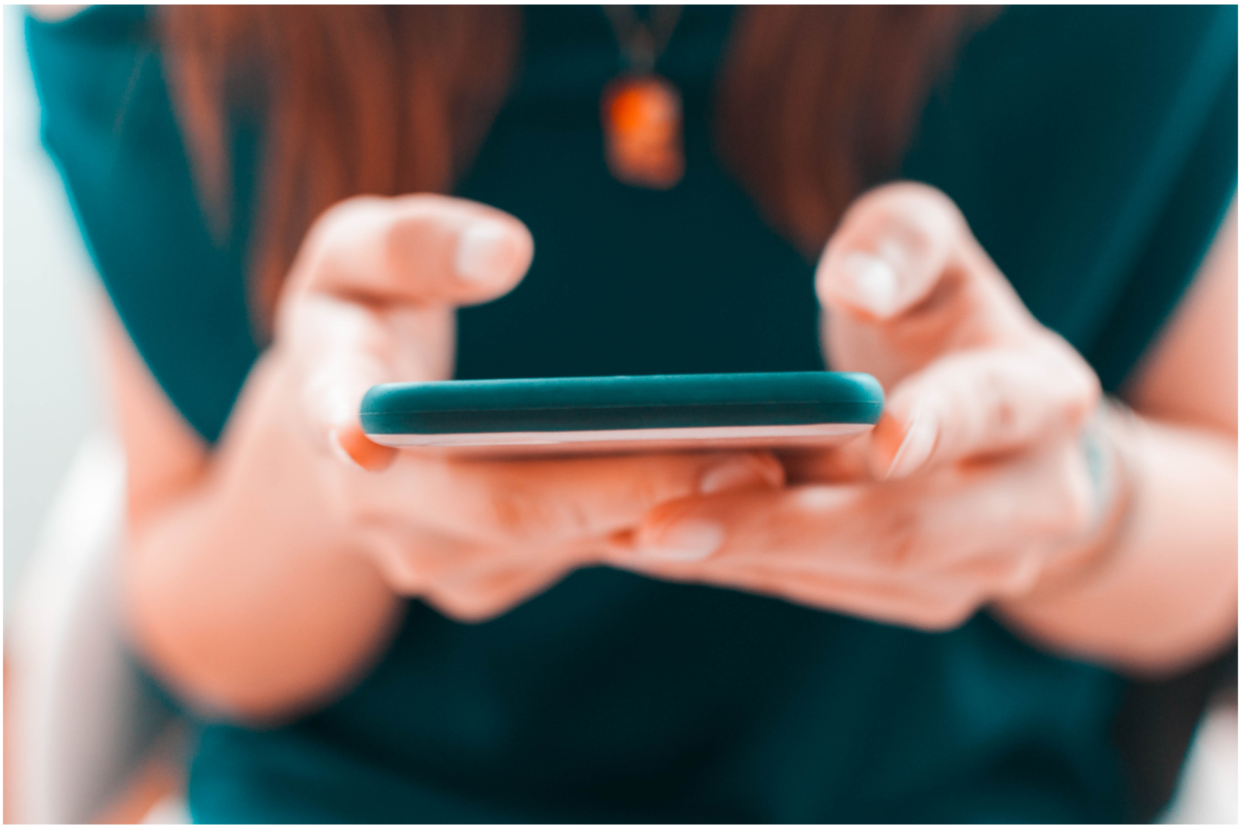
DESIGNING AND DEPLOYING CARRIER-GRADE, CLOUD-NATIVE INFRASTRUCTURE FOR TELCO AND EDGE CLOUD



By Kashif Nawaz

Published by Juniper Networks Books
Authors: Kashif Nawaz, Venu Kumar Kolli, Sohail Arham, Guy Davies, Pradeep H Krishnamurthy
Technical Reviewers: TBD
Editor in Chief: Patrick Ames
Version History: v1, July 2023

About the Authors

Kashif Nawaz
Kashif Nawaz is a Staff Consultant at Juniper Networks in the Global Services Group. His recent engagements have provided him with opportunities to design, deploy, and support telco cloud solutions with major telecom and communication service providers across Europe and the Middle East. He is the lead author and principal contributor to this book.

Venu Kumar Kolli
Venu is a solutions test lead for Contrail Networking at Juniper Networks. He has multiple years of experience in testing scalable datacenters and enterprise SDN topologies with Contrail as a network plugin. He is the lead co-contributor of Chapter 7 and 12, and contributed to Chapter 6, while also reviewing the contents of Chapters 1,2,6, and7.

Sohail Arham
Sohail Arham is Senior Sales Consultant (focused on cloud technologies) in Juniper Networks Sales organization. He has vast experience of telco cloud design, deployment and operational support across North America, Europe and Middle East. He is also a reviewer of this book and provided input for Chapters 6 and 11 as well.

Guy Davies
Guy is currently an Architect within Juniper Global Service Provider Architecture Organization and has rich experience of designing, deploying, and supporting telco cloud solutions with major telecom providers in Europe and the UK. Guy is a reviewer of this book and helped in editing/formatting.. Guy also contributed for Chapter 6's BGPaaS section. He is the author of Designing and Developing Scalable IP Architectures and several published other Day One books from Juniper Networks.

Pradeep H Krishnamurthy
Pradeep H Krishnamurthy is currently a product developer with the Contrail CN2 engineering team and leading SRIOV integration with CN2. Pradeep reviewed Chapter 8 and contributed to its CN2-IPAM section.

Author's Acknowledgments

Kashif Nawaz: I dedicate this book to my parents who gave me the greatest gift of a continuous appetite for learning, and to my wife who stood by me during hardships. Due to her unflinching support, I was able to concentrate and completed this book. I acknowledge the efforts of co-contributors, especially Venu Kamar Kolli, for his great support. I also acknowledge the efforts of my son Abdul Shazif Nawaz who helped me by designing illustrations within this book.

Chapter Summary

- Chapter 1 describes background information, problem statement, and proposed solution.
- Chapter 2 describes Juniper Network Cloud Native Contrail Networking (CN2) high-level architecture and its components.
- Chapter 3 describes Canonical Metal as a Service (MAAS) and preparing the physical and virtual infrastructure via MAAS using Infrastructure as a Code (IaC) approach.
- Chapter 4 covers K8S worker node for Data Plane Development Kit (DPDK) deployment.
- Chapter 5 covers preparing kube-spray for K8s deployment, deploying K8S cluster, creating HAProxy on deployer-node, preparing manifest for CN2 deployment, and finally deploying CN2.
- Chapter 6 discusses various CN2 custom resources definition along with working examples of each feature.
- Chapter 7 discusses various options for extending K8S workloads connectivity to the outer world.
- Chapter 8 describes Single Root I/O Virtualization (SR-IOV) and how CN2 handles SRIOV VFs plumbing into K8S workloads.
- Chapter 9 describes requirements for cloud native persistent storage and its implementation via ROOK (a K8S Operator).
- Chapter 10 describes virtual machines use case in K8S and its implementation via kubevirt.
- Chapter 11 covers two use cases using CN2 constructs:Micro Segmentation in IT (Information Technology) cloud environment, and LTE (Long Term Evolution) Traffic Flow simulation in telco cloud environment.

Table of Contents

Table of Figures

Chapter 1

Network Function Virtualization Challenges and Future Prospects

Background

In the recent past, Infrastructure as a Service (IaaS) (e.g., OpenStack) has become the de-facto standard for Network Function Virtualized Infrastructure (NFVI) solutions where Network Functions are deployed as Virtual Machines and these functions are named as Virtual Network Functions (VNF). However, with the advent of container technology, VNF vendors started shipping their products in Containers which are also called Containerized Network Function (CNF). Many telecoms' providers have adopted containerized technology and deployed Kubernetes (K8s) clusters over IaaS VMs in their data centers. Juniper Networks' Contrail SDN Controller has been used by large-scale enterprises and communication service providers across the globe to provide overlay networking to workloads deployed on Open Stack. This overlay networking provides several major advantages including:

- Isolation of tenant traffic from each other and from the management and control traffic in the underlay.
- Highly flexible network constructs to enable the creation of complex networking architectures for IaaS workloads.
- High throughput for carrier grade applications / VNFs by using Data Plane Development Kit (DPDK) based forwarding plane.
- Strong automation capabilities enabling telecoms providers' staff to focus on designing new services rather than operating existing ones.

Problem Statement

Deploying nested K8s clusters inside IaaS VMs introduced additional challenges; for example, performance concerns and nested networking complexities where K8s network plane adds another encapsulation in addition to IaaS network plane encapsulation. Inherited problems coupled with nested K8s deployment over IaaS VMs are driving telecom service providers to deploy CNF based NFVI solution over bare metal servers.

Moreover, the advent of 5G Cloud Native RAN/C-RAN has ushered an era where 5G Virtual Control Unit (vCU) and virtual Distribution Unit (vDU) will be deployed in small clusters consisting of a small number (typically fewer than 10) of compute nodes distributed across potentially hundreds of sites (depending upon the geography of the countries and the distribution of the customer base of the telecom provider). Using IaaS to bootstrap and manage bare-metal server infrastructure for small K8s clusters (to be used for vCUs & vDUs) is not a viable option due to infrastructure requirement of IaaS itself. Therefore, deploying 5G vCUs and vDUs K8s clusters over bare metal servers is preferred.

Many open-source community projects, as well as Linux distro vendors, offer tools to manage bare metal sever infrastructure efficiently (for example metal3, RedHat Advance Cluster Manager (ACM), and Canonical Metal as a Service (MAAS). However, running CNFs in a K8s cluster over bare metal servers is not enough until several characteristics can be met:

- High-performance mode networking capabilities must be provided to containerized workloads.
- Cloud-native persistent storage capabilities must be added to the K8s cluster so that CNF application data can survive the failure of a worker node.
- Application data storage should be accessible in case a worker node is broken, or application/ container is broken. It implies that data storage should be central so that failure of any component should not impact its availability, but data storage should also be distributed as well for robust access and to achieve resiliency.

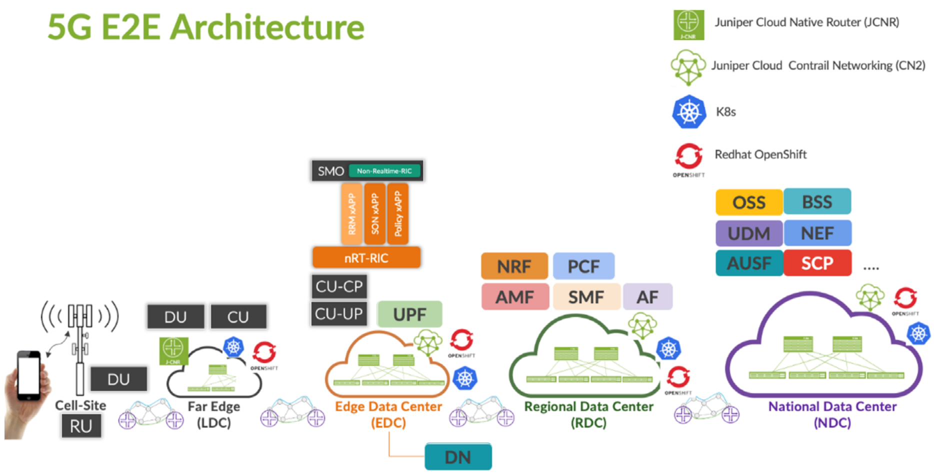


Figure 1 *5G RAN & Telco Cloud High Level Architecture*
Courtesy to Sohail Arham (Juniper Networks) for above diagram

Proposed Solution

In this book, we use Canonical MAAS to bootstrap physical and virtual infrastructure to host K8s clusters. Canonical MAAS (Metal as a Service) offers an "Infrastructure as a Code" way for lifecycle management of bare-metal servers and virtual infrastructure. For those who prefer to use a GUI, MAAS provides a nice and easy to use GUI and for terminal lovers MAAS offers feature set rich CLI commands and API calls.

Juniper Networks has recently released Cloud Native Contrail Networking (CN2), a SDN Controller, which can be integrated with one or more K8s clusters (as the CNI) and it offers a rich feature-set which are considered essentials for Carrier-Grade containerized applications.

Juniper CN2 also offers a DPDK vRouter to meet throughput needs of performance-oriented virtualized and containerized workloads. If a workload supports DPDK Poll-Mode Driver (PMD), then the attachment of such a pod with K8s worker nodes, DPDK PMD-bound interfaces could satisfy the network throughput requirements. If containerized workload does not support DPDK but still requires high throughput network interfaces, this can be achieved by plumbing SRIOV VFs into CNF Pods. Juniper CN2 also supports integration with SR-IOV CNI via the Multus Meta-CNI, which solves the above-described problem by plumbing SR-IOV VFs directly into the workloads.

To provide persistent storage to containerized workloads in case of worker node failures, Ceph is the first choice. Ceph is a Software Defined Storage, which is already widely deployed in NFVI solutions.

Chapter 2

Cloud Native Contrail Networking (CN2) Architecture

This chapter describes Juniper Networks Cloud Native Contrail Networking (CN2) high-level architecture and its components. Reference documentation is available here: (https://www.juniper.net/documentation/us/en/software/cn-cloud-native22.1/cn-cloud-native-K8s-install-and-lcm/topics/concept/cn-cloud-native-contrail-components.html).

CN2 Overview

Juniper Cloud Native Contrail Networking (CN2) is a Software Defined Controller which provides Container Network Interface functionality (CNI) for Cloud-Native workloads across various Kubernetes distributions (for example OpenShift, upstream Kubernetes, and EKS).

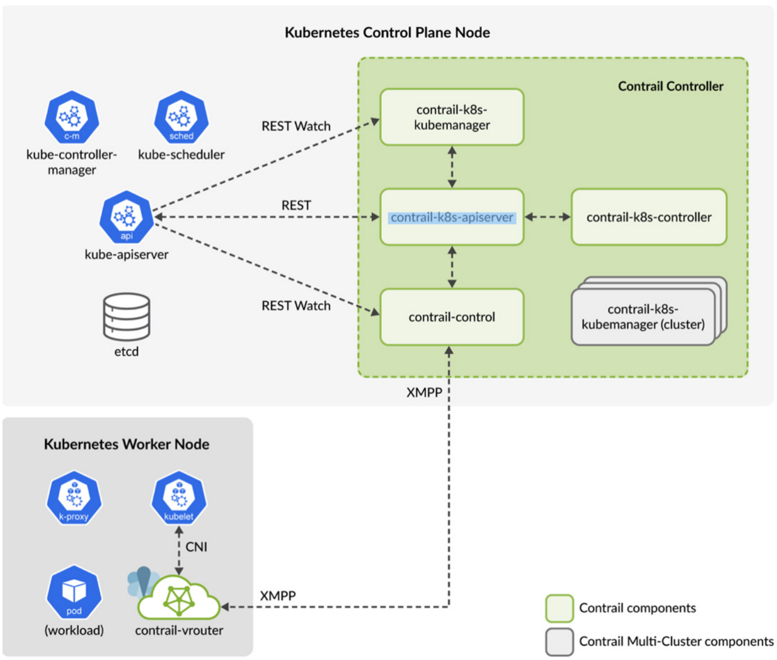


Figure 2 Cloud Native Contrail Networking High Level Architecture
Courtesy to Juniper Networks for above diagram

CN2 cluster consists of Configuration Plane, Control Plane, and Data Plane components.

Configuration Plane

At a high level, the CN2 Configuration plane listens to K8s API call and translates those calls into CN2 constructs. The CN2 Configuration plane further has three sub-components which run on the K8s Master node. Contrail-K8s-kubemanager listens to events created by K8S API servers and invokes Contrail-K8s-apiserver to create corresponding CN2 resource. Contrail-K8s-controller ensures that required intent for CN2 resources is in place.

Control Plane

Contrail-control connects to CN2 vRouter agent on worker nodes to distribute configuration and program control-plane routing information via XMPP messages. It also runs Multi-Protocol-Border Gateway Protocol (MP-BGP) to exchange routing information between Contrail-Control nodes and with the Software Defined Gateway (SDN-GW) routers.

Data Plane

Contrail-vrouter-nodes reside on the worker node and consist of the agent microservice and the vRouter forwarding component. The agent microservice performs control-related functions. The agent microservice receives configurations from the control node and converts that configuration for the forwarding component. It also performs firewall-rule processing, sets up flows for the forwarding component, and interfaces with the Kubernetes CNI plugin. The agent microservice generates routes as workloads (Pods or VMs) come up on the node and exchanges them with control nodes for distribution. It also withdraws routes as workloads are terminated. The vRouter supports multiple forwarding modes, for example, Kernel mode, DPDK mode, and Smart NIC vRouter.

Contrail-vRouter-masters provides the same functionality as contrail-vRouter-nodes but runs on K8s master nodes.

Chapter 3

Infrastructure Bootstrapping with Canonical MAAS

This chapter describes Canonical Metal as a Service (MAAS) and preparing the physical/ virtual infrastructure via MAAS using the Infrastructure as Code (IaC) approach. Although we have used Canonical MAAS for infrastructure bootstrapping and the Ansible deployer for K8S and CN2 cluster bring up, however if someone is interested to use RedHat OpenShift with Assisted Installer and Redfish API for infrastructure bootstrapping, then then this publication can be used: https://github.com/kashif-nawaz/RHOCP_CN2_AI_NMState_Redfish_API. All the CN2 custom resources definition and working example given in this book should be valid for RedHat OpenShift environment as it is, or with slight modification.

Metal As a Service (MAAS) Introduction

This chapter will extensively use Canonical Metal as Service MAAS (https://maas.io) to manage bare metal/virtual infrastructure. MAAS has a feature rich API-driven CLI and GUI to manage the bare metal/virtual infrastructure.

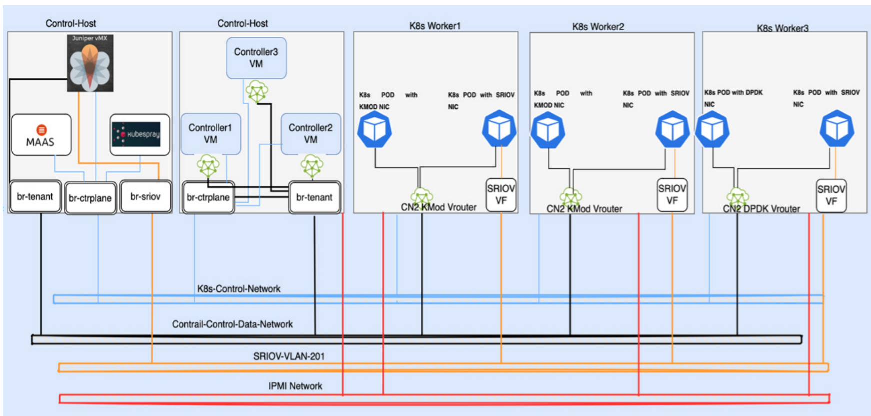


Figure 3 *Lab Diagram*

Note: For purposes of demonstration, we have deployed K8s controllers as VMs on a single bare metal server, but we do not recommend hosting K8s controllers in production environments because of their high availability requirement.

Preparing the Control-Host

The Control-host is a KVM host and hosts the MAAS VM, deployer-node VM ,and K8s Controller VMs. It is assumed that the Control-host is already bootstrapped with your favorite Linux distro (we are using Ubuntu 20.04). Let's get started by installing the required packages on the Control-Host:

```
sudo apt -y install bridge-utils cpu-checker libvirt-clients libvirt-
daemon qemu qemu-kvm cloud-image-utils whois libguestfs-tools vlan
libvirt-daemon-system hwinfo virtinst
```

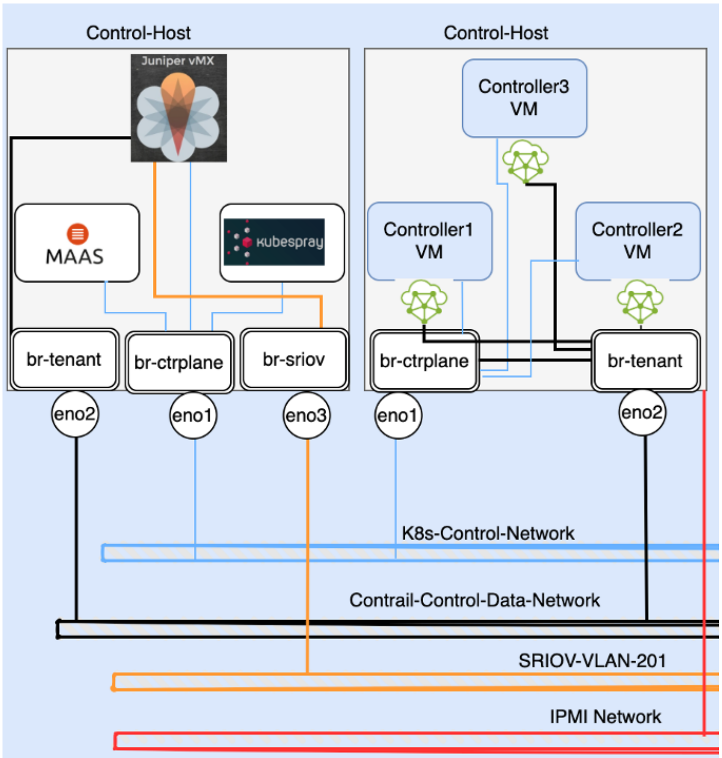


Figure 4 *Control Host Network Connectivity*

All the VMs hosted on control-host and their network topology are shown in Figure 4. Although a single KVM host has been used as control-host but to avoid cluttering in Figure 4, it is bisected into two parts, one part showing K8s Controller VMs, and other part showing utility VMs.

To achieve the above depicted network connectivity for Control-Host, the required net plane configuration is here provided. Three Linux bridges (br-ctrplane bound with eno1, br-Tenant bound with eno2, and br-sriov is bound with eno3 interface) are created in the following configuration snippet:

```
network:
    ethernets:
        eno1:
            dhcp4: no
            optional: true
        eno2:
            dhcp4: no
            optional: true
            match:
                macaddress: 90:b1:1c:44:f2:fb
            mtu: 9000
        eno3:
            dhcp4: no
            optional: true
    bridges:
        br-ctrplane:
            interfaces: [eno1]
            addresses: [192.168.24.10/24]
            gateway4: 192.168.24.1
            nameservers:
               addresses: [1.1.1.1, 8.8.8.8]
        br-Tenant:
            interfaces: [eno2]
            addresses: [192.168.5.10/24]
            mtu: 9000
        br-siov:
            interfaces: [eno3]
            addresses: [192.168.201.253/24]
    version: 2
```

Create MAAS VM

MAAS VM is created on the KVM Server (Control-Host). To bring up MAAS VM, first we need to get the Ubuntu Cloud Image.

```
wget 'http://cloud-images-archive.ubuntu.com/releases/focal/
release-20210921/ubuntu-20.04-server-cloudimg-amd64-disk-kvm.img'
```

Once the image is downloaded, we need to prepare the cloud-init configuration so that VM can instantiated without manual intervention. More about cloud-init can be found here: (https://cloudinit.readthedocs.io/en/latest/)

```
cat > maas_cloud_init.cfg <<END_OF_SCRIPT
#cloud-config
package_upgrade: true
hostname: maas
fqdn: mass.knawaz.lab.jnpr
manage_etc_hosts: true
users:
  - name: ubuntu
    lock_passwd: false
    shell: /bin/bash
    ssh_pwauth: true
    home: /home/ubuntu
    sudo: ['ALL=(ALL) NOPASSWD:ALL']
    ssh-authorized-keys:
      - ssh-rsa "key"
  - name: contrail
    lock_passwd: false
    shell: /bin/bash
    home: /home/contrail
    ssh_pwauth: true
    sudo: ['ALL=(ALL) NOPASSWD:ALL']
    ssh-authorized-keys:
      - ssh-rsa "key"
chpasswd:
  list: |
     ubuntu:password786
  expire: False
chpasswd:
  list: |
     contrail:password786
  expire: False

write_files:
  - path:  /etc/netplan/50-cloud-init.yaml
```

```
    permissions: '0644'
    content: |
         network:
           version: 2
           renderer: networkd
           ethernets:
             ens3:
               addresses: [192.168.24.40/24]
               gateway4: 192.168.24.1
               nameservers:
                 addresses: [8.8.8.8]

runcmd:
 - [sudo, ifconfig, IFNAME, up]
 - [sudo, netplan, generate]
 - [sudo, netplan, apply]
 - [sudo, sed ,-i, 's/PasswordAuthentication no/PasswordAuthentication
yes/g', /etc/ssh/sshd_config]
 - [sudo, systemctl, restart, sshd]
 EOF
 END_OF_SCRIPT
```

Once the cloud-init configuration is defined, we need to create an image from it.

```
cloud-localds -v  maas_cloud_init.img maas_cloud_init.cfg
```

Now let's create a disk image (maas.qcow2) in qcow2 (Qemu Copy on Write) format with the capacity of 200G and Ubuntu cloud image downloaded in the previous step, will be written into it.

```
qemu-img create -b ubuntu-20.04-server-cloudimg-amd64-disk-kvm.img  -f
qcow2 -F qcow2 /var/lib/libvirt/images/maas.qcow2 200G
```

Finally let's create MAAS VM. A cloud-init.img will be used as the CDROM image and the base disk image will be maas.qcow2 which is generated in the previous step.

```
virt-install --name maas \
  --virt-type kvm --memory 4096  --vcpus 4 \
  --boot hd,menu=on \
  --disk path=maas_cloud_init.img,device=cdrom \
  --disk path=/var/lib/libvirt/images/maas.qcow2,device=disk \
  --os-type=Linux \
  --os-variant=ubuntu20.04 \
```

```
  --network bridge:br-ctrplane \
  --graphics vnc,listen=0.0.0.0 --noautoconsole

Starting install...
Domain creation completed
```

Now just wait while the MAAS VM is getting prepared.

Putting MAAS into Action

Acknowledgment: We have taken help from the work done by an open-source contributor Anton Smith whose work is available on MAAS-Setup (https://github.com/antongisli/maas-baremetal-K8s-tutorial/blob/main/maas-setup.sh) .

The following sequence will set up the Metal as a Service, which will be used to manage the remaining infrastructure. We have added instructions on how to configure 802.3ad Bond on physical servers using MAAS CLI/ API calls.

Update & Upgrade

The following steps need to be executed inside MAAS VM:

```
sudo apt-get update && sudo apt-get upgrade -y
reboot
```

Install MAAS

```
sudo snap install jq
sudo snap install --channel=3.2/edge maas
maas (3.2/edge) 3.2.7-12039-g.0cfb6dacf from Canonical✓ installed
```

MAAS uses PostgreSQL to store required data and Canonical provides a separate snap, called maas-test-db, which contains a PostgreSQL database for use in testing and evaluating MAAS. See the Canonical documentation (https://maas.io/docs/how-to-do-a-fresh-install-of-maas) for the production grade deployment of PostgreSQL for MAAS.

```
sudo snap install maas-test-db
maas-test-db (3.3/stable) 14.2-29-g.ed8d7f2 from Canonical✓ installed
```

Setup MAAS

```
export INTERFACE=$(ip route | grep default | cut -d ' ' -f 5)
export IP_ADDRESS=$(ip -4 addr show dev $INTERFACE | grep -oP
'(?<=inet\s)\d+(\.\d+){3}')
sudo maas init region+rack --database-uri maas-test-db:/// --maas-url
http://${IP_ADDRESS}:5240/MAAS
```

Upon successful execution of the above command, the following text will be display on the terminal screen.

```
MAAS has been set up

 If you want to configure external authentication or use
MAAS with Canonical RBAC, please run

sudo maas configauth
To enable TLS for secured communication
sudo maas config-tls enable

To create admins when not using external authentication
sudo maas createadmin
```

Setup MAAS Admin User

```
sudo maas createadmin --username admin --password admin --email admin@
maas
export APIKEY=$(sudo maas apikey --username admin)
```

MAAS Admin login

```
export INTERFACE=$(ip route | grep default | cut -d ' ' -f 5)
export IP_ADDRESS=$(ip -4 addr show dev $INTERFACE | grep -oP
'(?<=inet\s)\d+(\.\d+){3}')
export SUBNET=$(ip -4 -br a | grep $INTERFACE | awk '{print $3}')
export APIKEY=$(sudo maas apikey --username admin)
maas login admin 'http://localhost:5240/MAAS/' $APIKEY
echo $SUBNET
echo $IP_ADDRESS
```

Setup MAAS SSH-Key

```
ssh-keygen -q -t rsa -N '' -f ~/.ssh/id_rsa <<<y >/dev/null 2>&1
maas admin sshkeys create key="$(cat /home/contrail/.ssh/id_rsa.pub)"
```

Setup MAAS Networks

See also About MAAS DHCP Snippets

In this setup we are using two networks:

- Contrail-control-data-network subnet 192.168.5.0/24, Untagged, MTU 9000 bytes.
- K8s-Control-network subnet 192.168.24.0/24, Untagged, MTU default 1500 bytes.

The MAAS VM is only connected with the K8s-Control-network network, which is extended to the MAAS VM via br-ctrplane from Control-host. Subsequently, this network will be used to provide IP addresses to the VMs/bare metal servers and will also be used for PXE boot of target nodes. For the sake of simplicity, let's call this network the provisioning network. This network will also be used for the K8s Control-Plane communication and is labeled as the "K8s-Control-network" in the topology diagram. Hence, MAAS will provide IPs to target nodes via DHCP from this subnet so it will be described as a MAAS managed network. To know more about MAAS managed networks (https://maas.io/docs/about-maas-networks#heading--Its-always-DHCP).

The MAAS VM is only connected with the K8s-Control-network network, which is extended to the MAAS VM via br-ctrplane from Control-host. Subsequently, this network will be used to provide IP addresses to the VMs/bare metal servers and will also be used for PXE boot of target nodes. For the sake of simplicity, let's call this network the provisioning network. This network will also be used for the K8s Control-Plane communication and is labeled as the "K8s-Control-network" in the topology diagram. Hence, MAAS will provide IPs to target nodes via DHCP from this subnet so it will be described as a MAAS managed network. To know more about MAAS managed networks (https://maas.io/docs/about-maas-networks#heading--Its-always-DHCP).

Another network labeled as Contrail-control-data-network in the topology diagram will not be managed by MAAS nor extended to MAAS VM. However, MAAS will use this subnet to assign static IPs to VMs and BareMetal servers in this setup. This network will be used for control and data plane communication between CN2 nodes. See more about MAAS unmanaged networks (https://maas.io/docs/about-networking#heading--about-unmanaged-subnets).

Update Provisioning Network

In the following configuration snippet, we are updating DNS forwarder and gateway on Provisioning Network (K8s-Control-network):

```
FABRIC_ID=$(maas admin subnet read "$SUBNET" | jq -r ".vlan.fabric_id")

VLAN_TAG=$(maas admin subnet read "$SUBNET" | jq -r ".vlan.vid")

PRIMARY_RACK=$(maas admin rack-controllers read | jq -r ".[] | .system_
id")

SUBNET_ID=$(maas admin subnets read | jq '.[] |
select(."cidr"=="192.168.24.0/24") | .["id"]')

maas admin subnet update $SUBNET gateway_ip=192.168.24.1

maas admin subnet update $SUBNET dns=1.1.1.1
```

Setup IP Address Ranges on Provisioning Network

In the following configuration snippet, we are creating one dynamic range and two reserved ranges on the Provisioning Network (K8s-Control-network). IP addresses that fall in reserved ranges will not be available to MAAS-managed DHCP on the Provisioning Network, and IP addresses fall in the dynamic range are used during PXE boot of MAAS-managed nodes. IP addresses which do not fall in reserved and dynamic ranges will be used to allocate IP addresses to MAAS-managed nodes via DHCP.(See also IP Ranges.)

```
maas admin ipranges create type=reserved start_ip=192.168.24.1 end_
ip=192.168.24.50

maas admin ipranges create type=reserved start_ip=192.168.24.195 end_
ip=192.168.24.254

maas admin ipranges create type=dynamic start_ip=192.168.24.51 end_
ip=192.168.24.81

maas admin vlan update $FABRIC_ID $VLAN_TAG dhcp_on=True primary_
rack=$PRIMARY_RACK

maas admin spaces create name=oam-space

maas admin vlan update  $FABRIC_ID $VLAN_TAG space=oam-space
```

Setting up Contrail-control-data-network in MAAS

Since this network is not attached with MAAS VM, it will be created from scratch in MAAS (but no interface on MAAS VM will be configured on this network).

```
maas admin fabrics create name=vrouter-transport-fabric

maas admin spaces create name=vrouter-transport-space

FABRIC_ID=$(maas admin fabrics read | jq '.[] | select(."name"=="vrouter-
transport-fabric") | .["id"]')

maas admin vlan update $FABRIC_ID 0 name=vrouter-transport-vlan mtu=9000
space=vrouter-transport-space

maas admin subnets create name=vrouter-transsport-subnet
cidr=192.168.5.0/24 fabric=$FABRIC_ID

SUBNET_ID=$(maas admin subnets read | jq '.[] |
select(."cidr"=="192.168.5.0/24") | .["id"]')

maas admin ipranges create type=reserved start_ip=192.168.5.1 end_
ip=192.168.5.50 subnet=$SUBNET_ID

maas admin ipranges create type=reserved start_ip=192.168.5.175 end_
ip=192.168.5.254 subnet=$SUBNET_ID

maas admin subnet update $SUBNET_ID rdns_mode=0

maas admin subnets update name=vrouter-transsport-subnet
fabric=$FABRIC_ID
```

Prepare Required Linux Image for the CN2 vRouter

MAAS will bootstrap the K8s Control plane VMs and physical worker nodes with the latest Ubuntu images released by Canonical. However, the CN2 router has dependency on specific Linux Kernel versions and CN2 22.3 is qualified with multiple Linux Kernel versions and different Linux distributions CN23.1 (https://github.com/Juniper/contrail-networking/blob/main/releases/23.1/kernels/supported-kernels.json). We will use Ubuntu 20.04.3 image and hard code it with kernel 5.4.0-135-generic, as it is qualified for Linux distribution and Kernel version for CN2 version 23.1.

```
sudo su -
wget http://cloud-images-archive.ubuntu.com/releases/focal/release-20210819/ubuntu-20.04-server-cloudimg-amd64-root.tar.xz
mkdir /tmp/work && cd /tmp/work
tar xfv /root/ubuntu-20.04-server-cloudimg-amd64-root.tar.xz
cat etc/os-release
mount -o bind /proc /tmp/work/proc
mount -o bind /dev /tmp/work/dev
mount -o bind /sys /tmp/work/sys
mv /tmp/work/etc/resolv.conf /tmp/work/etc/resolv.conf.bak
cp /etc/resolv.conf /tmp/work/etc/
chroot /tmp/work /bin/bash
ls boot/
apt update -y
apt install linux-image-5.4.0-135-generic -y
sudo sed -i "s/GRUB_DEFAULT=0/GRUB_DEFAULT='Advanced options for Ubuntu>Ubuntu, with Linux 5.4.0-135-generic'/" /etc/default/grub
echo ubuntu 'ALL=(ALL) NOPASSWD:ALL' > /etc/sudoers.d/ubuntu
ls boot/
exit
umount /tmp/work/proc
umount /tmp/work/dev
umount /tmp/work/sys
mv /tmp/work/etc/resolv.conf.bak /tmp/work/etc/resolv.conf
tar -czf /tmp/focal-20.04.3.tgz -C /tmp/work .
mv /tmp/focal-20.04.3.tgz /home/contrail/
exit
cd ~/
sudo chown contrail:contrail focal-20.04.3.tgz
maas admin boot-resources create name='custom/focal-20.04.3' title='Ubuntu-20.04.3' architecture='amd64/generic' filetype='tgz' content@=focal-20.04.3.tgz
```

Prepare Deployer VM

A VM named "deployer-node" will be created on the KVM host (Control-host) and the deployer-node VM will act as a Jump host to deploy K8s Cluster.

