
```

Configure Static Routes for a VMI

Configure an `InterfaceRouteTable` to apply static routes to a VMI. A VMI references an `InterfaceRouteTable` in it's `InterfaceRouteTableReference` section. The following is a VMI object with a reference to an `InterfaceRouteTable`:

```
apiVersion: v3
kind: VirtualMachineInterface
metadata:
  name: static-route-pod
  namespace: static-route
  annotations:
     core.juniper.net/interface-route-table: '[{"name": "static-rt", "namespace": "static-
route"}]'

spec:
```

```
    <VMI_SPEC>

status:
   interfaceRouteTableReferences:
    - apiVersion: core.contrail.juniper.net/v3
      kind: InterfaceRouteTable
      namespace: static-route
      name: static-rt
```

Configure Static Routes on Pod Interfaces

You can use the annotation section of a pod's manifest to configure static routes for a pod's default or secondary interface. The pod reconciler processes the annotation section to create a VMI object with an associated `InterfaceRouteTable`. The reconciler looks for the string key: "core.juniper.net/interface-route-table" in the annotation section. The pod's VMI uses that string as a metadata label to associate with an `InterfaceRouteTable`.

The following is an example of a pod manifest with an `InterfaceRouteTable` defined for the default interface:

```
apiVersion: v1
kind: Pod
metadata:
  name: static-route-pod
  namespace: static-route
  annotations:
    core.juniper.net/interface-route-table: '[{"name": "vmi-rt", "namespace": "static-route"}]'
spec:
  containers:
    - name: praqma
      image: <image-repository>:<tag>
      imagePullPolicy: Always
      securityContext:
        capabilities:
          add:
            - NET_ADMIN
        privileged: true
```

The following is an example of a pod manifest with an `InterfaceRouteTable` defined for the secondary interface:

```
apiVersion: v1
kind: Pod
metadata:
  name: static-route-pod
  namespace: static-route
  annotations:
    k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "vn-route",
          "namespace": "static-route",
          "cni-args": {
            "core.juniper.net/interface-route-table": "[{\"name\": \"vmi-rt\", \"namespace\":
"static-route\"}]"
          }
        }
      ]
spec:
  containers:
    - name: praqma
      image: <image-repository>:<tag>
      imagePullPolicy: Always
      securityContext:
        capabilities:
          add:
            - NET_ADMIN
        privileged: true
```

Note that the `name` for the primary interface `InterfaceRouteTable` is `vmi-rt` and that the `name` for the secondary interface is `vn-route`. Defining two `InterfaceRouteTables` with different `names` in the same `namespace` automatically creates an `InterfaceRouteTable` for the primary and secondary interface of that pod.

Configure Static Routes for a Virtual Network with a NAD

You can also specify static route properties in a network attachment definition (NAD) object. After the NAD is reconciled or applied, a `RouteTable` is created and the resulting VN object references that `RouteTable`. The following is an example of a NAD with static route information defined:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vn-route
  namespace: static-route
  labels:
    vn: vn-route
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "108.108.2.0/24"
      "routeTableReferences": '[{"name": "vn-rt", "namespace": "static-route"}]'
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "vn-route",
  "type": "contrail-k8s-cni"
}'
```

Multiple Static Routes on Pod Interfaces

Using `InterfaceRouteTable`, you can associate multiple static routes to a single pod interface (VMI). This means that that VMI object has multiple default next hop destinations, depending on the IP prefix. You can specify multiple `InterfaceRouteTable` references using cluster service version (CSV) syntax or JSON syntax `annotations`.

NOTE: You must reference an `InterfaceRouteTable` in a "namespace/name" format. In the following example, `static-route` is the namespace and `to-right` and `to-zone-1` are the `InterfaceRouteTable` objects, or next hop destination for the `left-vn` VMI.

The following example is a `Deployment` with multiple `InterfaceRouteTable` references:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: forwarder
  namespace: static-route
  labels:
    app: forwarder
spec:
  replicas: 3
  selector:
    matchLabels:
      app: forwarder
  template:
    metadata:
      labels:
        app: forwarder
      annotations:
        k8s.v1.cni.cncf.io/networks: |
          [
           {
             "name": "left-vn",
             "namespace": "static-route",
             "cni-args": {
               "core.juniper.net/interface-route-table": "static-route/to-right,static-route/to-
zone-1"
             }
           },
           {
             "name": "right-vn",
             "namespace": "static-route",
             "cni-args": {
               "core.juniper.net/interface-route-table": "static-route/to-left"
             }
           },
           {
             "name": "zone-1",
             "namespace": "static-route",
             "cni-args": {
               "core.juniper.net/interface-route-table": "static-route/to-left"
             }
```

```
          },
          {
            "name": "zone-2",
            "namespace": "static-route",
             "cni-args": {
              "core.juniper.net/interface-route-table": "static-route/to-left"
            }
          }
         ]
    spec:
      containers:
        - name: praqma
          image: <repository>:<tag>
          securityContext:
            capabilities:
              add:
                - NET_ADMIN
            privileged: true
```

The following example is a pod manifest with multiple `InterfaceRouteTable` references using JSON syntax:

```
apiVersion: v1
kind: Pod
metadata:
  name: irt-right
  namespace: static-route
  annotations:
    k8s.v1.cni.cncf.io/networks: |
      [{
          "name": "right-vn",
          "namespace": "static-route",
          "cni-args": {
               "core.juniper.net/interface-route-table": "[{\"namespace\": \"static-route\",
"\name\": \"to-left\"}, {\"namespace\": \"static-route\", \"name\": \"to-zone-1\"}]"
            }
      }]
spec:
  containers:
    - name: praqma
      image: <image-repository>:<tag>
      securityContext:
        capabilities:
```

```
      add:
        - NET_ADMIN
    privileged: true
```

NOTE: You must use backward slashes in JSON syntax. Backward slashes are required to encode a JSON string inside another JSON string.

Troubleshooting RouteTable and InterfaceRouteTable

The following sections contain useful commands when troubleshooting various `RouteTable` and `InterfaceRouteTable` issues.

Config Plane Verification

- Verify the state of the `RouteTable` and `InterfaceRouteTable` objects.
  - Check the status of the reconciler for the `InterfaceRouteTable` object.

    ```
    kubectl get interfaceroutetable  -n
    ```

  - Check the status of the reconciler for the `RouteTable` object.

    ```
    kubectl get routetable  -n
    ```

- Verify the `RouteTable` reference in the associated VN. Verify the `InterfaceRouteTable` reference in the associated VMI.

- Check the status of the reconciler for the VMI. You should see the `InterfaceRouteTable` in the VMI with an associated universally unique identifier (UUID) the Contrail FQ (meta info such as `apiversion`, `kind`, `namespace`, `name`) name.

```
kubectl get vmi  -n  -oyaml | grep -i interfaceRouteTable
```

```
kubectl get vn  -n  -oyaml | grep -i routeTable
```

Dataplane Verification

- In the introspect, verify that the VRF of the VN shows a row with a matching static route prefix specified in the RT using the following steps:
  - Verify that the VRF is associated with the VN.

    https://%3Cvroute_ip%3E:8085/Snh_VrfListReq
  - Navigate to the ucindex column in the VRF unicast `RouteTable`.
  - Verify that the table contains a row with the correct static route prefix.
- In the introspect, verify that the next hop properties of the VN are valid. In the introspect, the next hop column for the prefix should contain the following:
  - The next hop interface name must be a valid tap interface.
  - The `label` must be a positive integer.
  - The `resolved` value must be `true`.
  - The `route-type:` value must be `InterfaceStaticRoute`.

VPC to CN2 Communication in AWS EKS

SUMMARY

Juniper Cloud-Native Contrail Networking (CN2) release 23.1 supports communication between AWS virtual private cloud (VPC) networks, external networks, and CN2 clusters. This feature only applies to AWS EKS environments using CN2 as the CNI. This article provides information about how CN2 implements this feature.

IN THIS SECTION

- Understanding Kubernetes and VPC Networks | 211
- Prerequisites | 212
- Gateway Service Instance Components | 212
- Custom Resource Implementation | 216
- Custom Controller Implementation | 218
- Troubleshooting | 218

Understanding Kubernetes and VPC Networks

Typically, you cannot access a Kubernetes workload in an overlay network running on Amazon Elastic Kubernetes Service (EKS) from a VPC. In order to achieve AWS VPC to Kubernetes communication, you must expose the host network of your Kubernetes cluster to the VPC. Although some public cloud Kubernetes distributions offer solutions that support this feature, these solutions are tailored for traditional VM workloads instead of Kubernetes workloads. As a result, these solutions have the following drawbacks:

- You must configure pod IPs as secondary IP addresses on node interfaces. This imposes resource constraints on the nodes, reducing the number of pods that can be supported.
- Services are exposed through public Load Balancers. Every time a service is created, the Load Balancer begins an instantiation process which might result in more time until service exposure

CN2 release 23.1 addresses this issue by introducing a Gateway Service Instance (GSI). A GSI is a collection of Amazon Web Service (AWS) and Kubernetes resources that work together to seamlessly interconnect CN2 with VPC and external networks. Apply a GSI manifest and CN2 facilitates communication between pods and services in an Amazon EKS cluster and workloads in the same VPC.

Prerequisites

The following are required to enable VPC to CN2 communication:

- A license for cRPD. To purchase a license, visit https://www.juniper.net/us/en/products/routers/containerized-routing-protocol-daemon-crpd.html
  - You must install the license into the EKS cluster as a `Secret` within the `contrail-gsi` namespace.
  - The secret must contain the base64-encoded version of the license under the crpd-license key of .Data.
  - The following is an example `Secret` with a reference to a license:

```
apiVersion: v1
kind: Secret
metadata:
  name: crpd-license
  namespace: contrail-gsi
data:
  crpd-license: ****** # base64 encoded crpd license
```

- An EKS cluster running CN2 release 23.1 or later
- AWS Identity and Access Management (IAM) role access. See the following link for instructions about how to configure a service account to assume an IAM role: "Configure a Service Account to Assume an IAM role" on page 220
- Nodes within the EKS cluster must have the label: `core.juniper.net/crpd-node: ""`.
  - The controller will only schedule the cRPD container on nodes with the following key: `core.juniper.net/crpd-node`. After this label is added onto a node, none of the CN2 controllers (including the vRouter) will run on this node. This ensures that the proper amount of nodes are reserved for cRPD containers.

Gateway Service Instance Components

The following is an example of a gateway service instance (GSI) manifest:

```
apiVersion: v1
kind: Namespace
```

```
metadata:
  name: contrail-gsi
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: contrail-gsi-serviceaccount
  namespace: contrail-gsi
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: contrail-gsi-role
rules:
- apiGroups:
  - '*'
  resources:
  - '*'
  verbs:
  - '*'
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: contrail-gsi-rolebinding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: contrail-gsi-role
subjects:
- kind: ServiceAccount
  name: contrail-gsi-serviceaccount
  namespace: contrail-gsi
---
apiVersion: plugins.juniper.net/v1
kind: GSIPlugin
metadata:
  name: contrail-gsi-plugin
  namespace: contrail-gsi
spec:
  awsRegion: "us-east-2"
  iamRoleARN: "arn:aws:iam::********"
  vpcID: "vpc-**********"
```

```
common:
  containers:
  - image: <repository>/contrail-gsi-plugin:<tag>
    imagePullPolicy: Always
    name: contrail-gsi-plugin
  initContainers:
  - command:
    - kubectl
    - apply
    - -k
    - /crd
    image: <repository>/contrail-gsi-plugin-crdloader:<tag>
    imagePullPolicy: Always
    name: contrail-gsi-plugin-crdloader
  serviceAccountName: contrail-gsi-serviceaccount
```

Note that the `awsRegion`, `iamRoleARN`, and `vpcID` fields are user-defined. The `iamRoleARN` value is the Amazon Resource Name (ARN) of the IAM Role that you create as part of the prerequisites for this feature.

Applying a GSI manifest creates a custom controller that creates and manages the following:

AWS resources:

- Transit gateway: A transit gateway is a network transit hub that can interconnect VPCs. A transit gateway can have `Attachments` which are one or more VPCs.

- Connect attachment: A transit gateway connect attachment establishes a connection between a transit gateway and third-party virtual appliances (JCNRs) running in a VPC. After you create a connect attachment, one or more Generic Routing Encapsulation (GRE) tunnels, or Transit Gateway Connect peers, can be created on the Connect attachment to connect the transit gateway and the third-party appliance. A Transit Gateway Connect peer is comprised of two BGP peering sessions over the GRE tunnel which provide routing redundancy. After you install the transit gateway resource, CN2 performs this process automatically.

- VPC attachment: A VPC `Attachment` attaches to a transit gateway. When you attach a VPC to a transit gateway, resource and routing rules apply to that gateway.

Kubernetes resource:

- Connected peers: A transit gateway connect peer is a GRE tunnel that facilitates communication between a transit gateway and a third-party appliance or JCNR.

CN2 GSI resource:

- Juniper Cloud-Native Router (JCNR): JCNR is an extension of CN2 that acts as a gateway between EC2 instances and other AWS resources and the EKS cluster running CN2.

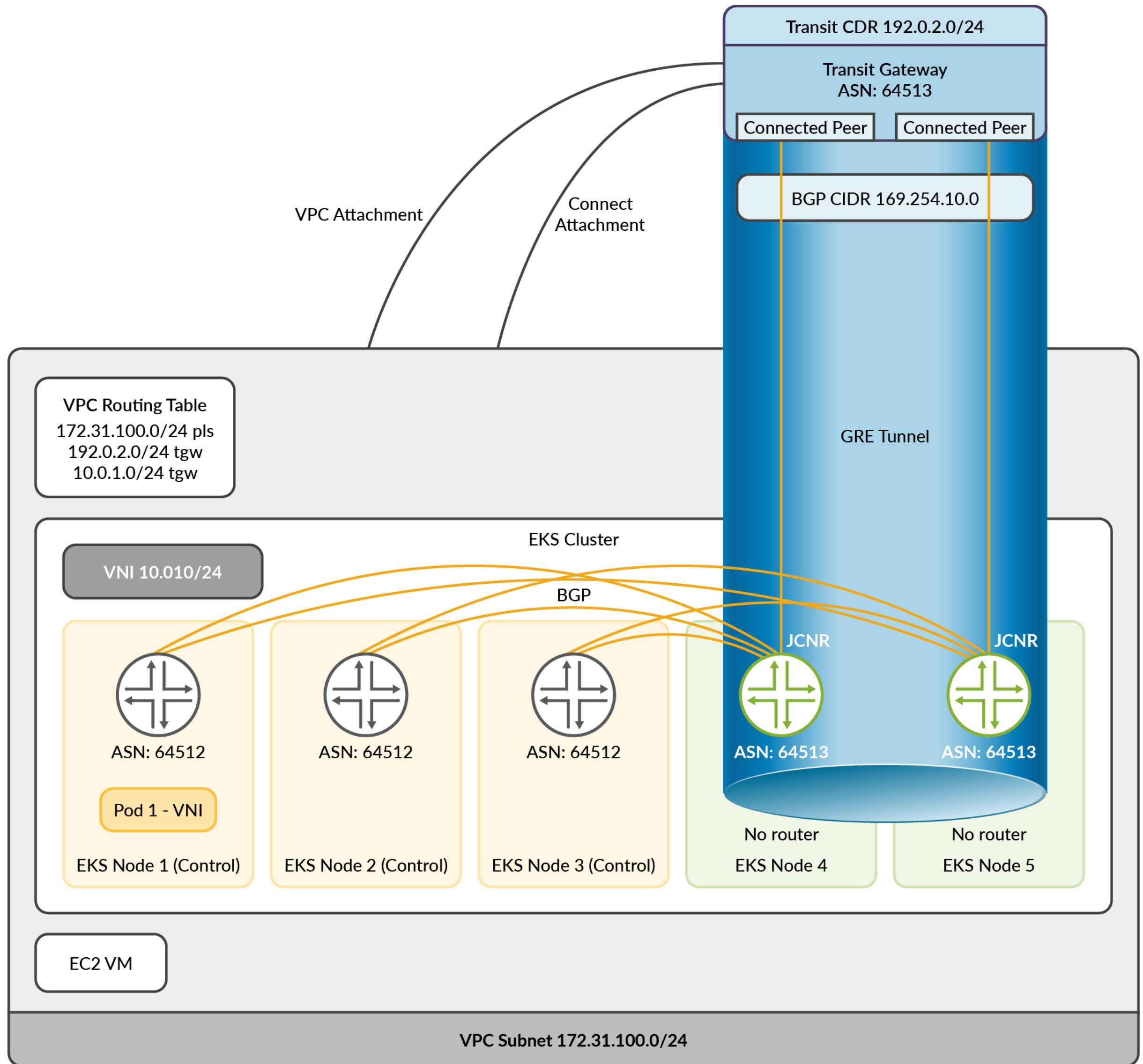
JCNRs do the following:

- Connect via Multiprotocol BGP (MP-BGP) to CN2 control nodes and use MPLSoUDP in the data plane
- Provide active/active L3 and L4 load balancing across the CN2 EKS nodes

You can scale JCNRs and their connected peers to a maximum of four instances. In this case, a transit gateway provides active/active L3 and L4 load balancing across the JCNRs. As an example of JCNR functionality, if you add a `label` onto a VN that you want to expose, the JCNR advertises that VN's subnet to the transit gateway. A next hop route between the VN's subnet and the transit gateway is added to the VPC's routing table. As a result, the VN is routed in the VPC's network and accessible via workloads within the VPC. This functionality is similar to a traditional physical SDN gateway, where the transit gateway and the JCNR act as a physical gateway between the EKS cluster and the rest of the AWS environment without the drawbacks of `hostNetwork`.

The following illustration depicts connectivity in a typical Amazon EKS cluster. Note that CN2 automatically generates the VPC Routing Table.

Figure 9: EKS Cluster Traffic Flow



Custom Resource Implementation

CN2 implements this feature through the following custom resources (CRs):

- `TransitGateway`: Represents the AWS `TransitGateway` resource.

- `Route`: Represents an entry in a VPC routing table. A `Route` is an internally-created object that doesn't require user input.
- `ConnectedPeer`: Represents an appliance (JCNR in CN2) that establishes BGP sessions with the transit gateway. A `ConnectedPeer` is an internally-created object that doesn't require user input.

The following is an example of a `TransitGateway` manifest.

```
apiVersion: core.gsi.juniper.net/v1
kind: TransitGateway
metadata:
  name: tgw1
  namespace: contrail-gsi
spec:
  subnetIDs:
    - subnet-*******
    - subnet-*******
    - subnet-*******
  connectedPeerScale: 1
  transitASN: 64513
  peerASN: 64513
  transitCIDR: 192.0.2.0/24
  bgpCidr: 169.254.10.0
  controlNodeASN: 64512
  licenseSecretName: crpd-license
  controllerContainer:
    name: controller
    command: ["/manager", "-mode=client"]
    image: <repository>/contrail-gsi-plugin:<tag>
    imagePullPolicy: Always
  crpdContainer:
    name: crpd
    image: <repository>/crpd:<tag>
  initContainer:
    name: init
    image: <repository>/busybox:<tag>
```

Use the following command for detailed information about the `TransitGateway` spec:

```
kubectl explain transitgateway.spec
```

Custom Controller Implementation

The GSI is implemented through the use of custom Kubernetes controllers with a client/server plugiin. The server-side custom controllers run on the CN2 control plane nodes. The client runs alongside JCNR. These controllers are automatically configured when you apply the `TransitGateway` and GSI objects.

Troubleshooting

This section provides information about troubleshooting various `ConnectedPeer` connectivity issues, custom controller issues, and workload reachability issues.

For custom controller issues:

- After you apply the GSI manifest, ensure that all of the custom controller pods associated with the GSI manifest (Server, Client) are active.

  ```
  kubectl get pods -n contrail-gsi -l app=contrail-gsi-plugin
  ```

  The CLI output should show three active controller pods in a regular EKS deployment. If you don't receive the correct CLI output, use the following command to verify that the GSI plugin is installed.

  ```
  kubectl get gsiplugins -n contrail-gsi
  ```

  Check the logs of the `contrail-k8s-deployer` pod. Filter results for the GSI plugin reconciler and look for any errors.

- Ensure that the custom controllers can make create, read, update, delete (CRUD) requests to the required AWS resources.

  - Check the logs of the `contrail-gsi` pods; one of the three pods should be active and will output log messages.

  - Verify that the logs of the active pod don't contain errors about not being able to make API calls to AWS. If you do see these errors, ensure that the IAM role granted to the `contrail-gsi-serviceaccount` is configured properly (refer to the "Prerequistes" on page 212 section of this topic).

For `TransitGateway` issues:

- After you install the `TransitGateways`, ensure that their statuses change from "pending" to "available."

- Ensure that the `ConnectedPeer`/cRPD pod is active.

```
kubectl get pods -n contrail-gsi -l app=connectedpeer
```

  If no pods show up in the output, ensure that a node is available for the `ConnectedPeer` pod to be scheduled on. A valid pod contains the following label: `core.juniper.net/crpd-node: ""`.

For `ConnectedPeer` issues:

- After a `ConnectedPeer` pod is active, ensure that the cRPD can establish BGP sessions with CN2 control nodes, and AWS's transit gateway. Run the following command in the CLI of the cRPD:

```
show bgp summary
```

  This process might fail for the following reasons:

  - A valid, active license is not installed in the cluster
  - IP connectivity for BGP is not correct

For workload reachability issues:

- If you cannot access an EKS cluster from an EC2 instance:
  - Ensure that the workload's VN is exposed to the `TransitGateway`.
  - Ensure that the route from the exposed VN to the VPC appears in the VPC's routing table.
  - Ensure that the EC2 instance has security groups configured to access the CIDR of the VN.
    - The custom controllers automatically create one securtity group (`gsi-sg`) that allows access to all of the routes exposed to a transit gateway.

Configure a Service Account to Assume an IAM role

SUMMARY

This topic provides information about how to configure a Kubernetes service account to assume an AWS Identity and Access Management (IAM) role. This is a prerequisite for configuring VPC to CN2 communication in Juniper Cloud-Native Contrail release 23.1.

IN THIS SECTION

- Configure a Service Account to Assume an IAM Role | 220

Configure a Service Account to Assume an IAM Role

CN2 release 23.1 supports the ability to access Amazon VPC networks from CN2 or EKS clusters. In order to enable this feature, you must associate a Kubernetes service account with an AWS IAM role. Since this feature creates AWS resources, the custom CN2 controllers that reside in the `contrail-gsi` namespace need create, read, update, and delete (CRUD) access for these resources.

In order to grant CRUD access to the custom CN2 controllers, you must configure a service account to assume an IAM role with access to the CRUD operations the controllers must perform. Follow the steps in the following Amazon Web Services link to complete this process: Configuring a Kubernetes service account to assume an IAM role.

In step 1a under the "To associate an IAM role with a Kubernetes account" section, you are prompted to create a file that includes permissions for the AWS services that you want your controller pods to access. Juniper provides the JSON file below for this purpose.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1674864366447",
            "Action": [
                "ec2:AuthorizeSecurityGroupEgress",
                "ec2:AuthorizeSecurityGroupIngress",
                "ec2:CreateRoute",
                "ec2:CreateSecurityGroup",
                "ec2:CreateTags",
                "ec2:CreateTransitGateway",
```

```
                "ec2:CreateTransitGatewayConnect",
                "ec2:CreateTransitGatewayConnectPeer",
                "ec2:CreateTransitGatewayVpcAttachment",
                "ec2:DeleteRoute",
                "ec2:DeleteSecurityGroup",
                "ec2:DeleteTags",
                "ec2:DeleteTransitGateway",
                "ec2:DeleteTransitGatewayConnect",
                "ec2:DeleteTransitGatewayConnectPeer",
                "ec2:DeleteTransitGatewayVpcAttachment",
                "ec2:DescribeRouteTables",
                "ec2:DescribeSecurityGroups",
                "ec2:DescribeTags",
                "ec2:DescribeTransitGatewayConnectPeers",
                "ec2:DescribeTransitGatewayConnects",
                "ec2:DescribeTransitGatewayVpcAttachments",
                "ec2:DescribeTransitGateways",
                "ec2:RevokeSecurityGroupEgress",
                "ec2:RevokeSecurityGroupIngress"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```


In step 2, you are prompted to create a Kubernetes service account. You can use the example service account YAML provided in the AWS article, but you must use `contrail-gsi` for the namespace and `contrail-gsi-serviceaccount` for the name of the service account.

RELATED DOCUMENTATION

VPC to CN2 Communication in AWS EKS | 211

5

CHAPTER

Configure DPDK

Deploy DPDK vRouter | 223

Deploy DPDK vRouter

IN THIS SECTION

- DPDK Overview | 223
- DPDK vRouter Support for DPDK and Non-DPDK Workloads | 224
- Non-DPDK Pod Overview | 224
- DPDK Pod Overview | 224
- Mix of Non-DPDK and DPDK Pod Overview | 225
- DPDK vRouter Architecture | 225
- DPDK Interface Support for Containers | 225
- DPDK vRouter Host Prerequisites | 226
- Deploy a Kubernetes Cluster with DPDK vRouter in Compute Node | 228
- DPDK vRouter Custom Resource Settings | 229

DPDK Overview

Cloud-Native Contrail® Networking™ supports the Data Plane Development Kit (DPDK). DPDK is an open-source set of libraries and drivers for rapid packet processing. Cloud-Native Contrail Networking accelerates container networking with DPDK vRouter technology. DPDK enables fast packet processing by allowing network interface cards (NICs) to send direct memory access (DMA) packets directly into an application's address space. This method of packet routing lets the application poll for packets, which prevents the overhead of interrupts from the NIC.

Utilizing DPDK enables the Cloud-Native Contrail vRouter to process more packets per second than it could when running as a kernel module DPDK interface for container service functions. CN2 utilizes DPDK vRouter's processing power leverages the processing power for high-demand container service functions.

When you provision a Contrail compute node with DPDK, the corresponding YAML file specifies the:

- Number of CPU cores to use for forwarding packets.
- Number of huge pages to allocate for DPDK.
- UIO driver to use for DPDK.

DPDK vRouter Support for DPDK and Non-DPDK Workloads

When a container or pod needs access to the DPDK vRouter, the following workload types occur:

1. Non-DPDK workload (pod): This workload includes non-DPDK pod applications that are unaware of the underlying DPDK vRouter. These applications are not designed for DPDK and do not use DPDK capabilities. In Cloud-Native Contrail Networking, this workload type functions normally in a DPDK vRouter-enabled cluster.
2. Containerized DPDK workload: These workloads are built on the DPDK platform. DPDK interfaces are brought up using (Virtual Host) vhost protocol, which acts as a datapath for management and control functions. The vhost protocol enables virtualized host devices to be mapped to virtual input & output (virtio) devices. Pods act as the vHost Server, and the underlying DPDK vRouter acts as the vHost Client.
3. Mix of Non-DPDK and DPDK workloads: The management or control channel on an application in this pod is a non-DPDK (Veth pair), and the datapath is a DPDK interface. A Veth pair is a pair of connected virtual Ethernet interfaces.

Non-DPDK Pod Overview

A virtual ethernet (Veth) pair plumbs the networking of a non-DPDK pod. One end of the Veth pair attaches to the pod's namespace. The other end attaches to the kernel of the host machine. The Container Networking Interface (CNI) establishes the Veth pair and allocates IP addresses using IP Address Management (IPAM).

DPDK Pod Overview

A DPDK pod contains a vhost interface and a virtio interface. The pod uses the vhost interface for management purposes and the virtio interface for high-throughput packet processing applications. A DPDK application in the pod uses the vhost protocol to establish communication with the DPDK vRouter in the host. The DPDK application receives an argument to establish a filepath for a UNIX socket. The vRouter uses this socket to establish the control channel, run negotiations, and create vrings over huge pages of shared memory for high-speed datapaths.

Mix of Non-DPDK and DPDK Pod Overview

This pod might contain non-DPDK and DPDK applications. A non-DPDK application uses a non-DPDK interface (Veth pair), and the DPDK application uses the DPDK interfaces (vhost, virtio). These two workloads occur simultaneously.

DPDK vRouter Architecture

The Contrail DPDK vRouter is a container that runs inside the Contrail compute node. The vRouter runs as either a Linux kernel module or a user space DPDK process. The vRouter is responsible for transmitting packets between virtual workloads (tenants, guests) on physical devices. The vRouter also transmits packets between virtual interfaces and physical interfaces.

The CN2 vRouter supports the following encapsulation protocols:

- MPLS over UDP (MPLSoUDP)
- MPLS over GRE (MPLSoGRE)
- Virtual Extensible LAN (VXLAN)

Compared with the traditional Linux kernel deployment, deploying the vRouter as a user space DPDK process drastically increases the performance and processing speed of the vRouter application. This increase in performance is the result of the following factors:

- The virtual network functions (VNFs) operating in user space are built for DPDK and designed to take advantage of DPDK’s packet processing power.
- DPDK's poll mode drivers (PMDs) use the physical interface (NIC) of a VM's host instead of the Linux kernel's interrupt-based drivers. The NIC’s registers operate in user space, which makes them accessible by DPDK’s PMDs.

As a result, the Linux OS does not need to manage the NIC's registers. This means that the DPDK application manages all packet polling, packet processing, and packet forwarding of a NIC. Instead of waiting for an I/O interrupt to occur, a DPDK application constantly polls for packets and processes these packets immediately upon receiving them.

DPDK Interface Support for Containers

The benefits and architecture of DPDK usually optimize VM networking. Cloud-Native Contrail Networking lets your Kubernetes containers take full advantage of these features. In Kubernetes, a

containerized DPDK pod typically contains two or more interfaces. The following interfaces form the backbone of a DPDK pod:

- **Vhost user protocol (for management):** The vhost user protocol is a backend component that interfaces with the host. In Cloud-Native Contrail Networking, the vhost interface acts as a datapath for management and control functions between the pod and vRouter. This protocol comprises the following two planes:
  - The control plane exchanges information (memory mapping for DMA, capability negotiation for establishing and terminating the data plane) between a pod and vRouter through a Unix socket.
  - The data plane is implemented through direct memory access and transmits data packets between a pod and vRouter.
- **Virtio interface (for high-throughput applications):** At a high level, virtio is a virtual device that transmits packets between a pod and vRouter. The virtio interface is a shared memory (shm) solution that lets pods access DPDK libraries and features.

These interfaces enable the DPDK vRouter to transmit packets between pods. The interfaces give pods access to advanced networking features provided by the vRouter (huge pages, lockless ring buffers, poll mode drivers). For more information about these features, visit A journey to the vhost-users realm.

Applications use DPDK to create vhost and virtio interfaces. The application or pod then uses DPDK libraries directly to establish control channels using Unix domain sockets. The interfaces establish datapaths between a pod and vRouter using shared memory vrings.

DPDK vRouter Host Prerequisites

In order to deploy a DPDK vRouter, you must configure the following huge pages and NICs on the host node. Huge pages enable the OS to use memory pages larger than the default 4KB:

- Huge pages configuration: Specify the percentage of host memory to be reserved for the DPDK huge pages. The following command line shows huge pages set at 2MB:

  ```
  GRUB_CMDLINE_LINUX_DEFAULT="console=tty1 console=ttyS0 default_hugepagesz=2M hugepagesz=2M
  hugepages=8192"
  ```

  The following example allocates four 1GB huge pages and 1024 2MB huge pages:

  ```
  GRUB_CMDLINE_LINUX_DEFAULT="console=tty1 console=ttyS0 default_hugepagesz=1G hugepagesz=1G
  hugepages=4 hugepagesz=2M hugepages=1024"
  ```

NOTE: We recommend that you allocate 1GB for the huge pages size.

- Enable (input-output memory management unit (IOMMU): DPDK applications require IOMMU support. Configure IOMMU settings and enable IOMMU from the BIOS. Apply the following flags as boot parameters to enable IOMMU:

```
"intel_iommu=on iommu=pt"
```

- Ensure that the Kernel driver is loaded onto Port Forward 0 (port 0) of the host's NIC. Ensure that DPDK PMD drivers are loaded onto Port Forward 1 (port 1) of the host's NIC.

NOTE:
I40E Poll Mode Driver

- PCI driver (vfio-pci, uio_pci_generic): Specify which PCI driver to use based on NIC type.

NOTE: The vfio-pci is built-in.

  - ```
    uio_pci_generic
    ```

    - Manually install the uio_pci_generic module if needed:

      ```
      root@node-dpdk1:~# apt install linux-modules-extra-$(uname -r)
      ```

    - Verify that the uio_pci_generic module is installed:

      ```
      root@node-dpdk1:~# ls /lib/modules/5.4.0-59-generic/kernel/drivers/uio/
      uio.ko      uio_dmem_genirq.ko  uio_netx.ko         uio_pruss.ko
      uio_aec.ko  uio_hv_generic.ko   'uio_pci_generic.ko'  uio_sercos3.ko
      uio_cif.ko  uio_mf624.ko        uio_pdrv_genirq.ko
      ```
      ```

## Deploy a Kubernetes Cluster with DPDK vRouter in Compute Node

Cloud-Native Contrail Networking utilizes a DPDK deployer to launch a Kubernetes cluster with DPDK compatibility. This deployer performs lifecycle management functions and applies DPDK vRouter prerequisites. A custom resource (CR) for the DPDK vRouter is a subset of the deployer. The CR contains the following:

- Controllers for deploying Cloud-Native Contrail Networking resources
- Built-in controller logic for the vRouter

Apply the DPDK deployer YAML file, and deploy the DPDK vRouter CR with `agentModeType: dpdk` using the following command:

```
kubectl apply -f <vrouter_cr.yaml>
```

After applying the CR YAML file, the deployer creates a daemonset for the vRouter. This deamonset spins up a pod with a DPDK container.

If you get an error message, ensure that your cluster has the custom resource definition (CRD) for the vRouter using the following command:

```
kubectl get crds
```

The following is an example of the output you receive:

```
NAME                                                CREATED AT
vrouters.dataplane.juniper.net                      2021-06-16T16:06:34Z
```

If no CRD is present in the cluster, check the deployer using the following command:

```
kubectl get deployment contrail-k8s-deployer -n contrail-deploy -o yaml
```

Check the image used by the `contrail-k8s-crdloader` container. This image should be the latest image the deployer uses. Update the image and ensure that your new pod uses this image.

After you verify that your new pod is running the latest image, use the following command to verify that the CRD for the vRouter is present:

```
kubectl get crds
```

After you verify that the CRD for the vRouter is present, use the following command to apply the vRouter CR:

```
kubectl apply -f <vrouter_cr.yaml>
```

DPDK vRouter Custom Resource Settings

You can configure the following settings of the vRouter's CR:

- `service_core_mask`: Specify a service core mask. The service core mask enables you to dynamically allocate CPU cores for services.
- You can enter the following input formats:
  - Hexadecimal (for example, 0xf)
  - List of CPUs separated by commas (for example, 1,2,4)
  - Range of CPUs separated by a dash (for example, 1-4)

  > **NOTE**: PMDs require the bulk of your available CPU cores for packet processing. As a result, we recommend that you reserve a maximum of 1 to 2 CPU cores for `service_core_mask` and `dpdk_ctrl_thread_mask`. These two cores share CPU power.

- `cpu_core_mask`: Specify a CPU core mask. DPDK's PMDs use these cores for high-throughput packet-processing applications.

  The following are supported input formats:

  - Hexadecimal (for example, 0xf)
  - List of CPUs separated by commas (for example, 1,2,4)
  - Range of CPUs separated by a dash (for example, 1-4)
- `dpdk_ctrl_thread_mask`: Specify a control thread mask. DPDK uses these core threads for internal processing.

  The following are supported input formats:

  - Hexadecimal (for example, 0xf)

- List of CPUs separated by commas (for example, 1,2,4)
- Range of CPUs separated by a dash (for example, 1-4)

> **NOTE**: PMDs require the bulk of your available CPU cores for packet processing. As a result, we recommend that you reserve a maximum of 1 to 2 CPU cores for `service_core_mask` and `dpdk_ctrl_thread_mask`. These two cores share CPU power.

- `dpdk_command_additional_args`: Specify DPDK vRouter settings that are not default settings. Arguments that you enter here are appended to the DPDK PMD command line.

  The following is an example argument:`--yield_option 0`.

CHAPTER

Configure Services

Configure ClusterIP Service by Assigning Endpoints | 232

NodePort Service Support in Cloud-Native Contrail Networking | 236

Create a Load Balancer Service | 246

FloatingIP/DNAT for IPv6 Addresses | 258

Configure ClusterIP Service by Assigning Endpoints

IN THIS SECTION

- ClusterIP Service without a Selector and Manually Assigned Endpoints | 232
- Configure ClusterIP Service | 233

ClusterIP Service without a Selector and Manually Assigned Endpoints

Juniper® Cloud-Native Contrail Networking (CN2) supports the ClusterIP service to work with manually assigned endpoints without adding a selector in the service. ClusterIP is the default type of service, which is used to expose a service on an IP address internal to the cluster. Access is only permitted from within the cluster.

When creating the endpoint for the service, it's important to add the IP address and `targetRef` in the endpoint. The `targetRef` should include the pod details such as kind, name, and namespace. Without these details, connectivity to the ClusterIP service will not work.

Pod details provided in the `targetRef` of the endpoint are used to add the virtual machine interface (VMI) reference of the corresponding pod in the service floating IP (FIP) object.

See the following example of pod details provided in `targetRef`:

```
apiVersion: v1
kind: Endpoints
metadata:
  labels:
    app: nginx
  name: nginx
  namespace: clusterip
subsets:
- addresses:
  - ip: 10.128.0.151
    targetRef:
      kind: Pod
      name: nginx-7d79f94b45-9tfjm
      namespace: clusterip
```

```
- ip: 10.128.0.175
  targetRef:
    kind: Pod
    name: nginx-7d79f94b45-kcb4s
    namespace: clusterip
ports:
- name: http
  port: 8080
  protocol: TCP
```

Configure ClusterIP Service

Following is an example procedure to configure ClusterIP service by manually assigning endpoints and without adding a selector.

1. Deploy the application deployment. In this example, the NGINX application is deployed.

```
apiVersion: v1
kind: Namespace
metadata:
  name: clusterip
---
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx
  namespace: clusterip
spec:
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 1 pods matching the template
  template: # create pods using pod definition in this template
    metadata:
      labels:
        app: nginx
    spec:
      containers:
```

```
      - name: nginx
        image: svl-artifactory.juniper.net/atom-docker/nginxinc/nginx-unprivileged:1.21
        ports:
        - containerPort: 8080
```

2. Check the pods.

```
[core@ocp-avyaw-bc6wig-ctrl-3 ~]$ kubectl get po -n clusterip -o wide
NAME                     READY   STATUS    RESTARTS   AGE    IP
NODE                        NOMINATED NODE   READINESS GATES
nginx-7d79f94b45-9tfjm   1/1     Running   0          10m   10.128.0.151   ocp-avyaw-bc6wig-
worker-2   <none>           <none>
nginx-7d79f94b45-kcb4s   1/1     Running   0          10m   10.128.0.175   ocp-avyaw-bc6wig-
worker-1   <none>           <none>
```

3. Deploy the ClusterIP service without defining a selector in `spec`. In this example, the ClusterIP service maps to port 8080 on the application pod.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: clusterip
  labels:
    app: nginx
spec:
  ports:
  - name: http
    port: 8080
    protocol: TCP
    targetPort: 8080
  type: ClusterIP
```

4. Verify the service.

```
[core@ocp-avyaw-bc6wig-ctrl-3 ~]$ kubectl get svc -n clusterip
NAME    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
nginx   ClusterIP   172.30.74.100   <none>        8080/TCP   3m
```

5. Create the endpoints for the service. Add the IP address and `targetRef` with pod details in the endpoints.

```
apiVersion: v1
kind: Endpoints
metadata:
  labels:
    app: nginx
  name: nginx
  namespace: clusterip
subsets:
- addresses:
  - ip: 10.128.0.151
    targetRef:
      kind: Pod
      name: nginx-7d79f94b45-9tfjm
      namespace: clusterip
  - ip: 10.128.0.175
    targetRef:
      kind: Pod
      name: nginx-7d79f94b45-kcb4s
      namespace: clusterip
  ports:
  - name: http
    port: 8080
    protocol: TCP
```

6. Check the connectivity to the ClusterIP service from any test pod.

```
[core@ocp-avyaw-bc6wig-ctrl-2 ~]$ kubectl exec -it curl-test -n clusterip sh
# curl 172.30.74.100:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
```

```
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

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

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

NodePort Service Support in Cloud-Native Contrail Networking

IN THIS SECTION

- Contrail Networking Load Balancer Objects | 237
- NodePort Service in Contrail Networking | 239
- Workflow of Creating NodePort Service | 240
- Kubernetes Probes and Kubernetes NodePort Service | 242
- NodePort Service Port Mapping | 242
- Example: NodePort Service Request Journey | 243
- Local Option Limitation in External Traffic Policy | 245
- Update or Delete a Service, or Remove a Pod from Service | 245

Juniper® Networks supports Kubernetes NodePort service in environments using Cloud-Native Contrail Networking (CN2) Release 22.1 or later in a Kubernetes-orchestrated environment.

In Kubernetes, a service is an abstraction that defines a logical set of pods and the policy by which you (the administrator) can access the pods. Kubernetes selects the set of pods implementing a service

based on the LabelSelector object in the service definition. NodePort service exposes a service on each node's IP at a static port. It maps the static port on each node with a port of the application on the pod.

In CN2, Kubernetes NodePort service is implemented using the InstanceIP resource and FloatingIP resource, both of which are similar to the ClusterIP service.

Kubernetes provides a flat networking model in which all pods can talk to each other. Network policy is added to provide security between the pods. CN2 integrated with Kubernetes adds networking functionality, including multi-tenancy, network isolation, micro-segmentation with network policies, and load balancing.

The following table lists the mapping between Kubernetes concepts and CN2 resources.

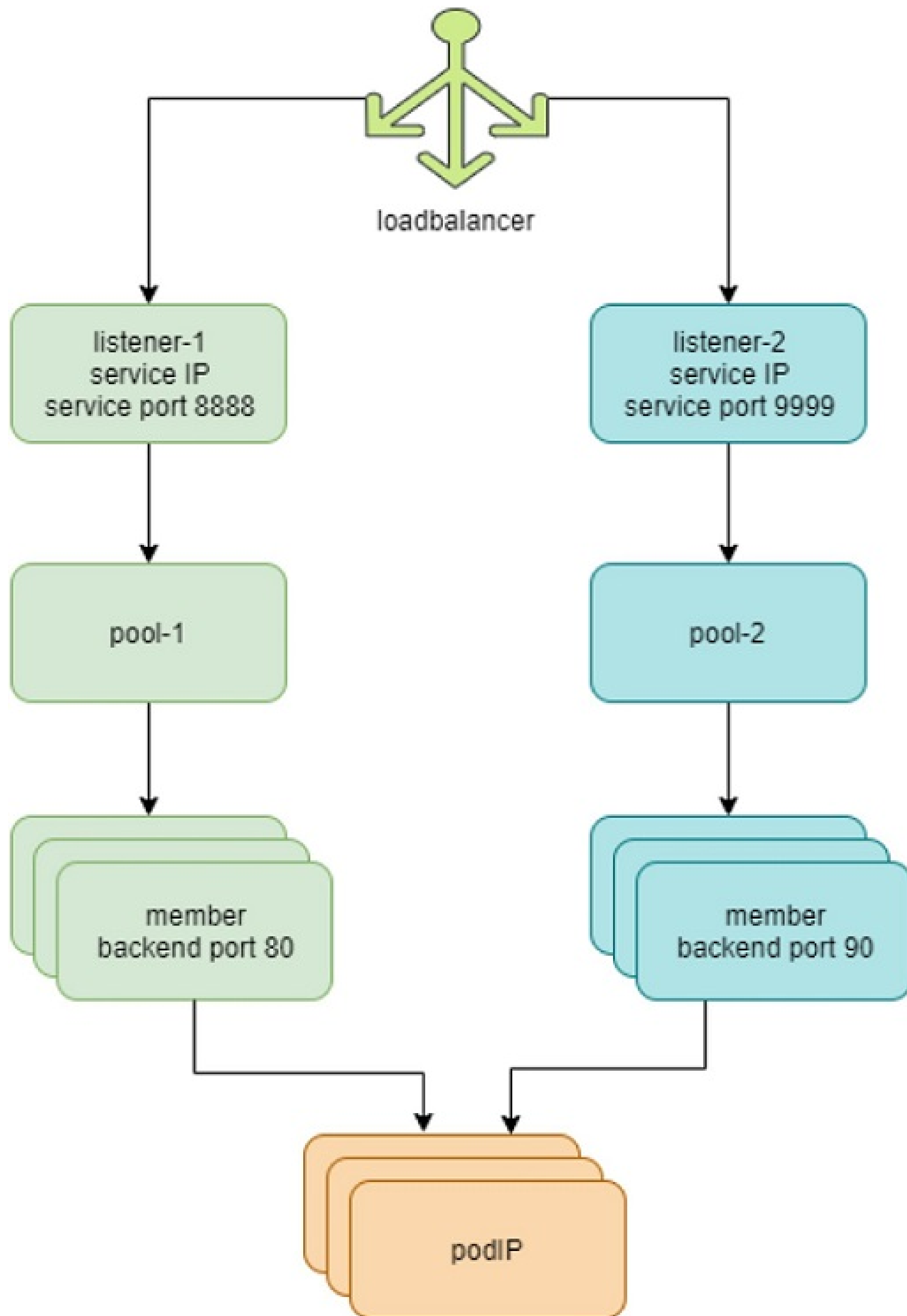
Table 18: Kubernetes Concepts to CN2 Resource Mapping

Kubernetes Concept	CN2 Resource
Namespace	Shared or single project
Pod	Virtual Machine
Service	Equal-cost multipath (ECMP) LoadBalancer
Ingress	HAProxy LoadBalancer for URL
Network Policy	Contrail Security

Contrail Networking Load Balancer Objects

Figure 10 on page 238 and the following list describe the load balancer objects in CN2.

Figure 10: Load Balancer Objects



- Each service in CN2 is represented by a load balancer object.
- For each service port, a listener object is created for the same service load balancer.

- For each listener there is a pool object.
- The pool contains members. Depending on the number of backend pods, one pool might have multiple members.
- Each member object in the pool maps to one of the backend pods.
- The `contrail-kube-manager` listens to `kube-apiserver` for the Kubernetes service. When a service is created, a load balancer object with `loadbalancer_provider` type `native` is created.
- The load balancer has a virtual IP address (VIP), which is the same as the service IP address.
- The service IP/VIP address is linked to the interface of each backend pod. This is accomplished with an ECMP load-balancer driver.
- The linkage from the service IP address to the interfaces of multiple backend pods creates an ECMP next hop in CN2. Traffic is load balanced from the source pod directly to one of the backend pods.
- The `contrail-kube-manager` continues to listen to `kube-apiserver` for any changes. Based on the pod list in the endpoints, `contrail-kube-manager` identifies the most current backend pods and updates members in the pool.

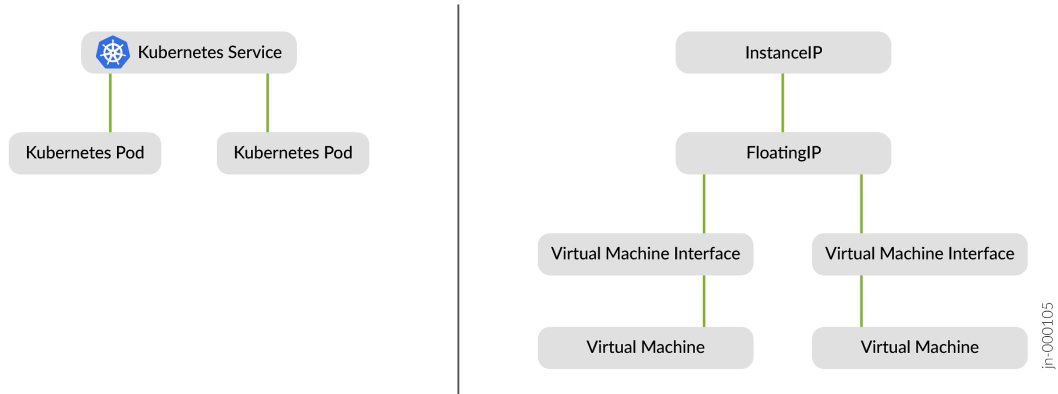
NodePort Service in Contrail Networking

A controller service is implemented in `kube-manager`. The `kube-manager` is the interface between Kubernetes core resources, such as service, and the extended Contrail resources, such as `VirtualNetwork` and `RoutingInstance`. This controller service watches events going through the resource endpoints. An endpoint receives an event for any change related to its service. The endpoint also receives an event for pods created and deleted that match the service selector. The controller service handles creating the Contrail resources needed: See Figure 11 on page 240.

- `InstanceIP` resource related to the `ServiceNetwork`
- `FloatingIP` resource and the associated `VirtualMachineInterfaces`

When you create a service, an associated endpoint is automatically created by Kubernetes, which allows the controller service to receive new requests.

Figure 11: Controller Service Creates Contrail Resources



Workflow of Creating NodePort Service

Figure 12 on page 241 and the following steps detail the workflow when NodePort service is created.

Figure 12: Creating NodePort Service

1. When the NodePort service is created, `InstanceIP` (IIP) is created. The `InstanceIP` resource specifies a fixed IP address and its characteristics that belong to a subnet of a referred virtual network.

2. Once the endpoint is connected to the NodePort service, the `FloatingIP` is created. The `kube-manager` watches for the creation of endpoints connected to a service.

