Engineering
Simplicity

Juniper Cloud Native Router User Guide

Published
2023-12-27

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

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.

Juniper Cloud Native Router User Guide

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

Table of Contents

1

Introduction

Juniper Cloud-Native Router Overview | 2

Juniper Cloud-Native Router Components | 5

JCNR Deployment Modes | 10

JCNR Interfaces Overview | 11

2

Common Features (All Deployment Modes)

JCNR Common Features | 24

Enabling Dynamic Device Personalization (DDP) on Individual Interfaces | 24

VLAN Sub-Interfaces | 26

3

L2 Features

L2 Features Overview | 30

Access Control Lists (Firewall Filters) | 30

MAC Learning and Aging | 33

Storm Control | 36

APIs and CLI Commands for Bond Interfaces | 38

Quality of Service (QoS) | 41

Native VLAN | 47

Prevent Local Switching | 48

4

L3 Features

L3 Features Overview | 53

IPsec Security Services | 53

JCNR as a Transit Gateway | 54

EVPN Type 5 Routing over VXLAN Tunnels | 56

Integrated Routing and Bridging on JCNR | 64

L3 Routing Protocols | 71

MPLS Support | 75

Bidirectional Forwarding Detection (BFD) | 76

Virtual Router Redundancy Protocol (VRRP) | 77

Virtual Routing Instance (VRF-Lite) | 78

ECMP | 81

BGP Unnumbered | 82

5 JCNR CNI Configuration Examples

JCNR Use-Cases and Configuration Overview | 84

L2 Kernel Access-Mode Interface Configuration Example | 89

Overview | 90

Configuration Example | 90

L2 `virtio` Trunk-Mode Interface Configuration Example | 94

Overview | 94

Configuration Example | 95

L2 VLAN Sub-Interface Configuration Example | 99

Overview | 99

Configuration Example | 100

L3 VPN Interface Configuration Example | 104

Overview | 104

Configuration Example | 105

L3 VLAN Sub-Interface Configuration Example | 111

Overview | 111

Configuration Example | 112

6

Monitoring and Logging

Monitor JCNR via CLI | 119

Telemetry Capabilities of Cloud-Native Router | 126

Logging and Notifications | 147

7

Troubleshooting

Troubleshoot via the vRouter CLI | 152

Troubleshoot via Introspect | 164

8

Appendix

Access cRPD CLI | 167

Access vRouter CLI | 168

Juniper Technology Previews (Tech Previews) | 170

1

CHAPTER

Introduction

Juniper Cloud-Native Router Overview | 2

Juniper Cloud-Native Router Components | 5

JCNR Deployment Modes | 10

JCNR Interfaces Overview | 11

Juniper Cloud-Native Router Overview

IN THIS SECTION

- Overview | 2
- Use Cases | 2
- Architecture and Key Components | 3
- Features | 4

Overview

While 5G unleashes higher bandwidth, lower latency and higher capacity, it also brings in new infrastructure challenges such as increased number of base stations or cell sites, more backhaul links with larger capacity and more cell site routers and aggregation routers. Service providers are integrating cloud-native infrastructure in distributed RAN (D-RAN) topologies, which are usually small, leased spaces, with limited power, space and cooling. The disaggregation of radio access network (RAN) and the expansion of 5G data centers into cloud hyperscalers has added newer requirements for cloud-native routing.

The Juniper Cloud-Native Router provides the service providers the flexibility to roll out the expansion requirements for 5G rollouts, reducing both the CapEx and OpEx.

Juniper Cloud-Native Router (JCNR) is a containerized router that combines Juniper's proven routing technology with the Junos containerized routing protocol daemon (cRPD) as the controller and a high-performance Contrail® Data Plane Development Kit (DPDK) vRouter forwarding plane. It is implemented in Kubernetes and interacts seemlessly with a Kubernetes container network (CNI) framework.

Use Cases

The Cloud-Native Router has the following use cases:

- **Radio Access Network (RAN)**

  The new 5G-only sites are a mix of centralized RAN (C-RAN) and distributed RAN (D-RAN). The C-RAN sites are typically large sites owned by the carrier and continue to deploy physical routers. The D-RAN sites, on the other hand, are tens of thousands of smaller sites, closer to the users.

Optimization of CapEx and OpEx is a huge factor for the large number of D-RAN sites. These sites are also typically leased, with limited space, power and cooling capacities. There is limited connectivity over leased lines for transit back to the mobile core. Juniper Cloud-Native Router is designed to work in the constraints of a D-RAN. It is integrated with the distributed unit (DU) and installable on an existing 1 U server.

- **Telco virtual private cloud (VPC)**

  The 5G data centers are expanding into cloud hyperscalers to support more radio sites. The cloud-native routing available in public cloud environments do not support the routing demands of telco VPCs, such as MPLS, quality of service (QoS), L3 VPN, and more. The Juniper Cloud-Native Router integrates directly into the cloud as a containerized network function (CNF), managed as a cloud-native Kubernetes component, while providing advanced routing capabilities.

Architecture and Key Components

The Juniper Cloud-Native Router consists of the Junos containerized routing protocol Daemon (cRPD) as the control plane (JCNR Controller), providing topology discovery, route advertisement and forwarding information base (FIB) programming, as well as dynamic underlays and overlays. It uses the Data Plane Development Kit (DPDK) enabled vRouter as a forwarding plane, providing packet forwarding for DPDK applications in a pod and host path I/O for protocol sessions. The third component is the JCNR container network interface (CNI) that interacts with Kubernetes as a secondary CNI to create pod interfaces, assign addresses and generate the router configuration.

The Data Plane Development Kit (DPDK) is an open source set of libraries and drivers. 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. The applications poll for packets, to avoid the overhead of interrupts from the NIC. Integrating with DPDK allows a vRouter to process more packets per second than is possible when the vRouter runs as a kernel module.

In this integrated solution, the JCNR Controller uses gRPC, a high performance Remote Procedure Call, based services to exchange messages and to communicate with the vRouter, thus creating the fully functional Cloud-Native Router. This close communication allows you to:

- Learn about fabric and workload interfaces.
- Provision DPDK- or kernel-based interfaces for Kubernetes pods as needed.
- Configure IPv4 and IPv6 address allocation for Pods.
- Run routing protocols such as ISIS, BGP, and OSPF.

Features

- Easy deployment, removal, and upgrade on general purpose compute devices using Helm.
- Higher packet forwarding performance with DPDK-based JCNR-vRouter.
- Full routing, switching, and forwarding stacks in software.
- Out-of-the-box software-based open radio access network (O-RAN) support.
- Quick spin up with containerized deployment.
- Highly scalable solution.
- L3 features such as transit gateway, support for routing protocols, BFD, VRRP, VRF-Lite, EVPN Type-5, ECMP and BGP Unnumbered.
- L2 functionality, such as MAC learning, MAC aging, MAC limiting, native VLAN and L2 statistics.
- L2 reachability to Radio Units (RU) for management traffic.
- L2 or L3 reachability to physical distributed units (DU) such as 5G millimeter wave DUs or 4G DUs.
- VLAN tagging and bridge domains.
- Trunk and access ports.
- Support for multiple virtual functions (VF) on Ethernet NICs.
- Support for bonded VF interfaces.
- Configurable L2 access control lists (ACLs).
- Rate limiting of egress broadcast, unknown unicast, and multicast traffic on fabric interfaces.
- IPv4 and IPv6 routing.

Juniper Cloud-Native Router Components

SUMMARY

The Juniper Cloud-Native Router solution consists of several components including the JCNR controller, JCNR vRouter and the JCNR-CNI. This topic provides a brief overview of the components of the Juniper Cloud-Native Router.

IN THIS SECTION

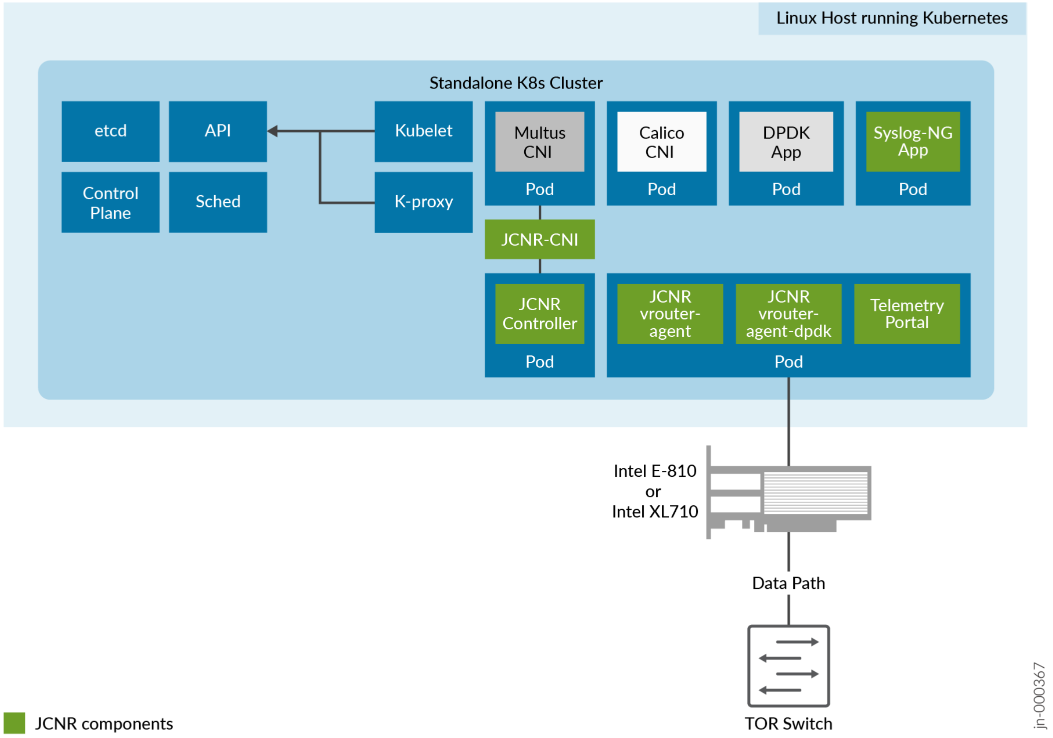
- JCNR Components | 5
- JCNR Controller | 6
- JCNR vRouter | 7
- JCNR-CNI | 8
- Syslog-NG | 9

JCNR Components

The Juniper Cloud-Native Router has primarily three components—JCNR Controller control plane, the JCNR vRouter DPDK forwarding plane and JCNR-CNI for Kubernetes integration. All JCNR components are deployed as containers.

The Figure 1 on page 6 shows the components of the Juniper Cloud-Native Router inside a Kubernetes cluster

Figure 1: Components of Juniper Cloud-Native Router



JCNR Controller

The JCNR Controller is the control-plane of the cloud-native router solution that runs the Junos containerized routing protocol Daemon (cRPD). It is implemented as a statefulset. The controller communicates with the other elements of the cloud-native router. Configuration, policies and rules that you set on the controller at deployment time are communicated to other components, primarily the JCNR vRouter, for implementation.

For example, firewall filters (ACLs) are supported on the controller to configure L2 access lists with deny rules. The controller sends the configuration information to the JCNR vRouter through the vRouter agent.

Juniper Cloud-Native Router Controller Functionality:

- Exposes Junos OS compatible CLI configuration and operation commands that are accessible to external automation and orchestration systems using the NETCONF protocol.

- Supports vRouter as the high-speed forwarding plane. This enables applications that are built using the DPDK framework to send and receive packets directly to the application and the vRouter without passing through the kernel.
- Supports configuration of VLAN-tagged sub-interfaces on physical function (PF), virtual function (VF), virtio, access, and trunk interfaces managed by the DPDK-enabled vRouter.
- Supports configuration of bridge domains, VLANs, and virtual-switches.
- Advertises DPDK application reachability to core network using routing protocols primarily with BGP, IS-IS and OSPF.
- Distributes L3 network reachability information of the pods inside and outside a cluster.
- Maintains configuration for L2 firewall.
- Passes configuration information to the vRouter through the vRouter-agent.
- Stores license key information.
- Works as a BGP Speaker from Release 23.2, establishing peer relationships with other BGP speakers to exchange routing information.

Configuration Options

During deployment, you can *Customize JCNR Configuration* .

After deployment, we recommend that you use the NETCONF protocol with PyEZ to configure the controller. You can SSH or connect via NETCONF. Finally, you can also configure the cloud-native router by "accessing the JCNR controller CLI" on page 167 using Kubernetes commands.

JCNR vRouter

The JCNR vRouter is a high-performance datapath component. It is an alternative to the Linux bridge or the Open vSwitch (OVS) module in the Linux kernel. It runs as a user-space process and is integrated with the Data Plane Development Kit (DPDK) library. The vRouter pod consists of three containers—vrouter-agent, vrouter-agent-dpdk and vrouter-telemetry-exporter.

JCNR vRouter Functionality:

- Performs routing with Layer 3 virtual private networks.
- Performs L2 forwarding.
- Supports high-performance DPDK-based forwarding.

Benefits of vRouter:

- Integration of the DPDK into the JCNR-vRouter.
- Forwarding plane provides faster forwarding capabilities than kernel-based forwarding.
- Forwarding plane is more scalable than kernel-based forwarding.
- Support for the following NICs:
  - Intel E810 (Columbiaville) family
  - Intel XL710 (Fortville) family

JCNR-CNI

JCNR-CNI is a new container network interface (CNI) developed by Juniper. JCNR-CNI is a Kubernetes CNI plugin installed on each node to provision network interfaces for application pods. During pod creation, Kubernetes delegates pod interface creation and configuration to JCNR-CNI. JCNR-CNI interacts with JCNR controller and the vRouter to setup DPDK interfaces. When a pod is removed, JCNR-CNI is invoked to de-provision the pod interface, configuration, and associated state in Kubernetes and cloud-native router components. JCNR-CNI works as a secondary CNI, along with the Multus CNI to add and configure pod interfaces.

JCNR-CNI Functionality:

- Manages the networking tasks in Kubernetes pods such as:
  - assigning IP addresses.
  - allocating MAC addresses.
  - setting up untagged, access, and other interfaces between the pod and vRouter in a Kubernetes cluster.
  - creating VLAN sub-interfaces.
  - creating L3 interfaces.
- Acts on pod events such as add and delete.
- Generates cRPD configuration.

The JCNR-CNI manages the secondary interfaces that the pods use. It creates the required interfaces based on the configuration in YAML-formatted network attachment definition (NAD) files. The JCNR-CNI configures some interfaces before passing them to their final location or connection point and provides an API for further interface configuration options such as:

- Instantiating different kinds of pod interfaces.
- Creating virtio-based high performance interfaces for pods that leverage the DPDK data plane.
- Creating veth pair interfaces that allow pods to communicate using the Linux Kernel networking stack.
- Creating pod interfaces in access or trunk mode.
- Attaching pod interfaces to bridge domains and virtual routers.
- Supporting IPAM plug-in for Dynamic IP address allocation.
- Allocating unique socket interfaces for virtio interfaces.
- Managing the networking tasks in pods such as assigning IP addresses and setting up of interfaces between the pod and vRouter in a Kubernetes cluster.
- Connecting pod interface to a network including pod-to-pod and pod-to-network.
- Integrating with the vRouter for offloading packet processing.

Benefits of JCNR-CNI:

- Improved pod interface management
- Customizable administrative and monitoring capabilities
- Increased performance through tight integration with the controller and vRouter components

The Role of JCNR-CNI in Pod Creation:

When you create a pod for use in the cloud-native router, the Kubernetes component known as **kubelet** calls the Multus CNI to set up pod networking and interfaces. Multus reads the annotations section of the **pod.yaml** file to find the NADs. If a NAD points to JCNR-CNI as the CNI plug in, Multus calls the JCNR-CNI to set up the pod interface. JCNR-CNI creates the interface as specified in the NAD. JCNR-CNI then generates and pushes a configuration into the controller.

Syslog-NG

Juniper Cloud-Native Router uses a syslog-ng pod to gather event logs from cRPD and vRouter and transform the logs into JSON-based notifications. The notifications are logged to a file. Syslog-ng runs as a daemonset.

JCNR Deployment Modes

SUMMARY

Read this topic to know about the various modes of deploying the cloud-native router.

IN THIS SECTION

- Deployment Modes | 10

Deployment Modes

Starting with Juniper Cloud-Native Router Release 23.2, you can deploy and operate Juniper Cloud-Native Router in L2, L3 and L2-L3 modes, auto-derived based on the interface configuration in the `values.yaml` file prior to deployment.

NOTE: In the `values.yaml` file:

- When all the interfaces have an `interface_mode` key configured, then the mode of deployment would be L2.
- When one or more interfaces have an `interface_mode` key configured and some of the interfaces do not have the `interface_mode` key configured, then the mode of deployment would be L2-L3.
- When none of the interfaces have the `interface_mode` key configured, then the mode of deployment would be L3.

In L2 mode, the cloud-native router behaves like a switch and therefore does not performs any routing functions and it doesn not run any routing protocols. The pod network uses VLANs to direct traffic to various destinations.

In L3 mode, the cloud-native router behaves like a router and therefore performs routing functions and runs routing protocols such as ISIS, BGP, OSPF, and segment routing-MPLS. In L3 mode, the pod network is divided into an IPv4 or IPv6 underlay network and an IPv4 or IPv6 overlay network. The underlay network is used for control plane traffic.

The L2-L3 mode provides the functionality of both the switch and the router at the same time. It enables JCNR to act as both a switch and a router simultaneously by performing switching in a set of interfaces and routing in the other set of interfaces. Cell site routers in a 5G deployment need to handle both L2 and L3 traffic. DHCP packets from radio outdoor unit (RU) is an example of L2 traffic and data packets moving from outdoor unit (ODU) to central unit (CU) is an example of L3 traffic.

JCNR Interfaces Overview

SUMMARY

This topic provides information on the network communication interfaces provided by the JCNR-Controller. Fabric interfaces are aggregated interfaces that receive traffic from multiple interfaces. Interfaces to which different workloads are connected are called workload interfaces.

IN THIS SECTION

- Juniper Cloud-Native Router Interface Types | 11

Read this topic to understand the network communication interfaces provided by the JCNR-Controller. We cover interface names, what they connect to, how they communicate. and the services they provide.

Juniper Cloud-Native Router Interface Types

Juniper Cloud-Native Router supports two types of interfaces:

- **Fabric interfaces**—Aggregated interfaces that receive traffic from multiple interfaces. Fabric interfaces are always physical interfaces. They can either be a physical function (PF) or a virtual function (VF). The throughput requirement for these interfaces is higher, hence multiple hardware queues are allocated to them. Each hardware queue is allocated with a dedicated CPU core . The interfaces are configured for the cloud-native router using the appropriate `values.yaml` file in the deployer helmcharts. You can view the interface mapping using the `dpdkinfo -c` command. View the Troubleshoot via the vRouter CLI topic in the Deployment Guide for more details. You also have fabric workload interfaces that have low throughput requirement. Only one hardware queue is allocated to the interface, thereby saving precious CPU resources. These interfaces can be configured using the appropriate `values.yaml` file in the deployer helmcharts.

- **Workload interfaces**—Interfaces to which different workloads are connected. They can either be software-based or hardware-based interfaces. Software-based interfaces are either high-performance interfaces using the Data Plane Development Kit (DPDK) poll mode driver (PMD) or a low-performance interfaces using the kernel driver. Typically the DPDK interfaces are used for data traffic such as the GPRS Tunneling Protocol for user data (GTP-U) traffic and the kernel-based interfaces are used for control plane data traffic such as TCP. The kernel pod interfaces are typically for the operations, administration and maintenance (OAM) traffic. The interfaces are configured as a veth-pair, with one end of the interface in the pod and the other end in the Linux kernel on the host. JCNR also supports bonded interfaces via the link bonding PMD. These interfaces can be configured using the appropriate `values.yaml` file in the deployer helmcharts.

JCNR supports different types of VLAN interfaces including trunk, access and sub-interfaces across fabric and workload interfaces.

JCNR Interface Details

The different JCNR interfaces are provided in detail below:

- **Agent interface**

  vRouter has only one agent interface. The agent interface enables communication between the vRouter-agent and the vRouter. On the vRouter CLI when you issue the `vif --list` command, the agent interface looks like this:

```
vif0/0      Socket: unix
            Type:Agent HWaddr:00:00:5e:00:01:00
            Vrf:65535 Flags:L2 QOS:-1 Ref:3
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:650  bytes:99307 errors:0
            Drops:0
```

- **DPDK VF workload interfaces**

  These interfaces connect to the radio units (RUs) or millimeter-wave distributed units (mmWave-DUs). On the vRouter CLI when you issue the `vif --list` command, the DPDK VF workload interface looks like this:

```
vif0/5      PCI: 0000:ca:19.1 (Speed 10000, Duplex 1)
            Type:Workload HWaddr:9e:52:29:9e:97:9b
            Vrf:0 Flags:L2Vof QOS:-1 Ref:9
            RX queue  packets:29087 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: 0000:ca:19.1  Status: UP  Driver: net_iavf
            Vlan Mode: Access  Vlan Id: 1250  OVlan Id: 1250
            RX packets:29082  bytes:6766212 errors:5
            TX packets:0  bytes:0 errors:0
            Drops:29896
```

- **DPDK VF fabric interfaces (Physical Trunk)**

DPDK VF fabric interfaces, which are associated with the physical network interface card (NIC) on the host server, accept traffic from multiple VLANs.

The cRPD interface configuration using the `show configuration` command looks like this (the output is trimmed for brevity):

```
interfaces {
    ens786f0v0 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 1001-1100;
            }
        }
    }
}
```

On the vRouter CLI when you issue the `vif --list` command, the DPDK VF fabric interface looks like this:

```
vif0/1    PCI: 0000:31:01.0 (Speed 10000, Duplex 1)
          Type:Physical HWaddr:d6:22:c5:42:de:c3
          Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
          RX queue packets:11813 errors:1
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 1 0
          Fabric Interface: 0000:31:01.0 Status: UP Driver: net_iavf
          Vlan Mode: Trunk Vlan: 1001-1100
          RX packets:0 bytes:0 errors:49962
          TX packets:18188356 bytes:2037400554 errors:0
          Drops:49963
```

- Active or standby bond interfaces (Bond Trunk)

  Bond interfaces accept traffic from multiple VLANs. A bond interface runs in the active or standby mode (mode 0). You define the bond interface in the helm chart configuration as follows:

```
bondInterfaceConfigs:
- name: "bond0"
  mode: 1          # ACTIVE_BACKUP MODE
  slaveInterfaces:
  - "ens2f0v1"
  - "ens2f1v1"
```

```
   - bond0:
        ddp: "auto"
        interface_mode: trunk
        vlan-id-list: [1001-1100]
        storm-control-profile: rate_limit_pf1
        native-vlan-id: 1001
        no-local-switching: true
```

The cRPD interface configuration using the `show configuration` command looks like this (the output is trimmed for brevity):

```
interfaces {
    bond0 {
         unit 0 {
             family bridge
             interface-mode trunk;
             vlan-id-list 1001-1100;
         }
    }
}
```

On the vRouter CLI when you issue the `vif --list` command, the bond interface looks like this:

```
vif0/2      PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
            Type:Physical HWaddr:32:f8:ad:8c:d3:bc
            Vrf:65535 Flags:L2Vof QOS:-1 Ref:8
            RX queue  packets:1882 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: eth_bond_bond0  Status: UP  Driver: net_bonding
            Slave Interface(0): 0000:81:01.0  Status: UP  Driver: net_iavf
            Slave Interface(1): 0000:81:03.0  Status: UP  Driver: net_iavf
            Vlan Mode: Trunk  Vlan: 1001-1100
            RX packets:8108366000  bytes:486501960000 errors:4234
            TX packets:65083776  bytes:4949969408 errors:0
            Drops:8108370394
```

- Pod interfaces using DPDK data plane (Virtio Trunk) virtio

The trunk interfaces accept only tagged packets. Any untagged packets are dropped. These interfaces can accept a VLAN filter to allow only specific VLAN packets. A trunk interface can be a part of multiple bridge-domains (BD). A bridge domain is a set of logical ports that share the same flooding or broadcast characteristics. Like a VLAN, a bridge domain spans one or more ports of multiple devices. Virtio interfaces are associated with pod interfaces that use virtio on the DPDK data plane.

The cRPD interface configuration using the `show configuration` command looks like this (the output is trimmed for brevity):

```
interfaces {
    vhost242ip-93883f16-9ebb-4acf-b {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list 1001-1003;
            }
        }
    }
}
```

On the vRouter CLI when you issue the `vif --list` command, the virtio with DPDK data plane interface looks like this:

```
vif0/3    PMD: vhost242ip-93883f16-9ebb-4acf-b
          Type:Virtual HWaddr:00:16:3e:7e:84:a3
          Vrf:65535 Flags:L2 QOS:-1 Ref:13
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          Vlan Mode: Trunk Vlan: 1001-1003
          RX packets:0 bytes:0 errors:0
          TX packets:10604432 bytes:1314930908 errors:0
          Drops:0
          TX port packets:0 errors:10604432
```

- **Pod interfaces using Kernel interface**

  The access interfaces accept both tagged and untagged packets. Untagged packets are tagged with the access VLAN or access BD. Any tagged packets other than the ones with access VLAN are dropped. The access interfaces is a part of a single bridge-domain. It does not have any parent interface.

The cRPD interface configuration using the `show configuration` command looks like this (the output is trimmed for brevity):

```
routing-instances {
    switch {
         instance-type virtual-switch;
         bridge-domains 
{

             bd1001 {
                 vlan-id 1001;
                 interface jvknet1-eed79ff;
             }
         }
    }
}
```

On the vRouter CLI when you issue the `vif --list` command, the veth pair interface looks like this:

```
vif0/4      Ethernet: jvknet1-88c44c3
            Type:Virtual HWaddr:02:00:00:3a:8f:73
            Vrf:0 Flags:L2Vof QOS:-1 Ref:10
            RX queue packets:524 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Vlan Mode: Access Vlan Id: 1001 OVlan Id: 1001
            RX packets:9 bytes:802 errors:515
            TX packets:0 bytes:0 errors:0
            Drops: 525
```

- L2 VLAN sub-interfaces

  You can configure a user pod with a Layer 2 VLAN sub-interface and attach it to the JCNR instance. VLAN sub-interfaces are like logical interfaces on a physical switch or router. They access only tagged packets that match the configured VLAN tag. A sub-interface has a parent interface. A parent interface can have multiple sub-interfaces, each with a VLAN ID. When you run the cloud-native router, you must associate each sub-interface with a specific VLAN.

  The cRPD interface configuration viewed using the `show configuration` command is as shown below (the output is trimmed for brevity).

For **L2**:

```
routing-instances {
    switch {
         instance-type virtual-switch;
         bridge-domains
{

             bd100 {
                 vlan-id 100;
                 interface vhostnet1-1e555ee1-7d93-40.100;
             }
         }
     }
}
```

On the vRouter, a VLAN sub-interface configuration is as shown below:

```
vif0/5      Virtual: vhostnet1-71cd7db1-1a5e-49.3003 Vlan(o/i)(,S): 3003/3003 Parent:vif0/4
            Type:Virtual(Vlan) HWaddr:00:99:99:99:33:09
            Vrf:0 Flags:L2 QOS:-1 Ref:3
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:0
```

NOTE: To see the VLAN sub-interfaces on the vRouter, connect to the vRouter agent by executing the command `kubectl exec -it -n contrail contrail-vrouter-<agent container> -- bash` command, and then run the command `vif --get`.