Define a base disk image to host deployer-node:

```
sudo qemu-img create -f qcow2 /var/lib/libvirt/images/deployer-node.qcow2
100G
```

Once the base disk image is defined then we will create a deployer-node VM using the following snippet:

```
sudo virt-install --ram 4096 --vcpus 4 --os-variant ubuntu20.04 --disk
path=/var/lib/libvirt/images/deployer-node.
qcow2,device=disk,bus=virtio,format=qcow2 --graphics vnc,listen=0.0.0.0
--network bridge=br-ctrplane  --boot=network,hd --name deployer-node
--cpu Nehalem,+vmx --dry-run --print-xml > /tmp/deployer-node.xml; virsh
define --file /tmp/deployer-node.xml
```

Once the deployer-node VM is defined, we need to obtain the MAC address to register it in MAAS:

```
virsh domiflist deployer-node

Interface     Type      Source          Model     MAC

-------------------------------------------------------

-             bridge    br-ctrplane     virtio    52:54:00:87:67:27
```

Now let's ensure that MAAS VM can access control-host (KVM host hosting K8s Controller VMs):

```
ssh-copy-id contrail@control-host-ip
```

Commission the deployer-node:

```
maas admin machines create \

hostname=deployer-node \

tag_names=deployer-node \

architecture="amd64/generic" \

mac_addresses=52:54:00:87:67:27 \

power_type=virsh \

power_parameters_power_id=deployer-node \

power_parameters_power_address=qemu+ssh://contrail@192.168.24.10/system \

power_parameters_power_pass=contrail123 \

osystem=centos distro_series=focal-20.04.3

(wait until deployer-node commissioning is completed and once
commissioning is completed then it will be powered down by MAAS)
```

Let's deploy focal-20.04.3 on deployer-node:

```
maas admin tags create name=deployer-node comment='deployer-node'

NODE_SYSID=$(maas admin machines read | jq '.[] |
select(."hostname"=="deployer-node")| .["system_id"]' | tr -d '"')

maas admin tag update-nodes "deployer-node" add=$NODE_SYSID

maas admin machine deploy $NODE_SYSID osystem=custom distro_
series='focal-20.04.3'
```

Wait until the deployer-node deployment is completed and then install the required packages on the deployer-node VM:

```
ssh into deployer-node VM from MAAS VM

ssh  ubuntu@192.168.24.80 (check IP of your deployer VM)

sudo apt-get update -y

sudo apt-get upgrade -y

sudo apt install python3-pip jq haproxy -y

sudo pip3 install --upgrade pip

pip -V

pip 23.0.1 from /usr/local/lib/python3.8/dist-packages/pip (python 3.8)

git clone https://github.com/kubernetes-sigs/kubespray.git

cd kubespray

sudo pip install -r requirements.txt
```

Note: While packages install in deployer-node, you might receive an error "Waiting for cache lock: Could not get lock /var/lib/dpkg/lock-frontend. It is held by process 4545 (unattended-upgr)... 69s". To resolve above issue , follow below given sequence:

```
sudo lsof /var/lib/dpkg/lock

COMMAND   PID USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME

dpkg    81706 root    3uW  REG  252,2        0 3670725 /var/lib/dpkg/lock

sudo lsof /var/lib/apt/lists/lock

sudo lsof /var/cache/apt/archives/lock

COMMAND    PID USER   FD   TYPE DEVICE SIZE/OFF    NODE NAME

unattende 4545 root   71uW  REG  252,2        0 3670037 /var/cache/apt/
archives/lock

sudo kill -9 81706

kill: (81706): No such process

sudo kill -9 3670725

kill: (3670725): No such process

sudo kill -9 4545
```

Generate SSH-key in deployer-node VM:

```
ssh-keygen -q -t rsa -N '' -f ~/.ssh/id_rsa <<<y >/dev/null 2>&1
scp .ssh/id_rsa.pub contrail@maas:deployer-node-id_rsa.pub
```

Upload deployer-node VM ssh public key into MAAS:

```
maas admin sshkeys create key="$(cat /home/contrail/deployer-node-id_rsa.
pub)"
```

Prepare K8s Controller VMs

At this stage let's just create the K8s master nodes VMs only without any OS installation. In the following step, we are defining the VMs and not doing any OS installation:

```
for node in 1 2 3
do
sudo qemu-img create -f qcow2 /var/lib/libvirt/images/controller${node}.
qcow2 200G
sudo virt-install --ram 16384 --vcpus 16 --os-variant ubuntu20.04 --disk
path=/var/lib/libvirt/images/controller${node}.
qcow2,device=disk,bus=virtio,format=qcow2 --graphics vnc,listen=0.0.0.0
--network bridge=br-ctrplane  --network bridge=br-Tenant
--boot=network,hd --name controller${node} --cpu Nehalem,+vmx --dry-run
--print-xml > /tmp/controller${node}.xml; virsh define --file /tmp/
controller${node}.xml
done

contrail@control-host:~$ virsh domiflist controller1
Interface   Type       Source     Model       MAC
-------------------------------------------------------
-           bridge     br-ctrplane virtio      52:54:00:84:3d:56
-           bridge     br-Tenant  virtio      52:54:00:ef:ed:57

contrail@control-host:~$ virsh domiflist controller2
Interface   Type       Source     Model       MAC
-------------------------------------------------------
-           bridge     br-ctrplane virtio      52:54:00:eb:23:6b
-           bridge     br-Tenant  virtio      52:54:00:cc:45:53

contrail@control-host:~$ virsh domiflist controller3
Interface   Type       Source     Model       MAC
-------------------------------------------------------
```

```
-           bridge     br-ctrplane virtio      52:54:00:78:ea:7c
-           bridge     br-Tenant  virtio      52:54:00:0f:d4:b5
```

Bootstrap Controller VMs

At this stage we will bootstrap VMs dedicated for K8s Controller nodes without deploying the K8s Cluster. Controller VMs will be registered into MAAS with power type virsh and a MAC address of br-ctrplane will be used to register the VMs into MAAS, as this NIC will be used to identify the machines and for life cycle management of the VMs. Power_parameters_power-id parameter must match the VMs name in KVM host and power_parameters_address contains the IP address of the KVM server hosting a particular VM and username to access that KVM server. Power-parameters_pass must be the password to access KVM server, osystem parameter will be set to custom and distro-series must match custom image created in the previous step

Controller1 VM

Commission the Controller1 VM into MAAS:

```
maas admin machines create \
hostname=controller1 \
tag_names=controller \
architecture="amd64/generic" \
mac_addresses=52:54:00:84:3d:56 \
power_type=virsh \
power_parameters_power_id=controller1 \
power_parameters_power_address=qemu+ssh://contrail@192.168.24.10/system \
power_parameters_power_pass=contrail123 \
osystem=custom distro_series=focal-20.04.3


(wait until Controller1 commissioning is completed and once commissioning
is completed then it will be powered down by MAAS)
```

The Controller1 interface ens3 is assigned to Provisioning Network (i.e., K8s-Control-network subnet 192.168.24.0/24). This subnet is managed by MAAS DHCP, so no action is required on ens3 interface, but ens4 interface (reserved for CN2 vRouter) is not assigned to any subnet. Let's assign ens4 to Contrail-control-data-network (subnet 192.168.5.0/24):

```
maas admin tags create name=controller1 comment='controller1'
NODE_SYSID=$(maas admin machines read | jq '.[] |
select(."hostname"=="controller1")| .["system_id"]' | tr -d '"')
maas admin tag update-nodes "controller1" add=$NODE_SYSID >/dev/null
```

```
maas admin interfaces read $NODE_SYSID | jq ".[] | {id:.id, name:.name, mac:.mac_address, vid:.vlan.vid, fabric:.vlan.fabric}" --compact-output

maas admin fabrics read | jq ".[] | {name:.name, vlans:.vlans[] | {id:.id, vid:.vid}}" --compact-output

{"id":5,"name":"ens3","mac":"52:54:00:84:3d:56"," vid":0,"fabric":"fabric-0"}

{"id":8,"name":"ens4","mac":"52:54:00:ef:ed:57","vid":0,"fabric":":"fabric-2"}

maas admin fabrics read | jq ".[] | {name:.name, vlans:.vlans[] | {id:.id, vid:.vid}}" --compact-output

{"name":"fabric-0","vlans":{"id":5001,"vid":0}}

{"name":"vrouter-transport-fabric","vlans":{"id":5002,"vid":0}}

{"name":"fabric-2","vlans":{"id":5003,"vid":0}}

{"name":"fabric-3","vlans":{"id":5004,"vid":0}}

{"name":"fabric-4","vlans":{"id":5005,"vid":0}}

IFD_ID=$(maas admin interfaces read $NODE_SYSID | jq '.[] | select(."name"=="ens4") | .["id"]')

VLAN_ID=$(maas admin fabrics read | jq '.[]| select(."name"=="vrouter-transport-fabric") | .vlans[].id')

maas admin interface  update $NODE_SYSID $IFD_ID vlan=$VLAN_ID >/dev/null

SUBNET_ID=$(maas admin subnets read | jq '.[] | select(."cidr"=="192.168.5.0/24") | .["id"]')

maas admin interface link-subnet $NODE_SYSID ${IFD_ID}  subnet=${SUBNET_ID} mode=auto


 maas admin interfaces read $NODE_SYSID | jq ".[] | {id:.id, name:.name, mac:.mac_address, vid:.vlan.vid, fabric:.vlan.fabric}" --compact-output

{"id":5,"name":"ens3","mac":"52:54:00:84:3d:56","vid":0,"fabric":"fabric-0"}

{"id":8,"name":"ens4","mac":"52:54:00:ef:ed:57","vid":0,"fabric":"vrouter-transport-fabric"}
```

Deploy Controller1 VM:

```
NODE_SYSID=$(maas admin machines read | jq '.[] | select(."hostname"=="controller1")| .["system_id"]' | tr -d '"')

maas admin machine deploy $NODE_SYSID osystem=custom distro_series=focal-20.04.3
```

Bootstrap Controller2 VM

Commission the Controller2 VM into MAAS:

```
maas admin machines create \

hostname=controller2 \

tag_names=controller \
```

```
architecture="amd64/generic" \
mac_addresses=52:54:00:eb:23:6b \
power_type=virsh \
power_parameters_power_id=controller2 \
power_parameters_power_address=qemu+ssh://contrail@192.168.24.10/system \
power_parameters_power_pass=contrail123 \
osystem=custom distro_series=focal-20.04.3
(wait until Controller2 commissioning is completed and once commissioning
is completed then it will be powered down by MAAS)
```

The Controller2 interface ens3 is assigned to Provisioning Network (i.e., K8s-Control-network subnet 192.168.24.0/24) as this subnet is managed by MAAS DHCP so no action is required on the ens3 interface but the ens4 interface (reserved for CN2 vRouter) is not assigned to any subnet. Assign ens4 to Contrail-control-data-network (subnet 192.168.5.0/24):

```
maas admin tags create name=controller2 comment='controller2'
NODE_SYSID=$(maas admin machines read | jq '.[] |
select(."hostname"=="controller2")| .["system_id"]' | tr -d '"')
maas admin tag update-nodes "controller2" add=$NODE_SYSID >/dev/null
maas admin interfaces read $NODE_SYSID | jq ".[] | {id:.id, name:.name,
mac:.mac_address, vid:.vlan.vid, fabric:.vlan.fabric}" --compact-output
{"id":6,"name":"ens3","mac":"52:54:00:eb:23:6b","vid":0,"fabric":"
fabric-0"}
{"id":9,"name":"ens4","mac":"52:54:00:cc:45:53","vid":0,"fabric":"
fabric-3"}


maas admin fabrics read | jq ".[] | {name:.name, vlans:.vlans[] | {id:.
id, vid:.vid}}" --compact-output
{"name":"fabric-0","vlans":{"id":5001,"vid":0}}
{"name":"vrouter-transport-fabric","vlans":{"id":5002,"vid":0}}
{"name":"fabric-3","vlans":{"id":5004,"vid":0}}
{"name":"fabric-4","vlans":{"id":5005,"vid":0}}


IFD_ID=$(maas admin interfaces read $NODE_SYSID | jq '.[] |
select(."name"=="ens4") | .["id"]')
VLAN_ID=$(maas admin fabrics read | jq '.[]| select(."name"=="vrouter-
transport-fabric") | .vlans[].id')
maas admin interface  update $NODE_SYSID $IFD_ID vlan=$VLAN_ID >/dev/null
SUBNET_ID=$(maas admin subnets read | jq '.[] |
select(."cidr"=="192.168.5.0/24") | .["id"]')
maas admin interface link-subnet $NODE_SYSID ${IFD_ID}  subnet=${SUBNET_
ID} mode=auto
```

```
maas admin interfaces read $NODE_SYSID | jq ".[] | {id:.id, name:.name, mac:.mac_address, vid:.vlan.vid, fabric:.vlan.fabric}" --compact-output
```

```
{"id":6,"name":"ens3","mac":"52:54:00:eb:23:6b","vid":0,"fabric":"fabric-0"}
{"id":9,"name":"ens4","mac":"52:54:00:cc:45:53","vid":0,"fabric":"vrouter-transport-fabric"}
```

Deploy the Controller2 VM:

```
NODE_SYSID=$(maas admin machines read | jq '.[] | select(."hostname"=="controller2")| .["system_id"]' | tr -d '"')
maas admin machine deploy $NODE_SYSID osystem=custom distro_series=focal-20.04.3
```

Bootstrap Controller3 VM

Commission the Controller3 VM into MAAS:

```
maas admin machines create \
hostname=controller3 \
tag_names=controller \
architecture="amd64/generic" \
mac_addresses=52:54:00:78:ea:7c \
power_type=virsh \
power_parameters_power_id=controller3 \
power_parameters_power_address=qemu+ssh://contrail@192.168.24.10/system \
power_parameters_power_pass=contrail123 \
osystem=custom distro_series=focal-20.04.3
(wait until Controller3  commissioning is completed and once commissioning is completed then it will be powered down by MAAS)
```

The Controller3 interface ens3 is assigned to Provisioning Network (i.e., K8s-Control-network subnet 192.168.24.0/24). This subnet is managed by MAAS DHCP, so no action is required on ens3 interface, but ens4 interface (reserved for CN2 vRouter) is not assigned to any subnet. Let's assign ens4 to Contrail-control-data-network (subnet 192.168.5.0/24):

```
maas admin tags create name=controller3 comment='controller3'
NODE_SYSID=$(maas admin machines read | jq '.[] | select(."hostname"=="controller3")| .["system_id"]' | tr -d '"')
maas admin tag update-nodes "controller3" add=$NODE_SYSID >/dev/null
maas admin interfaces read $NODE_SYSID | jq ".[] | {id:.id, name:.name, mac:.mac_address, vid:.vlan.vid, fabric:.vlan.fabric}" --compact-output
```

```
{"id":7,"name":"ens3","mac":"52:54:00:78:ea:7c","vid":0,"fabric":"fabric-0"}
```

```
{"id":10,"name":"ens4","mac":"52:54:00:0f:d4:b5","vid":0,"fabric":"
fabric-4"}

maas admin fabrics read | jq ".[] | {name:.name, vlans:.vlans[] | {id:.
id, vid:.vid}}" --compact-output

{"name":"fabric-0","vlans":{"id":5001,"vid":0}}

{"name":"vrouter-transport-fabric","vlans":{"id":5002,"vid":0}}

{"name":"fabric-4","vlans":{"id":5005,"vid":0}}


IFD_ID=$(maas admin interfaces read $NODE_SYSID | jq '.[] |
select(."name"=="ens4") | .["id"]')

VLAN_ID=$(maas admin fabrics read | jq '.[]| select(."name"=="vrouter-
transport-fabric") | .vlans[].id')

maas admin interface  update $NODE_SYSID $IFD_ID vlan=$VLAN_ID >/dev/null


SUBNET_ID=$(maas admin subnets read | jq '.[] |
select(."cidr"=="192.168.5.0/24") | .["id"]')

maas admin interface link-subnet $NODE_SYSID ${IFD_ID}  subnet=${SUBNET_
ID} mode=auto


 maas admin interfaces read $NODE_SYSID | jq ".[] | {id:.id, name:.name,
mac:.mac_address, vid:.vlan.vid, fabric:.vlan.fabric}" --compact-output


{"id":7,"name":"ens3","mac":"52:54:00:78:ea:7c","vid":0,"fabric":"
fabric-0"}

{"id":10,"name":"ens4","mac":"52:54:00:0f:d4:b5","vid":0,"fabric":"
vrouter-transport-fabric"}
```

Deploy Controller3 node.

```
NODE_SYSID=$(maas admin machines read | jq '.[] |
select(."hostname"=="controller3")| .["system_id"]' | tr -d '"')

maas admin machine deploy $NODE_SYSID osystem=custom distro_
series=focal-20.04.3
```

Bootstrap Worker Nodes

Now at this stage we will bootstrap bare metal servers which will be further used as K8s worker nodes. We have used Dell R720 servers as the K8s worker nodes. The IPMI IP will be used to commission physical servers into MAAS. The MAC Address of interface, where PXE boot is enabled is also required.

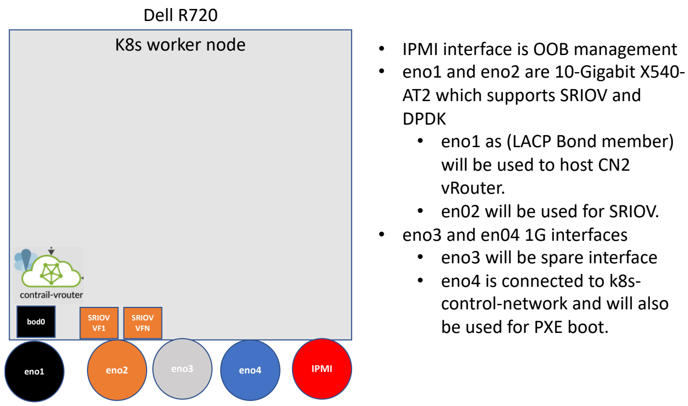


Figure 5 *Worker Nodes Network Connectivity*

Bootstrap Worker1 Node

Commission the Worker1 node into MAAS:

```
ipmi_user=root
ipmi_password=calvin
ipmi_ip=192.168.100.121
ipmitool -I lanplus -H $ipmi_ip -U $ipmi_user -P $ipmi_password  chassis bootdev pxe

maas admin machines create \
    hostname=worker1 \
    fqdn=worker1.maas \
    mac_addresses=BC:30:5B:F2:87:55 \
    architecture=amd64 \
    power_type=ipmi \
    power_parameters_power_driver=LAN_2_0 \
    power_parameters_power_user=root \
    power_parameters_power_pass=calvin \
    power_parameters_power_address=192.168.100.121
(Wait until Worker1 commissioning is completed and once commissioning is completed then it will be powered down by MAAS)
```

The Worker1 interface eno4 is assigned to Provisioning Network (i.e., K8s-Control-network subnet 192.168.24.0/24) as this subnet is managed by MAAS DHCP so no action is required on the eno4 interface, but the eno1 interface (reserved for CN2 vRouter) is not assigned to any subnet. We will create a 802.3ad bond (i.e., bond0) and assign eno1 to bond0 and the Contrail-control-data-network (subnet

192.168.5.0/24) will be assigned to bond0. No action is required on eno2 as SRIOV VFs will be created on this interface in a later part of this book and eno3 will be a spare interface:

```
maas admin tags create name=worker1 comment='worker1'

NODE_SYSID=$(maas admin machines read | jq '.[] |
select(."hostname"=="worker1")| .["system_id"]' | tr -d '"')

maas admin tag update-nodes "worker1" add=$NODE_SYSID  >/dev/null

contrail@maas:~$ maas admin interfaces read $NODE_SYSID | jq ".[] | {id:.
id, name:.name, mac:.mac_address, vid:.vlan.vid, fabric:.vlan.fabric}"
--compact-output

{"id":12,"name":"eno4","mac":"bc:30:5b:f2:87:55","vid":0,"fabric":"f
abric-0"}

{"id":13,"name":"eno1","mac":"bc:30:5b:f2:87:50","vid":0,"fabric":"
fabric-5"}

{"id":14,"name":"eno2","mac":"bc:30:5b:f2:87:52","vid":0,"fabric":"
fabric-6"}

{"id":15,"name":"eno3","mac":"bc:30:5b:f2:87:54","vid":0,"fabric":"
fabric-7"}


maas admin fabrics read | jq ".[] | {name:.name, vlans:.vlans[] | {id:.
id, vid:.vid}}" --compact-output

{"name":"fabric-0","vlans":{"id":5001,"vid":0}}

{"name":"vrouter-transport-fabric","vlans":{"id":5002,"vid":0}}

{"name":"fabric-5","vlans":{"id":5006,"vid":0}}

{"name":"fabric-6","vlans":{"id":5007,"vid":0}}

{"name":"fabric-7","vlans":{"id":5008,"vid":0}}


IFD_ID=$(maas admin interfaces read $NODE_SYSID | jq '.[] |
select(."name"=="eno1") | .["id"]')

VLAN_ID=$(maas admin fabrics read | jq '.[]| select(."name"=="vrouter-
transport-fabric") | .vlans[].id')

maas admin interface  update $NODE_SYSID $IFD_ID vlan=$VLAN_ID >/dev/null


maas admin interfaces create-bond $NODE_SYSID name=bond0 parents=$IFD_ID
bond_mode=802.3ad mtu=9000

maas admin interfaces read $NODE_SYSID | jq ".[] | {id:.id, name:.name,
mac:.mac_address, vid:.vlan.vid, fabric:.vlan.fabric}" --compact-output

{"id":12,"name":"eno4","mac":"bc:30:5b:f2:87:55","vid":0,"fabric":"
fabric-0"}

{"id":13,"name":"eno1","mac":"bc:30:5b:f2:87:50","vid":0,"fabric":"
vrouter-transport-fabric"}

{"id":14,"name":"eno2","mac":"bc:30:5b:f2:87:52","vid":0,"fabric":"
fabric-6"}
```

```
{"id":15,"name":"eno3","mac":"bc:30:5b:f2:87:54","vid":0,"fabric":"
fabric-7"}
{"id":18,"name":"bond0","mac":"bc:30:5b:f2:87:50","vid":0,"fabric":"
vrouter-transport-fabric"}
```

```
SUBNET_ID=$(maas admin subnets read | jq '.[] |
select(."cidr"=="192.168.5.0/24") | .["id"]')
IFD_ID=$(maas admin interfaces read $NODE_SYSID | jq '.[] |
select(."name"=="bond0") | .["id"]')
maas admin interface link-subnet $NODE_SYSID ${IFD_ID}  subnet=${SUBNET_
ID} mode=auto
```

Deploy Worker1 node.

```
ipmi_user=root
ipmi_password=calvin
ipmi_ip=192.168.100.121
ipmitool -I lanplus -H $ipmi_ip -U $ipmi_user -P $ipmi_password  chassis
bootdev pxe
NODE_SYSID=$(maas admin machines read | jq '.[] |
select(."hostname"=="worker1")| .["system_id"]' | tr -d '"')
maas admin machine deploy $NODE_SYSID osystem=custom distro_
series=focal-20.04.3
```

Bootstrap Worker2 Node

Commission worker2 node into MAAS:

```
ipmi_user=root
ipmi_password=calvin
ipmi_ip=192.168.100.122
ipmitool -I lanplus -H $ipmi_ip -U $ipmi_user -P $ipmi_password  chassis
bootdev pxe
maas admin machines create \
    hostname=worker2 \
    fqdn=worker2.maas \
    mac_addresses=BC:30:5B:F2:3F:75 \
    architecture=amd64 \
    power_type=ipmi \
    power_parameters_power_driver=LAN_2_0 \
    power_parameters_power_user=root \
    power_parameters_power_pass=calvin \
    power_parameters_power_address=192.168.100.122 \
    osystem=custom distro_series=focal-20.04.3
```

```
(Wait until Worker2 commissioning is completed and once commissioning is completed then it will be powered down by MAAS)
```

The Worker2 interface eno4 is assigned to Provisioning Network (i.e., K8s-Control-network subnet 192.168.24.0/24) as this subnet is managed by MAAS DHCP so no action is required on the eno4 interface but the eno1 interface (reserved for CN2 vRouter) is not assigned to any subnet. We'llll create 802.3ad bond (i.e., bond0) and assign eno1 to bond0 and the Contrail-control-data-network (subnet 192.168.5.0/24) will be assigned to bond0. No action is required on eno2 as SRIOV VFs will be created on this interface in a later part of this book and eno3 will be spare interface:

```
maas admin tags create name=worker2 comment='worker2'

NODE_SYSID=$(maas admin machines read | jq '.[] |
select(."hostname"=="worker2")| .["system_id"]' | tr -d '"')

maas admin tag update-nodes "worker2" add=$NODE_SYSID  >/dev/null

maas admin interfaces read $NODE_SYSID | jq ".[] | {id:.id, name:.name,
mac:.mac_address, vid:.vlan.vid, fabric:.vlan.fabric}" --compact-output

{"id":16,"name":"eno4","mac":"bc:30:5b:f2:3f:75","vid":0,"fabric":"
fabric-0"}

{"id":19,"name":"eno1","mac":"bc:30:5b:f2:3f:70","vid":0,"fabric":"
fabric-8"}

{"id":20,"name":"eno2","mac":"bc:30:5b:f2:3f:72","vid":0,"fabric":"
fabric-9"}

{"id":21,"name":"eno3","mac":"bc:30:5b:f2:3f:74","vid":0,"fabric":"
fabric-10"}


maas admin fabrics read | jq ".[] | {name:.name, vlans:.vlans[] | {id:.
id, vid:.vid}}" --compact-output

{"name":"fabric-0","vlans":{"id":5001,"vid":0}}

{"name":"vrouter-transport-fabric","vlans":{"id":5002,"vid":0}}

{"name":"fabric-6","vlans":{"id":5007,"vid":0}}

{"name":"fabric-7","vlans":{"id":5008,"vid":0}}

{"name":"fabric-8","vlans":{"id":5009,"vid":0}}

{"name":"fabric-9","vlans":{"id":5010,"vid":0}}

{"name":"fabric-10","vlans":{"id":5011,"vid":0}}

{"name":"fabric-11","vlans":{"id":5012,"vid":0}}

{"name":"fabric-12","vlans":{"id":5013,"vid":0}}

{"name":"fabric-13","vlans":{"id":5014,"vid":0}}


IFD_ID=$(maas admin interfaces read $NODE_SYSID | jq '.[] |
select(."name"=="eno1") | .["id"]')

VLAN_ID=$(maas admin fabrics read | jq '.[]| select(."name"=="vrouter-
transport-fabric") | .vlans[].id')

maas admin interface  update $NODE_SYSID $IFD_ID vlan=$VLAN_ID >/dev/null
```

```
maas admin interfaces create-bond $NODE_SYSID name=bond0 parents=$IFD_ID
bond_mode=802.3ad mtu=9000


maas admin interfaces read $NODE_SYSID | jq ".[] | {id:.id, name:.name,
mac:.mac_address, vid:.vlan.vid, fabric:.vlan.fabric}" --compact-output
{"id":16,"name":"eno4","mac":"bc:30:5b:f2:3f:75","vid":0,"fabric":"
fabric-0"}
{"id":19,"name":"eno1","mac":"bc:30:5b:f2:3f:70","vid":0,"fabric":"
vrouter-transport-fabric"}
{"id":20,"name":"eno2","mac":"bc:30:5b:f2:3f:72","vid":0,"fabric":"
fabric-9"}
{"id":21,"name":"eno3","mac":"bc:30:5b:f2:3f:74","vid":0,"fabric":"
fabric-10"}
{"id":25,"name":"bond0","mac":"bc:30:5b:f2:3f:70","vid":0,"fabric":"
vrouter-transport-fabric"}


SUBNET_ID=$(maas admin subnets read | jq '.[] |
select(."cidr"=="192.168.5.0/24") | .["id"]')
IFD_ID=$(maas admin interfaces read $NODE_SYSID | jq '.[] |
select(."name"=="bond0") | .["id"]')
maas admin interface link-subnet $NODE_SYSID ${IFD_ID}  subnet=${SUBNET_
ID} mode=auto
```

Deploy Worker2 node:

```
ipmi_user=root
ipmi_password=calvin
ipmi_ip=192.168.100.122
ipmitool -I lanplus -H $ipmi_ip -U $ipmi_user -P $ipmi_password  chassis
bootdev pxe
NODE_SYSID=$(maas admin machines read | jq '.[] |
select(."hostname"=="worker2")| .["system_id"]' | tr -d '"')
maas admin machine deploy $NODE_SYSID osystem=custom distro_
series=focal-20.04.3
```

Bootstrap Worker3 Node

Commission Worker3 node into MAAS:

```
ipmi_user=root
ipmi_password=calvin
ipmi_ip=192.168.100.123
ipmitool -I lanplus -H $ipmi_ip -U $ipmi_user -P $ipmi_password  chassis
bootdev pxe


maas admin machines create \
```

```
    hostname=worker3 \
    fqdn=worker3.maas \
    mac_addresses=BC:30:5B:F1:C2:05 \
    architecture=amd64 \
    power_type=ipmi \
    power_parameters_power_driver=LAN_2_0 \
    power_parameters_power_user=root \
    power_parameters_power_pass=calvin \
    power_parameters_power_address=192.168.100.123 \
    osystem=custom distro_series=focal-20.04.3
(wait until Worker3 commissioning is completed and once commissioning is
completed then it will be powered down by MAAS)
```

With the Worker3 interface, eno4 is assigned to Provisioning Network (i.e., K8s-Control-network subnet 192.168.24.0/24) as this subnet is managed by MAAS DHCP so no action is required on the eno4 interface but the eno1 interface (reserved for CN2 vRouter) is not assigned to any subnet. Let's create 802.3ad bond (i.e., bond0) and assign eno1 to bond0 and the Contrail-control-data-network (subnet 192.168.5.0/24) will be assigned to bond0. No action is required on eno2 as SRIOV VFs will be created on this interface in a later part of this book and eno3 will be spare interface:

```
maas admin tags create name=worker3 comment='worker3'

NODE_SYSID=$(maas admin machines read | jq '.[] |
select(."hostname"=="worker3")| .["system_id"]' | tr -d '"')

maas admin tag update-nodes "worker3" add=$NODE_SYSID  >/dev/null

maas admin interfaces read $NODE_SYSID | jq ".[] | {id:.id, name:.name,
mac:.mac_address, vid:.vlan.vid, fabric:.vlan.fabric}" --compact-output

{"id":17,"name":"eno4","mac":"bc:30:5b:f1:c2:05","vid":0,"fabric":"
fabric-0"}

{"id":22,"name":"eno1","mac":"bc:30:5b:f1:c2:00","vid":0,"fabric":"
fabric-11"}

{"id":23,"name":"eno2","mac":"bc:30:5b:f1:c2:02","vid":0,"fabric":"
fabric-12"}

{"id":24,"name":"eno3","mac":"bc:30:5b:f1:c2:04","vid":0,"fabric":"
fabric-13"}


maas admin fabrics read | jq ".[] | {name:.name, vlans:.vlans[] | {id:.
id, vid:.vid}}" --compact-output

{"name":"fabric-0","vlans":{"id":5001,"vid":0}}

{"name":"vrouter-transport-fabric","vlans":{"id":5002,"vid":0}}

{"name":"fabric-6","vlans":{"id":5007,"vid":0}}

{"name":"fabric-7","vlans":{"id":5008,"vid":0}}

{"name":"fabric-9","vlans":{"id":5010,"vid":0}}
```

```
{"name":"fabric-10","vlans":{"id":5011,"vid":0}}
{"name":"fabric-11","vlans":{"id":5012,"vid":0}}
{"name":"fabric-12","vlans":{"id":5013,"vid":0}}
{"name":"fabric-13","vlans":{"id":5014,"vid":0}}

IFD_ID=$(maas admin interfaces read $NODE_SYSID | jq '.[] |
select(."name"=="eno1") | .["id"]')
VLAN_ID=$(maas admin fabrics read | jq '.[]| select(."name"=="vrouter-
transport-fabric") | .vlans[].id')
maas admin interface  update $NODE_SYSID $IFD_ID vlan=$VLAN_ID >/dev/null
maas admin interfaces create-bond $NODE_SYSID name=bond0 parents=$IFD_ID
bond_mode=802.3ad mtu=9000

maas admin interfaces read $NODE_SYSID | jq ".[] | {id:.id, name:.name,
mac:.mac_address, vid:.vlan.vid, fabric:.vlan.fabric}" --compact-output
{"id":17,"name":"eno4","mac":"bc:30:5b:f1:c2:05","vid":0,"fabric":"
fabric-0"}
{"id":22,"name":"eno1","mac":"bc:30:5b:f1:c2:00","vid":0,"fabric":"
vrouter-transport-fabric"}
{"id":23,"name":"eno2","mac":"bc:30:5b:f1:c2:02","vid":0,"fabric":"
fabric-12"}
{"id":24,"name":"eno3","mac":"bc:30:5b:f1:c2:04","vid":0,"fabric":"
fabric-13"}
{"id":26,"name":"bond0","mac":"bc:30:5b:f1:c2:00","vid":0,"fabric":"
vrouter-transport-fabric"}

SUBNET_ID=$(maas admin subnets read | jq '.[] |
select(."cidr"=="192.168.5.0/24") | .["id"]')
IFD_ID=$(maas admin interfaces read $NODE_SYSID | jq '.[] |
select(."name"=="bond0") | .["id"]')
maas admin interface link-subnet $NODE_SYSID ${IFD_ID}  subnet=${SUBNET_
ID} mode=auto
Deploy Worker3 node.
ipmi_user=root
ipmi_password=calvin
ipmi_ip=192.168.100.123
ipmitool -I lanplus -H $ipmi_ip -U $ipmi_user -P $ipmi_password  chassis
bootdev pxe
NODE_SYSID=$(maas admin machines read | jq '.[] |
select(."hostname"=="worker3")| .["system_id"]' | tr -d '"')
maas admin machine deploy $NODE_SYSID osystem=custom distro_
series=focal-20.04.3
```

Chapter 4

Preparing DPDK Worker Nodes

This chapter covers K8s worker node for Data Plane Development Kit (DPDK) deployment.