3. After a new endpoint is created, `kube-manager` creates an `InstanceIP` in the `ServiceVirtualNetwork` subnet. The `kube-manager` then creates a `FloatingIP` using the `InstanceIP` as the parent.

4. The `FloatingIP` resource specifies a special kind of IP address that does not belong to a specific `VirtualMachineInterface` (VMI). The `FloatingIP` is assigned from a separate `VirtualNetwork` subnet and can be associated with multiple VMIs. When associated with multiple VMIs, traffic destined to the `FloatingIP` is distributed using ECMP across all VMIs.

Notes about VMIs:

- VMIs are dynamically updated as pods and labels are added and deleted.

- A VMI represents an interface (port) into a virtual network and might or might not have a corresponding virtual machine.

- A VMI has at minimum a MAC address and an IP address.

Notes about VMs:

- A VM resource represents a compute container such as VM, baremetal, pod, or container.
- Each VM can communicate with other VMs on the same tenant network, subject to policy restrictions.
- As tenant networks are isolated, VMs in one tenant cannot communicate with VMs in another tenant unless specifically allowed by policy.

Kubernetes Probes and Kubernetes NodePort Service

The `kubelet`, an agent that runs on each node, needs reachability to pods for liveness and readiness probes. Contrail network policy is created between the IP fabric network and pod network to provide reachability between node and pods. Whenever the pod network is created, the network policy is attached to the pod network to provide reachability between node and pods. As a result, any process in the node can reach the pods.

The Kubernetes NodePort service is based on node reachability to pods. Since CN2 provides connectivity between nodes and pods through the Contrail network policy, NodePort is supported.

The NodePort service supports two types of traffic:

- East-West
- Fabric to Pod

NodePort Service Port Mapping

The port mappings for the Kubernetes NodePort service are located in the `FloatingIp` resource in the YAML file. In `FloatingIp`, the ports are added in `"floatingIpPortMappings"`.

If the `targetPort` is not mentioned in the service, then the `port` value is specified as the default.

Following is an example `spec` YAML file for the NodePort service with port details:

```
spec:
 clusterIP: 10.100.13.106
 clusterIPs:
```

```
  - 10.100.13.106
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: my-nginx
  sessionAffinity: None
```

In the preceding example spec YAML file, "floatingIpPortMappings" are created in the FloatingIp resource.

Following is an example "floatingIpPortMappings" YAML file:

```
"floatingIpPortMappings": {
            "portMappings": [
                {
                    "srcPort": 80,
                    "dstPort": 80,
                    "protocol": "TCP"
                }
            ]
        }
```

Example: NodePort Service Request Journey

Let's follow the journey of a NodePort service request from when the request gets to the node port until the service request reaches the backend pod.

The Nodeport service relies on kubeproxy. The Kubernetes network proxy (kube-proxy) is a daemon running on each node. The daemon reflects the services defined in the cluster and manages the rules to load balance requests to a service’s backend pods.

In the following example, the NodePort service apple-service is created and its endpoints are associated with the service.

```
user@domain ~ % kubectl describe svc apple-service
Name:                     apple-service
Namespace:                default
Labels:                   <none>
Annotations:              <none>
```

```
Selector:                 app=apple
Type:                     NodePort
IP Families:              <none>
IP:                       10.105.135.144
IPs:                      10.105.135.144
Port:                     <unset>  5678/TCP
TargetPort:               5678/TCP
NodePort:                 <unset>  31050/TCP
Endpoints:                10.244.0.4:5678
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>

user@domain ~ % kubectl get endpoints apple-service
NAME            ENDPOINTS         AGE
apple-service   10.244.0.4:5678   2d18h
```

Each time a service is created or deleted or the endpoints are modified, `kube-proxy` updates the `iptables` rules on each node of the cluster. View the `iptables` chains to understand and follow the journey of the request.

First, the KUBE-NODEPORTS chain allows the packets coming on service of type `NodePort`.

```
$ sudo iptables -L KUBE-NODEPORTS -t nat
Chain KUBE-NODEPORTS (1 references)
target     prot opt source               destination
KUBE-MARK-MASQ  tcp  --  anywhere             anywhere             /* default/apple-service */
tcp dpt:31050
KUBE-SVC-Y4TE457BRBWMNDKG  tcp  --  anywhere             anywhere             /* default/apple-
service */ tcp dpt:31050
```

Each packet coming into port `31050` is first handled by the KUBE-MARK-MASQ, which tags the packet with a `0x4000` value.

Next, the packet is handled by the KUBE-SVC-Y4TE457BRBWMNDKG chain (referenced in the KUBE-NODEPORTS chain above). If we take a closer look at that chain, we can see additional `iptables` chains:

```
$ sudo iptables -L KUBE-SVC-Y4TE457BRBWMNDKG -t nat
Chain KUBE-SVC-Y4TE457BRBWMNDKG (2 references)
target     prot opt source               destination
```

```
KUBE-SEP-LCGKUEHRD52LOEFX  all  --  anywhere             anywhere             /* default/apple-
service */
```

Inspect the KUBE-SEP-LCGKUEHRD52LOEFX chains to see that they define the routing to one of the backend pods running the `apple-service` application.

```
$ sudo iptables -L KUBE-SEP-LCGKUEHRD52LOEFX -t nat
Chain KUBE-SEP-LCGKUEHRD52LOEFX (1 references)
target     prot opt source               destination
KUBE-MARK-MASQ  all  --  10.244.0.4           anywhere             /* default/apple-service */
DNAT       tcp  --  anywhere             anywhere             /* default/apple-service */ tcp
to:10.244.0.4:5678
```

This completes the journey of a NodePort service request from the point at which the request gets to the node port until the service request reaches the backend pod.

Local Option Limitation in External Traffic Policy

The NodePort service with `externalTrafficPolicy` set as `Local` is not supported in CN2 Release 22.1.

The `externalTrafficPolicy` denotes if this service wants to route external traffic to node-local or cluster-wide endpoints.

- `Local` preserves the client source IP address and avoids a second hop for NodePort type services.
- `Cluster` obscures the client source IP address and might cause a second hop to another node.

`Cluster` is the default for `externalTrafficPolicy`.

Update or Delete a Service, or Remove a Pod from Service

- Update service—You can change any modifiable fields, excluding `Name` and `Namespace`. For example, you can change Nodeport service to `ClusterIp` by changing the `Type` field in the service YAML definition.
- Delete service—You can delete a service, irrespective of `Type`, with the command

  ```
  kubectl delete -n <name_space> <service_name> .
  ```

- Remove pod from service—You can remove a pod from service by changing the `Labels` and `Selector` on the service or pod.

Create a Load Balancer Service

SUMMARY

This topic describes how to create a Load Balancer service in Juniper Cloud-Native Contrail® Networking (CN2). Juniper Networks supports this feature using Contrail Networking Release 22.1 or later in a Kubernetes-orchestrated environment.

IN THIS SECTION

- Load Balancer Service Overview | 246
- Create a Load Balancer Service | 247
- Configure Load Balancer Services Without Selectors | 254

Load Balancer Service Overview

In Kubernetes, a service is an abstract way to expose an application running on a set of pods as a network service. Kubernetes supports three types of services: ClusterIP, NodePort and LoadBalancer. This topic describes how to create a load balancer service in CN2.

In CN2, a load balancer service is implemented with the `InstanceIP` resource and `FloatingIP` resource as described below:

- The `FloatingIP` is used in the service implementation to expose an external IP to the load balancer service. The `FloatingIP` resource is also associated with the pod's `VirtualMachineInterfaces`.
- The `InstanceIP` resource is related to the `VirtualNetwork`. Two `instanceIPs` are created, one for the service network and one for the external network.

A controller service is implemented in Contrail's kube-manager. Kube-manager is the interface between Kubernetes core resources and the extended Contrail resources, such as the `VirtualNetwork`. When you create a load balancer service, kube-manager listens and allocates the IP from an external virtual network. This external virtual network exposes the load balancer service on the external IPs. Any requests received through the provisioned external IP is ECMP load-balanced across the pods associated with the load balancer.

Create a Load Balancer Service

IN THIS SECTION

- Dual-Stack Networking Support | 254

The following sections describe how to create a load balancer service in CN2.

Prerequisites

Before you begin, make sure of the following:

- You have set up a working cloud networking environment with Kubernetes.
- CN2 is installed and is operational.
- You have configured kube-manager to define the external networks to be used by the load balancer service.

Define an External Virtual Network

First, define an external virtual network. You can define the virtual network two ways: by creating a Network Attachment Definition (NAD) or by creating a virtual network.

> **NOTE**: A Multus deployment requires that you use a NAD to define an external network.

The following example shows how to define an external virtual network using a `NetworkAttachmentDefinition`. In this example, the external IP is allocated from the subnet range `192.168.102.0/24`. When the `NetworkAttachmentDefinition` is applied, kube-manager creates a virtual network with the name `ecmp-default` in the namespace `ecmp-project`.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
 name: ecmp-default
 namespace: ecmp-project
 annotations:
```

```
    juniper.net/networks: '{
    "ipamV4Subnet": "192.168.102.0/24",
    "fabricSNAT": false
    "core.juniper.net/display-name: "External Virtual Network"
  }'
  core.juniper.net/display-name: "External Virtual Network"
 labels:
   service.contrail.juniper.net/externalNetworkSelector: default-external
 spec:
   config: '{
     "cniVersion": "0.3.1",
     "name": "ecmp-default",
     "type": "contrail-k8s-cni"
 }'
```

Specify the External Networks

By default, kube-manager allocates the external IP for a load balancer service from the `default-external` network. If desired, you can allocate the external IP from a different network by defining custom selectors, as shown in the following example:

```
apiVersion: configplane.juniper.net/v1alpha1
kind: Kubemanager
metadata:
 generation: 148
 name: contrail-k8s-kubemanager
 namespace: contrail
spec:
 externalNetworkSelectors:
  default-external:
   networkSelector:
    matchLabels:
     service.contrail.juniper.net/externalNetwork: default-external
 custom-external:
  namespaceSelector:
   matchLabels:
    customNamespaceKey: custom-namespace-value
 networkSelector:
  matchLabels:
   customNetworkKey: custom-network-value
 custom-external-in-service-namespace:
```

```
  networkSelector:
   matchLabels:
    customExternalInServiceNetworkKey: custom-external-in-service-network-value
```

The `VirtualNetworks` in this example match the labels shown in the previous example (in relative order).

```
apiVersion: core.contrail.juniper.net/v3
kind: Subnet
metadata:
  namespace: contrail
  name: external-subnet
spec:
  cidr: "10.244.0.0/16"
  defaultGateway: 10.244.0.1
---
apiVersion: core.contrail.juniper.net/v3kind: VirtualNetwork # matches example 1
metadata:
  name: default-external-vn
  namespace: contrail
  labels:
    service.contrail.juniper.net/externalNetworkSelector: default-external
spec:
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v3
    kind: Subnet
    namespace: contrail
    name: external-subnet
---
# this is how you define namespace selector
# Namespace must have appropriate label if required by namespaceSelector
apiVersion: v1
kind: Namespace
metadata:
  labels:
      customNamespaceKey: custom-namespace-value #user for your external ip
  name: custom-namespace
---
apiVersion: core.contrail.juniper.net/v3kind: Subnet
metadata:
  namespace: custom-namespace
  name: external-subnet-custom-namespace
spec:
```

```
  cidr: "10.0.0.0/16"
  defaultGateway: 10.0.0.1
---
apiVersion: core.contrail.juniper.net/v3
kind: VirtualNetwork
metadata:
  name: external-vn-1 # matches example 2 and example 3
  namespace: custom-namespace
  labels:
    customNetworkKey: custom-network-value
spec:
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v3
    kind: Subnet
    namespace: custom-namespace
    name: external-subnet-custom-namespace
---
apiVersion: core.contrail.juniper.net/v3kind: Subnet
metadata:
  namespace: custom-namespace
  name: external-subnet-in-service
spec:
  cidr: "192.168.0.0/16"
  defaultGateway: 192.168.0.1
---
apiVersion: core.contrail.juniper.net/v3
kind: VirtualNetwork
metadata:
  name: external-vn-2 # matches example 4
  namespace: custom-namespace
  labels:
    customExternalInServiceNetworkKey: custom-external-in-service-network-value
spec:
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v3
    kind: Subnet
    namespace: custom-namespace
    name: external-subnet-in-service
```

Define Service-Level Annotations

Additionally, you can define the following service-level annotations for external network discovery.

Annotation: externalNetwork.

In this example, the `externalNetwork` annotation allocates an external IP from the `evn` virtual network in the namespace `ns`.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  annotation:
    service.contrail.juniper.net/externalNetwork: ns/evn
spec:
  type: LoadBalancer
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Annotation: externalNetworkSelector

In this example, the `externalNetworkSelector` matches the name of the `externalNetworkSelector` defined in kube-manager.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  annotation:
    service.contrail.juniper.net/externalNetworkSelector: custom-external
spec:
  type: LoadBalancer
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

NOTE: You can also define service-level annotations in the namespace of the Kubernetes cluster or in the namespace of the Contrail cluster. The service-level annotation take precedence.

Examples: External Network Selection

The external virtual network is selected from one of the following in priority order:

NOTE: The virtual networks defined in "Specify the External Networks" on page 248 are linked to the annotations in the following examples.

Example 1: Default Selector

Kube-manager first searches for the default external network. This example uses the `default-external` selector because no annotation is specified.

This example matches the network `contrail/default-external-vn`.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: LoadBalancer
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Example 2: Custom Namespace

This example matches the network `custom-namespace/external-vn-1`.

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: my-service
  annotation:
    service.contrail.juniper.net/externalNetwork: custom-namespace/external-vn-1
spec:
  type: LoadBalancer
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Example 3: External Network Matches Preconfigured Selector in a Namespace

This example matches the network `custom-namespace/external-vn-1`.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  annotation:
    service.contrail.juniper.net/externalNetworkSelector: custom-external
spec:
  type: LoadBalancer
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Example 4: External Network Matches Preconfigured Selector in Service Namespace

This example matches the network `custom-namespace/external-vn-2`.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
```

```
  namespace: custom-namespace
  annotation:
    customExternalInServiceNetworkKey: custom-external-in-service-network-value
spec:
  type: LoadBalancer
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Dual-Stack Networking Support

IPv4 or IPv6 dual-stack networking enables the allocation of both IPv4 and IPv6 addresses to pods and services. As an administrator, you might need to select the IP family (IPv4 or IPv6) to use when defining a service. If you do not define the IP family, the default IPv4 address is used.

```
apiVersion: v1
kind: Service
metadata:
  name: MyService
specs:
  ipFamilies: ["IPv4", "IPv6"]
```

For more information, see Overview: IPv4 and IPv6 Dual-Stack Networking.

Configure Load Balancer Services Without Selectors

In Kubernetes, you can expose an application running on a set of pods as a network service. Kubernetes uses selectors to automatically create a load balancer service, but only uses the default primary interface for load balancing.

Starting in CN2 Release 22.3, you can load balance a service across multiple secondary interfaces. You can create secondary interfaces in CN2 without using a selector. Because the load balancer service has no selector, you must create the endpoint manually.

To configure load balancer services without selectors:

1. Create two virtual networks.

The following example shows two networks. One network for the pod's secondary interface (**pod-subnet**) and another network (**lb-subnet**) for the load balancer service external IP. These networks are connected by a common route target that routes traffic between the two networks.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
 name: pod-subnet
 namespace: my-lb
 annotations:
   juniper.net/networks: '{
     "ipamV4Subnet": "172.16.12.0/24",
     "routeTargetList": ["target:64521:1164"]
 }'
spec:
  config: '{
    "cniVersion": "0.3.1",
    "name": "pod-subnet",
    "type": "contrail-k8s-cni"
}'
---
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: lb-subnet
  namespace: my-lb
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "172.16.13.0/24",
      "routeTargetList": ["target:64521:1164"]
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "lb-subnet",
  "type": "contrail-k8s-cni"
}'
```

2. Create the pods on which you want to load balance the service. You can create multiple pods.

In this example, we'll create two pods in the **my-lb** namespace, each with its own IP address.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  namespace: my-lb
  labels:
    run: ecmp
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
        {
          "name": "pod-subnet",
          "namespace": "my-lb",
          "ips": ["172.16.23.0"]
        }
      ]'
spec:
  containers:
    - name: front01-multiintf
      image: svl-artifactory.juniper.net/atom-docker/cn2/bazel-build/dev/google-containers/
toolbox
      command:
        ["bash", "-c", "ip route add 172.16.23.0/24 via 172.16.23.1 dev eth1;while true; do
echo front01 | nc -w 1 -l -p 8080; done"]
      securityContext:
        privileged: true
---
apiVersion: v1
kind: Pod
metadata:
  name: my-pod1
  namespace: my-lb
  labels:
    run: ecmp
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
        {
         "name": "pod-subnet",
          "namespace": "my-lb",
          "ips": ["172.16.24.0"]
        }
```

```
      ]'
spec:
  containers:
    - name: front02-multiintf
      image: svl-artifactory.juniper.net/atom-docker/cn2/bazel-build/dev/google-containers/
toolbox
      command:
        ["bash", "-c", "ip route add 172.16.24.0/24 via 172.16.23.1 dev eth1; while true; do
echo front02 | nc -w 1 -l -p 8080; done"]
      securityContext:
        privileged: true
; done"]
```

3. Create a Load Balancer service.

   In this example, we'll create a load balancer service (**service-lb**) in the **my-lb** namespace without using a selector.

```
kind: Service
metadata:
  name: service-lb
  namespace: my-lb
  annotations:
    service.contrail.juniper.net/externalNetwork: my-lb/lb-subnet
spec:
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

4. Specify the endpoints (IP addresses) on which you want to load balance the service. Make sure that the endpoint has the same name as the load balancer service.

   In this example, we specified two pod endpoints for the secondary interfaces (ip: 172.16.23.0 and ip: 172.16.24.0).

```
apiVersion: v1
kind: Endpoints
metadata:
  name: service-lb
```

```
  namespace: my-lb
subsets:
  - addresses:
      - ip: 172.16.23.0
      - ip: 172.16.24.0
    ports:
      - port: 8080
```

Success! You can now load balance a service across multiple pods with the secondary interface.

NOTE: In addition to creating a load balancer service on the secondary interface, you can use a selector to create a load balancer on the default primary interface. The default primary interface can work in tandem with the secondary interface. You can use either interface to load balance across your desired service.

SEE ALSO

Kubernetes Services

FloatingIP/DNAT for IPv6 Addresses

SUMMARY

Juniper Cloud-Native Contrail release 23.1 supports FloatingIP, or Dynamic Network Address Translation (DNAT), for dual stack-enabled services (`ClusterIP`). This article provides information about how this feature works in CN2.

IN THIS SECTION

- Prerequisites | 258
- FloatingIP/DNAT Overview | 259
- DNAT for IPv6 Overview | 259
- Deploy FloatingIP/DNAT | 259

Prerequisites

This feature requires the following:

- An environment running CN2 release 23.1 or later
- A Kubeadm or Kubespray Kubernetes cluster with dual-stack `featureGate` enabled. For more information, see "IPv4 and IPv6 Dual-Stack Networking" on page 23.
- Kubernetes nodes configured with dual stack network interfaces

FloatingIP/DNAT Overview

In CN2, a FloatingIP implements ClusterIP functionality. After you create a service, a FloatingIP is allocated to that service from the service subnet and associated to all the back-end pod VMIs in the cluster. The vRouter performs DNAT for the back-end pods. This process comprises Equal-Cost Multi-Path Routing (ECMP) load balancing, where the back-end pod VMIs act as ECMP paths.

DNAT for IPv6 Overview

CN2 release 23.1 supports DNAT (FloatingIP) for IPv4 and IPv6 addresses for the CN2 ClusterIP service. DNAT for IPv6 functions the same as DNAT for IPv4; create a service (ClusterIP), specify `PreferDualStack` for the `ipFamilyPolicy`, and an IPv6 FloatingIP is allocated to that service. The vRouter performs DNAT and routes traffic to the next hop, or the translated destination address (back end pod VMI). from external networks to your back-end pod VMIs.

Deploy FloatingIP/DNAT

Complete the following steps to deploy this feature.

- Configure and install a `Deployment`. The `Deployment` object creates the back-end pods for the ClusterIP service. The following is an example `Deployment`. This `Deployment` creates a pod named `nginx` with a mounted `nginx-xconf` config.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: example-clusterip6
spec:
  selector:
    matchLabels:
```

```
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      tolerations:
        - key: "node.kubernetes.io/unreachable"
          operator: "Exists"
          effect: "NoExecute"
          tolerationSeconds: 2
        - key: "node.kubernetes.io/not-ready"
          operator: "Exists"
          effect: "NoExecute"
          tolerationSeconds: 2
      containers:
      - name: nginx
        image: <repository>:<tag>
        ports:
        - containerPort: 8080
        volumeMounts:
            - name: nginx-conf
              mountPath: /etc/nginx/nginx.conf
              subPath: nginx.conf
              readOnly: true
      volumes:
      - name: nginx-conf
        configMap:
          name: nginx-conf
          items:
            - key: nginx.conf
              path: nginx.conf
```

- Create a ClusterIP service. The following is an example service.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: clusterip6
  labels:
```

```
    app: nginx
spec:
  ports:
  - name: http
    port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: nginx
  ipFamilies:
  - "IPv6"
```

Note the following fields:

- `labels`: Identifies back-end pods with the `app: nginx label`.
- `selector`: Instructs the service to select VMIs belonging to back-end pods the with `app: nginx` label.
- `ipFamilies`: Specifies the IP family the ClusterIP service uses. The default is IPv4. To use both IP families, use the value `IpFamilyPolicy: PreferDualStack`.

7

CHAPTER

Analytics

Contrail Networking Analytics | 263

Contrail Networking Metric List | 269

Kubernetes Metric List | 283

Cluster Node Metric List | 321

Contrail Networking Alert List | 339

vRouter Session Analytics in Contrail Networking | 349

Centralized Logging | 357

Port-Based Mirroring | 360

Configurable Categories of Metrics Collection and Reporting (Tech Preview) | 365

Juniper CN2 Technology Previews (Tech Previews) | 371

Contrail Networking Analytics

IN THIS SECTION

- Overview: Analytics | 263
- Metrics | 264
- Supported Metrics | 264
- Alerts | 265
- Architecture | 266
- Configuration | 267
- Grafana | 268

Overview: Analytics

Analytics is an optional feature set in Juniper® Cloud-Native Contrail Networking (CN2) Release 22.1. The analytics are packaged separately from the CN2 core Container Network Interface (CNI) components. Analytics also has its own installation procedure. The package consists of a combination of open-source software and Juniper developed software that integrates with CN2.

The analytics features fit into the following high-level functional areas:

- Metrics—Statistical time series data collected from the Contrail Networking components and the base Kubernetes system
- Flow and Session Records—Network traffic information collected from the CN2 vRouter
- Sandesh User Visible Entities (UVE)—Records representing the system-wide state of externally visible objects that are collected from the CN2 vRouter and control node components
- Logs—Log messages collected from Kubernetes pods
- Introspect—A diagnostic utility that provides the ability to browse the internal state of the CN2 components

Metrics

Data Model

Metric information is based on a numerical time series data model. Each data point in a series is a sample of some system state that gets collected at a regular interval. A sampled value is recorded along with a timestamp at which the collection occurred. A sample record can also contain an optional set of key-value pairs called labels. Labels provide a dimension capability for metrics where a given combination of labels for the same metric name identifies a particular dimensional instantiation of that metric. For example, a metric named `api_http_requests_total` can utilize labels to provide visibility into the request counts at a URL and method type level. In the following example, the metric record for a sample value of 10 includes labels that indicate the type of request.

```
api_http_requests_total{method="POST", handler="/messages"} 10
```

Metric Data Types

Although all metric sample values are just numbers, the concept of data type exists within this numerical data model. A metric can be one of the following types:

- Counter—A cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart.
- Gauge—A metric that represents a single numerical value that can arbitrarily go up and down.
- Histogram—A histogram samples observations (such as request durations or response sizes) and counts them in configurable buckets. The histogram also provides a sum of all observed values.
- Summary—Similar to a histogram, a summary samples observations (such as request durations and response sizes). While it also provides a total count of observations and a sum of all observed values, the summary calculates configurable quantiles over a sliding time window.

The metric functionality in CN2 is implemented by Prometheus. For additional details about the metric data model, see the documentation at Prometheus.

Supported Metrics

The analytics solution supports the following sets of metrics:

- "Contrail Networking Metric List" on page 269—Metrics collected from the vRouter and control node components.

- "Kubernetes Metric List" on page 283—Metrics collected from various Kubernetes components, such as `apiserver`, `etcd`, `kubelet`, and so on.
- "Cluster Node Metrics" on page 321—Host-level metrics collected from the Kubernetes cluster nodes.