- L3 Physical Interface

```
vif0/1      PCI: 0000:17:01.1 (Speed 25000, Duplex 1) NH: 7 MTU: 9000 <- PCI
Address
            Type:Physical HWaddr:d6:93:87:91:45:6c IPaddr: 192.21.2.4 <- Physical interface
            IP6addr:2001:192:21:2::4 <- IPv6 address
            DDP: OFF SwLB: ON
            Vrf:2 Mcast Vrf:2 Flags:L3L2Vof QOS:0 Ref:16 <- L3 (only) interface
            RX port   packets:423168341 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: 0000:17:01.1  Status: UP  Driver: net_iavf
            RX packets:423168341  bytes:29123418594 errors:0
            TX packets:417508247  bytes:417226216530 errors:0
            Drops:8
            TX port   packets:417508247 errors:0
```

```
vif0/2      PMD: ens2f2 NH: 12 MTU: 9000 <- Tap interface name as seen by cRPD
            Type:Host HWaddr:d6:93:87:91:45:6c IPaddr: 192.21.2.4 <- Tap interface type
            IP6addr:2001:192:21:2::4
            DDP: OFF SwLB: ON
            Vrf:2 Mcast Vrf:65535 Flags:L3DProxyEr QOS:-1 Ref:15 TxXVif:1  <-cross-connected
to vif 1
            RX device packets:306995  bytes:25719830 errors:0
            RX queue  packets:306995 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:306995  bytes:25719830 errors:0
            TX packets:307489  bytes:25880250 errors:0
            Drops:0
            TX queue  packets:307489 errors:0
            TX device packets:307489  bytes:25880250 errors:0
```

Corresponding interface state in the cRPD:

```
show interfaces routing ens2f2
Interface        State Addresses
ens2f2           Up    MPLS  enabled
                       ISO   enabled
                       INET  192.21.2.4
                       INET6 2001:192:21:2::4
                       INET6 fe80::c5da:7e9c:e168:56d7
                       INET6 fe80::a0be:69ff:fe59:8b58
```

L3 Bond Interface

```
vif0/3      PCI: 0000:00:00.0 (Speed 25000, Duplex 1) NH: 6 MTU: 1514 <- Bond interface (PCI
id 0)
            Type:Physical HWaddr:50:7c:6f:48:75:74 IPaddr:192.7.7.4 <- Physical interface
            IP6addr:2001:192:7:7::4
            DDP: OFF SwLB: ON
            Vrf:1 Mcast Vrf:1 Flags:TcL3L2Vof QOS:0 Ref:18
            RX port   packets:402183888 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: eth_bond_bond34  Status: UP  Driver: net_bonding <- Bonded 
master
            Slave Interface(0): 0000:5e:00.0  Status: UP  Driver: net_ice <- Bond slave - 1
            Slave Interface(1): 0000:af:00.0  Status: UP  Driver: net_ice <- Bond slave - 2
            RX packets:402183888  bytes:49519387070 errors:0
            TX packets:79226  bytes:7330912 errors:0
            Drops:1393
            TX port   packets:79226 errors:0
```

```
vif0/4      PMD: bond34 NH: 11 MTU: 9000
            Type:Host HWaddr:50:7c:6f:48:75:74 IPaddr:192.7.7.4 <- Tap interface
            IP6addr:2001:192:7:7::4
            DDP: OFF SwLB: ON
            Vrf:1 Mcast Vrf:65535 Flags:L3DProxyEr QOS:-1 Ref:15 TxXVif:3 <- Tap interface 
for bond
            RX device packets:76357  bytes:7101918 errors:0
            RX queue  packets:76357 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:76357  bytes:7101918 errors:0
            TX packets:75349  bytes:6946908 errors:0
            Drops:0
            TX queue  packets:75349 errors:0
            TX device packets:75349  bytes:6946908 errors:0
```

Corresponding interface state in the cRPD:

```
show interfaces routing bond34
Interface        State Addresses
bond34           Up    INET6 2001:192:7:7::4
                       ISO   enabled
                       INET  192.7.7.4
                       INET6 fe80::527c:6fff:fe48:7574
```

- L3 Pod Vhost-User Interface

```
vif0/8      PMD: vhostnet1-aa0984c7-0c1d-40a4-87 NH: 35 MTU: 9160 <- vhost-user interface of 
CNF
```

```
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:2.51.1.3 <- pod/ workload
            IP6addr:abcd:2:51:1::3 <- IPv6 address of the pod
            DDP: OFF SwLB: ON
            Vrf:3 Mcast Vrf:3 Flags:PL3DProxyEr QOS:-1 Ref:14
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:0
```

Corresponding interface state in the cRPD:

```
show interfaces routing vhostnet1-aa0984c7-0c1d-40a4-87
Interface        State Addresses
vhostnet1-aa0984c7-0c1d-40a4-87 Up    INET6 enabled
                                      INET6 abcd:2:51:1::3
                                      ISO   enabled
                                      INET  enabled
                                      INET  2.51.1.3
```

- **L3 Kernel Interface**

```
vif0/13     Ethernet: jvknet1-0af476e NH: 35 MTU: 9160 <- Kernel interface (jvk) of CNF
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:2.51.1.4 <- pod/ workload
            IP6addr:abcd:2:51:1::4
            DDP: OFF SwLB: ON
            Vrf:1 Mcast Vrf:1 Flags:PL3DVofProxyEr QOS:-1 Ref:11
            RX port   packets:47 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:47  bytes:13012 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:47
```

Corresponding interface state in the cRPD:

```
show interfaces routing jvknet1-0af476e
Interface        State Addresses
jvknet1-0af476e  Up    INET6 enabled
                       INET6 abcd:2:51:1::4
                       ISO   enabled
                       INET  enabled
                       INET  2.51.1.4
```

- L3 VLAN Sub-Interfaces

  Starting in Juniper Cloud-Native Router Release 23.2, the cloud-native router supports the use of VLAN sub-interfaces in L3 mode.

```
vif0/2      PCI: 0000:17:01.1 (Speed 25000, Duplex 1) NH: 7 MTU: 9000
            Type:Physical HWaddr:d6:93:87:91:45:6c IPaddr:0.0.0.0
            IP6addr:fe80::d493:87ff:fe91:456c <- IPv6 address
            DDP: OFF SwLB: ON
            Vrf:2 Mcast Vrf:2 Flags:L3L2Vof QOS:0 Ref:16 <- L3 (only) interface
            RX port   packets:423168341 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: 0000:17:01.1  Status: UP  Driver: net_iavf
            RX packets:423168341  bytes:29123418594 errors:0
            TX packets:417508247  bytes:417226216530 errors:0
            Drops:8
            TX port   packets:417508247 errors:0
```

```
vif0/5      PMD: ens1f0v1 NH: 12 MTU: 9000
            Type:Host HWaddr:d6:93:87:91:45:6c IPaddr:0.0.0.0
            IP6addr:fe80::d493:87ff:fe91:456c
            DDP: OFF SwLB: ON
            Vrf:2 Mcast Vrf:65535 Flags:L3DProxyEr QOS:-1 Ref:15 TxXVif:2 <- L3 (only) tap
interface
            RX device packets:306995  bytes:25719830 errors:0
            RX queue  packets:306995 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:306995  bytes:25719830 errors:0
            TX packets:307489  bytes:25880250
errors:0

            Drops:0
            TX queue  packets:307489 errors:0
            TX device packets:307489  bytes:25880250 errors:0
```

```
vif0/9      Virtual: ens1f0v1.201 Vlan(o/i)(,S): 201/201 Parent:vif0/2 NH: 36 MTU: 1514 <-
VLAN fabric sub-intf with parent as vif 2 and VLAN tag as 201
            Type:Virtual(Vlan) HWaddr:d6:93:87:91:45:6c IPaddr:103.1.1.2
            IP6addr:fe80::d493:87ff:fe91:456c
            DDP: OFF SwLB: ON
            Vrf:1 Mcast Vrf:1 Flags:L3DProxyEr QOS:-1 Ref:4
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:0
```

```
vif0/10     Virtual: ens1f0v1.201 Vlan(o/i)(,S): 201/201 Parent:vif0/5 NH: 21 MTU: 9000
            Type:Virtual(Vlan) HWaddr:d6:93:87:91:45:6c IPaddr:103.1.1.2
            IP6addr:fe80::d493:87ff:fe91:456c
            DDP: OFF SwLB: ON
            Vrf:1 Mcast Vrf:65535 Flags:L3DProxyEr QOS:-1 Ref:4 TxXVif:9 <- VLAN tap sub-intf
cross connected to fabric sub-intf vif 9 and parent as tap intf vif 5
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:0
```

Corresponding interface state in cRPD:

```
show interfaces routing ens1f0v1.201
Interface          State Addresses
ens1f0v1.201       Up    MPLS  enabled
                   ISO   enabled
                   INET6 fe80::b89c:fff:feab:e2c9
```

2

CHAPTER

Common Features (All Deployment Modes)

JCNR Common Features | 24

Enabling Dynamic Device Personalization (DDP) on Individual Interfaces | 24

VLAN Sub-Interfaces | 26

JCNR Common Features

SUMMARY

Read this topic to learn about the Juniper Cloud-Native Router common features for all deployment modes.

The Juniper Cloud-Native Router supports multiple "deployment modes" on page 10.

This chapter explains the common features for all deployment modes.

Enabling Dynamic Device Personalization (DDP) on Individual Interfaces

SUMMARY

Dynamic Device Personalization (DDP) is a technology that enables programmable packet processing pipeline provided by Intel as a profile to their NICs. JCNR supports enabling Dynamic Device Personalization (DDP) on individual interfaces.

Starting with Juniper Cloud-Native Router (JCNR) Release 23.2, JCNR supports enabling Dynamic Device Personalization (DDP) on individual interfaces. This feature is available on JCNR in L2, L3, and L2-L3 modes.

Dynamic Device Personalization (DDP) is a technology that enables programmable packet processing pipeline provided by Intel as a profile to their NICs. Multiple Intel NICs support this technology. The support varies based on the Intel NIC type. DDP is used in packet classification where the profiles applied to the NIC can classify multiple packet formats on the NIC enabling speeds and feeds to the Data Plane Development Kit (DPDK).

Juniper cloud native router (JCNR) provides routing and switching functionality. JCNR supports interfaces from different NIC cards. Some of the Intel NICs support DDP and some of them don’t support DDP. Therefore, in a deployment scenario, JCNR might have one interface from one NIC that supports DDP and another interface from a different NIC that does not support DDP. JCNR supports enabling DDP per interface to overcome such issues.

NOTE: For E810 PF, JCNR loads the DDP package which is bundled with JCNR. However, for other NICs, ensure you load the DDP package on the NICs before starting JCNR.

A DDP configuration is available per interface. This configuration option overrides global DDP (`ddp`) configuration for that interface. If you do not configure an interface DDP, then the global configuration value serves as the value for that interface. If you do not configure the global DDP configuration, then the default value for the global configuration which is `off` takes effect.

NOTE: DDP is supported on the following NICs:

- E810 VF
- E810 PF
- X710 PF
- XXV710 PF

DDP support is not available when interfaces are defined under subnets.

You should configure DDP in the helm chart before deployment. Configuring the DDP configurations in the helm charts for both global and at interface levels is optional. If you do not configure the DDP keys, then the default value for global DDP which is `off` takes effect.

The global DDP configuration is available in the `values.yaml` file as shown below:

```
# Set ddp to enable Dynamic Device Personalization (DDP)
# Provides datapath optimization at NIC for traffic like GTPU, SCTP etc.
# Options include auto or on or off; default: off
ddp: "auto"
```

You can configure one of the following options for `ddp` at the interface level:

1. Auto—when set to auto, JCNR checks if the NIC supports DDP or not during deployment and configures DPDK accordingly. Detecting whether a NIC supports DDP at run time makes is easier to deploy JCNR in volumes.

2. On—option enables DDP on the interface without validating the NIC. Use this option only if you are sure that the NIC supports DDP.

3. Off—is the default option at the interface level. This option disables DDP on the interface.

For example,

```
-    eth1:
        ddp: "off" ## auto or on or off
```

NOTE: Each interface can have a different configuration for `ddp`. DDP is enabled for a bond interface only if all the slave interface NICs support DDP.

VLAN Sub-Interfaces

IN THIS SECTION

- Configuration Example | **26**

VLAN sub-interfaces are like logical interfaces on a physical switch or router. They access only tagged packets that match the configured VLAN tag. A sub-interface has a parent interface. A parent interface can have multiple sub-interfaces, each with a VLAN ID. When you run the cloud-native router, you must associate each sub-interface with a specific VLAN. Starting in Juniper Cloud-Native Router Release 23.2, the cloud-native router supports the use of VLAN sub-interfaces in L3 mode along with the previously supported L2 mode.

Configuration Example

The VLAN sub-interfaces are configured using the Netowrk Attachment Definition (NAD) and pod YAML manifests. Please see the "JCNR Use-Cases and Configuration Overview " on page 84 and relevant configuration examples for more information.

The JCNR controller interface configuration viewed using the `show configuration` command is as shown below (the output is trimmed for brevity).

For L2 mode:

```
routing-instances {
    switch {
         instance-type virtual-switch;
         bridge-domains
{

             bd100 {
                 vlan-id 100;
                 interface vhostnet1-1e555ee1-7d93-40.100;
             }
         }
     }
}
```

For L3 mode:

```
enp24s0f0 {
        unit 1 {
            vlan-id 10;
            family inet {
                address 172.168.20.3/24;
            }
        }
}
```

On the vRouter, a VLAN sub-interface configuration is as shown below:

For L2 mode:

```
vif0/5      Virtual: vhostnet1-71cd7db1-1a5e-49.100 Vlan(o/i)(,S): 3003/3003 Parent:vif0/4
            Type:Virtual(Vlan) HWaddr:00:99:99:99:33:09
            Vrf:0 Flags:L2 QOS:-1 Ref:3
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:0
```

For L3 mode:

```
vif0/9      Virtual: ens1f0v1.201 Vlan(o/i)(,S): 201/201 Parent:vif0/2 NH: 36 MTU: 1514
            Type:Virtual(Vlan) HWaddr:d6:93:87:91:45:6c IPaddr:103.1.1.2
            IP6addr:fe80::d493:87ff:fe91:456c
            DDP: OFF SwLB: ON
            Vrf:1 Mcast Vrf:1 Flags:L3DProxyEr QOS:-1 Ref:4
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:0

vif0/10     Virtual: ens1f0v1.201 Vlan(o/i)(,S): 201/201 Parent:vif0/5 NH: 21 MTU: 9000
            Type:Virtual(Vlan) HWaddr:d6:93:87:91:45:6c IPaddr:103.1.1.2
            IP6addr:fe80::d493:87ff:fe91:456c
            DDP: OFF SwLB: ON
            Vrf:1 Mcast Vrf:65535 Flags:L3DProxyEr QOS:-1 Ref:4 TxXVif:9
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:0
```

3

CHAPTER

L2 Features

L2 Features Overview | 30

Access Control Lists (Firewall Filters) | 30

MAC Learning and Aging | 33

Storm Control | 36

APIs and CLI Commands for Bond Interfaces | 38

Quality of Service (QoS) | 41

Native VLAN | 47

Prevent Local Switching | 48

L2 Features Overview

SUMMARY

Read this topic to learn about the features available in the Juniper Cloud-Native Router when deployed in L2 (switch) mode.

The Juniper Cloud-Native Router supports multiple "deployment modes" on page 10.

In L2 mode, the cloud-native router behaves like a switch and so performs no routing functions and runs no routing protocols. The pod network uses VLANs to direct traffic to various destinations.

This chapter provides information about the various L2 features supported by JCNR.

Access Control Lists (Firewall Filters)

SUMMARY

Read this topic to learn about Layer 2 access control lists (Firewall filters) in the cloud-native router.

IN THIS SECTION

- Access Control Lists (Firewall Filters) | 30
- Configuration Example | 31
- Troubleshooting | 32

Access Control Lists (Firewall Filters)

Starting with Juniper Cloud-Native Router Release 22.2 we've included a limited firewall filter capability. You can configure the filters using the Junos OS CLI within the cloud-native router controller, using NETCONF, or the cloud-native router APIs. Starting with Juniper Cloud-Native Router Release 23.2, you can also configure firewall filters using node annotations and custom configuration template at the time of JCNR deployment. Please review the deployment guide for more details.

During deployment, the system defines and applies firewall filters to block traffic from passing directly between the router interfaces. You can dynamically define and apply more filters. Use the firewall filters to:

- Define firewall filters for bridge family traffic.
- Define filters based on one or more of the following fields: source MAC address, destination MAC address, or EtherType.
- Define multiple terms within each filter.
- Discard the traffic that matches the filter.
- Apply filters to bridge domains.

Configuration Example

Below you can see an example of a firewall filter configuration from a cloud-native router deployment:

```
root@jcnr01> show configuration firewall
firewall {
    family {
        bridge {
            filter example {
                term t1 {
                    from {
                        destination-mac-address 10:10:10:10:10:11;
                        source-mac-address 10:10:10:10:10:10;
                        ether-type arp;
                    }
                    then {
                        discard;
                    }
                }
            }
        }
    }
}
```

NOTE:

*then*discard

After configuration, you must apply your firewall filters to a bridge domain using the `set routing-instances vswitch bridge-domains` *`bd3001`* `forwarding-options filter input` *`filter1`* configuration command. Then you must commit the configuration for the firewall filter to take effect.

To see how many packets matched the filter (per VLAN), you can issue the `show firewall filter` *`filter1`* command on the controller CLI. For example:

```
show firewall filter filter1
 Filter : filter1     vlan-id : 3001
 Term                   Packet
  t1                    0
```

In the preceding example, we applied the filter to the bridge domain `bd3001`. The filter has not yet matched any packets.

Troubleshooting

The following table lists some of the potential problems that you might face when you implement firewall rules or ACLs in the cloud-native router. You run most of these commands on the host server.

Table 1: L2 Firewall Filter or ACL Troubleshooting

Problem	Possible Causes and Resolution	Command
Firewall filters or ACLs not working	gRPC connection (port 50052) to the vRouter is down. Check the gRPC connection.	`netstat -antp|grep 50052`
	The `ui-pubd` process is not running. Check whether `ui-pubd` is running.	`ps aux|grep ui-pubd`
Firewall filter or ACL show commands not working	The gRPC connection (port 50052) to the vRouter is down. Check the gRPC connection.	`netstat -antp|grep 50052`

Table 1: L2 Firewall Filter or ACL Troubleshooting *(Continued)*

Problem	Possible Causes and Resolution	Command
	The firewall service is not running.	`ps aux|grep firewall`
		`show log filter.log` You must run this command in the JCNR-controller (cRPD) CLI.

MAC Learning and Aging

SUMMARY

Juniper Cloud-Native Router provides automated learning and aging of MAC addresses. Read this topic for an overview of the MAC learning and aging functionality in the cloud-native router.

IN THIS SECTION

- MAC Learning | 33
- MAC Entry Aging | 35

MAC Learning

MAC learning enables the cloud-native router to efficiently send the received packets to their respective destinations. The cloud-native router maintains a table of MAC addresses grouped by interface. The table includes MAC addresses, VLANs, and the interface on which the vRouter learns each MAC address and VLAN. The MAC table informs the vRouter about the MAC addresses that each interface can reach.

The cloud-native router caches the source MAC address for a new packet flow to record the incoming interface into the MAC table. The router learns the MAC addresses for each VLAN or bridge domain. The cloud-native router creates a key in the MAC table from the MAC address and VLAN of the packet. Queries sent to the MAC table return the interface associated with the key. To enable MAC learning, the cloud-native router performs these steps:

- Records the incoming interface into the MAC table by caching the source MAC address for a new packet flow.

- Learns the MAC addresses for each VLAN or bridge domain.

- Creates a key in the MAC table from the MAC address and VLAN of the packet.

If the destination MAC address and VLAN are missing (lookup failure), the cloud-native router floods the packet out all the interfaces (except the incoming interface) in the bridge domain.

By default:

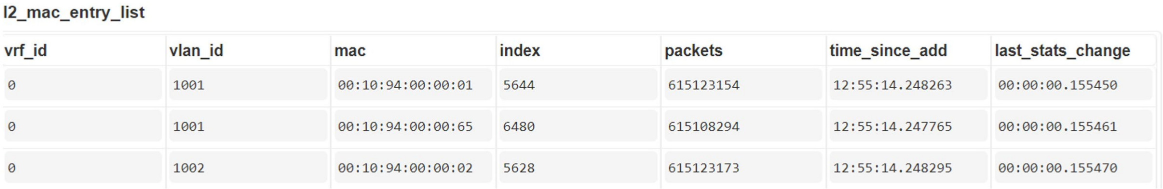
- MAC table entries time out after 60 seconds.

- The MAC table size is limited to 10,240 entries.

We recommend that you do not change the default values. Please contact Juniper Support if you need to change the default values.

You can see the MAC table entries by using:

- Introspect agent at **http://*host server IP*:8085/mac_learning.xml#Snh_FetchL2MacEntry**



l2_mac_entry_list

vrf_id	vlan_id	mac	index	packets	time_since_add	last_stats_change
0	1001	00:10:94:00:00:01	5644	615123154	12:55:14.248263	00:00:00.155450
0	1001	00:10:94:00:00:65	6480	615108294	12:55:14.247765	00:00:00.155461
0	1002	00:10:94:00:00:02	5628	615123173	12:55:14.248295	00:00:00.155470

- The command **show bridge mac-table** on the JCNR controller CLI:

```
show bridge mac-table
Routing Instance : default-domain:default-project:ip-fabric:__default__
Bridging domain VLAN id : 3002
MAC                   MAC                 Logical
address               flags               interface

00:00:5E:00:53:01      D                   bond0
```

- The command **purel2cli --mac show** on the CLI of the vRouter pod:

```
purel2cli --mac show
==================================================
||  MAC             vlan      port      hit_count||
==================================================
00:01:01:01:01:03  1221      2          1101892
00:01:01:01:01:02  1221      2          1101819
00:01:01:01:01:04  1221      2          1101863
```

```
00:01:01:01:01:01  1221      2          1101879
5a:4c:4c:75:90:fe  1250      5          12
Total Mac entries 5
```

If you exceed the MAC address limit, the counter **pkt_drop_due_to_mactable_limit** increments. You can see this counter by using the introspect agent at **http://*host server IP*:8085/Snh_AgentStatsReq**.

If you delete or disable an interface, the cloud-native router deletes all the MAC entries associated with that interface from the MAC table.

MAC Entry Aging

The aging timeout for cached MAC entries is 60 seconds. You can configure the aging timeout at deployment time by editing the **values.yaml** file. The minimum timeout is 60 seconds and the maximum timeout is 10,240 seconds. You can see the time that is left for each MAC entry through introspect at **http://*host server IP*:8085/mac_learning.xml#Snh_FetchL2MacEntry**. We show an example of the output below:

```
l2_mac_entry_list
vrf_id          vlan_id          mac                  index          packets
time_since_add          last_stats_change
0               1001             00:10:94:00:00:01    5644           615123154
12:55:14.248785         00:00:00.155450
0               1001             00:10:94:00:00:65    6480           615108294
12:55:14.247765         00:00:00.155461
0               1002             01:10:94:00:00:02    5628           615123173
12:55:14.248295         00:00:00.155470
```

Storm Control

SUMMARY

Read this topic to understand how the broadcast rate limiting feature is implemented by the cloud-native router when deployed in L2 mode.

IN THIS SECTION

- Configuration Example | 36

The storm control or rate limiting feature controls the rate of egress broadcast, unknown unicast, and multicast (BUM) traffic on fabric interfaces.

Configuration Example

You specify the rate limit in bytes per second by adjusting **stormControlProfiles** in the **values.yaml** file before deployment.

```
# rate limit profiles for bum traffic on fabric interfaces in bytes per second
 stormControlProfiles:
   rate_limit_pf1:
     bandwidth:
       level: 0
```

Once a profile is created, it can be assigned to the interface via the `storm-control-profile` interface attribute. For example:

```
- eth1:
      ddp: on
      interface_mode: trunk
      vlan-id-list: [100, 200, 300, 700-705]
      storm-control-profile: rate_limit_pf1
      native-vlan-id: 100
      no-local-switching: true
```

The system applies the configured profiles to all specified fabric interfaces in the cloud-native router. The maximum per-interface rate limit value you can set is 1,000,000 bytes per second.

If the unknown unicast, broadcast, or multicast traffic rate exceeds the set limit on a specified fabric interface, the vRouter drops the traffic. You can see the drop counter values by running the `dropstats` command in the vRouter CLI. You can see the per-interface rate limit drop counters by running the vRouter CLI command `vif --get` *fabric_vif_id* `--get-drop-stats`. For example:

```
dropstats
L2 untag pkt drop            8832
L2 Src Mac lookup fail       880
Rate limit exceeded 29312474
```

When you configure a rate limit profile on a fabric interface, you can see the configured limit in bytes per second when you run either `vif --list` or `vif --get` *fabric_vif_id*.

```
vif0/2         PCI: 0000: af: 01.1 (Speed 10000, Duplex 1)
               Type: Physical HWaddr: 76:5d: f5: f5: c1:7a
               Vrf:0 Flags: L2Vof QOS:-1 Ref: 8 BUM Rate Limit: 1000000
               RX port    packets:1 errors:0
               RX queue packets:1 errors:0
               RX queue errors to lore 000000000000
               Driver: net_iavf
               Fabric Interface: 0000:af:01.1 Status: UP
               Vlan Mode: Trunk Vlan: 300 500 600
               RX packets:0  bytes:0
errors:1
               TX packets:0 bytes:0 errors:0
               Drops: 1
```

NOTE:

- The rate limit is only configurable on physical interfaces and only during deployment.
- The existing global rate limit configuration *fabricBMCastRateLimit* is deprecated from release 22.4.

APIs and CLI Commands for Bond Interfaces

SUMMARY

Read this topic to learn about the APIs and CLIs available in the L2 mode of the Juniper Cloud-Native Router. JCNR supports an API that can be used to force traffic to switch from the active interface to the standby interface in a bonded pair. Another JCNR API and a CLI can be used to view the active node details in a bond interface.

IN THIS SECTION

- APIs for Bond Interfaces | 38
- CLI Commands for Bond Interfaces | 39

APIs for Bond Interfaces

When you run cloud-native router in L2 mode with cascaded nodes, you can configure those nodes to use bond interfaces. You can configure the bond mode in the `values.yaml` file before deployment. For example:

```
bondInterfaceConfigs:
    - name: "bond0"
      mode: 1             # ACTIVE_BACKUP MODE
      slaveInterfaces:
      - "enp59s0f0v0"
      - "enp59s0f0v1"
```

API to View the Active and Backup Interfaces in a Bond Interface Pair

Starting with JCNR Release 23.3, use the REST API call: `curl -X GET http://127.0.0.1:9091/bond-get-active/bond0` on localhost port 9091 to fetch the active and backup interface details of a bond interface pair.

A sample output is shown below:

```
root@nodep23:~# curl -X GET http://127.0.0.1:9091/bond-get-active/bond0
{"active": "0000:af:01.0", "backup": "0000:af:01.1"}
```

API to Force Bond Link Switchover

Starting with JCNR Release 22.4, you can force traffic switchover from an active to backup interface in a bond interface pair using a REST API. If you have configured the bond interface pair in the `ACTIVE_BACKUP` mode before deploying JCNR, then the vRouter-agent exposes the REST API call: `curl -X POST http://127.0.0.1:9091/bond-switch/bond0` on localhost port 9091. Use this REST API call to force traffic to switch from the active interface to the backup interface.

A sample output is shown below:

```
root@nodep23:~# curl -X GET http://127.0.0.1:9091/bond-get-active/bond0
{"active": "0000:af:01.0", "backup": "0000:af:01.1"}
root@nodep23:~# curl -X POST http://127.0.0.1:9091/bond-switch/bond0
{}
root@nodep23:~# curl -X GET http://127.0.0.1:9091/bond-get-active/bond0
{"active": "0000:af:01.1", "backup": "0000:af:01.0"}
```

CLI Commands for Bond Interfaces

The vRouter contains the following CLI commands which are related to bond interfaces:

- `dpdkinfo -b`—displays the active interface in a bonded pair.

```
[[root@jcnr-01 /]# dpdkinfo -b
No. of bond slaves: 2
Bonding Mode: Active Backup
Transmit Hash Policy: Layer 2 (Ethernet MAC)
MII status: UP
MII Link Speed: 10000 Mbps
Up Delay (ms): 0
Down Delay (ms): 0
Driver: net_bonding

Slave Interface(0): 0000:17:01.0
Slave Interface Driver: net_iavf
Slave Interface (0): Active
Slave Interface Mac : 6E: BD: 45:0F: 4A:02

MII status: UP
MII Link Speed: 10000 Mbps

Slave Interface (1): 0000:17:11.0
Slave Interface Driver: net_iavf
Slave Interface Mac      6E: BD: 45:0F: 4A: C2

MII status: UP
MII Link Speed: 25000 Mbps
```

- `dpdkinfo -n`—displays the traffic statistics associated with your bond interfaces.

```
[root@jcnr-01 /]# dpdkinfo -n2
Master Info (eth_bond_bond0):
RX Device Packets: 72019, Bytes: 96419113, Errors:0, Nombufs:0
Dropped RX Packets: 37475
TX Device Packets:0, Bytes:0, Errors:0
Queue Rx:
Tx:
Rx Bytes:
Tx Bytes:
Errors:

Slave Info (0000:17:01.0):
Rx Device Packets: 72019, Bytes:66073908, Errors:0, Nombufs:0
Dropped RX Packets: 588
TX Device Packets:0, Bytes:0, Errors:0
Queue Rx:
Tx:
Rx Bytes:
Tx Bytes:
Errors:

Slave Info (0000:17:11.0):
RX Device Packets:0, Bytes:30345205, Errors:0, Nombufs:0
Dropped R Packets:36887
TX Device Packets:0, Bytes:0, Errors:0
Queue Rx:
Tx:
Rx Bytes:
Tx Bytes:
Errors:
```

Quality of Service (QoS)

SUMMARY

Read this topic to learn about the quality of service (QoS) feature of the Juniper Cloud-Native Router when deployed in L2 mode.

IN THIS SECTION

- QoS Overview | 41
- Configuration Example | 43
- Troubleshooting | 44

Starting in Juniper Cloud-Native Router Release 22.4, you can configure quality of service (QoS) parameters including classification, marking, and queuing. The cloud-native router performs classification and marking operations in vRouter and queing (scheduling) operations in the physical network interface card (NIC). Scheduling is only supported on the E810 NIC.

QoS Overview

You enable QoS prior to the deploy time by editing the `values.yaml` file in **Juniper-Cloud-Native-Router-*version-number*/helmchart** directory and changing the `qosEnable` value to `true`. The default value for the QoS feature is `false` (disabled). For example:

```
# Set true/false to Enable or Disable QOS, note: QOS is not supported on X710 NIC.
  qosEnable: true
```

NOTE: You can only enable the QoS feature if the host server on which you install your cloud-native router contains an Intel E810 NIC that is running lldp.