DPDK Introduction

The Data Plane Development Kit (DPDK https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-the-data-plane-development-kit-dpdk-packet-framework.html) consists of libraries to accelerate packet processing workloads running on a wide variety of CPU architectures. CN2 also supports DPDK based vRouter.

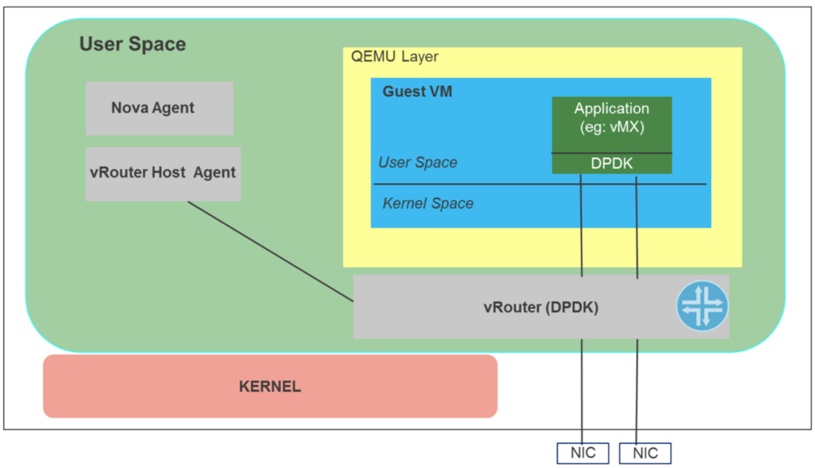


Figure 6 DPDK vRouter High Level Architecture
Courtesy to Juniper Networks for above diagram

Care should be taken with NUMA. If you are using two or more ports from different NICs, it is best to ensure that these NICs are on the same CPU socket (https://doc.dpdk.org/guides-16.04/linux_gsg/nic_perf_intel_platform.html). Unexpected issues may result when an i40e device is in multidriver mode and the kernel driver and DPDK driver are sharing the device. This is because access to the global NIC resources is not synchronized between multiple drivers. Any change to the global NIC configuration (writing to a global register, setting global configuration by AQ, or changing switch modes) will affect all ports and drivers on the device. Loading DPDK with the "multidriver" module parameter may mitigate some of the issues (https://www.kernel.org/doc/html/v5.1/networking/device_drivers/intel/i40e.html)

Deployment Workflow

- Load DPDK PMD kernel driver supported by your NIC.
- Identify CPU NUMA topology of DPDK worker node.
- Add DPDK Support in the Host OS Kernel by editing `/etc/defaults/grub'.
- Add "iommu=pt intel_iommu=on".
- Add the required number of huge pages.
- Isolate the DPDK Forwarding, Services, and Control threads from the host OS scheduler.
- Update grub and reboot the worker node.

Load DPDK PMD Kernel Driver

The CN2 DPDK vRouter supports vfio-pci and uio_pci_generic. Hence, the NIC available in this setup only supports uio_pci_generic, so let's load uio_pci_generic kernel drivers:

```
apt install linux-modules-extra-$(uname -r)
/sbin/modprobe uio_pci_generic


lsmod | grep uio_pci_generic
uio_pci_generic        16384  0
uio                    20480  1 uio_pci_generic
echo uio_pci_generic | sudo tee -a /etc/modules
ls -larth /lib/modules/$(uname -r)/kernel/drivers/uio/
total 176K
-rw-r--r--   1 root root  11K Nov 23 19:51 uio_sercos3.ko
-rw-r--r--   1 root root  14K Nov 23 19:51 uio_pruss.ko
-rw-r--r--   1 root root  13K Nov 23 19:51 uio_pdrv_genirq.ko
-rw-r--r--   1 root root 9.2K Nov 23 19:51 uio_pci_generic.ko
-rw-r--r--   1 root root  11K Nov 23 19:51 uio_netx.ko
-rw-r--r--   1 root root  11K Nov 23 19:51 uio_mf624.ko
-rw-r--r--   1 root root  15K Nov 23 19:51 uio_hv_generic.ko
-rw-r--r--   1 root root  14K Nov 23 19:51 uio_dmem_genirq.ko
-rw-r--r--   1 root root 9.2K Nov 23 19:51 uio_cif.ko
-rw-r--r--   1 root root  11K Nov 23 19:51 uio_aec.ko
-rw-r--r--   1 root root  31K Nov 23 19:51 uio.ko
drwxr-xr-x 102 root root 4.0K Mar 19 09:42 ..
drwxr-xr-x   2 root root 4.0K Mar 19 09:42 .
```

Identify CPU NUMA Topology

Worker2:

```
lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          24
On-line CPU(s) list:             0-23
Thread(s) per core:              2
Core(s) per socket:              6
Socket(s):                       2
NUMA node(s):                    2
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           62
Model name:                      Intel(R) Xeon(R) CPU E5-2620 v2 @
2.10GHz
Stepping:                        4
CPU MHz:                         1199.898
CPU max MHz:                     2600.0000
CPU min MHz:                     1200.0000
BogoMIPS:                        4200.41
Virtualization:                  VT-x
L1d cache:                       384 KiB
L1i cache:                       384 KiB
L2 cache:                        3 MiB
L3 cache:                        30 MiB
NUMA node0 CPU(s):               0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s):               1,3,5,7,9,11,13,15,17,19,21,23

sudo cat $(find /sys/devices/system/cpu -regex ".*cpu[0-9]+/topology/
thread_siblings_list") | sort -n | uniq
0,12
1,13
2,14
3,15
4,16
```

```
5,17
6,18
7,19
8,20
9,21
10,22
11,23
```

Worker3:

```
lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          32
On-line CPU(s) list:             0-31
Thread(s) per core:              2
Core(s) per socket:              8
Socket(s):                       2
NUMA node(s):                    2
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           62
Model name:                      Intel(R) Xeon(R) CPU E5-2650 v2 @
2.60GHz
Stepping:                        4
CPU MHz:                         1200.612
CPU max MHz:                     3400.0000
CPU min MHz:                     1200.0000
BogoMIPS:                        5200.26
Virtualization:                  VT-x
L1d cache:                       512 KiB
L1i cache:                       512 KiB
L2 cache:                        4 MiB
L3 cache:                        40 MiB
NUMA node0 CPU(s):               0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,
30
NUMA node1 CPU(s):               1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,
31
```

```
sudo cat $(find /sys/devices/system/cpu -regex ".*cpu[0-9]+/topology/
thread_siblings_list") | sort -n | uniq
0,16
1,17
2,18
3,19
4,20
5,21
6,22
7,23
8,24
9,25
10,26
11,27
12,28
13,29
14,30
15,31
```

Add DPDK Support in the Host OS Kernel

```
sed -i  's/GRUB_CMDLINE_LINUX_DEFAULT=""/GRUB_CMDLINE_LINUX_
DEFAULT="default_hugepagesz=1G hugepagesz=1G hugepages=16 iommu=pt
intel_iommu=on isolcpus=2-5,14-17"/g' /etc/default/grub
update-grub
reboot
```

Chapter 5

K8s- CN2 Cluster and Other Utilities bring up

This chapter prepares Kube-Spray for K8s deployment, deploying the K8s cluster, creating HAProxy on deployer-node, preparing manifest for CN2 deployment ,and finally, deploying CN2. Let's get at it!

Prepare Kube-Spray

The deployer-node VM created in one of the previous steps will be used as a Jump host from which to deploy a K8s cluster via Kubspray. Kube-Spray (https://github.com/kubernetes-sigs/kubespray) is an Ansible-based tool to deploy a K8s cluster. The reference guide can be found at Create a Kubernetes Cluster (https://www.juniper.net/documentation/us/en/software/cn-cloud-native22/cn-cloud-native-K8s-install-and-lcm/topics/task/cn-cloud-native-K8s-create-kubernetes-cluster.html).

```
ssh ubuntu@deployer-node
cd ~/kubespray
cp -rfp inventory/sample/ inventory/testcluster
cd inventory/testcluster
cp inventory.ini hosts.ini
```

The inventory file from the setup is appended here:

```
cd /home/ubuntu/kubespray/inventory/testcluster
cat > hosts.ini <<END_OF_SCRIPT
[all]
controller1 ansible_host=192.168.24.87 ansible_user=ubuntu
controller2 ansible_host=192.168.24.88 ansible_user=ubuntu
controller3 ansible_host=192.168.24.89 ansible_user=ubuntu
worker1     ansible_host=192.168.24.90 ansible_user=ubuntu
worker2     ansible_host=192.168.24.91 ansible_user=ubuntu
worker3     ansible_host=192.168.24.92 ansible_user=ubuntu

[kube_control_plane]
controller1
controller2
controller3

[etcd]
```

```
controller1
controller2
controller3

[kube_node]
worker1
worker2
worker3

[K8s_cluster:children]
kube_control_plane
kube_node
<<END_OF_SCRIPT
```

Edit the K8s-cluster.yml file:

```
cd $HOME/kubespray/inventory/testcluster/group_vars/K8s_cluster
vim K8s-cluster.yml
kube_version: v1.25.0
container_manager: crio
kube_network_plugin_multus: true
multus_cni_version: "0.3.1"
```

Deploy K8s Cluster using Kubespray

```
export ANSIBLE_HOST_KEY_CHECKING=False
cd ~/kubespray
ansible -i inventory/testcluster/hosts.ini -m ping all
ansible-playbook -i inventory/testcluster/hosts.ini cluster.yml -u ubuntu
--become
```

On successful completion you should see following logs:

```
RUNNING HANDLER [kubernetes/preinstall : Preinstall | wait for the
apiserver to be running] ************************************************
************************************
ok: [controller2]
ok: [controller1]
ok: [controller3]
Sunday 02 April 2023  20:02:58 +0000 (0:00:08.278)       0:32:23.315
**********

RUNNING HANDLER [kubernetes/preinstall : Preinstall | Restart systemd-
resolved] **************************************************************
***********************************
```

```
changed: [worker2]
changed: [worker1]
changed: [controller3]
changed: [controller2]
changed: [controller1]
changed: [worker3]
Sunday 02 April 2023  20:02:59 +0000 (0:00:01.632)       0:32:24.947
**********
Sunday 02 April 2023  20:02:59 +0000 (0:00:00.277)       0:32:25.225
**********
Sunday 02 April 2023  20:03:00 +0000 (0:00:00.251)       0:32:25.477
**********
Sunday 02 April 2023  20:03:00 +0000 (0:00:00.240)       0:32:25.718
**********
Sunday 02 April 2023  20:03:00 +0000 (0:00:00.252)       0:32:25.971
**********
Sunday 02 April 2023  20:03:00 +0000 (0:00:00.241)       0:32:26.212
**********
Sunday 02 April 2023  20:03:01 +0000 (0:00:00.262)       0:32:26.475
**********
Sunday 02 April 2023  20:03:01 +0000 (0:00:00.237)       0:32:26.712
**********
Sunday 02 April 2023  20:03:01 +0000 (0:00:00.277)       0:32:26.990
**********

PLAY RECAP ***************************************************************
**************************************************************************
********************************
controller1                : ok=602  changed=131  unreachable=0
failed=0    skipped=1135 rescued=0    ignored=5
controller2                : ok=544  changed=119  unreachable=0
failed=0    skipped=993  rescued=0    ignored=3
controller3                : ok=546  changed=120  unreachable=0
failed=0    skipped=991  rescued=0    ignored=3
localhost                  : ok=3    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
worker1                    : ok=421  changed=84   unreachable=0
failed=0    skipped=664  rescued=0    ignored=1
worker2                    : ok=421  changed=84   unreachable=0
failed=0    skipped=663  rescued=0    ignored=1
worker3                    : ok=421  changed=84   unreachable=0
failed=0    skipped=663  rescued=0    ignored=1
```

Deploy HAProxy on deployer-node

We need to configure Proxy on the deployer-node so that the subsequent API calls from deployer-node to K8s controllers' node can be load balanced across controller nodes:

```
sudo su -
cat > /etc/haproxy/haproxy.cfg <<END_OF_SCRIPT
#---------------------------------------------------------------------
# Example configuration for a possible web application.  See the
# full configuration options online.
#
#   http://haproxy.1wt.eu/download/1.4/doc/configuration.txt
#
#---------------------------------------------------------------------

#---------------------------------------------------------------------
# Global settings
#---------------------------------------------------------------------
global
    # to have these messages end up in /var/log/haproxy.log you will
    # need to:
    #
    # 1) configure syslog to accept network log events.  This is done
    #    by adding the '-r' option to the SYSLOGD_OPTIONS in
    #    /etc/sysconfig/syslog
    #
    # 2) configure local2 events to go to the /var/log/haproxy.log
    #   file. A line like the following can be added to
    #   /etc/sysconfig/syslog
    #
    #    local2.*                       /var/log/haproxy.log
    #
    log         127.0.0.1 local2

    chroot      /var/lib/haproxy
    pidfile     /var/run/haproxy.pid
    maxconn     4000
    user        haproxy
```

```
    group       haproxy
    daemon

    # turn on stats unix socket
    stats socket /var/lib/haproxy/stats

#---------------------------------------------------------------------
# common defaults that all the 'listen' and 'backend' sections will
# use if not designated in their block
#---------------------------------------------------------------------
defaults
    mode                    tcp
    log                     global
    option                  httplog
    option                  dontlognull
    option http-server-close
    option forwardfor       except 127.0.0.0/8
    option                  redispatch
    retries                 3
    timeout http-request    10s
    timeout queue           1m
    timeout connect         10s
    timeout client          4h
    timeout server          4h
    timeout http-keep-alive 10s
    timeout check           10s
    maxconn                 3000

#---------------------------------------------------------------------

listen stats
    bind :9000
    mode http
    stats enable
    stats uri /
    monitor-uri /healthz
```

```
frontend 127.0.0.1
    bind *:6443
    default_backend 127.0.0.1
    option tcplog


backend 127.0.0.1
    balance roundrobin
    server controller1.maas 192.168.24.99:6443 check
    server controller2.maas 192.168.24.101:6443 check
    server controller3.maas 192.168.24.102:6443 check
END_OF_SCRIPT
systemctl restart haproxy
systemctl status  haproxy
exit
```

HAProxy monitoring is enabled on port 9000, so we can always check HAProxy stats by browsing http://deployer-node.maas:9000.

Get Kubectl Binary and Kube-config on Deployer-node

```
cd ~/
mkdir .kube/
ssh controller1 "sudo cp /root/.kube/config . && sudo chown ubuntu:ubuntu config"
scp controller1:config .kube/config
ssh controller1 "sudo cp /usr/local/bin/kubectl . && sudo chown ubuntu:ubuntu kubectl"
scp controller1:kubectl . && sudo mv kubectl /usr/local/bin/
```

Monitor the deployment:

```
kubectl get nodes

NAME          STATUS     ROLES           AGE    VERSION
controller1   NotReady   control-plane   9m17s  v1.25.0
controller2   NotReady   control-plane   8m49s  v1.25.0
controller3   NotReady   control-plane   8m38s  v1.25.0
worker1       NotReady   <none>          7m14s  v1.25.0
worker2       NotReady   <none>          7m14s  v1.25.0
worker3       NotReady   <none>          7m14s  v1.25.0

kubectl get pods -A
```

```
NAMESPACE     NAME                                  READY   STATUS
RESTARTS   AGE
kube-system   coredns-588bb58b94-rmxhn              0/1     Pending    0
7m24s
kube-system   dns-autoscaler-5b9959d7fc-khcbh       0/1     Pending    0
7m15s
kube-system   kube-apiserver-controller1            1/1     Running    1
11m
kube-system   kube-apiserver-controller2            1/1     Running    1
10m
kube-system   kube-apiserver-controller3            1/1     Running    1
10m
kube-system   kube-controller-manager-controller1   1/1     Running    1
10m
kube-system   kube-controller-manager-controller2   1/1     Running    1
10m
kube-system   kube-controller-manager-controller3   1/1     Running    1
10m
kube-system   kube-multus-ds-amd64-9sv5b            1/1     Running    0
7m57s
kube-system   kube-multus-ds-amd64-gth6c            1/1     Running    0
7m57s
kube-system   kube-multus-ds-amd64-rkr78            1/1     Running    0
7m57s
kube-system   kube-multus-ds-amd64-t2fnw            1/1     Running    0
7m57s
kube-system   kube-multus-ds-amd64-vpkpc            1/1     Running    0
7m57s
kube-system   kube-multus-ds-amd64-wnmmn            1/1     Running    0
7m57s
kube-system   kube-proxy-4fznv                      1/1     Running    0
8m58s
kube-system   kube-proxy-566bl                      1/1     Running    0
8m58s
kube-system   kube-proxy-5dgnr                      1/1     Running    0
8m58s
kube-system   kube-proxy-l9qr8                      1/1     Running    0
8m58s
kube-system   kube-proxy-lrwlk                      1/1     Running    0
8m58s
kube-system   kube-proxy-qdtrl                      1/1     Running    0
8m58s
kube-system   kube-scheduler-controller1            1/1     Running    1
11m
kube-system   kube-scheduler-controller2            1/1     Running    1
10m
```

```
kube-system   kube-scheduler-controller3           1/1     Running   1
10m
kube-system   nginx-proxy-worker1                  1/1     Running   0
8m16s
kube-system   nginx-proxy-worker2                  1/1     Running   0
8m16s
kube-system   nginx-proxy-worker3                  1/1     Running   0
8m16s
```

All pods should have a STATUS of Running except for the DNS pods. The DNS pods do not come up because there is no networking. This is expected.

Prepare CN2 Deployment

Download the CN2 manifest (https://support.juniper.net/support/downloads/?p=contrail#sw) and untar the archive then update the deployer file as per your environment:

```
mkdir ~/cn2-23.1
curl 'https://cdn.juniper.net/software/contrail/23.1.0/contrail-manifests-K8s-23.1.0.282.tgz?SM_USER=knawaz&__gda__' -o ~/cn2-23.1/contrail-manifests-K8s-23.1.0.282.tgz
tar xf contrail-manifests-K8s-23.1.0.282.tgz
```

Add your base64 credentials to the deployer file to download container images from the Juniper Container Images repository:

```
sudo apt install docker.io -y
docker login enterprise-hub.juniper.net
Username:
Password:
WARNING! Your password will be stored unencrypted in /home/ubuntu/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store


Login Succeeded
ENCODED_CREDS=$(base64 -w 0 .docker/config.json)
cd ~/cn2-23.1/K8s/single-cluster
sed -i  s/'<base64-encoded-credential>'/$ENCODED_CREDS/ *.yaml
```

To add separate Data and Control Networks for CN2, the K8s Control Plane will use a separate network, please refer to the diagram at the following link.

```
vim ~/cn2-23.1/K8s/single-cluster/single_cluster_deployer_example.yaml
---
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: contrail-network-config
  namespace: contrail
data:
  networkConfig: |
    controlDataNetworks:
    - subnet: 192.168.5.0/24
      gateway: 192.168.5.1
```

Add the appropriate Hugepage for DPDK node. By default 3Gi Hugepages is referred in deployer file, but for the production grade environment 6Gi is recommended.

```
vim ~/cn2-23.1/K8s/single-cluster/single_cluster_deployer_example.yaml
find following
  resources:
            limits:
              hugepages-1Gi: 3Gi
            requests:
              memory: 3Gi
  and replace above  with the following

  resources:
            limits:
              hugepages-1Gi: 6Gi
            requests:
              memory: 6Gi
```

Change the DPDK PMD value in deployer (by default it is vfio-pci), hence NICs in the setup support only uio_pci_generic DPDK PMD so let's change it accordingly:

```
vim ~/cn2-23.1/K8s/single-cluster/single_cluster_deployer_example.yaml
dpdk:

        #dpdkUioDriver: vfio-pci
        dpdkUioDriver: uio_pci_generic
```

Review the DPDK Forwarding, Services, and Control thread cores in the deployer file and change it as per your setup, default values are:

```
dpdk:
```

```
        cpuCoreMask: 2,3,14,15
        dpdkCommandAdditionalArgs: --yield_option 0
        dpdkCtrlThreadMask: 4,5,16,17
        serviceCoreMask: 4,5,16,17
```

Starting CN2, 22.3 DPDK nodes require the following label:

```
kubectl label node worker2 agent-mode=dpdk
kubectl label node worker3 agent-mode=dpdk
```

Kick-Off CN2 Deployment

Deploy the CN2 cluster with the kubectl command:

```
kubectl apply -f ~/cn2-23.1/K8s/single-cluster/single_cluster_deployer_
example.yaml
namespace/contrail created
namespace/contrail-deploy created
namespace/contrail-system created
serviceaccount/contrail-deploy-serviceaccount created
serviceaccount/contrail-system-serviceaccount created
serviceaccount/contrail-serviceaccount created
clusterrole.rbac.authorization.K8s.io/contrail-deploy-role created
clusterrole.rbac.authorization.K8s.io/contrail-role created
clusterrole.rbac.authorization.K8s.io/contrail-system-role created
clusterrolebinding.rbac.authorization.K8s.io/contrail-deploy-rolebinding
created
clusterrolebinding.rbac.authorization.K8s.io/contrail-rolebinding created
clusterrolebinding.rbac.authorization.K8s.io/contrail-system-rolebinding
created
configmap/contrail-K8s-controller-cm created
secret/registrypullsecret created
secret/registrypullsecret created
secret/registrypullsecret created
deployment.apps/contrail-K8s-deployer created
configmap/contrail-cr created
job.batch/apply-contrail created
configmap/contrail-network-config created
```

CN2 Deployment Verification

Check the nodes' status:

```
kubectl get nodes -o wide
```

```
NAME          STATUS   ROLES           AGE     VERSION   INTERNAL-IP
EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION      CONTAINER-RUNTIME
controller1   Ready    control-plane   5d18h   v1.25.0   192.168.24.87
<none>        Ubuntu 20.04.3 LTS   5.4.0-135-generic   containerd://1.7.0
controller2   Ready    control-plane   5d18h   v1.25.0   192.168.24.88
<none>        Ubuntu 20.04.3 LTS   5.4.0-135-generic   containerd://1.7.0
controller3   Ready    control-plane   5d18h   v1.25.0   192.168.24.89
<none>        Ubuntu 20.04.3 LTS   5.4.0-135-generic   containerd://1.7.0
worker1       Ready    <none>          5d18h   v1.25.0   192.168.24.90
<none>        Ubuntu 20.04.3 LTS   5.4.0-135-generic   containerd://1.7.0
worker2       Ready    <none>          5d18h   v1.25.0   192.168.24.91
<none>        Ubuntu 20.04.3 LTS   5.4.0-135-generic   containerd://1.7.0
worker3       Ready    <none>          5d18h   v1.25.0   192.168.24.92
<none>        Ubuntu 20.04.3 LTS   5.4.0-135-generic   containerd://1.7.0
```

Check pod status:

```
kubectl get pods -o wide -n contrail | grep vrouter
contrail-vrouter-masters-56kn8                       3/3     Running   1
(5d17h ago)   5d18h   192.168.24.87   controller1   <none>
<none>
contrail-vrouter-masters-kzpzm                       3/3     Running   1
(5d17h ago)   5d18h   192.168.24.89   controller3   <none>
<none>
contrail-vrouter-masters-vm6m2                       3/3     Running   1
(5d17h ago)   5d18h   192.168.24.88   controller2   <none>
<none>
contrail-vrouter-dpdk-nodes-2p5sn                    3/3     Running   0
3h47m   192.168.24.91   worker2       <none>           <none>
contrail-vrouter-dpdk-nodes-cn2ks                    3/3     Running   0
3h47m   192.168.24.92   worker3       <none>           <none>
contrail-vrouter-nodes-mjlgj                         3/3     Running   0
3h47m   192.168.24.90   worker1       <none>           <none>
```

Verify if each DPDK worker node interface is bound with DPDK PMD (https://doc.dpdk.org/guides/tools/devbind.html):

```
python3 dpdk-devbind -s


Network devices using DPDK-compatible driver
============================================
0000:01:00.0 'Ethernet Controller 10-Gigabit X540-AT2 1528' drv=uio_pci_
generic unused=ixgbe,vfio-pci


Network devices using kernel driver
===================================
```

```
0000:01:00.1 'Ethernet Controller 10-Gigabit X540-AT2 1528' if=eno2
drv=ixgbe unused=vfio-pci,uio_pci_generic

0000:01:10.1 'X540 Ethernet Controller Virtual Function 1515' if=eno2v0
drv=ixgbevf unused=vfio-pci,uio_pci_generic

0000:01:10.3 'X540 Ethernet Controller Virtual Function 1515' if=eno2v1
drv=ixgbevf unused=vfio-pci,uio_pci_generic

0000:01:10.5 'X540 Ethernet Controller Virtual Function 1515' if=eno2v2
drv=ixgbevf unused=vfio-pci,uio_pci_generic

0000:01:10.7 'X540 Ethernet Controller Virtual Function 1515' if=eno2v3
drv=ixgbevf unused=vfio-pci,uio_pci_generic

0000:01:11.1 'X540 Ethernet Controller Virtual Function 1515' if=eno2v4
drv=ixgbevf unused=vfio-pci,uio_pci_generic

0000:01:11.3 'X540 Ethernet Controller Virtual Function 1515' if=eno2v5
drv=ixgbevf unused=vfio-pci,uio_pci_generic

0000:01:11.5 'X540 Ethernet Controller Virtual Function 1515' if=eno2v6
drv=ixgbevf unused=vfio-pci,uio_pci_generic

0000:01:11.7 'X540 Ethernet Controller Virtual Function 1515' if=
drv=ixgbevf unused=vfio-pci,uio_pci_generic

0000:08:00.0 'I350 Gigabit Network Connection 1521' if=eno3 drv=igb
unused=vfio-pci,uio_pci_generic *Active*

0000:08:00.1 'I350 Gigabit Network Connection 1521' if=eno4 drv=igb
unused=vfio-pci,uio_pci_generic *Active*
```

Local Container Registry and Custom Image

Let's create a container image registry in the "deployer-node" VM which will be used in later stages:

```
sudo docker run -d -p 5000:5000 --restart=always --name registry
registry:2


sudo cat > /etc/docker/daemon.json <<END_OF_SCRIPT
{
  "insecure-registries" : ["deployer-node.maas:5000"]
}
END_OF_SCRIPT


sudo service docker restart
```

Let's build a container image that we will use in the rest of the sections:

```
mkdir -p ~/docker_images/ubuntu
cd ~/docker_images/ubuntu
sudo cat > Dockerfile <<END_OF_SCRIPT
FROM ubuntu:focal
RUN apt update
```

```
# Install packages required for traffic libraries
RUN DEBIAN_FRONTEND=noninteractive apt-get install -y --no-install-recommends tzdata tshark
RUN apt install -y build-essential libssl-dev libffi-dev net-tools iperf3 hping3 python3-pip python-is-python3 libnuma1
RUN apt install -y iputils-ping curl wget vim tcpdump
RUN pip install ipaddress pyYaml
# Install packages related to other services
# Post installation of bird2 needs display which fails on docker env but the binary installation is fine hence the WA
RUN apt install -y keepalived bird2 || echo "Success: WA for bird2 installation"
COPY traffic_entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
END_OF_SCRIPT

sudo cat > traffic_entrypoint.sh <<END_OF_SCRIPT
#!/bin/bash
set -x
echo $HOSTNAME > /tmp/index.html
# Start a Service at port 7868 for liveness probe
python3 -m http.server 7868 --directory /tmp/ &
echo $! >& /var/run/webservice-port-7868.pid
# Update a file for liveness probe
echo "Alive" >& /tmp/am_i_alive
if [ -z $@ ]
then
sleep 3650d
else
exec $@
fi
END_OF_SCRIPT

docker build -t deployer-node.maas:5000/ubuntu-traffic:latest .
docker push deployer-node.maas:5000/ubuntu-traffic:latest

curl http://deployer-node.maas:5000/v2/_catalog | jq .repositories[]
curl http://deployer-node.maas:5000/v2/ubuntu-traffic/tags/list | jq .
```

Modify /etc/crio/crio.conf in all Kubernetes nodes in the cluster to add a reference for local container image registry. After the line “crio.image” add the following two lines and restart the crio service:

```
vim /etc/crio/crio.conf


[crio.image]
insecure_registries = [“deployer-node.maas:5000”]
registries = [“deployer-node.maas:5000”]
systemctl daemon-reload && systemctl restart crio
```

Chapter 6

CN2 Custom Resources

This chapter discusses various CN2 custom resources definitions along with working examples of each feature.

CN2 Custom Resources Definition (CRD)

A custom resource is an extension of the Kubernetes API that is not necessarily available in a default Kubernetes installation. It represents a customization of a particular Kubernetes installation. However, many core Kubernetes functions are now built using custom resources, making Kubernetes more modular. More about K8s CRD can be found here:(https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/).

CN2 creates its own Custom Resources using CRD and those can be explored using the following approach. This chapter explores CN2 CRDs in length.

```
kubectl api-resources
kubectl api-resources  | grep contrail
kubectl explain subnet.spec --recursive
kubectl explain vn.spec --recursive
kubectl explain vn.spec.providerNetworkReference
kubectl explain vn.spec.virtualNetworkProperties
```

Default Pod Network

CN2 creates a Pod network as part of cluster deployment and primary interface for each Pod attached with that default Pod network. CN2 custom resource name for the Pod network is a Virtual Network. A Virtual Network is associated with another construct called Subnet for the purpose of IP Address Management (IPAM). Besides the default Pod Virtual Network, additional Virtual Networks can be defined using CN2 Custom Resources Definition (CRD) constructs:

```
kubectl get subnet -A

NAMESPACE                                         NAME
CIDR               USAGE   STATE     AGE

contrail-K8s-kubemanager-cluster-local-contrail   default-podnetwork-pod-
v4-subnet       10.233.64.0/18     0.06%    Success   5d21h

contrail-K8s-kubemanager-cluster-local-contrail   default-servicenetwork-
pod-v4-subnet   10.233.0.0/18      0.05%    Success   5d21h


kubectl describe subnet default-podnetwork-pod-v4-subnet -n contrail-K8s-
kubemanager-cn2-cluster-local-contrail
```

```
kubectl get vn -A
NAMESPACE                                          NAME
VNI   IP FAMILIES   STATE     AGE
contrail-K8s-kubemanager-cluster-local-contrail   default-podnetwork
2     v4            Success   5d21h
contrail-K8s-kubemanager-cluster-local-contrail   default-servicenetwork
1     v4            Success   5d21h


kubectl describe vn default-podnetwork -n contrail-K8s-kubemanager-cn2-
cluster-local-contrail
```

In the following example, three Pods on default-podnetwork are created and traffic across the Pods is tested:

```
cat > default-vn-test-pods.yaml <<END_OF_SCRIPT
apiVersion: v1
kind: Pod
metadata:
  name: default-vn-test-pod-1
spec:
  containers:
  - name: default-vn-test-pod-1
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", 'echo The app is running! && sleep
3600' ]
  nodeName: worker1
---
apiVersion: v1
kind: Pod
metadata:
  name: default-vn-test-pod-2
spec:
  containers:
  - name: default-vn-test-pod-2
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", 'echo The app is running! && sleep
3600' ]
  nodeName: worker2
---
apiVersion: v1
```

```
kind: Pod
metadata:
  name: default-vn-test-pod-3
spec:
  containers:
  - name: default-vn-test-pod-3
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", 'echo The app is running! && sleep
3600' ]
  nodeName: worker3
END_OF_SCRIPT
```

Create the Pods:

```
kubectl apply -f default-vn-test-pods.yaml
pod/default-vn-test-pod-1 created
pod/default-vn-test-pod-2 created
pod/default-vn-test-pod-3 created

kubectl get pods | grep default-vn-test
default-vn-test-pod-1   1/1     Running   0          44s
default-vn-test-pod-2   1/1     Running   0          44s
default-vn-test-pod-3   1/1     Running   0          44s

kubectl exec -it default-vn-test-pod-1 -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
19: eth0@if20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:0d:5f:f0:d2:d8 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.233.69.0/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::18c0:92ff:fe5c:42b2/64 scope link
       valid_lft forever preferred_lft forever
```

```
kubectl exec -it default-vn-test-pod-2 -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
23: eth0@if24: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:b3:45:c6:33:36 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.233.67.2/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::2077:90ff:fe57:8f18/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec -it default-vn-test-pod-3 -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
19: eth0@if20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:58:24:c4:2d:59 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.233.68.0/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::ecc6:89ff:fed9:ac6a/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec -it default-vn-test-pod-1 -- ping 10.233.67.2 -c1
PING 10.233.67.2 (10.233.67.2) 56(84) bytes of data.
64 bytes from 10.233.67.2: icmp_seq=1 ttl=64 time=1.92 ms

--- 10.233.67.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.915/1.915/1.915/0.000 ms
```

```
kubectl exec -it default-vn-test-pod-1 -- ping 10.233.68.0 -c1
PING 10.233.68.0 (10.233.68.0) 56(84) bytes of data.
64 bytes from 10.233.68.0: icmp_seq=1 ttl=64 time=2.03 ms

--- 10.233.68.0 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.028/2.028/2.028/0.000 ms
```

Custom-PodNetwork

By default, each Pod will have interfaced from the default-podnetwork and all the Pods across all the namespaces on default-PodNetwork can talk to each other, and this also inherits the limitation on maximum allowed Pods IPs due to subnet mask of default-podnetwork. CN2 has a feature to create a Pod without attaching it to default-podnetwork and attaching Pod primary interface with a CN2 custom-podnetwork.