Alerts

Alerts are generated based on an analysis of collected metric data. Every supported alert type is based on a rule definition that contains the following information:

- Alert Name—A unique string identifier for the alert type
- Condition Expression—A Prometheus query language expression that gets evaluated against collected metric values to determine if the alert condition exists
- Condition Duration—The amount of time the problematic condition has to exist for the alert to be generated
- Severity—The alert level (critical, major, warning, info.)
- Summary—A short description of the problematic condition
- Description—A detailed description of the problematic condition

The CN2 analytics solution installs a set of "predefined alert rules" on page 339. You can also define your own custom alert rules. The creation of PrometheusRule Kubernetes resources in the namespace where the analytics Helm chart is deployed supports defining custom alerts. Following is an example of a custom alert rule.

```
apiVersion: monitoring.coreos.com/v1
  kind: PrometheusRule
  metadata:
    name: acme-corp-rules
  spec:
    groups:
    - name: acme-corp.rules
      rules:
      - alert: HostUnusualNetworkThroughputOut
        expr: "sum by (instance) (rate(node_network_transmit_bytes_total[2m])) / 1024 / 1024 >
100"
        labels:
          severity: warning
```

```
        annotations:
          summary: "Host unusual network throughput out (instance {{ $labels.instance }})"
          description: "Host network interfaces are sending too much data (> 100 MB/s)\n  VALUE
= {{ $value }}"
```

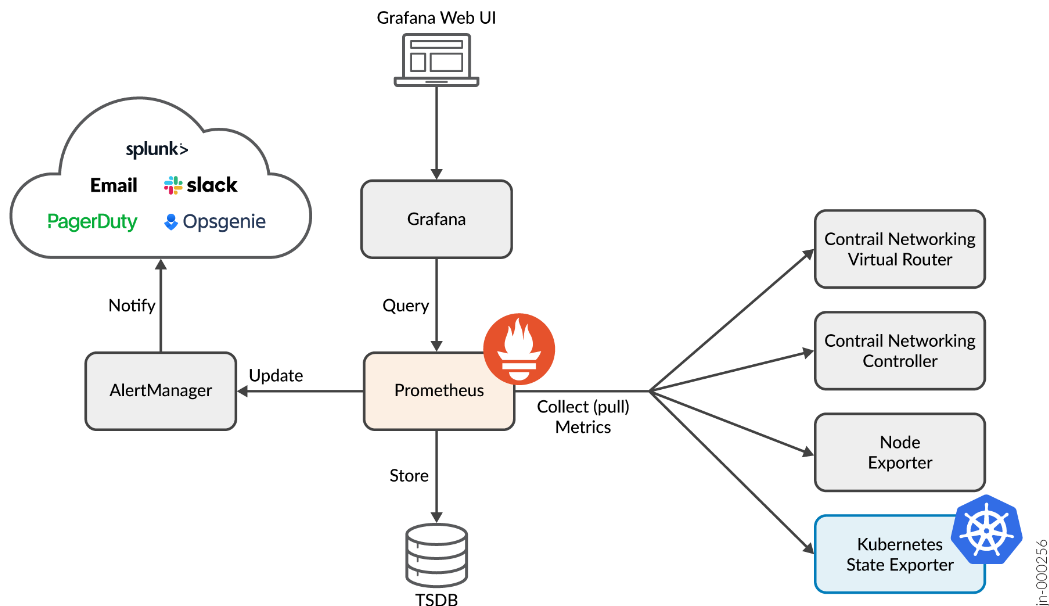
Prometheus stores generated alerts as records that can be viewed in the Grafana UI. The AlertManager component supports integration with external systems, such as PagerDuty, OpsGenie, or email for alert notification.

Architecture

As shown in Figure 13 on page 267, Prometheus is the core component of the metrics architecture. Prometheus implements the following functionality:

- Collection—A periodic polling mechanism that invokes API calls against other components (exporters) to pull values for a set of metrics
- Storage—A time series database that provides persistence for the metrics collected from the exporters
- Query—An API supporting an expression language called PromQL (Prometheus query language) that allows the historical metric information to be retrieved from the database
- Alerting—A framework providing an ability to define rules that produce alerts when certain conditions are observed in the collected metric data

Figure 13: Metrics Architecture



The other components of the metrics architecture are:

- Grafana—A service that provides a Web UI interface allowing the user to visualize the metric data in graphs.
- AlertManager—An integration service that notifies external systems of alerts generated by Prometheus.

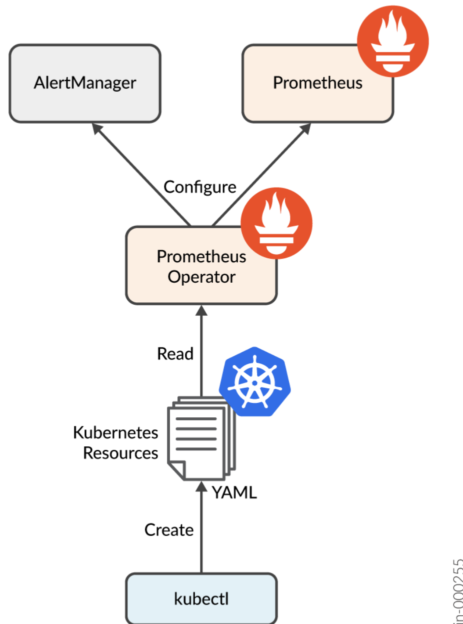
Configuration

The metrics functionality does not require any configuration by the end user. The installation of analytics takes care of configuring Prometheus to collect metrics from the exporters that provide all of the metrics described in "Supported Metrics" on page 264. A group of default alerting rules is also automatically set up as part of the installation. You can extend functionality through additional configuration after the installation. For example, you can define customer-specific alerting rules. You can also configure the AlertManager to integrate with any of the supported external systems in your environment.

The configuration of Prometheus and AlertManager involves an additional architectural component called the Prometheus Operator. As shown in Figure 14 on page 268, configuration is specified as Kubernetes custom resources. The Prometheus Operator translates the contents of these resources into

the native configuration that the Prometheus components recognize. The Operator also updates the components accordingly and restarts them whenever a configuration change requires a restart.

Figure 14: Prometheus Operator



Documentation for the full set of resources that the Prometheus Operator supports is available at Prometheus Operator API. Juniper Networks recommends that you limit your configurations to the subset of resource types related to alert rule definition and external system integration.

Grafana

The main UI for viewing metric data and alerts is Grafana. The analytics installation sets up the Grafana service and configures it with Prometheus as a data source. A set of default dashboards are also created.

Access the Grafana Web UI at `https://<k8sClusterIP>/grafana/login`. The default login credentials are user `admin` and password `prom-operator`.

RELATED DOCUMENTATION

vRouter Session Analytics in Contrail Networking | 349

Centralized Logging | 357

Contrail Networking Metric List

Table 19: Cloud-Native Contrail Networking (CN2) Metric List

Metric Name	Type	Description
controller_state	gauge	Controller state (0=Functional, 1=Non-Functional).
controller_connection_status	gauge	Connection status (0=Up, 1=Down, 2=Initializing).
controller_bgp_router_output_queue_depth	gauge	BGP router output queue depth.
controller_bgp_router_num_bgp_peers	gauge	Number of BGP peers.
controller_bgp_router_num_up_bgp_peers	gauge	Number of up BGP peers.
controller_bgp_router_num_deleting_bgp_peers	gauge	Number of deleting BGP peers.
controller_bgp_router_num_bgpaas_peers	gauge	Number of BGPaas peers.
controller_bgp_router_num_up_bgpaas_peers	gauge	Number of up BGPaas peers.
controller_bgp_router_num_deleting_bgpaas_peers	gauge	Number of deleting BGPaas peers.
controller_bgp_router_num_xmpp_peers	gauge	Number of XMPP peers.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
controller_bgp_router_num_up_xmpp_peers	gauge	Number of up XMPP peers.
controller_bgp_router_num_deleting_xmpp_peers	gauge	Number of deleting XMPP peers.
controller_bgp_router_num_routing_instances	gauge	BGP router number of routing instances.
controller_bgp_router_num_deleting_routing_instances	gauge	BGP router number of deleting routing instances.
controller_bgp_router_num_service_chains	gauge	Number of service chains.
controller_bgp_router_num_down_service_chains	gauge	Number of down service chains.
controller_bgp_router_num_static_routes	gauge	Number of static routes.
controller_bgp_router_num_down_static_routes	gauge	Number of down static routes.
controller_bgp_router_ifmap_num_peer_clients	gauge	Number of IF-MAP peer clients.
controller_bgp_router_config_db_conn_status	gauge	Status of config database connection (0=Down, 1=Up).
controller_bgp_peer_state	gauge	BGP peer state (0=Idle, 1=Active, 2=Connect, 3=OpenSent, 4=OpenConfirm, 5=Established).
controller_bgp_peer_flaps_total	counter	BGP peer total flaps.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
controller_bgp_peer_received_messages_total	counter	Total BGP peer messages received.
controller_bgp_peer_received_open_messages_total	counter	Total BGP peer open messages received.
controller_bgp_peer_received_keepalive_messages_total	counter	Total BGP peer keepalive messages received.
controller_bgp_peer_received_notification_messages_total	counter	Total BGP peer notification messages received.
controller_bgp_peer_received_update_messages_total	counter	Total BGP peer update messages received.
controller_bgp_peer_received_close_messages_total	counter	Total BGP peer close messages received.
controller_bgp_peer_sent_messages_total	counter	Total BGP peer messages sent.
controller_bgp_peer_sent_open_messages_total	counter	Total BGP peer open messages sent.
controller_bgp_peer_sent_keepalive_messages_total	counter	Total BGP peer keepalive messages sent.
controller_bgp_peer_sent_notification_messages_total	counter	Total BGP peer notification messages sent.
controller_bgp_peer_sent_update_messages_total	counter	Total BGP peer update messages sent.
controller_bgp_peer_sent_close_messages_total	counter	Total BGP peer close messages sent.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
controller_bgp_peer_received_reachable_routes_total	counter	Total BGP peer reachable routes received.
controller_bgp_peer_received_unreachable_routes_total	counter	Total BGP peer unreachable routes received.
controller_bgp_peer_received_end_of_rib_total	counter	Total BGP peer end-of-RIB markers received.
controller_bgp_peer_sent_reachable_routes_total	counter	Total BGP peer reachable routes sent.
controller_bgp_peer_sent_unreachable_routes_total	counter	Total BGP peer unreachable routes sent.
controller_bgp_peer_sent_end_of_rib_total	counter	Total BGP peer end-of-RIB markers sent.
controller_bgp_peer_received_bytes_total	counter	Total BGP peer bytes received.
controller_bgp_peer_receive_socket_calls_total	counter	Total BGP peer receive socket calls.
controller_bgp_peer_blocked_receive_socket_calls_microsecond_duration_total	counter	BGP peer total microseconds blocked on socket receive calls.
controller_bgp_peer_blocked_receive_socket_calls_total	counter	Total BGP peer receive socket calls blocked.
controller_bgp_peer_sent_bytes_total	counter	Total BGP peer bytes sent.
controller_bgp_peer_send_socket_calls_total	counter	Total BGP peer send socket calls.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
controller_bgp_peer_blocked_send_socket_calls_microsecond_duration_total	counter	BGP peer total microseconds blocked on socket send calls.
controller_bgp_peer_blocked_send_socket_calls_total	counter	Total BGP peer send socket calls blocked.
controller_bgp_peer_route_update_error_bad_inet6_xml_token_total	counter	BGP peer total route update errors (bad inet6 XML token).
controller_bgp_peer_route_update_error_bad_inet6_prefix_total	counter	BGP peer total route update errors (bad inet6 prefix).
controller_bgp_peer_route_update_error_bad_inet6_nexthop_total	counter	BGP peer total route update errors (bad inet6 next hop).
controller_bgp_peer_route_update_error_bad_inet6_afi_safi_total	counter	BGP peer total route update errors (bad inet6 AFI/SAFI).
controller_bgp_peer_received_route_paths_total	counter	Total BGP peer route paths received.
controller_bgp_peer_received_route_primary_paths_total	counter	Total BGP peer route primary paths received.
controller_xmpp_peer_state	counter	XMPP peer state (0=Idle, 1=Active, 2=Connect, 3=OpenSent, 4=OpenConfirm, 5=Established).
controller_xmpp_peer_received_messages_total	counter	Total messages received from XMPP peer.
controller_xmpp_peer_received_open_messages_total	counter	Total open messages received from XMPP peer.
controller_xmpp_peer_received_keepalive_messages_total	counter	Total keepalive messages received from XMPP peer.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
controller_xmpp_peer_received_notification_messages_total	counter	Total notification messages received from XMPP peer.
controller_xmpp_peer_received_update_messages_total	counter	Total update messages received from XMPP peer.
controller_xmpp_peer_received_close_messages_total	counter	Total close messages received from XMPP peer.
controller_xmpp_peer_sent_messages_total	counter	Total messages sent to XMPP peer.
controller_xmpp_peer_sent_open_messages_total	counter	Total open messages sent to XMPP peer.
controller_xmpp_peer_sent_keepalive_messages_total	counter	Total keepalive messages sent to XMPP peer.
controller_xmpp_peer_sent_notification_messages_total	counter	Total notification messages sent to XMPP peer.
controller_xmpp_peer_sent_update_messages_total	counter	Total update messages sent to XMPP peer.
controller_xmpp_peer_sent_close_messages_total	counter	Total close messages sent to XMPP peer.
controller_xmpp_peer_received_reachable_routes_total	counter	Total reachable routes received from XMPP peer.
controller_xmpp_peer_received_unreachable_routes_total	counter	Total unreachable routes received from XMPP peer.
controller_xmpp_peer_received_end_of_rib_total	counter	Total end-of-RIB markers received from XMPP peer.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
controller_xmpp_peer_sent_reachable_routes_total	counter	Total reachable routes sent to XMPP peer.
controller_xmpp_peer_sent_unreachable_routes_total	counter	Total unreachable routes sent to XMPP peer.
controller_xmpp_peer_sent_end_of_rib_total	counter	Total end-of-RIB markers sent to XMPP peer.
controller_xmpp_peer_route_update_error_bad_inet6_xml_token_total	counter	XMPP peer total route update errors (bad inet6 XML token).
controller_xmpp_peer_route_update_error_bad_inet6_prefix_total	counter	XMPP peer total route update errors (bad inet6 prefix).
controller_xmpp_peer_route_update_error_bad_inet6_nexthop_total	counter	XMPP peer total route update errors (bad inet6 next hop).
controller_xmpp_peer_route_update_error_bad_inet6_afi_safi_total	counter	XMPP peer total route update errors (bad inet6 AFI/SAFI).
controller_xmpp_peer_received_route_paths_total	counter	Total XMPP peer route paths received.
controller_xmpp_peer_received_route_primary_paths_total	counter	Total XMPP peer route primary paths received.
controller_peer_received_reachable_routes_total	counter	Total reachable routes received from peer.
controller_peer_received_unreachable_routes_total	counter	Total unreachable routes received from peer.
controller_peer_received_end_of_rib_total	counter	Total end-of-RIB markers received from peer.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
controller_peer_sent_reachable_routes_total	counter	Total reachable routes sent to peer.
controller_peer_sent_unreachable_routes_total	counter	Total unreachable routes sent to peer.
controller_peer_sent_end_of_rib_total	counter	Total end-of-RIB markers sent to peer.
controller_virtual_network_routing_instance_ipv4_table_prefixes	gauge	Virtual network IPv4 routing table prefixes.
controller_virtual_network_routing_instance_ipv4_table_primary_paths	gauge	Virtual network IPv4 routing table primary paths.
controller_virtual_network_routing_instance_ipv4_table_secondary_paths	gauge	Virtual network IPv4 routing table secondary paths.
controller_virtual_network_routing_instance_ipv4_table_infeasible_paths	gauge	Virtual network IPv4 routing table infeasible paths.
controller_virtual_network_routing_instance_ipv4_table_total_paths	gauge	Virtual network IPv4 routing table total paths.
controller_virtual_network_routing_instance_ipv6_table_prefixes	gauge	Virtual network IPv6 routing table prefixes.
controller_virtual_network_routing_instance_ipv6_table_primary_paths	gauge	Virtual network IPv6 routing table primary paths.
controller_virtual_network_routing_instance_ipv6_table_secondary_paths	gauge	Virtual network IPv6 routing table secondary paths.
controller_virtual_network_routing_instance_ipv6_table_infeasible_paths	gauge	Virtual network IPv6 routing table infeasible paths.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
controller_virtual_network_routing_instance_ipv6_table_total_paths	gauge	Virtual network IPv6 routing table total paths.
controller_virtual_network_routing_instance_evpn_table_prefixes	gauge	Virtual network EVPN routing table prefixes.
controller_virtual_network_routing_instance_evpn_table_primary_paths	gauge	Virtual network EVPN routing table primary paths.
controller_virtual_network_routing_instance_evpn_table_secondary_paths	gauge	Virtual network EVPN routing table secondary paths.
controller_virtual_network_routing_instance_evpn_table_infeasible_paths	gauge	Virtual network EVPN routing table infeasible paths.
controller_virtual_network_routing_instance_evpn_table_total_paths	gauge	Virtual network EVPN routing table total paths.
controller_virtual_network_routing_instance_ermvpn_table_prefixes	gauge	Virtual network ERMVPN routing table prefixes.
controller_virtual_network_routing_instance_ermvpn_table_primary_paths	gauge	Virtual network ERMVPN routing table primary paths.
controller_virtual_network_routing_instance_ermvpn_table_secondary_paths	gauge	Virtual network ERMVPN routing table secondary paths.
controller_virtual_network_routing_instance_ermvpn_table_infeasible_paths	gauge	Virtual network ERMVPN routing table infeasible paths.
controller_virtual_network_routing_instance_ermvpn_table_total_paths	gauge	Virtual network ERMVPN routing table total paths.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
controller_virtual_network_routing_instance_mvpn_table_prefixes	gauge	Virtual network MVPN routing table prefixes.
controller_virtual_network_routing_instance_mvpn_table_primary_paths	gauge	Virtual network MVPN routing table primary paths.
controller_virtual_network_routing_instance_mvpn_table_secondary_paths	gauge	Virtual network MVPN routing table secondary paths.
controller_virtual_network_routing_instance_mvpn_table_infeasible_paths	gauge	Virtual network MVPN routing table infeasible paths.
controller_virtual_network_routing_instance_mvpn_table_total_paths	gauge	Virtual network MVPN routing table total paths.
virtual_router_cpu_1min_load_avg	gauge	Virtual router CPU 1 minute load average.
virtual_router_cpu_5min_load_avg	gauge	Virtual router CPU 5 minute load average.
virtual_router_cpu_15min_load_avg	gauge	Virtual router CPU 15 minute load average.
virtual_router_system_memory_bytes	gauge	Virtual router total system memory.
virtual_router_system_memory_free_bytes	gauge	Virtual router system memory free.
virtual_router_system_memory_used_bytes	gauge	Virtual router system memory used.
virtual_router_system_memory_cached_bytes	gauge	Virtual router system memory cached.
virtual_router_system_memory_buffers	gauge	Virtual router system memory buffers.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
virtual_router_virtual_memory_kilobytes	gauge	Virtual router virtual memory.
virtual_router_resident_memory_kilobytes	gauge	Virtual router resident memory.
virtual_router_peak_virtual_memory_bytes	gauge	Virtual router peak virtual memory.
virtual_router_phys_if_input_packets_total	counter	Total packets received by physical interface.
virtual_router_phys_if_output_packets_total	counter	Total packets sent by physical interface.
virtual_router_phys_if_input_bytes_total	counter	Total bytes received by physical interface.
virtual_router_phys_if_output_bytes_total	counter	Total bytes sent by physical interface.
virtual_router_input_packets_total	counter	Total packets received by virtual router.
virtual_router_output_packets_total	counter	Total packets sent by virtual router.
virtual_router_input_bytes_total	counter	Total bytes received by virtual router.
virtual_router_output_bytes_total	counter	Total bytes sent by virtual router.
virtual_router_flows_total	counter	Total virtual router flows.
virtual_router_aged_flows_total	counter	Total virtual router aged flows.
virtual_router_active_flows	gauge	Current virtual router active flows.
virtual_router_hold_flows	gauge	Current virtual router hold flows.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
virtual_router_added_flows_diff_total	gauge	Virtual router added flows since last sample.
virtual_router_exception_packets_total	counter	Total virtual router exception packets.
virtual_router_exception_packets_allowed_total	counter	Total virtual router exception packets allowed.
virtual_router_exception_packets_dropped_total	counter	Total virtual router exception packets dropped.
virtual_router_dropped_packets_total	counter	Total packets dropped.
virtual_router_vhost_dropped_packets_total	counter	Total virtual host packets dropped.
virtual_router_input_bandwidth_utilization	gauge	Ingress bandwidth of physical interface where the value is obtained by dividing the bandwidth computed in bits per second (bps) by speed of the physical interface.
virtual_router_output_bandwidth_utilization	gauge	Egress bandwidth of physical interface where the value is obtained by dividing the bandwidth computed in bps by speed of the physical interface.
virtual_router_vhost_interface_input_bytes_total	counter	Total bytes received by virtual host interface.
virtual_router_vhost_interface_output_bytes_total	counter	Total bytes sent by virtual host interface.
virtual_router_vhost_interface_input_packets_total	counter	Total packets received by virtual host interface.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
virtual_router_vhost_interface_output_packets_total	counter	Total packets sent by virtual host interface.
virtual_router_virtual_networks	gauge	Current number of virtual networks.
virtual_router_virtual_machines	gauge	Current number of virtual machines.
virtual_router_virtual_machine_interfaces	gauge	Current number of virtual machine interfaces.
virtual_router_interfaces_down	gauge	Current number of down interfaces.
virtual_router_agent_state	gauge	Virtual router agent state (0=Functional, 1=Non-Functional).
virtual_router_connection_status	gauge	Connection status (0=Up, 1=Down, 2=Initializing).
virtual_router_virtual_network_input_packets_total	counter	Total input packets received.
virtual_router_virtual_network_output_packets_total	counter	Total output packets sent.
virtual_router_virtual_network_input_bytes_total	counter	Total input bytes received.
virtual_router_virtual_network_output_bytes_total	counter	Total output bytes sent.
virtual_router_virtual_network_flows	gauge	Current number of flows.
virtual_router_virtual_network_ingress_flows	gauge	Current number of ingress flows.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
virtual_router_virtual_network_egress_flows	gauge	Current number of egress flows.
virtual_router_virtual_network_floating_ips	gauge	Current number of floating IP addresses.
virtual_router_virtual_network_flow_policy_rule_hits_total	counter	Total number of flow policy rule hits.
virtual_router_virtual_network_vrf_bridge_route_table_entries	gauge	Virtual routing and forwarding bridge route table current entries.
virtual_router_virtual_network_vrf_evpn_route_table_entries	gauge	Virtual routing and forwarding EVPN route table current entries.
virtual_router_virtual_network_vrf_inet4_unicast_route_table_entries	gauge	Virtual routing and forwarding inet4 unicast table current entries.
virtual_router_virtual_network_vrf_inet4_multicast_route_table_entries	gauge	Virtual routing and forwarding inet4 multicast table current entries.
virtual_router_virtual_network_vrf_inet6_unicast_route_table_entries	gauge	Virtual routing and forwarding inet6 unicast table current entries.
virtual_router_virtual_machine_interface_input_bytes_total	counter	Total input bytes received by virtual machine interface.
virtual_router_virtual_machine_interface_output_bytes_total	counter	Total output bytes sent by virtual machine interface.
virtual_router_virtual_machine_interface_input_packets_total	counter	Total input packets received by virtual machine interface.
virtual_router_virtual_machine_interface_output_packets_total	counter	Total output packets sent by virtual machine interface.

Table 19: Cloud-Native Contrail Networking (CN2) Metric List *(Continued)*

Metric Name	Type	Description
virtual_router_virtual_machine_interface_active_flows	gauge	Current virtual machine interface active flows.
virtual_router_virtual_machine_interface_hold_flows	gauge	Current virtual machine interface hold flows.
virtual_router_virtual_machine_interface_added_flows_diff_total	gauge	Virtual machine interface added flows since last sample.
virtual_router_virtual_machine_interface_dropped_packets_total	counter	Virtual machine interface total dropped packets.

