

Juniper Cloud Native Router User Guide

Published
2022-10-03

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

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

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

Juniper Cloud Native Router User Guide

Copyright © 2022 Juniper Networks, Inc. All rights reserved.

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

YEAR 2000 NOTICE

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

END USER LICENSE AGREEMENT

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

Table of Contents

1

Juniper Cloud-Native Router (JCNR)

Juniper Cloud-Native Router - Overview | 2

Juniper Cloud-Native Router Controller (cRPD) | 8

JCNR-vRouter | 15

JCNR-CNI | 31

2

Juniper Cloud-Native Router – Features

Cloud-Native Router Features | 35

Juniper Cloud-Native Router Interface Types | 35

Logging and Notifications | 38

View L2 Metrics and Telemetry | 41

L2 ACLs (Firewall Filters) | 47

Licensing | 50

MAC Learning and Aging | 51

Broadcast Rate Limiting | 52

3

Juniper Cloud-Native Router (JCNR) - Examples

User Pod Example - Kernel Access | 55

Overview | 55

Before You Begin | 56

Detailed Steps | 57

User Pod Example - Virtio Trunk | 65

Overview | 65

Before You Begin | 66

Detailed Steps | 68

1

PART

Juniper Cloud-Native Router (JCNR)

[Juniper Cloud-Native Router - Overview](#) | 2

[Juniper Cloud-Native Router Controller \(cRPD\)](#) | 8

[JCNR-vRouter](#) | 15

[JCNR-CNI](#) | 31

Juniper Cloud-Native Router - Overview

IN THIS CHAPTER

- [Overview | 2](#)
- [Benefits of Juniper Cloud-Native Router | 2](#)
- [Kubernetes | 3](#)
- [JCNR Components | 4](#)
- [Ports Used by Cloud-Native Router | 6](#)

Overview

The Juniper Cloud-Native Router (cloud-native router) is a container-based software solution, orchestrated by Kubernetes (K8s). Cloud-native router combines the containerized routing protocol process (cRPD) and a DPDK-enabled Contrail virtual router (vRouter). With the cloud-native router, you can enable full Junos-based control plane with the enhanced forwarding capabilities of the DPDK-enabled vRouter.

Benefits of Juniper Cloud-Native Router

Some of the benefits provided by the cloud-native router solution are:

- Higher packet forwarding performance with DPDK-enabled vRouter
- Easy deployment on general purpose compute devices
- Out-of-the-box software-based open radio access network (O-RAN) support
- Quick spin up with containerized deployment on Kubernetes
- Highly scalable solution

Kubernetes

Let's talk a little about Kubernetes in this section. Kubernetes is an orchestration platform for running containerized applications in a clustered computing environment. It provides automatic deployment, scaling, networking, and management of containerized applications. Since Juniper Cloud-Native Router is a container-based solution, we have chosen Kubernetes as the orchestration platform. For complete details about Kubernetes, including installation, cluster creation, management, and maintenance, see <https://kubernetes.io/>.

Below we provide a brief description of the major components that make up a K8s cluster.

- **Nodes**

Kubernetes uses two types of nodes: a master (control) node and a compute (worker) node. A K8s cluster usually consists of one or more master nodes (in active/standby mode) and one or more worker nodes. You create a node on a physical computer or a VM.

NOTE: For the 22.X Release of Juniper Cloud-Native Router, you must provide a working, single-node Kubernetes cluster. Cloud-native router does not support multinode clusters, with master and worker nodes on separate VMs or BMS.

- **Pods**

Pods live in nodes and provide a space for containerized applications to run. A K8s Pod consists of one or more containers, with each Pod representing an instance of the application(s). A Pod is the smallest unit that K8s can manage. All containers in a Pod share the same network namespace.

- **Namespaces**

In K8s, Pods operate within a namespace to isolate groups of resources within a cluster. All K8s clusters have a *kube-system* namespace, which you might guess is for objects created by the Kubernetes system. Kubernetes also has a *default* namespace which holds all objects that don't provide their own namespace. The last two preconfigured Kubernetes namespaces are *kube-public* and *kube-node-lease*. The **kube-public** namespace is used to allow unauthenticated users to read some aspects of the cluster. Node leases allow the **kubelet** to send heartbeats so the control plane can detect node failure.

In the 22.X Release of Juniper Cloud-Native Router, some of the Pods run in the kube-system namespace while others provide their own namespace.

- **Kubelet**

The kubelet is the primary node agent that runs on each node. In the case of Juniper Cloud-Native Router, only a single kubelet runs on the cluster since we do not support multinode deployments.

- **Containers**

A container is a single package that consists of an entire runtime environment including the application and its:

- Configuration files
- Dependencies
- Libraries
- Other binaries

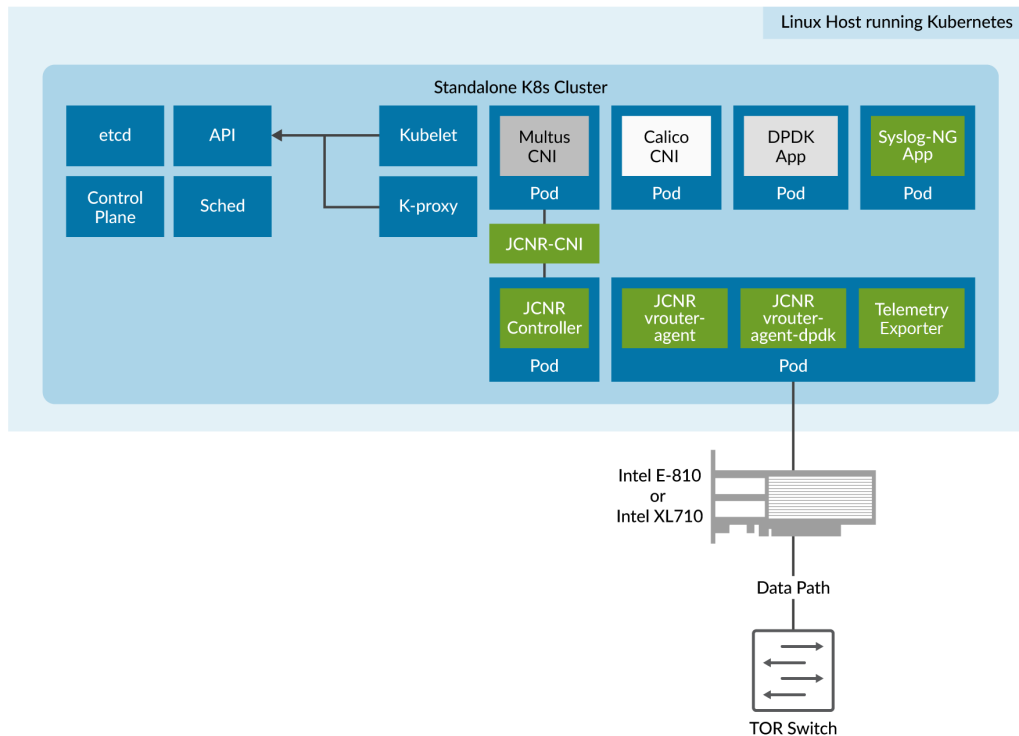
Software that runs in containers can, for the most part, ignore the differences in the those binaries, libraries, and configurations that may exist between the container environment and the environment that hosts the container. Common container types are docker, containerd, and CRI-O.

For the 22.X Release of Juniper Cloud-Native Router, docker is the only supported container type (container runtime).

JCNR Components

Several components make up the Juniper Cloud-Native Router solution. We give a brief overview of those components in this section.

The image below shows the components of the Juniper Cloud-Native Router inside a Kubernetes cluster. The green-colored components are JCNR-specific, while the others are required third-party components.



- **Juniper Cloud-Native Router Controller (JCNR-controller or cRPD)**

The cRPD acts as the control plane for the cloud-native router. It performs management functions and maintains configuration information for the vRouter forwarding plane. cRPD is based on the Junos OS control plane. You can configure it using:

- YAML-formatted Helm charts
- Third party management platforms that use the NETCONF protocol
- API calls to the cRPD MGD
- Direct CLI access to the cRPD Pod

- **Juniper Cloud-Native Router Dataplane (JCNR-vRouter or vRouter)**

JCNR-vRouter acts as the forwarding, or data, plane for Juniper Cloud-Native Router. It interacts with the JCNR-controller through the vRouter-agent and receives and forwards packets through its various interfaces.

JCNR-vRouter enables applications built using the DPDK framework to send and to receive packets directly between the application and the vRouter without passing through the kernel.

The vRouter receives configuration and management information from JCNR-controller through the JCNR vRouter-agent using the gRPC protocol.