In the following example, we will create two Pods and each Pod will have a primary interface from the CN2 custom-podNetwork. A Network Attachment Definition (NAD) 'test-vn' is created by adding a property "podNetwork: true" under annotations juniper.net/networks and then NAD 'test-vn' is referred in the Pod definition by adding annotation to "net.juniper.contrail.podnetwork":

```
cat > custom_pod_network.yaml <<END_OF_SCRIPT
---
apiVersion: v1
kind: Namespace
metadata:
  name: test-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: test-vn
  namespace: test-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.10.10.0/24",
      "podNetwork": true
    }'
spec:
  config: '{
```

```
  "cniVersion": "0.3.1",
  "name": "test-vn",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: test-vn-test-pod-1
  namespace: test-ns
  annotations:
    net.juniper.contrail.podnetwork: test-ns/test-vn
spec:
  containers:
  - name: test-vn-test-pod-1
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
---
apiVersion: v1
kind: Pod
metadata:
  name: test-vn-test-pod-2
  namespace: test-ns
  annotations:
    net.juniper.contrail.podnetwork: test-ns/test-vn
spec:
  containers:
  - name: test-vn-test-pod-2
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
---
apiVersion: v1
kind: Pod
metadata:
  name: test-vn-test-pod-3
  namespace: test-ns
  annotations:
    net.juniper.contrail.podnetwork: test-ns/test-vn
```

```
spec:
  containers:
  - name: test-vn-test-pod-3
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
END_OF_SCRIPT
```

Create NADs, Pods, and verify:

```
kubectl apply -f custom_pod_network.yaml
namespace/test-ns created
networkattachmentdefinition.K8s.cni.cncf.io/test-vn created
pod/test-vn-test-pod-1 created
pod/test-vn-test-pod-2 created
pod/test-vn-test-pod-3 created



kubectl get pods -n test-ns
NAME                  READY   STATUS    RESTARTS   AGE
test-vn-test-pod-1    1/1     Running   0          2m11s
test-vn-test-pod-2    1/1     Running   0          26s
test-vn-test-pod-3    1/1     Running   0          2m11s



for i in {1..3};do kubectl exec -it -n test-ns test-vn-test-pod-${i} /
bin/bash -- ip addr show eth0;done
337: eth0@if338: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:bf:af:45:72:56 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.10.10.3/24 brd 10.10.10.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::7c9b:24ff:feda:830e/64 scope link
       valid_lft forever preferred_lft forever
3969: eth0@if3970: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP group default
    link/ether 02:09:44:00:a3:e0 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.10.10.4/24 brd 10.10.10.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::e016:86ff:fedf:11c4/64 scope link
       valid_lft forever preferred_lft forever
632: eth0@if633: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
```

```
state UP group default
    link/ether 02:56:8f:ea:f9:56 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.10.10.2/24 brd 10.10.10.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::4827:21ff:febb:2923/64 scope link
       valid_lft forever preferred_lft forever

for i in {1..3};do kubectl exec -it -n test-ns test-vn-test-pod-1 /bin/
bash -- ping 10.10.10.${i} -c1;done
PING 10.10.10.1 (10.10.10.1) 56(84) bytes of data.
64 bytes from 10.10.10.1: icmp_seq=1 ttl=64 time=1.28 ms

--- 10.10.10.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.276/1.276/1.276/0.000 ms
PING 10.10.10.2 (10.10.10.2) 56(84) bytes of data.
64 bytes from 10.10.10.2: icmp_seq=1 ttl=64 time=3.63 ms

--- 10.10.10.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.632/3.632/3.632/0.000 ms
PING 10.10.10.3 (10.10.10.3) 56(84) bytes of data.
64 bytes from 10.10.10.3: icmp_seq=1 ttl=64 time=0.132 ms

--- 10.10.10.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.132/0.132/0.132/0.000 ms

for i in {1..3};do kubectl exec -it -n test-ns test-vn-test-pod-3 /bin/
bash -- ping 10.10.10.${i} -c1;done
PING 10.10.10.1 (10.10.10.1) 56(84) bytes of data.
64 bytes from 10.10.10.1: icmp_seq=1 ttl=64 time=1.24 ms

--- 10.10.10.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.243/1.243/1.243/0.000 ms
PING 10.10.10.2 (10.10.10.2) 56(84) bytes of data.
64 bytes from 10.10.10.2: icmp_seq=1 ttl=64 time=0.115 ms
```

```
--- 10.10.10.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.115/0.115/0.115/0.000 ms
PING 10.10.10.3 (10.10.10.3) 56(84) bytes of data.
64 bytes from 10.10.10.3: icmp_seq=1 ttl=64 time=3.60 ms

--- 10.10.10.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.600/3.600/3.600/0.000 ms
```

Pod Additional Interfaces

CN2 natively supports attachment of multiple interfaces (https://www.juniper.net/documentation/us/en/software/cn-cloud-native22/cn-cloud-native-feature-guide/cn-cloud-native-network-feature/topics/task/cn-cloud-native-multiple-interface-pod.html) to a Pod using the CN2 VirtualNetwork construct. Other CNIs use Multus Meta CNI (https://github.com/K8snetworkplumbingwg/multus-cni) to attach multiple interfaces with a Pod, and CN2 supports interoperability with Multus as well. Also, note that once Multus is enabled in a CN2 cluster, you need to define NAD for attaching Pods to different networks. NAD subsequently creates a corresponding VirtualNetwork and Subnet in CN2.

In the following example we create three Pods and each Pod will have primary interface from default-podnetwork and an additional interface will also be attached to each Pod from an additional VN created by NAD 'red-1'. The NAD 'red-1' is referred to in Pod definition by adding an annotation to "K8s.v1.cni.cncf.io/networks":

```
cat > nad-red-1-pods.yaml  <<END_OF_SCRIPT
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: red-1
  namespace: default
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "172.16.10.0/24"
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "red-1",
```

```
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-1-red-1
  annotations:
    K8s.v1.cni.cncf.io/networks: red-1
spec:
  containers:
  - name: pod-1-red-1
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", 'echo The app is running! && sleep 
3600' ]
  nodeName: worker1
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-2-red-1
  annotations:
    K8s.v1.cni.cncf.io/networks: red-1
spec:
  containers:
  - name: pod-2-red-1
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  nodeName: worker2
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-3-red-1
  annotations:
    K8s.v1.cni.cncf.io/networks: red-1
spec:
```

```
  containers:
  - name: pod-3-red-1
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  nodeName: worker3
END_OF_SCRIPT
```

Create NAD and Pods:

```
kubectl apply  -f nad-red-1-pods.yaml
networkattachmentdefinition.K8s.cni.cncf.io/red-1 created
pod/pod-1-red-1 created
pod/pod-2-red-1 created
pod/pod-3-red-1 created

kubectl get pods | grep red
pod-1-red-1             1/1     Running   0          62s
pod-2-red-1             1/1     Running   0          62s
pod-3-red-1             1/1     Running   0          62s

kubectl exec pod-1-red-1 -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
25: eth0@if26: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:bb:bd:52:30:ea brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.233.69.1/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::208f:4aff:fe16:a11c/64 scope link
       valid_lft forever preferred_lft forever
27: eth1@if28: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:71:75:e1:25:ff brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.16.10.3/24 brd 172.16.10.255 scope global eth1
       valid_lft forever preferred_lft forever
```

```
    inet6 fe80::c8e:d0ff:fee1:8fc9/64 scope link
       valid_lft forever preferred_lft forever
kubectl exec pod-2-red-1 -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
29: eth0@if30: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:dc:c8:0c:c7:6b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.233.67.3/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::4810:eeff:fe6a:d1f3/64 scope link
       valid_lft forever preferred_lft forever
31: eth1@if32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:08:f2:35:a5:ee brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.16.10.2/24 brd 172.16.10.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::a811:57ff:fedf:d315/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec pod-3-red-1 -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
25: eth0@if26: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:27:6a:66:65:ea brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.233.68.1/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::b096:beff:fe83:85f2/64 scope link
       valid_lft forever preferred_lft forever
```

```
27: eth1@if28: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:0a:9f:36:52:86 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.16.10.4/24 brd 172.16.10.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::f03d:ddff:fe0b:81e3/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec pod-1-red-1 -- ping 172.16.10.2 -c1
PING 172.16.10.2 (172.16.10.2) 56(84) bytes of data.
64 bytes from 172.16.10.2: icmp_seq=1 ttl=64 time=1.89 ms

--- 172.16.10.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.885/1.885/1.885/0.000 ms

kubectl exec pod-1-red-1 -- ping 172.16.10.4 -c1
PING 172.16.10.4 (172.16.10.4) 56(84) bytes of data.
64 bytes from 172.16.10.4: icmp_seq=1 ttl=64 time=1.85 ms

--- 172.16.10.4 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.848/1.848/1.848/0.000 ms
```

Inter VirtualNetwork Communication

By default, traffic between Pods running on different VNs is restricted. There are two ways to allow communication between Pods on different VNs.

Virtual Network Router (VNR)

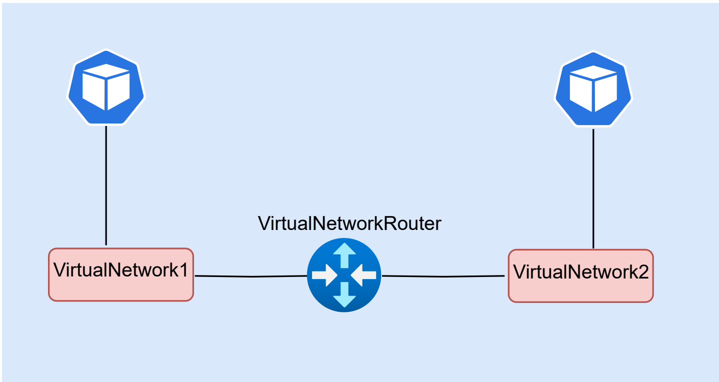


Figure 7 Inter-Virtual Networks Connectivity via Virtual Network Router

A CN2 custom resource, VirtualNetworkRouter,(https://www.juniper.net/documentation/us/en/software/cn-cloud-native22/cn-cloud-native-feature-guide/cn-cloud-native-network-feature/topics/concept/Contrail_Network_Policy_Implementation_in_CN2.html) can be used to allow communication between VNs. VNR has two subcategories (Mesh, and, Hub or Spoke VNR). Keyword type with value (Mesh, Hub or Spoke) can be used while creating a VNR. Inter-VNR communication is governed by VNR traffic rules :

- Mesh allows communication to all workload across all VNs.
- Hub VNR workloads can communicate with Spoke VNR workloads and vice versa.
- Spoke VNR workloads cannot communicate with other Spoke VNR workloads.
- Mesh VNR to Hub VNR communication is not allowed.

Once inter-VN traffic is allowed by using VNR then we can further segment the traffic using K8s Network Policy or Contrail Security Policy which are described in later sections in this chapter.

BGP Route-target (RT) Community

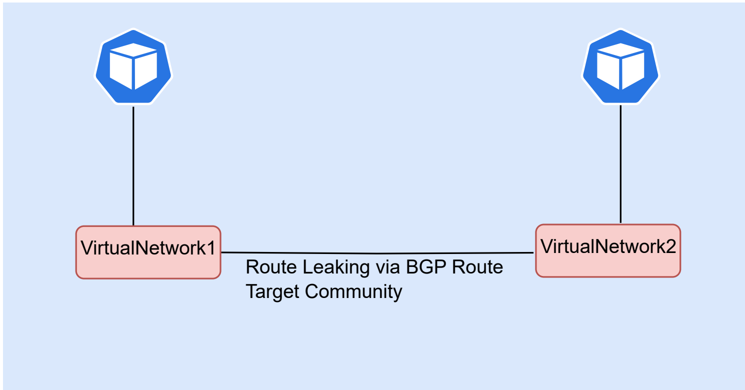


Figure 8 Inter-Virtual Network Route Leaking via BGP Route Target Community

Route-target (RT) Community can also be used to exchange routes between VNs. RTs are extended communities that are appended to a route as a BGP attribute. They are used to define which routes are exported and imported into a VRF routing table. The downside of this approach is that you lose the KNP and CSP functionalities. Inter-VN communication is easier to achieve by using RT but it's values should be carefully assigned to each VN (once manual RT values are assigned to VNs) else it could break multi tenancy isolation by allowing VNs to communicate with each other.

Mesh VirtualNetworkRouter

Okay, all the VNs connected to a Mesh VNR can communicate with each other and then one Mesh VNR can leak the routes to another Mesh VNR.

In the following example, we will create four NADs (vn1, v2, vn3 and vn4) in a namespace (ns1) and each NAD will have a Pod attached to it. Two VNRs are also created (vnr1 & vnr2). Furthermore, vnr1 is attached to NADs (vn1 & vn2) by using property "virtualNetworkSelector" spec sections and matching labels of vn1 and vn2. Similarly, vnr2 is attached to NADs (vn3 & vn4).

Full mesh between vnr1 and vnr2 is created by importing VNR into each other using virtualNetworkRouterSelector property under spec, import stanza. While creating NADs we are also using routeTargetList and importRouteTargetList properties under annotations "juniper.net/networks", these two properties are used to exchange routes with VNs in a different namespace which will be explained in the next section.

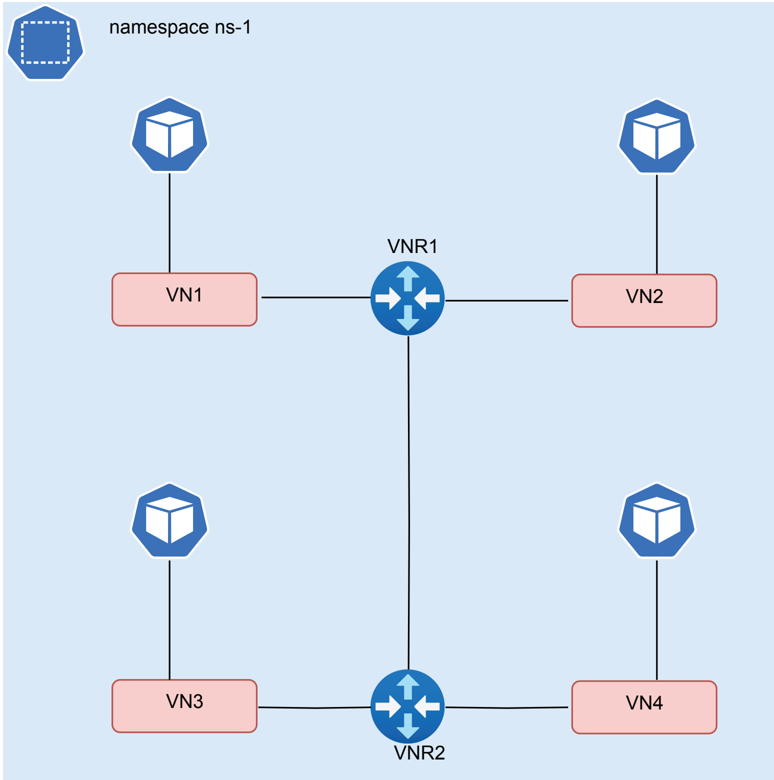


Figure 9 *Mesh Virtual Network Router*

```
cat > intra_namespace_mesh_vnr.yaml <<END_OF_SCRIPT
apiVersion: v1
kind: Namespace
metadata:
  labels:
    ns: ns1
  name: ns1
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vn1
  labels:
    vn: vn1
  namespace: ns1
```

```
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.20.1.0/24",
      "routeTargetList": ["target:64562:2101"],
      "importRouteTargetList": ["target:64562:2105",
"target:64562:2106"],
      "podNetwork": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "vn1",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  labels:
    vn: vn2
  name: vn2
  namespace: ns1
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.20.2.0/24",
      "routeTargetList": ["target:64562:2102"],
      "importRouteTargetList": ["target:64562:2105",
"target:64562:2106"],
      "podNetwork": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "vn2",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
```

```
metadata:
  name: vn3
  labels:
    vn: vn3
  namespace: ns1
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.20.3.0/24",
      "routeTargetList": ["target:64562:2103"],
      "podNetwork": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "vn3",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  labels:
    vn: vn4
  name: vn4
  namespace: ns1
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.20.4.0/24",
      "routeTargetList": ["target:64562:2104"],
      "podNetwork": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "vn4",
  "type": "contrail-K8s-cni"
}'
---
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  namespace: ns1
  annotations:
   net.juniper.contrail.podnetwork: ns1/vn1
spec:
  containers:
  - name: pod1
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    securityContext:
      privileged: true
      capabilities:
        add:
          - NET_ADMIN
  nodeName: worker4
---
apiVersion: v1
kind: Pod
metadata:
  name: pod2
  namespace: ns1
  annotations:
   net.juniper.contrail.podnetwork: ns1/vn2
spec:
  containers:
  - name: pod2
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    securityContext:
      privileged: true
      capabilities:
        add:
          - NET_ADMIN
  nodeName: worker5
---
apiVersion: v1
kind: Pod
```

```
metadata:
  name: pod3
  namespace: ns1
  annotations:
   net.juniper.contrail.podnetwork: ns1/vn3
spec:
  containers:
  - name: pod3
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    securityContext:
      privileged: true
      capabilities:
        add:
          - NET_ADMIN
  nodeName: worker6
---
apiVersion: v1
kind: Pod
metadata:
  name: pod4
  namespace: ns1
  annotations:
   net.juniper.contrail.podnetwork: ns1/vn4
spec:
  containers:
  - name: pod4
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    securityContext:
      privileged: true
      capabilities:
        add:
          - NET_ADMIN
  nodeName: worker4
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: ns1
```

```
  name: vnr1
  annotations:
    core.juniper.net/display-name: vnr1
  labels:
    vnr: vnr1
spec:
  type: mesh
  virtualNetworkSelector:
    matchExpressions:
      - key: vn
        operator: In
        values: [vn1, vn2]
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: vnr2
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: ns1
  name: vnr2
  annotations:
    core.juniper.net/display-name: vnr2
  labels:
    vnr: vnr2
spec:
  type: mesh
  virtualNetworkSelector:
    matchExpressions:
      - key: vn
        operator: In
        values: [vn3, vn4]
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
```

```
            vnr: vnr1
---
END_OF_SCRIPT
```

Now create NAD, PodS and VNRs:

```
kubectl apply -f intra_namespace_mesh_vnr
namespace/ns1 created
networkattachmentdefinition.K8s.cni.cncf.io/vn1 created
networkattachmentdefinition.K8s.cni.cncf.io/vn2 created
networkattachmentdefinition.K8s.cni.cncf.io/vn3 created
networkattachmentdefinition.K8s.cni.cncf.io/vn4 created
pod/pod1 created
pod/pod2 created
pod/pod3 created
pod/pod4 created
virtualnetworkrouter.core.contrail.juniper.net/vnr1 created
virtualnetworkrouter.core.contrail.juniper.net/vnr2 created


kubectl get pods -n ns1
NAME   READY   STATUS    RESTARTS   AGE
pod1   1/1     Running   0          90s
pod2   1/1     Running   0          90s
pod3   1/1     Running   0          90s
pod4   1/1     Running   0          90s

kubectl get vnr -n ns1
NAME   TYPE   STATE     AGE
vnr1   mesh   Success   109s
vnr2   mesh   Success   108s

for i in {1..4}; do kubectl exec -it -n ns1 pod${i} /bin/bash -- ip addr show eth0; done
634: eth0@if635: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:ce:af:97:23:60 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.20.1.2/24 brd 10.20.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::2c66:d4ff:fe5b:84b7/64 scope link
```

```
       valid_lft forever preferred_lft forever
3971: eth0@if3972: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP group default
    link/ether 02:d8:07:c8:83:01 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.20.2.2/24 brd 10.20.2.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::2872:c8ff:fed7:3c7b/64 scope link
       valid_lft forever preferred_lft forever
339: eth0@if340: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:85:8e:bb:6f:e6 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.20.3.2/24 brd 10.20.3.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::f070:c4ff:feba:ee7c/64 scope link
       valid_lft forever preferred_lft forever
636: eth0@if637: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:94:f6:cf:55:81 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.20.4.2/24 brd 10.20.4.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::80b3:58ff:fe57:bd77/64 scope link
       valid_lft forever preferred_lft forever

for i in {1..4}; do kubectl exec -it -n ns1 pod1 /bin/bash -- ping
10.20.${i}.2 -c1; done
PING 10.20.1.2 (10.20.1.2) 56(84) bytes of data.
64 bytes from 10.20.1.2: icmp_seq=1 ttl=64 time=0.109 ms

--- 10.20.1.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.109/0.109/0.109/0.000 ms
PING 10.20.2.2 (10.20.2.2) 56(84) bytes of data.
64 bytes from 10.20.2.2: icmp_seq=1 ttl=64 time=0.083 ms

--- 10.20.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.083/0.083/0.083/0.000 ms
PING 10.20.3.2 (10.20.3.2) 56(84) bytes of data.
64 bytes from 10.20.3.2: icmp_seq=1 ttl=64 time=0.085 ms
```

```
--- 10.20.3.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.085/0.085/0.085/0.000 ms
PING 10.20.4.2 (10.20.4.2) 56(84) bytes of data.
64 bytes from 10.20.4.2: icmp_seq=1 ttl=64 time=0.167 ms

--- 10.20.4.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.167/0.167/0.167/0.000 ms

for i in {1..4}; do kubectl exec -it -n ns1 pod2 /bin/bash -- ping
10.20.${i}.2 -c1; done
PING 10.20.1.2 (10.20.1.2) 56(84) bytes of data.
64 bytes from 10.20.1.2: icmp_seq=1 ttl=64 time=3.07 ms

--- 10.20.1.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.068/3.068/3.068/0.000 ms
PING 10.20.2.2 (10.20.2.2) 56(84) bytes of data.
64 bytes from 10.20.2.2: icmp_seq=1 ttl=64 time=0.135 ms

--- 10.20.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.135/0.135/0.135/0.000 ms
PING 10.20.3.2 (10.20.3.2) 56(84) bytes of data.
64 bytes from 10.20.3.2: icmp_seq=1 ttl=64 time=2.58 ms

--- 10.20.3.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.579/2.579/2.579/0.000 ms
PING 10.20.4.2 (10.20.4.2) 56(84) bytes of data.
64 bytes from 10.20.4.2: icmp_seq=1 ttl=64 time=2.80 ms

--- 10.20.4.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.801/2.801/2.801/0.000 ms


for i in {1..4}; do kubectl exec -it -n ns1 pod3 /bin/bash -- ping
```

```
10.20.${i}.2 -c1; done
PING 10.20.1.2 (10.20.1.2) 56(84) bytes of data.
64 bytes from 10.20.1.2: icmp_seq=1 ttl=64 time=3.01 ms

--- 10.20.1.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.010/3.010/3.010/0.000 ms
PING 10.20.2.2 (10.20.2.2) 56(84) bytes of data.
64 bytes from 10.20.2.2: icmp_seq=1 ttl=64 time=2.58 ms

--- 10.20.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.584/2.584/2.584/0.000 ms
PING 10.20.3.2 (10.20.3.2) 56(84) bytes of data.
64 bytes from 10.20.3.2: icmp_seq=1 ttl=64 time=0.104 ms

--- 10.20.3.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.104/0.104/0.104/0.000 ms
PING 10.20.4.2 (10.20.4.2) 56(84) bytes of data.
64 bytes from 10.20.4.2: icmp_seq=1 ttl=64 time=3.07 ms

--- 10.20.4.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.066/3.066/3.066/0.000 ms

for i in {1..4}; do kubectl exec -it -n ns1 pod4 /bin/bash -- ping
10.20.${i}.2 -c1; done
PING 10.20.1.2 (10.20.1.2) 56(84) bytes of data.
64 bytes from 10.20.1.2: icmp_seq=1 ttl=63 time=1.18 ms

--- 10.20.1.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.176/1.176/1.176/0.000 ms
PING 10.20.2.2 (10.20.2.2) 56(84) bytes of data.
64 bytes from 10.20.2.2: icmp_seq=1 ttl=64 time=2.51 ms

--- 10.20.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

```
rtt min/avg/max/mdev = 2.513/2.513/2.513/0.000 ms
PING 10.20.3.2 (10.20.3.2) 56(84) bytes of data.
64 bytes from 10.20.3.2: icmp_seq=1 ttl=64 time=3.06 ms

--- 10.20.3.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.055/3.055/3.055/0.000 ms
PING 10.20.4.2 (10.20.4.2) 56(84) bytes of data.
64 bytes from 10.20.4.2: icmp_seq=1 ttl=64 time=0.146 ms

--- 10.20.4.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.146/0.146/0.146/0.000 ms
```

In the next example vn5 and vn6 are attached to vnr3 in ns2 and cross namespace communication is also achieved between ns1 vnr2 and ns2 vnr3 in a similar way to how vnr1 and vnr2 in ns1 were imported to each other in the preceding example. Moreover, vn5 and vn6 are connected to vn1 and vn2 by using importRouteTargetList property: for example, vn5 & vn6 are referring RT value of vn1 and vn2 from ns1 in importRouteTargetList under annotations "juniper.net/networks" stanza.

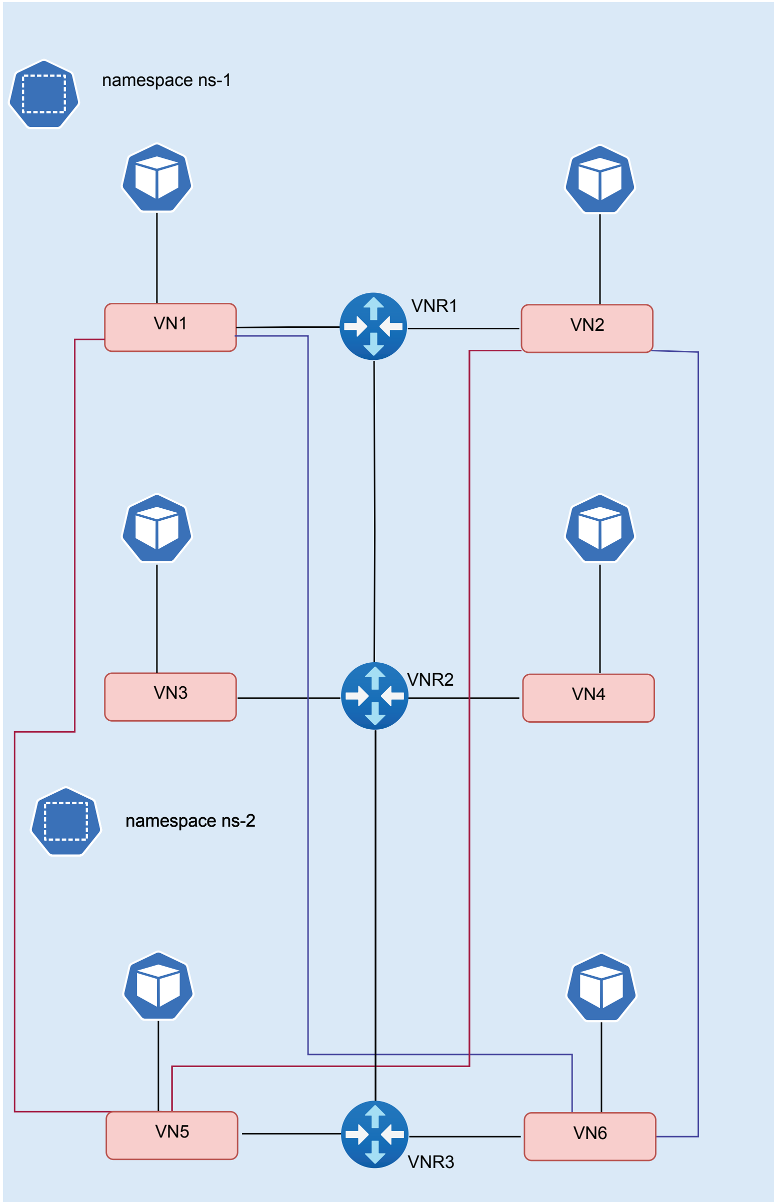


Figure 10 *Route Leaking via Route Target Community and Mesh VNR*

```
cat > inter_namespace_mesh_vnr.yaml  <<END_OF_SCRIPT
apiVersion: v1
kind: Namespace
```

```
metadata:
  labels:
    ns: ns2
  name: ns2
---
apiVersion: “K8s.cni.cncf.io/v1”
kind: NetworkAttachmentDefinition
metadata:
  name: “vn5”
  labels:
    vn: vn5
  namespace: ns2
  annotations:
    juniper.net/networks: ‘{
      “ipamV4Subnet”: “10.20.5.0/24”,
      “routeTargetList”: [“target:64562:2105”],
      “importRouteTargetList”: [“target:64562:2101”,
“target:64562:2102”],
      “podNetwork”: true
    }’
spec:
  config: ‘{
  “cniVersion”: “0.3.1”,
  “name”: “vn5”,
  “type”: “contrail-K8s-cni”
}’
---
apiVersion: “K8s.cni.cncf.io/v1”
kind: NetworkAttachmentDefinition
metadata:
  name: “vn6”
  labels:
    vn: vn6
  namespace: ns2
  annotations:
    juniper.net/networks: ‘{
      “ipamV4Subnet”: “10.20.6.0/24”,
      “routeTargetList”: [“target:64562:2106”],
      “importRouteTargetList”: [“target:64562:2101”,
```

```
“target:64562:2102”],
      “podNetwork”: true
    }’
spec:
  config: ‘{
  “cniVersion”: “0.3.1”,
  “name”: “vn6”,
  “type”: “contrail-K8s-cni”
}’
---
apiVersion: v1
kind: Pod
metadata:
  name: pod5
  namespace: ns2
  annotations:
    net.juniper.contrail.podnetwork: ns2/vn5
spec:
  containers:
  - name: pod5
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    securityContext:
      privileged: true
      capabilities:
        add:
          - NET_ADMIN
  nodeName: worker4
---
apiVersion: v1
kind: Pod
metadata:
  name: pod6
  namespace: ns2
  annotations:
    net.juniper.contrail.podnetwork: ns2/vn6
spec:
  containers:
  - name: pod6
```

```
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    securityContext:
      privileged: true
      capabilities:
        add:
          - NET_ADMIN
  nodeName: worker5
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: ns1
  name: vnr2
  annotations:
    core.juniper.net/display-name: vnr2
  labels:
    vnr: vnr2
spec:
  type: mesh
  virtualNetworkSelector:
    matchExpressions:
      - key: vn
        operator: In
        values: [vn3, vn4]
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: vnr1
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: vnr3
        namespaceSelector:
          matchLabels:
            ns: ns2
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
```

```
metadata:
  namespace: ns2
  name: vnr3
  annotations:
    core.juniper.net/display-name: vnr3
  labels:
    vnr: vnr3
spec:
  type: mesh
  virtualNetworkSelector:
    matchExpressions:
      - key: vn
        operator: In
        values: [vn5, vn6]
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: vnr2
        namespaceSelector:
          matchLabels:
            ns: ns1
END_OF_SCRIPT
---
```

Create NAD, PodS and VNRs:

```
kubectl apply -f inter_namespace_mesh_vnr.yaml
namespace/ns2 created
networkattachmentdefinition.K8s.cni.cncf.io/vn5 created
networkattachmentdefinition.K8s.cni.cncf.io/vn6 created
pod/pod5 created
pod/pod6 created
virtualnetworkrouter.core.contrail.juniper.net/vnr2 configured
virtualnetworkrouter.core.contrail.juniper.net/vnr3 created

kubectl get pods -n ns2
NAME   READY   STATUS    RESTARTS   AGE
pod5   1/1     Running   0          48s
pod6   1/1     Running   0          48s
```

```
kubectl get vnr -n ns2
NAME   TYPE   STATE     AGE
vnr3   mesh   Success   63s



for i in {5..6}; do kubectl exec -it -n ns2 pod${i} /bin/bash -- ip addr
show eth0; done
638: eth0@if639: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:05:c4:21:ef:d6 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.20.5.2/24 brd 10.20.5.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::70c7:a7ff:fe8a:8bad/64 scope link
       valid_lft forever preferred_lft forever
3973: eth0@if3974: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP group default
    link/ether 02:07:7a:57:bd:b8 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.20.6.2/24 brd 10.20.6.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::38ec:6fff:feea:d0e8/64 scope link
       valid_lft forever preferred_lft forever



for i in {1..6}; do kubectl exec -it -n ns2 pod5 /bin/bash -- ping
10.20.${i}.2 -c1; done
PING 10.20.1.2 (10.20.1.2) 56(84) bytes of data.
64 bytes from 10.20.1.2: icmp_seq=1 ttl=63 time=1.25 ms

--- 10.20.1.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.252/1.252/1.252/0.000 ms
PING 10.20.2.2 (10.20.2.2) 56(84) bytes of data.
64 bytes from 10.20.2.2: icmp_seq=1 ttl=64 time=3.01 ms

--- 10.20.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.010/3.010/3.010/0.000 ms
PING 10.20.3.2 (10.20.3.2) 56(84) bytes of data.
64 bytes from 10.20.3.2: icmp_seq=1 ttl=64 time=3.48 ms
```