RELATED DOCUMENTATION

Contrail Networking Analytics | 263
Kubernetes Metric List | 283
Cluster Node Metric List | 321
Contrail Networking Alert List | 339

Kubernetes Metric List

Table 20: Kubernetes Metric List

Metric Name	Type	Description
apiextensions_openapi_v2_regeneration_count	counter	[ALPHA] Counter of OpenAPI v2 spec regeneration count broken down by causing CRD name and reason.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
apiserver_admission_controller_admission_duration_seconds	histogram	[ALPHA] Admission controller latency histogram in seconds, identified by name and broken out for each operation and API resource and type (validate or admit).
apiserver_admission_step_admission_duration_seconds_summary	summary	[ALPHA] Admission sub-step latency summary in seconds, broken out for each operation and API resource and step type (validate or admit).
apiserver_admission_webhook_admission_duration_seconds	histogram	[ALPHA] Admission webhook latency histogram in seconds, identified by name and broken out for each operation and API resource and type (validate or admit).
apiserver_admission_webhook_rejection_count	counter	[ALPHA] Admission webhook rejection count, identified by name and broken out for each admission type (validating or admit) and operation. Additional labels specify an error type (calling_webhook_error or apiserver_internal_error if an error occurred; no_error otherwise) and optionally a non-zero rejection code if the webhook rejects the request with an HTTP status code (honored by the apiserver when the code is greater than or equal to 400). Codes greater than 600 are truncated to 600, to keep the metrics bounded by cardinality.
apiserver_audit_event_total	counter	[ALPHA] Counter of audit events generated and sent to the audit backend.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
`apiserver_audit_requests_rejected_total`	counter	[ALPHA] Counter of apiserver requests rejected due to an error in the audit logging backend.
`apiserver_client_certificate_expiration_seconds`	histogram	[ALPHA] Distribution of the remaining lifetime on the certificate used to authenticate a request.
`apiserver_current_inflight_requests`	gauge	[ALPHA] Maximal number of currently used inflight requests limit of this apiserver per request kind in the last second.
`apiserver_current_inqueue_requests`	gauge	[ALPHA] Maximal number of queued requests in this apiserver per request kind in the last second.
`apiserver_envelope_encryption_dek_cache_fill_percent`	gauge	[ALPHA] Percent of the cache slots currently occupied by cached data encryption keys (DEK)s.
`apiserver_flowcontrol_current_executing_requests`	gauge	[ALPHA] Number of requests currently executing in the API Priority and Fairness system.
`apiserver_flowcontrol_current_inqueue_requests`	gauge	[ALPHA] Number of requests currently pending in queues of the API Priority and Fairness system.
`apiserver_flowcontrol_dispatched_requests_total`	counter	[ALPHA] Number of requests released by the API Priority and Fairness system for service.
`apiserver_flowcontrol_priority_level_request_count_samples`	histogram	[ALPHA] Periodic observations of the number of requests.
`apiserver_flowcontrol_priority_level_request_count_watermarks`	histogram	[ALPHA] Watermarks of the number of requests.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
apiserver_flowcontrol_read_vs_write_request_count_samples	histogram	[ALPHA] Periodic observations of the number of requests.
apiserver_flowcontrol_read_vs_write_request_count_watermarks	histogram	[ALPHA] Watermarks of the number of requests.
apiserver_flowcontrol_request_concurrency_limit	gauge	[ALPHA] Shared concurrency limit in the API Priority and Fairness system.
apiserver_flowcontrol_request_execution_seconds	histogram	[ALPHA] Duration of request execution in the API Priority and Fairness system.
apiserver_flowcontrol_request_queue_length_after_enqueue	histogram	[ALPHA] Length of queue in the API Priority and Fairness system, as seen by each request after it is enqueued.
apiserver_flowcontrol_request_wait_duration_seconds	histogram	[ALPHA] Length of time a request spent waiting in its queue.
apiserver_init_events_total	counter	[ALPHA] Counter of init events processed in watchcache broken by resource type.
apiserver_longrunning_gauge	gauge	[ALPHA] Gauge of all active long-running apiserver requests broken out by verb, group, version, resource, scope and component. Not all requests are tracked this way.
apiserver_registered_watchers	gauge	[ALPHA] Number of currently registered watchers for a given resources.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
apiserver_request_duration_seconds	histogram	[ALPHA] Response latency distribution in seconds for each verb, dry run value, group, version, resource, subresource, scope, and component.
apiserver_request_filter_duration_seconds	histogram	[ALPHA] Request filter latency distribution in seconds, for each filter type.
apiserver_request_total	counter	[ALPHA] Counter of apiserver requests broken out for each verb, dry run value, group, version, resource, scope, component, and HTTP response contentType and code.
apiserver_requested_deprecated_apis	gauge	[ALPHA] Gauge of deprecated APIs that have been requested, broken out by API group, version, resource, subresource, and removed_release.
apiserver_response_sizes	histogram	[ALPHA] Response size distribution in bytes for each group, version, verb, resource, subresource, scope, and component.
apiserver_selfrequest_total	counter	[ALPHA] Counter of apiserver self-requests broken out for each verb, API resource, and subresource.
apiserver_storage_data_key_generation_duration_seconds	histogram	[ALPHA] Latencies in seconds of DEK generation operations.
apiserver_storage_data_key_generation_failures_total	counter	[ALPHA] Total number of failed DEK generation operations.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
apiserver_storage_envelope_transformation_cache_misses_total	counter	[ALPHA] Total number of cache misses while accessing key decryption key (KDK).
apiserver_tls_handshake_errors_total	counter	[ALPHA] Number of requests dropped with 'TLS handshake error from' error.
apiserver_watch_events_sizes	histogram	[ALPHA] Watch event size distribution in bytes.
apiserver_watch_events_total	counter	[ALPHA] Number of events sent in watch clients.
authenticated_user_requests	counter	[ALPHA] Counter of authenticated requests broken out by username.
authentication_attempts	counter	[ALPHA] Counter of authenticated attempts.
authentication_duration_seconds	histogram	[ALPHA] Authentication duration in seconds broken out by result.
authentication_token_cache_active_fetch_count	gauge	[ALPHA]
authentication_token_cache_fetch_total	counter	[ALPHA]
authentication_token_cache_request_duration_seconds	histogram	[ALPHA]
authentication_token_cache_request_total	counter	[ALPHA]

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
cadvisor_version_info	gauge	A metric with a constant '1' value labeled by kernel version, OS version, docker version, cadvisor version, and cadvisor revision.
container_cpu_cfs_periods_total	counter	Number of elapsed enforcement period intervals.
container_cpu_cfs_throttled_periods_total	counter	Number of throttled period intervals.
container_cpu_cfs_throttled_seconds_total	counter	Total time duration the container has been throttled.
container_cpu_load_average_10s	gauge	Value of container CPU load average over the last 10 seconds.
container_cpu_system_seconds_total	counter	Cumulative system CPU time consumed in seconds.
container_cpu_usage_seconds_total	counter	Cumulative CPU time consumed in seconds.
container_cpu_user_seconds_total	counter	Cumulative user CPU time consumed in seconds.
container_file_descriptors	gauge	Number of open file descriptors for the container.
container_fs_inodes_free	gauge	Number of available Inodes.
container_fs_inodes_total	gauge	Number of Inodes.
container_fs_io_current	gauge	Number of I/Os currently in progress.
container_fs_io_time_seconds_total	counter	Cumulative count of seconds spent doing I/Os.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
container_fs_io_time_weighted_seconds_total	counter	Cumulative weighted I/O time in seconds.
container_fs_limit_bytes	gauge	Number of bytes that the container on this filesystem can consume.
container_fs_read_seconds_total	counter	Cumulative count of seconds spent reading.
container_fs_reads_bytes_total	counter	Cumulative count of bytes read.
container_fs_reads_merged_total	counter	Cumulative count of reads merged.
container_fs_reads_total	counter	Cumulative count of reads completed.
container_fs_sector_reads_total	counter	Cumulative count of sector reads completed.
container_fs_sector_writes_total	counter	Cumulative count of sector writes completed.
container_fs_usage_bytes	gauge	Number of bytes that the container on this filesystem can consume.
container_fs_write_seconds_total	counter	Cumulative count of seconds spent writing.
container_fs_writes_bytes_total	counter	Cumulative count of bytes written.
container_fs_writes_merged_total	counter	Cumulative count of writes merged.
container_fs_writes_total	counter	Cumulative count of writes completed.
container_last_seen	gauge	Last time the exporter recognized a container.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
container_memory_cache	gauge	Number of bytes of page cache memory.
container_memory_failcnt	counter	Number of memory usage hit limits.
container_memory_failures_total	counter	Cumulative count of memory allocation failures.
container_memory_mapped_file	gauge	Size of memory mapped files in bytes.
container_memory_max_usage_bytes	gauge	Maximum memory usage recorded in bytes.
container_memory_rss	gauge	Size of RSS in bytes.
container_memory_swap	gauge	Container swap usage in bytes.
container_memory_usage_bytes	gauge	Current memory usage in bytes, including all memory regardless of when it was accessed.
container_memory_working_set_bytes	gauge	Current working set in bytes.
container_network_receive_bytes_total	counter	Cumulative count of bytes received.
container_network_receive_errors_total	counter	Cumulative count of errors encountered while receiving.
container_network_receive_packets_dropped_total	counter	Cumulative count of packets dropped while receiving.
container_network_receive_packets_total	counter	Cumulative count of packets received.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
container_network_transmit_bytes_total	counter	Cumulative count of bytes transmitted.
container_network_transmit_errors_total	counter	Cumulative count of errors encountered while transmitting.
container_network_transmit_packets_dropped_total	counter	Cumulative count of packets dropped while transmitting.
container_network_transmit_packets_total	counter	Cumulative count of packets transmitted.
container_processes	gauge	Number of processes running inside the container.
container_scrape_error	gauge	1 if there was an error while getting container metrics, 0 otherwise.
container_sockets	gauge	Number of open sockets for the container.
container_spec_cpu_period	gauge	CPU period of the container.
container_spec_cpu_quota	gauge	CPU quota of the container.
container_spec_cpu_shares	gauge	CPU share of the container.
container_spec_memory_limit_bytes	gauge	Memory limit for the container.
container_spec_memory_reservation_limit_bytes	gauge	Memory reservation limit for the container.
container_spec_memory_swap_limit_bytes	gauge	Memory swap limit for the container.
container_start_time_seconds	gauge	Start time of the container since Unix epoch in seconds.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
container_tasks_state	gauge	Number of tasks in a given state.
container_threads	gauge	Number of threads running inside the container.
container_threads_max	gauge	Maximum number of threads allowed inside the container; infinity if value is zero.
container_ulimits_soft	gauge	Soft ulimit values for the container root process. Unlimited if -1, except priority and nice.
coredns_build_info	gauge	A metric with a constant '1' value labeled by version, revision, and goversion from which CoreDNS was built.
coredns_cache_entries	gauge	The number of elements in the cache.
coredns_cache_hits_total	counter	The count of cache hits.
coredns_cache_misses_total	counter	The count of cache misses.
coredns_dns_request_duration_seconds	histogram	Histogram of the time (in seconds) each request took.
coredns_dns_request_size_bytes	histogram	Size of the EDNS0 UDP buffer in bytes (64K for TCP).
coredns_dns_requests_total	counter	Counter of DNS requests made per zone, protocol, and family.
coredns_dns_response_size_bytes	histogram	Size of the returned response in bytes.
coredns_dns_responses_total	counter	Counter of response status codes.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
coredns_forward_healthcheck_failures_total	counter	Counter of the number of failed healthchecks.
coredns_forward_max_concurrent_rejects_total	counter	Counter of the number of queries rejected because the concurrent queries were at maximum.
coredns_forward_request_duration_seconds	histogram	Histogram of the time each request took.
coredns_forward_requests_total	counter	Counter of requests made per upstream.
coredns_forward_responses_total	counter	Counter of requests made per upstream.
coredns_health_request_duration_seconds	histogram	Histogram of the time (in seconds) each request took.
coredns_panics_total	counter	A metric that counts the number of panics.
coredns_plugin_enabled	gauge	A metric that indicates whether a plugin is enabled on a per-server and zone basis.
etcd_db_total_size_in_bytes	gauge	[ALPHA] Total size of the etcd database file physically allocated in bytes.
etcd_object_counts	gauge	[ALPHA] Number of stored objects at the time of last check, split by kind.
etcd_request_duration_seconds	histogram	[ALPHA] Etcd request latency in seconds for each operation and object type.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_certificatesigningrequest_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_certificatesigningrequest_cert_length	gauge	Length of the issued certificate.
kube_certificatesigningrequest_condition	gauge	The number of each certificatesigningrequest condition.
kube_certificatesigningrequest_created	gauge	Unix creation timestamp.
kube_certificatesigningrequest_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_configmap_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_configmap_created	gauge	Unix creation timestamp.
kube_configmap_info	gauge	Information about configmap.
kube_configmap_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_configmap_metadata_resource_version	gauge	Resource version representing a specific version of the configmap.
kube_cronjob_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_cronjob_created	gauge	Unix creation timestamp.
kube_cronjob_info	gauge	Info about cronjob.
kube_cronjob_labels	gauge	Kubernetes labels converted to Prometheus labels.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_cronjob_metadata_resource_version	gauge	Resource version representing a specific version of the cronjob.
kube_cronjob_next_schedule_time	gauge	Next time the cronjob should be scheduled. The time after lastScheduleTime, or after the cron job's creation time if it has never been scheduled. Use this to determine if the job is delayed.
kube_cronjob_spec_failed_job_history_limit	gauge	Failed job history limit tells the controller how many failed jobs should be preserved.
kube_cronjob_spec_starting_deadline_seconds	gauge	Deadline in seconds for starting the job if it misses scheduled time for any reason.
kube_cronjob_spec_successful_job_history_limit	gauge	Successful job history limit tells the controller how many completed jobs should be preserved.
kube_cronjob_spec_suspend	gauge	Suspend flag tells the controller to suspend subsequent executions.
kube_cronjob_status_active	gauge	Active holds pointers to currently running jobs.
kube_cronjob_status_last_schedule_time	gauge	LastScheduleTime keeps information about the last time the job was successfully scheduled.
kube_daemonset_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_daemonset_created	gauge	Unix creation timestamp.
kube_daemonset_labels	gauge	Kubernetes labels converted to Prometheus labels.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_daemonset_metadata_generation	gauge	Sequence number representing a specific generation of the desired state.
kube_daemonset_status_current_number_scheduled	gauge	The number of nodes running at least one daemon pod.
kube_daemonset_status_desired_number_scheduled	gauge	The number of nodes that should be running the daemon pod.
kube_daemonset_status_number_available	gauge	The number of nodes that should be running the daemon pod and have one or more of the daemon pods running and available.
kube_daemonset_status_number_misscheduled	gauge	The number of nodes running a daemon pod that should not be running a daemon pod.
kube_daemonset_status_number_ready	gauge	The number of nodes that should be running the daemon pod and have one or more of the daemon pods running and ready.
kube_daemonset_status_number_unavailable	gauge	The number of nodes that should be running the daemon pod but have none of the daemon pods running and available.
kube_daemonset_status_observed_generation	gauge	The most recent generation observed by the daemon set controller.
kube_daemonset_status_updated_number_scheduled	gauge	The total number of nodes that are running an updated daemon pod.
kube_deployment_annotations	gauge	Kubernetes annotations converted to Prometheus labels.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_deployment_created	gauge	Unix creation timestamp.
kube_deployment_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_deployment_metadata_generation	gauge	Sequence number representing a specific generation of the desired state.
kube_deployment_spec_paused	gauge	Whether the deployment is paused and will not be processed by the deployment controller.
kube_deployment_spec_replicas	gauge	Number of desired pods for a deployment.
kube_deployment_spec_strategy_rollingupdate_max_surge	gauge	Maximum number of replicas that can be scheduled above the desired number of replicas during a rolling update of a deployment.
kube_deployment_spec_strategy_rollingupdate_max_unavailable	gauge	Maximum number of unavailable replicas during a rolling update of a deployment.
kube_deployment_status_condition	gauge	The current status conditions of a deployment.
kube_deployment_status_observed_generation	gauge	The generation observed by the deployment controller.
kube_deployment_status_replicas	gauge	The number of replicas per deployment.
kube_deployment_status_replicas_available	gauge	The number of available replicas per deployment.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_deployment_status_replicas_ready	gauge	The number of ready replicas per deployment.
kube_deployment_status_replicas_unavailable	gauge	The number of unavailable replicas per deployment.
kube_deployment_status_replicas_updated	gauge	The number of updated replicas per deployment.
kube_endpoint_address_available	gauge	Number of addresses available in the endpoint.
kube_endpoint_address_not_ready	gauge	Number of addresses not ready in the endpoint.
kube_endpoint_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_endpoint_created	gauge	Unix creation timestamp.
kube_endpoint_info	gauge	Information about the endpoint.
kube_endpoint_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_endpoint_ports	gauge	Information about the endpoint ports.
kube_horizontalpodautoscaler_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_horizontalpodautoscaler_info	gauge	Information about this autoscaler.
kube_horizontalpodautoscaler_labels	gauge	Kubernetes labels converted to Prometheus labels.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_horizontalpodautoscaler_metadata_generation	gauge	The generation observed by the HorizontalPodAutoscaler controller.
kube_horizontalpodautoscaler_spec_max_replicas	gauge	Upper limit for the number of pods that is set by the autoscaler; cannot be smaller number than MinReplicas.
kube_horizontalpodautoscaler_spec_min_replicas	gauge	Lower limit for the number of pods that is set by the autoscaler, default 1.
kube_horizontalpodautoscaler_spec_target_metric	gauge	The metric specifications used by this autoscaler when calculating the desired replica count.
kube_horizontalpodautoscaler_status_condition	gauge	The condition of this autoscaler.
kube_horizontalpodautoscaler_status_current_replicas	gauge	Current number of replicas of pods managed by this autoscaler.
kube_horizontalpodautoscaler_status_desired_replicas	gauge	Desired number of replicas of pods managed by this autoscaler.
kube_ingress_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_ingress_created	gauge	Unix creation timestamp.
kube_ingress_info	gauge	Information about ingress.
kube_ingress_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_ingress_metadata_resource_version	gauge	Resource version representing a specific version of ingress.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_ingress_path	gauge	Ingress host, paths, and backend service information.
kube_ingress_tls	gauge	Ingress TLS host and secret information.
kube_job_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_job_complete	gauge	The job has completed its execution.
kube_job_created	gauge	Unix creation timestamp.
kube_job_failed	gauge	The job has failed its execution.
kube_job_info	gauge	Information about the job.
kube_job_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_job_owner	gauge	Information about the job's owner.
kube_job_spec_active_deadline_seconds	gauge	How long (in seconds) the job can be active after the startTime before the system tries to terminate it.
kube_job_spec_completions	gauge	The desired number of successfully finished pods the job should be run with.
kube_job_spec_parallelism	gauge	The maximum desired number of pods the job should run at any given time.
kube_job_status_active	gauge	The number of actively running pods.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_job_status_completion_time	gauge	The completionTime represents time when the job was completed.
kube_job_status_failed	gauge	The number of pods that reached Phase Failed, and the reason for failure.
kube_job_status_start_time	gauge	The startTime represents thetime when the job was acknowledged by the Job Manager.
kube_job_status_succeeded	gauge	The number of pods that reached Phase Succeeded.
kube_limitrange	gauge	Information about limit range.
kube_limitrange_created	gauge	Unix creation timestamp.
kube_mutatingwebhookconfiguration_created	gauge	Unix creation timestamp.
kube_mutatingwebhookconfiguration_info	gauge	Information about the MutatingWebhookConfiguration.
kube_mutatingwebhookconfiguration_metadata_resource_version	gauge	Resource version representing a specific version of the MutatingWebhookConfiguration.
kube_namespace_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_namespace_created	gauge	Unix creation timestamp.
kube_namespace_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_namespace_status_condition	gauge	The condition of a namespace.