You enable lldp on the NIC using the `lldptool` which runs on the host server as a CLI application. Issue the following command to enable lldp on the E810 NIC. For example, you could use the following command:

```
lldptool -T -i INTERFACE -V ETS-CFG willing=no
tsa=0:strict,1:strict,2:strict,3:strict,4:strict,
5:strict,6:strict,7:strict
up2tc=0:0,1:1,2:2,3:3,4:0,5:1,6:2,7:3
```

The details of the above command are:

- **ETS**–Enhanced Transmission Selection
- **willing**–The willing attribute determines whether the system uses locally configured packet forwarding classification (PFC) or not. If you set `willing` to `no`(the default setting), the cloud-native router applies local PFC configuration. If you set willing to yes, and the cloud-native router receives TLV from the peer router, the cloud-native router applies the received values.
- **tsa**–The transmission selection algorithm is a comma seperated list of traffic class to selection algorithm maps. You can choose `ets`, `strict`, or `vendor` as selection algorithms.
- **up2tc**–Comma-separated list that maps user priorities to traffic classes

The list below provides an overview of the classification, marking, and queueing operations performed by cloud-native router.

- Classification:
  - vRouter classifies packets by examining the priority bits in the packet
  - vRouter derives traffic class and loss priority
  - vRouter can apply traffic classifiers to fabric, traffic, and workload interface types
  - vRouter maintains 16 entries in its classifier map
- Marking (Re-write):
  - vRouter performs marking operations
  - vRouter performs rewriting of p-bits in the egress path
  - vRouter derives new traffic priority based on traffic class and drop priority at egress
  - vRouter can apply marking to packets only on fabric interfaces
  - vRouter maintains 8 entries in its marking map
- Queueing (Scheduling):
  - Cloud-native router performs strict priority scheduling in hardware (E810 NIC)
  - Cloud-native router maps each traffic class to one queue
  - Cloud-native router limits the maximum number of traffic queue to 4

- Cloud-native router maps 8 possible priorities to 4 traffic classes; It also maps each traffic class 1 hardware queue
- Cloud-native router can apply scheduling to fabric interface only
- Virtual functions (VFs) leverage the queues that you configure in the physical functions (interfaces)
- vRouter maintains 8 entries in its scheduler map

Configuration Example

You configure QoS classifiers, rewrite rules, and schedulers in the controller using Junos set commands or remotely using NETCONF. We display a Junos-based example configuration below:

```
set class-of-service classifiers ieee-802.1 class1 forwarding-class assured-forwarding loss-priority high code-points 011
set class-of-service rewrite-rules ieee-802.1 Rule_1 forwarding-class assured-forwarding loss-priority high code-point 110
set class-of-service schedulers sch1 priority high
set class-of-service scheduler-maps sch1 forwarding-class assured-forwarding scheduler sch1
set class-of-service interfaces enp175s1 scheduler-map sch1
set class-of-service interfaces enp175s1 unit 0 rewrite-rules ieee-802.1 Rule_1
set class-of-service interfaces vhostnet123-3546aefd-7af8-4fe5 unit 0 classifiers ieee-802.1 class1
```

You view the QoS configuration by "accessing the JCNR controller CLI" on page 167. Use the `show` commands in Junos operation mode. The show commands reveal the configuration of classifiers, rewrite rules, or scheduler maps individually. For example:

Show Classifier

```
user@jcnr-01> show class-of-service classifier

Classifier: class1, Code point type: ieee802.1p
Code point               Forwarding class              Loss priority
011                      assured-forwarding            high
```

Show Rewrite-Rule

```
user@jcnr-01> show class-of-service rewrite-rule

Rewrite rule: Rule_1, Code point type: ieee802.1p
Forwarding class            Loss priority       Code point
assured-forwarding          high                110
```

Show Scheduler-Map

```
user@jcnr-01> show class-of-service scheduler-map sch1
Scheduler map: sch1
  Scheduler: sch1,  Forwarding class: assured-forwarding
    Transmit rate: unspecified, Rate Limit: none, Priority: high
```

Show Interface

```
user@jcnr-01> show class-of-service interface vhostnet123-5a1e3079-d45e-4ab5
Physical interface: vhostnet123-5a1e3079-d45e-4ab5
Maximum usable queues: 4, Queues in use: 4

  Logical interface: vhostnet123-5a1e3079-d45e-4ab5.0
Object            Name              Type
Classifier        class1            ieee802.1p
```

```
user@jcnr-01> show class-of-service interface enp175s1
Physical interface: enp175s1
Maximum usable queues: 4, Queues in use: 4
  Scheduler map: sch1
  Logical interface: enp175s1.0
Object            Name              Type
Rewrite-Output    Rule_1            ieee802.1p
```

Troubleshooting

You can troubleshooting the QoS configuration "by accessing the vRouter CLI" on page 168. Use the `purel2cli` command and by viewing the interface mapping.

Display Classifier Config

```
purel2cli --qos cla class1
Classifer name: class1 Classifier Index: 0
===============================================
code-points   loss priority    forwarding-class
===============================================
   000          low            best-effort
   001          low            best-effort
   010          low            best-effort
   011          high           assured-forwarding
  100          low            best-effort
   101          low            best-effort
   110          low            best-effort
   111          low            best-effort
```

```
vif0/2      PMD: vhostnet123-3546aefd-7af8-4fe5
            Type:Virtual HWaddr:aa:bb:cc:dd:ee:12
            Vrf:0 Flags:L2Mon QOS:-1 Ref:13
            RX port   packets:20 errors:0
            RX queue  packets:20 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Vlan Mode: Trunk  Vlan: 100 200 300
            Qos classifier: class1
           RX packets:20  bytes:1200 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:40
```

Display Re-write Config

```
purel2cli --qos rw Rule_1
Re-Write name: Rule_1 Re-write Index: 0
=========================================loss priority      Forwarding-class      re-write prio
=========================================
   low          best-effort                  n/a
   low          expedited-forwarding         n/a
   low          assured-forwarding           n/a
   low          network-control              n/a
   high         best-effort                  n/a
   high         expedited-forwarding         n/a
   high          assured-forwarding            110
   high          network-control               n/a
```

```
vif0/1      PCI: 0000:af:01.0 (Speed 10000, Duplex 1)
            Type:Physical HWaddr:46:d5:f3:fc:fc:92
            Vrf:0 Flags:L2Vof QOS:-1 Ref:42
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: 0000:af:01.0  Status: UP  Driver: net_iavf
            Vlan Mode: Trunk  Vlan: 100 200 300 700-705 2001
                Rewrite:    Rule_1
               Scheduler:  sch1
            RX packets:0  bytes:0 errors:0
            TX packets:20  bytes:1200 errors:0
            Drops:0
            TX port   packets:20 errors:0
```

Display Scheduler Output

```
purel2cli --qos sch sch1
Scheduler name: sch1 Scheduler  Index: 0
=====================================
 forwarding-class      priority_map
=====================================
  best-effort                         0
  expedited-forwarding      0
  assured-forwarding          2
  network-control               0
```

```
vif0/1      PCI: 0000:af:01.0 (Speed 10000, Duplex 1)
            Type:Physical HWaddr:46:d5:f3:fc:fc:92
            Vrf:0 Flags:L2Vof QOS:-1 Ref:42
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: 0000:af:01.0  Status: UP  Driver: net_iavf
            Vlan Mode: Trunk  Vlan: 100 200 300 700-705 2001
               Rewrite:    Rule_1
               Scheduler:  sch1
            RX packets:0  bytes:0 errors:0
            TX packets:20  bytes:1200 errors:0
            Drops:0
            TX port   packets:20 errors:0
```

Native VLAN

IN THIS SECTION

- Native VLAN | 47

Starting in Juniper Cloud-Native Router Release 23.1, JCNR supports receiving and forwarding untagged packets on a trunk interface. Typically, trunk ports accept only tagged packets, and the untagged packets are dropped. You can enable a JCNR fabric trunk port to accept untagged packets by configuring a native VLAN identifier (ID) on the interface on which you want the untagged packets to be received. When a JCNR fabric trunk port is enabled to accept untagged packets, such packets are forwarded in the native VLAN domain.

Native VLAN

Enable the `native-vlan-id` key in the Helm chart, at the time of deployment, to configure the VLAN identifier and associate it with untagged data packets received on the fabric trunk interface. Edit the `values.yaml` file in **Juniper_Cloud_Native_Router_*<release-number>*/helmchart** directory and add the key `native-vlan-id` along with a value for it. For example:

```
fabricInterface:
  - eth1:
      ddp: on
      interface_mode: trunk
      vlan-id-list: [100, 200, 300, 700-705]
      storm-control-profile: rate_limit_pf1
      native-vlan-id: 100
      no-local-switching: true
```

NOTE: After editing the **values.yaml** file, you have to install or upgrade JCNR using the edited **values.yaml** to ensure that the `native-vlan-id` key is enabled.

To verify, if native VLAN is enabled for an interface, connect to the vRouter agent by executing the command `kubectl exec -it -n contrail contrail-vrouter-`*`<agent container>`*` -- bash` command, and then run the command `vif --get `*`<interface index id>`*. A sample output is shown below:

```
vif0/1      PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
            Type:Physical HWaddr:6a:45:b2:a8:ce:5c
            Vrf:0 Flags:L2Vof QOS:-1 Ref:11
            RX port   packets:36550 errors:0
            RX queue  packets:36550 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: eth_bond_bond0  Status: UP  Driver: net_bonding
            Slave Interface(0): 0000:3b:02.0  Status: UP  Driver: net_iavf
            Vlan Mode: Trunk  Vlan: 100 200 300
            Native vlan id: 100
            RX packets:36550  bytes:5875795 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:613
```

Prevent Local Switching

IN THIS SECTION

- Configuration Example | 49

Starting in Juniper Cloud-Native Router Release 23.1, JCNR provides support to prevent interfaces in a bridge domain that are a part of the same VLAN group, from transmitting ethernet frame copies in between those interfaces. The **noLocalSwitching** key provides the option to enable the functionality on the selected VLAN IDs.

To prevent interfaces in a bridge domain from transmitting and receiving ethernet frame copies, enable the **noLocalSwitching** key and assign a VLAN ID to it to ensure that the interfaces belonging to the VLAN ID do not transmit frames to one another. Note that the **noLocalSwitching** functionality is enabled only on the access interfaces. To enable **noLocalSwitching** on a trunk interface that is a part of the same VLAN ID, you have to separately enable the trunk interface by setting the **no-local-switching** key in the trunk interface to **true**. Use the **noLocalSwitching** functionality when you want to block interfaces that are a part of a VLAN group to stop transmitting traffic directly to one another.

NOTE:
no-local-switching

Configuration Example

To prevent local switching, perform the steps below prior to the deploy time:

1. Edit the **values.yaml** file in **Juniper_Cloud_Native_Router_*<release-number>*/helmchart** directory.

2. Enable the **noLocalSwitching** key and provide the VLAN IDs.

```
noLocalSwitching: [700]
```

NOTE:

a. The value for the **noLocalSwitching** key can be an indivdual VLAN ID, or multipe comma-separated VLAN ID values, or a VLAN ID range, or a combination of comma-separated VLAN ID values and a VLAN ID range. For example, **noLocalSwitching: [700, 701, 705-710]**.

b. With this step the feature is enabled for all access interfaces having the specified VLAN ID. You can skip the next step if you do not want to enable the feature on the trunk interface.

3. To enable the feature on a trunk interface, add the key **no-local-switching** and set it to **true** under the trunk interface configuration.

. For example:

```
fabricInterface:
  - bond0:
      ddp: on
      interface_mode: trunk
      vlan-id-list: [100, 200, 300, 700-705]
      storm-control-profile: rate_limit_pf1
      #native-vlan-id: 100
      no-local-switching: true
```

4. Install or upgrade JCNR using the **values.yaml**.

Verify Configuration

To verify the configuration, you can use the `purel2cli` utility available on the vRouter. View the "Access vRouter CLI" on page 168 topic to access the vRouter shell. You can run the `purel2cli` commands from the vRouter CLI. For example:

1. Run the command `purel2cli --nolocal show` to know all the interfaces that are enabled for **noLocalSwitching** functionality on all the VLANs. A sample output is shown below:

```
[root@jcnr-01 /]# purel2cli --nolocal show
============================
vlan    no_local_switch_list
============================
100     1, 2, 4,
200
300
700
701
702
703
```

2. Run the command `purel2cli --nolocal get` *<VLAN ID>* to check if **noLocalSwitching** functionality is enabled on a specific VLAN ID. A sample output is shown below:

```
[root@jcnr-01 /]# purel2cli --nolocal get 100
============================
vlan    no_local_switch_list
```

```
============================
100     1, 2, 4,
```

4

CHAPTER

L3 Features

L3 Features Overview | 53

IPsec Security Services | 53

JCNR as a Transit Gateway | 54

EVPN Type 5 Routing over VXLAN Tunnels | 56

Integrated Routing and Bridging on JCNR | 64

L3 Routing Protocols | 71

MPLS Support | 75

Bidirectional Forwarding Detection (BFD) | 76

Virtual Router Redundancy Protocol (VRRP) | 77

Virtual Routing Instance (VRF-Lite) | 78

ECMP | 81

BGP Unnumbered | 82

L3 Features Overview

SUMMARY

Read this topic to learn about the features available in the Juniper Cloud-Native Router when deployed in L3 (router) mode.

The Juniper Cloud-Native Router supports multiple "deployment modes" on page 10.

In L3 mode, the cloud-native router behaves like a router and so performs routing functions and runs routing protocols such as ISIS, BGP, OSPF, and segment routing-MPLS. In L3 mode, the pod network is divided into an IPv6 underlay network and an IPv4 or IPv6 overlay network. The IPv6 underlay network is used for control plane traffic.

This chapter provides information about the various L3 features supported by JCNR.

IPsec Security Services

IN THIS SECTION

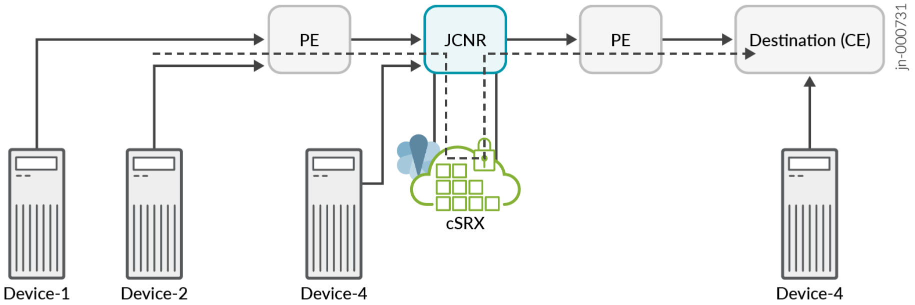
- Overview | 54

Read this topic to understand how the cloud-native router integrates with Juniper's cSRX to provide IPsec security services.

Juniper Cloud-Native Router (JCNR) offers containerized routing functionality for both cloud-based and on-premise 5G environments. There is a growing demand for integrating security services with JCNR. This functionality can be achieved using host-based service chaining. Starting Release 23.4, the cloud-native router is integrated with Juniper's containerized SRX (cSRX) platform to provide security services such as IPsec.

Overview

Let us consider an IPsec security services use case with JCNR. In the figure below, the cloud-native router connects the provider edge (PE) routers in a service provider network. The customer edge (CE) routers or devices in the source network securely transfer data to the destination CEs via an IPsec tunnel. In the given scenario, the IPsec tunnel initiates from the cloud-native router's security services (cSRX) and terminates on the destination CEs. The cloud-native router and its peer PE provides the underlay connectivity to the IPsec tunnel.



The cloud-native router is chained with a security service instance (cSRX) in the same Kubernetes cluster. The cSRX instance runs as a pod service in L3 mode. Please review the *Deploying cSRX with JCNR* topic for details on how to deploy cSRX for service chaining with JCNR.

NOTE: A cloud-native router instance is service chained with only one instance of cSRX and therefore supports only one IPsec tunnel.

RELATED DOCUMENTATION

IPsec Overview

JCNR as a Transit Gateway

JCNR can act as a transit gateway for external traffic. As a transit gateway, JCNR is neither the source nor the destination for the traffic, but an intermediate hop. It acts as a vanilla router to switch traffic between multiple physical interfaces.

Starting with Juniper Cloud-Native Router (JCNR) Release 23.2, JCNR can now act as a transit gateway for external traffic. As a transit gateway, JCNR is neither the source nor the destination for the traffic, but an intermediate hop. It acts as a vanilla router to switch traffic between multiple physical interfaces. Depending on the forwarding state, JCNR can encapsulate or decapsulate the traffic between interfaces.

NOTE: Starting with JCNR Release 23.2, JCNR supports multiple fabric interfaces that enable it to function as a transit gateway.

JCNR has to be deployed in the L3 mode to perform the transit router functionality. Add all physical interfaces (physical and virtual functions) as fabric interfaces in the helm chart before deploying the JCNR. The deployed JCNR does not support editing or changing the fabric interfaces during run time. However, you can create or remove pod interfaces during run time. Here are example helm chart configurations:

```
fabricInterface:
  - ens2f2:
      ddp: "auto"
  - ens1f1:
      ddp: "auto"
```

```
 fabricInterface:
    - subnet: 10.0.3.0/24
      gateway: 10.0.3.1
      ddp: "off"
    - subnet: 10.0.5.0/24
      gateway: 10.0.5.1
      ddp: "off"
```

You need to configure an IP address on the loopback interface and use it as a tunnel endpoint for each JCNR instance. The loopback IP address is the next hop address which BGP advertises to its peers. All data packets with encapsulations like MPLSoUDP will have the outer IP address as this loopback IP address. The loopback IP address is reachable via any of the physical interfaces. The loopback IP address should be in a /32 subnet without a MAC address. For example:

```
set interfaces lo1 unit 1 family inet address 10.0.0.1/32
```

EVPN Type 5 Routing over VXLAN Tunnels

IN THIS SECTION

- Enabling EVPN Type 5 Routing over VXLAN Tunnels | 57
- Configuration Example and CLI Commands for EVPN Type 5 Routing over VXLAN Setup | 58

Ethernet Virtual Private Network (EVPN) with Virtual Extensible LAN (VXLAN) Type 5 routing is designed for use in data center and cloud environments to provide efficient and scalable network connectivity for virtualized workloads. It combines the benefits of EVPN and VXLAN to enable flexible and seamless communication between virtual machines (VMs) and physical devices across different IP subnets and locations. Starting with Juniper Cloud-Native Router (JCNR) Release 23.3, JCNR supports EVPN Type 5 Routing over VXLAN tunnels.

Ethernet Virtual Private Network (EVPN) technology provides a scalable and efficient way to extend Layer 2 and Layer 3 connectivity across multiple sites. EVPN uses Border Gateway Protocol (BGP) to exchange information between Provider Edge (PE) routers, allowing them to learn the location of Ethernet segments and IP prefixes. This allows for the creation of virtual networks that can span multiple sites, while providing traffic separation and isolation through the use of virtual routing and forwarding (VRF) instances. EVPN supports several encapsulation methods, including VXLAN and MPLS, which can be used to transport traffic across the service provider network.

VXLAN is a network overlay technology that allows the creation of virtual Layer 2 networks on top of an existing Layer 3 network infrastructure. It extends the reach of Layer 2 segments beyond the confines of a single physical network, which is especially useful in large-scale virtualized environments.

EVPN supports two types of routes: MAC Advertisement Route (Type 2) and IP Prefix Route (Type 5). Type 2 routes are used to exchange MAC addresses and VLANs between PE routers, while Type 5 routes are used to exchange Layer 3 network routes. In EVPN VXLAN, Type 5 routes are used to advertise IP prefixes and their associated MAC addresses. To reach a tenant using connectivity provided by the EVPN VXLAN Type 5 IP prefix route, data packets are sent as Layer 2 Ethernet frames encapsulated in the VXLAN header over the IP network across the data centers.

EVPN VXLAN Type 5 routing allows for efficient distribution of MAC and IP routing information, enabling large-scale networks with numerous virtualized workloads to operate seamlessly. The technology supports secure isolation of tenant traffic in shared environments, providing a virtual network overlay that maintains separation between tenants.

To learn more about EVPN VXLAN Type 5 routing, see *Understanding EVPN Pure Type-5 Routes*.

NOTE: Transit router functionality should be enabled for JCNR to support EVPN VXLAN Type 5 routing. See, "JCNR as a Transit Gateway" on page 54.

Enabling EVPN Type 5 Routing over VXLAN Tunnels

Enable EVPN Type 5 routing over VXLAN tunnels using custom JCNR controller configuration via the go template. Apply the custom configuration before installing JCNR, or for an existing JCNR installation, delete the cRPD pod and respawn.

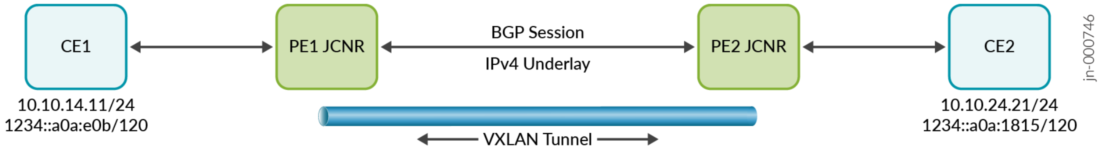
Use the following sample to configure EVPN Type 5 routing over VXLAN tunnels in JCNR using the **jcnr-cni-custom-config-cm.tmpl** file located in **Juniper_Cloud_Native_Router_<release-number>/ cRPD_examples** directory.

```
groups {
    custom {
        routing-instances {
            EVPN-TYPE5-VXLAN-VRF {
                instance-type vrf;
                protocols {
                    evpn {
                        ip-prefix-routes {
                            advertise direct-nexthop;
                            encapsulation vxlan;
                            vni 1000;
                             export EVPN-TYPE5-VXLAN-VRF-EXPORT-POLICY;
                        }
                    }
                }
                interface ge-0/0/1.0;
                route-distinguisher 10.255.0.1:100;
                vrf-target target:100:100;
            }
        }
}
```

To learn more about node annotations and custom configuration, see *Customize JCNR Configuration* .

To learn about EVPN Type 5 configuration in Junos, see *Example: Configuring EVPN with Support for Virtual Switch*.

Configuration Example and CLI Commands for EVPN Type 5 Routing over VXLAN Setup



The topology shown above describes a simple setup with two JCNRs deployed as provider edge routers PE1 and PE2. The CE1 and CE2 represent hosts behind each of the PEs. As a pre-requisite, a BGP session must exist between PE1 and PE2. Consider the following EVPN-VXLAN configuration on PE1, with the interface enp4s0 towards CE1:

```
groups {
    custom {
        routing-instances {
            orange {
                instance-type vrf;
                routing-options {
                    rib orange.inet6.0 {
                        multipath;
                    }
                    multipath;
                }
                protocols {
                    evpn {
                        ip-prefix-routes {
                            advertise direct-nexthop;
                            encapsulation vxlan;
                            vni 10010;
                        }
                    }
                }
                interface enp4s0;
                route-distinguisher 1.1.1.1:4;
                vrf-target target:4:4;
            }
      }
}
```

A VXLAN tunnel is created between routers PE1 and PE2. The 10.10.14.0/24 network routes are locally learnt on PE1 and are advertised via EVPN Type 5 to the remote PE. Similarly, the 10.10.24.0/24 network routes are locally learnt on PE2 and advertised via EVPN Type 5 to the remote PE. All traffic between CE1 and CE2 is forwarded between PE1 and PE2 over the VXLAN tunnel.

Use the commands listed in the sections below to troubleshoot a EVPN VXLAN Type 5 routing setup.

cRPD CLI Commands

The following CLI commands can be executed on the cRPD CLI. To access the cRPD CLI, see "Access cRPD CLI" on page 167.

- `show bgp <summary | neighbor>`: Provides a summary of the EVPN connection to the peer and the status of the connection.

  A sample output is shown below:

  ```
  host@pe1> show bgp summary
  Threading mode: BGP I/O
  Default eBGP mode: advertise - accept, receive - accept
  Groups: 1 Peers: 2 Down peers: 1
  Table                Tot Paths  Act Paths  Suppressed  History Damp  State     Pending
  bgp. evpn. 0         2          2          0           0             0         0

  Peer                           AS     InPkt    OutPkt    OutQ  Flaps   Last   Up/Dwn
  State|#Active/Received/Accepted/Damped…
  2.2.2.2                     4      10345    10336     0      2      3d     5:32:50
  Establ
  bgp.evpn.0: 2/2/2/0
  orange.evpn.0: 2/2/2/0
  3.3.3.3                        4      0       0          0      0      4w4d   13:28:22
  Connect
  ```

- `show route <summary | table | prefix>`: Displays the active entries in the routing tables.

- `show evpn instance`: Displays information about the EVPN routing instance.

- `show evpn l3-context`: Displays the configured L3 context on the local box.

A sample output is shown below:

```
host@pe1> show evpn l3-context
L3 context                      Type  Adv      Encap  VNI/Label  Router MAC/GW intf
orange                          Cfg   Direct   VXLAN  10010      48:5a:0d:78:78:d7
```

- `show evpn ip-prefix-database`: Provides a list of exported and imported EVPN route prefixes and the status of these routes.

  A sample output is shown below:

```
root@evpn-pe1-node> show evpn ip-prefix-database
L3 context: orange

IPv4->EVPN Exported Prefixes
Prefix                                       EVPN route status
2.55.1.0/24                                  Created
4.1.1.4/30                                   Created
10.10.14.0/24                                Created

IPv6->EVPN Exported Prefixes
Prefix                                       EVPN route status
1234::a0a:e00/120                            Created
abcd::401:104/126                            Created
abcd::2:55:1:0/120                           Created

EVPN->IPv4 Imported Prefixes
Prefix                                       Etag
2.55.2.0/24                                  0
  Route distinguisher    VNI/Label  Router MAC         Nexthop/Overlay GW/ESI   Route-Status
Reject-Reason
  2.2.2.2:4              10020      48:5a:0d:49:fc:63  2.2.2.2
Accepted      n/a
10.10.24.0/24                                0
  Route distinguisher    VNI/Label  Router MAC         Nexthop/Overlay GW/ESI   Route-Status
Reject-Reason
  2.2.2.2:4              10020      48:5a:0d:49:fc:63  2.2.2.2
Accepted      n/a

EVPN->IPv6 Imported Prefixes
Prefix                                       Etag
```

```
1234::a0a:1800/120                           0
  Route distinguisher    VNI/Label  Router MAC         Nexthop/Overlay GW/ESI   Route-Status
Reject-Reason
  2.2.2.2:4              10020      48:5a:0d:49:fc:63  2.2.2.2
Accepted      n/a
abcd::2:55:2:0/120                           0
  Route distinguisher    VNI/Label  Router MAC         Nexthop/Overlay GW/ESI   Route-Status
Reject-Reason
  2.2.2.2:4              10020      48:5a:0d:49:fc:63  2.2.2.2
Accepted      n/a
```

- `show route table <VRF>.evpn.0`: Displays the route entries in the specified routing table.

  A sample output is shown below.

```
host@pe1> show route table orange. evpn. 0

orange.evpn.0: 4 destinations, 0 routes (4 active, 0 holddown, 0 hidden)
+ = Active Route,  - = Last Active,  * = Both

5:1.1.1.1:4::0::10.10.14.0::24/248
                            *[EVPN/170] 4W4d 13:29:25
                                Fictitious
5:2.2.2.2:4::0::10.10.24.0::24/248
                              *[BGP/170] 3d 05:33:52, localpref 100, from 2.2.2.2
                                    AS path: I, validation-state: unverified
                                     to 10.10.1.20 via enp2s0
5:1.1.1.1:4::0::1234::00a:000::120/248
                              *[EVPN/170] 4w4d 13:29:25
                                    Fictitious
5:2.2.2.2:4::0::1234::a0a:1800::120/248
                              *[BGP/170] 3d 05:33:52, localpref 100, from 2.2.2.2
                                    AS path: I, validation- state: unverified
                                     to 10.10.1.20 via enp2s0
```

- `show route table <VRF>.inet.0`: Displays the route entries in the specified routing table.
- `show route table bgp.evpn.0`: Displays the route entries in the specified routing table.

A sample output with a local prefix is shown below.

```
host@pe1> show route table bgp.evpn.0 match-prefix 5:1.1.1.1:4::0::10.10.14.0::24

bgp.evpn.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both
5:1.1.1.1:4::0::10.10.14.0::24/248
                   *[EVPN/170] 2w1d 05:11:43
                       Fictitious
```

A sample output with a remote prefix is shown below.

```
host@pe1> show route table bgp.evpn.0 match-prefix 5:2.2.2.2:4::0::10.10.24.0::24
bgp.evpn.0: 10 destinations, 10 routes (10 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both
5:2.2.2.2:4::0::10.10.24.0::24/248
                   *[BGP/170] 2w1d 05:11:48, localpref 100, from 2.2.2.2
                      AS path: I, validation-state: unverified
                    >  to 10.10.1.20 via enp2s0
```

- `show krt next-hop`: Displays the configured next hop.

vRouter CLI Commands

The following CLI commands can be executed on the vRouter CLI. To access the vRouter CLI, see "Access vRouter CLI" on page 168.

- `rt --get <prefix> --vrf <vrf-id> --family <inet4/inet6>`: Provides the route which is pointing to the specified IPv4 address.

  A sample output is shown below.

```
[host@pe1 /]# rt --get 10.10.24.0/24 --vrf 1
Match 10.10.24.0/24 in vRouter inet4 table 0/1/unicast
Flags: L=Label Valid, P=Proxy ARP, T=Trap ARP, F=Flood ARP, Ml=MAC-IP learnt route
vRouter inet4 routing table 0/1/unicast
Destination           PPL        Flags        Label         Nexthop    Stitched MAC(Index)
10.10.24.0/24           0          LPT      10020             30        -
```

- `vxlan --dump`: Provides information regarding the VNIs that are configured and the next hop.