```
--- 10.20.3.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.475/3.475/3.475/0.000 ms
PING 10.20.4.2 (10.20.4.2) 56(84) bytes of data.
64 bytes from 10.20.4.2: icmp_seq=1 ttl=63 time=1.29 ms

--- 10.20.4.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.285/1.285/1.285/0.000 ms
PING 10.20.5.2 (10.20.5.2) 56(84) bytes of data.
64 bytes from 10.20.5.2: icmp_seq=1 ttl=64 time=0.120 ms

--- 10.20.5.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.120/0.120/0.120/0.000 ms
PING 10.20.6.2 (10.20.6.2) 56(84) bytes of data.
64 bytes from 10.20.6.2: icmp_seq=1 ttl=64 time=3.24 ms

--- 10.20.6.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.239/3.239/3.239/0.000 ms
```

```
for i in {1..6}; do kubectl exec -it -n ns2 pod6 /bin/bash -- ping 10.20.${i}.2 -c1; done
PING 10.20.1.2 (10.20.1.2) 56(84) bytes of data.
64 bytes from 10.20.1.2: icmp_seq=1 ttl=64 time=2.97 ms

--- 10.20.1.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.969/2.969/2.969/0.000 ms
PING 10.20.2.2 (10.20.2.2) 56(84) bytes of data.
64 bytes from 10.20.2.2: icmp_seq=1 ttl=63 time=108 ms

--- 10.20.2.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 107.882/107.882/107.882/0.000 ms
PING 10.20.3.2 (10.20.3.2) 56(84) bytes of data.
64 bytes from 10.20.3.2: icmp_seq=1 ttl=64 time=2.74 ms
```

```
--- 10.20.3.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.739/2.739/2.739/0.000 ms
PING 10.20.4.2 (10.20.4.2) 56(84) bytes of data.
64 bytes from 10.20.4.2: icmp_seq=1 ttl=64 time=2.51 ms

--- 10.20.4.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.512/2.512/2.512/0.000 ms
PING 10.20.5.2 (10.20.5.2) 56(84) bytes of data.
64 bytes from 10.20.5.2: icmp_seq=1 ttl=64 time=2.66 ms

--- 10.20.5.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.655/2.655/2.655/0.000 ms
PING 10.20.6.2 (10.20.6.2) 56(84) bytes of data.
64 bytes from 10.20.6.2: icmp_seq=1 ttl=64 time=0.150 ms

--- 10.20.6.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.150/0.150/0.150/0.000 ms
```

Hub & Spoke VirtualNetworkRouter

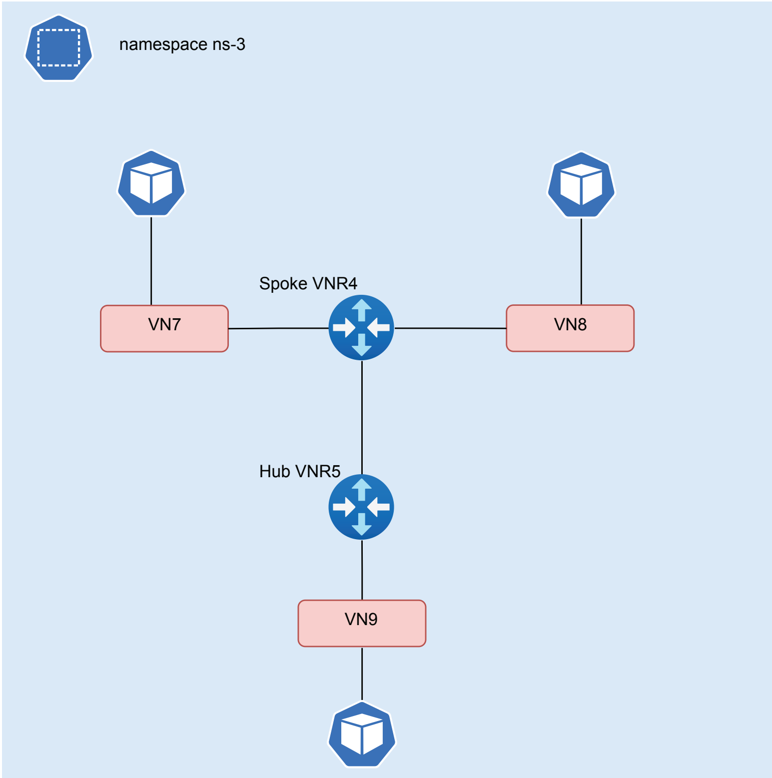


Figure 11 *Hub and Spoke Virtual Network Router*

In the following example, we are creating three NADs and two VNRs to demonstrate hub and spoke topology. Spoke vnr4 has vn7 and vn8 attached to it and hub vnr5 vn5 vn9 attached to it. Inter VN communication on spoke VNR is not allowed and inter VN communication between hub and spoke VNR and vice versa is allowed:

```
cat > hub_spoke_vnr.yaml <<END_OF_SCRIPT
apiVersion: v1
kind: Namespace
metadata:
  labels:
    ns: ns3
  name: ns3
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
```

```
metadata:
  name: “vn7”
  labels:
    vn: vn7
  namespace: ns3
  annotations:
    juniper.net/networks: ‘{
      “ipamV4Subnet”: “10.20.7.0/24”,
      “routeTargetList”: [“target:64562:2107”]
    }’
spec:
  config: ‘{
  “cniVersion”: “0.3.1”,
  “name”: “vn7”,
  “type”: “contrail-K8s-cni”
}’
---
apiVersion: “K8s.cni.cncf.io/v1”
kind: NetworkAttachmentDefinition
metadata:
  name: “vn8”
  labels:
    vn: vn8
  namespace: ns3
  annotations:
    juniper.net/networks: ‘{
      “ipamV4Subnet”: “10.20.8.0/24”,
      “routeTargetList”: [“target:64562:2108”]
    }’
spec:
  config: ‘{
  “cniVersion”: “0.3.1”,
  “name”: “vn8”,
  “type”: “contrail-K8s-cni”
}’
---
apiVersion: “K8s.cni.cncf.io/v1”
kind: NetworkAttachmentDefinition
```

```
metadata:
  name: "vn9"
  labels:
    vn: vn9
  namespace: ns3
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.20.9.0/24",
      "routeTargetList": ["target:64562:2109"]
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "vn9",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: pod7
  namespace: ns3
  annotations:
    K8s.v1.cni.cncf.io/networks: vn7
spec:
  containers:
  - name: pod7
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    command: [ "/bin/bash", "-c" ]
    args:
      - ip route del default;
        ip route add default via 10.20.7.1;
        sleep infinity;
    securityContext:
      privileged: true
      capabilities:
        add:
          - NET_ADMIN
```

```
  nodeName: worker4
---
apiVersion: v1
kind: Pod
metadata:
  name: pod8
  namespace: ns3
  annotations:
    K8s.v1.cni.cncf.io/networks: vn8
spec:
  containers:
  - name: pod8
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    command: [ "/bin/bash", "-c" ]
    args:
      - ip route del default;
        ip route add default via 10.20.8.1;
        sleep infinity;
    securityContext:
      privileged: true
      capabilities:
        add:
          - NET_ADMIN
  nodeName: worker5
---
apiVersion: v1
kind: Pod
metadata:
  name: pod9
  namespace: ns3
  annotations:
    K8s.v1.cni.cncf.io/networks: vn9
spec:
  containers:
  - name: pod9
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    command: [ "/bin/bash", "-c" ]
    args:
```

```
      - ip route del default;
        ip route add default via 10.20.9.1;
        sleep infinity;
    securityContext:
      privileged: true
      capabilities:
        add:
          - NET_ADMIN
  nodeName: worker6
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: ns3
  name: vnr4
  annotations:
    core.juniper.net/display-name: vnr4
  labels:
    vnr: vnr4
spec:
  type: spoke
  virtualNetworkSelector:
    matchExpressions:
      - key: vn
        operator: In
        values: [vn7, vn8]
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: vnr5
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: ns3
  name: vnr5
  annotations:
```

```
    core.juniper.net/display-name: vnr5
  labels:
    vnr: vnr5
spec:
  type: hub
  virtualNetworkSelector:
    matchExpressions:
      - key: vn
        operator: In
        values: [vn9]
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: vnr4
END_OF_SCRIPT
```

Create NADs, Pods and VNRs:

```
kubectl apply -f hub_spoke_vnr.yaml
namespace/ns3 created
networkattachmentdefinition.K8s.cni.cncf.io/vn7 created
networkattachmentdefinition.K8s.cni.cncf.io/vn8 created
networkattachmentdefinition.K8s.cni.cncf.io/vn9 created
pod/pod7 created
pod/pod8 created
pod/pod9 created
virtualnetworkrouter.core.contrail.juniper.net/vnr4 created
virtualnetworkrouter.core.contrail.juniper.net/vnr5 created

kubectl get pods -n ns3
NAME   READY   STATUS    RESTARTS   AGE
pod7   1/1     Running   0          48s
pod8   1/1     Running   0          48s
pod9   1/1     Running   0          48s


for i in {7..9}; do kubectl exec -it -n ns3 pod${i} /bin/bash -- ip addr
show eth1;done
628: eth1@if629: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
```

```
state UP group default
    link/ether 02:1a:c6:e5:19:2c brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.20.7.2/24 brd 10.20.7.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::54fd:15ff:feca:ec68/64 scope link
       valid_lft forever preferred_lft forever
3963: eth1@if3964: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP group default
    link/ether 02:56:30:23:30:cb brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.20.8.2/24 brd 10.20.8.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::7887:11ff:fe52:2164/64 scope link
       valid_lft forever preferred_lft forever
333: eth1@if334: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:6d:b4:03:59:19 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.20.9.2/24 brd 10.20.9.255 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::d8e1:31ff:fee9:24bd/64 scope link
       valid_lft forever preferred_lft forever
```

```
for i in {7..9}; do kubectl exec -it -n ns3 pod7 /bin/bash -- ping
10.20.${i}.2 -c1 ;done
PING 10.20.7.2 (10.20.7.2) 56(84) bytes of data.
64 bytes from 10.20.7.2: icmp_seq=1 ttl=64 time=0.089 ms

--- 10.20.7.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.089/0.089/0.089/0.000 ms
PING 10.20.8.2 (10.20.8.2) 56(84) bytes of data.

--- 10.20.8.2 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

command terminated with exit code 1
PING 10.20.9.2 (10.20.9.2) 56(84) bytes of data.
64 bytes from 10.20.9.2: icmp_seq=1 ttl=64 time=3.54 ms
```

```
--- 10.20.9.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.538/3.538/3.538/0.000 ms

for i in {7..9}; do kubectl exec -it -n ns3 pod8 /bin/bash -- ping
10.20.${i}.2 -c1 ;done
PING 10.20.7.2 (10.20.7.2) 56(84) bytes of data.

--- 10.20.7.2 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

command terminated with exit code 1
PING 10.20.8.2 (10.20.8.2) 56(84) bytes of data.
64 bytes from 10.20.8.2: icmp_seq=1 ttl=64 time=0.070 ms

--- 10.20.8.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.070/0.070/0.070/0.000 ms
PING 10.20.9.2 (10.20.9.2) 56(84) bytes of data.
64 bytes from 10.20.9.2: icmp_seq=1 ttl=64 time=2.97 ms

--- 10.20.9.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.970/2.970/2.970/0.000 ms

for i in {7..9}; do kubectl exec -it -n ns3 pod9 /bin/bash -- ping
10.20.${i}.2 -c1 ;done
PING 10.20.7.2 (10.20.7.2) 56(84) bytes of data.
64 bytes from 10.20.7.2: icmp_seq=1 ttl=64 time=3.06 ms

--- 10.20.7.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.058/3.058/3.058/0.000 ms
PING 10.20.8.2 (10.20.8.2) 56(84) bytes of data.
64 bytes from 10.20.8.2: icmp_seq=1 ttl=64 time=2.96 ms

--- 10.20.8.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.962/2.962/2.962/0.000 ms
```

```
PING 10.20.9.2 (10.20.9.2) 56(84) bytes of data.
64 bytes from 10.20.9.2: icmp_seq=1 ttl=64 time=0.076 ms

--- 10.20.9.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.076/0.076/0.076/0.000 ms
```

K8s Network Policy (KNP)

CN2 supports KNP allowing developers to control ingress and egress traffic to a particular workload or end point. Ingress and egress traffic rules are created and applied on corresponding workloads to enforce traffic restrictions. Once a KNP is applied on workloads then an implicit deny rule will govern the traffic which is not allowed on corresponding workloads by KNP rules.

Intra VirtualNetwork KNP

By default, all workloads and Pods on a VN can communicate to each other, but you can restrict communication between workloads and Pods by applying ingress and egress KNP rules. The following example applies a KNP policy to allow TCP port 80 egress traffic from the frontend Pod. And on the middle ware Pod or service a KNP rule will be applied in the ingress direction to allow TCP port 80 traffic.

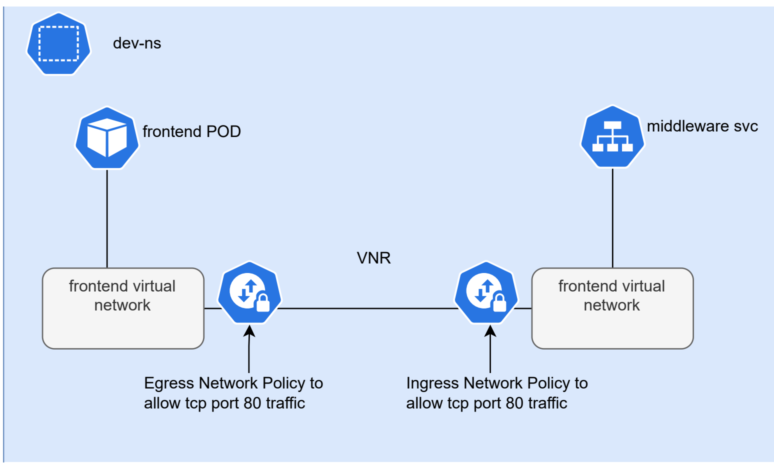


Figure 12 Intra Virtual Network Traffic Control via K8S Network Policy

```
cat > intra_vn_pods_svc_np.yml <<END_OF_SCRIPT
---
apiVersion: v1
kind: Namespace
metadata:
```

```
  name: dev-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: dev-vn
  namespace: dev-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.10.10.0/24",
      "podNetwork": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "dev-vn",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-1
  namespace: dev-ns
  labels:
    app: frontend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: frontend-dev-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker1
```

```
---
apiVersion: v1
kind: Pod
metadata:
  name: middleware-dev-2
  namespace: dev-ns
  labels:
    app: middleware-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: middleware-dev-2
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker2
---
apiVersion: v1
kind: Service
metadata:
  name: middleware-dev
  namespace: dev-ns
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  ports:
  - name: port-80
    targetPort: 80
    protocol: TCP
    port: 80
  selector:
    app: middleware-dev
  type: ClusterIP
---
```

```
apiVersion: networking.K8s.io/v1
kind: NetworkPolicy
metadata:
  name: frontend-dev
  namespace: dev-ns
spec:
  egress:
  - ports:
    - port: 80
      protocol: TCP
  podSelector:
    matchLabels:
      app: frontend-dev
  policyTypes:
  - Egress
---
apiVersion: networking.K8s.io/v1
kind: NetworkPolicy
metadata:
  name: middleware-dev
  namespace: dev-ns
spec:
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend-dev
    ports:
    - port: 80
      protocol: TCP
  podSelector:
    matchLabels:
      app: middleware-dev
  policyTypes:
  - Ingress
---
END_OF_SCRIPT
```

Create NAD, Pods, Service, and Network Policy:

```
kubectl apply -f intra_vn_pods_svc_np.yml
namespace/dev-ns created
networkattachmentdefinition.K8s.cni.cncf.io/dev-vn created
pod/frontend-dev-1 created
pod/middleware-dev-2 created
service/middleware-dev created
networkpolicy.networking.K8s.io/frontend-dev created
networkpolicy.networking.K8s.io/middleware-dev created

kubectl get pods -n dev-ns
NAME               READY   STATUS    RESTARTS   AGE
frontend-dev-1     1/1     Running   0          3m10s
middleware-dev-2   1/1     Running   0          3m10s

kubectl get svc -n dev-ns
NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
middleware-dev   ClusterIP   10.233.53.233   <none>        80/TCP    
3m19s

kubectl exec -n dev-ns middleware-dev-2 /bin/bash -- nohup python3 -m 
http.server 80 --directory /tmp/ &
[2] 81861

svc_ip=$(kubectl get svc -n dev-ns | grep middleware-dev | awk {‘print 
$3’})

kubectl exec -n dev-ns frontend-dev-1 /bin/bash -- curl $svc_ip
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  
Current
                                 Dload  Upload   Total   Spent    Left  
Speed
100    14  100    14    0     0   3500      0 --:--:-- --:--:-- --:--:--  
2800
deployer-node

kubectl exec -n dev-ns frontend-dev-1 /bin/bash -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group 
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

```
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
163: eth0@if164: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:ab:a4:28:0d:fb brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.10.10.2/24 brd 10.10.10.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::7888:8dff:fe29:8cde/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec -n dev-ns frontend-dev-1 /bin/bash -- ping 10.10.20.2 -c1
PING 10.10.20.2 (10.10.20.2) 56(84) bytes of data.

--- 10.10.20.2 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

Inter VirtualNetwork KNP

By default, traffic between workloads running on separate VNs is isolated. However, we can allow inter-VN communication by creating VNR and then further restricting the traffic as needed by applying KNP. The following example will create two VNs (i.e., a frontend and a middle ware) then those VNs are connected via a VNR (vn01). A KNP will be applied on the frontend Pods to allow TCP port 80 traffic in the egress direction and a corresponding ingress KNP will be applied on the middle ware service to allow TCP port 80 traffic.

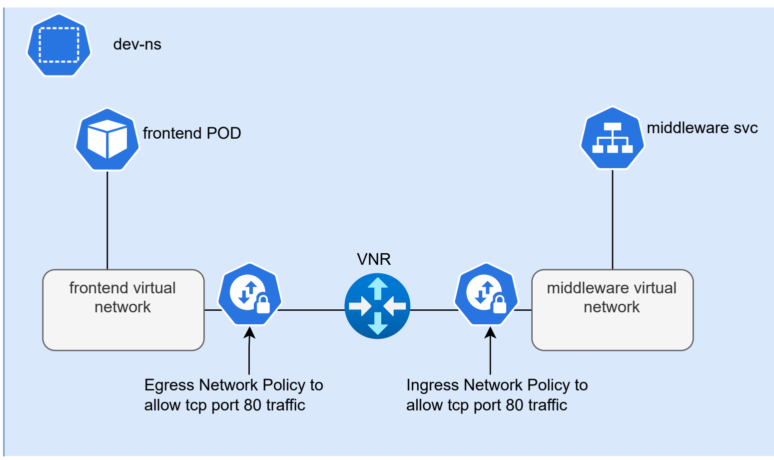


Figure 13 Inter Virtual Network Traffic Control via Network Policy and VNR

```
cat > inter_vn_pods.yaml <<END_OF_SCRIPT
---
apiVersion: v1
kind: Namespace
metadata:
  name: dev-ns
---
apiVersion: “K8s.cni.cncf.io/v1”
kind: NetworkAttachmentDefinition
metadata:
  name: frontend
  namespace: dev-ns
  annotations:
    juniper.net/networks: ‘{
      “ipamV4Subnet”: “10.10.10.0/24”,
      “podNetwork”: true
    }’
  labels:
    vn: mesh_vnr
spec:
  config: ‘{
  “cniVersion”: “0.3.1”,
  “name”: “frontend”,
  “type”: “contrail-K8s-cni”
}’
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-1
  namespace: dev-ns
  labels:
    app: frontend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/frontend
spec:
  containers:
    - name: frontend-dev-1
```

```
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker1
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: middleware
  namespace: dev-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.10.20.0/24",
      "podNetwork": true
    }'
  labels:
    vn: mesh_vnr
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "middleware",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: middleware-dev-1
  namespace: dev-ns
  labels:
    app: middleware-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/middleware
spec:
  containers:
```

```
    - name: middleware-dev-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker2
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: dev-ns
  name: vnr01
  annotations:
    core.juniper.net/display-name: vnr01
  labels:
      vnr: mesh_vnr
spec:
  type: mesh
  virtualNetworkSelector:
    matchLabels:
      vn: mesh_vnr
END_OF_SCRIPT
```

Create NADs, Pods and VNRs. At this stage Network Policy is not applied so all traffic from the frontend to middleware should be allowed:

```
kubectl apply -f inter_vn_pods_svc.yaml
namespace/dev-ns created
networkattachmentdefinition.K8s.cni.cncf.io/frontend created
pod/frontend-dev-1 created
networkattachmentdefinition.K8s.cni.cncf.io/middleware created
pod/middleware-dev-1 created
service/middleware-dev created
virtualnetworkrouter.core.contrail.juniper.net/vnr01 created


kubectl get pods -n dev-ns
NAME                  READY   STATUS    RESTARTS   AGE
```

```
frontend-dev-1     1/1     Running   0          3m38s
middleware-dev-1   1/1     Running   0          3m38s

kubectl get svc -n dev-ns
NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
middleware-dev   ClusterIP   10.233.53.77   <none>        80/TCP    3m48s

kubectl exec -n dev-ns middleware-dev-1 /bin/bash --  nohup python3 -m
http.server 80 --directory /tmp/ &
[2] 81966

kubectl exec -n dev-ns frontend-dev-1 /bin/bash -- ping 10.10.20.2 -c1
PING 10.10.20.2 (10.10.20.2) 56(84) bytes of data.
64 bytes from 10.10.20.2: icmp_seq=1 ttl=64 time=1.84 ms

--- 10.10.20.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.842/1.842/1.842/0.000 ms

kubectl exec -n dev-ns frontend-dev-1 /bin/bash -- curl 10.10.20.2
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--
0deployer-node
100    14  100    14    0     0   3500      0 --:--:-- --:--:-- --:--:--
3500
```

You can see ICMP and http traffic is allowed from father rontend-dev-1 Pod to middleware-dev-1 Pod. Let's apply NetworkPolicy to allow only http traffic from frontend-dev-1 Pod to middleware-dev-1 Pod:

```
cat > inter_vn_np.yaml <<END_OF_SCRIPT
apiVersion: networking.K8s.io/v1
kind: NetworkPolicy
metadata:
  name: frontend-dev
  namespace: dev-ns
spec:
  egress:
  - ports:
```

```
    - port: 80
      protocol: TCP
  podSelector:
    matchLabels:
      app: frontend-dev
  policyTypes:
  - Egress
---
apiVersion: networking.K8s.io/v1
kind: NetworkPolicy
metadata:
  name: middleware-dev
  namespace: dev-ns
spec:
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend-dev
    ports:
    - port: 80
      protocol: TCP
  podSelector:
    matchLabels:
      app: middleware-dev
  policyTypes:
  - Ingress
END_OF_SCRIPT
---
```

Let's apply Network Policy to validate the results:

```
kubectl apply -f inter_vn_np.yaml
networkpolicy.networking.K8s.io/frontend-dev created
networkpolicy.networking.K8s.io/middleware-dev created

kubectl exec -n dev-ns frontend-dev-1 /bin/bash -- ping 10.10.20.2 -c1
PING 10.10.20.2 (10.10.20.2) 56(84) bytes of data.

--- 10.10.20.2 ping statistics ---
```

```
1 packets transmitted, 0 received, 100% packet loss, time 0ms

command terminated with exit code 1

kubectl exec -n dev-ns frontend-dev-1 /bin/bash -- curl 10.10.20.2
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--
0deployer-node
100    14  100    14    0     0   3500      0 --:--:-- --:--:-- --:--:--
3500
```

Contrail Security Policy (CSP)

CN2 already supports Kubernetes Network Policies (KNP) to control flow of traffic to and from Kubernetes workloads to other Kubernetes workloads or IP addresses. Kubernetes Network Policies are scoped to only those corresponding workloads which are referenced in Network Policies rules, imeaning remaining traffic in that name space in unaffected from the applied Network Policies. This approach is more developer centric as developers can use Network Policies to test traffic restriction only to the specific workloads based on Network Policy rules, while not considering if other workloads still have open access. Network Policies are also half-duplex policies meaning each policy only controls traffic ingress/egress from a selected set of pods. For end-to-end traffic flow to work, another Kubernetes policy is required on the remote side.

A cluster or namespace administrator's approach would be that any traffic to any workload in a specific namespace should be restricted and only requirements-based traffic should be allowed. CSP offers this functionality, once a CSP is applied to any endpoints in a namespace with action "Pass," then only that traffic is allowed, and remaining traffic is blocked inside that name space with a CSP default "Deny" action. The administrator needs to configure specific rules for any required ingress or egress traffic in a namespace considering holistic views of traffic allow and deny requirements. Moreover, there is no need to define separate rules for ingress and egress end points. This approach is simpler as compared to KNP where a developer would require defining separate policy rules for ingress and egress end points for a single flow. Another upside for CSP over KNP is that CN2 supports multi-cluster deployments and CSP can be used in multi-cluster deployments, but KNP does not support multi-cluster deployments.

CN2 custom resource Contrail Security Policy (CSP) comprises the following:

- SrcEP: List of source endpoints for a rule match criterion. End points can either be a Pod Selector or IP Block.

- DstEP: List of destination endpoints for a rule match criterion. End points can either be a Pod Selector or an IP Block.
- Ports: List of destination ports to be matched for this rule.
- Action: Action to be taken if any policy rule matches. Currently supported actions are “Pass” or “Deny.

Figure 14 illustrates the order of matching policies for a flow.

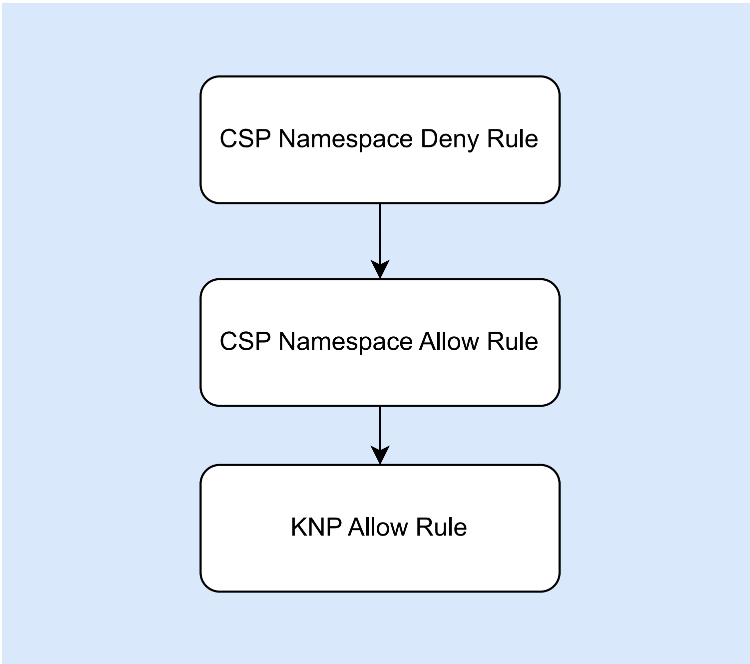


Figure 14 Contrail Security Policy vs K8S Policy Rules Preference

In the next example four Pods will be deployed in a Namespace (ns-svl) and attached to a NAD (sp-vn-svl).

- In absence of any policy, traffic among all Pods is allowed.
- A CSP (allow-hr-to-dev) is defined to allow traffic from the “hr” Pod to “dev” Pod. This policy will also apply implicit deny rules to block any traffic which is not allowed in policy rules.
- Another CSP is defined to allow traffic from the “hr” Pod to “fac” Pod and also from CIDR 174.19.12.11/32 to CIDR 174.19.12.12/32.

```
cat > csp_nad_pod_ns.yaml <<END_OF_SCRIPT
apiVersion: v1
kind: Namespace
```

```
metadata:
  name: ns-svl
  labels:
    ns: ns-svl
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: sp-vn-svl
  namespace: ns-svl
  labels:
    spvn: vn-svl
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "174.19.12.0/27"
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "sp-vn-svl",
  "type": "contrail-K8s-cni"
 }'
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-hr-svl
  namespace: ns-svl
  annotations:
    K8s.v1.cni.cncf.io/networks: '[{"name":"sp-vn-svl","namespace":"ns-
svl","cni-args":null,"ips":["174.19.12.10"],"mac":"de:ad:00:01:ee:eb","in
terface":"intf"}]'
  labels:
    dept: hr
    site: svl
    tier: one
spec:
  containers:
  - name: sp-pod-hr-svl
```

```
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    command: ["bash","-c","while true; do sleep 60s; done"]
    securityContext:
      capabilities:
        add:
          - NET_ADMIN
      privileged: true
  tolerations:
    - key: "key"
      operator: "Equal"
      value: "value"
      effect: "NoSchedule"
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-dev-svl
  namespace: ns-svl
  annotations:
    K8s.v1.cni.cncf.io/networks: '[{"name":"sp-vn-svl","namespace":"ns-svl","cni-args":null,"ips":["174.19.12.11"],"mac":"de:ad:00:02:ee:eb","interface":"intf"}]'
  labels:
    dept: dev
    site: svl
    tier: two
spec:
  containers:
  - name: sp-pod-dev-svl
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    command: ["bash","-c","while true; do sleep 60s; done"]
    securityContext:
      capabilities:
        add:
          - NET_ADMIN
      privileged: true
  tolerations:
    - key: "key"
      operator: "Equal"
```

```
      value: "value"
      effect: "NoSchedule"
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-fin-svl
  namespace: ns-svl
  annotations:
    K8s.v1.cni.cncf.io/networks: '[{"name":"sp-vn-svl","namespace":"ns-svl","cni-args":null,"ips":["174.19.12.12"],"mac":"de:ad:00:03:ee:eb","interface":"intf"}]'
  labels:
    dept: fin
    site: svl
    tier: three
spec:
  containers:
  - name: sp-pod-fin-svl
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    command: ["bash","-c","while true; do sleep 60s; done"]
    securityContext:
      capabilities:
        add:
          - NET_ADMIN
      privileged: true
  tolerations:
    - key: "key"
      operator: "Equal"
      value: "value"
      effect: "NoSchedule"
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-fac-svl
  namespace: ns-svl
  annotations:
    K8s.v1.cni.cncf.io/networks: '[{"name":"sp-vn-svl","namespace":"ns-
```

```
svl","cni-args":null,"ips":["174.19.12.13"],"mac":"de:ad:00:04:ee:eb","in
terface":"intf"}]'
  labels:
    dept: fac
    site: svl
    tier: four
spec:
  containers:
  - name: sp-pod-fac-svl
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    command: ["bash","-c","while true; do sleep 60s; done"]
    securityContext:
      capabilities:
        add:
          - NET_ADMIN
      privileged: true
  tolerations:
    - key: "key"
      operator: "Equal"
      value: "value"
      effect: "NoSchedule"
---
END_OF_SCRIPT
```

CSP Rules Verification

With no CSP applied all traffic is allowed.

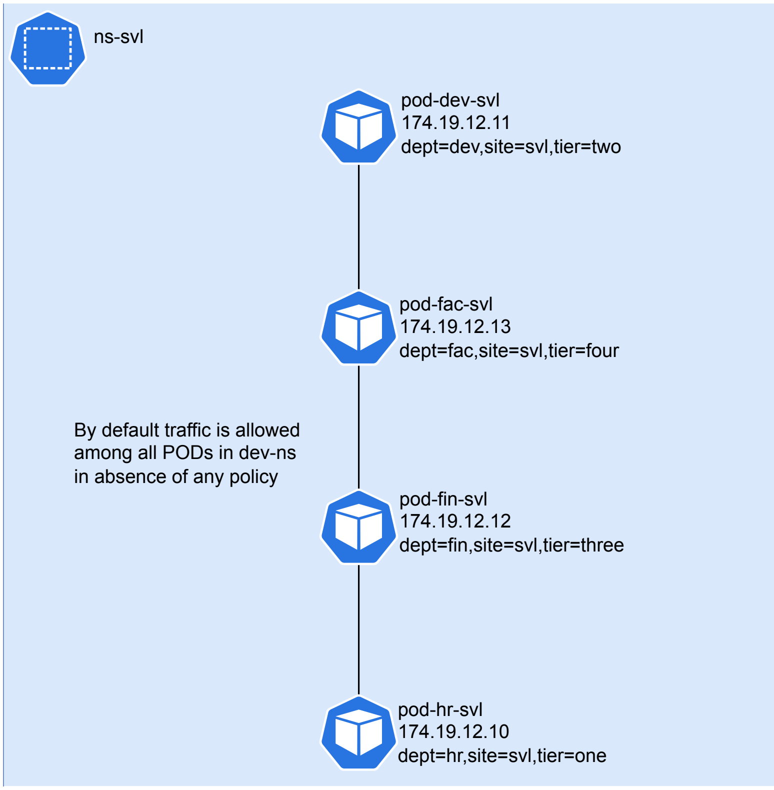


Figure 15 Contrail Security Policy Rules Verification-1

```
kubectl get pods -n ns-svl
NAME           READY    STATUS     RESTARTS    AGE
pod-dev-svl    1/1      Running    0           2m41s
pod-fac-svl    1/1      Running    0           2m41s
pod-fin-svl    1/1      Running    0           2m41s
pod-hr-svl     1/1      Running    0           2m41s

kubectl exec -it -n ns-svl pod-hr-svl --  ping  174.19.12.11 -c1
PING 174.19.12.11 (174.19.12.11) 56(84) bytes of data.
64 bytes from 174.19.12.11: icmp_seq=1 ttl=64 time=1.90 ms

--- 174.19.12.11 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

```
rtt min/avg/max/mdev = 1.897/1.897/1.897/0.000 ms

kubectl exec -it -n ns-svl pod-hr-svl --  ping  174.19.12.12 -c1
PING 174.19.12.12 (174.19.12.12) 56(84) bytes of data.
64 bytes from 174.19.12.12: icmp_seq=1 ttl=64 time=0.810 ms

--- 174.19.12.12 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.810/0.810/0.810/0.000 ms

kubectl exec -it -n ns-svl pod-hr-svl --  ping  174.19.12.13 -c1
PING 174.19.12.13 (174.19.12.13) 56(84) bytes of data.
64 bytes from 174.19.12.13: icmp_seq=1 ttl=64 time=0.777 ms

--- 174.19.12.13 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.777/0.777/0.777/0.000 ms
```

CSP allows traffic from “hr” Pod to “dev” Pod while blocking any other traffic.