Table 20: Kubernetes Metric List ***(Continued)***

Metric Name	Type	Description
kube_namespace_status_phase	gauge	Kubernetes namespace status phase.
kube_networkpolicy_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_networkpolicy_created	gauge	Unix creation timestamp of a network policy.
kube_networkpolicy_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_networkpolicy_spec_egress_rules	gauge	Number of egress rules on the networkpolicy.
kube_networkpolicy_spec_ingress_rules	gauge	Number of ingress rules on the networkpolicy.
kube_node_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_node_created	gauge	Unix creation timestamp.
kube_node_info	gauge	Information about a cluster node.
kube_node_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_node_role	gauge	The role of a cluster node.
kube_node_spec_taint	gauge	The taint of a cluster node.
kube_node_spec_unschedulable	gauge	Whether a node can schedule new pods.
kube_node_status_allocatable	gauge	The allocatable for different resources of a node that are available for scheduling.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_node_status_capacity	gauge	The capacity for different resources of a node.
kube_node_status_condition	gauge	The condition of a cluster node.
kube_persistentvolume_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_persistentvolume_capacity_bytes	gauge	Persistentvolume capacity in bytes.
kube_persistentvolume_claim_ref	gauge	Information about the Persistent Volume Claim Reference.
kube_persistentvolume_info	gauge	Information about persistentvolume.
kube_persistentvolume_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_persistentvolume_status_phase	gauge	The phase indicates if a volume is available, bound to a claim, or released by a claim.
kube_persistentvolumeclaim_access_mode	gauge	The access mode(s) specified by the persistent volume claim.
kube_persistentvolumeclaim_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_persistentvolumeclaim_info	gauge	Information about persistent volume claim.
kube_persistentvolumeclaim_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_persistentvolumeclaim_resource_requests_storage_bytes	gauge	The capacity of storage requested by the persistent volume claim.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_persistentvolumeclaim_status_condition	gauge	Information about status of different conditions of persistent volume claim.
kube_persistentvolumeclaim_status_phase	gauge	The phase the persistent volume claim is currently in.
kube_pod_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_pod_completion_time	gauge	Completion time in Unix timestamp for a pod.
kube_pod_container_info	gauge	Information about a container in a pod.
kube_pod_container_resource_limits	gauge	The number of resource limits requested by a container.
kube_pod_container_resource_requests	gauge	The number of resources requested by a container.
kube_pod_container_state_started	gauge	Start time in Unix timestamp for a pod container.
kube_pod_container_status_last_terminated_reason	gauge	Describes the last reason the container was in a terminated state.
kube_pod_container_status_ready	gauge	Describes whether the container's readiness check succeeded.
kube_pod_container_status_restarts_total	counter	The number of restarts per container.
kube_pod_container_status_running	gauge	Describes whether the container is currently in running state.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_pod_container_status_terminated	gauge	Describes whether the container is currently in a terminated state.
kube_pod_container_status_terminated_reason	gauge	Describes the reason the container is currently in a terminated state.
kube_pod_container_status_waiting	gauge	Describes whether the container is currently in a waiting state.
kube_pod_container_status_waiting_reason	gauge	Describes the reason the container is currently in a waiting state.
kube_pod_created	gauge	Unix creation timestamp.
kube_pod_deletion_timestamp	gauge	Unix deletion timestamp.
kube_pod_info	gauge	Information about the pod.
kube_pod_init_container_info	gauge	Information about an init container in a pod.
kube_pod_init_container_resource_limits	gauge	The number of requested resource limits by an init container.
kube_pod_init_container_resource_requests	gauge	The number of resource requests by an init container.
kube_pod_init_container_status_last_terminated_reason	gauge	Describes the last reason the init container was in a terminated state.
kube_pod_init_container_status_ready	gauge	Describes whether the init container's readiness check succeeded.
kube_pod_init_container_status_restarts_total	counter	The number of restarts for the init container.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_pod_init_container_status_running	gauge	Describes whether the init container is currently in a running state.
kube_pod_init_container_status_terminated	gauge	Describes whether the init container is currently in a terminated state.
kube_pod_init_container_status_terminated_reason	gauge	Describes the last reason the init container was in a terminated state.
kube_pod_init_container_status_ready	gauge	Describes whether the init container's readiness check succeeded.
kube_pod_init_container_status_restarts_total	counter	The number of restarts for the init container.
kube_pod_init_container_status_running	gauge	Describes whether the init container is currently in a running state.
kube_pod_init_container_status_terminated	gauge	Describes whether the init container is currently in a terminated state.
kube_pod_init_container_status_terminated_reason	gauge	Describes the reason the init container is currently in a terminated state.
kube_pod_init_container_status_waiting	gauge	Describes whether the init container is currently in a waiting state.
kube_pod_init_container_status_waiting_reason	gauge	Describes the reason the init container is currently in a waiting state.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_pod_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_pod_overhead_cpu_cores	gauge	The pod overhead in regard to CPU cores associated with running a pod.
kube_pod_overhead_memory_bytes	gauge	The pod overhead in regard to memory associated with running a pod.
kube_pod_owner	gauge	Information about the pod's owner.
kube_pod_restart_policy	gauge	Describes the restart policy in use by this pod.
kube_pod_runtimeclass_name_info	gauge	The runtimeclass associated with the pod.
kube_pod_spec_volumes_persistentvolumeclaims_info	gauge	Information about persistentvolumeclaim volumes in a pod.
kube_pod_spec_volumes_persistentvolumeclaims_readonly	gauge	Describes whether a persistentvolumeclaim is mounted as read only.
kube_pod_start_time	gauge	Start time in the Unix timestamp for a pod.
kube_pod_status_phase	gauge	The pod's current phase.
kube_pod_status_ready	gauge	Describes whether the pod is ready to serve requests.
kube_pod_status_reason	gauge	The pod status reasons.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_pod_status_scheduled	gauge	Describes the status of the scheduling process for the pod.
kube_pod_status_scheduled_time	gauge	Unix timestamp when a pod moved into scheduled status.
kube_pod_status_unschedulable	gauge	Describes the unschedulable status for the pod.
kube_poddisruptionbudget_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_poddisruptionbudget_created	gauge	Unix creation timestamp.
kube_poddisruptionbudget_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_poddisruptionbudget_status_current_healthy	gauge	Current number of healthy pods.
kube_poddisruptionbudget_status_desired_healthy	gauge	Minimum desired number of healthy pods.
kube_poddisruptionbudget_status_expected_pods	gauge	Total number of pods counted by this disruption budget.
kube_poddisruptionbudget_status_observed_generation	gauge	Most recent generation observed when updating this disruption budget status.
kube_poddisruptionbudget_status_pod_disruptions_allowed	gauge	Number of pod disruptions that are currently allowed.
kube_replicaset_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_replicaset_created	gauge	Unix creation timestamp.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_replicaset_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_replicaset_metadata_generation	gauge	Sequence number representing a specific generation of the desired state.
kube_replicaset_owner	gauge	Information about the ReplicaSet's owner.
kube_replicaset_spec_replicas	gauge	Number of desired pods for a ReplicaSet.
kube_replicaset_status_fully_labeled_replicas	gauge	The number of fully labeled replicas per ReplicaSet.
kube_replicaset_status_observed_generation	gauge	The generation observed by the ReplicaSet controller.
kube_replicaset_status_ready_replicas	gauge	The number of ready replicas per ReplicaSet.
kube_replicaset_status_replicas	gauge	The number of replicas per ReplicaSet.
kube_replicationcontroller_created	gauge	Unix creation timestamp.
kube_replicationcontroller_metadata_generation	gauge	Sequence number representing a specific generation of the desired state.
kube_replicationcontroller_owner	gauge	Information about the ReplicationController's owner.
kube_replicationcontroller_spec_replicas	gauge	Number of desired pods for a ReplicationController.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_replicationcontroller_status_available_replicas	gauge	The number of available replicas per ReplicationController.
kube_replicationcontroller_status_fully_labeled_replicas	gauge	The number of fully labeled replicas per ReplicationController.
kube_replicationcontroller_status_observed_generation	gauge	The generation observed by the ReplicationController controller.
kube_replicationcontroller_status_ready_replicas	gauge	The number of ready replicas per ReplicationController.
kube_replicationcontroller_status_replicas	gauge	The number of replicas per ReplicationController.
kube_resourcequota	gauge	Information about resource quota.
kube_resourcequota_created	gauge	Unix creation timestamp.
kube_secret_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_secret_created	gauge	Unix creation timestamp.
kube_secret_info	gauge	Information about secret.
kube_secret_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_secret_metadata_resource_version	gauge	Resource version representing a specific version of secret.
kube_secret_type	gauge	Type about secret.
kube_service_annotations	gauge	Kubernetes annotations converted to Prometheus labels.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_service_created	gauge	Unix creation timestamp.
kube_service_info	gauge	Information about service.
kube_service_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_service_spec_external_ip	gauge	One series for each service external IP.
kube_service_spec_type	gauge	Specifies the service type.
kube_service_status_load_balancer_ingress	gauge	Service load balancer ingress status.
kube_statefulset_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_statefulset_created	gauge	Unix creation timestamp.
kube_statefulset_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_statefulset_metadata_generation	gauge	Sequence number representing a specific generation of the desired state for the StatefulSet.
kube_statefulset_replicas	gauge	Number of desired pods for a StatefulSet.
kube_statefulset_status_current_revision	gauge	Indicates the version of the StatefulSet used to generate pods in the sequence (0,currentReplicas).
kube_statefulset_status_observed_generation	gauge	The generation observed by the StatefulSet controller.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_statefulset_status_replicas	gauge	The number of replicas per StatefulSet.
kube_statefulset_status_replicas_available	gauge	The number of available replicas per StatefulSet.
kube_statefulset_status_replicas_current	gauge	The number of current replicas per StatefulSet.
kube_statefulset_status_replicas_ready	gauge	The number of ready replicas per StatefulSet.
kube_statefulset_status_replicas_updated	gauge	The number of updated replicas per StatefulSet.
kube_statefulset_status_update_revision	gauge	Indicates the version of the StatefulSet used to generate pods in the sequence (replicas-updatedReplicas,replicas).
kube_storageclass_annotations	gauge	Kubernetes annotations converted to Prometheus labels.
kube_storageclass_created	gauge	Unix creation timestamp.
kube_storageclass_info	gauge	Information about storageclass.
kube_storageclass_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_validatingwebhookconfiguration_created	gauge	Unix creation timestamp.
kube_validatingwebhookconfiguration_info	gauge	Information about the ValidatingWebhookConfiguration.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kube_validatingwebhookconfiguration_metadata_resource_version	gauge	Resource version representing a specific version of the ValidatingWebhookConfiguration.
kube_volumeattachment_created	gauge	Unix creation timestamp.
kube_volumeattachment_info	gauge	Information about volumeattachment.
kube_volumeattachment_labels	gauge	Kubernetes labels converted to Prometheus labels.
kube_volumeattachment_spec_source_persistentvolume	gauge	PersistentVolume source reference.
kube_volumeattachment_status_attached	gauge	Information about volumeattachment.
kube_volumeattachment_status_attachment_metadata	gauge	The volumeattachment metadata.
kubelet_certificate_manager_client_expiration_renew_errors	counter	[ALPHA] Counter of certificate renewal errors.
kubelet_certificate_manager_client_ttl_seconds	gauge	[ALPHA] Gauge of the time-to-live (TTL) of the Kubelet's client certificate. The value is in seconds until certificate expiry (negative if already expired). If the client certificate is invalid or unused, the value will be +INF.
kubelet_cgroup_manager_duration_seconds	histogram	[ALPHA] Duration in seconds for cgroup manager operations. Broken down by method.
kubelet_container_log_filesystem_used_bytes	gauge	[ALPHA] Bytes used by the container's logs on the filesystem.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kubelet_containers_per_pod_count	histogram	[ALPHA] The number of containers per pod.
kubelet_docker_operations_duration_seconds	histogram	[ALPHA] Latency in seconds of Docker operations. Broken down by operation type.
kubelet_docker_operations_errors_total	counter	[ALPHA] Cumulative number of Docker operation errors by operation type.
kubelet_docker_operations_total	counter	[ALPHA] Cumulative number of Docker operations by operation type.
kubelet_http_inflight_requests	gauge	[ALPHA] Number of the inflight http requests.
kubelet_http_requests_duration_seconds	histogram	[ALPHA] Duration in seconds to serve http requests.
kubelet_http_requests_total	counter	[ALPHA] Number of the http requests received since the server started.
kubelet_network_plugin_operations_duration_seconds	histogram	[ALPHA] Latency in seconds of network plugin operations, broken down by operation type.
kubelet_network_plugin_operations_total	counter	[ALPHA] Cumulative number of network plugin operations by operation type.
kubelet_node_config_error	gauge	[ALPHA] This metric is true (1) if the node is experiencing a configuration-related error, false (0) otherwise.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kubelet_node_name	gauge	[ALPHA] The node's name. The count is always 1.
kubelet_pleg_discard_events	counter	[ALPHA] The number of discard events in PLEG.
kubelet_pleg_last_seen_seconds	gauge	[ALPHA] Timestamp in seconds when PLEG was last seen active.
kubelet_pleg_relist_duration_seconds	histogram	[ALPHA] Duration in seconds for relisting pods in PLEG.
kubelet_pleg_relist_interval_seconds	histogram	[ALPHA] Interval in seconds between relisting in PLEG.
kubelet_pod_start_duration_seconds	histogram	[ALPHA] Duration in seconds for a single pod to go from pending to running.
kubelet_pod_worker_duration_seconds	histogram	[ALPHA] Duration in seconds to sync a single pod. Broken down by operation type: create, update, or sync.
kubelet_pod_worker_start_duration_seconds	histogram	[ALPHA] Duration in seconds from seeing a pod to starting a worker.
kubelet_run_podsandbox_duration_seconds	histogram	[ALPHA] Duration in seconds of the run_podsandbox operations. Broken down by RuntimeClass.Handler.
kubelet_running_containers	gauge	[ALPHA] Number of containers currently running.
kubelet_running_pods	gauge	[ALPHA] Number of pods currently running.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kubelet_runtime_operations_duration_seconds	histogram	[ALPHA] Duration in seconds of runtime operations. Broken down by operation type.
kubelet_runtime_operations_errors_total	counter	[ALPHA] Cumulative number of runtime operation errors by operation type.
kubelet_runtime_operations_total	counter	[ALPHA] Cumulative number of runtime operations by operation type.
kubelet_volume_stats_available_bytes	gauge	[ALPHA] Number of available bytes in the volume.
kubelet_volume_stats_capacity_bytes	gauge	[ALPHA] Capacity in bytes of the volume.
kubelet_volume_stats_inodes	gauge	[ALPHA] Maximum number of inodes in the volume.
kubelet_volume_stats_inodes_free	gauge	[ALPHA] Number of free inodes in the volume.
kubelet_volume_stats_inodes_used	gauge	[ALPHA] Number of used inodes in the volume.
kubelet_volume_stats_used_bytes	gauge	[ALPHA] Number of used bytes in the volume.
kubeproxy_network_programming_duration_seconds	histogram	[ALPHA] In Cluster Network Programming Latency in seconds.
kubeproxy_sync_proxy_rules_duration_seconds	histogram	[ALPHA] SyncProxyRules latency in seconds.
kubeproxy_sync_proxy_rules_endpoint_changes_pending	gauge	[ALPHA] Pending proxy rules Endpoint changes.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
kubeproxy_sync_proxy_rules_endpoint_changes_total	counter	[ALPHA] Cumulative proxy rules Endpoint changes.
kubeproxy_sync_proxy_rules_iptables_restore_failures_total	counter	[ALPHA] Cumulative proxy iptables restore failures.
kubeproxy_sync_proxy_rules_last_queued_timestamp_seconds	gauge	[ALPHA] The last time a sync of proxy rules was queued.
kubeproxy_sync_proxy_rules_last_timestamp_seconds	gauge	[ALPHA] The last time proxy rules were successfully synced.
kubeproxy_sync_proxy_rules_service_changes_pending	gauge	[ALPHA] Pending proxy rules Service changes.
kubeproxy_sync_proxy_rules_service_changes_total	counter	[ALPHA] Cumulative proxy rules Service changes.
kubernetes_build_info	gauge	[ALPHA] A metric with a constant '1' value labeled by major, minor, git version, git commit, git tree state, build date, Go version, and compiler from which Kubernetes was built, and platform on which it is running.
prober_probe_total	counter	[ALPHA] Cumulative number of a liveness, readiness, or startup probe for a container by result.
process_cpu_seconds_total	counter	Total user and system CPU time spent in seconds.
process_max_fds	gauge	Maximum number of open file descriptors.
process_open_fds	gauge	Number of open file descriptors.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
process_resident_memory_bytes	gauge	Resident memory size in bytes.
process_start_time_seconds	gauge	Start time of the process since Unix epoch in seconds.
process_virtual_memory_bytes	gauge	Virtual memory size in bytes.
process_virtual_memory_max_bytes	gauge	Maximum amount of virtual memory available in bytes.
rest_client_exec_plugin_certificate_rotation_age	histogram	[ALPHA] Histogram of the number of seconds the last auth exec plugin client certificate lived before being rotated. If auth exec plugin client certificates are unused, histogram will contain no data.
rest_client_exec_plugin_ttl_seconds	gauge	[ALPHA] Gauge of the shortest TTL of the client certificate or certificates managed by the auth exec plugin. The value is in seconds until certificate expiry (negative if already expired). If auth exec plugins are unused or manage no TLS certificates, the value will be +INF.
rest_client_request_duration_seconds	histogram	[ALPHA] Request latency in seconds. Broken down by verb and URL.
rest_client_requests_total	counter	[ALPHA] Number of HTTP requests, partitioned by status code, method, and host.
serviceaccount_legacy_tokens_total	counter	[ALPHA] Cumulative legacy service account tokens used.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
serviceaccount_stale_tokens_total	counter	[ALPHA] Cumulative stale projected service account tokens used.
serviceaccount_valid_tokens_total	counter	[ALPHA] Cumulative valid projected service account tokens used.
ssh_tunnel_open_count	counter	[ALPHA] Counter of ssh tunnel total open attempts.
ssh_tunnel_open_fail_count	counter	[ALPHA] Counter of ssh tunnel failed open attempts.
storage_operation_duration_seconds	histogram	[ALPHA] Storage operation duration.
storage_operation_errors_total	counter	[ALPHA] Storage operation errors.
storage_operation_status_count	counter	[ALPHA] Storage operation return statuses count.
volume_manager_total_volumes	gauge	[ALPHA] Number of volumes in Volume Manager.
workqueue_adds_total	counter	[ALPHA] Total number of adds handled by workqueue.
workqueue_depth	gauge	[ALPHA] Current depth of workqueue.
workqueue_longest_running_processor_seconds	gauge	[ALPHA] Number of seconds the longest-running processor for the workqueue has been running.
workqueue_queue_duration_seconds	histogram	[ALPHA] How long in seconds an item stays in the workqueue before being requested.

Table 20: Kubernetes Metric List *(Continued)*

Metric Name	Type	Description
`workqueue_retries_total`	counter	[ALPHA] Total number of retries handled by the workqueue.
`workqueue_unfinished_work_seconds`	gauge	[ALPHA] Number of seconds of work that have been done but haven't been observed by work_duration. Large values indicate stuck threads. You can deduce the number of stuck threads by observing the rate at which this value increases.
`workqueue_work_duration_seconds`	histogram	[ALPHA] How long in seconds it takes to process an item from the workqueue.

RELATED DOCUMENTATION

Contrail Networking Analytics | 263
Contrail Networking Metric List | 269
Cluster Node Metric List | 321
Contrail Networking Alert List | 339

Cluster Node Metric List

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List

Metric Name	Type	Description
`node_arp_entries`	gauge	ARP entries by device.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_authorizer_graph_actions_duration_seconds	histogram	[ALPHA] Histogram of duration of graph actions in node authorizer.
node_boot_time_seconds	gauge	Node boot time, in Unix time.
node_context_switches_total	counter	Total number of context switches.
node_cooling_device_cur_state	gauge	Current throttle state of the cooling device.
node_cooling_device_max_state	gauge	Maximum throttle state of the cooling device.
node_cpu_guest_seconds_total	counter	Seconds the CPUs spent in guests (VMs) for each mode.
node_cpu_seconds_total	counter	Seconds the CPUs spent in each mode.
node_disk_info	gauge	info of **/sys/block/<block_device>**.
node_disk_io_now	gauge	The number of I/Os currently in progress.
node_disk_io_time_seconds_total	counter	Total seconds spent doing I/Os.
node_disk_io_time_weighted_seconds_total	counter	The weighted number of seconds spent doing I/Os.
node_disk_read_bytes_total	counter	The total number of bytes read successfully.
node_disk_read_time_seconds_total	counter	The total number of seconds spent by all reads.
node_disk_reads_completed_total	counter	The total number of reads completed successfully.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_disk_reads_merged_total	counter	The total number of reads merged.
node_disk_write_time_seconds_total	counter	The total number of seconds spent by all writes.
node_disk_writes_completed_total	counter	The total number of writes completed successfully.
node_disk_writes_merged_total	counter	The number of writes merged.
node_disk_written_bytes_total	counter	The total number of bytes written successfully.
node_dmi_info	gauge	A metric with a constant '1' value labeled by bios_date, bios_release, bios_vendor, bios_version, board_asset_tag, board_name, board_serial, board_vendor, board_version, chassis_asset_tag, chassis_serial, chassis_vendor, chassis_version, product_family, product_name, product_serial, product_sku, product_uuid, product_version, and system_vendor if provided by DMI.
node_entropy_available_bits	gauge	Bits of available entropy.
node_entropy_pool_size_bits	gauge	Bits of entropy pool.
node_exporter_build_info	gauge	A metric with a constant '1' value labeled by version, revision, branch, and goversion from which node_exporter was built.
node_filefd_allocated	gauge	File descriptor statistics: allocated.
node_filefd_maximum	gauge	File descriptor statistics: maximum.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_filesystem_avail_bytes	gauge	Filesystem space available to non-root users in bytes.
node_filesystem_device_error	gauge	Whether an error occurred while getting statistics for the given device.
node_filesystem_files	gauge	Filesystem total file nodes.
node_filesystem_files_free	gauge	Filesystem total free file nodes.
node_filesystem_free_bytes	gauge	Filesystem free space in bytes.
node_filesystem_readonly	gauge	Filesystem read-only status.
node_filesystem_size_bytes	gauge	Filesystem size in bytes.
node_forks_total	counter	Total number of forks.
node_intr_total	counter	Total number of interrupts serviced.
node_ipvs_connections_total	counter	The total number of connections made.
node_ipvs_incoming_bytes_total	counter	The total amount of incoming data.
node_ipvs_incoming_packets_total	counter	The total number of incoming packets.
node_ipvs_outgoing_bytes_total	counter	The total amount of outgoing data.
node_ipvs_outgoing_packets_total	counter	The total number of outgoing packets.
node_load1	gauge	1m load average.
node_load15	gauge	15m load average.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_load5	gauge	5m load average.
node_memory_Active_anon_bytes	gauge	Memory information field Active_anon_bytes.
node_memory_Active_bytes	gauge	Memory information field Active_bytes.
node_memory_Active_file_bytes	gauge	Memory information field Active_file_bytes.
node_memory_AnonHugePages_bytes	gauge	Memory information field AnonHugePages_bytes.
node_memory_AnonPages_bytes	gauge	Memory information field AnonPages_bytes.
node_memory_Bounce_bytes	gauge	Memory information field Bounce_bytes.
node_memory_Buffers_bytes	gauge	Memory information field Buffers_bytes.
node_memory_Cached_bytes	gauge	Memory information field Cached_bytes.
node_memory_CmaFree_bytes	gauge	Memory information field CmaFree_bytes.
node_memory_CmaTotal_bytes	gauge	Memory information field CmaTotal_bytes.
node_memory_CommitLimit_bytes	gauge	Memory information field CommitLimit_bytes.
node_memory_Committed_AS_bytes	gauge	Memory information field Committed_AS_bytes.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_memory_DirectMap2M_bytes	gauge	Memory information field DirectMap2M_bytes.
node_memory_DirectMap4k_bytes	gauge	Memory information field DirectMap4k_bytes.
node_memory_Dirty_bytes	gauge	Memory information field Dirty_bytes.
node_memory_HardwareCorrupted_bytes	gauge	Memory information field HardwareCorrupted_bytes.
node_memory_HugePages_Free	gauge	Memory information field HugePages_Free.
node_memory_HugePages_Rsvd	gauge	Memory information field HugePages_Rsvd.
node_memory_HugePages_Surp	gauge	Memory information field HugePages_Surp.
node_memory_HugePages_Total	gauge	Memory information field HugePages_Total.
node_memory_Hugepagesize_bytes	gauge	Memory information field Hugepagesize_bytes.
node_memory_Inactive_anon_bytes	gauge	Memory information field Inactive_anon_bytes.
node_memory_Inactive_bytes	gauge	Memory information field Inactive_bytes.
node_memory_Inactive_file_bytes	gauge	Memory information field Inactive_file_bytes.
node_memory_KernelStack_bytes	gauge	Memory information field KernelStack_bytes.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_memory_Mapped_bytes	gauge	Memory information field Mapped_bytes.
node_memory_MemAvailable_bytes	gauge	Memory information field MemAvailable_bytes.
node_memory_MemFree_bytes	gauge	Memory information field MemFree_bytes.
node_memory_MemTotal_bytes	gauge	Memory information field MemTotal_bytes.
node_memory_Mlocked_bytes	gauge	Memory information field Mlocked_bytes.
node_memory_NFS_Unstable_bytes	gauge	Memory information field NFS_Unstable_bytes.
node_memory_PageTables_bytes	gauge	Memory information field PageTables_bytes.
node_memory_SReclaimable_bytes	gauge	Memory information field SReclaimable_bytes.
node_memory_SUnreclaim_bytes	gauge	Memory information field SUnreclaim_bytes.
node_memory_ShmemHugePages_bytes	gauge	Memory information field ShmemHugePages_bytes.
node_memory_ShmemPmdMapped_bytes	gauge	Memory information field ShmemPmdMapped_bytes.
node_memory_Shmem_bytes	gauge	Memory information field Shmem_bytes.
node_memory_Slab_bytes	gauge	Memory information field Slab_bytes.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_memory_SwapCached_bytes	gauge	Memory information field SwapCached_bytes.
node_memory_SwapFree_bytes	gauge	Memory information field SwapFree_bytes.
node_memory_SwapTotal_bytes	gauge	Memory information field SwapTotal_bytes.
node_memory_Unevictable_bytes	gauge	Memory information field Unevictable_bytes.
node_memory_VmallocChunk_bytes	gauge	Memory information field VmallocChunk_bytes.
node_memory_VmallocTotal_bytes	gauge	Memory information field VmallocTotal_bytes.
node_memory_VmallocUsed_bytes	gauge	Memory information field VmallocUsed_bytes.
node_memory_WritebackTmp_bytes	gauge	Memory information field WritebackTmp_bytes.
node_memory_Writeback_bytes	gauge	Memory information field Writeback_bytes.
node_netstat_Icmp6_InErrors	counter	Statistic Icmp6InErrors.
node_netstat_Icmp6_InMsgs	counter	Statistic Icmp6InMsgs.
node_netstat_Icmp6_OutMsgs	counter	Statistic Icmp6OutMsgs.
node_netstat_Icmp_InErrors	counter	Statistic IcmpInErrors.
node_netstat_Icmp_InMsgs	counter	Statistic IcmpInMsgs.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_netstat_Icmp_OutMsgs	counter	Statistic IcmpOutMsgs.
node_netstat_Ip6_InOctets	counter	Statistic Ip6InOctets.
node_netstat_Ip6_OutOctets	counter	Statistic Ip6OutOctets.
node_netstat_IpExt_InOctets	counter	Statistic IpExtInOctets.
node_netstat_IpExt_OutOctets	counter	Statistic IpExtOutOctets.
node_netstat_Ip_Forwarding	counter	Statistic IpForwarding.
node_netstat_TcpExt_ListenDrops	counter	Statistic TcpExtListenDrops.
node_netstat_TcpExt_ListenOverflows	counter	Statistic TcpExtListenOverflows.
node_netstat_TcpExt_SyncookiesFailed	counter	Statistic TcpExtSyncookiesFailed.
node_netstat_TcpExt_SyncookiesRecv	counter	Statistic TcpExtSyncookiesRecv.
node_netstat_TcpExt_SyncookiesSent	counter	Statistic TcpExtSyncookiesSent.
node_netstat_TcpExt_TCPSynRetrans	counter	Statistic TcpExtTCPSynRetrans.
node_netstat_TcpExt_TCPTimeouts	counter	Statistic TcpExtTCPTimeouts.
node_netstat_Tcp_ActiveOpens	counter	Statistic TcpActiveOpens.
node_netstat_Tcp_CurrEstab	counter	Statistic TcpCurrEstab.
node_netstat_Tcp_InErrs	counter	Statistic TcpInErrs.
node_netstat_Tcp_InSegs	counter	Statistic TcpInSegs.
node_netstat_Tcp_OutRsts	counter	Statistic TcpOutRsts.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_netstat_Tcp_OutSegs	counter	Statistic TcpOutSegs.
node_netstat_Tcp_PassiveOpens	counter	Statistic TcpPassiveOpens.
node_netstat_Tcp_RetransSegs	counter	Statistic TcpRetransSegs.
node_netstat_Udp6_InDatagrams	counter	Statistic Udp6InDatagrams.
node_netstat_Udp6_InErrors	counter	Statistic Udp6InErrors.
node_netstat_Udp6_NoPorts	counter	Statistic Udp6NoPorts.
node_netstat_Udp6_OutDatagrams	counter	Statistic Udp6OutDatagrams.
node_netstat_Udp6_RcvbufErrors	counter	Statistic Udp6RcvbufErrors.
node_netstat_Udp6_SndbufErrors	counter	Statistic Udp6SndbufErrors.
node_netstat_UdpLite6_InErrors	counter	Statistic UdpLite6InErrors.
node_netstat_UdpLite_InErrors	counter	Statistic UdpLiteInErrors.
node_netstat_Udp_InDatagrams	counter	Statistic UdpInDatagrams.
node_netstat_Udp_InErrors	counter	Statistic UdpInErrors.
node_netstat_Udp_NoPorts	counter	Statistic UdpNoPorts.
node_netstat_Udp_OutDatagrams	counter	Statistic UdpOutDatagrams.
node_netstat_Udp_RcvbufErrors	counter	Statistic UdpRcvbufErrors.
node_netstat_Udp_SndbufErrors	counter	Statistic UdpSndbufErrors.
node_network_address_assign_type	gauge	address_assign_type value of **/sys/class/net/**.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_network_carrier	gauge	carrier value of **/sys/class/net/**.
node_network_carrier_changes_total	counter	carrier_changes_total value of **/sys/class/net/**.
node_network_carrier_down_changes_total	counter	carrier_down_changes_total value of **/sys/class/net/**.
node_network_carrier_up_changes_total	counter	carrier_up_changes_total value of **/sys/class/net/**.
node_network_device_id	gauge	device_id value of **/sys/class/net/**.
node_network_dormant	gauge	dormant value of **/sys/class/net/**.
node_network_flags	gauge	flags value of **/sys/class/net/**.
node_network_iface_id	gauge	iface_id value of **/sys/class/net/**.
node_network_iface_link	gauge	iface_link value of **/sys/class/net/**.
node_network_iface_link_mode	gauge	iface_link_mode value of **/sys/class/net/**.
node_network_info	gauge	Non-numeric data from **/sys/class/net/**, value is always 1.
node_network_mtu_bytes	gauge	mtu_bytes value of **/sys/class/net/**.
node_network_name_assign_type	gauge	name_assign_type value of **/sys/class/net/**.
node_network_net_dev_group	gauge	net_dev_group value of **/sys/class/net/**.
node_network_protocol_type	gauge	protocol_type value of **/sys/class/net/**.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_network_receive_bytes_total	counter	Network device statistic receive_bytes.
node_network_receive_compressed_total	counter	Network device statistic receive_compressed.
node_network_receive_drop_total	counter	Network device statistic receive_drop.
node_network_receive_errs_total	counter	Network device statistic receive_errs.
node_network_receive_fifo_total	counter	Network device statistic receive_fifo.
node_network_receive_frame_total	counter	Network device statistic receive_frame.
node_network_receive_multicast_total	counter	Network device statistic receive_multicast.
node_network_receive_packets_total	counter	Network device statistic receive_packets.
node_network_speed_bytes	gauge	speed_bytes value of **/sys/class/net/**.
node_network_transmit_bytes_total	counter	Network device statistic transmit_bytes.
node_network_transmit_carrier_total	counter	Network device statistic transmit_carrier.
node_network_transmit_colls_total	counter	Network device statistic transmit_colls.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_network_transmit_compressed_total	counter	Network device statistic transmit_compressed.
node_network_transmit_drop_total	counter	Network device statistic transmit_drop.
node_network_transmit_errs_total	counter	Network device statistic transmit_errs.
node_network_transmit_fifo_total	counter	Network device statistic transmit_fifo.
node_network_transmit_packets_total	counter	Network device statistic transmit_packets.
node_network_transmit_queue_length	gauge	transmit_queue_length value of **/sys/class/net/**.
node_network_up	gauge	Value is 1 if operstate is 'up', 0 otherwise.
node_nf_conntrack_entries	gauge	Number of currently allocated flow entries for connection tracking.
node_nf_conntrack_entries_limit	gauge	Maximum size of connection tracking table.
node_os_info	gauge	A metric with a constant '1' value labeled by build_id, id, id_like, image_id, image_version, name, pretty_name, variant, variant_id, version, version_codename, version_id.
node_os_version	gauge	Metric containing the major.minor part of the OS version.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_power_supply_info	gauge	info of **/sys/class/power_supply/ <power_supply>**.
node_power_supply_online	gauge	online value of **/sys/class/ power_supply/<power_supply>**.
node_procs_blocked	gauge	Number of processes blocked waiting for I/O to be completed.
node_procs_running	gauge	Number of processes in runnable state.
node_schedstat_running_seconds_total	counter	Number of seconds CPU spent running a process.
node_schedstat_timeslices_total	counter	Number of timeslices executed by CPU.
node_schedstat_waiting_seconds_total	counter	Number of seconds spent by processing waiting for this CPU.
node_scrape_collector_duration_seconds	gauge	node_exporter: Duration of a collector scrape.
node_scrape_collector_success	gauge	node_exporter: Whether a collector succeeded.
node_sockstat_FRAG6_inuse	gauge	Number of FRAG6 sockets in state inuse.
node_sockstat_FRAG6_memory	gauge	Number of FRAG6 sockets in state memory.
node_sockstat_FRAG_inuse	gauge	Number of FRAG sockets in state inuse.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List ***(Continued)***