A sample output is shown below.

```
[host@pe1 /]# vxlan --dump
VXLAN Table
VNID    NextHop
----------------
  10010    25
```

- `nh --get <nh-id>`: Provides the next hop details.

  A sample output is shown below.

```
[root@evpn-pe1-node /]# nh --get 30
Id:30         Type:Tunnel         Fmly: AF_INET  Rid:0  Ref_cnt:5          Vrf:0
              Flags:Valid, Policy, Vxlan, Etree Root, l3_vxlan,
              Oif:1 Len:14 Data:52 54 00 78 c8 f2 52 54 00 ee 83 cd 08 00 Sip:1.1.1.1
Dip:2.2.2.2
              L3_Vxlan_SMac:48:5a:0d:78:78:d7 L3_Vxlan_DMac:48:5a:0d:49:fc:63
```

- `vif --list`: Provides a list of enterprises configured with the `vif`.
- `flow --l`: Displays all the active flows in the system.

  Use this command to verify the traffic flowing between CE1 and CE2 on the vRouter. A sample output is shown below.

```
[host@pe1 /]# flow -l
Flow table(size 161218560, entries 629760)

Entries: Created 11 Added 11 Deleted 20 Changed 26Processed 11 Used Overflow entries 0
(Created FlOwS/CPU: 0 0 0 0 0 0 0 0 0 0 11 0 (oflows 0)

Action: F-Forward, D=Drop N=NAT(S-SNAT, D=DNAT, PS=SPAT, Pd=DPAT, L=Link Local Port)
 Other: K(nh)=Key Nexthop, S(nh)=RPF Nexthop
 Flags: E-Evicted, Ec-Evict Candidate, N=New Flow, M-Modified Dm=Delete Marked
TCP(r=reverse): S-SYN, F=FIN, R=RST, C-HalfClose, E-Established, D=Dead
 Stats: Packets/Bytes

Index                   Source: Port/Destination: Port                       Proto(V)
------------------------------------------------------------------------------------
95644<=>443840          10.10.24.21:30                                       1 (1)
```

```
                        10.10.14.11:0
 (Gen: 1, K(nh): 8, Action:F, Flags:, 005: -1, S(nh):30, Stats: 16/1344,
  SPort 56932, TTL 0. Sinfo 2.2.2.2)

443840<=>95644          10.10.14.11:30                                       1 (1)
                        10.10.24.21:0
 (Gen: 1, K(nh):8, Action:F, Flags:, Q0S: -1, S(nh):41, Stats: 16/1344,
  SPort 53983, TTL 0, Sinfo 0.0.0.0)
```

- `vifdump <vif-number>`: Displays all the packet details for the specified `vif`.

  A sample output is shown below.

```
[host@pe1 /]# vifdump 3 -nevv
vif0/3      PCI: 0000:04:00.0 NH: 8 MTU: 9000
dropped privs to tcpdump
tcpdump: listening on mon3, link-type EN10MB (Ethernet), snapshot length 262144 bytes
20:15:15.611827 52:54:00:2c:f6:16 > 52:54:00:ef:3c:4d, ethertype IPv4 (0x0800), length 98:
(tos 0x0, ttl 64, id 1764, offset 0, flags [DF], proto ICMP (1), length 84)
    10.10.14.11 > 10.10.24.21: ICMP echo request, id 16, seq 25, length 64
20:15:15.612472 52:54:00:ef:3c:4d > 52:54:00:2c:f6:16, ethertype IPv4 (0x0800), length 98:
(tos 0x0, ttl 62, id 14142, offset 0, flags [none], proto ICMP (1), length 84)
    10.10.24.21 > 10.10.14.11: ICMP echo reply, id 16, seq 25, length 64
20:15:16.626773 52:54:00:2c:f6:16 > 52:54:00:ef:3c:4d, ethertype IPv4 (0x0800), length 98:
(tos 0x0, ttl 64, id 1863, offset 0, flags [DF], proto ICMP (1), length 84)
    10.10.14.11 > 10.10.24.21: ICMP echo request, id 16, seq 26, length 64
20:15:16.627404 52:54:00:ef:3c:4d > 52:54:00:2c:f6:16, ethertype IPv4 (0x0800), length 98:
(tos 0x0, ttl 62, id 14187, offset 0, flags [none], proto ICMP (1), length 84)
    10.10.24.21 > 10.10.14.11: ICMP echo reply, id 16, seq 26, length 64
```

Integrated Routing and Bridging on JCNR

IN THIS SECTION

- Configuring IRB | **65**
- Troubleshooting IRB | **68**

Integrated Routing and Bridging (IRB) is a networking concept that combines the functionalities of routing and bridging within a single network infrastructure. This integration allows for seamless communication between devices on different network segments or subnets.

In a router, packets are forwarded based on their destination IP addresses. Routers operate at Layer 3 (Network Layer) of the OSI model and make decisions about the best path for a packet to reach its destination. In a bridge, frames are forwarded based on MAC addresses. Bridges operate at Layer 2 (Data Link Layer) and use MAC addresses to determine the appropriate segment for a frame.

IRB combines the features of routing and bridging in a single device, typically a router. This allows the device to make forwarding decisions based on both IP addresses and MAC addresses. IRB is particularly useful when you want to enable communication between devices on different subnets in a network. It allows the router to route traffic between subnets based on IP addresses. Instead of having separate routers and bridges, IRB simplifies network design by consolidating these functions into a single device. In VLAN environments, each VLAN can be considered a separate subnet, and the router with IRB capability can route traffic between these VLANs.

Starting with Juniper Cloud-Native Router (JCNR) Release 23.4, JCNR supports IRB, using which you can configure both routing and bridging settings in a unified manner. You can configure IRB interfaces and connect Bridge Domains (BD's) to perform routing between bridge domains.

To learn more about IRB, see *Integrated Routing and Bridging*.

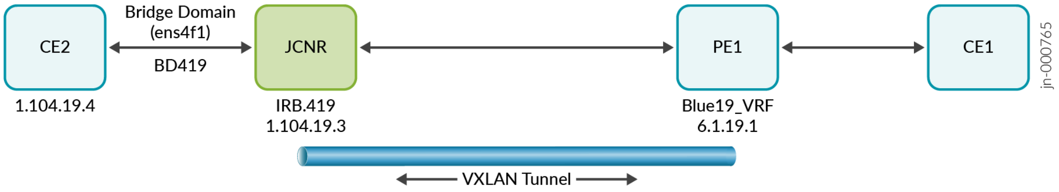
NOTE:

- Configurable MAC address on IRB is not supported in JCNR Release 23.4
- MTU is not configurable on IRB
- BGP unnumbered is not supported on IRB interfaces

Configuring IRB

A pair of IRB interfaces are created for each BD, one for host connectivity (i.e., tap IRB interface) and another for forwarding traffic on fabric (i.e., fabric IRB interface). A single tap interface is created per L2 instance and all tap IRB interfaces that are configured in that L2 instance are created as sub-interfaces on that tap interface.

Consider the following topology shown below and configure IRB on JCNR.



Configuring an IRB interface

Configure an IRB interface as shown in the example below using the `jcnr-cni-custom-config-cm.tmpl` file located in **Juniper_Cloud_Native_Router_<release-number>/cRPD_examples** directory.

```
interfaces {
    irb {
        unit 419 {
            family inet {
                address 1.104.19.3/24;
            }
            family inet6 {
                address 2419::3/64;
            }
        }
    }
    ens4f1 {
        unit 0 {
            family bridge {
                interface-mode trunk;
                vlan-id-list [ 100 200 400 414-423 500 ];
            }
        }
    }
}
```

Attaching an IRB interface as an L3-routing interface to a Bridge

Attach an IRB interface as an L3-routing interface to a Bridge using the example below.

```
routing-instances {
    vswitch {
        instance-type virtual-switch;
        bridge-domains {
            bd419 {
                vlan-id 419;
                routing-interface irb.419;
            }
        }
        interface ens4f1;
    }
}
```

Attaching an IRB interface to VRF

An IRB interface can be a part of VRF-0 or VRF-N. The example shown below demonstrates how you can attach IRB.419 to a VRF Blue19.

```
routing-instances {
    blue19 {
        instance-type vrf;
        protocols {
            bgp {
                group ce_pe_19_v4 {
                    type external;
                    local-address 1.104.19.3;
                    peer-as 1002;
                    local-as 64512;
                    bfd-liveness-detection {
                        minimum-interval 300;
                    }
                    neighbor 1.104.19.4;
                }
                group ce_pe_19_v6 {
                    type external;
                    local-address 2419::3;
                    peer-as 1002;
                    local-as 64512;
                    bfd-liveness-detection {
                        minimum-interval 300;
                    }
                    neighbor 2419::4;
                }
            }
            evpn {
                ip-prefix-routes {
                    advertise direct-nexthop;
                    encapsulation vxlan;
                    vni 2019;
                    export vrf_route_19;
                }
            }
        }
        interface irb.419;
        interface lo0.19;
        route-distinguisher 100.100.100.1:2019;
        vrf-target target:20:2019;
    }
```

Troubleshooting IRB

Use the commands listed in the sections below to troubleshoot an IRB setup.

cRPD CLI Commands

The following CLI commands can be executed on the cRPD CLI. To access the cRPD CLI, see "Access cRPD CLI" on page 167.

- `run show bridge mac-table vlan-id <id>`: Provides the Bridge MAC table details.

```
root@jcnr# run show bridge mac-table vlan-id 419

MAC flags         (S - Static MAC, D - Dynamic MAC)
Routing Instance : default-domain:contrail:ip-fabric:default
Bridging domain VLAN id : 419
MAC                  MAC                  Logical
address              flags                interface

02:22:ec:ac:6b:24      D                   irb.419
e4:5d:37:2b:2a:aa      D                   ens4f1
```

- `run show bgp summary`: Provides a summary of the BGP session running on the IRB.

```
root@jcnr# run show bgp summary

Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#Active/
Received/Accepted/Damped...
1.104.19.4             1002        284        280       0       0     2:11:02 Establ
  blue19.inet.0: 9/9/9/0
2419::4                1002        283        280       0       0     2:10:58 Establ
  blue19.inet6.0: 9/9/9/0
```

- `run show bfd session`: Provides a summary of the BFD session running on the IRB.

```
root@jcnr# run show bfd session
                                                  Detect   Transmit
Address                  State     Interface      Time     Interval  Multiplier
1.104.19.4               Up        irb.419        0.900     0.300        3
2419::4                  Up        irb.419        0.900     0.300        3
```

- `run ping routing-instance blue19 1.104.19.3 source 1.104.19.4 count 1 rapid`: Provides a confirmation of the network connectivity to the IRB interface from CE2.

```
root@CE2# run ping routing-instance blue19 1.104.19.3 source 1.104.19.4 count 1 rapid
PING 1.104.19.3 (1.104.19.3): 56 data bytes
!
--- 1.104.19.3 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max/stddev = 9.041/9.041/9.041/0.000 ms
```

- `run ping routing-instance blue19 6.1.19.1 source 1.104.19.4 count 1 rapid`: Provides a confirmation of the network connectivity to remote prefixes from CE2 through the IRB interface.

```
root@CE2# run ping routing-instance blue19 6.1.19.1 source 1.104.19.4 count 1 rapid
PING 6.1.19.1 (6.1.19.1): 56 data bytes
!
--- 6.1.19.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max/stddev = 17.773/17.773/17.773/0.000 ms
```

- `run traceroute routing-instance blue19 6.1.19.1 source 1.104.19.4 no-resolve`: Provides the trace routes to remote prefixes from CE2 through the IRB interface.

```
root@CE2# run traceroute routing-instance blue19 6.1.19.1 source 1.104.19.4 no-resolve
traceroute to 6.1.19.1 (6.1.19.1) from 1.104.19.4, 30 hops max, 52 byte packets
1  1.104.19.3  14.341 ms  14.932 ms  14.997 ms
2  6.1.19.1  14.962 ms  9.985 ms  14.906 ms
```

vRouter CLI Commands

The following CLI commands can be executed on the vRouter CLI. To access the vRouter CLI, see "Access vRouter CLI" on page 168.

- `vif --list | grep <interface ID>`: Provides the VIF ID of the specified interface.

```
bash-5.1# vif --list | grep irb.419
vif0/26     Virtual: irb.419 NH: 73
vif0/27     Virtual: irb.419 Vlan(o/i)(,S): 419/419
```

- `vif --get 26`: Provides the VRF ID where IRB.419 is attached to.

```
bash-5.1# vif --get 26
vif0/26     Virtual: irb.419 NH: 73
            Type:Irb HWaddr:02:22:ec:ac:6b:24 IPaddr:1.104.19.3
            IP6addr:2419::3
            DDP: OFF SwLB: ON
            Vrf:2 Mcast Vrf:2 Flags:L3L2DProxyEr QOS:-1 Ref:16
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Vlan Mode: Access  Vlan Id: 419  OVlan Id: 419
            RX packets:66910  bytes:5409152 errors:0
            TX packets:71340  bytes:5718843 errors:0
            Drops:9

bash-5.1# vif --get 27
vif0/27     Virtual: irb.419 Vlan(o/i)(,S): 419/419
            Parent:vif0/9  Sub-type:  host-irb-tap
            Type:Virtual(Vlan) HWaddr:02:22:ec:ac:6b:24 IPaddr:1.104.19.3
            IP6addr:2419::3
            DDP: OFF SwLB: ON
            Vrf:2 Mcast Vrf:65535 Flags:L3L2DProxyEr QOS:-1 Ref:1 TxXVif:26
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:71248  bytes:5711219 errors:0
            TX packets:66828  bytes:5134644 errors:0
            Drops:0
```

- `rt --get 1.104.19.4/32 --vrf 2`: Provides the data plane encapsulation for CE2’s IP.

```
bash-5.1# rt --get 1.104.19.4/32 --vrf 2
Match 1.104.19.4/32 in vRouter inet4 table 0/2/unicast

Flags: L=Label Valid, P=Proxy ARP, T=Trap ARP, F=Flood ARP, Ml=MAC-IP learnt route
vRouter inet4 routing table 0/2/unicast
Destination           PPL        Flags        Label         Nexthop    Stitched MAC(Index)
1.104.19.4/32           0           PT            -            113        -

bash-5.1# nh --get 113
Id:113        Type:Encap          Fmly:AF_INET/6  Rid:0  Ref_cnt:11         Vrf:2
              Flags:Valid, Policy, Etree Root,
              EncapFmly:0806 Oif:26 Len:14
              Encap Data: e4 5d 37 2b 2a aa 02 22 ec ac 6b 24
```

- `purel2cli --mac show`: Provides the MAC table in the vRouter.

```
bash-5.1# purel2cli --mac show | grep 419
02:22:ec:ac:6b:24 419       26         1
e4:5d:37:2b:2a:aa 419       3          68174
```

L3 Routing Protocols

SUMMARY

Read this topic to know about the L3 routing protocols that are supported by the Juniper Cloud Native Router, including BGP, IS-IS, and OSPF.

IN THIS SECTION

- Supported L3 protocols | 72
- BGP | 72

IS-IS | 73
OSPF | 74

Supported L3 protocols

The Juniper Cloud-Native router supports the following L3 routing protocols, each of which can be configured via node annotations at the time of deployment or via the "cRPD CLI" on page 167 for a running cRPD pod. Here is an example configuration snippet from the go template with node annotations:

```
protocols {
    isis {
        interface all;
        {{if and .Env.SRGB_START_LABEL .Env.SRGB_INDEX_RANGE}}
        source-packet-routing {
            srgb start-label {{.Env.SRGB_START_LABEL}} index-range {{.Env.SRGB_INDEX_RANGE}};
            node-segment {
                {{if .Node.srIPv4NodeIndex}}
                ipv4-index {{.Node.srIPv4NodeIndex}};
                {{end}}
                {{if .Node.srIPv6NodeIndex}}
                ipv6-index {{.Node.srIPv6NodeIndex}};
                {{end}}
            }
        }
        {{end}}
        level 1 disable;
    }
}
```

BGP

BGP is an exterior gateway protocol (EGP) that is used to exchange routing information among routers in different autonomous systems (ASs). BGP routing information includes the complete route to each destination. BGP uses the routing information to maintain a database of network reachability information, which it exchanges with other BGP systems. BGP uses the network reachability information to construct a graph of AS connectivity, which enables BGP to remove routing loops and enforce policy decisions at the AS level. The cloud-native router supports BGP version 4. Here is an example to configure BGP protocol on the cloud-native router "via the cRPD shell" on page 167:

```
set protocols bgp group CNI type internal
set protocols bgp group CNI local-address 10.0.0.1
set protocols bgp group CNI family inet-vpn unicast
set protocols bgp group CNI family inet6-vpn unicast
set protocols bgp group CNI neighbor 10.0.1.1 peer-as 64512
set protocols bgp group CNI neighbor 10.0.1.1 local-as 64512
set routing-options route-distinguisher-id 10.0.0.1
```

You can issue the `show bgp summary` command on the cRPD shell to view the BGP summary information for all routing instances. For example:

```
user@host> show bgp summary
Threading mode: BGP I/O
Default eBGP mode: advertise - accept, receive - accept
Groups: 1 Peers: 1 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
bgp.l3vpn.0
                       2          2          0          0          0          0
bgp.l3vpn-inet6.0
                       2          2          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#Active/
Received/Accepted/Damped...
10.0.1.1             64512        249        211       0       0     1:32:42 Establ
  bgp.l3vpn.0: 2/2/2/0
  bgp.l3vpn-inet6.0: 2/2/2/0
  jcnr-3.inet.0: 2/2/2/0
  jcnr-3.inet6.0: 2/2/2/0
```

Refer the BGP User Guide for more information.

IS-IS

The IS-IS protocol is an interior gateway protocol (IGP) that uses link-state information to make routing decisions. IS-IS is a link-state IGP that uses the shortest-path-first (SPF) algorithm to determine routes. IS-IS evaluates the topology changes and determines whether to perform a full SPF recalculation or a partial route calculation (PRC). IS-IS uses hello packets that allow network convergence to occur quickly when network changes are detected. The cloud-native router supports IS-IS.

Here is an example to configure IS-IS protocol on the cloud-native router "via the cRPD shell" on page 167:

```
set security forwarding-options family iso mode packet-based
set interfaces eno3v0 unit 0 family inet address 10.100.12.1/30
set interfaces eno3v0 unit 0 family iso
set interfaces lo0 unit 0 family inet address 192.168.0.1/32
set interfaces lo0 unit 0 family iso address 49.0002.0192.0168.0001.00
set protocols isis interface eno3v0
set protocols isis interface lo0.0
```

You can issue the `show isis adjacency` and `show isis interface` commands to verify the protocol configuration. Refer the IS-IS User Guide for information.

OSPF

OSPF is an interior gateway protocol (IGP) that routes packets within a single autonomous system (AS). OSPF uses link-state information to make routing decisions, making route calculations using the shortest-path-first (SPF) algorithm (also referred to as the Dijkstra algorithm). Each router running OSPF floods link-state advertisements throughout the AS or area that contain information about that router's attached interfaces and routing metrics. Each router uses the information in these link-state advertisements to calculate the least cost path to each network and create a routing table for the protocol. The cloud-native router supports OSPF version 2 (OSPFv2) and OSPF version 3 (OSPFv3). Here is an example to configure IS-IS protocol on the cloud-native router "via the cRPD shell" on page 167:

```
set protocols ospf area 0.0.0.0 interface bond0
set protocols ospf area 0.0.0.0 interface lo passive
```

Once you bring up the pods, verify the OSPF configuration:

```
show ospf neighbor
Address          Interface              State           ID               Pri  Dead
192.168.123.254  bond0                  Full            123.1.1.254      128    36
```

```
show route 1.1.24.24

inet.0: 27 destinations, 29 routes (27 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

1.1.24.24/32       *[OSPF/10] 00:07:08, metric 2
                    >  to 192.168.123.254 via bond0
```

Refer the OSPF User Guide for more information.

MPLS Support

IN THIS SECTION

- MPLS Support | 76

The Juniper Cloud-Native Router contains support for MPLS routing protocols. You use the JCNR-controller, or cRPD, to configure MPLS using the node annotations at the time of deployment or via the "cRPD CLI" on page 167.

The cRPD then sends the configuration to the vRouter-agent, using gRPC. The vRouter-agent then converts the configuration to network policies that it imlements in the vRouter. The cloud-native router supports the following MPLS-based routing protocols:

MPLS Support

- **L3 MPLS VPN (MPLS)**—L3 MPLS VPNs are also known as BGP/MPLS VPNs because BGP is used to distribute VPN routing information across the provider's backbone, and MPLS is used to forward VPN traffic across the backbone to remote VPN sites. The cloud-native router can particpate as a sending, receiving or transit router using the MPLS protocol. Review the L3 VPN User Guide for more information.

- **Segment Routing-MPLS (SR-MPLS)**—Segment routing is a control-plane architecture that enables an ingress router to steer a packet through a specific set of nodes and links in the network without relying on the intermediate nodes in the network to determine the actual path it should take. SR-MPLS employs segment routing in MPLS. The cloud-native router can participate as a sending, receiving or transit router in SR-MPLS networks. Review the Junos source packet routing topic for a configuration example.

- **MPLS over UDP (MPLSoUDP)**—MPLSoUDP is an overlay technology that encapsulates MPLS packets within UDP packets to traverse through some networks that do not support native MPLS or SR-MPLS. The cloud-native router can participate as a sending, receiving or transit router using MPLSoUDP. Review the Configuring Next-Hop-Based MPLSoUDP Tunnels topic for a configuration example.

- **Label Distribution Protocol (LDP)**—The Label Distribution Protocol (LDP) is a protocol for distributing labels in non-traffic-engineered applications. LDP allows routers to establish label-switched paths (LSPs) through a network by mapping network-layer routing information directly to data link layer-switched paths. The cloud-native router can participate as a sending, receiving or transit router using LDP. Review the LDP Overview topic for more information.

Bidirectional Forwarding Detection (BFD)

SUMMARY

Read this topic to know about the support for Bidirectional Forwarding Detection (BFD) in the Juniper Cloud-Native router.

The Bidirectional Forwarding Detection (BFD) protocol is a simple hello mechanism that detects failures in a network. A pair of routing devices exchange BFD packets. The devices send hello packets at a specified, regular interval. The device detects a neighbor failure when the routing device stops receiving a reply after a specified interval. The cloud-native router supports BFD. Review the Understanding BFD topic for more information.

Virtual Router Redundancy Protocol (VRRP)

SUMMARY

Read this topic to learn about the support for the Virtual Router Redundancy Protocol (VRRP) in Juniper Cloud-Native router.

The Virtual Router Redundancy Protocol (VRRP) enables hosts on a LAN to make use of redundant routing platforms on that LAN without requiring more than the static configuration of a single default route on the hosts. The VRRP routing platforms share the IP address corresponding to the default route configured on the hosts. At any time, one of the VRRP routing platforms is the primary (active) and the others are backups. If the primary routing platform fails, one of the backup routing platforms becomes the new primary routing platform, providing a virtual default routing platform and enabling traffic on the LAN to be routed without relying on a single routing platform. Using VRRP, a backup device can take over a failed default device within a few seconds. This is done with minimum VRRP traffic and without any interaction with the hosts. When JCNR is deployed in the containerized network function (CNF) mode in cloud deployments, the VRRP unicast can be used to decide between the active and backup JCNR nodes. Review the Understanding VRRP topic for more information.

NOTE: To enable VRRP for JCNR on an EKS cluster, a ConfigMap must be configured. Please review *JCNR ConfigMap for VRRP* topic for more information

Virtual Routing Instance (VRF-Lite)

SUMMARY

Read this topic to understand the implementation of virtual routing instances in JCNR.

IN THIS SECTION

- Configuration | 78

Virtual routing instances allow administrators to divide the cloud-native router into multiple independent virtual routers, each with its own routing table. Splitting a device into many virtual routing instances isolates traffic traveling across the network without requiring multiple devices to segment the network. You can use virtual routing instances to isolate customer traffic on your network and to bind customer-specific instances to customer-owned interfaces. Virtual routing and forwarding (VRF) is often used in conjunction with Layer 3 subinterfaces, allowing traffic on a single physical interface to be differentiated and associated with multiple virtual routers. Each logical Layer 3 subinterface can belong to only one routing instance. Review the Virtual Router Instances topic for more information.

Configuration

You can create a virtual routing instance in JCNR via a network attachment definition (NAD) manifest. Here is an example NAD to create a `bluenet` virtual router routing instance:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: blue
spec:
  config: '{
    "cniVersion":"0.4.0",
    "name": "blue-net",
    "plugins": [
      {
        "type": "jcnr",
        "args": {
          "instanceName": "bluenet",
          "instanceType": "virtual-router"
        },
        "kubeConfig":"/root/.kube/config"
      }
    ]
  }'
```

Note the `instanceType` is set to `virtual-router`. Refer to "JCNR Use-Cases and Configuration Overview " on page 84 for more information on NAD.

Here is an example configuration for a `podblue` pod with an interface (`192.168.11.10/24`) attached to the `blue` network (output is trimmed for brevity):

```
apiVersion: v1
kind: Pod
metadata:
  name: podblue
  annotations:
    k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "blue",
          "interface":"net1",
          "cni-args": {
            "interfaceType":"veth",
            "dataplane":"dpdk",
            "mac":"aa:bb:cc:dd:ee:10",
            "ipConfig":{
              "ipv4":{
                "address":"192.168.11.10/24",
                "gateway":"192.168.11.1",
                "routes":["192.168.11.0/24"]
              },
              "ipv6":{
                "address":"abcd::192.168.11.10/112",
                "gateway":"abcd::192.168.11.1",
                "routes":["abcd::192.168.11.0/112"]
              }
            }
          }
        }
      ]
spec:
...
```

As you apply the NAD and the pod manifests using the `kubectl apply -f` *`manifest`* command, the `bluenet` routing instance and `bluenet.inet.0` routing table is created in the JCNR controller. You can configure JCNR to enable communication from `podblue` to pods on the remote network. Additional cRPD configuration can be perfomed by "accessing the cRPD shell" on page 167. Here is an example cRPD configuration:

1. Configure the local fabric interface and the BGP protocol:

   ```
   set interfaces ens2f0 unit 0 family inet address 10.10.10.11/24
   set protocols bgp group overlay type internal
   set protocols bgp group overlay local-address 10.10.10.11
   set protocols bgp group overlay local-as 64520
   set protocols bgp group overlay neighbor 10.10.10.12 peer-as 64520
   ```

   where `10.10.10.12/24` is the IP address of the BGP peer or neighbor router.

2. Export the `inet` routes using the BGP protocol:

   ```
   set policy-options policy-statement send_direct term 1 from protocol direct
   set policy-options policy-statement send_direct term 1 then accept
   set policy-options policy-statement send_direct term reject then reject
   set protocols bgp group overlay export send_direct
   ```

3. Leak the routes from the `bluenet` routing instance to the `default` routing instance:

   ```
   set groups cni routing-instances bluenet routing-options interface-routes rib-group inet
   blue_to_inet
   set routing-options rib-groups blue_to_inet import-rib bluenet.inet.0
   set routing-options rib-groups blue_to_inet import-rib inet.0
   ```

4. Leak only the BGP routes matching prefix `192.168.12.0` from `inet.0` to the `bluenet` routing instance, where `192.168.12.0/24` is the remote pod network:

   ```
   set policy-options policy-statement inet_to_blue term from_bgp from instance master
   set policy-options policy-statement inet_to_blue term from_bgp from protocol bgp
   set policy-options policy-statement inet_to_blue term from_bgp from route-filter
   192.168.12.0/24 orlonger
   set policy-options policy-statement inet_to_blue term from_bgp then accept
   set policy-options policy-statement inet_to_blue term reject then reject
   set routing-options rib-groups inet_to_blue import-rib inet.0
   ```