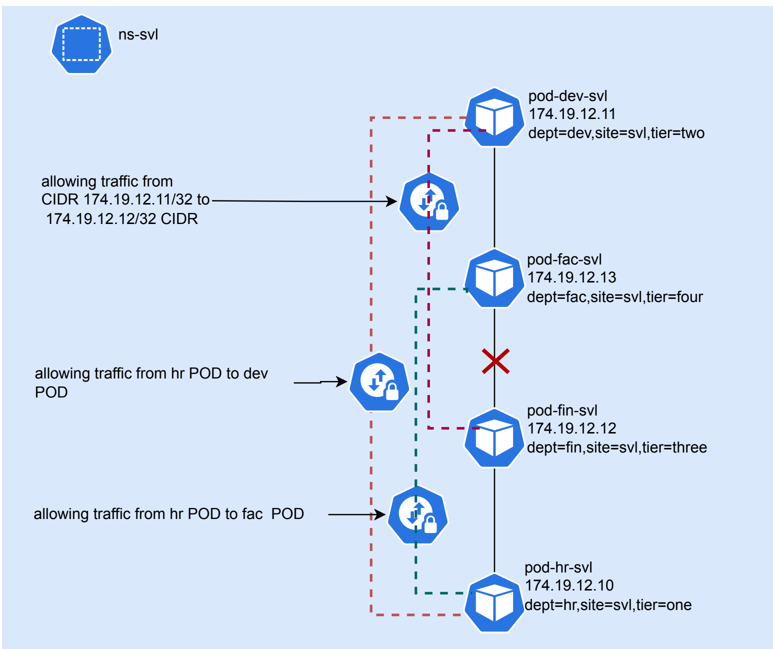


Figure 16 *Contrail Security Policy Rules Verification-2*

```
cat > csp_hr_to_dev.yaml  <<END_OF_SCRIPT
apiVersion: core.contrail.juniper.net/v3
kind: ContrailSecurityPolicy
metadata:
  name: allow-hr-to-dev
  namespace: ns-svl
spec:
  rules:
    - srcEP:
        endPoints:
          - podSelector:
              matchLabels:
                dept: hr

      dstEP:
        endPoints:
          - podSelector:
              matchLabels:
                dept: dev
  action: pass
END_OF_SCRIPT
```

Let's create CSP and verify its functionality:

```
kubectl apply -f csp_hr_to_dev.yaml

kubectl exec -it -n ns-svl pod-hr-svl -- ping 174.19.12.11 -c1
PING 174.19.12.11 (174.19.12.11) 56(84) bytes of data.
64 bytes from 174.19.12.11: icmp_seq=1 ttl=64 time=1.86 ms

--- 174.19.12.11 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.862/1.862/1.862/0.000 ms

kubectl exec -it -n ns-svl pod-hr-svl --  ping  174.19.12.12 -c1
PING 174.19.12.12 (174.19.12.12) 56(84) bytes of data.

--- 174.19.12.12 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

```
command terminated with exit code 1

kubectl exec -it -n ns-svl pod-hr-svl -- ping 174.19.12.13 -c1
PING 174.19.12.13 (174.19.12.13) 56(84) bytes of data.

--- 174.19.12.13 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
command terminated with exit code 1
```

CSP to allow traffic from the "hr" Pod to "fac" Pod and from CIDR 174.19.12.11/32 to CIDR 174.19.12.12/32.

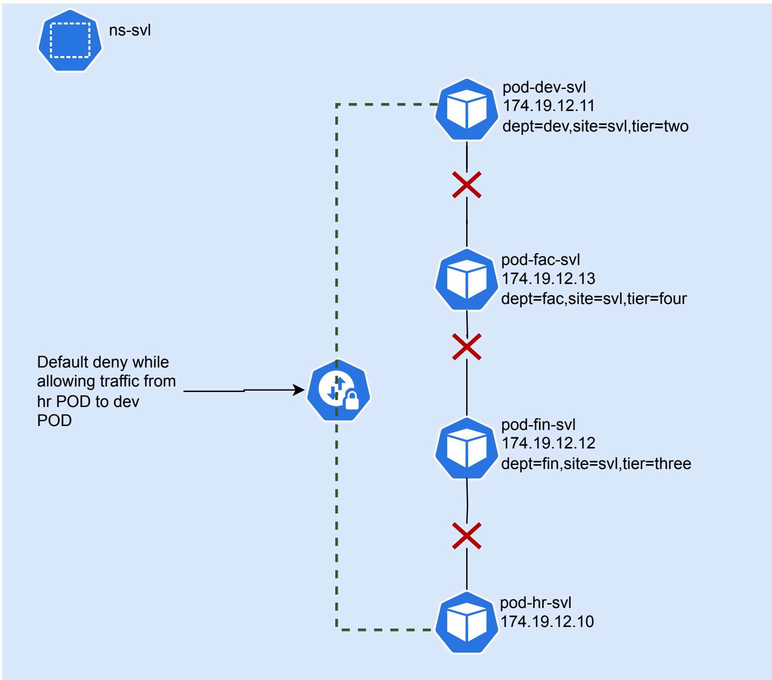


Figure 17 *Contrail Security Policy Rules Verification-4*

```
cat > csp_allow_hr_to_fac_and_11_12.yaml  <<END_OF_SCRIPT
apiVersion: core.contrail.juniper.net/v3
kind: ContrailSecurityPolicy
metadata:
  name: allow-hr-to-fac-or-11-12
  namespace: ns-svl
spec:
  rules:
```

```
    - srcEP:
        endPoints:
          - podSelector:
              matchLabels:
                dept: hr
                tier: one

      dstEP:
        endPoints:
          - podSelector:
              matchLabels:
                dept: fac
    - srcEP:
        endPoints:
          - ipBlock:
              cidr: 174.19.12.11/32

      dstEP:
        endPoints:
          - ipBlock:
              cidr: 174.19.12.12/32

  action: pass
END_OF_SCRIPT

kubectl apply -f csp_allow_hr_to_fac_and_11_12.yaml

kubectl exec -it -n ns-svl pod-hr-svl -- ping 174.19.12.11 -c1
PING 174.19.12.11 (174.19.12.11) 56(84) bytes of data.
64 bytes from 174.19.12.11: icmp_seq=1 ttl=64 time=2.01 ms

--- 174.19.12.11 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.008/2.008/2.008/0.000 ms

kubectl exec -it -n ns-svl pod-hr-svl --  ping  174.19.12.12 -c1
PING 174.19.12.12 (174.19.12.12) 56(84) bytes of data.
```

```
--- 174.19.12.12 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

command terminated with exit code 1

kubectl exec -it -n ns-svl pod-hr-svl -- ping 174.19.12.13 -c1
PING 174.19.12.13 (174.19.12.13) 56(84) bytes of data.
64 bytes from 174.19.12.13: icmp_seq=1 ttl=64 time=0.947 ms

--- 174.19.12.13 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.947/0.947/0.947/0.000 ms

kubectl exec -it -n ns-svl pod-dev-svl -- ping 174.19.12.12 -c1
PING 174.19.12.12 (174.19.12.12) 56(84) bytes of data.
64 bytes from 174.19.12.12: icmp_seq=1 ttl=64 time=1.98 ms

--- 174.19.12.12 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.979/1.979/1.979/0.000 ms
```

Isolated Namespace

By default, all the K8s Pods will be connected with default-podnetwork and communication is allowed between all Pods on default-podnetwork but CN2 offers a feature which is called isolated-namespace which bisects the default-podnetwork and default-servicenetwork for the Pods running on isolated-namespace, thus stopping communication to and from Pods and services running on isolated-namespace from all other Pods and services running in any other isolated-namespace.

The following example creates two namespaces, isolated-ns01 and isolated-ns01. These namespaces will be marked isolated by adding the label core.juniper.net/isolated-namespace:'true'. Pods created inside isolated namespaces cannot communicate to each other.

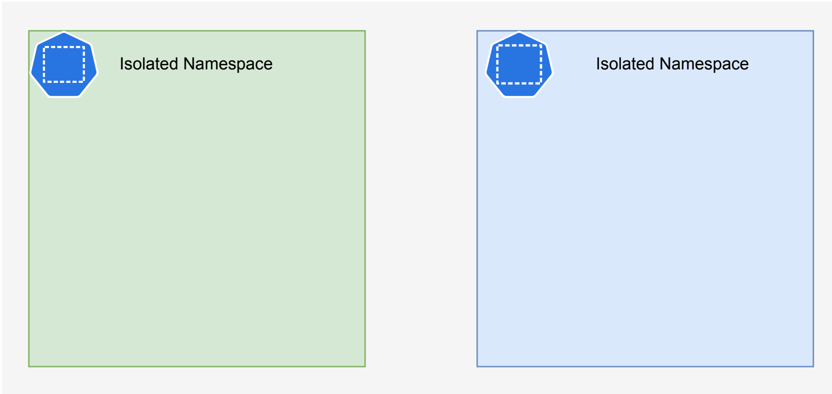


Figure 18 *Isolated Namespace*

```
cat > ns_isolation.yaml <<END_OF_SCRIPT
---
apiVersion: v1
kind: Namespace
metadata:
  labels:
    name: isolated-ns01
    core.juniper.net/isolated-namespace: 'true'
  name: isolated-ns01
---
apiVersion: v1
kind: Pod
metadata:
  name: pod01-ns01
  namespace: isolated-ns01
spec:
  containers:
  - name: pod01-ns01
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
    securityContext:
      privileged: true
  nodeName: worker4
---
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod02-ns01
  namespace: isolated-ns01
spec:
  containers:
  - name: pod02-ns01
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
    securityContext:
      privileged: true
  nodeName: worker5
---
apiVersion: v1
kind: Namespace
metadata:
  labels:
    name: isolated-ns02
    core.juniper.net/isolated-namespace: 'true'
  name: isolated-ns02
---
apiVersion: v1
kind: Pod
metadata:
  name: pod01-ns02
  namespace: isolated-ns02
spec:
  containers:
  - name: pod01-ns02
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
    securityContext:
      privileged: true
  nodeName: worker6
---
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod02-ns02
  namespace: isolated-ns02
spec:
  containers:
  - name: pod02-ns02
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
    securityContext:
      privileged: true
  nodeName: worker6
END_OF_SCRIPT
```

Create Pods in Isolated-Namepsace.

```
kubectl apply  -f ns_isolation.yaml
namespace/isolated-ns01 created
pod/pod01-ns01 created
pod/pod02-ns01 created
namespace/isolated-ns02 created
pod/pod01-ns02 created
pod/pod02-ns02 created


kubectl get all -n isolated-ns01
NAME              READY   STATUS    RESTARTS   AGE
pod/pod01-ns01    1/1     Running   0          36s
pod/pod02-ns01    1/1     Running   0          36s


kubectl get all -n isolated-ns02
NAME              READY   STATUS    RESTARTS   AGE
pod/pod01-ns02    1/1     Running   0          49s
pod/pod02-ns02    1/1     Running   0          49s


kubectl exec -it -n isolated-ns01 pod01-ns01 /bin/bash -- ip addr show
```

```
eth0
588: eth0@if589: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:52:3b:0b:21:c4 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.233.71.21/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::c011:d9ff:fedf:ad20/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec -it -n isolated-ns01 pod02-ns01 /bin/bash -- ip addr show
eth0
3941: eth0@if3942: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP group default
    link/ether 02:4e:25:d4:52:3d brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.233.70.23/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::d890:b5ff:fe0a:13fd/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec -it -n isolated-ns01 pod02-ns01 /bin/bash -- ping
10.233.71.21 -c1
PING 10.233.71.21 (10.233.71.21) 56(84) bytes of data.
64 bytes from 10.233.71.21: icmp_seq=1 ttl=64 time=3.54 ms

--- 10.233.71.21 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.540/3.540/3.540/0.000 ms


kubectl exec -it -n isolated-ns02 pod01-ns02 /bin/bash -- ip addr show
eth0
313: eth0@if314: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:c1:72:9d:1e:8f brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.233.72.32/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::2405:45ff:feae:53b8/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec -it -n isolated-ns02 pod02-ns02 /bin/bash -- ip addr show
eth0
```

```
315: eth0@if316: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:96:e2:68:1c:fb brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.233.72.31/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::dce8:21ff:fe4e:84d4/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec -it -n isolated-ns02 pod02-ns02 /bin/bash -- ping
10.233.72.32 -c1
PING 10.233.72.32 (10.233.72.32) 56(84) bytes of data.
64 bytes from 10.233.72.32: icmp_seq=1 ttl=63 time=1.85 ms

--- 10.233.72.32 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.846/1.846/1.846/0.000 ms

kubectl exec -it -n isolated-ns02 pod02-ns02 /bin/bash -- ping
10.233.71.21 -c1
PING 10.233.71.21 (10.233.71.21) 56(84) bytes of data.

--- 10.233.71.21 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

command terminated with exit code 1
```

BGPaaS

One of CN2's (and its predecessor, Contrail Networking's) significant features aimed at supporting Telco workloads is BGPaaS. This mechanism allows any workload to establish a BGP session to one or two BGP neighbors, handled by the vRouter. By handling the BGP sessions from the workloads at the vRouters, CN2 further limits the configuration required on fabric infrastructure such as the DCGWs, SDN GW, or ToR (Top of Rack) switches, and enables workloads to be deployed onto any K8s node with no impact on the location at which corresponding BGP sessions should be configured. This also dramatically reduces the number of BGP sessions that each DCGW, SDN-GW, or ToR must handle and ensures that any prefixes advertised by a workload within a virtual network are propagated within the VN without any policy required to leak them into the appropriate VRF.

BGPaaS Implementation Details

BGPaaS is deployed in CN2 using two main steps. First, the BGPaaS service is defined with all the associated attributes that we would expect (local ASN, remote ASN, Authentication, Address Families, etc). Next, when a pod is created, an annotation is applied to the pod to indicate on which VNs a BGPaaS session is required. In this way, the BGPaaS session will follow the workload if it has connections on the identified VNs.

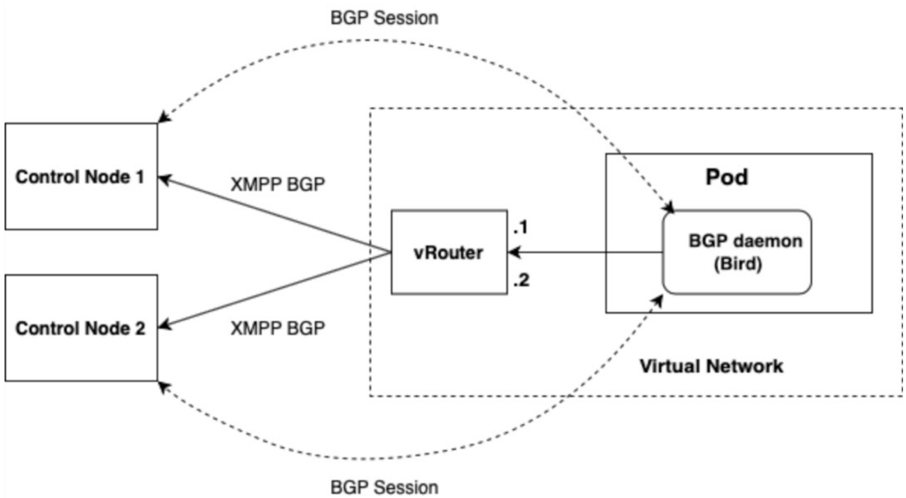


Figure 19 *BGP as a Service*
Courtesy to Juniper Networks for above diagram

Here are some examples of the two BGPaaS steps:

```
apiVersion: core.contrail.juniper.net/v1alpha1
kind: BGPAsAService
metadata:
  namespace: 5gupf
  name: bgpaas-n6
spec:
  shared: false
  autonomousSystem: 64512
  bgpAsAServiceSessionAttributes:
    routeOriginOverride:
      origin: EGP
    addressFamilies:
      family:
```

```
        - inet
        - inet6
  virtualMachineInterfacesSelector:
    - matchLabels:
        core.juniper.net/bgpaasVN: vn-n6-internet
    - matchLabels:
        core.juniper.net/bgpaasVN: vn-n6-local
```

You can see that we created a BGPaaS for a 5G UPF on the N6. Since we may have both a local N6 (for example for IMS or other locally delivered services,) or an Internet N6, there are two labels that we create to match against when selecting the VN on which a workload requires this BGPaaS.

The YAML file is applied using kubectl.

```
kubectl apply -f bgpaas-n6.yaml
```

Now that we have the BGPaaS defined, you can apply it to a 5GNC UPF Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: upf-pod-1
  namespace: 5gupf
  annotations:
    K8s.v1.cni.cncf.io/networks: |
        [{
          "name": "vn-n6-local",
          "namespace": "5gupf",
          "cni-args": null
          "interface": "eth1"
        },{
          "name": "vn-n6-internet",
          "namespace": "5gupf",
          "cni-args": null
          "interface": "eth2"
        },{
          "name": "vn-n4",
          "namespace": "5gsmf",
          "cni-args": null
          "interface": "eth3"
        },{
```

```
          "name": "vn-n3",
          "namespace": "5gran",
          "cni-args": null
          "interface": "eth4"
        }]
    core.juniper.net/bgpaas-networks: vn-n6-local,vn-n6-internet
spec:
  containers:
    - name: upf-pod-1
      image: upf:latest
      ...
```

The above Pod upf-pod-1 will have four interfaces plus the default-pod-network on eth0. Of those interfaces, only the N6 interfaces will have BGPaaS using the bgpaas-n6 definition. However, it is equally possible to create another BGPaaS definition, for N4 for example, and bind that to vn-n4 simply by adding that VN name to the core.juniper.net/bgpaas-networks attribute.

It's worth noting the difference in terminology between the BGPaaS definition, where each VN name to which BGPaaS can be applied is individually named in core.juniper.net/bgpaas VN, while in the Pod definition, the list of networks is assigned using core.juniper.net/bgpaas-networks. A working example of BGPaaS is given in the Chapter 11 section on LTE Traffic Simulation.

Static Routing

Static routing is a preferable method to access stub networks where involving dynamic routing would add complications. CN2 also offers features to configure static routing to provide access to stub networks laying inside K8s workloads. CN2 offers two custom resources to configure static routing: RouteTable and InterfaceRouteTable. The former allows you to configure attributes required for static route for a VN and later allows you to configure static routes over Virtual Machine Interfaces (VMI). A working example of RouteTable is given in the Chapter 11 section on LTE Traffic Simulation.

Chapter 7

Connecting Cloud-native Workloads with Physical World

Chapter 7 discusses various options for extending K8s workloads connectivity to the outer world.

Fabric SNAT & Fabric Forwarding

CN2 offers distributed source NAT (SNAT) and Fabric Forwarding (Forwarding) to allow communication from and to K8s workloads to the outside world. Once Fabric SNAT is enabled on VirtualNetwork, it allows outgoing traffic from a Pod to reach destination outside K8s cluster (by changing the source IP of the Pod with CN2 vRouter interface IP). Fabric Forwarding allows outside traffic to reach a Pod using underlay routing and any overlay without encapsulation.

NodePort Service Type

K8S Node Port type service provides accessibility to K8s services from the outside world by accessing worker nodes' IP addresses. In vanilla K8s deployment, kube-proxy uses netfiliters and IP tables to allow access to implement Node Port type service. CN2 implements NodePort type service in conjunction with the Fabric Forwarding feature described above.

SNAT, FForwarding and NodePort Implementation

In this next example we will create a namespace dev-ns that has a custom-podNetwork "dev-vn". The NAD is defined to implement SNAT and Fabric Forwarding by adding annotations "fabricSNAT: true" and "fabricForwarding: true" to the juniper.net/networks section. A NodPort type service with targetport 7868 is also created which is bound with three Pods (frontend-dev-1, frontend-dev-2, and frontend-dev-3) by matching label "app: frontend-dev". Pod images are defined to run a http server on port 7868 upon a Pods creation.

```
cat > fforward-snat.yaml <<END_OF_SCRIPT
---
apiVersion: v1
kind: Namespace
metadata:
  name: dev-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
```

```
kind: NetworkAttachmentDefinition
metadata:
  name: dev-vn
  namespace: dev-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.10.10.0/24",
      "podNetwork": true,
      "fabricSNAT": true,
      "fabricForwarding": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "dev-vn",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-1
  namespace: dev-ns
  labels:
    app: frontend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: frontend-dev-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      imagePullPolicy: IfNotPresent
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker4
```

```
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-2
  namespace: dev-ns
  labels:
    app: frontend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: frontend-dev-2
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      imagePullPolicy: IfNotPresent
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker5
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-3
  namespace: dev-ns
  labels:
    app: frontend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: frontend-dev-3
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      imagePullPolicy: IfNotPresent
      securityContext:
        privileged: true
```

```
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker6
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-dev
  namespace: dev-ns
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  ports:
  - name: port-80
    targetPort: 7868
    protocol: TCP
    port: 80
  selector:
    app: frontend-dev
  type: NodePort
---
END_OF_SCRIPT
```

Create NAD, Pods, NodePort Service:

```
kubectl apply -f fforward-snat.yaml
namespace/dev-ns created
networkattachmentdefinition.K8s.cni.cncf.io/dev-vn created
pod/frontend-dev-1 created
pod/frontend-dev-2 created
pod/frontend-dev-3 created
service/frontend-dev created


kubectl get pods -n dev-ns -o wide
NAME             READY   STATUS    RESTARTS   AGE    IP            NODE
NOMINATED NODE   READINESS GATES
frontend-dev-1   1/1     Running   0          9m9s   10.10.10.4    worker4
<none>           <none>
```

```
frontend-dev-2   1/1     Running   0          9m9s   10.10.10.2   worker5
<none>           <none>

frontend-dev-3   1/1     Running   0          9m9s   10.10.10.3   worker6
<none>           <none>


kubectl get svc -n dev-ns
NAME           TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
frontend-dev   NodePort   10.233.45.71   <none>        80:30087/TCP   10m
```

To test fabric forwarding functionality and NodePort type service, first identify a host which has connectivity with K8s worker nodes (in our case it is deployer-node VM) and access the NodePort service by using worker nodes IP/ DNS entry where the target Pods are running.

```
curl worker4:30087
deployer-node

curl worker5:30087
deployer-node

curl worker6:30087
deployer-node
```

To test SNAT functionality identify a host which has connectivity with K8s worker nodes (in our case it is deployer-node with IP address 192.168.24.82) and access that host from the Pods:

```
ip -4 -br a
lo               UNKNOWN        127.0.0.1/8
ens3             UP             192.168.24.82/24
docker0          UP             172.17.0.1/16


kubectl exec -it -n dev-ns frontend-dev-1 /bin/bash -- ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
612: eth0@if613: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
```

```
    link/ether 02:37:b6:6b:08:b9 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.10.10.4/24 brd 10.10.10.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::74f8:64ff:febd:3125/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec -it -n dev-ns frontend-dev-2 /bin/bash -- ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3951: eth0@if3952: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP group default
    link/ether 02:2e:b9:e8:e6:7d brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.10.10.2/24 brd 10.10.10.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::70c3:e4ff:fecf:cb04/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec -it -n dev-ns frontend-dev-3 /bin/bash -- ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
325: eth0@if326: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:bf:8d:bd:79:01 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.10.10.3/24 brd 10.10.10.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::d49c:85ff:fe80:9a0f/64 scope link
       valid_lft forever preferred_lft forever
ubuntu@deployer-node:~/deployments/fabric_forw

kubectl exec -it -n dev-ns frontend-dev-1 /bin/bash -- ping 192.168.24.82
```

```
-c1
PING 192.168.24.82 (192.168.24.82) 56(84) bytes of data.
64 bytes from 192.168.24.82: icmp_seq=1 ttl=62 time=1.97 ms

kubectl exec -it -n dev-ns frontend-dev-2 /bin/bash -- ping 192.168.24.82
-c1
PING 192.168.24.82 (192.168.24.82) 56(84) bytes of data.
64 bytes from 192.168.24.82: icmp_seq=1 ttl=62 time=2.28 ms

--- 192.168.24.82 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.281/2.281/2.281/0.000 ms

kubectl exec -it -n dev-ns frontend-dev-3 /bin/bash -- ping 192.168.24.82
-c1
PING 192.168.24.82 (192.168.24.82) 56(84) bytes of data.
64 bytes from 192.168.24.82: icmp_seq=1 ttl=62 time=1.81 ms

--- 192.168.24.82 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.811/1.811/1.811/0.000 ms
```

Software Defined Gateway (SDN-GW) Design Considerations

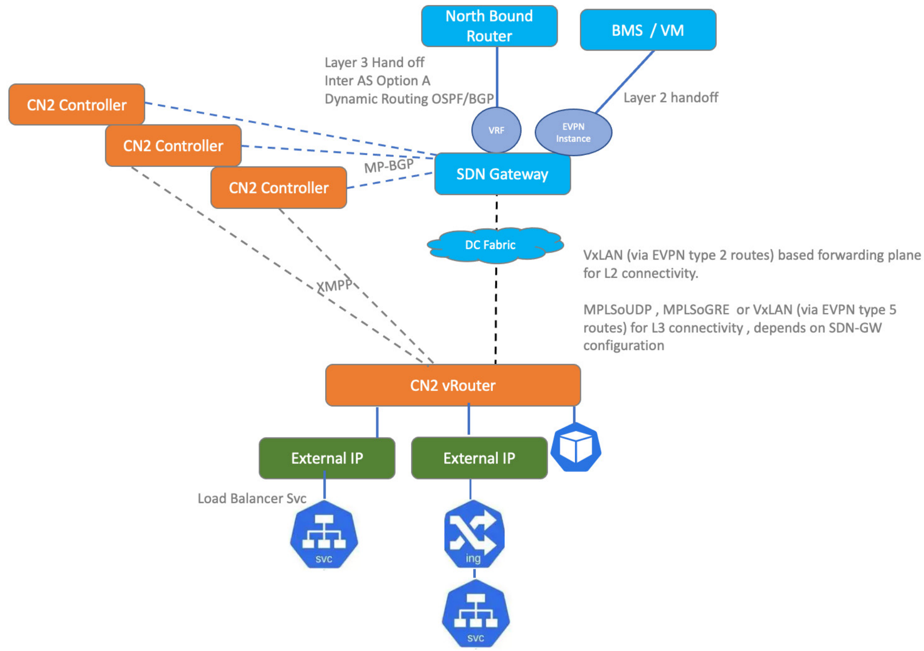


Figure 20 *Software Defined Gateway Router Design Consideration*

The Juniper CN2 SDN Controller runs a MB-BGP-based control plane and this feature provides an advantage to CN2 over other cloud native SDN Controllers by enabling the capability to extend Layer 2 and Layer 3 connectivity from and to cloud native workloads from or to the outer world. You can configure MP-BGP sessions between CN2 controllers and any router from any vendor which supports MP-BGP, and that router will be called a Software Defined Gateway (SDN GW). CN2 supports various MP-BGP extended families (for example, inet, inet-vpn, route-target, inet6-vpn, and e-vpn) to exchange a Control Plane updated with SDN GW. Once the Control Plane is established then you can use MPLSoGRE, VxLAN or MPLSoUDP tunnels for forwarding plane traffic between SDN GW and CN2 routers. In this book we have used a Juniper Networks virtualized MX (vMX, Junos 20.2R3.9) router for implementation of SDN GW but in real world scenarios a physical router would be required.

Any vendor router can be selected for SDN GW functionality which supports MP-BGP based control planes and any forwarding plane technology such as MPLSoGRE, VxLAN, or MPLSoUDP tunnels. We are using only one SDN GW in this book but in real world scenarios it is recommended to use two SDN GWs for high availability and resiliency.

CN2 VN has a default forwarding mode of L2/L3 which implies that we can extend L2 connectivity or L3 connectivity for a VN with the outer world via SDN-GW. VxLAN (via EVPN Type 2 routes) will be the default forwarding plane for L2 connectivity between cloud-native workloads and the outer world via SDN-GW. For L3 connectivity between cloud-native workloads and othe uter world via SDN-GW, a forwarding plane can be MPLSoUDP, MPLSoGRE, or VxLAN (via EVPN type 5 routes) and it depends on SDN-GW configuration.

In our experience MPLSoUDP and VxLAN-based forwarding plane mechanisms are equally good once it comes to packet hashing if CN2 vRouter is multi-homed to multiple switches for high availability. A MPLSoUDP-based forwarding plane is widely used in commercial deployments with the Juniper Contrail Networking SDN Controller. Here we are using a MPLSoUDP-based forwarding plane for L3 connectivity and a VxLAN based forwarding plane (with EVPN type 2 routes) for L2 connectivity with via SDN-GW.

Required routing should be in place between the SDN GW loopback IP and CN2 Controller nodes vRouter IP address, otherwise MP-BGP sessions between SDN GW and CN2 Controllers would not be established. Similarly, SDN GW loopback IP and CN2 worker nodes vRouter IP addresses should be reachable to each other else overlay tunnels would not be established between SDN GW and CN2 vRouter.

Okay, let's add routes in K8S Master and worker nodes to reach SDN-GW loopback IPs via vhost0 interface:

```
for node in {1..3};do ssh  controller$node sudo ip r add 172.172.172.172
dev vhost0 via 192.168.5.1;done


for node in {1..3};do ssh  controller$node sudo ping 172.172.172.172 -I
vhost0  -c1;done

PING 172.172.172.172 (172.172.172.172) from 192.168.5.51 vhost0: 56(84)
bytes of data.

64 bytes from 172.172.172.172: icmp_seq=1 ttl=64 time=3.23 ms


--- 172.172.172.172 ping statistics ---

1 packets transmitted, 1 received, 0% packet loss, time 0ms

rtt min/avg/max/mdev = 3.230/3.230/3.230/0.000 ms

PING 172.172.172.172 (172.172.172.172) from 192.168.5.52 vhost0: 56(84)
bytes of data.

64 bytes from 172.172.172.172: icmp_seq=1 ttl=64 time=1.86 ms


--- 172.172.172.172 ping statistics ---

1 packets transmitted, 1 received, 0% packet loss, time 0ms

rtt min/avg/max/mdev = 1.862/1.862/1.862/0.000 ms
```

```
PING 172.172.172.172 (172.172.172.172) from 192.168.5.53 vhost0: 56(84)
bytes of data.
64 bytes from 172.172.172.172: icmp_seq=1 ttl=64 time=2.78 ms

--- 172.172.172.172 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.781/2.781/2.781/0.000 ms

for node in {1..3};do ssh  worker$node sudo ip r add 172.172.172.172 dev
vhost0 via 192.168.5.1;done

for node in {1..6};do ssh  worker$node sudo ping 172.172.172.172 -I
vhost0  -c1;done
PING 172.172.172.172 (172.172.172.172) from 192.168.5.54 vhost0: 56(84)
bytes of data.
64 bytes from 172.172.172.172: icmp_seq=1 ttl=64 time=103 ms

--- 172.172.172.172 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 102.785/102.785/102.785/0.000 ms
PING 172.172.172.172 (172.172.172.172) from 192.168.5.55 vhost0: 56(84)
bytes of data.
64 bytes from 172.172.172.172: icmp_seq=1 ttl=64 time=39.2 ms

--- 172.172.172.172 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 39.208/39.208/39.208/0.000 ms
PING 172.172.172.172 (172.172.172.172) from 192.168.5.56 vhost0: 56(84)
bytes of data.
64 bytes from 172.172.172.172: icmp_seq=1 ttl=64 time=474 ms
```

The next config snippet shows that MP-iBGP is configured under protocols bgp group contrail_gw, where local-address 172.172.172.172 is the SDN-GW loopback IP and neighbor are CN2 master nodes vRouter IP address. In the MP-iBGP config we have configured address families (inet unicast, inet-vpn unicast, route-target, and evpn signaling). MPLSoUDP dynamic tunnels configuration is also present where source address is SDN-GW loopback IP and destination subnet is CN2 vRouter IP address pool. A special community "0x030c:64512:13" is applied while advertising MP-iBGP routes to CN2. This community is required to establish MPLSoUDP dynamic tunnels between SDN-GW and CN2 vRouters. We are also setting SDN-GW as the route reflector and adding statement "no-client-reflect" to ensure that routes received from one CN2 controller should not reflected to another CN2 controller because CN2 controllers already have MP-IBGP sessions with each other.