- **Juniper Cloud-Native Router-Container Network Interface (JCNR-CNI)**

JCNR-CNI is a Kubernetes CNI and is responsible for provisioning network interfaces for application Pods. vRouter acts as the data-plane for these application Pod interfaces. JCNR-CNI interacts with Kubernetes, JCNR-controller and JCNR-vRouter. JCNR-CNI manages vRouter interface life cycles and cRPD configuration. When you remove an application Pod, JCNR-CNI removes the corresponding interface configuration from cRPD and state information from the vRouter-DPDK data plane.

Ports Used by Cloud-Native Router

Juniper Cloud-Native Router listens on certain TCP and UDP ports. The table below shows the ports, protocols, and a description for each one.

Table 1: Cloud-Native Router Listening Ports

Protocol	Port	Description
TCP	8085	vRouter introspect–Used to gain internal statistical information about vRouter
TCP	8070	Telemetry information–Used to see telemetry data from cloud-native router
TCP	9091	vRouter health check–JCNR checks to ensure contrail-vrouter-dpdk process is running, etc.
TCP	50052	gRPC port–JCNR listens on both IPv4 and IPv6
TCP	24	cRPD SSH
TCP	830	cRPD NETCONF
TCP	666	rpd
TCP	1883	Mosquito mqtt–Publish/subscribe messaging utility

Table 1: Cloud-Native Router Listening Ports *(Continued)*

Protocol	Port	Description
TCP	9500	agentd on cRPD
TCP	21883	na-mqttd
TCP	50051	jsd on cRPD
TCP	51051	jsd on cRPD
UDP	50055	Syslog-NG

Juniper Cloud-Native Router Controller (cRPD)

IN THIS CHAPTER

- [Benefits of Juniper Cloud-Native Router Controller | 8](#)
- [Configuration Options | 9](#)
- [Access to the CLI | 9](#)
- [Useful CLI Commands | 11](#)

Read this chapter to understand the Juniper Cloud-Native Router-controller (cloud-native router controller or cRPD), the Juniper Cloud-Native Router control plane.

Benefits of Juniper Cloud-Native Router Controller

The cRPD serves the role of the control plane in the Juniper Cloud-Native Router solution. The cRPD provides configuration interfaces to users (CLI) and applications (API) alike. You can use these interfaces to configure or program the JCNR-vRouter forwarding plane. You can configure a number of things using the JCNR-Controller:

- Virtual function (VF) fabric interfaces
- VF workload interfaces
- Trunk interfaces
- Access interfaces
- L2 ACLs (firewall rules)
- Bridge domains
- Ethernet switching
- VLANs

The cRPD performs the following functions:

- Supports JCNr-vRouter as the forwarding plane
- Maintains configuration for vRouter interfaces including trunk and access interfaces, virtual function interfaces (VFs), VLANs, and more
- Maintains configuration of bridge domains
- Maintains configuration for L2 firewall
- Maintains configuration for bridge domains, VLANs, virtual-switches, and so on
- Passes configuration information to the vRouter through the vRouter-agent
- Stores license key information

Configuration Options

During deployment, you configure the cRPD by changing the values of the *key:value* pairs contained within the **values.yaml** file which we include in the software distribution TAR file.

After deployment, we recommend that you use the NETCONF protocol with PyEZ to configure cRPD. See <https://www.juniper.net/documentation/us/en/software/junos-pyez/junos-pyez-developer/index.html> for details about PyEZ. Alternatively, you can SSH directly to the cRPD on TCP port 24 or using NETCONF on TCP port 830. Finally, another option for you to configure the cloud-native router is to access the Junos CLI on the cRPD using Kubernetes commands to connect to the cRPD Pod.

Access to the CLI

In this procedure we provide CLI commands that you run on the host server. We do not show a prompt before the commands so you can copy and paste the commands into your own cloud-native router.

```
kubect1 get pods -A
```

The output should look like:

NAMESPACE	NAME	READY	STATUS
RESTARTS	AGE		

contrail-deploy 0	contrail-k8s-deployer-7b5dd699b9-smqgn 37h	1/1	Running
contrail 0	contrail-vrouter-masters-htcv 37h	3/3	Running
default 0	delete-crpd-dirs--1-bjngd 37h	0/1	Completed
default 0	delete-vrouter-dirs--1-k5wgb 37h	0/1	Completed
default 0	odu-pktgen-trunkint 24h	1/1	Running
default 0	odu-subinterface-3003 7d	1/1	Running
kube-system 52 (8d ago)	calico-kube-controllers-57b9767bdb-76fvw 107d	1/1	Running
kube-system 18 (8d ago)	calico-node-pgljp 107d	1/1	Running
kube-system 38 (8d ago)	coredns-8474476ff8-2nbnv 107d	1/1	Running
kube-system 18 (8d ago)	dns-autoscaler-7f76f4dd6-8b4w5 107d	1/1	Running
kube-system 45 (8d ago)	kube-apiserver-nodem27.englab.juniper.net 107d	1/1	Running
kube-system 34 (8d ago)	kube-controller-manager-nodem27.englab.juniper.net 107d	1/1	Running
kube-system 0	kube-crpd-worker-ds-89wzg 32h	1/1	Running
kube-system 0	kube-multus-ds-amd64-f2pls 8d	1/1	Running
kube-system 18 (8d ago)	kube-proxy-vrqjm 107d	1/1	Running
kube-system 35 (8d ago)	kube-scheduler-nodem27.englab.juniper.net 107d	1/1	Running
kube-system 43 (8d ago)	node-local-dns-hm56k 107d	1/1	Running
kube-system 0	syslog-ng-54749b7b77-tqvpk 37h	1/1	Running

The command to access the cRPD CLI has the form: `kubectl exec -n kube-system -it <full cRPD Pod name> -- bash`. If we use the output from above, the command appears as: `kubectl exec -n kube-system -it kube-crpd-worker-ds-89wzg -- bash`.

The output from the command above (when you use the full name of your cRPD Pod) should look like:

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

===>
      Containerized Routing Protocols Daemon (CRPD)
      Copyright (C) 2020-2021, Juniper Networks, Inc. All rights reserved.
                                     <===
```

This output indicates that you have attached to the cRPD CLI. At this point, your access level is root and you are in shell mode. Just as when you connect as root to any Junos OS-based device, you must enter the `cli` command to access the Junos CLI in operation mode.

Useful CLI Commands

This section provides some example CLI commands and their outputs. We also provide some command completion example outputs. These outputs allow you to see the available command hierarchy which you can explore on your cloud-native router system.

You can see the bridge command hierarchy with the `show bridge ?` command as shown below.

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

If you look further into the hierarchy, you see:

```
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
```

If you use the <[Enter]> option, you see something like:

```
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 show bridge mac-table command displays the L2 MAC table which is dynamically learned by the vRouter.

If you look at the other option, statistics, you see:

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

If you use the <[Enter]> option, you see something like:

```
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

```

The `show bridge statistics` command displays the L2 VLAN traffic statistics per interface within a bridge domain.

To see the firewall (ACL) configuration:

```

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;
        }
      }
    }
  }
}

```

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

To see the how many packets matched the filter (per VLAN) you can use the cRPD CLI and issue the command:

```
show firewall filter filter1
```

The output from the above command looks like:

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

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

CHAPTER 3

JCNR-vRouter

IN THIS CHAPTER

- Benefits of JCNR vRouter | 15
- How to Access vRouter CLI | 17
- Packet Flow in vRouter | 17
- Monitoring vRouter with CLI Commands | 19
- The **dropstats** Command | 23
- The **dpdinfo** Command | 24
- Troubleshooting vRouter | 28

Read this chapter to understand details about JCNR-vRouter, the JCNR DPDK-based forwarding plane.

Benefits of JCNR 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) with Intel Adapter Virtual Function (IAVF) and Dynamic Device Personalization (DDP)
 - Intel XL710 (Fortville) with Intel Adapter Virtual Function (IAVF)

NOTE: DDP is not supported on the Intel XL710 NIC

- **Interface Support:**

- POD interfaces using virtio
- POD interfaces using kernel veth pair
- DPDK Virtual Function (VF) workload interfaces
- DPDK VF fabric trunk interfaces

You define DPDK VF fabric trunk interfaces in the **values.yaml** file that is used in vRouter deployment. This makes JCNr aware of the names of the interfaces, their MAC addresses, and their PCI slot ID. To keep traffic flows manageable, we apply VLAN filtering to the physical interfaces. When you configure VLANs for use with cloud-native router, only those configured VLANs can pass by the physical interfaces.

Physical interfaces defined in **values.yaml** as workload interfaces are equipped with only one receive and one transmit queue. The system assigns one forwarding CPU core to the task of polling the interface for traffic. Physical interfaces defined in **values.yaml** as fabric interfaces are equipped with as many receive and transmit queues as you assign forwarding CPU cores to handle the polling. For example, if you assign three forwarding CPU cores to the fabric interface, the system allocates three receive and three transmit queues to the interface.