```
set routing-options rib-groups inet_to_blue import-rib bluenet.inet.0
set routing-options rib-groups inet_to_blue import-policy inet_to_blue
set groups cni routing-instances bluenet routing-options instance-import inet_to_blue
```

NOTE: JCNR supports route leaking between virtual router routing instances for routes with interface, receive, resolve and table next-hops.

RELATED DOCUMENTATION

Rib-Groups

ECMP

SUMMARY

Read this topic to know about the support for ECMP with flow stickiness in the Juniper Cloud-Native Router.

Equal-cost multipath (ECMP) is a network routing strategy that allows for traffic of the same session, or flow—that is, traffic with the same source and destination—to be transmitted across multiple paths of equal cost. It is a mechanism that allows you to load balance traffic and increase bandwidth by fully utilizing otherwise unused bandwidth on links to the same destination.

When forwarding a packet, the routing technology must decide which next-hop path to use. In making a determination, the device takes into account the packet header fields that identify a flow. When ECMP is used, next-hop paths of equal cost are identified based on routing metric calculations and hash algorithms. That is, routes of equal cost have the same preference and metric values, and the same cost to the network. The ECMP process identifies a set of routers, each of which is a legitimate equal cost next hop towards the destination. The routes that are identified are referred to as an ECMP set. Because it addresses only the next hop destination, ECMP can be used with most routing protocols.

An equal-cost multipath (ECMP) set is formed when the routing table contains multiple next-hop addresses for the same destination with equal cost. (Routes of equal cost have the same preference and metric values.) If there is an ECMP set for the active route, the Cloud-Native Router uses a consistent hash to choose *one* of the next-hop addresses from the ECMP members to forward the packet.

The cloud-native router supports ECMP for both Container Network Interface (CNI) and transit router modes. It supports flow stickiness when the number of next-hops is changed. The cloud-native router also supports ECMP next-hop for tunneled traffic.

BGP Unnumbered

SUMMARY

Read this topic to know about the support for BGP unnumbered in the cloud-native router.

Juniper Cloud-Native Router supports BGP unnumbered peering starting in Release 23.2. This feature allows BGP to auto-discover and to create peer neighbor sessions using the link-local IPv6 addresses of directly connected neighbors. Using BGP unnumbered peering, which dynamically discovers IPV6 neighbors, reduces the burden of manually configuring an IPv6 underlay. It is used in N-tier Clos architecture for point-to-point links. BGP unnumbered is supported in the default VRF (VRF-0) and virtual routing instances (virtual-router). Read the BGP Unnumbered topic for more information.

NOTE: When a BGP unnumbered IPv6 session is established between 2 provider edge routers (PEs) and IPv4 routes are being exchanged over that session, then the next hop for an IPv4 route is an IPv6 address. This feature is supported on PEs having Linux kernel version 5 and above. If the Linux kernel version is below 5, then the IPv4 routes are not added to the routing table.

5

CHAPTER

JCNR CNI Configuration Examples

JCNR Use-Cases and Configuration Overview | 84

L2 Kernel Access-Mode Interface Configuration Example | 89

L2 `virtio` Trunk-Mode Interface Configuration Example | 94

L2 VLAN Sub-Interface Configuration Example | 99

L3 VPN Interface Configuration Example | 104

L3 VLAN Sub-Interface Configuration Example | 111

JCNR Use-Cases and Configuration Overview

SUMMARY

Read this chapter to review configuration examples for various Juniper Cloud-Native Router use cases when deployed in the container network interface (CNI) mode.

IN THIS SECTION

- Configuration Example | 84
- Troubleshooting | 88

The Juniper Cloud-Native Router can be deployed as a virtual switch or a transit router, either as a pure container network function (CNF) or as a container network interface (CNI). In the CNF mode, there are no application pods running on the node and the router only performs packeting switching or forwarding through various interfaces on the system. In the CNI mode, application pods using software-based network interfaces such as veth-pairs or DPDK vhost-user based interfaces, attach to the cloud-native router. This chapter provides configuration examples for attaching different workload interface types to the cloud-native router CNI instance.

Configuration Example

The JCNR CNI is deployed as a secondary CNI along with Multus as a primary CNI, to create different types of secondary interfaces for the application pod. Multus uses a network attachment definition (NAD) file to configure a secondary interface for the application pod. The NAD specifies how to create a secondary interface, IP address allocation, network instance and more. A pod can have one or more NADs, typically one per pod interface. The `config:` field in the NAD file defines the JCNR CNI configuration. Here is a generic format of the NAD:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: <vrf-name>
spec:
  config: '{
    "cniVersion":"0.4.0",
    "name": "<vrf-name>",
    "plugins": [
      {
        "type": "jcnr",
        "args": {
           "key1":"value1",
           "key2","value2",
           ....
        },
        "ipam": {
          "type": "<ipam-type>",
          ....
        },
        "kubeConfig":"/etc/kubernetes/kubelet.conf"
      }
    ]
  }'
```

While configuring the NAD for the JCNR plugin type, the following keys are supported:

Table 2: Supported Keys in NAD

Key	Description
instanceName	The routing-instance name
instanceType	One of: virtual-router—for non-VPN-related applications vrf—Layer 3 VPN implementations virtual-switch—Layer 2 implementations
interfaceType	Either "veth" or "virtio"
vlanId	A valid vlan id "1-4095"
bridgeVlanId	A valid vlan id "1-4095"
vlanIdList	A list of command separated vlan-id, e.g: "1, 5, 7, 10-20"
parentInterface	Valid interface name as it should appear in the pod. Child/sub-interfaces have parentInterface as their prefix followed by "." If parentInterface is specified, sub interface must be explicitly specified.
vrfTarget	The route-target for vrf routing instance

Table 2: Supported Keys in NAD *(Continued)*

Key	Description
bridgeDomain	Bridge Domain under which pod interface should be attached in the virtual-switch instance.
type (ipam)	static—assigns same IP to all pods, to assign a unique IP per pod define a unique NAD per pod per interface host-local—unique IP address per pod interface on the same host. IP addresses are not unique across two different nodes whereabouts—unique IP address per pod across all nodes (https://github.com/k8snetworkplumbingwg/whereabouts)

Consider the example NAD for a layer 2 kernel access mode interface:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vswitch-pod1-bd100
spec:
  config: '{
    "cniVersion":"0.4.0",
    "name": "vswitch-pod1-bd100",
    "plugins": [
      {
        "type": "jcnr",
        "args": {
          "instanceName": "vswitch",
          "instanceType": "virtual-switch",
       "interfaceType": "veth",
          "bridgeDomain": "bd100",
          "bridgeVlanId": "100"
        },
        "ipam": {
          "type": "static",
          "addresses":[
            {
              "address":"99.61.0.2/16",
              "gateway":"99.61.0.1"
            },
            {
              "address":"1234::99.61.0.2/120",
              "gateway":"1234::99.61.0.1"
            }
          ]
        },
        "kubeConfig":"/etc/kubernetes/kubelet.conf"
      }
    ]
  }'
```

The pod attaches to the router instance using the `k8s.v1.cni.cncf.io/networks` annotation. For example:

```
apiVersion: v1
kind: Pod
metadata:
  name:   pod1
  annotations:
    k8s.v1.cni.cncf.io/networks: vswitch-pod1-bd100
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - kind-worker
  containers:
    - name: pod1
      image: ubuntu:latest
      imagePullPolicy: IfNotPresent
      securityContext:
        privileged: false
      env:
        - name: KUBERNETES_POD_UID
          valueFrom:
            fieldRef:
               fieldPath: metadata.uid
      volumeMounts:
        - name: dpdk
```

```
          mountPath: /dpdk
          subPathExpr: $(KUBERNETES_POD_UID)
  volumes:
    - name: dpdk
      hostPath:
        path: /var/run/jcnr/containers
```

The volume mount host path exposes the UNIX domain socket of the vhost-user port to the DPDK application. The DPDK interface details are stored at `/dpdk/dpdk-interfaces.json` inside the application container for the DPDK application to consume. It is also exported into the pod as a pod annotation.

When you create a pod for use in the cloud-native router, the Kubernetes component known as **kubelet** calls the Multus CNI to set up pod networking and interfaces. Multus reads the annotations section of the **pod.yaml** file to refer the corresponding NAD. If a NAD points to `jcnr` as the CNI plug in, Multus calls the JCNR-CNI to set up the pod interface. JCNR-CNI creates the interface as specified in the NAD. JCNR-CNI then generates and pushes a configuration into cRPD.

Troubleshooting

Pods main fail to come up for various reasons:

- Image not found
- CNI failed to add interfaces
- CNI failed to push configuration into cRPD
- CNI failed to invoke vRouter REST APIs
- The NAD is invalid or undefined

The following commands will be useful to troubleshooting pod issues:

```
# Check the Pod status
kubectl get pods -A
```

```
# Check pod state and CNI logs
kubectl describe pod <pod-name>
```

```
# Check the pod logs
kubectl logs pod <pod-name>
```

```
# Check the net-attach-def
kubectl get net-attach-def <net-attach-def-name> -o yaml
```

```
# Check CNI logs
tail -f /var/log/jcnr/jcnr-cni.log
```

```
# Check the cRPD config added by CNI (on the cRPD CLI)
cli> show configuration groups cni
```

L2 Kernel Access-Mode Interface Configuration Example

SUMMARY

Read this topic to learn how to add a user pod with a `kernel/veth` access-mode interface to an instance of the cloud-native router.

IN THIS SECTION

- Overview | 90

● Configuration Example | 90

Overview

You can configure a user pod with a Layer 2 access-mode `kernel` interface and attach it to the JCNR instance. The Juniper Cloud-Native Router must have an L2 interface configured at the time of deployment. Your high-level tasks are:

- Define and apply a network attachment definition (NAD)—The NAD file defines the required configuration for Multus to invoke the JCNR-CNI and create a network to attach the pod interface to.
- Define and apply a pod YAML file to your cloud-native router cluster—The pod YAML contains the pod specifications and an annotation to the network created by the JCNR-CNI.

> **NOTE**: Please review the "JCNR Use-Cases and Configuration Overview " on page 84 topic for more information on NAD and pod YAML files.

Configuration Example

1. Here is an example NAD to create a Layer 2 `kernel/veth` access-mode interface with static IPAM:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vswitch-pod1-bd100
spec:
  config: '{
    "cniVersion":"0.4.0",
    "name": "vswitch-pod1-bd100",
    "plugins": [
      {
        "type": "jcnr",
        "args": {
          "instanceName": "vswitch",
          "instanceType": "virtual-switch",
       "interfaceType": "veth",
          "bridgeDomain": "bd100",
          "bridgeVlanId": "100"
        },
        "ipam": {
          "type": "static",
          "addresses":[
            {
              "address":"99.61.0.2/16",
              "gateway":"99.61.0.1"
            },
            {
              "address":"1234::99.61.0.2/120",
              "gateway":"1234::99.61.0.1"
            }
          ]
        },
        "kubeConfig":"/etc/kubernetes/kubelet.conf"
      }
    ]
  }'
```

The NAD defines a bridge domain `bd100` under which a `veth` type pod interface should be attached in the `virtual-switch` instance.

It also defines a static IP address to be assigned to the pod interface.

2. Apply the NAD manifest to create the network.

```
kubectl apply -f nad-access_mode.yaml
networkattachmentdefinition.k8s.cni.cncf.io/vswitch-pod1-bd100 created
```

3. Verify the NAD is created.

```
[root@jcnr-01]# kubectl get net-attach-def
NAME                 AGE
vswitch-pod1-bd100   59s
```

4. Here is an example yaml to create a pod attached to the `vswitch-pod1-bd100` network:

```
apiVersion: v1
kind: Pod
metadata:
  name:   pod1
  annotations:
    k8s.v1.cni.cncf.io/networks: vswitch-pod1-bd100
spec:
  containers:
    - name: pod1
      image: ubuntu:latest
      imagePullPolicy: IfNotPresent
      securityContext:
        privileged: false
      env:
        - name: KUBERNETES_POD_UID
          valueFrom:
            fieldRef:
               fieldPath: metadata.uid
      volumeMounts:
        - name: dpdk
          mountPath: /dpdk
          subPathExpr: $(KUBERNETES_POD_UID)
  volumes:
    - name: dpdk
      hostPath:
        path: /var/run/jcnr/containers
```

The pod attaches to the router instance using the `k8s.v1.cni.cncf.io/networks` annotation

.

5. Apply the pod manifest.

```
[root@jcnr-01]# kubectl apply -f pod_access_mode.yaml
pod/pod1 created
```

6. Verify the pod is running.

```
[root@jcnr-01 ~]# kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
pod1   1/1     Running   0          2m38s
```

7. Describe the pod to verify a secondary interface is created and attached to the `vswitch-pod1-bd100` network. (The output is trimmed for brevity).

```
[root@jcnr-01 ~]# kubectl describe pod pod1
Name:         pod1
Namespace:    default
Priority:     0
Node:         jcnr-01/10.100.20.25
Start Time:   Mon, 26 Jun 2023 09:36:57 -0400
Labels:       <none>
Annotations:  cni.projectcalico.org/containerID:
5b92668a6d7580e587de951d660c99969ce98bc239502afab6f9d191653f1e9b
              cni.projectcalico.org/podIP: 10.233.91.79/32
              cni.projectcalico.org/podIPs: 10.233.91.79/32
              k8s.v1.cni.cncf.io/network-status:
                [{
                    "name": "k8s-pod-network",
                    "ips": [
                        "10.233.91.79"
                    ],
                    "default": true,
                    "dns": {}
                },{
                    "name": "default/vswitch-pod1-bd100",
                    "interface": "net1",
                    "ips": [
                        "99.61.0.2",
                        "1234::633d:2"
                    ],
                    "mac": "02:00:00:5D:74:76",
                    "dns": {}
                }]
...
```

8. Verify the vRouter has the corresponding interface created. "Access the vRouter CLI" on page 168 and issue the `vif --list` command.

```
vif0/2      Ethernet: jvknet1-7c557fe MTU: 9160
            Type:Virtual HWaddr:02:00:00:66:01:56
            DDP: OFF SwLB: ON
            Vrf:0 Flags:L2Vof QOS:-1 Ref:8
            RX port   packets:20 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Vlan Mode: Access  Vlan Id: 100  OVlan Id: 100
            RX packets:7  bytes:518 errors:13
            TX packets:31  bytes:2438 errors:0
            Drops:14
            TX port   packets:31 errors:0
```

   Note that the interface type is `Virtual` and the Vlan mode is set to `access` with the Vlan ID set to `100`. The VRF is always 0 for L2 interfaces.

L2 `virtio` Trunk-Mode Interface Configuration Example

SUMMARY

Read this topic to learn how to add a user pod with a `virtio` trunk-mode interface to an instance of the cloud-native router.

IN THIS SECTION

- Overview | 94
- Configuration Example | 95

Overview

You can configure a user pod with a Layer 2 trunk-mode virtio interface and attach it to the JCNR instance. The Juniper Cloud-Native Router must have an L2 interface configured at the time of deployment. Your high-level tasks are:

- Define and apply a network attachment definition (NAD)—The NAD file defines the required configuration for Multus to invoke the JCNR-CNI and create a network to attach the pod interface to.

- Define and apply a pod YAML file to your cloud-native router cluster—The pod YAML contains the pod specifications and an annotation to the network created by the JCNR-CNI.

  NOTE: Please review the "JCNR Use-Cases and Configuration Overview " on page 84 topic for more information on NAD and pod YAML files.

Configuration Example

1. Here is an example NAD to create a Layer 2 trunk-mode virtio interface interface with static IPAM:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vswitch
spec:
  config: '{
    "cniVersion":"0.4.0",
    "name": "vswitch",
    "type": "jcnr",
    "args": {
      "instanceName": "vswitch",
      "instanceType": "virtual-switch",
      "vlanIdList":"201, 202, 203"
    },
    "ipam": {
         "type": "static",
         "capabilities":{"ips":true},
         "addresses":[
           {
             "address":"10.2.1.1/24",
             "gateway":"10.2.1.253"
           },
           {
             "address":"2001::10.2.1.1/120",
             "gateway":"2001::10.2.1.253"
           }
         ]
    },
    "kubeConfig":"/etc/kubernetes/kubelet.conf"
  }'
```

The NAD defines the VLAN IDs for the `virtual-switch` instance to which the pod's trunk interface will be attached.

2. Apply the NAD manifest to create the network.

```
kubectl apply -f nad_trunk_mode.yaml
networkattachmentdefinition.k8s.cni.cncf.io/vswitch created
```

3. Verify the NAD is created.

```
[root@jcnr-01]# kubectl get net-attach-def
NAME                 AGE
vswitch              57s
```

4. Here is an example yaml to create a pod attached to the `vswitch` network:

```
apiVersion: v1
kind: Pod
metadata:
  name:   pod1
  annotations:
    k8s.v1.cni.cncf.io/networks: vswitch
spec:
  containers:
    - name: pod1
      image: ubuntu:latest
      imagePullPolicy: IfNotPresent
      securityContext:
        privileged: false
      env:
        - name: KUBERNETES_POD_UID
          valueFrom:
            fieldRef:
               fieldPath: metadata.uid
      volumeMounts:
        - name: dpdk
          mountPath: /dpdk
          subPathExpr: $(KUBERNETES_POD_UID)
  volumes:
    - name: dpdk
      hostPath:
        path: /var/run/jcnr/containers
```

The pod attaches to the router instance using the `k8s.v1.cni.cncf.io/networks` annotation.

5. Apply the pod manifest.

```
[root@jcnr-01]# kubectl apply -f pod_trunk_mode.yaml
pod/pod1 created
```

6. Verify the pod is running.

```
[root@jcnr-01 ~]# kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
pod1   1/1     Running   0          38s
```

7. Describe the pod to verify a secondary interface is created and attached to the `vswitch` network. (The output is trimmed for brevity).

```
[root@jcnr-01 ~]# kubectl describe pod pod1
Name:         pod1
Namespace:    default
Priority:     0
Node:         jcnr-01/10.100.20.25
Start Time:   Mon, 26 Jun 2023 09:53:31 -0400
Labels:       <none>
Annotations:  cni.projectcalico.org/containerID:
ac6f0a26ebfe68adf3b020d0def96f09e6b2b5c6303f55c0dde277b1ce7f9d9f
              cni.projectcalico.org/podIP: 10.233.91.81/32
              cni.projectcalico.org/podIPs: 10.233.91.81/32
              jcnr.juniper.net/dpdk-interfaces:
                [
                    {
                        "name": "net1",
                        "vhost-adaptor-path": "/dpdk/vhost-net1.sock",
                        "vhost-adaptor-mode": "client",
                        "ipv4-address": "10.2.1.1/24",
                        "ipv6-address": "2001::a02:101/120",
                        "mac-address": "02:00:00:5B:C7:9F"
                    }
                ]
              k8s.v1.cni.cncf.io/network-status:
                [{
                    "name": "k8s-pod-network",
                    "ips": [
                        "10.233.91.81"
                    ],
                    "default": true,
                    "dns": {}
                },{
                    "name": "default/vswitch",
                    "interface": "net1",
                    "ips": [
                        "10.2.1.1",
                        "2001::a02:101"
                    ],
                    "mac": "02:00:00:5B:C7:9F",
                    "dns": {}
                }]
...
```

8. Verify the vRouter has the corresponding interface created. "Access the vRouter CLI" on page 168 and issue the `vif --list` command.

```
vif0/2      PMD: vhostnet1-57f38cc0-6555-4bc2-ac MTU: 9160
            Type:Virtual HWaddr:02:00:00:dc:c9:27
            DDP: OFF SwLB: ON
            Vrf:0 Flags:L2 QOS:-1 Ref:11
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Vlan Mode: Trunk  Vlan: 201-203
            RX packets:0  bytes:0 errors:0
            TX packets:4  bytes:256 errors:0
            Drops:0
            TX port   packets:0 errors:4
```

Note that the interface type is `Virtual` and the Vlan mode is set to `trunk` with the Vlan ID set to `201-203`. The VRF is always 0 for L2 interfaces.

L2 VLAN Sub-Interface Configuration Example

SUMMARY

Read this topic to learn how to add a user pod with a Layer 2 VLAN sub-interface to an instance of the cloud-native router.

IN THIS SECTION

- Overview | 99
- Configuration Example | 100

Overview

You can configure a user pod with a Layer 2 VLAN sub-interface and attach it to the JCNR instance. The Juniper Cloud-Native Router must have an L2 interface configured at the time of deployment. The cRPD must be configured with the valid VLAN configuration for the fabric interface. For example:

```
set interfaces eth1 unit 100 vlan-id 100
```

NOTE: Note that the unit number and the VLAN ID must match.

Your high-level tasks are:

- Define and apply a network attachment definition (NAD)—The NAD file defines the required configuration for Multus to invoke the JCNR-CNI and create a network to attach the pod interface to.
- Define and apply a pod YAML file to your cloud-native router cluster—The pod YAML contains the pod specifications and an annotation to the network created by the JCNR-CNI

  NOTE: Please review the "JCNR Use-Cases and Configuration Overview " on page 84 topic for more information on NAD and pod YAML files.

Configuration Example

1. Here is an example NAD to create a Layer 2 VLAN sub-interface:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vswitch-bd201-sub
spec:
  config: '{
    "cniVersion":"0.4.0",
    "name": "vswitch-bd201-sub",
    "capabilities":{"ips":true},
    "plugins": [
      {
        "type": "jcnr",
        "args": {
          "instanceName": "vswitch",
          "instanceType": "virtual-switch",
          "bridgeDomain": "bd201",
          "bridgeVlanId": "201",
          "parentInterface": "net1",
          "interface": "net1.201"
        },
        "ipam": {
          "type": "static",
          "capabilities":{"ips":true},
          "addresses":[
            {
              "address":"10.3.0.1/24",
              "gateway":"10.3.0.254"
            },
            {
              "address":"2001:db8:3003::10.3.0.1/120",
              "gateway":"2001:db8:3003::10.3.0.1"
            }
          ]
        },
        "kubeConfig":"/etc/kubernetes/kubelet.conf"
      }
    ]
  }'
```

The NAD defines a bridge domain `bd201` and a sub-interface `net1.201` with a parent interface `net1`. The pod will be attached in the `virtual-switch` instance.. It also defines a static IP address to be assigned to the pod interface.

2. Apply the NAD manifest to create the network.

```
kubectl apply -f nad_l2_vlan_subinterface.yaml
networkattachmentdefinition.k8s.cni.cncf.io/vswitch-bd201-sub created
```

3. Verify the NAD is created.

```
[root@jcnr-01]# kubectl get net-attach-def
NAME                 AGE
vswitch-bd201-sub    43s
```

4. Here is an example yaml to create a pod attached to the `vswitch-bd201-sub` network:

```
apiVersion: v1
kind: Pod
metadata:
  name:   pod1
  annotations:
    k8s.v1.cni.cncf.io/networks: "vswitch-bd201-sub"
spec:
  containers:
    - name: pod1
      image: ubuntu:latest
      imagePullPolicy: IfNotPresent
      securityContext:
        privileged: false
      resources:
        requests:
          memory: 2Gi
        limits:
          hugepages-1Gi: 2Gi
      env:
        - name: KUBERNETES_POD_UID
```

```
          valueFrom:
            fieldRef:
               fieldPath: metadata.uid
      volumeMounts:
        - name: dpdk
          mountPath: /dpdk
          subPathExpr: $(KUBERNETES_POD_UID)
        - mountPath: /dev/hugepages
          name: hugepage
  volumes:
    - name: dpdk
      hostPath:
        path: /var/run/jcnr/containers
    - name: hugepage
      emptyDir:
        medium: HugePages
```

The pod attaches to the router instance using the `k8s.v1.cni.cncf.io/networks` annotation.

5. Apply the pod manifest.

```
[root@jcnr-01]# kubectl apply -f pod_access_mode.yaml
pod/pod1 created
```

6. Verify the pod is running.

```
[root@jcnr-01 ~]# kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
pod1   1/1     Running   0          40s
```

7. Describe the pod to verify a secondary interface is created and attached to the `vswitch-bd201-sub` network. (The output is trimmed for brevity).

```
[root@jcnr-01 ~]# kubectl describe pod pod1
Name:         pod1
Namespace:    default
Priority:     0
Node:         jcnr-01/10.100.20.25
Start Time:   Mon, 26 Jun 2023 09:53:31 -0400
Labels:       <none>
```

```
Annotations:  cni.projectcalico.org/containerID:
58642dd26f85769e14d302153357e84e6900398532d1b82b50a845ac1ede051a
              cni.projectcalico.org/podIP:
              cni.projectcalico.org/podIPs:
              jcnr.juniper.net/dpdk-interfaces:
                [
                    {
                        "name": "net1",
                        "vhost-adaptor-path": "/dpdk/vhost-net1.sock",
                        "vhost-adaptor-mode": "client",
                        "ipv4-address": "10.3.0.1/24",
                        "ipv6-address": "2001:db8:3003::a03:1/120",
                        "mac-address": "02:00:00:84:DC:42",
                        "vlan-id": "201"
                    }
                ]
              k8s.v1.cni.cncf.io/network-status:
                [{
                    "name": "k8s-pod-network",
                    "ips": [
                        "10.233.91.97"
                    ],
                    "default": true,
                    "dns": {}
                },{
                    "name": "default/vswitch-bd201-sub",
                    "interface": "net1",
                    "ips": [
                        "10.3.0.1",
                        "2001:db8:3003::a03:1"
                    ],
                    "mac": "02:00:00:84:DC:42",
                    "dns": {}
                }]
...
```

8. Verify the vRouter has the corresponding interface created. "Access the vRouter CLI" on page 168 and issue the `vif --list` command.

```
vif0/2      PMD: vhostnet1-d5eee4ec-dd7c-4e MTU: 9160
            Type:Virtual HWaddr:02:00:00:84:dc:42
            DDP: OFF SwLB: ON
            Vrf:65535 Flags:L2 QOS:-1 Ref:14
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:0
            TX port   packets:0 errors:293

vif0/3      Virtual: vhostnet1-d5eee4ec-dd7c-4e.201 Vlan(o/i)(,S): 201/201 Parent:vif0/2 MTU:
1514
            Type:Virtual(Vlan) HWaddr:02:00:00:84:dc:42
            DDP: OFF SwLB: ON
            Vrf:0 Flags:L2 QOS:-1 Ref:1
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:208  bytes:17071 errors:0
            Drops:0
```

Note that the interface type is `Virtual` and the Vlan ID set to `201`. The parent interface is `vif0/2`. The VRF is always 0 for L2 sub-interfaces.

L3 VPN Interface Configuration Example

SUMMARY

Read this topic to learn how to add a user pod with a `virtio` and `kernel` interfaces attached to an L3 VPN instance on the cloud-native router.

IN THIS SECTION

- Overview | **104**
- Configuration Example | **105**

Overview

You can configure a user pod with a `virtio` and `kernel` interfaces to an L3 VPN instance on the cloud-native router. The Juniper Cloud-Native Router must have an L3 interface configured at the time of deployment. Your high-level tasks are:

- Define and apply a network attachment definition (NAD)—The NAD file defines the required configuration for Multus to invoke the JCNR-CNI and create a network to attach the pod interface to.

- Define and apply a pod YAML file to your cloud-native router cluster—The pod YAML contains the pod specifications and an annotation to the network created by the JCNR-CNI.

> **NOTE**: Please review the "JCNR Use-Cases and Configuration Overview " on page 84 topic for more information on NAD and pod YAML files.

Configuration Example

1. Here is an example NAD to create a `virtio` interface attached to an L3 VPN instance:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vrf100
spec:
  config: '{
    "cniVersion":"0.4.0",
    "name": "vrf100",
    "plugins": [
      {
        "type": "jcnr",
        "args": {
          "instanceName": "vrf100",
          "instanceType": "vrf",
          "vrfTarget":"100:1"
        },
        "ipam": {
          "type": "static",
          "addresses":[
            {
              "address":"99.61.0.2/16",
              "gateway":"99.61.0.1"
            },
            {
              "address":"1234::99.61.0.2/120",
              "gateway":"1234::99.61.0.1"
            }
          ]
        },
        "kubeConfig":"/etc/kubernetes/kubelet.conf"
      }
    ]
  }'
```

The NAD defines a virtual routing and forwarding (VRF) instance `vrf100` to which the pod's virtio interface will be attached. You must use the `vrf` instance type for Layer 3 VPN implementations. The NAD also defines a static IP address to be assigned to the pod interface.

2. Apply the NAD manifest to create the network.

```
kubectl apply -f nad_virtio_L3vpn.yaml
networkattachmentdefinition.k8s.cni.cncf.io/vrf100 created
```

3. Here is an example NAD to create a `kernel` interface attached to an L3VPN instance:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vrf200
spec:
  config: '{
    "cniVersion":"0.4.0",
    "name": "vrf200",
    "plugins": [
      {
        "type": "jcnr",
        "args": {
          "instanceName": "vrf200",
          "instanceType": "vrf",
       "interfaceType": "veth",
          "vrfTarget":"200:1"
        },
        "ipam": {
          "type": "static",
          "addresses":[
            {
              "address":"99.62.0.2/16",
              "gateway":"99.62.0.1"
            },
            {
              "address":"1234::99.62.0.2/120",
              "gateway":"1234::99.62.0.1"
            }
          ]
        },
        "kubeConfig":"/etc/kubernetes/kubelet.conf"
      }
    ]
  }'
```

The NAD defines a virtual routing and forwarding (VRF) instance `vrf200` with a `veth` interface type to which the pod's kernel interface will be attached.

It also defines a static IP address to be assigned to the pod interface.

4. Apply the NAD manifest to create the network.

```
kubectl apply -f nad_kernel_L3vpn.yaml
networkattachmentdefinition.k8s.cni.cncf.io/vrf200 created
```

5. Verify the NADs are created.

```
[root@jcnr-01]# kubectl get net-attach-def
NAME                     AGE
vrf100                   8m40s
vrf200                   55s
```

6. Here is an example yaml to create a pod attached to the `vrf100` and `vrf200` networks:

```
apiVersion: v1
kind: Pod
metadata:
  name:   pod1
  annotations:
    k8s.v1.cni.cncf.io/networks: vrf100, vrf200
spec:
  containers:
    - name: pod1
      image: ubuntu:latest
      imagePullPolicy: IfNotPresent
      securityContext:
        privileged: false
      env:
        - name: KUBERNETES_POD_UID
          valueFrom:
            fieldRef:
               fieldPath: metadata.uid
      volumeMounts:
        - name: dpdk
          mountPath: /dpdk
          subPathExpr: $(KUBERNETES_POD_UID)
  volumes:
    - name: dpdk
      hostPath:
        path: /var/run/jcnr/containers
```

The pod attaches to the router instance using the `k8s.v1.cni.cncf.io/networks` annotation.