Metric Name	Type	Description
node_sockstat_FRAG_memory	gauge	Number of FRAG sockets in state memory.
node_sockstat_RAW6_inuse	gauge	Number of RAW6 sockets in state inuse.
node_sockstat_RAW_inuse	gauge	Number of RAW sockets in state inuse.
node_sockstat_TCP6_inuse	gauge	Number of TCP6 sockets in state inuse.
node_sockstat_TCP_alloc	gauge	Number of TCP sockets in state alloc.
node_sockstat_TCP_inuse	gauge	Number of TCP sockets in state inuse.
node_sockstat_TCP_mem	gauge	Number of TCP sockets in state mem.
node_sockstat_TCP_mem_bytes	gauge	Number of TCP sockets in state mem_bytes.
node_sockstat_TCP_orphan	gauge	Number of TCP sockets in state orphan.
node_sockstat_TCP_tw	gauge	Number of TCP sockets in state tw.
node_sockstat_UDP6_inuse	gauge	Number of UDP6 sockets in state inuse.
node_sockstat_UDPLITE6_inuse	gauge	Number of UDPLITE6 sockets in state inuse.
node_sockstat_UDPLITE_inuse	gauge	Number of UDPLITE sockets in state inuse.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_sockstat_UDP_inuse	gauge	Number of UDP sockets in state inuse.
node_sockstat_UDP_mem	gauge	Number of UDP sockets in state mem.
node_sockstat_UDP_mem_bytes	gauge	Number of UDP sockets in state mem_bytes.
node_sockstat_sockets_used	gauge	Number of IPv4 sockets in use.
node_softnet_dropped_total	counter	Number of dropped packets.
node_softnet_processed_total	counter	Number of processed packets
node_softnet_times_squeezed_total	counter	Number of times processing packets ran out of quota.
node_textfile_scrape_error	gauge	1 if there was an error opening or reading a file, otherwise, 0.
node_time_clocksource_available_info	gauge	Available clocksources read from **/sys/devices/system/clocksource**.
node_time_clocksource_current_info	gauge	Current clocksource read from **/sys/devices/system/clocksource**.
node_time_seconds	gauge	System time in seconds since epoch (1970).
node_time_zone_offset_seconds	gauge	System time zone offset in seconds.
node_timex_estimated_error_seconds	gauge	Estimated error in seconds.
node_timex_frequency_adjustment_ratio	gauge	Local clock frequency adjustment.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_timex_loop_time_constant	gauge	Phase-locked loop time constant.
node_timex_maxerror_seconds	gauge	Maximum error in seconds.
node_timex_offset_seconds	gauge	Time offset in between local system and reference clock.
node_timex_pps_calibration_total	counter	Pulse-per-second count of calibration intervals.
node_timex_pps_error_total	counter	Pulse-per-second of calibration errors.
node_timex_pps_frequency_hertz	gauge	Pulse-per-second frequency.
node_timex_pps_jitter_seconds	gauge	Pulse-per-second jitter.
node_timex_pps_jitter_total	counter	Pulse-per-second count of jitter-limit-exceeded events.
node_timex_pps_shift_seconds	gauge	Pulse-per-second interval duration.
node_timex_pps_stability_exceeded_total	counter	Pulse-per-second count of stability limit exceeded events.
node_timex_pps_stability_hertz	gauge	Pulse-per-second stability, average of recent frequency changes.
node_timex_status	gauge	Value of the status array bits.
node_timex_sync_status	gauge	Is clock synchronized to a reliable server (1 = yes, 0 = no)
node_timex_tai_offset_seconds	gauge	International Atomic Time (TAI) offset.
node_timex_tick_seconds	gauge	Seconds between clock ticks.

Table 21: Cloud-Native Contrail Networking (CN2) Cluster Node Metric List *(Continued)*

Metric Name	Type	Description
node_udp_queues	gauge	Number of allocated memory in the kernel for UDP datagrams in bytes.
node_uname_info	gauge	Labeled system information as provided by the uname system call.
node_vmstat_oom_kill	counter	**/proc/vmstat** information field oom_kill.
node_vmstat_pgfault	counter	**/proc/vmstat** information field pgfault.
node_vmstat_pgmajfault	counter	**/proc/vmstat** information field pgmajfault.
node_vmstat_pgpgin	counter	**/proc/vmstat** information field pgpgin.
node_vmstat_pgpgout	counter	**/proc/vmstat** information field pgpgout.
node_vmstat_pswpin	counter	**/proc/vmstat** information field pswpin.
node_vmstat_pswpout	counter	**/proc/vmstat** information field pswpout.

RELATED DOCUMENTATION

Contrail Networking Analytics | 263
Contrail Networking Metric List | 269
Kubernetes Metric List | 283
Contrail Networking Alert List | 339

Contrail Networking Alert List

Table 22: Cloud-Native Contrail Networking (CN2) Alert List

Alert Name	Severity	Description
VRouterConnectionDown	major	VRouter <name> <connection_type> connection to <connection_id> is down.
VRouterNonFunctional	major	VRouter <name> is non-functional.
ControllerNonFunctional	major	Controller <name> is non-functional.
ControllerConnectionDown	major	Controller <name> <connection_type> connection to <connection_id> is down.
ControllerDBConnectionDown	major	Controller <name> connection to database is down.
AlertmanagerFailedReload	critical	Reloading an Alertmanager configuration has failed.
AlertmanagerMembersInconsistent	critical	A member of an Alertmanager cluster has not found all other cluster members.
AlertmanagerFailedToSendAlerts	warning	An Alertmanager instance failed to send notifications.
AlertmanagerClusterFailedToSendAlerts	critical	All Alertmanager instances in a cluster failed to send notifications to a critical integration.
AlertmanagerClusterFailedToSendAlerts	warning	All Alertmanager instances in a cluster failed to send notifications to a non-critical integration.

Table 22: Cloud-Native Contrail Networking (CN2) Alert List *(Continued)*

Alert Name	Severity	Description
`AlertmanagerConfigInconsistent`	critical	Alertmanager instances within the same cluster have different configurations.
`AlertmanagerClusterDown`	critical	Half or more of the Alertmanager instances within the same cluster are down.
`AlertmanagerClusterCrashlooping`	critical	Half or more of the Alertmanager instances within the same cluster are crashlooping.
`ConfigReloaderSidecarErrors`	warning	`config-reloader` sidecar has not had a successful reload for 10m.
`etcdInsufficientMembers`	critical	`etcd` cluster "`<name>`": insufficient members (`<value>`).
`etcdNoLeader`	critical	`etcd` cluster "`<name>`": member `<instance>` has no leader.
`etcdHighNumberOfLeaderChanges`	warning	`etcd` cluster "`<name>`": instance `<instance>` has seen `<value>` leader changes within the last hour.
`etcdHighNumberOfFailedGRPCRequests`	warning	`etcd` cluster "`<name>`": `<value>`% of requests for `<grpc_method>` failed on etcd instance `<instance>`.
`etcdHighNumberOfFailedGRPCRequests`	critical	`etcd` cluster "`<name>`": `<value>`% of requests for `<grpc_method>` failed on etcd instance `<instance>`.
`etcdGRPCRequestsSlow`	critical	`etcd` cluster "`<name>`": gRPC requests to `<grpc_method>` are taking `<value>`s on etcd instance `<instance>`.

Table 22: Cloud-Native Contrail Networking (CN2) Alert List *(Continued)*

Alert Name	Severity	Description
etcdMemberCommunicationSlow	warning	etcd cluster "<name>": member communication with <name> is taking <value>s on etcd instance <instance>.
etcdHighNumberOfFailedProposals	warning	etcd cluster "<name>": <value> proposal failures within the last hour on etcd instance <instance>.
etcdHighFsyncDurations	warning	etcd cluster "<name>": 99th percentile fsync durations are <value>s on etcd instance <instance>.
etcdHighCommitDurations	warning	etcd cluster "<name>": 99th percentile commit durations <value>s on etcd instance <instance>.
etcdHighNumberOfFailedHTTPRequests	warning	<value>% of requests for <method> failed on etcd instance <instance>.
etcdHighNumberOfFailedHTTPRequests	critical	<value>% of requests for <method> failed on etcd instance <instance>.
etcdHTTPRequestsSlow	warning	etcd instance <instance> HTTP requests to <method> are slow.
TargetDown	warning	One or more targets are unreachable.
KubeAPIErrorBudgetBurn	critical	The API server is burning too much error budget.
KubeAPIErrorBudgetBurn	warning	The API server is burning too much error budget.

Table 22: Cloud-Native Contrail Networking (CN2) Alert List *(Continued)*

Alert Name	Severity	Description
KubeStateMetricsListErrors	critical	kube-state-metrics is experiencing errors in list operations.
KubeStateMetricsWatchErrors	critical	kube-state-metrics is experiencing errors in watch operations.
KubeStateMetricsShardingMismatch	critical	kube-state-metrics sharding is misconfigured.
KubeStateMetricsShardsMissing	critical	kube-state-metrics shards are missing.
KubePodCrashLooping	warning	Pod is crash looping.
KubePodNotReady	warning	Pod has been in a non-ready state for more than 15 minutes.
KubeDeploymentGenerationMismatch	warning	Deployment generation mismatch due to possible roll-back.
KubeDeploymentReplicasMismatch	warning	Deployment has not matched the expected number of replicas.
KubeStatefulSetReplicasMismatch	warning	Deployment has not matched the expected number of replicas.
KubeStatefulSetGenerationMismatch	warning	StatefulSet generation mismatch due to possible roll-back.
KubeStatefulSetUpdateNotRolledOut	warning	StatefulSet update has not been rolled out.
KubeDaemonSetRolloutStuck	warning	DaemonSet rollout is stuck.
KubeContainerWaiting	warning	Pod container waiting longer than 1 hour.
KubeDaemonSetNotScheduled	warning	DaemonSet pods are not scheduled.

Table 22: Cloud-Native Contrail Networking (CN2) Alert List *(Continued)*

Alert Name	Severity	Description
`KubeDaemonSetMisScheduled`	warning	`DaemonSet` pods are misscheduled.
`KubeJobCompletion`	warning	Job was not complete in time.
`KubeJobFailed`	warning	Job failed (was not completed).
`KubeHpaReplicasMismatch`	warning	HPA has not matched desired number of replicas.
`KubeHpaMaxedOut`	warning	HPA is running at max replicas.
`KubeCPUOvercommit`	warning	Cluster has overcommitted CPU resource requests.
`KubeMemoryOvercommit`	warning	Cluster has overcommitted CPU resource requests.
`KubeCPUQuotaOvercommit`	warning	Cluster has overcommitted CPU resource requests.
`KubeMemoryQuotaOvercommit`	warning	Cluster has overcommitted memory resource requests.
`KubeQuotaAlmostFull`	info	Namespace quota is going to be full.
`KubeQuotaFullyUsed`	info	Namespace quota is fully used.
`KubeQuotaExceeded`	warning	Namespace quota has exceeded the limits.
`CPUThrottlingHigh`	info	Processes experience elevated CPU throttling.
`KubePersistentVolumeFillingUp`	critical	`PersistentVolume` is filling up.
`KubePersistentVolumeFillingUp`	warning	`PersistentVolume` is filling up.

Table 22: Cloud-Native Contrail Networking (CN2) Alert List *(Continued)*

Alert Name	Severity	Description
`KubePersistentVolumeErrors`	critical	`PersistentVolume` is having issues with provisioning.
`KubeVersionMismatch`	warning	Different semantic versions of Kubernetes components are running.
`KubeClientErrors`	warning	Kubernetes API server client is experiencing errors.
`KubeClientCertificateExpiration`	warning	Client certificate is about to expire.
`KubeClientCertificateExpiration`	critical	Client certificate is about to expire.
`KubeAggregatedAPIErrors`	warning	Kubernetes aggregated API has reported errors.
`KubeAggregatedAPIDown`	warning	Kubernetes aggregated API is down.
`KubeAPIDown`	critical	Target disappeared from Prometheus target discovery.
`KubeAPITerminatedRequests`	warning	The Kubernetes `apiserver` has terminated `<value>` of its incoming requests.
`KubeControllerManagerDown`	critical	Target disappeared from Prometheus target discovery.
`KubeProxyDown`	critical	Target disappeared from Prometheus target discovery.
`KubeNodeNotReady`	warning	Node is not ready.
`KubeNodeUnreachable`	warning	Node is unreachable.
`KubeletTooManyPods`	info	Kubelet is running at capacity.

Table 22: Cloud-Native Contrail Networking (CN2) Alert List *(Continued)*

Alert Name	Severity	Description
`KubeNodeReadinessFlapping`	warning	Node readiness status is flapping.
`KubeletPlegDurationHigh`	warning	Kubelet Pod Lifecycle Event Generator is taking too long to relist.
`KubeletPodStartUpLatencyHigh`	warning	Kubelet Pod startup latency is too high.
`KubeletClientCertificateExpiration`	warning	Kubelet client certificate is about to expire.
`KubeletClientCertificateExpiration`	critical	Kubelet client certificate is about to expire.
`KubeletServerCertificateExpiration`	warning	Kubelet server certificate is about to expire.
`KubeletServerCertificateExpiration`	critical	Kubelet server certificate is about to expire.
`KubeletClientCertificateRenewalErrors`	warning	Kubelet has failed to renew its client certificate.
`KubeletServerCertificateRenewalErrors`	warning	Kubelet has failed to renew its server certificate.
`KubeletDown`	critical	Target disappeared from Prometheus target discovery.
`KubeSchedulerDown`	critical	Target disappeared from Prometheus target discovery.
`NodeFilesystemSpaceFillingUp`	warning	Filesystem is predicted to run out of space within the next 24 hours.
`NodeFilesystemSpaceFillingUp`	critical	Filesystem is predicted to run out of space within the next 4 hours.

Table 22: Cloud-Native Contrail Networking (CN2) Alert List *(Continued)*

Alert Name	Severity	Description
NodeFilesystemAlmostOutOfSpace	warning	Filesystem has less than 5% space left.
NodeFilesystemAlmostOutOfSpace	critical	Filesystem has less than 3% space left.
NodeFilesystemFilesFillingUp	warning	Filesystem is predicted to run out of inodes within the next 24 hours.
NodeFilesystemFilesFillingUp	critical	Filesystem is predicted to run out of inodes within the next 4 hours.
NodeFilesystemAlmostOutOfFiles	warning	Filesystem has less than 5% inodes left.
NodeFilesystemAlmostOutOfFiles	critical	Filesystem has less than 3% inodes left.
NodeNetworkReceiveErrs	warning	Network interface is reporting many receive errors.
NodeNetworkTransmitErrs	warning	Network interface is reporting many transmit errors.
NodeHighNumberConntrackEntriesUsed	warning	Number of conntrack are getting close to the limit.
NodeTextFileCollectorScrapeError	warning	Node Exporter text file collector failed to scrape.
NodeClockSkewDetected	warning	Clock skew detected.
NodeClockNotSynchronising	warning	Clock not synchronizing.
NodeRAIDDegraded	critical	RAID array is degraded.
NodeRAIDDiskFailure	warning	Failed device in RAID array.

Table 22: Cloud-Native Contrail Networking (CN2) Alert List *(Continued)*

Alert Name	Severity	Description
NodeFileDescriptorLimit	warning	Kernel is predicted to exhaust file descriptors limit soon.
NodeFileDescriptorLimit	critical	Kernel is predicted to exhaust file descriptors limit soon.
NodeNetworkInterfaceFlapping	warning	Network interface is often changing its status.
PrometheusBadConfig	critical	Failed Prometheus configuration reload.
PrometheusNotificationQueueRunningFull	warning	Prometheus alert notification queue predicted to run full in less than 30m.
PrometheusErrorSendingAlertsToSomeAlertmanagers	warning	Prometheus has encountered more than 1% errors sending alerts to a specific Alertmanager.
PrometheusNotConnectedToAlertmanagers	warning	Prometheus is not connected to any Alertmanagers.
PrometheusTSDBReloadsFailing	warning	Prometheus has issues reloading blocks from disk.
PrometheusTSDBCompactionsFailing	warning	Prometheus has issues compacting blocks.
PrometheusNotIngestingSamples	warning	Prometheus is not ingesting samples.
PrometheusDuplicateTimestamps	warning	Prometheus is dropping samples with duplicate timestamps.
PrometheusOutOfOrderTimestamps	warning	Prometheus drops samples with out-of-order timestamps.

Table 22: Cloud-Native Contrail Networking (CN2) Alert List *(Continued)*

Alert Name	Severity	Description
`PrometheusRemoteStorageFailures`	critical	Prometheus fails to send samples to remote storage.
`PrometheusRemoteWriteBehind`	critical	Prometheus remote write is behind.
`PrometheusRemoteWriteDesiredShards`	warning	Prometheus remote write desired shards calculation wants to run more than configured max shards.
`PrometheusRuleFailures`	critical	Prometheus is failing rule evaluations.
`PrometheusMissingRuleEvaluations`	warning	Prometheus is missing rule evaluations due to slow rule group evaluation.
`PrometheusTargetLimitHit`	warning	Prometheus has dropped targets because some scrape configs have exceeded the target limit.
`PrometheusLabelLimitHit`	warning	Prometheus has dropped targets because some scrape configs have exceeded the label limit.
`PrometheusTargetSyncFailure`	critical	Prometheus has failed to sync targets.
`PrometheusErrorSendingAlertsToAnyAlertmanager`	critical	Prometheus encounters more than 3% errors sending alerts to any Alertmanager.
`PrometheusOperatorListErrors`	warning	Errors while performing list operations in controller.
`PrometheusOperatorWatchErrors`	warning	Errors while performing list operations in controller.
`PrometheusOperatorSyncFailed`	warning	Last controller reconciliation failed.

Table 22: Cloud-Native Contrail Networking (CN2) Alert List *(Continued)*

Alert Name	Severity	Description
`PrometheusOperatorReconcileErrors`	warning	Errors while reconciling controller.
`PrometheusOperatorNodeLookupErrors`	warning	Errors while reconciling Prometheus.
`PrometheusOperatorNotReady`	warning	Prometheus operator not ready.
`PrometheusOperatorRejectedResources`	warning	Resources rejected by Prometheus operator.

RELATED DOCUMENTATION

Contrail Networking Analytics | 263
Contrail Networking Metric List | 269
Kubernetes Metric List | 283
Cluster Node Metric List | 321

vRouter Session Analytics in Contrail Networking

IN THIS SECTION

- Collector Module | 350
- Collector Deployment | 350
- Data Collection | 350
- Configure Data Collection | 353
- Collector Query | 353
- Run a Query | 353

Juniper® Networks supports the collection, storage, and query for vRouter traffic in environments using Cloud-Native Contrail Networking (CN2) Release 22.1 or later in a Kubernetes-orchestrated environment.

Collector Module

CN2 collects user visible entities (UVEs) and traffic information (session) for traffic analysis and troubleshooting. The `collector` module stores these objects and provides APIs to access the collected information.

The CN2 vRouter agent exports data records to the collector when events are created or deleted.

Collector Deployment

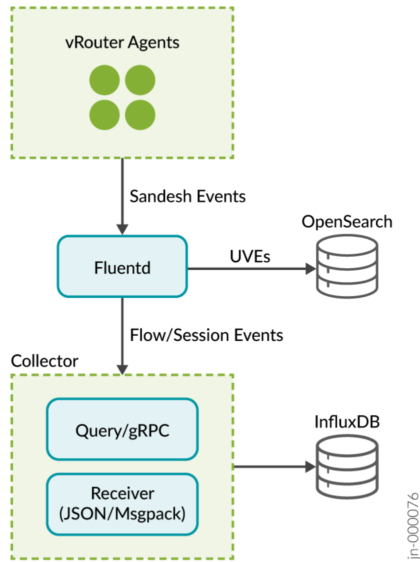
The following components are installed in the Contrail cluster in the `contrail` namespace (NS):

- Collector Microservice—Collects incoming events.
- InfluxDB—A time series database built specifically for storing time series data. Works with Grafana as a visualization tool for time series data.
- Fluentd—Logging agent that performs log collection, parsing, and distribution to other services such asOpenSearch.
- OpenSearch—OpenSearch is the search and analytics engine in the AWS OpenSearch Stack, providing real-time search and analytics for all types of data.
- OpenSearch Dashboards—User interface that lets you visualize your OpenSearch data and navigate the OpenSearch Stack.

Data Collection

Figure 15 on page 351 shows the data collection.

Figure 15: Cloud-Native Contrail Collector: Event and Log Ingestion



UVEs

UVEs are stored in OpenSearch in an index named by the name of the UVE.

Session

Session records are stored in InfluxDB. These records are pushed as events from all agents. This data is downsampled for longer duration. Retention periods of live, downsampled table, and downsampling windows are configurable using the configuration.