- **Interface Bonding**

DPDK vRouter supports interface bonding in active/standby mode on DPDK VF fabric interfaces. The **values.yaml** file specifies the interface names, mode value, and primary and secondary (slave) interface designations. DPDK contains a library with its own bonding driver that it uses for bonding. In operation, the vRouter uses the primary interface to pass traffic. If the primary link goes down, the secondary interface in the pair passes traffic until the primary interface reconnects.

- **Pod DPDK Interfaces**

JCNr-vRouter supports virtio communication to the POD application. The JCNr-CNI allocates unique socket directories which it passes to Pod applications and to vRouter. JCNr-CNI ensures that one Pod cannot access the resources of another Pod through isolation of vhost sockets and Pod volume mounts.

- **Pod Kernel Interfaces**

JCNr supports the *veth* interface type to communicate with Pod applications that use the Linux Kernel's networking stack.

How to Access vRouter CLI

```
kubectl get pods -n contrail
```

The output of the command above looks like:

NAME	READY	STATUS	RESTARTS	AGE
contrail-vrouter-masters-97v8z	3/3	Running	0	6d1h

To access the vRouter-agent CLI, you use the full Pod name from your system in the following command:

```
kubectl exec -n contrail -it contrail-vrouter-masters-97v8z -- bash
```

The output of the command above looks like: 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).

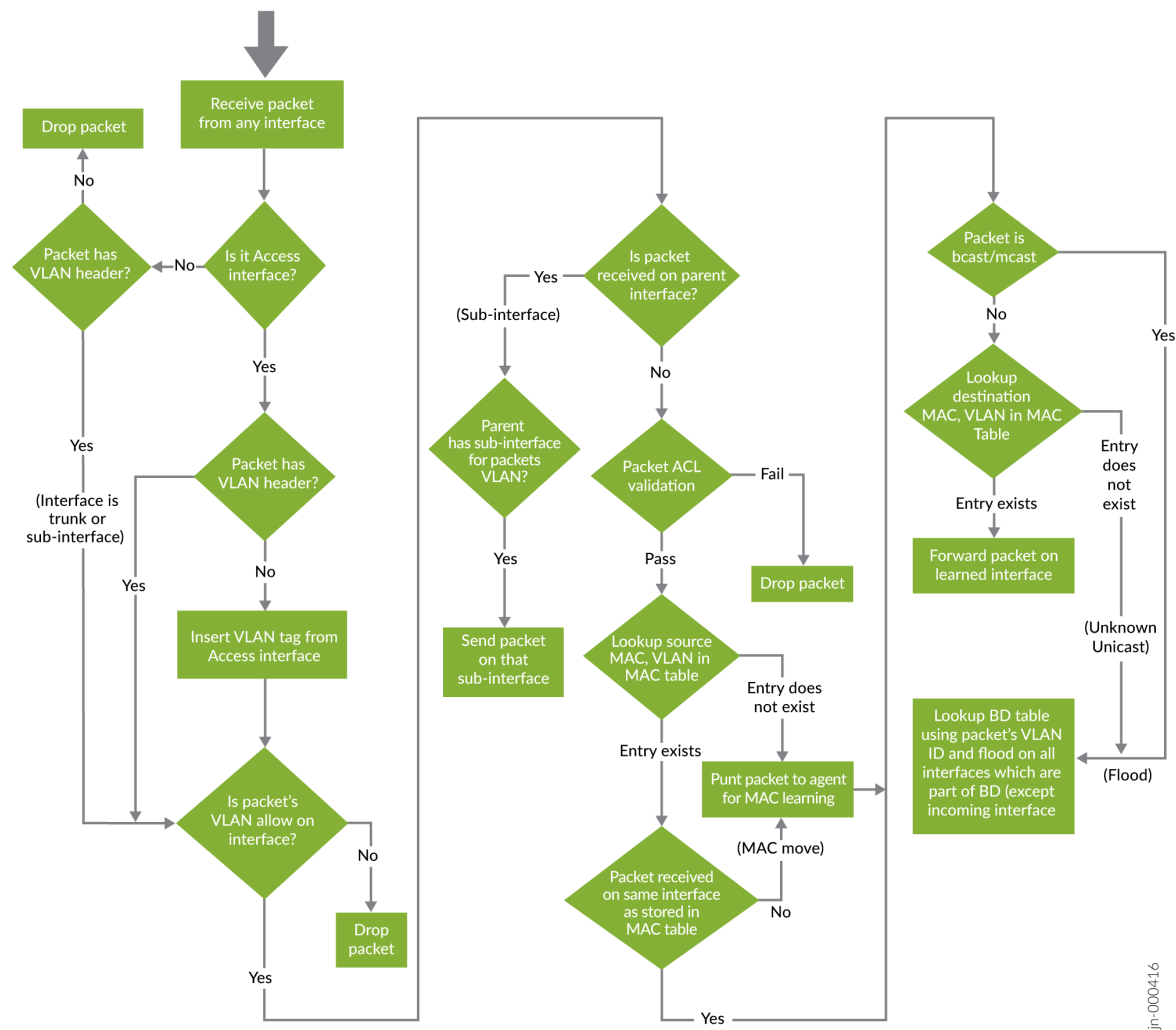
Once on the vRouter CLI, there are a number of commands that you can run to monitor and troubleshoot the system. We illustrate some of the available commands in ["Monitoring vRouter with CLI Commands" on page 19](#).

Packet Flow in vRouter

When you must understand something like a switch or router, it is useful to know what happens to packets as they flow through. This section describes the life of a packet in the vRouter. We use that description to illustrate how the vRouter MAC and bridge domain (BD) tables are populated and to introduce some of the CLI commands that you can use to see various parts of the vRouter from inside.

The flow chart below illustrates one possible generic packet flow through cloud-native router. It does not cover all possible interactions with the packet.

Figure 1: The Life of a Packet in vRouter



As you can see, the vRouter makes a lot of decisions about a received packet to ensure that the packet is handled correctly. Let's look at some of what the vRouter does with a packet. You can see in the figure above that there are several choices made based on the VLAN or bridge domain (BD). One of the tables that vRouter consults for making forwarding decisions is the BD table. A small example of a BD table is shown below.

Table 2: BD Table

VLAN ID (KEY)	Interface ID List (Value)
1024	2,3,4
1042	4,5
1022	1
1044	6

The BD table tells the vRouter which interfaces can carry traffic with a specific VLAN ID. Thus, the VLAN ID serves as the key for the table while the Interface ID List serves as the values for each entry.

Closely related to the BD table is the MAC table. The MAC table uses the MAC address and VLAN ID as a key pair. Then the Interface ID and hit count serve as the values for each entry in the table. We show an example below.

Table 3: MAC Table

MAC Address	VLAN ID	Interface ID	Hit Count
00:11:22:33:44:55	1024	2	123234
00:22:33:44:55:66	1042	4	823948
00:33:44:55:66:77	1022	1	45980
00:44:55:66:77:88	1044	6	86578

The primary purpose of the MAC table is to map which MAC addresses can be reached through which interface. vRouter makes entries in and consults the MAC table while processing packets.

Monitoring vRouter with CLI Commands

In the vRouter, CLI commands are useful for troubleshooting and monitoring purposes. As mentioned in ["How to Access vRouter CLI" on page 17](#), you can access the CLI of the vRouter. By executing commands in that CLI, you can learn about various aspects of the running vRouter. The examples below

assume that you have already connected to the vRouter CLI. The commands that we show in this section do not show a command prompt so that you can copy and paste them into your own vRouter.

We use the **purel2cli** command in most of the examples below. The command has more options than we show in the examples. In addition, the **purel2cli** has a help command that you can use to see the available options.

The purel2cli Command

To see all the options of the **purel2cli** command in the vRouter CLI, execute the command with the **--help** option.

```
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]
      [--sock-dir <sock dir>]
      [--help]
```

See the Current Status of Your Running vRouter

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

```
ps -eaf|grep dpdk
```

The output from the command above looks like: root 127 93 99 Jul29 ? 82-20:31:49 /contrail-vrouter-dpdk --no-daemon --socket-mem=1024 1024 --allow=0000:01:10.1 --allow=0000:01:10.0 --l2_table_size=10240 --yield_option 0 --ddp --l2_mode

There are several things shown in the output:

Table 4: vRouter Status

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.

Show MAC Table

The following command shows the MAC addresses that the vRouter has dynamically learned.

```
purel2cli --mac show
```

The output from the above command looks like:

```
=====
||  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
```


Show Bridge Domain Table

The following command shows the VLAN to port mapping in the vRouter.

```
purel2cli --vlan show
```

The output from the above command looks like:

```
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 use the following form of the **purel2cli** command to see the bridge domain table entry for a specific VLAN: `purel2cli --vlan get <vlan-id>`

Show L2 Statistics

There are several command variations of the **purel2cli** command that allow you to display and filter L2 statistics in the vRouter. The base form of the command is: `purel2cli -- l2stats get <virtual_interface_ID> <VLAN_ID>`. The table below shows the available command options and what they do. We follow the table with a sample output using one of the options.

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

As an example, the following command shows the L2 statistics for interface 2 and VLAN 1221:

```
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

```
-----
```

Clear L2 Statistics

The following example shows commands that allow you to clear L2 statistics information from the vRouter.