7. Apply the pod manifest.

```
[root@jcnr-01]# kubectl apply -f pod_access_mode.yaml
pod/pod1 created
```

8. Verify the pod is running.

```
[root@jcnr-01 ~]# kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
pod1   1/1     Running   0          2m38s
```

9. Describe the pod to verify two secondary interface are created and attached to the `vrf100` and `vrf200` networks. (The output is trimmed for brevity).

```
[root@jcnr-01 ~]# kubectl describe pod pod1
Name:         pod1
Namespace:    default
Priority:     0
```

```
Node:         jcnr-01/10.100.20.25
Start Time:   Mon, 26 Jun 2023 09:53:31 -0400
Labels:       <none>
Annotations:  cni.projectcalico.org/containerID:
6705c204abca5aeaa0241c1791ea911d57bd972336d969ac5d6a482c96348d95
              cni.projectcalico.org/podIP: 10.233.91.100/32
              cni.projectcalico.org/podIPs: 10.233.91.100/32
              jcnr.juniper.net/dpdk-interfaces:
                [
                    {
                        "name": "net1",
                        "vhost-adaptor-path": "/dpdk/vhost-net1.sock",
                        "vhost-adaptor-mode": "client",
                        "ipv4-address": "99.61.0.2/16",
                        "ipv6-address": "1234::633d:2/120",
                        "mac-address": "02:00:00:A9:B3:23"
                    }
                ]
              k8s.v1.cni.cncf.io/network-status:
                [{
                    "name": "k8s-pod-network",
                    "ips": [
                        "10.233.91.100"
                    ],
                    "default": true,
                    "dns": {}
                },{
                    "name": "default/vrf100",
                    "interface": "net1",
                    "ips": [
                        "99.61.0.2",
                        "1234::633d:2"
                    ],
                    "mac": "02:00:00:A9:B3:23",
                    "dns": {}
                },{
                    "name": "default/vrf200",
                    "interface": "net2",
                    "ips": [
                        "99.62.0.2",
                        "1234::633e:2"
                    ],
                    "mac": "02:00:00:E0:AC:59",
                    "dns": {}
                }]
...
```

10. Verify the vRouter has the corresponding interface created. "Access the vRouter CLI" on page 168 and issue the `vif --list` command.

```
vif0/5      PMD: vhostnet1-2464783d-1ddd-4bf5-b7 NH: 16 MTU: 9160
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:99.61.0.2
            IP6addr:1234::633d:2
            DDP: OFF SwLB: ON
            Vrf:1 Mcast Vrf:1 Flags:PL3DProxyEr QOS:-1 Ref:14
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:0

vif0/6      Ethernet: jvknet2-2464783 NH: 19 MTU: 9160
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:99.62.0.2
            IP6addr:1234::633e:2
            DDP: OFF SwLB: ON
            Vrf:2 Mcast Vrf:2 Flags:PL3DVofProxyEr QOS:-1 Ref:11
            RX port   packets:28 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:28  bytes:13612 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:28
```

Note that the interface type is `Virtual` and the type of interface is L3. You can see the IP addresses assigned to the interfaces for the corresponding valid VRF numbers.

L3 VLAN Sub-Interface Configuration Example

SUMMARY

Read this topic to learn how to add a user pod with a Layer 3 VLAN sub-interface to an instance of the cloud-native router.

IN THIS SECTION

- Overview | 111
- Configuration Example | 112

Overview

You can configure a user pod with a Layer 3 VLAN sub-interface and attach it to the JCNR instance. The Juniper Cloud-Native Router must have an L3 interface configured at the time of deployment. The cRPD must be configured with the valid VLAN configuration for the fabric interface. For example:

```
set interfaces ens1f1v1 unit 201 vlan-id 201
set interfaces ens1f1v1 unit 201 family inet address 192.168.123.1/24
set interfaces ens1f1v1 unit 201 family inet6 address abcd:192:168:123::1/64
set routing-instance blue interface ens1f1v1.201
```

Your high-level tasks are:

- Define and apply a network attachment definition (NAD)—The NAD file defines the required configuration for Multus to invoke the JCNR-CNI and create a network to attach the pod interface to.

- Define and apply a pod YAML file to your cloud-native router cluster—The pod YAML contains the pod specifications and an annotation to the network created by the JCNR-CNI

> **NOTE**: Please review the "JCNR Use-Cases and Configuration Overview " on page 84 topic for more information on NAD and pod YAML files.

Configuration Example

1. Here are example NADs to create a Layer 3 VLAN sub-interface:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vrf201
spec:
  config: '{
    "cniVersion":"0.4.0",
    "name": "vrf201",
    "plugins": [
      {
        "type": "jcnr",
        "args": {
          "instanceName": "vrf201",
          "instanceType": "virtual-router",
          "parentInterface":"net1",
          "vlanId": "201"
        },
        "ipam": {
          "type": "static",
          "addresses":[
            {
              "address":"99.61.0.2/16",
              "gateway":"99.61.0.1"
            },
            {
              "address":"1234::99.61.0.2/120",
              "gateway":"1234::99.61.0.1"
            }
          ]
        },
        "kubeConfig":"/etc/kubernetes/kubelet.conf"
      }
    ]
  }'
```

The NAD defines virtual-router instances `vrf201` with the parent interface `net1` and VLAN ID `201`. A `virtual-router` instance type is similar to a VPN routing and forwarding instance type, but used for non-VPN-related applications. There are no virtual routing and forwarding (VRF) import, VRF export, VRF target, or route distinguisher requirements for this instance type. The pod VLAN sub-interface is attached to vrf201 instance. The NAD also defines static IP addresses to be assigned to the pod interface.

2. Apply the NAD manifests to create the networks.

```
kubectl apply -f nad_l3_vlan_subinterface_201.yaml
networkattachmentdefinition.k8s.cni.cncf.io/vrf201 created
```

3. Verify the NADs are created.

```
kubectl get net-attach-def
NAME      AGE
vrf201    30s
```

4. Here is an example yaml to create a pod attached to the vrf201 and vrf202 networks:

```
apiVersion: v1
kind: Pod
metadata:
  name:   pod1
  annotations:
    k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "vrf201",
          "interface":"net1.201"
        }
      ]
spec:
  containers:
    - name: pod1
      image: ubuntu:latest
      imagePullPolicy: IfNotPresent
      securityContext:
        privileged: false
      env:
        - name: KUBERNETES_POD_UID
          valueFrom:
            fieldRef:
               fieldPath: metadata.uid
      volumeMounts:
        - name: dpdk
          mountPath: /dpdk
          subPathExpr: $(KUBERNETES_POD_UID)
  volumes:
    - name: dpdk
      hostPath:
        path: /var/run/jcnr/containers
```

The pod attaches to the router instances using the `k8s.v1.cni.cncf.io/networks` annotation.

5. Apply the pod manifest.

```
[root@jcnr-01]# kubectl apply -f pod_l3_subinterface.yaml
pod/pod1 created
```

6. Verify the pod is running.

```
[root@jcnr-01 ~]# kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
pod1   1/1     Running   0          38s
```

7. Describe the pod to verify a secondary interface is created and attached to the `vrf201` network. (The output is trimmed for brevity).

```
[root@jcnr-01 ~]# kubectl describe pod pod1
Name:         pod1
Namespace:    default
Priority:     0
Node:         jcnr-01/10.100.20.25
Start Time:   Mon, 26 Jun 2023 09:53:31 -0400
Labels:       <none>
Annotations:  cni.projectcalico.org/containerID:
90de252886b3e0a97526ac175544078fb03debf05650946d759e2de0d5179c17
              cni.projectcalico.org/podIP: 10.233.91.126/32
              cni.projectcalico.org/podIPs: 10.233.91.126/32
              jcnr.juniper.net/dpdk-interfaces:
                [
                    {
                        "name": "net1.201",
                        "vhost-adaptor-path": "/dpdk/vhost-net1.sock",
                        "vhost-adaptor-mode": "client",
                        "ipv4-address": "99.61.0.2/16",
                        "ipv6-address": "1234::633d:2/120",
                        "mac-address": "02:00:00:8C:97:A2",
                        "vlan-id": "201"
                    }
                ]
              k8s.v1.cni.cncf.io/network-status:
                [{
                    "name": "k8s-pod-network",
                    "ips": [
                        "10.233.91.126"
                    ],
                    "default": true,
                    "dns": {}
                },{
                    "name": "default/vrf201",
                    "interface": "net1.201",
                    "ips": [
                        "99.61.0.2",
                        "1234::633d:2"
                    ],
                    "mac": "02:00:00:8C:97:A2",
                    "dns": {}
                }]
...
```

8. Verify the vRouter has the corresponding interface created. "Access the vRouter CLI" on page 152 and issue the `vif --list` command.

```
vif0/11     PCI: 0000:b3:11.1 (Speed 10000, Duplex 1) NH: 16 MTU: 9014            ---> fabric
interface
            Type:Physical HWaddr:b2:56:78:5c:af:fa IPaddr:0.0.0.0
            DDP: OFF SwLB: ON
            Vrf:0 Mcast Vrf:0 Flags:L3L2Vof QOS:0 Ref:42
            RX port   packets:10988509 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: 0000:b3:11.1  Status: UP  Driver: net_iavf
            RX packets:10988509  bytes:5582067106 errors:0
            TX packets:10988484  bytes:5581953776 errors:0
            Drops:0
            TX port   packets:10988484 errors:0

vif0/17     PMD: ens1f1v1 NH: 44 MTU: 9000                                       ---> tap
interface
            Type:Host HWaddr:b2:56:78:5c:af:fa IPaddr:0.0.0.0
            DDP: OFF SwLB: ON
            Vrf:0 Mcast Vrf:0 Flags:L3L2 QOS:0 Ref:41 TxXVif:11
            RX device packets:2201  bytes:935980 errors:0
            RX queue  packets:2201 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:2201  bytes:935980 errors:0
            TX packets:493  bytes:161906 errors:0
            Drops:0
            TX queue  packets:493 errors:0
            TX device packets:493  bytes:161906 errors:0

vif0/48     Virtual: ens1f1v1.201 Vlan(o/i)(,S): 201/201 NH: 161 MTU: 1514
            Parent:vif0/11  Sub-type:  physical-tap                            ---> L3 sub-
interface, parent is a physical interface
            Type:Virtual(Vlan) HWaddr:b2:56:78:5c:af:fa IPaddr:192.168.123.1
            IP6addr:abcd:192:168:123::1
            DDP: OFF SwLB: ON
            Vrf:201 Mcast Vrf:201 Flags:L3DProxyEr QOS:-1 Ref:4
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:18  bytes:1836 errors:0
            Drops:0

vif0/49     Virtual: ens1f1v1.201 Vlan(o/i)(,S): 201/201 NH: 156 MTU: 9000
            Parent:vif0/17  Sub-type:  Host-tap                                ---> L3 sub-
interface, parent is a tap interface
            Type:Virtual(Vlan) HWaddr:b2:56:78:5c:af:fa IPaddr:192.168.123.1
            IP6addr:abcd:192:168:123::1
            DDP: OFF SwLB: ON
            Vrf:201 Mcast Vrf:65535 Flags:L3DProxyEr QOS:-1 Ref:4 TxXVif:48
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:18  bytes:1908 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:0

vif0/50     PMD: vhostnet1-9403fd77-648a-47 NH: 177 MTU: 9160                     ---> pod
```

```
interface
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:0.0.0.0
            DDP: OFF SwLB: ON
            Vrf:65535 Mcast Vrf:65535 Flags:L3DProxyEr QOS:-1 Ref:20
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:0

vif0/51     Virtual: vhostnet1-9403fd77-648a-47.202 Vlan(o/i)(,S): 202/202 NH: 17 MTU: 1514
            Parent:vif0/50                                                         ---->L3 pod
sub-interface, parent is the pod interface
            Type:Virtual(Vlan) HWaddr:00:00:5e:00:01:00 IPaddr:99.62.0.2
            IP6addr:1234::633e:2
            DDP: OFF SwLB: ON
            Vrf:2 Mcast Vrf:2 Flags:PL3DProxyEr QOS:-1 Ref:4
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:0  bytes:0 errors:0
            Drops:0
```

You can see the IP addresses assigned to the sub-interfaces for the corresponding valid VRF numbers.

6

CHAPTER

Monitoring and Logging

Monitor JCNR via CLI | 119

Telemetry Capabilities of Cloud-Native Router | 126

Logging and Notifications | 147

Monitor JCNR via CLI

SUMMARY

This topic contains instructions to access the JCNR controller (cRPD) CLI and run operational commands.

IN THIS SECTION

- Accessing the JCNR Controller (cRPD) CLI | 119
- Example Show Commands | 121
- Example Clear Commands | 126

Accessing the JCNR Controller (cRPD) CLI

You can access the command-line interface (CLI) of the cloud-native router controller by accessing the shell of the running cRPD container.

NOTE: The commands below are provided as an example. The cRPD pod name must be replaced from your environment. The command outputs may differ based on your environment.

List the K8s Pods Running in the Cluster

```
kubectl get pods -A
```

```
NAMESPACE        NAME                                       READY   STATUS     RESTARTS
AGE
contrail-deploy  contrail-k8s-deployer-7b5dd699b9-nd7xf     1/1     Running    0
41m
contrail         contrail-vrouter-masters-dfxgm             3/3     Running    0
41m
jcnr             kube-crpd-worker-ds-8tnf7                  1/1     Running    0
41m
jcnr             syslog-ng-54749b7b77-v24hq                 1/1     Running    0
41m
kube-system      calico-kube-controllers-57b9767bdb-5wbj6   1/1     Running    2 (92d ago)
129d
```

```
kube-system       calico-node-j4m5b                          1/1     Running     2 (92d ago)
129d
kube-system       coredns-8474476ff8-fpw78                   1/1     Running     2 (92d ago)
129d
kube-system       dns-autoscaler-7f76f4dd6-q5vdp             1/1     Running     2 (92d ago)
129d
kube-system       kube-apiserver-5a5s5-node2                 1/1     Running     3 (92d ago)
129d
kube-system       kube-controller-manager-5a5s5-node2        1/1     Running     4 (92d ago)
129d
kube-system       kube-multus-ds-amd64-4zm5k                 1/1     Running     2 (92d ago)
129d
kube-system       kube-proxy-l6xm8                           1/1     Running     2 (92d ago)
129d
kube-system       kube-scheduler-5a5s5-node2                 1/1     Running     4 (92d ago)
129d
kube-system       nodelocaldns-6kwg5                         1/1     Running     2 (92d ago)
129d
```

Copy the name of the cRPD pod—`kube-crpd-worker-ds-8tnf7` in this example output . You will use the pod name to connect to the running container's shell.

Connect to the cRPD CLI

Issue the `kubectl exec` command to access the running container's shell:

```
kubectl exec -n <namespace> -it <pod name> --container <container name> -- bash
```

where *<namespace>* identifies the namespace in which the pod is running, *<pod name>* specificies the name of the pod and the *<container name>* specifies the name of the container (to be specified if the pod has more than one container).

The cRPD pod has only one running container. Here is an example command:

```
kubectl exec -n jcnr -it kube-crpd-worker-ds-8tnf7 -- bash
```

The result of the above command should appear similar to:

```
Defaulted container "kube-crpd-worker" out of: kube-crpd-worker, jcnr-crpd-config (init),
install-cni (init)

===>
```

```
           Containerized Routing Protocols Daemon (CRPD)
 Copyright (C) 2020-2022, Juniper Networks, Inc. All rights reserved.
                                                                      <===

root@jcnr-01:/#
```

At this point, you have connected to the shell of the cRPD. Just as with other Junos-based shells, you access the operational mode of the cloud-native router the same way as if you were connected to the console of a physical Junos OS device.

```
root@jcnr-01:/# cli
root@jcnr-cni>
```

Example Show Commands

Here are some example show commands you can execute:

```
show interfaces terse
Interface@link   Oper State     Addresses
__crpd-brd1      UNKNOWN        fe80::acbf:beff:fe8a:e046/64
cali1b684d67bd4@if3 UP             fe80::ecee:eeff:feee:eeee/64
cali34cf41e29bb@if3 UP             fe80::ecee:eeff:feee:eeee/64
docker0          DOWN           172.17.0.1/16
eno1             UP             10.102.70.146/24 fe80::a94:efff:fe79:dcae/64
eno2             UP
eno3             UP             10.1.1.1/24 fe80::a94:efff:fe79:dcac/64
eno3v1           UP
eno4             DOWN
enp0s20f0u1u6    UNKNOWN
ens2f0           DOWN
ens2f1           DOWN
erspan0@NONE     DOWN
eth0             UNKNOWN        169.254.143.126/32 fe80::b4db:eeff:fe78:9f43/64
gre0@NONE        UNKNOWN
gretap0@NONE     DOWN
ip6tnl0@NONE     UNKNOWN        fe80::74b6:2cff:fea7:d850/64
irb              DOWN
kube-ipvs0       DOWN           10.233.0.1/32 10.233.0.3/32 10.233.35.229/32
lo               UNKNOWN        127.0.0.1/8 ::1/128
lsi              UNKNOWN        fe80::cc59:6dff:fe9c:4db3/64
```

```
nodelocaldns     DOWN           169.254.25.10/32
sit0@NONE
UNKNOWN        ::169.254.143.126/96 ::10.233.91.64/96 ::172.17.0.1/96 ::10.102.70.146/96 ::10.1.1
.1/96 ::127.0.0.1/96
tunl0@NONE       UNKNOWN
vxlan.calico     UNKNOWN        10.233.91.64/32 fe80::64c6:34ff:fecd:3522/64
```

```
show configuration routing-instances
vswitch {
    instance-type virtual-switch;
    bridge-domains {
        bd100 {
            vlan-id 100;
        }
        bd200 {
            vlan-id 200;
        }
        bd300 {
            vlan-id 300;
        }
        bd700 {
            vlan-id 700;
            interface enp59s0f1v0;
        }
        bd701 {
            vlan-id 701;
        }
        bd702 {
            vlan-id 702;
        }
        bd703 {
            vlan-id 703;
        }
        bd704 {
            vlan-id 704;
        }
        bd705 {
            vlan-id 705;
        }
    }
    interface bond0;
}
```

```
show bridge ?
Possible completions:
mac-table       Show media access control table
statistics      Show bridge statistics information
```

```
show bridge mac-table ?
Possible completions:
  <[Enter]>            Execute this command
  count                Number of MAC address
  mac-address          MAC address in the format XX:XX:XX:XX:XX:XX
  vlan-id              Display MAC address learned on a specified VLAN or 'all-vlan'
  |                    Pipe through a command
```

```
show bridge mac-table
Routing Instance : default-domain:default-project:ip-fabric:__default__
Bridging domain VLAN id : 3002
MAC                  MAC                Logical
address              flags              interface

00:00:5E:00:53:01      D                 bond0
```

```
show bridge statistics ?
Possible completions:
  <[Enter]>            Execute this command
  vlan-id              Display statistics for a particular vlan (1..4094)
  |                    Pipe through a command
```

```
show bridge statistics
Bridge domain vlan-id: 100
   Local interface:  bond0
      Broadcast packets Tx  : 0           Rx  : 0
      Multicast packets Tx  : 0           Rx  : 0
      Unicast packets Tx    : 0           Rx  : 0
      Broadcast bytes Tx    : 0           Rx  : 0
      Multicast bytes Tx    : 0           Rx  : 0
      Unicast bytes Tx      : 0           Rx  : 0
      Flooded packets       : 0
      Flooded bytes         : 0
   Local interface: ens1f0v1
      Broadcast packets Tx  : 0           Rx  : 0
      Multicast packets Tx  : 0           Rx  : 0
      Unicast packets Tx    : 0           Rx  : 0
      Broadcast bytes Tx    : 0           Rx  : 0
      Multicast bytes Tx    : 0           Rx  : 0
      Unicast bytes Tx      : 0           Rx  : 0
      Flooded packets       : 0
      Flooded bytes         : 0
   Local interface: ens1f3v1
      Broadcast packets Tx  : 0           Rx  : 0
      Multicast packets Tx  : 0           Rx  : 0
      Unicast packets Tx    : 0           Rx  : 0
      Broadcast bytes Tx    : 0           Rx  : 0
      Multicast bytes Tx    : 0           Rx  : 0
      Unicast bytes Tx      : 0           Rx  : 0
      Flooded packets       : 0
```

```
show firewall filter filter1
Filter : filter1    vlan-id : 3001
 Term                  Packet
  t1                   0
```

```
show configuration firewall:firewall
family {
    bridge {
        filter filter1 {
            term t1 {
                from {
                    destination-mac-address 10:30:30:30:30:31;
                    source-mac-address 10:30:30:30:30:30;
                    ether-type oam;
                }
                then {
                    discard;
                }
            }
        }
    }
}
```

```
show route 172.68.20.2/32 table nad1.inet
nad1.inet.0: 11 destinations, 15 routes (11 active, 0 holddown, 0 hidden)
@ = Routing Use Only, # = Forwarding Use Only
+ = Active Route, - = Last Active, * = Both

172.68.20.2/32     @[BGP/170] 00:00:23, localpref 100, from 1.1.1.220
                      AS path: I, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 1.1.1.35 dest 1.1.1.220), Push 48
                    [BGP/170] 00:13:18, localpref 100, from 1.1.24.24
                      AS path: I, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 1.1.1.35 dest 1.1.24.24), Push 16
                   #[Multipath/255] 00:00:23, metric2 2
                       via Tunnel Composite, UDP (src 1.1.1.35 dest 1.1.1.220), Push 48
                    >  via Tunnel Composite, UDP (src 1.1.1.35 dest 1.1.24.24), Push 16
```

```
show interfaces routing enp216s0f0
Interface        State Addresses
enp216s0f0       Up    MPLS  enabled
                       ISO   enabled
                       INET  192.168.123.3
                       INET6 2001:192:168:123::3
                       INET6 fe80::42a6:b7ff:fe2c:a448
```

```
show dynamic-tunnels database
*- Signal Tunnels #- PFE-down
Table: inet.3
Destination-network: 1.1.1.220/32
Destination-network: 1.1.24.24/32
Tunnel to: 1.1.24.24/32
  Reference count: 4
  Next-hop type: UDP (forwarding-nexthop)
    Source address: 1.1.1.35
    Next hop: v6 mapped, tunnel-composite, 0x557917afc91c, nhid 0
```

```
      VPN Label: Push 16, Reference count: 2
      Ingress Route: [OSPF] 1.1.24.24/32, via metric 2
      Traffic Statistics: Packets 0, Bytes 0
      State: Up
  Aggregate Traffic Statistics:
```

Example Clear Commands

Here are some example clear commands:

```
clear bridge mac-table ?
Possible completions:
  <[Enter]>            Execute this command
  mac-address          Clear specific MAC address
  vlan-id              Clear mac-table for a specified vlan-id (1..4094)
  |                    Pipe through a command
```

```
clear bridge statistics ?
Possible completions:
  <[Enter]>            Execute this command
  vlan-id              Clear L2 interface statistics for a specified vlan-id (1..4094)
  |                    Pipe through a command
```

Telemetry Capabilities of Cloud-Native Router

IN THIS SECTION

- JCNR Telemetry | **127**

Read this topic to learn about the telemetry data available from Juniper Cloud-Native Router.

JCNR Telemetry

Juniper Cloud-Native Router comes with telemetry capabilities that enable you to see performance metrics and telemetry data. Telemetry data is derived separately from the vRouter and the cRPD. For vRouter, the container **contrail-vrouter-telemetry-exporter** provides you this visibility. This container runs alongside the other vRouter containers in the **contrail-vrouter-masters** pod.

For vRouter, the telemetry exporter periodically queries the Introspect on the vRouter-agent for statistics and reports metrics information in response to the Prometheus scrape requests. You can directly view the telemetry data by using the following URL: **http://*host server IP address*:8070**. The following table shows the sample output.

NOTE: We've grouped the output shown in the following table. The cloud-native router does not group or sort the output on live systems.

Table 3: Sample vRouter Telemetry Output

Group	Sample Output
Memory usage per vRouter	# TYPE virtual_router_system_memory_cached_bytes gauge # HELP virtual_router_system_memory_cached_bytes Virtual router system memory cached virtual_router_system_memory_cached_bytes{vrouter_name="jcnr.example.com"} 2635970448 # TYPE virtual_router_system_memory_buffers gauge # HELP virtual_router_system_memory_buffers Virtual router system memory buffer virtual_router_system_memory_buffers{vrouter_name="jcnr.example.com"} 32689 # TYPE virtual_router_system_memory_bytes gauge # HELP virtual_router_system_memory_bytes Virtual router total system memory virtual_router_system_memory_bytes{vrouter_name="jcnr.example.com"} 2635970448 # TYPE virtual_router_system_memory_free_bytes gauge # HELP virtual_router_system_memory_free_bytes Virtual router system memory free virtual_router_system_memory_free_bytes{vrouter_name="jcnr.example.com"} 2635969296 # TYPE virtual_router_system_memory_used_bytes gauge # HELP virtual_router_system_memory_used_bytes Virtual router system memory used virtual_router_system_memory_used_bytes{vrouter_name="jcnr.example.com"} 32689 # TYPE virtual_router_virtual_memory_kilobytes gauge # HELP virtual_router_virtual_memory_kilobytes Virtual router virtual memory virtual_router_virtual_memory_kilobytes{vrouter_name="jcnr.example.com"} 0 # TYPE virtual_router_resident_memory_kilobytes gauge # HELP virtual_router_resident_memory_kilobytes Virtual router resident memory virtual_router_resident_memory_kilobytes{vrouter_name="jcnr.example.com"} 32689 # TYPE virtual_router_peak_virtual_memory_bytes gauge # HELP virtual_router_peak_virtual_memory_bytes Virtual router peak virtual memory virtual_router_peak_virtual_memory_bytes{vrouter_name="jcnr.example.com"} 2894328001

Table 3: Sample vRouter Telemetry Output *(Continued)*

Group	Sample Output
Packet count per interface	# TYPE virtual_router_phys_if_input_packets_total counter # HELP virtual_router_phys_if_input_packets_total Total packets received by physical interface virtual_router_phys_if_input_packets_total{vrouter_name="jcnr.example.com",interface_name="bond0"} 1483 # TYPE virtual_router_phys_if_output_packets_total counter # HELP virtual_router_phys_if_output_packets_total Total packets sent by physical interface virtual_router_phys_if_output_packets_total{vrouter_name="jcnr.example.com",interface_name="bond0"} 32969 # TYPE virtual_router_phys_if_input_bytes_total counter # HELP virtual_router_phys_if_input_bytes_total Total bytes received by physical interface virtual_router_phys_if_input_bytes_total{interface_name="bond0",vrouter_name="jcnr.example.com"} 125558 # TYPE virtual_router_phys_if_output_bytes_total counter # HELP virtual_router_phys_if_output_bytes_total Total bytes sent by physical interface virtual_router_phys_if_output_bytes_total{vrouter_name="jcnr.example.com",interface_name="bond0"} 4597076 virtual_router_phys_if_input_bytes_total{vrouter_name="jcnr.example.com",interface_name="bond0"} 228300499320 virtual_router_phys_if_output_bytes_total{interface_name="bond0",vrouter_name="jcnr.example.com"} 228297889634 virtual_router_phys_if_input_packets_total{interface_name="bond0",vrouter_name="jcnr.example.com"} 1585421179 virtual_router_phys_if_output_packets_total{vrouter_name="jcnr.example.com",interface_name="bond0"} 1585402623 virtual_router_phys_if_output_packets_total{interface_name="bond0",vrouter_name="jcnr.example.com"} 1585403344

Table 3: Sample vRouter Telemetry Output *(Continued)*

Group	Sample Output
CPU usage per vRouter	# TYPE virtual_router_cpu_1min_load_avg gauge # HELP virtual_router_cpu_1min_load_avg Virtual router CPU 1 minute load average virtual_router_cpu_1min_load_avg{vrouter_name="jcnr.example.com"} 0.11625 # TYPE virtual_router_cpu_5min_load_avg gauge # HELP virtual_router_cpu_5min_load_avg Virtual router CPU 5 minute load average virtual_router_cpu_5min_load_avg{vrouter_name="jcnr.example.com"} 0.109687 # TYPE virtual_router_cpu_15min_load_avg gauge # HELP virtual_router_cpu_15min_load_avg Virtual router CPU 15 minute load average virtual_router_cpu_15min_load_avg{vrouter_name="jcnr.example.com"} 0.110156
Drop packet count per vRouter	# TYPE virtual_router_dropped_packets_total counter # HELP virtual_router_dropped_packets_total Total packets dropped virtual_router_dropped_packets_total{vrouter_name="jcnr.example.com"} 35850

Table 3: Sample vRouter Telemetry Output *(Continued)*

Group	Sample Output
Packet count per interface per VLAN	# TYPE virtual_router_interface_vlan_multicast_input_packets_total counter # HELP virtual_router_interface_vlan_multicast_input_packets_total Total number of multicast packets received on interface VLAN virtual_router_interface_vlan_multicast_input_packets_total{interface_id="1",vlan_id="100"} 0 # TYPE virtual_router_interface_vlan_broadcast_output_packets_total counter # HELP virtual_router_interface_vlan_broadcast_output_packets_total Total number of broadcast packets sent on interface VLAN virtual_router_interface_vlan_broadcast_output_packets_total{interface_id="1",vlan_id="100"} 0 # TYPE virtual_router_interface_vlan_broadcast_input_packets_total counter # HELP virtual_router_interface_vlan_broadcast_input_packets_total Total number of broadcast packets received on interface VLAN virtual_router_interface_vlan_broadcast_input_packets_total{interface_id="1",vlan_id="100"} 0 # TYPE virtual_router_interface_vlan_multicast_output_packets_total counter # HELP virtual_router_interface_vlan_multicast_output_packets_total Total number of multicast packets sent on interface VLAN virtual_router_interface_vlan_multicast_output_packets_total{interface_id="1",vlan_id="100"} 0 # TYPE virtual_router_interface_vlan_unicast_input_packets_total counter # HELP virtual_router_interface_vlan_unicast_input_packets_total Total number of unicast packets received on interface VLAN virtual_router_interface_vlan_unicast_input_packets_total{interface_id="1",vlan_id="100"} 0 # TYPE virtual_router_interface_vlan_flooded_output_bytes_total counter # HELP virtual_router_interface_vlan_flooded_output_bytes_total Total number of output bytes flooded to interface VLAN virtual_router_interface_vlan_flooded_output_bytes_total{interface_id="1",vlan_id="100"} 0 # TYPE virtual_router_interface_vlan_multicast_output_bytes_total counter # HELP virtual_router_interface_vlan_multicast_output_bytes_total Total number of multicast bytes sent on interface VLAN virtual_router_interface_vlan_multicast_output_bytes_total{interface_id="1",vlan_id="100"} 0 # TYPE virtual_router_interface_vlan_unicast_output_packets_total counter # HELP virtual_router_interface_vlan_unicast_output_packets_total Total number of unicast packets sent on interface VLAN virtual_router_interface_vlan_unicast_output_packets_total{interface_id="1",vlan_id="100"} 0 # TYPE virtual_router_interface_vlan_broadcast_input_bytes_total counter

Table 3: Sample vRouter Telemetry Output *(Continued)*

Group	Sample Output
	# HELP virtual_router_interface_vlan_broadcast_input_bytes_total Total number of broadcast bytes received on interface VLAN virtual_router_interface_vlan_broadcast_input_bytes_total{interface_id="1",vlan_id="100 "} 0 # TYPE virtual_router_interface_vlan_multicast_input_bytes_total counter # HELP virtual_router_interface_vlan_multicast_input_bytes_total Total number of multicast bytes received on interface VLAN virtual_router_interface_vlan_multicast_input_bytes_total{vlan_id="100",interface_id="1 "} 0 # TYPE virtual_router_interface_vlan_unicast_input_bytes_total counter # HELP virtual_router_interface_vlan_unicast_input_bytes_total Total number of unicast bytes received on interface VLAN virtual_router_interface_vlan_unicast_input_bytes_total{interface_id="1",vlan_id="100"} 0 # TYPE virtual_router_interface_vlan_flooded_output_packets_total counter # HELP virtual_router_interface_vlan_flooded_output_packets_total Total number of output packets flooded to interface VLAN virtual_router_interface_vlan_flooded_output_packets_total{interface_id="1",vlan_id="10 0"} 0 # TYPE virtual_router_interface_vlan_broadcast_output_bytes_total counter # HELP virtual_router_interface_vlan_broadcast_output_bytes_total Total number of broadcast bytes sent on interface VLAN virtual_router_interface_vlan_broadcast_output_bytes_total{interface_id="1",vlan_id="10 0"} 0 # TYPE virtual_router_interface_vlan_unicast_output_bytes_total counter # HELP virtual_router_interface_vlan_unicast_output_bytes_total Total number of unicast bytes sent on interface VLAN virtual_router_interface_vlan_unicast_output_bytes_total{interface_id="1",vlan_id="100" } 0 ...

For cRPD, the telemetry exporter in the cRPD pod is disabled by default. You have to enable the telemetry exporter deployment by specifying the following override parameter in the `helm install` command while installing JCNR.