Table 23: Session Records Information

Column	Filterable	Detail
`vn`	Yes	Client Virtual Network
`vmi`	Yes	Interface
`remote_vn`	Yes	Server Virtual Network
`vrouter_ip`	Yes	Agent IP
`local_ip`	Yes	Client IP

Table 23: Session Records Information *(Continued)*

Column	Filterable	Detail
client_port	Yes	Client Port
remote_ip	Yes	Server IP
server_port	Yes	Server Port
protocol	Yes	Protocol
label.local.<label-name>	Yes	Client Pod Labels (for example, client pod with label site maps to label.local.site tag in database).
label.remote.<label-name>	Yes	Server Pod Labels
forward_sampled_bytes	No	Bytes Sent
forward_sampled_pkts	No	Packets Sent
reverse_sampled_bytes	No	Bytes Received
reverse_sampled_pkts	No	Packets Received
total_bytes	No	Total Bytes Exchanged

Configure Data Collection

To configure vRouter agents to send `SessionEndpoint` messages to the `fluentd` service, run the following three commands. Replace `<cluster-ip>` with the cluster IP address of the `fluentd` service in the contrail-analytics namespace.

```
kubectl -n contrail patch vrouter contrail-vrouter-masters --type=merge -p '{"spec":{"agent":
{"default":{"collectors":["<cluster-ip>:24224"]}}}}'
```

```
kubectl -n contrail patch vrouter contrail-vrouter-nodes --type=merge -p '{"spec":{"agent":
{"default":{"collectors":["<cluster-ip>:24224"]}}}}'
```

```
kubectl -n contrail patch gvc default-global-vrouter-config --type=merge -p '{"spec":
{"flowExportRate": 10000}}'
```

After running the three configuration commands, restart vRouter for the configuration to take effect. To restart vRouter, run the following command:

```
kubectl -n contrail delete $(kubectl get pods -l 'app in (contrail-vrouter-masters, contrail-
vrouter-nodes)' -n contrail -o name)
```

Collector Query

The `collector` modules provide a query interface for access.

Run a Query

Example Query

The following query gets total bytes exchanged between unique source-destination pairs (by labels) in the `contrail-analytics` namespace:

```
{
  "granularity": 3600,
  "column": [
    {
      "name": "total_bytes",
      "aggregation": "sum"
    },
    {
      "name": "/^label.*/",
      "regex": true
    }
  ],
  "skip_columns": [
    "label.remote.pod-template-hash",
    "label.local.pod-template-hash"
  ],
  "range":{
    "start_time": -3600
  },
  "filter": [
    {
      "field": "label.local.namespace",
      "operator": "==",
      "value": "contrail-analytics"
    },
    {
      "field": "label.remote.namespace",
      "operator": "==",
      "value": "contrail-analytics"
    }
  ]
}
```

Example Query Response

```
{
  "status": "success",
  "total": 5,
```

```
  "data": {
    "resultType": "matrix",
    "result": [
      {
        "metric": {
          "label.local.namespace": "contrail-analytics",
          "label.remote.app": "collector",
          "label.remote.namespace": "contrail-analytics"
        },
        "fields": [
          "_time",
          "total_bytes"
        ],
        "values": [
          [
            1645768800,
            31012095
          ]
        ]
      },
      {
        "metric": {
          "label.local.namespace": "contrail-analytics",
          "label.remote.app": "opensearch",
          "label.remote.chart": "opensearch",
          "label.remote.controller-revision-hash": "opensearch-7fcc8df678",
          "label.remote.namespace": "contrail-analytics",
          "label.remote.release": "contrail-analytics"
        },
        "fields": [
          "_time",
          "total_bytes"
        ],
        "values": [
          [
            1645768800,
            221493
          ]
        ]
      },
      {
        "metric": {
          "label.local.controller-revision-hash": "5599999fc7",
```

```
        "label.local.namespace": "contrail-analytics",
        "label.local.pod-template-generation": "1",
        "label.remote.namespace": "contrail-analytics"
      },
      "fields": [
        "_time",
        "total_bytes"
      ],
      "values": [
        [
          1645768800,
          23349247
        ]
      ]
    },
    {
      "metric": {
        "label.local.app": "collector",
        "label.local.namespace": "contrail-analytics",
        "label.remote.controller-revision-hash": "influxdb-7bdd86f8c",
        "label.remote.namespace": "contrail-analytics"
      },
      "fields": [
        "_time",
        "total_bytes"
      ],
      "values": [
        [
          1645768800,
          10412552
        ]
      ]
    },
    {
      "metric": {
        "label.local.app": "opensearch-dashboards",
        "label.local.namespace": "contrail-analytics",
        "label.local.release": "contrail-analytics",
        "label.remote.app": "opensearch",
        "label.remote.chart": "opensearch",
        "label.remote.controller-revision-hash": "opensearch-7fcc8df678",
        "label.remote.namespace": "contrail-analytics",
        "label.remote.release": "contrail-analytics"
```

```
          },
          "fields": [
            "_time",
            "total_bytes"
          ],
          "values": [
            [
              1645768800,
              25152
            ]
          ]
        }
      ]
    }
  }
```

RELATED DOCUMENTATION

Contrail Networking Analytics | 263
Centralized Logging | 357

Centralized Logging

IN THIS SECTION

- Benefits of Centralized Logging | 358
- Overview: Centralized Logging | 358
- Logs, Events, and Flows with Fluentd | 359

Juniper® Networks supports centralized logging using Cloud-Native Contrail Networking™ (CN2) Release 22.1 or later in a Kubernetes-orchestrated environment.

Benefits of Centralized Logging

- The centralization of all platform logs eases troubleshooting. Centralization allows you (the administrator) to take a holistic view of events or outages extending across the many components within the deployment.
- You have one portal, allowing you to monitor, view, filter, and search for events across all platform components from a single view.

Overview: Centralized Logging

Instead of browsing through individual log files, collected logs from all components of CN2 are available to you in a centralized location. The centralized location enables you to correlate the log files from multiple software components. For security, strict logging exists for all create, read, update, and delete (CRUD) actions. You perform these actions with individual access credentials so that you can identify individuals.

AWS OpenSearch Stack, an open source log collector and analyzer framework, provides out-of-box log collection and analysis functionality. The OpenSearch stack allows a single portal for analyzing logs from CN2. OpenSearch stack also analyzes logs from other software components and platforms deployed in the cluster. Examples include Linux OS logs, Kubernetes logs, and software components such as virtualized network functions (VNFs) and container network functions (CNFs).

OpenSearch Stack includes:

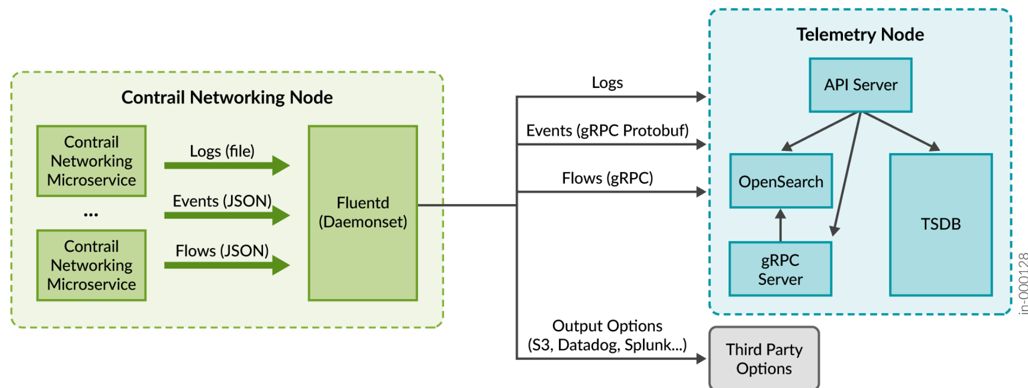
- OpenSearch—Real-time and scalable search engine that allows for full-text and structured search as well as analytics. This search engine indexes and searches through large volumes of log data.
- OpenSearch Dashboard—Allows you to explore your OpenSearch log data through a web interface and enables you to build dashboards and queries.
- Fluentd—Logging agent that performs log collection, parsing, and distribution to other services such as OpenSearch.
- Fluent Bit—Log processor and forwarder that collects data, such as metrics and logs, from different sources. High throughput with low CPU and memory usage. Fluent Bit is installed in every workload cluster.

The logging components are included and deployed in the optional telemetry node deployment. Installation commands are integrated in the telemetry installation.

Logs, Events, and Flows with Fluentd

Fluentd collects logs, events, and flows running on each CN2 node. Fluentd is the logging agent that performs log collection, parsing, and distribution to other services such as OpenSearch.

Figure 16: Logs, Events, and Flows with Fluentd



Logs

Logs are collected from log files or `stdout/stderr` data streams and directed to the OpenSearch library stack with cluster quorum. Each CN2 node (configuration, control, compute, Web UI, and telemetry node) runs Fluent Bit or Fluentd to collect logs. The logs are sent to multiple configured sinks, such as OpenSearch. Fluentd supports multiple output options to send collected logs.

- Control and compute nodes generate unstructured and structured logs through the Sandesh library. The Contrail Networking Sandesh library generates structured JSON files.
- Configuration, Web UI, and telemetry node components produce standard logs to files or to `stdout/stderr`, which are then sent to Fluentd or Fluent Bit.

Multiple Kubernetes clusters in any CN2 cluster or in multiple CN2 clusters can connect with a Fluentd/OpenSearch monitoring component.

Events

The vRouter agent and control node produce events through Sandesh. The Sandesh library produces JSON structured data and sends those files to the configured options. Configured options are `stdout`, file, or TCP port (Fluentd). Fluentd is configured with multiple output options to send data either to OpenSearch or to the telemetry node's gRPC server. The telemetry node keeps cache for the latest status events.

Flows

The vRouter agent produces flow data at regular configured intervals. Configuration options for flow data generation supported by the vRouter agent are syslogs, JSON structure, and the default Sandesh.

RELATED DOCUMENTATION

Contrail Networking Analytics | 263
vRouter Session Analytics in Contrail Networking | 349

Port-Based Mirroring

SUMMARY

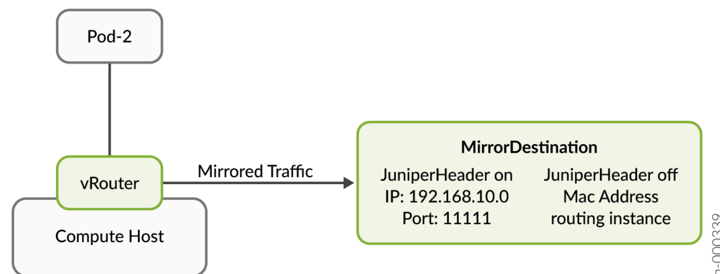
This section describes port-based mirroring in Juniper® Cloud-Native Contrail Networking (CN2) Release 22.2 and later in a Kubernetes-orchestrated environment.

IN THIS SECTION

- Overview: Port-Based Mirroring | 360
- Example: Configure Port-Based Mirroring | 361
- Summary | 364

Overview: Port-Based Mirroring

Figure 17: CN2 Port-Based Mirror Topology



Port mirroring sends network traffic from defined ports to a network analyzer where you can monitor and analyze the data. In CN2, the following is supported:

- Mirroring configuration is primarily driven from the pod configuration for both the receiver and interface being mirrored. You don't need to configure the virtual machine interface (VMI) directly.
- Mirroring configuration involves creating a `MirrorDestination` resource and associating the `MirrorDestination` resource to the pod interface to be mirrored.
- `MirrorDestination` identifies the mirrored traffic receiver pod and interface. When `juniperHeader` is enabled, the receiver pod IP address and port are used. When `juniperHeader` is disabled, the receiver pod MAC address `routingInstance` is used to forward mirrored traffic.
- A `MirrorDestination` can be associated with multiple VMIs to be mirrored.
- A `MirrorDestination` resource defines the mirrored traffic receiver such as IP address, port used for receiving mirrored traffic, Juniper header configuration, dynamic or static next-hop, and so on.
- A pod interface to be mirrored can be configured when creating the pod or by editing the pod.

Example: Configure Port-Based Mirroring

The following procedure is an example configuration that creates a `MirrorDestination` resource and specifies the `MirrorDestination` resource name, such as `mirrordestinationprofile1`, on the interface to be mirrored.

1. Use the `MirrorDestination` YAML file to create a `MirrorDestination` resource by adding multiple destination pods with the label `core.juniper.net/analyzer-pod-selector: analyzerpod`.

   The `MirrorDestination` resource uses the label `core.juniper.net/analyzer-pod-selector: analyzerpod` to calculate and determine the mirrored traffic pod receiver.

   Example `MirrorDestination` YAML file:

```
apiVersion: core.contrail.juniper.net/v1alpha1
kind: MirrorDestination
metadata:
  name: mirrordestinationprofile1
  labels:
    core.juniper.net/analyzer-pod-selector: analyzerpod
spec:
  trafficDirection: <ingress|egress|both>
  juniperHeader: <boolean>
```

```
  udpPort: <integer>
  staticNhHeader: <null for dynamic nh|vtep tunnel destip, mac, vxlanid for static nh>
  nextHopMode: <static|dynamic>
  nicAssistedMirroring: <boolean>
    nicAssistedVlanID:
  staticNextHopHeader:
    vTEPDestinationIP:
    vTEPDestinationMac:
    vxlanID:
```

When you deploy the YAML file, multiple pods could match the label `analyzerpod`. The first matching pod is selected as the mirrored traffic receiver. The selected pod remains sticky until the pod or interface is no longer available.

Following is the analyzer pod YAML file with label `analyzerpod`, indicating that `MirrorDestination` can use this pod.

- Note the label value for `core.juniper.net/analyzer-podanalyzerpod` is the same as specified in the `MirrorDestination` YAML file.

- The `MirrorDestination` controller uses this label to calculate the `analyzer_ip`, `macaddress`, and `routinginstance`.

- The pod interface to be used is specified in the annotation below:

  `core.juniper.net/analyzer-interface: true`

  You can specify the default pod interface directly under annotations. For a custom VN interface, you specify it in the `cni-args` of the network. The example `Pod/analyzerpod` YAML file shows both examples.

- `core.juniper.net/analyzer-interface: true` indicates that the `vn-1` pod interface will receive mirrored traffic.

Example `Pod/analyzerpod` YAML file:

```
apiVersion: v1
kind: Pod
metadata:
  name: analyzerpod
  namespace: mirror-ns
  labels:
    core.juniper.net/analyzer-pod: analyzerpod
  annotations:
    core.juniper.net/analyzer-interface: "true"
```

```
    k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "vn-1",
          "namespace": "mirror-ns",
          "cni-args": {
            "core.juniper.net/analyzer-interface": "true"
          }
        }
      ]
```

2. Add the pod annotations and specify the `mirroringDestination` resource name on the interface to be mirrored.

   In the following example YAML file, we enable mirroring on the pod `vn-1` interface. We specify the `MirrorDestination` resource name `mirrordestinationprofile1` on the interface to be mirrored.

   Example `Pod/mirrored-pod` YAML file:

```
apiVersion: v1
kind: Pod
metadata:
  name: mirrored-pod
  namespace: mirror-ns
  annotations:
    core.juniper.net/mirror-destination: "mirrordestinationprofile1"
    k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "vn-1",
          "namespace": "mirror-ns",
          "cni-args": {
            "core.juniper.net/mirror-destination": "mirrordestinationprofile1"
          }
        }
      ]
```

Summary

SUMMARY	IN THIS SECTION
This section describes configuration changes for port-based mirroring in CN2 Release 22.2.	

From the analyzer pod annotations and labels, the VM and VMI are associated with the pod to be used in the `MirrorDestination` controller.

Analyzer VM Labels:

The `VirtualMachine` resource corresponding to the pod will have the label `core.juniper.net/analyzer-pod` label.

```
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualMachine
metadata:
  annotations:
    kube-manager.juniper.net/pod-cluster-name: contrail-k8s-kubemanager-ocp-kparmar-6mpccd
    kube-manager.juniper.net/pod-name: analyzerpod
    kube-manager.juniper.net/pod-namespace: multinode-ns
  labels:
    core.juniper.net/analyzer-pod: analyzerpod
```

Analyzer VMI Labels:

The `VirtualMachineInterface` resource for the analyzer pod will have the label `core.juniper.net/analyzer-interface`.

```
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualMachineInterface
metadata:
  annotations:
    index: 0/1
    interface: eth0
    kube-manager.juniper.net/pod-cluster-name: contrail-k8s-kubemanager-ocp-kparmar-6mpccd
    kube-manager.juniper.net/pod-name: analyzerpod
    kube-manager.juniper.net/pod-namespace: multinode-ns
  labels:
    core.juniper.net/analyzer-interface: ""
```

Source VMI Label indicating `MirrorDestination`:

Source `VirtualMachineInterface` corresponding to the pod interface being mirrored will have the label `core.juniper.net/mirror-destination`. The annotations will have the mirror configuration.

```
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualMachineInterface
metadata:
  annotations:
    core.juniper.net/mirroring-configuration:
'{"analyzer_name":"mirrordestinationprofile1","analyzer_ip_address":"10.128.0.200","analyzer_maca
ddress":"02:76:6c:25:f2:8c","ri":"default-
    domain:contrail-k8s-kubemanager-ocp-kparmar-6mpccd-contrail:default-podnetwork:default-
podnetwork"}'
  labels:
    core.juniper.net/mirror-destination: mirrordestinationprofile1
```

Configurable Categories of Metrics Collection and Reporting (Tech Preview)

SUMMARY

In Juniper® Cloud-Native Contrail Networking (CN2) Release 22.2, you can enable and disable selected metrics for exporting.

IN THIS SECTION

- Overview: Configurable Categories of Metrics Collection and Reporting | 366
- Install and Upgrade | 367
- Manage MetricGroup with Kubectl Commands | 368
- Manage Metric Groups with UI | 369

Overview: Configurable Categories of Metrics Collection and Reporting

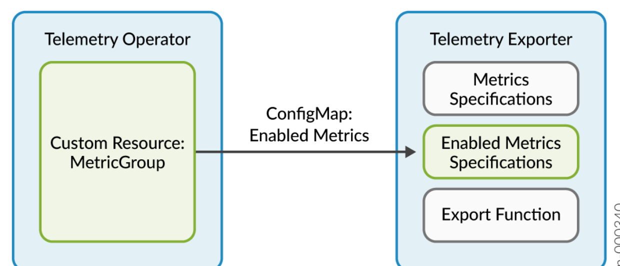
To provide more flexibility in the telemetry export component, CN2 Release 22.2 introduces a new Kubernetes custom resource: `MetricGroup`. `MetricGroup` allows you to enable or disable selected metrics for exporting.

- `MetricGroup` contains and manages a set of metrics for exporting.
- Metrics are grouped by their category. You can choose to enable or disable the metric export function at the group level.
- `MetricGroup` is implemented through a Kubernetes custom resource.

`MetricGroup` provides fine-grained control on which metrics the system collects and reports. You can turn on and off a subset of metrics reporting. At times, you may want to collect only a subset of metrics for efficiency and the lightest weight deployment possible.

> **NOTE**: This feature is classified as a Juniper CN2 Technology Preview feature. These features are "as is" and are for voluntary use. Juniper Support will attempt to resolve any issues that customers experience when using these features and create bug reports on behalf of support cases. However, Juniper may not provide comprehensive support services to Tech Preview features.
> For additional information, see "Juniper CN2 Technology Previews (Tech Previews)" on page 371 or contact Juniper Support.

Figure 18: Metrics Collection and Reporting Architecture



Telemetry Operator, see Figure 18 on page 366, monitors any change of metric groups. Based on the enabled metric groups, a list of enabled metrics is created and sent in the form of `ConfigMap` to metric export agents. Metric export agents collect and export these enabled metrics, instead of all metrics on the system.

- The MetricGroup reconciler builds a ConfigMap for each type of metric (vrouter or controller) from the enabled MetricGroup(s) and applies the ConfigMap to all clusters.
- The kube-manager reconciler does the same for a new cluster.

Telemetry Exporter combines metric specifications with this ConfigMap to create enabled metric specifications. The metric export function only exports metrics from the enabled metric specifications, instead of all metrics.

The following items list the YAML values for ConfigMap and MetricGroup.

ConfigMap: vrouter-export-enabled-metrics

- Revision number
- Array of enabled metric names

Custom Resource: MetricGroup

- Type: vrouter or controller
- Name: String
- Export: Boolean
- Metrics: Array of strings (metric name)

Install and Upgrade

MetricGroup is included in the analytics component in CN2 Release 22.2. The predefined metric groups are automatically installed during the CN2 analytics deployment. See Install Contrail Analytics for Upstream Kubernetes or Install Contrail Analytics for OpenShift Container Platform.Install Contrail Analytics for Amazon EKS.

Example: Predefined Metric Group

```
Bgpaas
Controller-bgp
Controller-info
Controller-peer
Controller-xmpp
Ermvpn
Evpn
Ipv4
```

```
Ipv6
Mvpn
Vrouter-cpu
Vrouter-info
Vrotuer-inv6
Vrouter-mem
Vrouter-traffic
Vrouter vmi
```

Example predefined `MetricGroup`: `vrouter-cpu` YAML file:

```
apiVersion: telemetry.juniper.net/v1alpha1
kind: MetricGroup
metadata:
  name: vrouter-cpu
  namespace: contrail-analytics
spec:
  export: true
  metricType: VROUTER
  metrics:
  - virtual_router_cpu_1min_load_avg
  - virtual_router_cpu_5min_load_avg
  - virtual_router_cpu_15min_load_avg
```

Manage MetricGroup with Kubectl Commands

You (the administrator) can manage `MetricGroup` with `kubectl` commands. Examples follow.

To delete `MetricGroup`:

```
kubectl delete metricgroup ipv6 -n contrail-analytics
```

To apply `MetricGroup`:

```
kubectl apply -f <yaml file with metric group definition>
```

To view `MetricGroup` resource:

```
kubectl get metricgroup ipv4 -n contrail-analytics -oyaml
```

To verify the existence of `ConfigMap`(s) run the following command.

```
kubectl get cm -n contrail

Names of ConfigMap
    controller-export-enabled-metrics
    vrouter-export-enabled-metrics
```

Each cluster has its own copy of the two `ConfigMap`(s); `controller-export-enabled-metrics` and `vrouter-export-enabled-metrics`.

Manage Metric Groups with UI

With this Tech Preview, you can manage `MetricGroups` using the CN2 Manager UI.
To manage Metric Groups in the UI:

1. Access the CN2 Manager UI from your browser:

   `https://`<**cluster-ip-address**>`/`

2. Log in to CN2 Manager by either method:

   - Browse and select a `kubeconfig` file to upload.

   - Log in using a token.

3. From the left-navigation menu, select **Configure** > **Metric Groups**.
   The Metric Groups window appears.

4. To add a Metric Group, click the "**+**" icon in the upper right.

   Add the **Name**, select **Type**, and select the metrics to apply. Click **Save**.

Figure 19: Add a Metric Group

Add Metric Group

Name *

Type*
Controller
vRouter

Metrics
Show Selected

	Type	Name	Description
	gauge	controller_state	Controller state (0=Functional, 1=Non-Functional)
	gauge	controller_connection_status	Connection status (0=Up, 1=Down, 2=Initializing)
	gauge	controller_bgp_router_output_queue_depth	BGP router output queue depth

110 items

Status
Enabled

Cancel
Save

5. Click the detail icon to display the Metric Group you added.

Figure 20: Display Metric Group Detail

Configure / Metric Groups

Metric Groups

	Name	Type	Status
	test1	CONTROLLER	Enabled

1 items
Detail

Details for test1

Type CONTROLLER
Status Enabled

Metrics

Type	Name	Description
gauge	controller_st...	Controller state (...
gauge	controller_co...	Connection stat...

2 items

Juniper CN2 Technology Previews (Tech Previews)

Tech Previews give you the ability to test functionality and provide feedback during the development process of innovations that are not final production features. The goal of a Tech Preview is for the feature to gain wider exposure and potential full support in a future release. We encourage you to provide feedback and functionality suggestions for a Technology Preview feature before it becomes fully supported.

Tech Previews may not be functionally complete, may have functional alterations in future releases, or may get dropped under changing markets or unexpected conditions, at Juniper's sole discretion. Juniper recommends that you use Tech Preview features in non-production environments only.

Juniper considers feedback to add and improve future iterations of the general availability of the innovations. Your feedback does not assert any intellectual property claim, and Juniper may implement your feedback without violating your or any other party's rights.

These features are "as is" and are for voluntary use. Juniper Support will attempt to resolve any issues that you experience when using these features and create bug reports on behalf of support cases. However, Juniper may not provide comprehensive support services for Tech Preview features. Certain features may have reduced or modified security, accessibility, availability, and reliability standards relative to General Availability software. Tech Preview features are not eligible for P1/P2 JTAC cases and are not subject to existing SLAs or service agreements.

For additional details, please contact Juniper Support or your local account team.