You can clear the statistics from the vRouter with the purel2cli command in the form: purel2cli --clear <VLAN_ID>.

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

clears all statistics from all VLANs in the vRouter.

```
purel2cli --clear 100
```

clears all statistics for VLAN id 100.

The dropstats Command

vRouter keeps track of the packets it drops for any reason. The table below shows some of the most common reasons that vRouter would 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 5: 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

```

The dpdkinfo Command

The **dpdkinfo** command provides insight into the status and statistics of DPDK. The **dpdkinfo** command has many options. First we show the available options, then we show some example output from the **dpdkinfo** command. You can only run the **dpdkinfo** command from within the vRouter-agent CLI.

dpdkinfo Command Usage

```
dpdkinfo

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)
```

dpdkinfo Lcore Information

This command 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
```

dpdkinfo Memory Pool Information

This command 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	

dpdkinfo Statistics Information

This command 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:
```

dpdkinfo Extended Statistics Information

This command 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

Troubleshooting vRouter

For vRouter-agent debugging we use Introspect. You access the Introspect data at **http://<host server IP>:8085**. A sample of the data you can see is shown below.

NOTE: We have grouped the output shown in the table below. Cloud-native router does not group or sort the output on live systems.

The page that displays when you go to **http://<host server IP address>:8085** shows only a list of html links. The items on the list of links are shown in bold.

Table 6: Cloud-Native Router vRouter Debug

Group	Links and Description
Modules for contrail-vrouter-agent	<ul style="list-style-type: none"> 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) 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.

Table 6: Cloud-Native Router vRouter Debug *(Continued)*

Group	Links and Description
	<ul style="list-style-type: none">• 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 entitites (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.

JCNR-CNI

IN THIS CHAPTER

- [Benefits of JCNR-CNI | 31](#)
- [JCNR-CNI Inside Cloud-Native Router | 32](#)
- [JCNR-CNI Role in Pod Creation | 33](#)
- [Network Attachment Definitions | 33](#)

Read this chapter to learn the details about JCNR-CNI, the primary container network interface for JCNR.

Benefits of JCNR-CNI

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

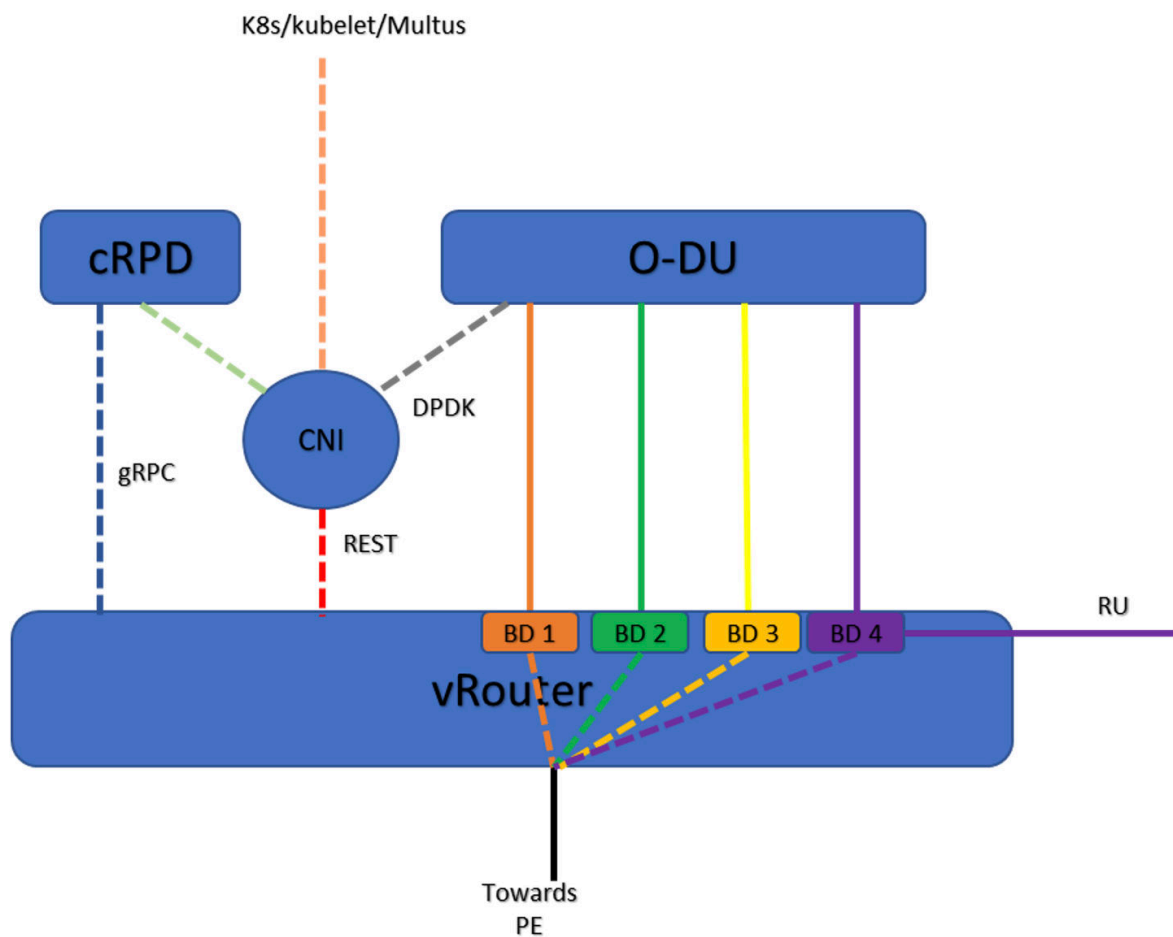
JCNR-CNI Instantiates many different kinds of Pod interfaces

- Creates virtio-based high performance interfaces for Pods that leverage the DPDK data plane
- Creates veth pair interfaces that allow Pods to communicate using the Linux Kernel networking stack
- Creates pod interfaces in access or trunk mode
- Attaches Pod interfaces to bridge domains
- Supports IPAM plugin for Dynamic IP address allocation
- Allocates unique socket interfaces for virtio interfaces,
- Applies L2 access control lists (ACLs) to JCNR-vRouter
- Attaches Pod interfaces to a bridge domain

- Manages the networking tasks in Pods such as assigning IP addresses and setting up of interfaces between the Pod and host in a Kubernetes cluster
- Applies K8s network policies that are translated to firewall filter rules. The policies are sent to JCNR-vRouter for application in the data plane
- Connects Pod interface to network: Pod-to-Pod and Pod-to-network
- Integrates with JCNR-vRouter for offloading packet processing

JCNR-CNI Inside Cloud-Native Router

JCNR-CNI is a specialized container network interface that can make a variety of different network connections. It operates in cooperation with the Multus CNI. The image below shows how JCNR-CNI interacts with the other components in Juniper Cloud-Native Router.



JCNR-CNI Role in Pod Creation

When you create a Pod for use in the cloud-native router, the Kubernetes (K8s) component known as **kubelet** calls the multus CNI to set up Pod networking and interfaces. Multus looks at the annotations section of the pod.yaml file to find the network attachment definitions (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 cRPD.

Network Attachment Definitions

Network attachment definition (NAD) files are Kubernetes (K8s) files that the multus CNI uses during the interface creation phase of Pod creation. A NAD specifies interface MAC addresses, allocates IP addresses, etc. Each Pod can use one or more NAD, typically one per Pod interface. In the Pod YAML file, the NAD to use for Pod creation is listed under the network annotations section. In addition to interface creation on Pods, NADs can create virtual switches. The NAD attaches Pod interfaces to L2 switching instances. The table below describes the L2 interface types and modes supported.

Table 7: NAD - L2 Interface Modes

Interface Mode	Characteristics	Comments
Access	Allows untagged packets to traverse the link to the Pod	Must be explicitly bound to a bridge domain
		Virtual switches use access mode for non-DPDK interfaces and applications like SSH and syslog
Trunk	Allows packets within specifically configured VLAN range	Implicitly part of one or more bridge domains
	No IP address allocation by CNI	Virtual switches in trunk mode carry DU user-plane traffic
	If IP address is needed, the Pod must have its own allocation method such as DHCP	Dynamically add and remove network slices in 5G environments without restarting the Pod

2

PART

Juniper Cloud-Native Router – Features

Cloud-Native Router Features | 35

Cloud-Native Router Features

SUMMARY

Use this guide to learn about day-to-day configuration, operation, and monitoring of the Juniper Cloud-Native Router.

IN THIS SECTION

- [Juniper Cloud-Native Router Interface Types | 35](#)
- [Logging and Notifications | 38](#)
- [View L2 Metrics and Telemetry | 41](#)
- [L2 ACLs \(Firewall Filters\) | 47](#)
- [Licensing | 50](#)
- [MAC Learning and Aging | 51](#)
- [Broadcast Rate Limiting | 52](#)

Juniper Cloud-Native Router Interface Types

Juniper Cloud-Native Router supports the following types of interfaces:

- **Agent interfaces**

vRouter has only one agent interface. It is used to communicate between the vRouter-agent and vRouter. On the vRouter CLI when you issue the `vif --list` command, the agent interface looks like:

```
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
            RX packets:0 bytes:0 errors:0
            TX packets:650 bytes:99307 errors:0
            Drops:0
```

- **DPDK Virtual Function (VF) workload interfaces**

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

```
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
            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 Virtual Function (VF) fabric interfaces**

DPDK VF fabric interfaces accept traffic from multiple VLANs and are associated with the physical NIC on the host server. On the vRouter CLI when you issue the `vif --list` command, the DPDK VF fabric interface looks like:

```
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 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/Standby bond interfaces**

Bond interfaces accept traffic from multiple VLANs. The bond interface runs in active/standby mode (mode 0).

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

```
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
          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: 751-755
          RX packets:8108366000 bytes:486501960000 errors:4234
          TX packets:65083776 bytes:4949969408 errors:0
          Drops:8108370394
```

- **POD interfaces using virtio and the DPDK data plane**

Virtio interfaces accept traffic from multiple VLANs and are associated with Pod interfaces that use virtio on the DPDK data plane.

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

```
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
          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 the veth pair and the DPDK data plane**

Pod interfaces that use **veth** pairs and the DPDK data plane are access interfaces rather than trunk interfaces. This type of Pod interface only allows traffic from one VLAN to pass.

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

```
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
            Vlan Mode: Access Vlan Id: 3001 OVlan Id: 3001
            RX packets:9 bytes:802 errors:515
            TX packets:0 bytes:0 errors:0
            Drops: 525
```

When you create interfaces, you must provide configuration values for the JCNR-CNI to use. Pod definition YAML files and NAD YAML files contain key:value pairs that you must configure for your installation. You can then apply the appropriate Pod and NAD YAML files to the Kubernetes cluster.

The NAD YAML files contain the interface definitions. You can see example NAD and Pod YAML files in the ["User Pod Example - Kernel Access" on page 55](#) and ["User Pod Example - Virtio Trunk" on page 65](#) examples.

Logging and Notifications

IN THIS SECTION

- [File Locations | 39](#)
- [Notifications | 39](#)

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

File Locations

The Juniper Cloud-Native Router Pods and containers use syslog as their logging mechanism. You can determine the location of the log files at deployment time by reading 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/jcncr**. The system stores log files from all the cloud-native router Pods and containers in the **log_path** directory.

In addition, we use a Syslog-NG Pod to store 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 location for the file is **/var/log/jcncr**; and the name of the file is **jcnr_notifications.json**. You can change the location and filename by changing the value of the **syslog_notifications** key before deployment of the cloud-native router.

When you use the default file locations, a list of the files in the **/var/log/jcncr** directory looks something like:

```
[root@jcncr-1 /var/log/jcncr]# ls
contrail-vrouter-agent.log    contrail-vrouter-agent.log.2    contrail-vrouter-dpdk.log
jcnr_notifications.json
contrail-vrouter-agent.log.1  contrail-vrouter-dpdk-init.log  jcnr-cni.log
vrouter-kernel-init.log
```

NOTE: Log rotation for the **contrail-vrouter-dpdk.log** and the **jcnr-cni.log** files must be handled by the host server.

Notifications

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

As of the 22.2 Release of Juniper Cloud-Native Router, the complete list of notifications that syslog-ng stores is shown in

Table 8: Supported Notifications

Notification	Source Pod
License Near Expiry	cRPD
License Expired	cRPD
License Invalid	cRPD
License OK	cRPD
JCNR Init Success	Deployer
JCNR Init 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
Downstream Fabric Link Up	vRouter
Downstream Fabric Link Down	vRouter
Appliance Link Up	vRouter
Appliance Link Down	vRouter
Any JCNR Application Critical Errors	vRouter
JCNR MAC Table Limit Reached	vRouter

Table 8: Supported Notifications *(Continued)*

Notification	Source Pod
JCNR CLI Start	cRPD/vRouter-Agent
JCNR CLI Stop	cRPD/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

SEE ALSO

[No Link Title](#)

View L2 Metrics and Telemetry

IN THIS SECTION

- [Benefits of Viewing L2 Metrics](#) | 41

Read this chapter to learn how to retrieve L2 metrics from an instance of Juniper Cloud-Native Router.

Benefits of Viewing L2 Metrics

Juniper Cloud-Native Router comes with telemetry capabilities that allow you to look into the system and see performance metrics. We accomplish this with the use of a container named **contrail-vrouter-**

telemetry-exporter that runs along side the other vRouter containers in the **contrail-vrouter-masters** Pod.

The telemetry exporter periodically queries the Introspect agent on the vRouterAgent for statistics and reports metrics information in response to Prometheus scrape requests. You can directly view both the telemetry data using the following URL: `http://<host server IP address>:8070`. We show sample output in the table below.

NOTE: We have grouped the output shown in the table below. Cloud-native router does not group or sort the output on live systems.

Table 9: Sample Telemetry Output

Group	Sample Output
Memory Usage per vRouter	<pre># 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</pre>

Table 9: Sample Telemetry Output (Continued)

Group	Sample Output
Packet count per interface	<pre> # 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 </pre>

Table 9: Sample Telemetry Output (Continued)

Group	Sample Output
CPU usage per vrouter	<pre># 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</pre>
Drop packet count per vrouter	<pre># 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</pre>

Table 9: Sample Telemetry Output (Continued)

Group	Sample Output
Packet count per interface per vlan	<pre> # 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="10 0"} 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="1 00"} 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="1 00"} 0 </pre>

Table 9: Sample Telemetry Output (Continued)

Group	Sample Output
	<pre> # TYPE virtual_router_interface_vlan_broadcast_input_bytes_total counter # 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="10 0"} 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="1 00"} 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="1 00"} 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 ... </pre>

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

```

- job_name: "prometheus-JCNR-1a2b3c"

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

```

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

SEE ALSO

No Link Title

L2 ACLs (Firewall Filters)

IN THIS SECTION

- [L2 Firewall Filters | 47](#)
- [Firewall Filter Example | 48](#)
- [L2 Firewall Filter \(ACL\) Troubleshooting | 49](#)

Read this chapter to learn about the use of L2 ACLs in the cloud-native router.

L2 Firewall Filters

Starting in the 22.2 Release of Juniper Cloud-Native Router, we included a limited firewall filter capability. You can configure the filters using the Junos OS CLI within the JCNR-Controller, using NETCONF, or using the cloud-native router APIs.

During deployment, the system defines and applies firewall filters to block traffic from passing directly between cloud-native router interfaces. You can dynamically define and apply more filters as needed. The firewall filters allow you 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 ether-type
- Define multiple terms within each filter
- Discard the traffic that matches the filter
- Apply filters to bridge-domains

Firewall Filter 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: You can configure up to 16 terms in a single firewall filter.

The only *then* action you can configure in a firewall filter is the discard action.

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

To see the how many packets matched the filter (per VLAN) you can use the cRPD CLI and issue the command:

```
show firewall filter filter1
```

The output from the above command looks like:

```
Filter : filter1    vlan-id : 3001
```

Term	Packet
t1	0

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

L2 Firewall Filter (ACL) Troubleshooting

The table below lists some of the potential problems that you might face when you implement firewall rules (ACLs) in cloud-native router. You run most of these commands on the host server. We indicate in the command column if the command shown needs to run somewhere else.

Table 10: L2 Firewall Filter (ACL) Troubleshooting

Problem	Possible Causes	Command
Firewall filters (ACLs) not working	gRPC connection (port 50052) to vRouter is down Check gRPC connection	netstat -antp grep 50052
	The ui-pubd process is not running Check if ui-pubd is running	ps aux grep ui-pubd
Firewall filter (ACL) show commands not working	gRPC connection (port 50052) to vRouter is down Check gRPC connection	netstat -antp grep 50052
	Firewall service is not running	ps aux grep firewall
		show log filter.log You must run this command in the JCNr-controller (cRPD) CLI.

SEE ALSO

No Link Title
No Link Title
No Link Title

Licensing

IN THIS SECTION

- [Licensing in the Juniper Cloud-Native Router | 50](#)

Read this section to learn about Juniper Cloud-Native Router licensing.