```
--set jcnr-cni.telemetryExporter.enable=true
```

The cRPD telemetry exporter periodically queries the NETCONF on the cRPD for statistics and reports metrics information in response to the Prometheus scrape requests. You can directly view the telemetry data by using the following URL: **http:// *host server IP address*:8072**.

NOTE: If the 8072 port is unavailable you can choose an alternate port to collect telemetry data by specifying the following override parameter in the helm install command while installing JCNR.

```
--set jcnr-cni.telemetryExporter.metricsPort=<number>
```

The following table shows the sample output.

Table 4: Sample cRPD Telemetry Output

Group	Sample Output
BGP summary	# TYPE crpd_bgp_rib_table_received_prefixes_total counter # HELP crpd_bgp_rib_table_received_prefixes_total Total number of BGP RIB table prefixes received crpd_bgp_rib_table_received_prefixes_total{node="example.juniper.net",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_rib_table_external_prefixes gauge # HELP crpd_bgp_rib_table_external_prefixes Number of BGP RIB table external prefixes crpd_bgp_rib_table_external_prefixes{node="example.juniper.net",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_rib_table_active_external_prefixes gauge # HELP crpd_bgp_rib_table_active_external_prefixes Number of BGP RIB table active external prefixes crpd_bgp_rib_table_active_external_prefixes{node="example.juniper.net",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_rib_table_suppressed_internal_prefixes gauge # HELP crpd_bgp_rib_table_suppressed_internal_prefixes Number of BGP RIB table internal prefixes currently inactive, because of damping or other reasons crpd_bgp_rib_table_suppressed_internal_prefixes{node="example.juniper.net",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_rib_table_prefixes gauge # HELP crpd_bgp_rib_table_prefixes Number of BGP RIB table prefixes crpd_bgp_rib_table_prefixes{node="example.juniper.net",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_rib_table_active_prefixes gauge # HELP crpd_bgp_rib_table_active_prefixes Number of BGP RIB table active prefixes crpd_bgp_rib_table_active_prefixes{table="bgp.l3vpn.0",node="example.juniper.net"} 0 # TYPE crpd_bgp_rib_table_history_prefixes gauge # HELP crpd_bgp_rib_table_history_prefixes Number of BGP RIB table withdrawn prefixes stored locally to keep track of damping history crpd_bgp_rib_table_history_prefixes{node="example.juniper.net",table="bgp.l3vpn.0"} 0

Table 4: Sample cRPD Telemetry Output *(Continued)*

Group	Sample Output
	# TYPE crpd_bgp_rib_table_suppressed_external_prefixes gauge # HELP crpd_bgp_rib_table_suppressed_external_prefixes Number of BGP RIB table external prefixes currently inactive, because of damping or other reasons crpd_bgp_rib_table_suppressed_external_prefixes{node="example.juniper.net",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_rib_table_active_internal_prefixes gauge # HELP crpd_bgp_rib_table_active_internal_prefixes Number of BGP RIB table active internal prefixes crpd_bgp_rib_table_active_internal_prefixes{node="example.juniper.net",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_rib_table_pending_prefixes gauge # HELP crpd_bgp_rib_table_pending_prefixes Number of BGP RIB table prefixes in process by BGP import policy crpd_bgp_rib_table_pending_prefixes{node="example.juniper.net",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_rib_table_accepted_prefixes_total counter # HELP crpd_bgp_rib_table_accepted_prefixes_total Total number of BGP RIB table prefixes accepted crpd_bgp_rib_table_accepted_prefixes_total{node="example.juniper.net",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_rib_table_damped_prefixes gauge # HELP crpd_bgp_rib_table_damped_prefixes Number of BGP RIB table prefixes with a figure of merit greater than zero, but still active because the value has not reached the threshold at which suppression occurs crpd_bgp_rib_table_damped_prefixes{node="example.juniper.net",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_rib_table_accepted_external_prefixes_total counter # HELP crpd_bgp_rib_table_accepted_external_prefixes_total Total number of BGP RIB table external prefixes accepted crpd_bgp_rib_table_accepted_external_prefixes_total{node="example.juniper.net",table="bgp.l3vpn.0"} 0

Table 4: Sample cRPD Telemetry Output *(Continued)*

Group	Sample Output
	# TYPE crpd_bgp_rib_table_internal_prefixes gauge # HELP crpd_bgp_rib_table_internal_prefixes Number of BGP RIB table internal prefixes crpd_bgp_rib_table_internal_prefixes{node="example.juniper.net",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_rib_table_suppressed_prefixes gauge # HELP crpd_bgp_rib_table_suppressed_prefixes Number of BGP RIB table prefixes currently inactive, because of damping or other reasons crpd_bgp_rib_table_suppressed_prefixes{node="example.juniper.net",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_rib_table_accepted_internal_prefixes_total counter # HELP crpd_bgp_rib_table_accepted_internal_prefixes_total Total number of BGP RIB table internal prefixes accepted crpd_bgp_rib_table_accepted_internal_prefixes_total{node="example.juniper.net",table="bgp.l3vpn.0"} 0 crpd_bgp_rib_table_external_prefixes{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_active_external_prefixes{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_suppressed_internal_prefixes{table="bgp.l3vpn-inet6.0",node="example.juniper.net"} 0 crpd_bgp_rib_table_received_prefixes_total{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_active_prefixes{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_history_prefixes{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_suppressed_external_prefixes{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_active_internal_prefixes{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_pending_prefixes{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_prefixes{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_damped_prefixes{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_accepted_external_prefixes_total{n

Table 4: Sample cRPD Telemetry Output *(Continued)*

Group	Sample Output
	ode="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_internal_prefixes{table="bgp.l3vpn-inet6.0",node="example.juniper.net"} 0 crpd_bgp_rib_table_accepted_prefixes_total{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_accepted_internal_prefixes_total{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_suppressed_prefixes{node="example.juniper.net",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_rib_table_received_prefixes_total{node="example.juniper.net",table="bgp.evpn.0"} 0 crpd_bgp_rib_table_external_prefixes{node="example.juniper.net",table="bgp.evpn.0"} 0 crpd_bgp_rib_table_active_external_prefixes{node="example.juniper.net",table="bgp.evpn.0"} 0 crpd_bgp_rib_table_suppressed_internal_prefixes{table="bgp.evpn.0",node="example.juniper.net"} 0 crpd_bgp_rib_table_prefixes{node="example.juniper.net",table="bgp.evpn.0"} 0 crpd_bgp_rib_table_active_prefixes{node="example.juniper.net",table="bgp.evpn.0"} 0 crpd_bgp_rib_table_history_prefixes{node="example.juniper.net",table="bgp.evpn.0"} 0 crpd_bgp_rib_table_suppressed_external_prefixes{table="bgp.evpn.0",node="example.juniper.net"} 0 crpd_bgp_rib_table_active_internal_prefixes{node="example.juniper.net",table="bgp.evpn.0"} 0 crpd_bgp_rib_table_pending_prefixes{node="example.juniper.net",table="bgp.evpn.0"} 0 crpd_bgp_rib_table_accepted_prefixes_total{table="bgp.evpn.0",node="example.juniper.net"} 0 crpd_bgp_rib_table_damped_prefixes{node="example.juniper.net",table="bgp.evpn.0"} 0 crpd_bgp_rib_table_accepted_external_prefixes_total{node="example.juniper.net",table="bgp.evpn.0"} 0 crpd_bgp_rib_table_internal_prefixes{node="example.juniper.net",table="bgp.evpn.0"} 0 crpd_bgp_rib_table_suppressed_prefixes{node="example.juniper.net",table="bgp.evpn.0"} 0 crpd_bgp_rib_table_accepted_internal_prefixes_total{node="example.juniper.net",table="bgp.evpn.0"} 0

Table 4: Sample cRPD Telemetry Output *(Continued)*

Group	Sample Output
	# TYPE crpd_bgp_peer_input_messages_total counter # HELP crpd_bgp_peer_input_messages_total Total number of messages received from BGP peer crpd_bgp_peer_input_messages_total{peer_address="11.11.11.11",peer_as="64512",node="example.juniper.net"} 5 # TYPE crpd_bgp_peer_output_messages_total counter # HELP crpd_bgp_peer_output_messages_total Total number of messages sent to BGP peer crpd_bgp_peer_output_messages_total{node="example.juniper.net",peer_address="11.11.11.11",peer_as="64512"} 4 # TYPE crpd_bgp_peer_route_queue_count gauge # HELP crpd_bgp_peer_route_queue_count Current number of messages that are queued to be sent to BGP peer crpd_bgp_peer_route_queue_count{node="example.juniper.net",peer_address="11.11.11.11",peer_as="64512"} 0 # TYPE crpd_bgp_peer_state gauge # HELP crpd_bgp_peer_state BGP peer state (1=Established, 2=Idle, 3=Connect, 4=Active, 5=OpenSent, 6=OpenConfirm) crpd_bgp_peer_state{node="example.juniper.net",peer_address="11.11.11.11",peer_as="64512"} 1 # TYPE crpd_bgp_peer_flaps_total counter # HELP crpd_bgp_peer_flaps_total Total number of times the BGP peer session has gone down and then come back up crpd_bgp_peer_flaps_total{peer_address="11.11.11.11",peer_as="64512",node="example.juniper.net"} 1 # TYPE crpd_bgp_peer_rib_table_accepted_prefixes_total counter # HELP crpd_bgp_peer_rib_table_accepted_prefixes_total Total number of BGP RIB table active prefixes accepted from BGP peer crpd_bgp_peer_rib_table_accepted_prefixes_total{peer_as="64512",table="bgp.l3vpn.0",node="example.juniper.net",peer_address="11.11.11.11"} 0 # TYPE crpd_bgp_peer_rib_table_suppressed_prefixes gauge # HELP crpd_bgp_peer_rib_table_suppressed_prefixes

Table 4: Sample cRPD Telemetry Output *(Continued)*

Group	Sample Output
	Number of BGP RIB table prefixes received from BGP peer currently inactive, because of damping or other reasons crpd_bgp_peer_rib_table_suppressed_prefixes{node="example.juniper.net",peer_address="11.11.11.11",peer_as="64512",table="bgp.l3vpn.0"} 0 # TYPE crpd_bgp_peer_rib_table_active_prefixes gauge # HELP crpd_bgp_peer_rib_table_active_prefixes Number of BGP RIB table active prefixes received from BGP peer crpd_bgp_peer_rib_table_active_prefixes{node="example.juniper.net",peer_address="11.11.11.11",peer_as="64512",table="bgp.l3vpn.0"} 0 crpd_bgp_peer_rib_table_active_prefixes{node="example.juniper.net",peer_address="11.11.11.11",peer_as="64512",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_peer_rib_table_accepted_prefixes_total{node="example.juniper.net",peer_address="11.11.11.11",peer_as="64512",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_peer_rib_table_suppressed_prefixes{node="example.juniper.net",peer_address="11.11.11.11",peer_as="64512",table="bgp.l3vpn-inet6.0"} 0 crpd_bgp_peer_rib_table_active_prefixes{node="example.juniper.net",peer_address="11.11.11.11",peer_as="64512",table="bgp.evpn.0"} 0 crpd_bgp_peer_rib_table_accepted_prefixes_total{node="example.juniper.net",peer_address="11.11.11.11",peer_as="64512",table="bgp.evpn.0"} 0 crpd_bgp_peer_rib_table_suppressed_prefixes{node="example.juniper.net",peer_address="11.11.11.11",peer_as="64512",table="bgp.evpn.0"} 0

Table 4: Sample cRPD Telemetry Output *(Continued)*

Group	Sample Output
Route table summary	# TYPE crpd_route_table_destinations gauge # HELP crpd_route_table_destinations Number of destinations for which there are routes in the routing table crpd_route_table_destinations{node="example.juniper.net",table="inet.0"} 13 # TYPE crpd_route_table_routes gauge # HELP crpd_route_table_routes Number of routes in the routing table crpd_route_table_routes{node="example.juniper.net",table="inet.0"} 15 # TYPE crpd_route_table_active_routes gauge # HELP crpd_route_table_active_routes Number of active routes in the routing table crpd_route_table_active_routes{node="example.juniper.net",table="inet.0"} 13 # TYPE crpd_route_table_holddown_routes gauge # HELP crpd_route_table_holddown_routes Number of routes in the routing table that are in the hold-down state before being declared inactive crpd_route_table_holddown_routes{node="example.juniper.net",table="inet.0"} 0 # TYPE crpd_route_table_hidden_routes gauge # HELP crpd_route_table_hidden_routes Number of routes in the routing table that are not used, because of routing policy crpd_route_table_hidden_routes{node="example.juniper.net",table="inet.0"} 0 crpd_route_table_routes{node="example.juniper.net",table="inet.3"} 1 crpd_route_table_active_routes{node="example.juniper.net",table="inet.3"} 1 crpd_route_table_holddown_routes{node="example.juniper.net",table="inet.3"} 0 crpd_route_table_hidden_routes{node="example.juniper.net",table="inet.3"} 0 crpd_route_table_destinations{node="example.juniper.net",table="inet.3"} 1 crpd_route_table_holddown_routes{node="example.juniper.net",table="mpls.0"} 0 crpd_route_table_hidden_routes{node="example.juniper.

Table 4: Sample cRPD Telemetry Output *(Continued)*

Group	Sample Output
	net",table="mpls.0"} 0 crpd_route_table_destinations{node="example.juniper.net",table="mpls.0"} 4 crpd_route_table_routes{node="example.juniper.net",table="mpls.0"} 4 crpd_route_table_active_routes{node="example.juniper.net",table="mpls.0"} 4 crpd_route_table_active_routes{node="example.juniper.net",table="inet6.0"} 34 crpd_route_table_holddown_routes{node="example.juniper.net",table="inet6.0"} 0 crpd_route_table_hidden_routes{node="example.juniper.net",table="inet6.0"} 0 crpd_route_table_destinations{node="example.juniper.net",table="inet6.0"} 34 crpd_route_table_routes{table="inet6.0",node="example.juniper.net"} 38 crpd_route_table_destinations{node="example.juniper.net",table="inet6.3"} 1 crpd_route_table_routes{node="example.juniper.net",table="inet6.3"} 1 crpd_route_table_active_routes{node="example.juniper.net",table="inet6.3"} 1 crpd_route_table_holddown_routes{node="example.juniper.net",table="inet6.3"} 0 crpd_route_table_hidden_routes{node="example.juniper.net",table="inet6.3"} 0 # TYPE crpd_route_table_protocol_routes gauge # HELP crpd_route_table_protocol_routes Number of routes in the routing table learned from the protocol crpd_route_table_protocol_routes{protocol="Direct",node="example.juniper.net",table="inet.0"} 6 # TYPE crpd_route_table_protocol_active_routes gauge # HELP crpd_route_table_protocol_active_routes Number of active routes in the routing table learned from the protocol crpd_route_table_protocol_active_routes{protocol="Direct",node="example.juniper.net",table="inet.0"} 6 crpd_route_table_protocol_active_routes{node="example.juniper.net",table="inet.0",protocol="Local"} 3 crpd_route_table_protocol_routes{node="example.juniper.net",table="inet.0",protocol="Local"} 5 crpd_route_table_protocol_routes{node="example.junipe

Table 4: Sample cRPD Telemetry Output *(Continued)*

Group	Sample Output
	r.net",table="inet.0",protocol="OSPF"} 4 crpd_route_table_protocol_active_routes{node="example.juniper.net",table="inet.0",protocol="OSPF"} 4 crpd_route_table_protocol_routes{node="example.juniper.net",table="inet.3",protocol="Tunnel"} 1 crpd_route_table_protocol_active_routes{node="example.juniper.net",table="inet.3",protocol="Tunnel"} 1 crpd_route_table_protocol_routes{node="example.juniper.net",table="mpls.0",protocol="MPLS"} 4 crpd_route_table_protocol_active_routes{node="example.juniper.net",table="mpls.0",protocol="MPLS"} 4 crpd_route_table_protocol_routes{node="example.juniper.net",table="inet6.0",protocol="Direct"} 8 crpd_route_table_protocol_active_routes{node="example.juniper.net",table="inet6.0",protocol="Direct"} 4 crpd_route_table_protocol_routes{table="inet6.0",protocol="Local",node="example.juniper.net"} 29 crpd_route_table_protocol_active_routes{node="example.juniper.net",table="inet6.0",protocol="Local"} 29 crpd_route_table_protocol_routes{node="example.juniper.net",table="inet6.0",protocol="INET6"} 1 crpd_route_table_protocol_active_routes{node="example.juniper.net",table="inet6.0",protocol="INET6"} 1 crpd_route_table_protocol_routes{node="example.juniper.net",table="inet6.3",protocol="Tunnel"} 1 crpd_route_table_protocol_active_routes{node="example.juniper.net",table="inet6.3",protocol="Tunnel"} 1

Table 4: Sample cRPD Telemetry Output *(Continued)*

Group	Sample Output

OSPF summary	`# TYPE crpd_ospf_packets_sent_total counter`
	`# HELP crpd_ospf_packets_sent_total Total number of OSPF packets sent`
	`crpd_ospf_packets_sent_total{node="example.juniper.net",packet_type="Hello"} 26`
	`# TYPE crpd_ospf_packets_received_total counter`
	`# HELP crpd_ospf_packets_received_total Total number of OSPF packets received`
	`crpd_ospf_packets_received_total{node="example.juniper.net",packet_type="Hello"} 4`
	`crpd_ospf_packets_sent_total{node="example.juniper.net",packet_type="DbD"} 3`
	`crpd_ospf_packets_received_total{node="example.juniper.net",packet_type="DbD"} 4`
	`crpd_ospf_packets_sent_total{node="example.juniper.net",packet_type="LSReq"} 1`
	`crpd_ospf_packets_received_total{node="example.juniper.net",packet_type="LSReq"} 1`
	`crpd_ospf_packets_sent_total{node="example.juniper.net",packet_type="LSUpdate"} 2`
	`crpd_ospf_packets_received_total{node="example.juniper.net",packet_type="LSUpdate"} 3`
	`crpd_ospf_packets_sent_total{node="example.juniper.net",packet_type="LSAck"} 3`
	`crpd_ospf_packets_received_total{node="example.juniper.net",packet_type="LSAck"} 2`
	`# TYPE crpd_ospf_dbd_packets_retransmitted_total counter`
	`# HELP crpd_ospf_dbd_packets_retransmitted_total Total number of OSPF database descriptor packets retransmitted`
	`crpd_ospf_dbd_packets_retransmitted_total{node="example.juniper.net"} 1`
	`# TYPE crpd_ospf_lsa_packets_retransmitted_total counter`
	`# HELP crpd_ospf_lsa_packets_retransmitted_total Total number of OSPF link-state advertisement packets retransmitted`
	`crpd_ospf_lsa_packets_retransmitted_total{node="example.juniper.net"} 0`
	`# TYPE crpd_ospf_lsa_packets_flooded_total counter`

Table 4: Sample cRPD Telemetry Output *(Continued)*

Group	Sample Output
	# HELP crpd_ospf_lsa_packets_flooded_total Total number of OSPF link-state advertisement packets flooded crpd_ospf_lsa_packets_flooded_total{node="example.juniper.net"} 1 # TYPE crpd_ospf_flood_queue_depth gauge # HELP crpd_ospf_flood_queue_depth Number of entries in the extended queue crpd_ospf_flood_queue_depth{node="example.juniper.net"} 0 # TYPE crpd_ospf_error_total counter # HELP crpd_ospf_error_total Total number of OSPF receive errors crpd_ospf_error_total{error_type="no-error",node="example.juniper.net"} 0 # TYPE crpd_ospf_neighbor_state gauge # HELP crpd_ospf_neighbor_state OSPF neighbor state (0=Down, 1=Full, 2=Attempt, 3=Exchange, 4=ExStart, 5=Init, 6=Loading, 7=2Way) crpd_ospf_neighbor_state{neighbor_address="113.113.113.3",interface_name="enp6s0",node="example.juniper.net"} 1

Table 4: Sample cRPD Telemetry Output *(Continued)*

Group	Sample Output

MPLS statistics	`# TYPE crpd_mpls_ingress_lsp_sessions_down gauge`
	`# HELP crpd_mpls_ingress_lsp_sessions_down Number of MPLS ingress LSP sessions`
	`crpd_mpls_ingress_lsp_sessions_down{node="example.juniper.net"} 0`
	`# TYPE crpd_mpls_ingress_lsp_sessions gauge`
	`# HELP crpd_mpls_ingress_lsp_sessions Number of MPLS ingress LSP sessions`
	`crpd_mpls_ingress_lsp_sessions{node="example.juniper.net"} 0`
	`# TYPE crpd_mpls_lsp_make_before_breaks_total counter`
	`# HELP crpd_mpls_lsp_make_before_breaks_total Total number of LSP make before break procedures performed`
	`crpd_mpls_lsp_make_before_breaks_total{node="example.juniper.net"} 0`
	`# TYPE crpd_mpls_lsp_bandwidth_increases_total counter`
	`# HELP crpd_mpls_lsp_bandwidth_increases_total Total number of LSP bandwidth increases performed`
	`crpd_mpls_lsp_bandwidth_increases_total{node="example.juniper.net"} 0`
	`# TYPE crpd_mpls_lsp_bandwidth_decreases_total counter`
	`# HELP crpd_mpls_lsp_bandwidth_decreases_total Total number of bandwidth decreases performed`
	`crpd_mpls_lsp_bandwidth_decreases_total{node="example.juniper.net"} 0`
	`# TYPE crpd_mpls_lsp_update_cspf_failures_total counter`
	`# HELP crpd_mpls_lsp_update_cspf_failures_total Total number of in-place LSP auto-bandwidth resizing failures at the CSPF path computation stage`
	`crpd_mpls_lsp_update_cspf_failures_total{node="example.juniper.net"} 0`
	`# TYPE crpd_mpls_lsp_update_signaling_errors_total counter`
	`# HELP crpd_mpls_lsp_update_signaling_errors_total Total number of in-place LSP auto-bandwidth resizing failures when RSVP signaling error is received`
	`crpd_mpls_lsp_update_signaling_errors_total{node="example.juniper.net"} 0`

Table 4: Sample cRPD Telemetry Output *(Continued)*

Group	Sample Output
	# TYPE crpd_mpls_lsp_update_signaling_timeouts_total counter # HELP crpd_mpls_lsp_update_signaling_timeouts_total Total number of in-place LSP auto-bandwidth resizing failures when RSVP signaling takes too long to complete crpd_mpls_lsp_update_signaling_timeouts_total{node="example.juniper.net"} 0 # TYPE crpd_mpls_label_space_total_labels gauge # HELP crpd_mpls_label_space_total_labels The total label space available crpd_mpls_label_space_total_labels{label_space="LSI", node="example.juniper.net"} 999984 # TYPE crpd_mpls_label_space_free_labels gauge # HELP crpd_mpls_label_space_free_labels The number of freely available labels crpd_mpls_label_space_free_labels{node="example.juniper.net",label_space="LSI"} 999984 crpd_mpls_label_space_total_labels{node="example.juniper.net",label_space="Block"} 999984 crpd_mpls_label_space_free_labels{node="example.juniper.net",label_space="Block"} 999984 crpd_mpls_label_space_total_labels{node="example.juniper.net",label_space="Dynamic"} 999984 crpd_mpls_label_space_free_labels{node="example.juniper.net",label_space="Dynamic"} 999984 crpd_mpls_label_space_total_labels{node="example.juniper.net",label_space="Static"} 48576 crpd_mpls_label_space_free_labels{node="example.juniper.net",label_space="Static"} 48576

Prometheus is an open-source systems monitoring and alerting toolkit. You can use Prometheus to retrieve telemetry data from the cloud-native router host servers and view that data in the HTTP format. A sample of Prometheus configuration looks like this:

```
- job_name: "prometheus-JCNR-1a2b3c"

# metrics_path defaults to '/metrics'
# scheme defaults to 'http'.
```

```
static_configs:
- targets: ["<host-server-IP>:8070"]
```

Logging and Notifications

IN THIS SECTION

- Logging | 147
- Notifications | 148

Read this topic to learn about logging and notification functions in Juniper Cloud-Native Router. We discuss the location of log files, what you can log, and various log levels. You can also learn about the available notifications and how the notifications are implemented in the cloud-native router.

Logging

The Juniper Cloud-Native Router pods and containers use syslog as their logging mechanism. You can determine the location of the log files at the deployment time by retaining or changing the value of the **log_path** key in the **values.yaml** file. By default, the location of the log files is **/var/log/jcnr**. The system stores log files from all the cloud-native router pods and containers in the **log_path** directory.

In addition, a syslog-ng pod stores event notification data in JSON format on the host server. The syslog-ng pod stores the JSON-formatted notifications in the directory specified by the **syslog_notifications** key in the **values.yaml** file. By default, the file location is **/var/log/jcnr** and the filename is **jcnr_notifications.json**. You can change the location and filename by changing the value of the **syslog_notifications** key before the cloud-native router deployment.

When you use the default file locations, the **/var/log/jcnr** directory displays the following files:

```
[root@jcnr-01 jcnr]# ls
action.log                    contrail-vrouter-dpdk-init.log  filter
l2cos.log      __policy_names_rpdc__
contrail-vrouter-agent.log    contrail-vrouter-dpdk.log       filter.log
license        mgd-api
```