If we have two or SDN-GW, then either we can share the same cluster ID for among all route reflectors or each reflector could have its own cluster ID. In either case, if each route reflector is configured with a different cluster, then we need to configure MP-iBGP sessions between route reflectors as well.

```
set chassis fpc 0 pic 0 tunnel-services
set chassis network-services enhanced-ip
set protocols bgp group contrail_gw type internal
set protocols bgp group contrail_gw local-address 172.172.172.172
set protocols bgp group contrail_gw hold-time 90
set protocols bgp group contrail_gw keep all
set protocols bgp group contrail_gw log-updown
set protocols bgp group contrail_gw family inet unicast
set protocols bgp group contrail_gw family inet-vpn unicast
set protocols bgp group contrail_gw family evpn signaling
set protocols bgp group contrail_gw family route-target
set protocols bgp group contrail_gw export mpls_over_udp
set protocols bgp group contrail_gw local-as 64512
set protocols bgp group contrail_gw multipath
set protocols bgp group contrail_gw neighbor 192.168.5.51 peer-as 64512
set protocols bgp group contrail_gw neighbor 192.168.5.52 peer-as 64512
set protocols bgp group contrail_gw neighbor 192.168.5.53 peer-as 64512
set protocols bgp group contrail_gw vpn-apply-export

set protocols bgp group contrail_gw cluster 172.172.172.172
set protocols bgp group contrail_gw no-client-reflect

set policy-options policy-statement mpls_over_udp term 1 then community
add udp
set policy-options policy-statement mpls_over_udp term 1 then accept
set policy-options community udp members 0x030c:64512:13

set routing-options dynamic-tunnels to-CN2 source-address 172.172.172.172
set routing-options dynamic-tunnels to-CN2 udp
set routing-options dynamic-tunnels to-CN2 destination-networks
192.168.5.0/24

set interfaces lo0 unit 0 family inet address 172.172.172.172/32
```

In the next example we are using CN2 BGPRouter CRD to configure BGP neighborship between CN2 Controllers and SDN-GW along with MP-iBGP families (inet, inet-vpn, route-target , inet6-vpn and e-vpn):

```
cat > bgprouter.yaml <<END_OF_SCRIPT
apiVersion:  core.contrail.juniper.net/v1alpha1
kind: BGPRouter
metadata:
  namespace: contrail
  name: sdngw
  labels:
    dcgw: vmx
spec:
  parent:
    apiVersion: core.contrail.juniper.net/v1
    kind:       RoutingInstance
    name:       default
    namespace:  contrail
  bgpRouterParameters:
    vendor: Juniper
    routerType: router
    address:  172.172.172.172
    identifier:  172.172.172.172
    holdTime: 60
    addressFamilies:
      family:
        - inet
        - inet-vpn
        - route-target
        - inet6-vpn
        - e-vpn
    autonomousSystem:  64512
END_OF_SCRIPT
```

Apply configuration on CN2 and verify status:

```
kubectl apply -f bgprouter.yaml
kubectl get bgprouters -n contrail
NAME          TYPE            IDENTIFIER     STATE     AGE
controller1   control-node    192.168.5.51   Success   20h
```

```
controller2   control-node   192.168.5.52   Success   20h
controller3   control-node   192.168.5.53   Success   20h
sdngw         router         192.168.5.14   Success   59m
```

MP-iBGP and MPLSoUDP dynamic tunnels status verification from SDN-GW:

```
show bgp summary
Threading mode: BGP I/O
Groups: 1 Peers: 3 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State
Pending
bgp.rtarget.0
                      79         79          0          0          0
0
inet.0
                      12         12          0          0          0
0
bgp.l3vpn.0
                       2          2          0          0          0
0
bgp.evpn.0
                       0          0          0          0          0
0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last
Up/Dwn State|#Active/Received/Accepted/Damped...
192.168.5.51          64512        200        209       0       0
1:01:14 Establ
  bgp.rtarget.0: 33/33/33/0
  inet.0: 4/4/4/0
  bgp.l3vpn.0: 1/1/1/0
  mesh_vnr_2101.inet.0: 1/1/1/0
192.168.5.52          64512        200        209       0       0
1:01:14 Establ
  bgp.rtarget.0: 33/33/33/0
  inet.0: 4/4/4/0
  bgp.l3vpn.0: 1/1/1/0
  mesh_vnr_2101.inet.0: 0/1/1/0
192.168.5.53          64512        193        208       0       0
1:01:14 Establ
  bgp.rtarget.0: 13/13/13/0
  inet.0: 4/4/4/0
  bgp.l3vpn.0: 0/0/0/0
```

```
show dynamic-tunnels database terse
*- Signal Tunnels #- PFE-down
Table: inet.3

Destination-network: 192.168.5.0/24
Destination                        Source            Next-hop
Type        Status
192.168.5.54/32                    172.172.172.172    0xd6e86bc nhid 625
UDP         Up
192.168.5.54/32                    172.172.172.172    0xd6e89dc nhid 623
UDP         Up
192.168.5.55/32                    172.172.172.172    0xd6e8914 nhid 621
UDP         Up
192.168.5.52/32                    172.172.172.172    0xd6e820c nhid 619
UDP         Up
192.168.5.53/32                    172.172.172.172    0xd6e82d4 nhid 618
UDP         Up
192.168.5.56/32                    172.172.172.172    0xd6e8aa4 nhid 622
UDP         Up
192.168.5.51/32                    172.172.172.172    0xd6e8144 nhid 620
UDP         Up
```

Extending Layer 3 Connectivity to SDN GW

CN2 VN can be extended to SDN-GW by configuring a VRF on SDN-GW with matched BGP RouteTarget value configured on CN2 VN. We can extend CN2 VN connectivity further from SDN GW towards a northbound router or service layer device.

In the following example we will configure a VN on CN2 with target:64562:2101 and a corresponding VRF will be configured on the SDN GW using same route target. Once the CN2 Pod host IP is available on the SDN GW the we have multiple options for northbound connectivity.

- Inter AS Option A (i.e., VRF to VRF connectivity via dynamic routing and IP based forwarding). The northbound router can send default routes or any other prefixes towards the corresponding VRF on SDN-GW and that route can be further re-distributed towards CN2 controllers and thus Pods would have reachability information toward north bound destinations via SDN-GW. This is the most widely used approach by Contrail Classic customers.
- Inter AS Option B, where no VRF will be configured on the SDN-GW and the SDN-GW can simply have MP-BGP based control plane and MPLS-based forwarding towards north bound router. We have never seen this approach followed by any customer of Contrail Classic.

- Another option is to do route leaking between VRF and the master routing table in the SDN-GW and all outbound traffic from VRF can traverse to destinations routing information available in the master routing table. But this could break multi-tenancy if route leaking between VRFs and master routing table is not done carefully. Careful planning is required while assigning the route target to CN2 VNs as overlapping route target to multiple VNs can break multi-tenancy isolation.
- In this book we have used Inter AS Option A (with OSPF as dynamic routing protocol toward the northbound router).

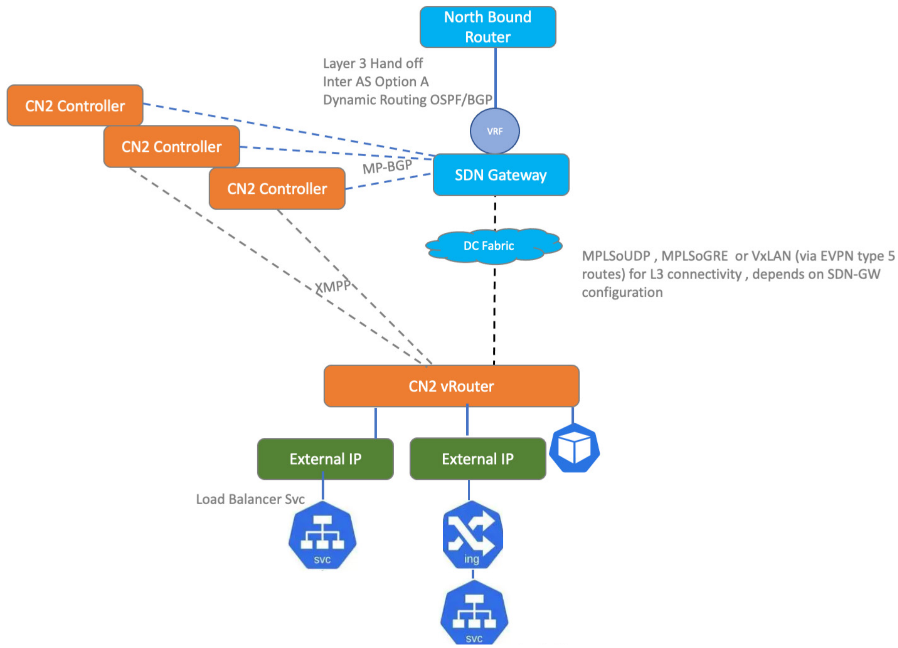


Figure 21 Layer 3 Connectivity to External World via SDN Gateway Router

In the following example we are configuring a VRF in the SDN GW (Juniper vMX in our case). RouteTarget (target:64562:2101) is assigned to this VRF and it should match with CN2 NAD RT which is required to be extended to the SDN-GW. Inside the VRF we have configured OSPF on a physical interface which is connected with the northbound router and CN2 Pods and host IPs are further advertised to the northbound router after aggregating CN2 Pods subnets.

```
set routing-instances l3_use_case routing-options aggregate route 
10.20.1.0/24

set routing-instances l3_use_case routing-options aggregate route 
172.30.130.0/29

set routing-instances l3_use_case routing-options aggregate route 
10.30.1.0/24

set routing-instances l3_use_case protocols ospf area 0.0.0.0 interface
```

```
xe-0/0/2.0 interface-type p2p
set routing-instances l3_use_case protocols ospf export to_internet_
router
set routing-instances l3_use_case instance-type vrf
set routing-instances l3_use_case interface xe-0/0/2.0
set routing-instances l3_use_case route-distinguisher 192.168.5.14:2101
set routing-instances l3_use_case vrf-target target:64562:2101
set routing-instances l3_use_case vrf-table-label

set policy-options policy-statement to_internet_router term 1 from
protocol aggregate
set policy-options policy-statement to_internet_router term 1 from
route-filter 10.20.1.0/24 exact
set policy-options policy-statement to_internet_router term 1 then accept
set policy-options policy-statement to_internet_router term 2 from
protocol aggregate
set policy-options policy-statement to_internet_router term 2 from
route-filter 172.30.130.0/29 exact
set policy-options policy-statement to_internet_router term 2 then accept
set policy-options policy-statement to_internet_router term 3 from
protocol aggregate
set policy-options policy-statement to_internet_router term 3 from
route-filter 10.30.1.0/24 exact
set policy-options policy-statement to_internet_router term 3 then accept
set policy-options policy-statement to_internet_router term 4 from
protocol bgp
set policy-options policy-statement to_internet_router term 4 from
route-filter 100.100.100.100/32 exact
set policy-options policy-statement to_internet_router term 4 then accept
set policy-options policy-statement to_internet_router term else then
reject
```

Now let's create a NAD with RT (target:64562:2101) assigned to it and then a Pod is attached to this NAD. The Pod IP and host IP will be learned in the SDN-GW via MP-iBGP on the VRF routing table which has a matching RT (target:64562:2101):

```
cat > sdngw_l3_use_case.yaml <<END_OF_SCRIPT
apiVersion: v1
kind: Namespace
metadata:
  labels:
    ns: sdngw-ns
  name: sdngw-ns
```

```
---
apiVersion: “K8s.cni.cncf.io/v1”
kind: NetworkAttachmentDefinition
metadata:
  name: vn1
  labels:
    vn: vn1
  namespace: sdngw-ns
  annotations:
    juniper.net/networks: ‘{
      “ipamV4Subnet”: “10.20.1.0/24”,
      “routeTargetList”: [“target:64562:2101”],
      “podNetwork”: true
    }’
spec:
  config: ‘{
  “cniVersion”: “0.3.1”,
  “name”: “vn1”,
  “type”: “contrail-K8s-cni”
}’
---
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  namespace: sdngw-ns
  annotations:
    net.juniper.contrail.podnetwork: sdngw-ns/vn1
spec:
  containers:
  - name: pod1
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    securityContext:
      privileged: true
      capabilities:
        add:
          - NET_ADMIN
END_OF_SCRIPT
```

Create NAD, Pod, and verify its details:

```
kubectl apply -f sdngw_l3_use_case.yaml
namespace/sdngw-ns created
networkattachmentdefinition.K8s.cni.cncf.io/vn1 created
pod/pod1 created

kubectl get pods -n sdngw-ns -o wide
NAME   READY   STATUS    RESTARTS   AGE   IP          NODE      NOMINATED
NODE   READINESS GATES
pod1   1/1     Running   0          11m   10.20.1.2   worker6   <none>
<none>


ssh worker6 ip addr show vhost0
8: vhost0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc fq_codel
state UNKNOWN group default qlen 1000
    link/ether 52:54:00:26:96:11 brd ff:ff:ff:ff:ff:ff
    inet 192.168.5.59/24 brd 192.168.5.255 scope global vhost0
       valid_lft forever preferred_lft forever
    inet6 fe80::5054:ff:fe26:9611/64 scope link
       valid_lft forever preferred_lft forever


kubectl exec -it -n sdngw-ns pod1 /bin/bash -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
341: eth0@if342: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:1a:13:2a:ac:c9 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.20.1.2/24 brd 10.20.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::8012:37ff:feac:2f8a/64 scope link
       valid_lft forever preferred_lft forever
```

Let's confirm MP-iBGP control plane functionality between the CN2 controllers and the SDG-GW. You can see that the Pod host IP (10.20.1.2/32) is learned in the bgp.l3vpn.0 routing table and further installed in the l3_use_case.inet.0 routing table:

```
show route 10.20.1.2/32


l3_use_case.inet.0: 8 destinations, 9 routes (6 active, 0 holddown, 2
hidden)
+ = Active Route, - = Last Active, * = Both


10.20.1.2/32       *[BGP/170] 00:03:41, MED 100, localpref 200, from
192.168.5.52
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.59), Push 24
                    [BGP/170] 00:03:41, MED 100, localpref 200, from
192.168.5.53
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.59), Push 24


bgp.l3vpn.0: 1 destinations, 2 routes (1 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both


192.168.5.59:2:10.20.1.2/32
                   *[BGP/170] 00:03:41, MED 100, localpref 200, from
192.168.5.52
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.59), Push 24
                    [BGP/170] 00:03:41, MED 100, localpref 200, from
192.168.5.53
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.59), Push 24
```

Let's verify forwarding plane functionality between the CN2 worker node (vRouter) and the SDN-GW. You can see that the Pod host IP (10.20.1.2/32) is installed in the forwarding table with comp (composite) next hop index 625. In dynamic-tunnel database (MPLSoUDP tunnels) you can see that the comp next hop label 625 is assigned to 192.168.5.59/32 which is vRouter IP address of worker6 where the Pod (10.20.1.2/32) is instantiated:

```
show route forwarding-table destination 10.20.1.2 table l3_use_case
```

```
Routing table: l3_use_case.inet
Internet:
Destination        Type RtRef Next hop           Type Index    NhRef
Netif
10.20.1.2/32       user     0                    indr  1048574     2
                                                 comp      625     2
show dynamic-tunnels database terse
*- Signal Tunnels #- PFE-down
Table: inet.3

Destination-network: 192.168.5.0/24
Destination                      Source          Next-hop
Type       Status
192.168.5.51/32                  172.172.172.172 0xd6e86bc nhid 632
UDP        Up
192.168.5.53/32                  172.172.172.172 0xd6e852c nhid 634
UDP        Up
192.168.5.57/32                  172.172.172.172 0xd6e8464 nhid 635
UDP        Up
192.168.5.52/32                  172.172.172.172 0xd6e8c98 nhid 636
UDP        Up
192.168.5.59/32                  172.172.172.172 0xd6e7c30 nhid 625
UDP        Up
192.168.5.59/32                  172.172.172.172 0xd6e8c34 nhid 637
UDP        Up
192.168.5.58/32                  172.172.172.172 0xd6e9210 nhid 638
UDP        Up
```

Test connectivity from an external host which has reachability to the SDN-GW VRF and then further to the CN2 Pods:

```
root@control-host:~# ip r | grep 10.20.1.0/24
10.20.1.0/24 via 192.168.201.1 dev br-siov-201
root@control-host:~# ip -4 -br a | grep 192.168.201
br-siov-201      UP             192.168.201.253/24
root@control-host:~# ping 10.20.1.2 -I br-siov-201

root@control-host:~# ping 10.20.1.2 -I br-siov-201 -c1
PING 10.20.1.2 (10.20.1.2) from 192.168.201.253 br-siov-201: 56(84) bytes
of data.
64 bytes from 10.20.1.2: icmp_seq=1 ttl=62 time=4.80 ms

--- 10.20.1.2 ping statistics ---
```

```
1 packets transmitted, 1 received, 0% packet loss, time 0msExtending
Layer 2 Connectivity to SDN GW
```

As described CN2 supports EVPN (Ethernet Virtual Private Network) and we can use that to extend Layer 2 connectivity between the outer destinations and CN2 Pods. While creating a Virtual Network in CN2 we need to add virtualNetworkNetworkID (VNID) property. It should match VxLAN ID configured on the SDN GW. We also need to make VxLAN as the first encapsulation priority in the default-global-vrouter-config CRD. Layer 2 connectivity for CN2 workload and outer destinations should be the same subnet but IP addresses should not overlap. The best practice is to assign manual IP addresses to the Pod and outer destination to avoid IP address duplication. Careful planning is also required while assigning the VNID to CN2 VNs as overlapping VNIDs to multiple VNs can break multi tenancy isolation.

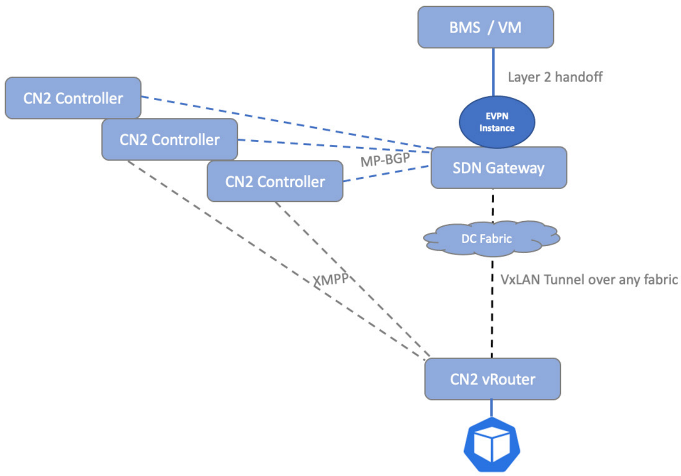


Figure 22 Layer 2 Connectivity to External World via SDN Gateway Router

Set VxLAN as 1st encapsulation priority:

```
cat > encap.yaml <<END_OF_SCRIPT
spec:
  encapsulationPriorities:
    encapsulation:
    - VXLAN
    - MPLSoGRE
    - MPLSoUDP
END_OF_SCRIPT
```

Patching encapsulation priority to default-global-vrouter-config CRD:

```
kubectl patch gvc default-global-vrouter-config  --patch-file encap.yaml
```

```
globalvrouterconfig.core.contrail.juniper.net/default-global-vrouter-config
patched
```

Now let's create a NAD with forwardingMode set to l2 with virtualNetworkNetworkID (10001) and a route target (target:64562:1001) assigned to it. A Pod is created with an additional interface "eth1" which has statically configured the MAC and IP addresses:

```
cat > sdngw_l2_use_case.yaml <<END_OF_SCRIPT
apiVersion: v1
kind: Namespace
metadata:
  name: sdngw-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: evpnl2l3
  namespace: sdngw-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "192.168.200.0/24",
      "virtualNetworkNetworkID": 10001,
      "forwardingMode": "l2",
      "routeTargetList": ["target:64562:1001"]
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "evpnl2l3",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: test-pod1
  namespace: sdngw-ns
  annotations:
    K8s.v1.cni.cncf.io/networks: '[{"name":"evpnl2l3","namespace":"sdn
gw-ns","cni-args":null,"ips":["192.168.200.200"],"mac":"de:ab:10:ff:ff:ff
```

```
","interface":"eth1"}]'
spec:
  containers:
  - name: test-pod1
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    securityContext:
      privileged: true
      capabilities:
        add:
          - NET_ADMIN
END_OF_SCRIPT
```

Next we are configuring an EVPN instance which has VxLAN 10001 and RT target:64562:1001 (matching to the CN2 NAD). A bridge domain is also configured, and the physical interface connected with a bare metal server is assigned to this bridge-domain:

```
set routing-instances l2_use_case protocols evpn extended-vni-list all
set routing-instances l2_use_case protocols evpn encapsulation vxlan
set routing-instances l2_use_case protocols evpn multicast-mode ingress-replication
set routing-instances l2_use_case vtep-source-interface lo0.0
set routing-instances l2_use_case instance-type virtual-switch
set routing-instances l2_use_case bridge-domains bd_1001 vlan-id 1001
set routing-instances l2_use_case bridge-domains bd_1001 interface xe-0/0/1.0
set routing-instances l2_use_case bridge-domains bd_1001 vxlan vni 10001
set routing-instances l2_use_case bridge-domains bd_1001 vxlan ingress-node-replication
set routing-instances l2_use_case route-distinguisher 172.172.172.172:1001
set routing-instances l2_use_case vrf-target target:64562:1001
set interfaces xe-0/0/1 encapsulation ethernet-bridge
set interfaces xe-0/0/1 unit 0 family bridge
```

Create NAD and Pod and verify the details:

```
kubectl apply -f sdngw_l2_use_case.yaml
namespace/sdngw-ns created
networkattachmentdefinition.K8s.cni.cncf.io/evpnl2l3 created
pod/test-pod1 created

kubectl get pods -n sdngw-ns -o wide
```

```
NAME         READY   STATUS    RESTARTS   AGE    IP             NODE
NOMINATED NODE   READINESS GATES

test-pod1    1/1     Running    0          102s   10.233.71.22   worker4
<none>           <none>
```

```
kubectl exec -it -n sdngw-ns test-pod1 /bin/bash -- ip addr show eth1

650: eth1@if651: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default

    link/ether de:ab:10:ff:ff:ff brd ff:ff:ff:ff:ff:ff link-netnsid 0

    inet 192.168.200.200/24 brd 192.168.200.255 scope global eth1

       valid_lft forever preferred_lft forever

    inet6 fe80::54d5:53ff:fe79:971e/64 scope link

       valid_lft forever preferred_lft forever


ssh worker4 ip addr show vhost0

8: vhost0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc fq_codel
state UNKNOWN group default qlen 1000

    link/ether 52:54:00:bf:70:fe brd ff:ff:ff:ff:ff:ff

    inet 192.168.5.57/24 brd 192.168.5.255 scope global vhost0

       valid_lft forever preferred_lft forever

    inet6 fe80::5054:ff:febf:70fe/64 scope link

       valid_lft forever preferred_lft forever

test-pod1 interface eth1 has mac address 'de:ab:10:ff:ff:ff' and IP
address "192.168.200.200".
```

Let's verify control plane functionality between the SDN-GW and CN2 controllers. We will verify if the MAC address 'de:ab:10:ff:ff:ff' is learned in the MP-BGP control plane by looking into the bgp.evpn.0 routing table. The next step would be to verify if the learned MAC address is installed in the EVPN data base and then from theEVPN data base it should be installed in the bridge mac table inside the EVPN instance:

```
show route table bgp.evpn.0 evpn-mac-address de:ab:10:ff:ff:ff


bgp.evpn.0: 8 destinations, 11 routes (8 active, 0 holddown, 0 hidden)

+ = Active Route, - = Last Active, * = Both


2:192.168.5.57:2::10001::de:ab:10:ff:ff:ff/304 MAC/IP

                   *[BGP/170] 00:03:45, MED 100, localpref 200, from
192.168.5.51

                      AS path: ?, validation-state: unverified

                    >  via Tunnel Composite, UDP (src 172.172.172.172
```

```
dest 192.168.5.57)
                    [BGP/170] 00:03:45, MED 100, localpref 200, from
192.168.5.52
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.57)
2:192.168.5.57:2::10001::de:ab:10:ff:ff:ff::192.168.200.200/304 MAC/IP
                   *[BGP/170] 00:03:45, MED 100, localpref 200, from
192.168.5.51
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.57)
                    [BGP/170] 00:03:45, MED 100, localpref 200, from
192.168.5.52
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.57)

show evpn database mac-address de:ab:10:ff:ff:ff
Instance: l2_use_case
VLAN  DomainId  MAC address        Active source
Timestamp        IP address
     10001      de:ab:10:ff:ff:ff  192.168.5.57                   Apr 30
18:52:01  192.168.200.200



show bridge mac-table instance l2_use_case


MAC flags       (S -static MAC, D -dynamic MAC, L -locally learned, C
-Control MAC
    O -OVSDB MAC, SE -Statistics enabled, NM -Non configured MAC, R
-Remote PE MAC, P -Pinned MAC)


Routing instance : l2_use_case
 Bridging domain : bd_1001, VLAN : 1001
   MAC                 MAC      Logical                Active
   address             flags    interface              source
   22:70:29:5c:b9:cf   D        xe-0/0/1.0
   ac:4b:c8:2b:77:c1   D        xe-0/0/1.0
   de:ab:10:ff:ff:ff   D        vtep.32769             192.168.5.57
```

Let's verify forwarding plane functionality between the SDN-GW and CN2 worker4 (vRouter IP address 192.168.5.57) where the Pod with MAC address 'de:ab:10:ff:ff:ff' is instantiated:

```
show interfaces vtep terse
Interface               Admin Link Proto    Local                 Remote
vtep                    up    up
vtep.32768              up    up
vtep.32769              up    up   bridge

show interfaces vtep.32769
  Logical interface vtep.32769 (Index 347) (SNMP ifIndex 563)
    Flags: Up SNMP-Traps Encapsulation: ENET2
    VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.57, L2 
Routing Instance: l2_use_case, L3 Routing Instance: default
    Input packets : 4
    Output packets: 0
    Protocol bridge, MTU: Unlimited
      Flags: Trunk-Mode
```

The Pod has L2 connectivity to an outer destination via SDN-GW and we can also see thar vtep.32769 interface input/ output packets are also incrementing:

```
kubectl exec -it -n sdngw-ns test-pod1 /bin/bash -- ping 192.168.200.253 
-c100
PING 192.168.200.253 (192.168.200.253) 56(84) bytes of data.
64 bytes from 192.168.200.253: icmp_seq=1 ttl=64 time=3.66 ms
64 bytes from 192.168.200.253: icmp_seq=2 ttl=64 time=3.09 ms
64 bytes from 192.168.200.253: icmp_seq=3 ttl=64 time=3.40 ms

--- 192.168.200.253 ping statistics ---
100 packets transmitted, 100 received, 0% packet loss, time 99159ms
rtt min/avg/max/mdev = 2.144/3.289/6.992/0.788 ms

show interfaces vtep.32769
  Logical interface vtep.32769 (Index 347) (SNMP ifIndex 563)
    Flags: Up SNMP-Traps Encapsulation: ENET2
    VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.57, L2 
Routing Instance: l2_use_case, L3 Routing Instance: default
    Input packets : 30
    Output packets: 34
    Protocol bridge, MTU: Unlimited
```

```
      Flags: Trunk-Mode

show interfaces vtep.32769
  Logical interface vtep.32769 (Index 347) (SNMP ifIndex 563)
    Flags: Up SNMP-Traps Encapsulation: ENET2
    VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.57 L2
Routing Instance: l2_use_case, L3 Routing Instance: default
    Input packets : 74
    Output packets: 78
    Protocol bridge, MTU: Unlimited
      Flags: Trunk-ModeExtending Layer 2 (via VxLAN) & L3 Connectivity
(via MPLSoUDP) to SDN GW
```

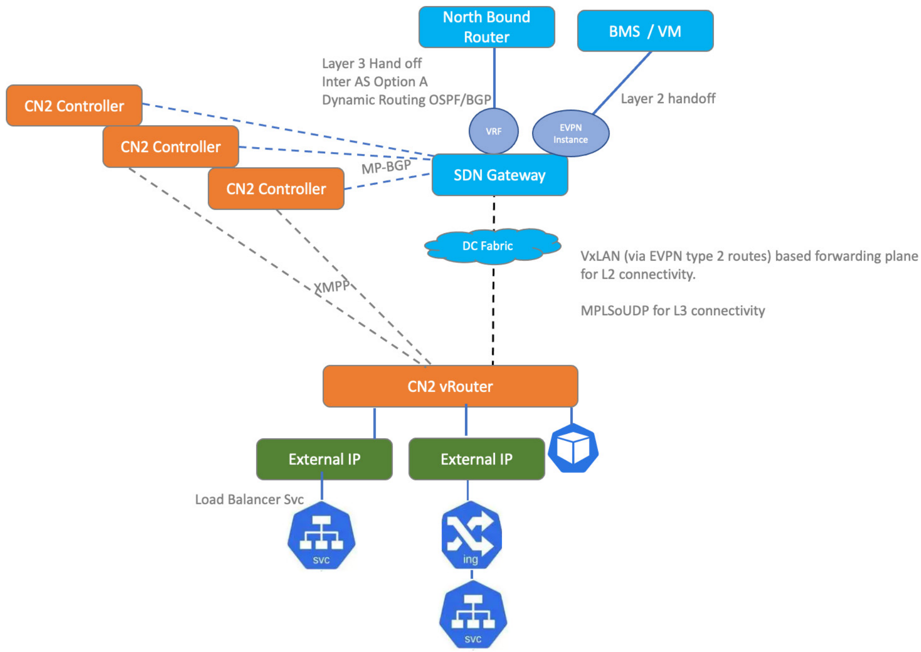


Figure 23 External Connectivity – L2 via VxLAN and L3 via MPLSoUDP

Now let's cover how to extend L2 and L3 connectivity for a cloud-native workload to the outer world via the SDN-GW. As described in the previous SDN-GW Design Consideration, the VxLAN (via EVPN type 2 routes) forwarding plane will be used for L2 connectivity and L3 connectivity. The forwarding plane could be MPLSoUDP, MPLSoGRE, or VxLAN (via EVPN Type 5 routes) and it depends on the SDN-GW configuration. In this example we will use MPLSoUDP for L3 connectivity.

In the following example, we are creating a NAD ("podNetwork": true) replacing default-PodNetwork) with default forwardingMode, therefore, L2L3, virtualNetworkNetworkID (10001), and routeTarget (target:64562:1001) assigned to it. A Pod is created with its primary interface "eth0" connected to a custom-PodNetwork.

```
cat > sdngw_l2_l3_use_case.yaml <<END_OF_SCRIPT
apiVersion: v1
kind: Namespace
metadata:
labels:
ns: sdngw-ns
name: sdngw-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
name: vn1
labels:
vn: vn1
namespace: sdngw-ns
annotations:
juniper.net/networks: '{
"ipamV4Subnet": "192.168.200.0/24",
"virtualNetworkNetworkID": 10001,
"routeTargetList": ["target:64562:2101"],
"podNetwork": true
}'
spec:
config: '{
"cniVersion": "0.3.1",
"name": "vn1",
"type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
name: pod1
```

```
namespace: sdngw-ns
annotations:
net.juniper.contrail.podnetwork: sdngw-ns/vn1
spec:
containers:
- name: pod1
image: deployer-node.maas:5000/ubuntu-traffic:latest
securityContext:
privileged: true
capabilities:
add:
- NET_ADMIN
END_OF_SCRIPT
```

Next we are configuring an EVPN instance which has VxLAN 10001 and RT target:64562:1001 (matching to CN2 NAD). A bridge domain is also configured and a physical interface connected with a bare metal server is assigned to this bridge-domain.

```
set routing-instances l2_use_case protocols evpn extended-vni-list all
set routing-instances l2_use_case protocols evpn encapsulation vxlan
set routing-instances l2_use_case protocols evpn multicast-mode ingress-replication
set routing-instances l2_use_case vtep-source-interface lo0.0
set routing-instances l2_use_case instance-type virtual-switch
set routing-instances l2_use_case bridge-domains bd_1001 vlan-id 1001
set routing-instances l2_use_case bridge-domains bd_1001 interface xe-0/0/1.0
set routing-instances l2_use_case bridge-domains bd_1001 vxlan vni 10001
set routing-instances l2_use_case bridge-domains bd_1001 vxlan ingress-node-replication
set routing-instances l2_use_case route-distinguisher 172.172.172.172:1001
set routing-instances l2_use_case vrf-target target:64562:1001
```

Next we are configuring a VRF in the SDN GW (a Juniper vMX in our case). A RouteTarget (target:64562:1001) is assigned to this VRF and it should match with the CN2 NAD RT which is required to be extended to the SDN-GW. Inside VRF we have configured OSPF on a physical interface which is connected with the northbound router and the CN2 Pods and host IPs are further advertised to the northbound router.

```
set routing-instances l3_use_case protocols ospf area 0.0.0.0 interface xe-0/0/2.0 interface-type p2p
```

```
set routing-instances l3_use_case protocols ospf export to_internet_
router
set routing-instances l3_use_case instance-type vrf
set routing-instances l3_use_case interface xe-0/0/2.0
set routing-instances l3_use_case route-distinguisher 192.168.5.14:2101
set routing-instances l3_use_case vrf-target target:64562:1001
set routing-instances l3_use_case vrf-table-label
set policy-options policy-statement to_internet_router term 5 from
route-filter 192.168.200.0/24 orlonger
set policy-options policy-statement to_internet_router term 5 then accept
```

Let's create NAD and Pod:

```
kubectl apply -f sdngw_l2_l3_use_case.yaml
namespace/sdngw-ns created
networkattachmentdefinition.K8s.cni.cncf.io/vn1 created
pod/pod1 created

kubectl get pods -n sdngw-ns -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
pod1 1/1 Running 0 8m58s 192.168.200.2 worker5 <none> <none>

ssh worker5 ip addr show vhost0
8: vhost0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc fq_codel
state UNKNOWN group default qlen 1000
link/ether 52:54:00:ce:d0:62 brd ff:ff:ff:ff:ff:ff
inet 192.168.5.58/24 brd 192.168.5.255 scope global vhost0
valid_lft forever preferred_lft forever
inet6 fe80::5054:ff:fece:d062/64 scope link
valid_lft forever preferred_lft forever
kubectl exec -it -n sdngw-ns pod1 /bin/bash -- ip addr show eth0
30: eth0@if31: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
link/ether 02:97:4c:0b:30:b4 brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet 192.168.200.2/24 brd 192.168.200.255 scope global eth0
valid_lft forever preferred_lft forever
inet6 fe80::28a4:48ff:fe19:de3d/64 scope link
valid_lft forever preferred_lft forever
```