Licensing in the Juniper Cloud-Native Router

In the 22.2 Release of Juniper Cloud-Native Router, we've enabled our Juniper Agile Licensing (JAL) model. JAL ensures that features are used in compliance with Juniper's End User License Agreement. You purchase licenses for Juniper Cloud-Native Router software through your account team. You apply licenses to the cloud-native router using the CLI of the cloud-native router controller. For details about managing multiple license files for multiple cloud-native router deployments, see [Juniper Agile Licensing Overview](#)

If your cRPD Pod is running when you issue the command `kubect1 get pods -A` on the host server, then your license file has been properly applied.

NOTE: In the 22.3 Release of cloud-native router, we only monitor license compliance. We do not enforce license compliance.

MAC Learning and Aging

IN THIS SECTION

- [MAC Learning | 51](#)
- [MAC Entry Aging | 52](#)

Juniper Cloud-Native Router provides automated learning and aging of MAC addresses. This chapter provides an overview of MAC learning and aging functionality in cloud-native router.

MAC Learning

MAC learning is needed in cloud-native router so that received packets can be sent efficiently to their destination. The cloud-native router maintains a table of MAC addresses by interface. The table includes the MAC address, the associated VLAN, and the interface that the MAC and VLAN were learned on. The MAC table informs the vRouter about which interfaces can reach which MAC addresses.

Cloud-native router caches the source MAC for a new packet flow to record the incoming interface into the MAC table. Cloud-native router learns MAC addresses per VLAN/bridge domain (BD). 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.

When vRouter needs to forward a packet to a destination MAC address, it consults the MAC table. If there is an entry for the destination MAC and VLAN, the packet gets forwarded to the associated interface stored in the MAC table. If there is no entry for the destination MAC and VLAN (lookup failure), cloud-native router floods the packet out all the interfaces in the bridge domain except the incoming interface.

By default:

- MAC table entries time out after 60 seconds
- The MAC table size is limited to 10,240 entries

You can configure the aging timeout and MAC table size during deployment by editing the **values.yaml** file under the **jcnr-vrouter** directory on the host server. We recommend that you do not change the default values.

You can see the MAC table entries:

- Using introspect at http://<host server IP>:8085/mac_learning.xml#Snh_FetchL2MacEntry

- Using the cRPD CLI and the command **show bridge mac-table**
- Using the CLI of the **contrail-tools** Pod with the command **purel2cli --mac show**

If you exceed the MAC limit, the counter **pkt_drop_due_to_mactable_limit** increments. You can see this counter through introspect at *http://<host server IP>:8085/Snh_AgentStatsReq*.

If you delete or disable an interface, 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 10240 seconds. You can see the time 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
```

Broadcast Rate Limiting

The broadcast rate limiting feature controls the rate of egress broadcast and multicast traffic on fabric interfaces. You specify the rate limit in bytes per second by setting the **fabricBMCastRateLimit** value in the **values.yaml** before deployment. The system applies the configured value to all fabric interfaces in cloud-native router. The maximum value you can set for the broadcast rate limit is 1000000 bytes per second.

If broadcast or multicast traffic rate exceeds the set limit, vRouter drops the traffic. You can see the counter values by running the dropstats command in the vRouter CLI. For example:

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


3

PART

Juniper Cloud-Native Router (JCNR) - Examples

User Pod Example - Kernel Access | 55

User Pod Example - Virtio Trunk | 65

User Pod Example - Kernel Access

SUMMARY

IN THIS SECTION

- [Overview | 55](#)
- [Before You Begin | 56](#)
- [Detailed Steps | 57](#)

In this topic, we show you how to add a user Pod with a kernel access interface to an instance of cloud-native router.

Overview

IN THIS SECTION

- [High-Level Steps | 55](#)

At a high level, the process of adding a user Pod to the cloud-native router requires that you ensure that a network attachment definition (NAD) exists and that you apply a Pod YAML file to your cloud-native router cluster. Throughout this example we use the **kubectl** command with various options. You must run this command on the CLI of the host server.

High-Level Steps

In this example we assume that this is the first user Pod that you are adding to your newly installed cloud-native router. Therefore, we create a new NAD on the cluster and then add the new user Pod.

Below we provide a list of the individual steps we take in this example. Each step in the list is a link to the detailed description of the step.

1. [View the vRouter interface list](#)

2. [Examine the example NAD YAML file](#)
3. [Apply the NAD to the cluster](#)
4. [Verify the NAD](#)
5. [Examine the example Pod YAML file](#)
6. [Apply the Pod to the cluster](#)
7. [Verify the Pod](#)
8. [View the updated vRouter interface list](#)

Before You Begin

IN THIS SECTION

- [Access the vRouter-Agent CLI | 56](#)

Access the vRouter-Agent CLI

The first and last steps of this example procedure are performed on the CLI of the vRouter-agent. We recommend that you open two SSH (terminal) sessions to the host server. You can use one session to run the CLI commands on the vRouter-agent and the other session to run the **kubect**l commands that deploy the NAD and the Pod on the cluster.

NOTE: To make it easy to copy and paste commands from here to your system, we do not include paths or shell prompts from the host server in the command listings.

To start, we access the CLI of the **contrail-vrouter-agent** container in the **contrail-vrouter-masters** Pod.

In one terminal enter the command:

```
kubectl get pods -n contrail
```

The output should be a single line that looks like:

```
NAME READY STATUS RESTARTS AGE
contrail-vrouter-masters-97v8z 3/3 Running 0 6h10m
```

This command gave us the name and specific instance hash of the vRouter Pod, `contrail-vrouter-masters-97v8z`. We use this name in the next command to access the vRouter CLI. The name of your vRouter Pod will have a different hash at the end. Use the Pod name from your system in place of `<contrail-vrouter-masters-hash>` in the command below.

Enter the command:

```
kubectl exec -n contrail -it <contrail-vrouter-masters-hash> -- bash
```

The output should be two lines that looks like:

```
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@jcnr1:/#
```

Note that the shell prompt should also have changed from whatever it was when you entered the command. On the system used to create this example, the prompt changed from `[root@jcnr1 ~]#` to `root@jcnr1:/#`. This change in prompt indicates that you have successfully connected to the CLI of the vRouter-agent.

You can now move to the detailed steps section to complete the example.

Detailed Steps

1. View the vRouter-agent interface list

In the terminal session connected to the vRouter-agent CLI enter the command

```
vif --list
```

The output looks like:

```
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 0
            RX packets:0 bytes:0 errors:0
            TX packets:2127928 bytes:510246290 errors:0
            Drops:0

vif0/1      PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
            Type:Physical HWaddr:3a:b2:ee:fe:a7:62
            Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
            RX queue packets:172174354904 errors:20998087137
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 7293345594 6559356797 7145384746
            Fabric Interface: eth_bond_bond0 Status: UP Driver: net_bonding
            Slave Interface(0): 0000:18:09.1 Status: UP Driver: net_iavf
            Slave Interface(1): 0000:18:05.1 Status: UP Driver: net_iavf
            Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
            RX packets:172172714121 bytes:33745848248728 errors:1642900
            TX packets:2272839360 bytes:4536582775436 errors:0
            Drops:80189427216

vif0/2      PCI: 0000:18:01.1 (Speed 1000, Duplex 1)
            Type:Physical HWaddr:a6:c1:0b:12:8c:44
            Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
            RX queue packets:108 errors:0
```

```

RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
Fabric Interface: 0000:18:01.1 Status: DOWN Driver: net_iavf
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:0 bytes:0 errors:108
TX packets:61278711540 bytes:15781059334468 errors:0
Drops:108

vif0/3 PCI: 0000:18:0d.1 (Speed 10000, Duplex 1)
Type:Physical HWaddr:7a:30:33:68:6c:70
Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
RX queue packets:91255 errors:626
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 120 397 109
Fabric Interface: 0000:18:0d.1 Status: UP Driver: net_iavf
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:2015 bytes:170734 errors:89240
TX packets:61279338241 bytes:15781182125402 errors:0
Drops:91703

vif0/4 PCI: 0000:86:00.0 (Speed 1000, Duplex 1)
Type:Physical HWaddr:40:a6:b7:0d:7b:b8
Vrf:65535 Flags:TcL2Vof QOS:-1 Ref:12
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
Fabric Interface: 0000:86:00.0 Status: DOWN Driver: net_i40e
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:0 bytes:0 errors:0
TX packets:61278779459 bytes:15781072646592 errors:0
Drops:0

```

As you can see, the vRouter-agent knows about three interfaces.

["Back to high-level steps" on page 55](#)

2. Examine the NAD YAML file

In this step, we look at a commented NAD file in YAML format. The comments start with a hash mark (#) and are highlighted in bold. Most of the values do not need to be changed since this file is an example of a NAD. The NAD defines the parameters of a virtual device that allows the Pod to connect to the network. You can use this example file on your cloud-native router only if you remove the comments from the file.