```
__policy_names_rpdn__         cos                             jcnr-cni.log
messages       mosquitto
vrouter-kernel-init.log       cscript.log                     jcnr_notifications.json
messages.0.gz  na-grpcd
```

NOTE:

contrail-vrouter-dpdk.logjcnr-cni.log

Notifications

The syslog-ng pod continuously monitors the preceding log files for notification events such as interface up, interface down, interface add, and so on. When these events appear in a log file, syslog-ng converts the log events into notification events and stores the events in JSON format within the **syslog_notifications** file configured in the **values.yaml** file.

Here is a sample of syslog-ng notifications:

Table 5: Supported Notifications

Notification	Source Pod
License Near Expiry	cRPD
License Expired	cRPD
License Invalid	cRPD
License OK	cRPD
License Grace Period	cRPD
License Not Present	cRPD
JCNR Init Success	Deployer
JCNR Init Failure	Deployer

Table 5: Supported Notifications *(Continued)*

Notification	Source Pod
JCNR Graceful Shutdown Request	Deployer
JCNR Graceful Shutdown Complete	Deployer
JCNR Graceful Shutdown Failure	Deployer
JCNR Restart	Deployer
JCNR Upgrade Success	Deployer
JCNR Upgrade Failure	Deployer
Upstream Fabric Bond Member Link Up	vRouter
Upstream Fabric Bond Member Link Down	vRouter
Upstream Fabric Bond Link Up	vRouter
Upstream Fabric Bond Link Down	vRouter
Upstream Fabric Bond Link Switchover	vRouter
Downstream Fabric Link Up	vRouter
Downstream Fabric Link Down	vRouter
Appliance Link Up	vRouter
Appliance Link Down	vRouter
Any JCNR Application Critical Errors	vRouter
Any JCNR Application Warnings	vRouter
Any JCNR Application Info	vRouter

Table 5: Supported Notifications *(Continued)*

Notification	Source Pod
JCNR Rate Limits Reached	vRouter
JCNR MAC Table Limit Reached	vRouter
JCNR CLI Start	cRPD or vRouter-Agent
JCNR CLI Stop	cRPD or vRouter-Agent
JCNR Kernel App Interface Up	vRouter
JCNR Kernel App Interface Down	vRouter
JCNR Virtio User Interface Up	vRouter
JCNR Virtio User Interface Down	vRouter

7 CHAPTER

Troubleshooting

Troubleshoot via the vRouter CLI | 152

Troubleshoot via Introspect | 164

Troubleshoot via the vRouter CLI

IN THIS SECTION

- Accessing the vRouter CLI | **152**
- Troubleshooting via the vRouter CLI | **154**

Read this topic to learn about the various troubleshooting commands available in the vRouter CLI. The following commands are covered in this topic:

- "Accessing the vRouter CLI" on page 152
- "Verify vRouter Interfaces via the `vif` Command" on page 154
- "View the running configuration of the vRouter" on page 155
- "View L2 Configuration and Statistics via the `purel2cli` Command" on page 156
- "View status and statistics of DPDK using the `dpdkinfo` Command" on page 159
- "Display routes and next hops using the `rt` and `nh` Commands" on page 162
- "Display all active flows using the `flow` Command" on page 163

Accessing the vRouter CLI

You can access the command-line interface (CLI) of the vRouter by accessing the shell of the running vRouter-agent container.

NOTE: The commands below are provided as an example. The vRouter pod name must be replaced from your environment. The command outputs may differ based on your environment.

List the K8s Pods running on the cluster

```
kubectl get pods -A
```

```
NAMESPACE         NAME                                           READY   STATUS      RESTARTS
AGE
contrail-deploy   contrail-k8s-deployer-7b5dd699b9-nd7xf         1/1     Running     0
41m
contrail          contrail-vrouter-masters-dfxgm                 3/3     Running     0
41m
jcnr              kube-crpd-worker-ds-8tnf7                      1/1     Running     0
41m
jcnr              syslog-ng-54749b7b77-v24hq                     1/1     Running     0
41m
kube-system       calico-kube-controllers-57b9767bdb-5wbj6       1/1     Running     2 (92d ago)
129d
kube-system       calico-node-j4m5b                              1/1     Running     2 (92d ago)
129d
kube-system       coredns-8474476ff8-fpw78                       1/1     Running     2 (92d ago)
129d
kube-system       dns-autoscaler-7f76f4dd6-q5vdp                 1/1     Running     2 (92d ago)
129d
kube-system       kube-apiserver-5a5s5-node2                     1/1     Running     3 (92d ago)
129d
kube-system       kube-controller-manager-5a5s5-node2            1/1     Running     4 (92d ago)
129d
kube-system       kube-multus-ds-amd64-4zm5k                     1/1     Running     2 (92d ago)
129d
kube-system       kube-proxy-l6xm8                               1/1     Running     2 (92d ago)
129d
kube-system       kube-scheduler-5a5s5-node2                     1/1     Running     4 (92d ago)
129d
kube-system       nodelocaldns-6kwg5                             1/1     Running     2 (92d ago)
129d
```

Copy the name of the vRouter pod—`contrail-vrouter-masters-dfxgm` in this example output . You will use the pod name to connect to the running container's shell.

Connect to the vRouter CLI

Issue the `kubectl exec` command to access the running container's shell:

```
kubectl exec -n <namespace> -it <pod name> --container <container name> -- bash
```

where *<namespace>* identifies the namespace in which the pod is running, *<pod name>* specificies the name of the pod and the *<container name>* specifies the name of the container (to be specified if the pod has more than one container).

The vRouter pod has three containers. When the container name is not specified, the command will default to the vrouter-agent container shell. Here is an example:

```
[root@jcnr-01]# kubectl exec -n contrail -it contrail-vrouter-masters-dfxgm -- bash
Defaulted container "contrail-vrouter-agent" out of: contrail-vrouter-agent, contrail-vrouter-
agent-dpdk,
contrail-vrouter-telemetry-exporter, contrail-init (init), contrail-vrouter-kernel-init-dpdk
(init)
[root@jcnr-01 /]#
```

At this point, you have connected to the vRouter's CLI.

Troubleshooting via the vRouter CLI

You can run commands in the CLI to learn about the state of the vRouter.

Verify vRouter Interfaces via the `vif` Command

The command shown below allows you to see which interfaces are present on the vRouter:

```
vif --list
Vrouter Operation Mode: PureL2
Vrouter Interface Table

Flags: P=Policy, X=Cross Connect, S=Service Chain, Mr=Receive Mirror
       Mt=Transmit Mirror, Tc=Transmit Checksum Offload, L3=Layer 3, L2=Layer 2
       D=DHCP, Vp=Vhost Physical, Pr=Promiscuous, Vnt=Native Vlan Tagged
       Mnp=No MAC Proxy, Dpdk=DPDK PMD Interface, Rfl=Receive Filtering Offload, Mon=Interface
is Monitored
       Uuf=Unknown Unicast Flood, Vof=VLAN insert/strip offload, Df=Drop New Flows, L=MAC
Learning Enabled
       Proxy=MAC Requests Proxied Always, Er=Etree Root, Mn=Mirror without Vlan Tag, HbsL=HBS
```

```
Left Intf
       HbsR=HBS Right Intf, Ig=Igmp Trap Enabled, Ml=MAC-IP Learning Enabled, Me=Multicast
Enabled

vif0/0      Socket: unix
            Type:Agent HWaddr:00:00:5e:00:01:00
            Vrf:65535 Flags:L2 QOS:-1 Ref:3
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0  bytes:0 errors:0
            TX packets:11  bytes:4169 errors:0
            Drops:0

vif0/1      PCI: 0000:00:00.0 (Speed 25000, Duplex 1)
            Type:Physical HWaddr:46:37:1f:de:df:bc
            Vrf:65535 Flags:L2Vof QOS:-1 Ref:8
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: eth_bond_bond0  Status: UP  Driver: net_bonding
            Slave Interface(0): 0000:3b:02.0  Status: UP  Driver: net_iavf
            Slave Interface(1): 0000:3b:02.1  Status: UP  Driver: net_iavf
            Vlan Mode: Trunk  Vlan: 100 200 300 700-705
            RX packets:0  bytes:0 errors:0
            TX packets:378  bytes:81438 errors:0
            Drops:0

vif0/2      PCI: 0000:3b:0a.0 (Speed 25000, Duplex 1)
            Type:Workload HWaddr:ba:69:c0:b7:1f:ba
            Vrf:0 Flags:L2Vof QOS:-1 Ref:7
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: 0000:3b:0a.0  Status: UP  Driver: net_iavf
            Vlan Mode: Access  Vlan Id: 700  OVlan Id: 700
            RX packets:378  bytes:81438 errors:2
            TX packets:0  bytes:0 errors:0
            Drops:391
```

View the running configuration of the vRouter

To see the status of the vRouter, enter the following command in the vRouter CLI:

```
[root@jcnr-01 /]# ps -eaf | grep vrouter-dpdk
root         116      90 99 Mar30 ?        118-08:05:37 /contrail-vrouter-dpdk --no-daemon --
socket-mem=1024 1024
--allow=0000:5a:02.0 --
vdev=eth_bond_bond0,mode=1,socket_id=0,mac=3a:1a:b7:86:1c:4f,primary=0000:5a:02.0,
slave=0000:5a:02.0 --l2_table_size=10240 --yield_option 0 --ddp --l2_mode
root     1134749 1134365  0 16:41 pts/0    00:00:00 grep --color=auto vrouter-dpdk
```

The output contains several elements.

Table 6: vRouter Status Attributes

Flag	Meaning
`--l2_mode`	The vRouter is running in L2 mode.
`--l2_table_size`	The current number of entries in the MAC table. The default size is 10240 entries.
`--allow=<PCI Id>`	The PCI ID of fabric and fabric workload interfaces. More than one ID can appear in the output. These IDs serve as an allowlist.
`--ddp`	Enable Intel DDP support. We enable DDP by default in the **values.yaml** file in the vRouter. **NOTE**: The Intel XL710 NIC does not support DDP.

View L2 Configuration and Statistics via the `purel2cli` Command

The `purel2cli` command is a useful utility to view the JCNR L2 configuration and statistics. Start by using the **`purel2cli --help`** command.

```
[root@jcnr-01 /]# purel2cli --help
Usage: purel2cli [--mac show]
           [--vlan show]
           [--vlan get <VLAN_ID>]
           [--acl show <VLAN_ID>]
           [--acl reset-counters <VLAN_ID>]
           [--l2stats get <VIF_ID> <VLAN_ID>]
           [--clear VLAN_ID]
           [--qos classifier/re-write/scheduler <NAME>]
           [--qos cla/rw/sch <NAME>]
           [--nolocal show]
           [--nolocal get <VLAN_ID>]
           [--sock-dir <sock dir>]
           [--help]
```

The `purel2cli --mac show` command shows the MAC addresses that the vRouter has dynamically learned.

```
purel2cli --mac show
==================================================
||  MAC              vlan      port      hit_count||
==================================================
00:01:01:01:01:03  1221      2          1101892
00:01:01:01:01:02  1221      2          1101819
00:01:01:01:01:04  1221      2          1101863
00:01:01:01:01:01  1221      2          1101879
5a:4c:4c:75:90:fe  1250      5          12
Total Mac entries 5
```

The **purel2cli --vlan show** command shows the VLANs and associated ports.

```
purel2cli --vlan show
VLAN      PORT
===============
1201      1,2,3,4,
1202      1,2,3,4,
1203      1,2,3,4,
1204      1,2,3,4,
1205      1,2,3,4,
```

You can also issue the `purel2cli --vlan get` command to get more details about the VLAN.

```
purel2cli --vlan get <vlan-id>
```

Issue the `purel2cli --l2stats` command to view L2 statistics. For example:

```
purel2cli --l2stats get <virtual_interface_ID> <VLAN_ID>
```

```
purel2cli --l2stats get 2 1221
Vlan id count: 1
--------------------------------------------------------------------------------
```

```
Statistics for vif 2 vlan 1221
--------------------------------------------------------------------------------
                Rx Pkts          Rx Bytes         Tx Pkts             Tx Bytes
Unicast        245344824       48152682842         835552           1667761792
Broadcast              0                 0              0                    0
Multicast              0                 0              0                    0
Flood                  0                 0              0                    0
--------------------------------------------------------------------------------
```

```
purel2cli --clear '*'
```

```
purel2cli --clear 100
```

Table 7: `purel2cli` Command Options for L2 Statistics

Sample Command	Function
`purel2cli --l2stats get '*' '*'`	Get statistics for all virtual interfaces (vif) and all VLAN IDs.
`purel2cli --l2stats get '*' 100`	Get statistics for all vif that are part of VLAN 100
`purel2cli --l2stats get 1 '*'`	Get statistics for all VLANs for which interface 1 is a member
`purel2cli --l2stats get 1 100`	Get statistics for interface 1 and VLAN 100

The command shows the VLAN to port mapping in the vRouter.You can use the command to see the bridge domain table entry for a specific VLAN: There are several variations of the command that allow you to display and filter L2 statistics in the vRouter. The base form of the command is: . The table below shows the available command options and what they do. It also provides a sample output using one of the options:The following command is an example of the L2 statistics for interface 2 and VLAN 1221:You can clear the statistics from the vRouter with the purel2cli command in the form: . Clears all statistics from all VLANs in the vRouter. Clears all statistics for VLAN id 100.

Packet Tracing via the `dropstats` Command

The vRouter tracks the packets that it drops and includes the reason for dropping them. The table below shows the common reasons for vRouter to drop a packet. When you execute the **dropstats** command, the vRouter does not show a counter if the count for that counter is 0.

Table 8: Dropstats Counters

Counter Name	Meaning
`L2 bd table drop`	No interfaces in bridge domain
`L2 untag pkt drop`	Untagged packet arrives on trunk or sub-interface
`L2 Invalid Vlan`	Packet VLAN does not match interface VLAN
`L2 Mac Table Full`	No more entries available in the MAC table
`L2 ACL drop`	Packet matched firewall filter (ACL) drop rule
`L2 Src Mac lookup fail`	Unable to match (or learn) the source MAC address

Example output from the **dropstats** command looks like:

```
dropstats
L2 bd table Drop              43
L2 untag pkt drop             716
L2 Invalid Vlan               7288253
Rate limit exceeded           673179706
L2 Mac Table Full             41398787
L2 ACL drop                   8937037
L2 Src Mac lookup fail        247046
```

View status and statistics of DPDK using the `dpdkinfo` Command

The **dpdkinfo** command provides insight into the status and statistics of DPDK. The **dpdkinfo** command has many options. The following sections describe the available options and the example output from the **dpdkinfo** command. You can run the **dpdkinfo** command only from within the vRouter-agent CLI.

```
dpdkinfo --help
Usage: dpdkinfo [--help]
                 --version|-v                                                     Show DPDK
Version
                 --bond|-b                                                        Show Master/
Slave bond information
                 --lacp|-l     <all/conf>                                         Show LACP
information from DPDK
```

```
                 --mempool|-m  <all/<mempool-name>>                                Show Mempool 
information
                 --stats|-n    <vif index value>                                   Show Stats 
information
                 --xstats|-x   <vif index value>                                   Show Extended 
Stats information
                 --lcore|-c                                                        Show Lcore 
information
                 --app|-a                                                          Show App 
information
                 --ddp|-d      <list> <list-flow>                           Show DDP information 
for X710 NIC
                 --rx_vlan|-z  <value>                                             Show VLan 
information
       Optional: --buffsz      <value>                                             Send output 
buffer size (less than 1000Mb)
```

The command **dpdkinfo -c** shows the Lcores assigned to DPDK VF fabric interfaces and the queue ID for each interface.

```
dpdkinfo -c
No. of forwarding lcores: 4

Lcore 10:
    Interface: 0000:18:01.1        Queue ID: 0
    Interface: 0000:18:0d.1        Queue ID: 0
    Interface: 0000:86:00.0        Queue ID: 0

Lcore 11:
    Interface: 0000:18:01.1        Queue ID: 1
    Interface: 0000:18:0d.1        Queue ID: 1
    Interface: 0000:86:00.0        Queue ID: 1

Lcore 12:
    Interface: 0000:18:01.1        Queue ID: 2
    Interface: 0000:18:0d.1        Queue ID: 2
    Interface: 0000:86:00.0        Queue ID: 2

Lcore 13:
    Interface: 0000:18:01.1        Queue ID: 3
    Interface: 0000:18:0d.1        Queue ID: 3
    Interface: 0000:86:00.0        Queue ID: 3
```

The command **dpdkinfo -m all** shows all of the memory pool information.

```
dpdkinfo -m all
---------------------------------------------------
Name            Size    Used    Available
---------------------------------------------------
rss_mempool             16384    1549    14835
frag_direct_mempool     4096    0    4096
frag_indirect_mempool    4096    0    4096
packet_mbuf_pool        8192    2    8190
```

The command **dpdkinfo -n 3** displays statistical information for a specific interface.

```
dpdkinfo -n 3
Interface Info(0000:18:0d.1):
RX Device Packets:6710, Bytes:1367533, Errors:0, Nombufs:0
Dropped RX Packets:0
TX Device Packets:0, Bytes:0, Errors:0
Queue Rx:
      Tx:
      Rx Bytes:
      Tx Bytes:
      Errors:
```

The command **dpdkinfo -x 3** displays extended statistical information for a specific interface.

```
dpdkinfo -x 3
Driver Name:net_iavf
Interface Info:0000:18:0d.1
Rx Packets:
    rx_good_packets: 6701
    rx_unicast_packets: 0
    rx_multicast_packets: 2987
    rx_broadcast_packets: 3714
    rx_dropped_packets: 0
Tx Packets:
    tx_good_packets: 0
    tx_unicast_packets: 0
    tx_multicast_packets: 0
    tx_broadcast_packets: 0
    tx_dropped_packets: 0
Rx Bytes:
    rx_good_bytes: 1365696
Tx Bytes:
    tx_good_bytes: 0
Errors:
    rx_missed_errors: 0
    rx_errors: 0
    tx_errors: 0
    rx_mbuf_allocation_errors: 0
    inline_ipsec_crypto_ierrors: 0
    inline_ipsec_crypto_ierrors_sad_lookup: 0
    inline_ipsec_crypto_ierrors_not_processed: 0
    inline_ipsec_crypto_ierrors_icv_fail: 0
    inline_ipsec_crypto_ierrors_length: 0
Others:
    inline_ipsec_crypto_ipackets: 0
---------------------------------------------------------------------
```

Display routes and next hops using the `rt` and `nh` Commands

Use the `rt` command to display all routes in a VRF. The `nh` command enables you to inspect the next hops that are known by the vRouter. Next hops tell the vRouter the next location to send a packet in the path to its final destination.

For example, for IPv4 traffic:

```
rt --get 172.68.20.2/32 --vrf 4
Match 172.68.20.2/32 in vRouter inet4 table 0/4/unicast
Flags: L=Label Valid, P=Proxy ARP, T=Trap ARP, F=Flood ARP, Ml=MAC-IP learnt route
vRouter inet4 routing table 0/4/unicast
Destination           PPL        Flags        Label         Nexthop    Stitched MAC(Index)
172.68.20.2/32          0          LPT         16            193        -
```

```
nh --get 193
Id:193        Type:Tunnel         Fmly: AF_INET  Rid:0  Ref_cnt:264        Vrf:0
                Flags:Valid, Policy, MPLSoUDP, Etree Root,
Oif:4 Len:14 Data:88 e6 4b 09 7d 46 40 a6 b7 2c a4 48 08 00 Sip:1.1.1.35 Dip:1.1.24.24
```

For example, for IPv6 traffic:

```
rt --get 2001:172:68:20::/64 --vrf 4 --family inet6
Match 2001:172:68:20::/64 in vRouter inet6 table 0/4/unicast
Flags: L=Label Valid, P=Proxy ARP, T=Trap ARP, F=Flood ARP, Ml=MAC-IP learnt route
vRouter inet6 routing table 0/4/unicast
Destination           PPL        Flags        Label         Nexthop    Stitched MAC(Index)
2001:172:68:20::/64     0          LPT         16             193        -
```

```
nh --get 193
Id:193        Type:Tunnel         Fmly: AF_INET  Rid:0  Ref_cnt:264        Vrf:0
              Flags:Valid, Policy, MPLSoUDP, Etree Root,
Oif:4 Len:14 Data:88 e6 4b 09 7d 46 40 a6 b7 2c a4 48 08 00 Sip:1.1.1.35 Dip:1.1.24.24
```

Display all active flows using the `flow` Command

Use the `flow` command to display all active flows in a system. For example:

```
flow -l --match 169.83.47.170:9398
Flow table(size 161218560, entries 629760)

Entries: Created 162630 Added 162614 Deleted 35136 Changed 35202Processed 162630 Used Overflow
entries 0
(Created Flows/CPU: 0 0 0 0 0 0 0 0 0 0 241 546 15 161828)(oflows 0)

Action:F=Forward, D=Drop N=NAT(S=SNAT, D=DNAT, Ps=SPAT, Pd=DPAT, L=Link Local Port)
 Other:K(nh)=Key_Nexthop, S(nh)=RPF_Nexthop
 Flags:E=Evicted, Ec=Evict Candidate, N=New Flow, M=Modified Dm=Delete Marked
TCP(r=reverse):S=SYN, F=FIN, R=RST, C=HalfClose, E=Established, D=Dead
 Stats:Packets/Bytes

Listing flows matching ([169.83.47.170]:9398)

    Index                Source:Port/Destination:Port                      Proto(V)
-----------------------------------------------------------------------------------
   328196<=>524233       169.83.47.170:9398                                  6 (2)
                         172.68.20.20:2159
(Gen: 3, K(nh):206, Action:F, Flags:, TCP:, E:1, QOS:-1, S(nh):206,  Stats:6/360,
 SPort 63929, TTL 0, Sinfo 38.0.0.0)
```

```
    524233<=>328196       172.68.20.20:2159                                 6 (2)
                          169.83.47.170:9398
 (Gen: 3, K(nh):206, Action:F, Flags:, TCP:, QOS:-1, S(nh):250,  Stats:0/0,
  SPort 60311, TTL 0, Sinfo 0.0.0.0)
```

Troubleshoot via Introspect

IN THIS SECTION

- Introspect | 164

Introspect

For vRouter-agent debugging, we use Introspect. You can access the Introspect data at **http://<host server IP>:8085**. Here is a sample of the Introspect data:

Table 9: Modules shown in contrail-vrouter-agent debug output

Link	and Description
agent.xml	Shows agent operational data. Using this introspect, you can see the list of interfaces, VMs, VNs, VRFs, security groups, ACLs and mirror configurations.
agent_ksync.xml	Shows agent ksync layer for data objects such as interfaces and bridge ports.
agent_profile.xml	shows agent **operdb**, tasks, flows, and statistics summary.
agent_stats_interval.xml	View and set collection period for statistics.
controller.xml	Shows the connection status of the jcnr-controller (cRPD)

Table 9: Modules shown in contrail-vrouter-agent debug output *(Continued)*

Link	and Description
cpuinfo.xml	Shows the CPU load and memory usage on the compute node.
ifmap_agent.xml	Shows the current configuration data received from **ifmap**.
kstate.xml	Shows data configured in the vRouter data path.
mac_learning.xml	Shows entries in vRouter-agent MAC learning table.
sandesh_trace.xml	Gives the different agent module traces such as **oper**, **ksync**, **mac learning**, and **grpc**.
sandesh_uve.xml	Lists all the user visible entitities (UVEs) in the vRouter-agent. The UVEs are used for analytics and telemetry.
stats.xml	Shows vRouter-agent slow path statistics such as error packets, trapped packets, and debug statistics.
task.xml	Shows vRouter-agent worker task details.

NOTE: The table shows grouped output. The cloud-native router does not group or sort the output on live systems.

The **http://*host server IP address*:8085** page displays only a list of HTML links.

8

CHAPTER

Appendix

Access cRPD CLI | 167

Access vRouter CLI | 168

Juniper Technology Previews (Tech Previews) | 170

Access cRPD CLI

You can access the command-line interface (CLI) of the cloud-native router controller by accessing the shell of the running cRPD container.

NOTE: The commands below are provided as an example. The cRPD pod name must be replaced from your environment. The command outputs may differ based on your environment.

View the running pods in the cluster:

```
kubectl get pods -A
NAMESPACE         NAME                                       READY   STATUS      RESTARTS
AGE
contrail-deploy   contrail-k8s-deployer-7b5dd699b9-nd7xf     1/1     Running     0
41m
contrail          contrail-vrouter-masters-dfxgm             3/3     Running     0
41m
jcnr              kube-crpd-worker-ds-8tnf7                  1/1     Running     0
41m
jcnr              syslog-ng-54749b7b77-v24hq                 1/1     Running     0
41m
kube-system       calico-kube-controllers-57b9767bdb-5wbj6   1/1     Running     2 (92d ago)
129d
kube-system       calico-node-j4m5b                          1/1     Running     2 (92d ago)
129d
kube-system       coredns-8474476ff8-fpw78                   1/1     Running     2 (92d ago)
129d
kube-system       dns-autoscaler-7f76f4dd6-q5vdp             1/1     Running     2 (92d ago)
129d
kube-system       kube-apiserver-5a5s5-node2                 1/1     Running     3 (92d ago)
129d
kube-system       kube-controller-manager-5a5s5-node2        1/1     Running     4 (92d ago)
129d
kube-system       kube-multus-ds-amd64-4zm5k                 1/1     Running     2 (92d ago)
129d
kube-system       kube-proxy-l6xm8                           1/1     Running     2 (92d ago)
129d
kube-system       kube-scheduler-5a5s5-node2                 1/1     Running     4 (92d ago)
129d
```

```
kube-system       nodelocaldns-6kwg5                         1/1     Running     2 (92d ago)
129d
```

Copy the name of the cRPD pod—`kube-crpd-worker-ds-8tnf7` in this example output . You will use the pod name to connect to the running container's shell.

Connect to the cRPD CLI

Issue the `kubectl exec` command to access the running container's shell:

```
kubectl exec -n <namespace> -it <pod name> --container <container name> -- bash
```

where *<namespace>* identifies the namespace in which the pod is running, *<pod name>* specificies the name of the pod and the *<container name>* specifies the name of the container (to be specified if the pod has more than one container).

The cRPD pod has only one running container. Here is an example command:

```
Defaulted container "kube-crpd-worker" out of: kube-crpd-worker, jcnr-crpd-config (init),
install-cni (init)

===>
           Containerized Routing Protocols Daemon (CRPD)
 Copyright (C) 2020-2022, Juniper Networks, Inc. All rights reserved.
                                                                      <===
root@jcnr-01:/#
```

At this point, you have connected to the shell of the cRPD. Just as with other Junos-based shells, you access the operational mode of the cloud-native router the same way as if you were connected to the console of a physical Junos OS device.

```
root@jcnr-01:/# cli
root@jcnr-cni>
```

Access vRouter CLI

You can access the command-line interface (CLI) of the vRouter by accessing the shell of the running vRouter-agent container.

NOTE: The commands below are provided as an example. The vRouter pod name must be replaced from your environment. The command outputs may differ based on your environment.

List the running pods on the K8s Cluster:

```
kubectl get pods -A
NAMESPACE         NAME                                      READY   STATUS      RESTARTS
AGE
contrail-deploy   contrail-k8s-deployer-7b5dd699b9-nd7xf    1/1     Running     0
41m
contrail          contrail-vrouter-masters-dfxgm            3/3     Running     0
41m
jcnr              kube-crpd-worker-ds-8tnf7                 1/1     Running     0
41m
jcnr              syslog-ng-54749b7b77-v24hq                1/1     Running     0
41m
kube-system       calico-kube-controllers-57b9767bdb-5wbj6  1/1     Running     2 (92d ago)
129d
kube-system       calico-node-j4m5b                         1/1     Running     2 (92d ago)
129d
kube-system       coredns-8474476ff8-fpw78                  1/1     Running     2 (92d ago)
129d
kube-system       dns-autoscaler-7f76f4dd6-q5vdp            1/1     Running     2 (92d ago)
129d
kube-system       kube-apiserver-5a5s5-node2                1/1     Running     3 (92d ago)
129d
kube-system       kube-controller-manager-5a5s5-node2       1/1     Running     4 (92d ago)
129d
kube-system       kube-multus-ds-amd64-4zm5k                1/1     Running     2 (92d ago)
129d
kube-system       kube-proxy-l6xm8                          1/1     Running     2 (92d ago)
129d
kube-system       kube-scheduler-5a5s5-node2                1/1     Running     4 (92d ago)
129d
kube-system       nodelocaldns-6kwg5                        1/1     Running     2 (92d ago)
129d
```

Copy the name of the vRouter pod—`contrail-vrouter-masters-dfxgm` in this example output . You will use the pod name to connect to the running container's shell.

Issue the `kubectl exec` command to access the running container's shell:

```
kubectl exec -n <namespace> -it <pod name> --container <container name> -- bash
```

where *<namespace>* identifies the namespace in which the pod is running, *<pod name>* specificies the name of the pod and the *<container name>* specifies the name of the container (to be specified if the pod has more than one container).

The vRouter pod has three containers. When the container name is not specified, the command will default to the vrouter-agent container shell. Here is an example:

```
[root@jcnr-01]# kubectl exec -n contrail -it contrail-vrouter-masters-dfxgm -- bash
Defaulted container "contrail-vrouter-agent" out of: contrail-vrouter-agent, contrail-vrouter-
agent-dpdk,
contrail-vrouter-telemetry-exporter, contrail-init (init), contrail-vrouter-kernel-init-dpdk
(init)
[root@jcnr-01 /]#
```

At this point, you have connected to the vRouter's CLI.

Juniper Technology Previews (Tech Previews)

Tech Previews enable you 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. Customers are encouraged 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 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. 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 should not be subject to existing SLAs or service agreements.

For additional details, please contact Juniper Support or your local account team.