```
cat nad-kernel_access_bd3001.yaml
```

```

apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition

```

```

metadata:
  name: nad-vswitch-bd3001 #Name of the NAD
spec:
  config: '{
    "cniVersion":"0.4.0",
    "name": "nad-vswitch-bd3001", #This is the name of the NAD as it appears in the K8s
cluster.
    "capabilities":{"ips":true},
    "plugins": [
      {
        "type": "jcnr", #Always define as jcnr
        "args": {
          "instanceName": "vswitch", #Prefix for the name of the NAD within the cluster
          "instanceType": "virtual-switch", #What type of NAD is this?
          "bridgeDomain": "bd3001", #The name of the bridge domain
          "bridgeVlanId": "3001", #Which VLAN ID is allowed on this bridge domain
          "dataplane":"dpdk", #Which dataplane to use. Options are dpdk and kernel
          "mtu": "9000",
          "interfaceType":"veth" #Options are veth or virtio. For this example veth is the
correct option
        },
        "ipam": {
          "type": "static", #IP address type. Leave as static in 22.2 release
          "capabilities":{"ips":true}, #Does this ipam definition support IP addresses?
          "addresses":[
            {
              "address":"2001:db8:3001::10.30.0.1/64", #IPv6 address of the IFL
              "gateway":"2001:db8:3001::10.30.0.254" #IPv6 gateway for the IFL
            },
            {
              "address":"10.30.0.1/24", #IPv4 address for the IFL
              "gateway":"10.30.0.254" #IPv4 Gateway for the IFL
            }
          ]
        },
        "kubeConfig":"/etc/kubernetes/kubelet.conf"
      }
    ]
  }'
```

When you apply the NAD YAML file to the cluster, the virtual device becomes visible in the K8s cluster.

["Back to high-level steps" on page 55](#)

3. Apply the NAD to the Cluster

If you used the same file name for your version of the NAD file, you can run the following command on the host server CLI:

```
kubectl apply -f nad-kernel_access_vlan_3001.yaml
```

The output from the command looks like:

```
networkattachmentdefinition.k8s.cni.cncf.io/nad-vswitch-bd3001 created
```

["Back to high-level steps" on page 55](#)

4. Verify the NAD

On the host server CLI, issue the command:

```
kubectl get network-attachment-definitions
```

The output from the command looks like:

NAME	AGE
vswitch	1d
nad-vswitch-bd3001	3m47s

["Back to high-level steps" on page 55](#)

5. Examine the example Pod YAML file

Like the NAD YAML file, the Pod YAML file, or Pod definition, specifies the configuration of the user Pod that you want to create. In this example, we create a Pod that works with the pod1-vswitch-bd3001 NAD that we just applied. Again, a commented example file is shown below. You can use this file on your cloud-native router deployment only if you remove the comments from the file.

```
cat pod-kernel-access-vlan-3001.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name:   odu-kernel-pod-bd3001
  annotations:
    k8s.v1.cni.cncf.io/networks: pod1-vswitch-bd3001
spec:
```



```

containers:
  - name: odu-kernel-pod-bd3001
    image: svl-artifactory.juniper.net/junos-docker-local/warthog/pktgen19116:20210303
    imagePullPolicy: IfNotPresent
    command: ["/bin/bash", "-c", "sleep infinity"]
    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

```

["Back to high-level steps" on page 55](#)

6. Apply the Pod to the Cluster

If you used the same filename as shown above for the Pod YAML file, then you can run the following command on the host server CLI to apply the Pod to the cluster:

```
kubectl apply -f pod-kernel-access-vlan-3001.yaml
```

The output from the command looks like:

```
pod/odu-kernel-pod-bd3001 created created
```

["Back to high-level steps" on page 55](#)

7. Verify the Pod

On the host server CLI issue the following command to verify the Pod creation:

```
kubectl get pods odu-kernel-pod-bd3001
```

The output from the command looks like:

NAME	READY	STATUS	RESTARTS	AGE
odu-kernel-pod-bd3001	1/1	Running	0	58s

["Back to high-level steps" on page 55](#)

8. View the updated vRouter interface list

On the vRouter-agent CLI, issue the command:

```
vif --list
```

The output from the command looks like:

```
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 0 0
          RX packets:0 bytes:0 errors:0
          TX packets:14 bytes:1672 errors:0
          Drops:0

vif0/1    PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
          Type:Physical HWaddr:3a:b2:ee:fe:a7:62
          Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
          RX queue packets:971 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_bond0 Status: UP Driver: net_bonding
```

```

Slave Interface(0): 0000:18:05.1 Status: UP Driver: net_iavf
Slave Interface(1): 0000:18:09.1 Status: UP Driver: net_iavf
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:18 bytes:1256 errors:965
TX packets:26 bytes:2204 errors:0
Drops:989

vif0/2 PCI: 0000:18:01.1 (Speed 1000, Duplex 1)
Type:Physical HWaddr:a6:c1:0b:12:8c:44
Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
Fabric Interface: 0000:18:01.1 Status: DOWN Driver: net_iavf
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:0 bytes:0 errors:0
TX packets:37 bytes:2862 errors:0
Drops:0

vif0/3 PCI: 0000:18:0d.1 (Speed 10000, Duplex 1)
Type:Physical HWaddr:7a:30:33:68:6c:70
Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
RX queue packets:331 errors:0
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
Fabric Interface: 0000:18:0d.1 Status: UP Driver: net_iavf
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:7 bytes:590 errors:324
TX packets:37 bytes:2870 errors:0
Drops:331

vif0/4 PCI: 0000:86:00.0 (Speed 1000, Duplex 1)
Type:Physical HWaddr:40:a6:b7:0d:7b:b8
Vrf:65535 Flags:TcL2Vof QOS:-1 Ref:12
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
Fabric Interface: 0000:86:00.0 Status: DOWN Driver: net_i40e
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:0 bytes:0 errors:0
TX packets:37 bytes:2862 errors:0
Drops:0

vif0/5 Ethernet: jvknet1-0ea0f72
Type:Virtual HWaddr:02:00:00:b3:b9:a1
Vrf:0 Flags:L2Vof QOS:-1 Ref:10
RX queue packets:23 errors:0
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0

```

```
Vlan Mode: Access Vlan Id: 3001 OVlan Id: 3001
RX packets:19 bytes:1614 errors:4
TX packets:0 bytes:0 errors:0
Drops:24
```

The vRouter-agent now knows about five interfaces. This is because the Pod created the sub-interface and the parent interface. You can see above that the virtual VLAN interface, **vif0/4**, shows the parent interface as **vif0/3**. The interface, **vif0/3**, is a virtual interface with a name that included "net1" as defined in the NAD and Pod YAML files.

["Back to high-level steps" on page 55](#)

User Pod Example - Virtio Trunk

SUMMARY

IN THIS SECTION

- [Overview | 65](#)
- [Before You Begin | 66](#)
- [Detailed Steps | 68](#)

In this topic, we show you how to add a user Pod with a VLAN sub-interface to an instance of cloud-native router.

Overview

IN THIS SECTION

- [High-Level Steps | 66](#)

At a high level, the process of adding a user Pod to the cloud-native router requires that you ensure that a network attachment definition (NAD) exists and that you apply a Pod YAML file to your cloud-native router cluster. Throughout this example we use the **kubectl** command with various options. You must run this command on the CLI of the host server.

High-Level Steps

In this example we assume that this is the first user Pod that you are adding to your newly installed cloud-native router. Therefore, we will create a new NAD on the cluster and then add the new user Pod.

Below is a list of the individual steps we take in this example. Each step in the list is a link to the detailed description of the step.

1. [View the vRouter interface list](#)
2. [Examine the example NAD YAML file](#)
3. [Apply the NAD to the cluster](#)
4. [Verify the NAD](#)
5. [Examine the example Pod YAML file](#)
6. [Apply the Pod to the cluster](#)
7. [Verify the Pod](#)
8. [View the updated vRouter interface list](#)

Before You Begin

IN THIS SECTION

- [Access the vRouter-Agent CLI | 66](#)

Access the vRouter-Agent CLI

The first and last steps of this example procedure are performed on the CLI of the vRouter-agent. We recommend that you open two SSH (terminal) sessions to the host server. You can use one session to

run the CLI commands on the vRouter-agent and the other session to run the **kubectl** commands that deploy the NAD and the Pod on the cluster.

NOTE: To make it easy to copy and paste commands from here to your system, we do not include paths or shell prompts from the host server in the command listings.

To start, we access the CLI of the **contrail-vrouter-agent** container in the **contrail-vrouter-masters** Pod.

In one terminal enter the command:

```
kubectl get pods -n contrail
```

The output should be a single line that looks like:

```
NAME READY STATUS RESTARTS AGE
contrail-vrouter-masters-97v8z 3/3 Running 0 6h10m
```

This command gave us the name and specific instance hash of the vRouter Pod, **contrail-vrouter-masters-97v8z**. We use this name in the next command to access the vRouter CLI. The name of your vRouter Pod will have a different hash at the end. Use the Pod name from your system in place of **<contrail-vrouter-masters-hash>** in the command below.

Enter the command:

```
kubectl exec -n contrail -it <contrail-vrouter-masters-hash> -- bash
```

The output should be two lines that looks like:

```
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@jcnr1:/#
```

Note that the shell prompt should also have changed from whatever it was when you entered the command. On the system used to create this example, the prompt changed from **[root@jcnr1 ~]#** to **root@jcnr1:/#**. This change in prompt indicates that you have successfully connected to the CLI of the vRouter-agent.

You can now move to the detailed steps section to complete the example.

Detailed Steps

1. View the vRouter-agent interface list

In the terminal session connected to the vRouter-agent CLI enter the command

```
vif --list
```

The output looks like:

```
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 0
            RX packets:0 bytes:0 errors:0
            TX packets:2127928 bytes:510246290 errors:0
            Drops:0

vif0/1      PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
            Type:Physical HWaddr:3a:b2:ee:fe:a7:62
            Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
            RX queue packets:172174354904 errors:20998087137
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 7293345594 6559356797 7145384746
            Fabric Interface: eth_bond_bond0 Status: UP Driver: net_bonding
            Slave Interface(0): 0000:18:09.1 Status: UP Driver: net_iavf
            Slave Interface(1): 0000:18:05.1 Status: UP Driver: net_iavf
            Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
```

```

RX packets:172172714121  bytes:33745848248728  errors:1642900
TX packets:2272839360  bytes:4536582775436  errors:0
Drops:80189427216

vif0/2  PCI: 0000:18:01.1 (Speed 1000, Duplex 1)
        Type:Physical HWaddr:a6:c1:0b:12:8c:44
        Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
        RX queue  packets:108 errors:0
        RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
        Fabric Interface: 0000:18:01.1  Status: DOWN  Driver: net_iavf
        Vlan Mode: Trunk  Vlan: 100 200 300 500 3001-3004 3201-3250
        RX packets:0  bytes:0 errors:108
        TX packets:61278711540  bytes:15781059334468  errors:0
        Drops:108

vif0/3  PCI: 0000:18:0d.1 (Speed 10000, Duplex 1)
        Type:Physical HWaddr:7a:30:33:68:6c:70
        Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
        RX queue  packets:91255 errors:626
        RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 120 397 109
        Fabric Interface: 0000:18:0d.1  Status: UP  Driver: net_iavf
        Vlan Mode: Trunk  Vlan: 100 200 300 500 3001-3004 3201-3250
        RX packets:2015  bytes:170734 errors:89240
        TX packets:61279338241  bytes:15781182125402  errors:0
        Drops:91703

vif0/4  PCI: 0000:86:00.0 (Speed 1000, Duplex 1)
        Type:Physical HWaddr:40:a6:b7:0d:7b:b8
        Vrf:65535 Flags:TcL2Vof QOS:-1 Ref:12
        RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
        Fabric Interface: 0000:86:00.0  Status: DOWN  Driver: net_i40e
        Vlan Mode: Trunk  Vlan: 100 200 300 500 3001-3004 3201-3250
        RX packets:0  bytes:0 errors:0
        TX packets:61278779459  bytes:15781072646592  errors:0
        Drops:0

```

As you can see, the vRouter-agent knows about five interfaces.

["Back to high-level steps" on page 66](#)

2. Examine the NAD YAML file

In this step, we look at a commented NAD file in YAML format. The comments start with a hash mark (#) and are highlighted in bold. Most of the values do not need to be changed since this file is an example of a NAD. The NAD defines the parameters of a virtual device that allows the Pod to

connect to the network. You can use this example file on your cloud-native router only if you remove the comments from the file.

```
cat nad-virtio-trunk1.yaml
```

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vswitch-trunk9 #This is the name of the NAD as it appears in the K8s cluster.
spec:
  config: '{
    "cniVersion": "0.4.0",
    "name": "vswitch-trunk9", # Name of the NAD
    "type": "jcnr", # Always define as jcnr
    "args": {
      "instanceName": "vswitch",
      "instanceType": "virtual-switch",
      "dataplane": "dpdk", #Leave as DPDK for this example. Options are DPDK or kernel
      "vlanIdList": "3001, 3002, 3003, 3210" #List of allowed VLANs. You can allow a range
like this: 100-200
    },
    "kubeConfig": "/etc/kubernetes/kubelet.conf"
  }'
```

When you apply the NAD YAML file to the cluster, the virtual device becomes visible in the K8s cluster.

["Back to high-level steps" on page 66](#)

3. Apply the NAD to the Cluster

If you used the same file name for your version of the NAD file, you can run the following command on the host server CLI:

```
kubectl apply -f nad-virtio-trunk1.yaml
```

The output from the command looks like:

```
networkattachmentdefinition.k8s.cni.cncf.io/nad-virtio-trunk1.yaml created
```

["Back to high-level steps" on page 66](#)

4. Verify the NAD

On the host server CLI, issue the command:

```
kubectl get network-attachment-definitions
```

The output from the command looks like:

NAME	AGE
vswitch	25h
vswitch-trunk1	54s

["Back to high-level steps" on page 66](#)

5. Examine the example Pod YAML file

Like the NAD YAML file, the Pod YAML file, or Pod definition, specifies the configuration of the user Pod that you want to create. In this example, we create a Pod that works with the vswitch-trunk1 NAD that we just applied. Again, a commented example file is shown below. You can use this file on your cloud-native router deployment only if you remove the comments from the file.

```
cat pod-virtio-trunk1.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-virtio-trunk1 #
  annotations:
    k8s.v1.cni.cncf.io/networks: vswitch-trunk1
spec:
  containers:
    - name: pod-virtio-trunk1
      image: svl-artifactory.juniper.net/junos-docker-local/warthog/pktgen19116:20210303
      imagePullPolicy: IfNotPresent
      securityContext:
        privileged: true
      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

```

["Back to high-level steps" on page 66](#)

6. Apply the Pod to the Cluster

If you used the same filename as shown above for the Pod YAML file, then you can run the following command on the host server CLI to apply the Pod to the cluster:

```
kubectl apply -f pod-virtio-trunk1.yaml
```

The output from the command looks like:

```
pod/pod-virtio-trunk1 created
```

["Back to high-level steps" on page 66](#)

7. Verify the Pod

On the host server CLI issue the following command to verify the Pod creation:

```
kubectl get pods pod-virtio-trunk1
```

The output from the command looks like:

NAME	READY	STATUS	RESTARTS	AGE
pod-virtio-trunk1	1/1	Running	0	1m21s

["Back to high-level steps" on page 66](#)

8. View the updated vRouter interface list

On the vRouter-agent CLI, issue the command:

```
vif --list
```

The output from the command looks like:

```
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
            RX packets:0 bytes:0 errors:0
            TX packets:205 bytes:74417 errors:0
            Drops:0

vif0/1      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:3120 errors:0
            RX queue errors to lcore 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
            Vlan Mode: Trunk Vlan: 751-755
            RX packets:0 bytes:0 errors:7020
            TX packets:0 bytes:0 errors:0
            Drops:7020
```

```

vif0/2    PCI: 0000:81:09.0 (Speed 10000, Duplex 1)
          Type:Workload HWaddr:ca:ce:fc:d3:28:1e
          Vrf:0 Flags:L2Vof QOS:-1 Ref:7
          RX queue  packets:3120 errors:0
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
          Fabric Interface: 0000:81:09.0 Status: UP Driver: net_iavf
          Vlan Mode: Access Vlan Id: 756 OVlan Id: 756
          RX packets:7020 bytes:1496820 errors:0
          TX packets:0 bytes:0 errors:0
          Drops:7215

vif0/3    PMD: vhostnet1-8ca7c251-481b-48
          Type:Virtual HWaddr:00:99:99:99:33:09
          Vrf:65535 Flags:L2 QOS:-1 Ref:10
          RX queue errors to lcore 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

vif0/4    Virtual: vhostnet1-8ca7c251-481b-48.3003 Vlan(o/i)(,S): 3003/3003 Parent:vif0/3
          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
          RX packets:0 bytes:0 errors:0
          TX packets:0 bytes:0 errors:0
          Drops:0

vif0/5    PMD: vhostnet1-35fee25e-7646-4281-ab
          Type:Virtual HWaddr:02:00:00:ac:88:fd
          Vrf:65535 Flags:L2 QOS:-1 Ref:13
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
          Vlan Mode: Trunk Vlan: 3001-3003 3210
          RX packets:0 bytes:0 errors:0
          TX packets:1 bytes:64 errors:0
          Drops:0
          TX port  packets:0 errors:1

```

The vRouter-agent now knows about five interfaces. This is because the Pod created the trunk interface. You can see above that the vif0/5 interface shows VLAN mode as Trunk and the correct list of VLANs that are allowed to pass.

["Back to high-level steps" on page 66](#)