Let's verify L2 control-plane functionality between the SDN-GW and CN2 controllers. We will verify if the MAC address '02:97:4c:0b:30:b4' is learned in MP-BGP control plane by looking into bgp.evpn.0 routing table. The next step would be to verify if the learned MAC address is installed in the EPVN database and then from EVPN database it should be installed in the bridge MAC table inside the EVPN instance:

```
show route table bgp.evpn.0 evpn-mac-address 02:97:4c:0b:30:b4


bgp.evpn.0: 8 destinations, 11 routes (8 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both


2:192.168.5.58:6::10001::02:97:4c:0b:30:b4/304 MAC/IP
*[BGP/170] 00:00:12, MED 100, localpref 200, from 192.168.5.51
AS path: ?, validation-state: unverified
> via Tunnel Composite, UDP (src 172.172.172.172 dest 192.168.5.58)
[BGP/170] 00:00:12, MED 100, localpref 200, from 192.168.5.52
AS path: ?, validation-state: unverified
> via Tunnel Composite, UDP (src 172.172.172.172 dest 192.168.5.58)
2:192.168.5.58:6::10001::02:97:4c:0b:30:b4::192.168.200.2/304 MAC/IP
*[BGP/170] 00:00:12, MED 100, localpref 200, from 192.168.5.51
AS path: ?, validation-state: unverified
> via Tunnel Composite, UDP (src 172.172.172.172 dest 192.168.5.58)
[BGP/170] 00:00:12, MED 100, localpref 200, from 192.168.5.52
AS path: ?, validation-state: unverified
> via Tunnel Composite, UDP (src 172.172.172.172 dest 192.168.5.58)


show evpn database mac-address 02:97:4c:0b:30:b4
Instance: l2_use_case
VLAN DomainId MAC address Active source Timestamp IP address
10001 02:97:4c:0b:30:b4 192.168.5.58 May 06 20:01:41 192.168.200.2


show bridge mac-table instance l2_use_case


MAC flags (S -static MAC, D -dynamic MAC, L -locally learned, C -Control
MAC
O -OVSDB MAC, SE -Statistics enabled, NM -Non configured MAC, R -Remote PE
MAC, P -Pinned MAC)
```

```
Routing instance : l2_use_case
Bridging domain : bd_1001, VLAN : 1001
MAC MAC Logical Active
address flags interface source
02:97:4c:0b:30:b4 D vtep.32769 192.168.5.58
fe:54:00:24:37:7d D xe-0/0/1.0
```

Let's verify forwarding plane functionality between SDN-GW and CN2 worker5 (vRouter IP address 192.168.5.58) where the Pod with MAC address 'de:ab:10:ff:ff:ff' is instantiated:

```
show route forwarding-table destination 192.168.200.2 table l3_use_case
Routing table: l3_use_case.inet
Internet:
Destination Type RtRef Next hop Type Index NhRef Netif
192.168.200.2/32 user 0 indr 1048580 2
comp 617 2
192.168.200.2/32 dest 0 2:97:4c:b:30:b4 ucst 634 1 vtep.32769

contrail@DCG> show interfaces vtep.32769
Logical interface vtep.32769 (Index 347) (SNMP ifIndex 563)
Flags: Up SNMP-Traps Encapsulation: ENET2
VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.58, L2
Routing Instance: l2_use_case, L3 Routing Instance: default
Input packets : 0
Output packets: 0
Protocol bridge, MTU: Unlimited
Flags: Trunk-Mode
```

The Pod has L2 connectivity to an outer destination via SDN-GW and we can also see that vtep.32769 interface input/ output packets are also incrementing.

```
kubectl exec -it -n sdngw-ns pod1 /bin/bash -- ping 192.168.200.253 -c100
PING 192.168.200.253 (192.168.200.253) 56(84) bytes of data.
64 bytes from 192.168.200.253: icmp_seq=1 ttl=64 time=3.66 ms
64 bytes from 192.168.200.253: icmp_seq=2 ttl=64 time=3.09 ms
64 bytes from 192.168.200.253: icmp_seq=3 ttl=64 time=3.40 ms


---- 192.168.200.253 ping statistics ---
```

```
100 packets transmitted, 100 received, 0% packet loss, time 99153ms
rtt min/avg/max/mdev = 1.837/3.116/6.741/1.058 ms



show interfaces vtep.32769
Logical interface vtep.32769 (Index 347) (SNMP ifIndex 563)
Flags: Up SNMP-Traps Encapsulation: ENET2
VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.58, L2
Routing Instance: l2_use_case, L3 Routing Instance: default
Input packets : 0
Output packets: 0
Protocol bridge, MTU: Unlimited
Flags: Trunk-Mode



show interfaces vtep.32769
Logical interface vtep.32769 (Index 347) (SNMP ifIndex 563)
Flags: Up SNMP-Traps Encapsulation: ENET2
VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.58, L2
Routing Instance: l2_use_case, L3 Routing Instance: default
Input packets : 18
Output packets: 17
Protocol bridge, MTU: Unlimited
Flags: Trunk-Mode

show interfaces vtep.32769
Logical interface vtep.32769 (Index 347) (SNMP ifIndex 563)
Flags: Up SNMP-Traps Encapsulation: ENET2
VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.58, L2
Routing Instance: l2_use_case, L3 Routing Instance: default
Input packets : 110
Output packets: 109
Protocol bridge, MTU: Unlimited
Flags: Trunk-Mode
```

Let's verify L3 control plane functionality between the CN2 Controllers and SDN-GW. You can see that the Pod host IP (192.168.200.2/32) is learned in bgp.l3vpn.0 routing table and further installed in l3_use_case.inet.0 routing table:

```
show route 192.168.200.2/32
l3_use_case.inet.0: 11 destinations, 13 routes (8 active, 0 holddown, 3
```

```
hidden)
+ = Active Route, - = Last Active, * = Both


192.168.200.2/32 *[BGP/170] 00:11:25, MED 100, localpref 200, from
192.168.5.51
AS path: ?, validation-state: unverified
> via Tunnel Composite, UDP (src 172.172.172.172 dest 192.168.5.58), Push
64
[BGP/170] 00:11:25, MED 100, localpref 200, from 192.168.5.52
AS path: ?, validation-state: unverified
> via Tunnel Composite, UDP (src 172.172.172.172 dest 192.168.5.58), Push
64


bgp.l3vpn.0: 1 destinations, 2 routes (1 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both


192.168.5.58:6:192.168.200.2/32
*[BGP/170] 00:11:25, MED 100, localpref 200, from 192.168.5.51
AS path: ?, validation-state: unverified
> via Tunnel Composite, UDP (src 172.172.172.172 dest 192.168.5.58), Push
64
[BGP/170] 00:11:25, MED 100, localpref 200, from 192.168.5.52
AS path: ?, validation-state: unverified
> via Tunnel Composite, UDP (src 172.172.172.172 dest 192.168.5.58), Push
64
```

Let's verify the forwarding plane functionality between the CN2 worker node (vRouter) and the SDN-GW. You can see that the Pod host IP (192.168.200.2/32) is installed in the forwarding table with comp (composite) next hop index 625. In the dynamic-tunnel database (MPLSoUDP tunnels) you can see that comp next hop label 617 is assigned to 192.168.5.58/32 which is the vRouter IP address of worker5 where the Pod (192.168.200.2/32) is instantiated:

```
show route forwarding-table destination 192.168.200.2 table l3_use_case
Routing table: l3_use_case.inet
Internet:
Destination Type RtRef Next hop Type Index NhRef Netif
192.168.200.2/32 user 0 indr 1048580 2
comp 617 2
```

```
192.168.200.2/32 dest 0 2:97:4c:b:30:b4 ucst 634 1 vtep.32769

show dynamic-tunnels database terse
*- Signal Tunnels #- PFE-down
Table: inet.3


Destination-network: 192.168.5.0/24
Destination Source Next-hop Type Status
192.168.5.51/32 172.172.172.172 0xd6e82d4 nhid 625 UDP Up
192.168.5.58/32 172.172.172.172 0xd6e8a40 nhid 630 UDP Up
192.168.5.58/32 172.172.172.172 0xd6e8cfc nhid 617 UDP Up
192.168.5.53/32 172.172.172.172 0xd6e839c nhid 624 UDP Up
192.168.5.52/32 172.172.172.172 0xd6e8464 nhid 623 UDP Up
192.168.5.57/32 172.172.172.172 0xd6e8b08 nhid 629 UDP Up
192.168.5.59/32 172.172.172.172 0xd6e820c nhid 626 UDP Up
```

Now we test connectivity from an external host which has reachability to SDN-GW VRF and then further to the CN2 Pods:

```
ubuntu@deployer-node:~$ ip -4 -br a
lo UNKNOWN 127.0.0.1/8
ens3 UP 192.168.24.82/24
docker0 UP 172.17.0.1/16

rubuntu@deployer-node:~$ ping 192.168.200.2 -c100
PING 192.168.200.2 (192.168.200.2) 56(84) bytes of data.
64 bytes from 192.168.200.2: icmp_seq=1 ttl=61 time=4.39 ms

--- 192.168.200.2 ping statistics ---
100 packets transmitted, 100 received, 0% packet loss, time 99160ms
rtt min/avg/max/mdev = 2.340/3.482/9.615/1.102 ms


show dynamic-tunnels database statistics tunnel-type udp
*- Signal Tunnels #- PFE-down
Table: inet.3


Destination-network: 192.168.5.0/24
```

```
Destination Packets Bytes Info
192.168.5.51/32 0 0 UDP
192.168.5.58/32 0 0 UDP
192.168.5.58/32 0 0 UDP, Push 64
192.168.5.53/32 0 0 UDP
192.168.5.52/32 0 0 UDP
192.168.5.57/32 0 0 UDP
192.168.5.59/32 0 0 UDP

show dynamic-tunnels database statistics tunnel-type udp
*- Signal Tunnels #- PFE-down
Table: inet.3

Destination-network: 192.168.5.0/24
Destination Packets Bytes Info
192.168.5.51/32 0 0 UDP
192.168.5.58/32 0 0 UDP
192.168.5.58/32 42 3528 UDP, Push 64
192.168.5.53/32 0 0 UDP
192.168.5.52/32 0 0 UDP
192.168.5.57/32 0 0 UDP
192.168.5.59/32 0 0 UDP

show dynamic-tunnels database statistics tunnel-type udp
*- Signal Tunnels #- PFE-down
Table: inet.3

Destination-network: 192.168.5.0/24
Destination Packets Bytes Info
192.168.5.51/32 0 0 UDP
192.168.5.58/32 0 0 UDP
192.168.5.58/32 47 3948 UDP, Push 64
192.168.5.53/32 0 0 UDP
192.168.5.52/32 0 0 UDP
192.168.5.57/32 0 0 UDP
192.168.5.59/32 0 0 UDP
```

You can see that statistics are increasing while the ICMP ping requests are traversing via MPLSoUDP tunnel.

Extending L2 and L3 Connectivity to SDN-GW via Pure EVPN Routes

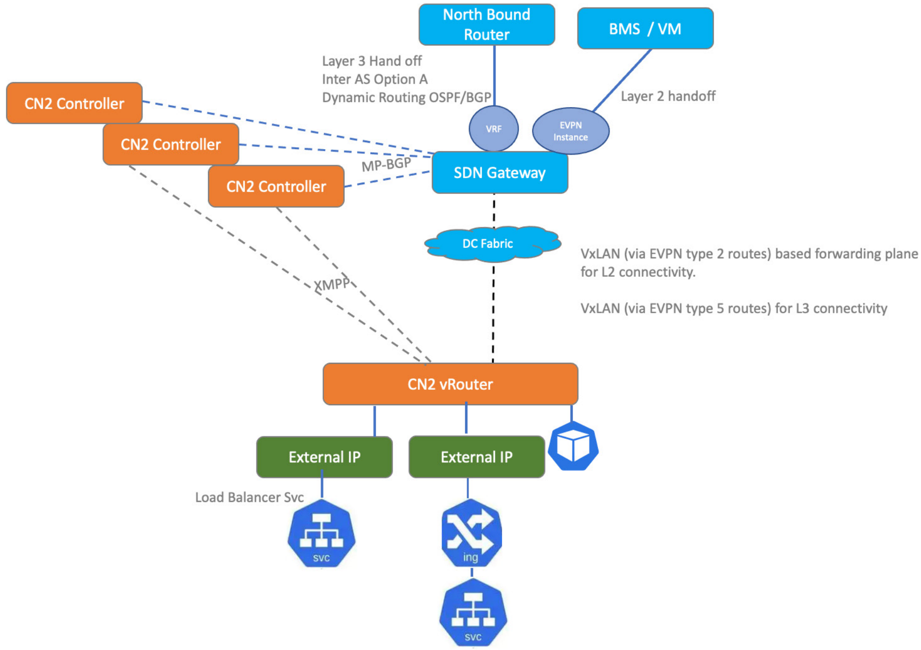


Figure 24 *External L2 and L3 Connectivity via pure EVPN Routes (type 2 & type 5)*

In this example we will show how to extend L2 and L3 connectivity for cloud-native workloads to the outer world via pure EVPN routes. To archive this use case the following configuration changes are made on the SDN-GW.

To ensure control plane exchanges between the SDN-GW and the CN2 controller are only via EVPN, we have removed family inet-vpn unicast under protocols bgp configuration.

To ensure forwarding plane between the SDN-GW and the CN2 vRouters is only based on VxLAN, the export policy mpls_over_udp configuration is removed from protocols bgp:

```
set protocols bgp group contrail_gw type internal
set protocols bgp group contrail_gw local-address 172.172.172.172
set protocols bgp group contrail_gw hold-time 90
set protocols bgp group contrail_gw keep all
set protocols bgp group contrail_gw log-updown
set protocols bgp group contrail_gw family inet unicast
set protocols bgp group contrail_gw family evpn signaling
set protocols bgp group contrail_gw family route-target
```

```
set protocols bgp group contrail_gw local-as 64512
set protocols bgp group contrail_gw multipath
set protocols bgp group contrail_gw neighbor 192.168.5.51 peer-as 64512
set protocols bgp group contrail_gw neighbor 192.168.5.52 peer-as 64512
set protocols bgp group contrail_gw neighbor 192.168.5.53 peer-as 64512
set protocols bgp group contrail_gw vpn-apply-export

set protocols bgp group contrail_gw cluster 172.172.172.172
set protocols bgp group contrail_gw no-client-reflect
```

An IRB (integrated routing and bridging interface) interface with Anycast gateway IP address is configured, which will act as default gateway for the CN2 VN:

```
set interfaces irb gratuitous-arp-reply
set interfaces irb unit 200 proxy-macip-advertisement
set interfaces irb unit 200 virtual-gateway-accept-data
set interfaces irb unit 200 family inet address 192.168.200.250/24
virtual-gateway-address 192.168.200.254
set interfaces irb unit 200 virtual-gateway-v4-mac 00:a0:a0:a0:00:a0
```

The IRB (irb.200) interface is bound with corresponding bridge domain and due to this configuration the CN2 workloadARP entries will be learned on the SDN-GW. However, those ARP entries need to be converted to host routes so that those host routes can be advertised to the internet router (northbound router). To achieve this requirement, we have added the additional knob "remote-ip-host-routes" under protocols evpn:

```
set routing-instances l2_use_case protocols evpn extended-vni-list all
set routing-instances l2_use_case protocols evpn remote-ip-host-routes
set routing-instances l2_use_case protocols evpn encapsulation vxlan
set routing-instances l2_use_case protocols evpn multicast-mode ingress-
replication
set routing-instances l2_use_case vtep-source-interface lo0.0
set routing-instances l2_use_case instance-type virtual-switch
set routing-instances l2_use_case bridge-domains bd_1001 vlan-id 1001
set routing-instances l2_use_case bridge-domains bd_1001 interface
xe-0/0/1.0
set routing-instances l2_use_case bridge-domains bd_1001 routing-
interface irb.200
set routing-instances l2_use_case bridge-domains bd_1001 vxlan vni 10001
set routing-instances l2_use_case bridge-domains bd_1001 vxlan ingress-
node-replication
set routing-instances l2_use_case route-distinguisher
172.172.172.172:1001
```

```
set routing-instances l2_use_case vrf-target target:64562:1001
```

A new VRF with EVPN type 5 route is configured. TheVRF configuration is typical to VRF configuration in Junos, but to support EVPN type 5 routes, additional configuration is added under protocols evpn:

```
set routing-instances evpn_5 protocols ospf area 0.0.0.0 interface
xe-0/0/2.0 interface-type p2p

set routing-instances evpn_5 protocols ospf export to_internet_router

set routing-instances evpn_5 protocols evpn ip-prefix-routes advertise
direct-nexthop

set routing-instances evpn_5 protocols evpn ip-prefix-routes encapsulation
vxlan

set routing-instances evpn_5 protocols evpn ip-prefix-routes vni 1001

set routing-instances evpn_5 instance-type vrf

set routing-instances evpn_5 interface xe-0/0/2.0

set routing-instances evpn_5 interface irb.200

set routing-instances evpn_5 route-distinguisher 172.172.172.172:1

set routing-instances evpn_5 vrf-target target:64562:1001

set routing-instances evpn_5 vrf-table-label
```

Hence, dynamic-tunnel configuration was removed from the SDN-GW, and it introduced a gap. The inet.3 routing table in the SDN-GW lost the routes toward the vRouter IP addresses and the forwarding plane functionality for L3 connectivity will not work in the absence of routes in the inet.3 routing table. To fill this gap, a static route in the rib inet.3 is added for the vRouter subnet pointing to inet.0 routing-table:

```
set routing-options rib inet.3 static route 192.168.5.0/24 next-table
inet.0
```

The policy statement to export EVPN learned routes to internet-router:

```
set policy-options policy-statement to_internet_router term 5 from
protocol evpn

set policy-options policy-statement to_internet_router term 5 from
route-filter 192.168.200.0/24 orlonger

set policy-options policy-statement to_internet_router term 5 then accept
```

On the CN2 side, the Pod and NAD configuration is the same as per previous example, except that adding a default route pointing to SDN-GW IRB interface anycast IP address:

```
cat > sdngw_pure_evpn_use_case.yaml <<END_OF_SCRIPT


apiVersion: v1

kind: Namespace
```

```
metadata:
  labels:
    ns: sdngw-ns
  name: sdngw-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vn1
  labels:
    vn: vn1
  namespace: sdngw-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "192.168.200.0/24",
      "virtualNetworkNetworkID": 10001,
      "routeTargetList": ["target:64562:1001"],
      "podNetwork": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "vn1",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  namespace: sdngw-ns
  annotations:
    net.juniper.contrail.podnetwork: sdngw-ns/vn1
spec:
  containers:
  - name: pod1
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    securityContext:
```

```
      privileged: true
      capabilities:
        add:
          - NET_ADMIN
END_OF_SCRIPT
```

Let’s create the NAD, Pod, and add a default route inside the Pod pointing to the SDN-GW anycast IP address:

```
kubectl apply -f sdngw_pure_evpn_use_case.yaml
namespace/sdngw-ns created
networkattachmentdefinition.K8s.cni.cncf.io/vn1 created
pod/pod1 created

kubectl get pods -n sdngw-ns -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
pod1 1/1 Running 0 8m58s 192.168.200.2 worker5 <none> <none>

ssh worker5 ip addr show vhost0
8: vhost0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9000 qdisc fq_codel
state UNKNOWN group default qlen 1000
link/ether 52:54:00:ce:d0:62 brd ff:ff:ff:ff:ff:ff
inet 192.168.5.58/24 brd 192.168.5.255 scope global vhost0
valid_lft forever preferred_lft forever
inet6 fe80::5054:ff:fece:d062/64 scope link
valid_lft forever preferred_lft forever

kubectl exec -it -n sdngw-ns pod1 /bin/bash -- ip addr show eth0
30: eth0@if31: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:97:4c:0b:30:b4 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.200.2/24 brd 192.168.200.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::28a4:48ff:fe19:de3d/64 scope link
       valid_lft forever preferred_lft forever


kubectl exec -it -n sdngw-ns pod1 /bin/bash -- ip r
default via 192.168.200.1 dev eth0
192.168.200.0/24 dev eth0 proto kernel scope link src 192.168.200.2
```

```
kubectl exec -it -n sdngw-ns pod1 /bin/bash -- ip r delete default


kubectl exec -it -n sdngw-ns pod1 /bin/bash -- ip r add default via
192.168.200.254 dev eth0


kubectl exec -it -n sdngw-ns pod1 /bin/bash -- ip r
default via 192.168.200.254 dev eth0
192.168.200.0/24 dev eth0 proto kernel scope link src 192.168.200.2
```

Let's verify L2 control plane functionality between the SDN-GW and CN2 controllers. We will verify if the MAC address '02:97:4c:0b:30:b4' is learned in the MP-BGP control plane by looking into the bgp.evpn.0 routing table. The next step would be to verify if the learned MAC address is installed in the EPVN data base and then from the EVPN data base it should be installed in the bridge mac table inside EVPN instance:

```
show route table bgp.evpn.0 evpn-mac-address 02:97:4c:0b:30:b4


bgp.evpn.0: 16 destinations, 19 routes (16 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both


2:192.168.5.58:6::10001::02:97:4c:0b:30:b4/304 MAC/IP
                   *[BGP/170] 09:22:49, MED 100, localpref 200, from
192.168.5.51
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.58)
                    [BGP/170] 09:22:46, MED 100, localpref 200, from
192.168.5.52
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.58)
2:192.168.5.58:6::10001::02:97:4c:0b:30:b4::192.168.200.2/304 MAC/IP
                   *[BGP/170] 09:22:49, MED 100, localpref 200, from
192.168.5.51
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.58)
                    [BGP/170] 09:22:46, MED 100, localpref 200, from
192.168.5.52
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.58)
```

```
show evpn database mac-address 02:97:4c:0b:30:b4
Instance: l2_use_case
VLAN  DomainId  MAC address        Active source
Timestamp        IP address
     10001      02:97:4c:0b:30:b4  192.168.5.58                   May 07
09:02:30  192.168.200.2


show bridge mac-table instance l2_use_case


MAC flags       (S -static MAC, D -dynamic MAC, L -locally learned, C
-Control MAC
    O -OVSDB MAC, SE -Statistics enabled, NM -Non configured MAC, R
-Remote PE MAC, P -Pinned MAC)


Routing instance : l2_use_case
Bridging domain : bd_1001, VLAN : 1001
   MAC                 MAC      Logical                Active
   address             flags    interface              source
   02:97:4c:0b:30:b4   D        vtep.32769             192.168.5.58
   22:70:29:5c:b9:cf   D        xe-0/0/1.0
   52:54:00:24:37:7d   D        xe-0/0/1.0
   fe:06:0a:0e:ff:f1   D        xe-0/0/1.0
```

Let's verify if we can ping the SDN-GW anycast gateway IP address from the CN2 Pod and if CN2 ARP is learned on the SDN-GW and converted to a corresponding host route:

```
kubectl exec -it -n sdngw-ns pod1 /bin/bash -- ping 192.168.200.254 -c5
PING 192.168.200.254 (192.168.200.254) 56(84) bytes of data.
64 bytes from 192.168.200.254: icmp_seq=1 ttl=64 time=2.21 ms
64 bytes from 192.168.200.254: icmp_seq=2 ttl=64 time=5.74 ms
64 bytes from 192.168.200.254: icmp_seq=3 ttl=64 time=1.73 ms
64 bytes from 192.168.200.254: icmp_seq=4 ttl=64 time=2.50 ms
64 bytes from 192.168.200.254: icmp_seq=5 ttl=64 time=1.68 ms


--- 192.168.200.254 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms



show arp | match 192.168.200.2
```

```
02:97:4c:0b:30:b4 192.168.200.2   192.168.200.2             irb.200
[vtep.32769]    none
22:70:29:5c:b9:cf 192.168.200.253 192.168.200.253           irb.200
[xe-0/0/1.0]    none

show route 192.168.200.2

evpn_5.inet.0: 9 destinations, 9 routes (9 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

192.168.200.2/32   *[EVPN/7] 09:20:42
                    >  via irb.200
```

Verify L2 connectivity from the CN2 Pod to a destination reachable via the SDN-GW:

```
show interfaces vtep.32769
  Logical interface vtep.32769 (Index 329) (SNMP ifIndex 563)
    Flags: Up SNMP-Traps Encapsulation: ENET2
    VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.58, L2 
Routing Instance: l2_use_case, L3 Routing Instance: default
    Input packets : 0
    Output packets: 0
    Protocol bridge, MTU: Unlimited
      Flags: Trunk-Mode

show interfaces vtep.32769
  Logical interface vtep.32769 (Index 329) (SNMP ifIndex 563)
    Flags: Up SNMP-Traps Encapsulation: ENET2
    VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.58, L2 
Routing Instance: l2_use_case, L3 Routing Instance: default
    Input packets : 4
    Output packets: 4
    Protocol bridge, MTU: Unlimited
      Flags: Trunk-Mode
kubectl exec -it -n sdngw-ns pod1 /bin/bash -- ping 192.168.200.253 -c20
PING 192.168.200.253 (192.168.200.253) 56(84) bytes of data.
64 bytes from 192.168.200.253: icmp_seq=1 ttl=64 time=4.37 ms
64 bytes from 192.168.200.253: icmp_seq=2 ttl=64 time=2.47 ms
--- 192.168.200.253 ping statistics ---
20 packets transmitted, 20 received, 0% packet loss, time 19030ms
```

```
rtt min/avg/max/mdev = 1.938/3.630/8.987/1.720 ms

--- 192.168.200.253 ping statistics ---
20 packets transmitted, 20 received, 0% packet loss, time 19029ms
rtt min/avg/max/mdev = 1.990/2.809/4.109/0.526 ms

show interfaces vtep.32769
  Logical interface vtep.32769 (Index 329) (SNMP ifIndex 563)
    Flags: Up SNMP-Traps Encapsulation: ENET2
    VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.58, L2
Routing Instance: l2_use_case, L3 Routing Instance: default
    Input packets : 21
    Output packets: 21
    Protocol bridge, MTU: Unlimited
      Flags: Trunk-Mode
```

Now verify EVPN type 5 control plane between the SDN-GW and CN2 Controllers. You can see that the EVPN host route (192.168.200.2) is learned and installed on the SDN-GW routing table evpn_5.inet.0 with community value of “encapsulation:vxlan(0x8)”:

```
show route 192.168.200.2 detail

evpn_5.inet.0: 9 destinations, 9 routes (9 active, 0 holddown, 0 hidden)
192.168.200.2/32 (1 entry, 1 announced)
        *EVPN   Preference: 7
                Next hop type: Interface, Next hop index: 0
                Address: 0xd6e80e0
                Next-hop reference count: 3
                Next hop: via irb.200, selected
                State: <Active Int Ext>
                Age: 9:43:02
                Validation State: unverified
                Task: l2_use_case-evpn
                Announcement bits (2): 1-evpn_5-OSPF 3-Resolve tree 6
                AS path: I
Communities: encapsulation:vxlan(0x8) mac-mobility:0x0
(sequence 1) evpn-etree:0x0:root (label 0) unknown
type 0x8071:0xfc00:0x2711 unknown type
0x8084:0xfc00:0x60007 unknown type
```

```
  0x8084:0xfc00:0x70007 unknown type
0x8084:0xfc00:0x80006 unknown type
0x8084:0xfc00:0xb0007 unknown type
0x8084:0xfc00:0x310007.
```

The following output shows that the EVPN type 5 routes are being advertised by SDN-GW to CN2 Controller from routing table evpn_5.evpn.0 (where evpn_5 is vrf name and evpn.0 is routing table name.

```
show bgp summary
Peer                         AS       InPkt     OutPkt    OutQ   Flaps Last
Up/Dwn State|#Active/Received/Accepted/Damped...
192.168.5.51          64512       1799       2089       0       3
9:56:02 Establ
  bgp.rtarget.0: 34/34/34/0
  inet.0: 4/4/4/0
  bgp.evpn.0: 3/3/3/0
  evpn_5.evpn.0: 3/3/3/0
  l2_use_case.evpn.0: 3/3/3/0
  __default_evpn__.evpn.0: 0/0/0/0
192.168.5.52          64512       1798       2088       0       3
9:55:58 Establ
  bgp.rtarget.0: 30/30/30/0
  inet.0: 4/4/4/0
  bgp.evpn.0: 0/3/3/0
  evpn_5.evpn.0: 0/3/3/0
  l2_use_case.evpn.0: 0/3/3/0
  __default_evpn__.evpn.0: 0/0/0/0
192.168.5.53          64512       1797       1979       0       3
9:56:05 Establ
  bgp.rtarget.0: 23/23/23/0
  inet.0: 4/4/4/0
  bgp.evpn.0: 0/0/0/0
  l2_use_case.evpn.0: 0/0/0/0
  __default_evpn__.evpn.0: 0/0/0/0


show route advertising-protocol bgp 192.168.5.51 table evpn_5.evpn.0


evpn_5.evpn.0: 6 destinations, 9 routes (6 active, 0 holddown, 0 hidden)
  Prefix          Nexthop              MED     Lclpref    AS path
```

```
  5:172.172.172.172:1::0::0.0.0.0::0/248
*                          Self                         100        I
  5:172.172.172.172:1::0::192.168.200.0::24/248
*                          Self                         100        I
  5:172.172.172.172:1::0::192.168.201.0::24/248
*                          Self                         100        I


show route advertising-protocol bgp 192.168.5.52 table evpn_5.evpn.0

evpn_5.evpn.0: 6 destinations, 9 routes (6 active, 0 holddown, 0 hidden)
  Prefix          Nexthop             MED     Lclpref    AS path
  5:172.172.172.172:1::0::0.0.0.0::0/248
*                          Self                         100        I
  5:172.172.172.172:1::0::192.168.200.0::24/248
*                          Self                         100        I
  5:172.172.172.172:1::0::192.168.201.0::24/248
*                          Self                         100        I
```

Now let's verify if the EVPN type 5 route advertised by SDN-GW are received and installed in corresponding VRF in CN2 vRouter. Hence, the pod1 is instantiated on worker5. We will log in to the worker5 vrouter-agent container and use the vRouter CLI to get various outputs:

```
kubectl get pods -n sdngw-ns -o wide
NAME   READY   STATUS    RESTARTS   AGE   IP              NODE
NOMINATED NODE   READINESS GATES
pod1   1/1     Running   0          23h   192.168.200.2   worker5
<none>           <none>


kubectl get pods -n contrail -o wide | grep vrouter | grep worker5
contrail-vrouter-nodes-g4ccb                         3/3     Running   8
24d   192.168.24.137   worker5       <none>           <none>


kubectl exec -n contrail -it -c contrail-vrouter-agent contrail-vrouter-
nodes-g4ccb /bin/bash
```

The vif –-list command returns the interface created on the vRouter and the corresponding VRF ID. We will find the vrf ID corresponding to pod1 IP (i.e., 192.168.200.2):

```
vif --list | grep 192.168.200.2 -a5
            RX packets:13601  bytes:572766 errors:0
```

```
            TX packets:13575  bytes:570150 errors:0
            Drops:13601

vif0/9      OS: tapeth0-5a559e NH: 113
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.200.2
            Vrf:6 Mcast Vrf:6 Flags:PL3L2Er QOS:-1 Ref:6
            RX packets:18212  bytes:1283084 errors:0
            TX packets:17184  bytes:1161972 errors:0
            Drops:10869
```

Verify if the route for the prefix 0.0.0.0/0 is installed in the VRF (vrf 6), or not:

```
rt --get 0.0.0.0/0 --vrf 6
Match 0.0.0.0/0 in vRouter inet4 table 0/6/unicast


Flags: L=Label Valid, P=Proxy ARP, T=Trap ARP, F=Flood ARP, Ml=MAC-IP
learnt route
vRouter inet4 routing table 0/6/unicast
Destination          PPL        Flags        Label         Nexthop
Stitched MAC(Index)
0.0.0.0/0              0           LP       1001            120
```

The output shows that the Nexthop value 120 is attached to the prefix 0.0.0.0/0. Let's verify if this Nexthop points to theSDN-GW loopback IP (i.e. 172.172.172.172) or not:

```
nh --get 120
Id:120        Type:Tunnel         Fmly: AF_INET  Rid:0  Ref_cnt:3043
Vrf:0
              Flags:Valid, Vxlan, Etree Root, l3_vxlan,
              Oif:0 Len:14 Data:ac 4b c8 2b 77 c1 52 54 00 ce d0 62 08 00


              Sip:192.168.5.58 Dip:172.172.172.172 L3_Vxlan_Mac:
2c:6b:f5:12:5b:f0
```

EVPN type 5 route control plane functionality is verified between the SDN-GW and the CN2 vRouter. Let's verify the forwarding plane functionality. We will initiate ICMP ping requests from an outer destination that has connectivity to the SDN-GW and then further to the CN2 Pod via EVPN type 5 routes:

```
ip -4 -br a
lo                UNKNOWN        127.0.0.1/8
ens3              UP             192.168.24.82/24
docker0           UP             172.17.0.1/16
```

```
show interfaces vtep.32769
  Logical interface vtep.32769 (Index 329) (SNMP ifIndex 563)
    Flags: Up SNMP-Traps Encapsulation: ENET2
    VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.58, L2
Routing Instance: l2_use_case, L3 Routing Instance: default
    Input packets : 0
    Output packets: 0
    Protocol bridge, MTU: Unlimited
      Flags: Trunk-Mode

ping 192.168.200.2 -c20
PING 192.168.200.2 (192.168.200.2) 56(84) bytes of data.
64 bytes from 192.168.200.2: icmp_seq=1 ttl=62 time=3.93 ms

show interfaces vtep.32769
  Logical interface vtep.32769 (Index 329) (SNMP ifIndex 563)
    Flags: Up SNMP-Traps Encapsulation: ENET2
    VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.58, L2
Routing Instance: l2_use_case, L3 Routing Instance: default
    Input packets : 7

    Output packets: 7
    Protocol bridge, MTU: Unlimited
      Flags: Trunk-Mode

64 bytes from 192.168.200.2: icmp_seq=20 ttl=62 time=2.82 ms

--- 192.168.200.2 ping statistics ---
20 packets transmitted, 20 received, 0% packet loss, time 19029ms
rtt min/avg/max/mdev = 2.181/3.306/8.143/1.223 ms

show interfaces vtep.32769
  Logical interface vtep.32769 (Index 329) (SNMP ifIndex 563)
    Flags: Up SNMP-Traps Encapsulation: ENET2
    VXLAN Endpoint Type: Remote, VXLAN Endpoint Address: 192.168.5.58, L2
Routing Instance: l2_use_case, L3 Routing Instance: default
    Input packets : 20
    Output packets: 20
    Protocol bridge, MTU: Unlimited
      Flags: Trunk-Mode
```

The above output shows that 20 ICMP ping packets were sent to the CN2 Pod from an outer source and vtep.32769 statistics also confirm that those 20 packets traversed through it.

LoadBalancer (LBR) Service Connectivity to SDN GW

LBR service is a Kubernetes construct where a service is exposed using an externally reachable IP, which acts like a floating IP and all incoming requests to service floating and load balancer IP are load balanced to the ervice end point Pods. CN2 also supports LBR type service, and you can extend load balancer floating IP to the SDN GW using same methodology described in the previous section "Extending Layer 3 Connectivity to SDN GW."

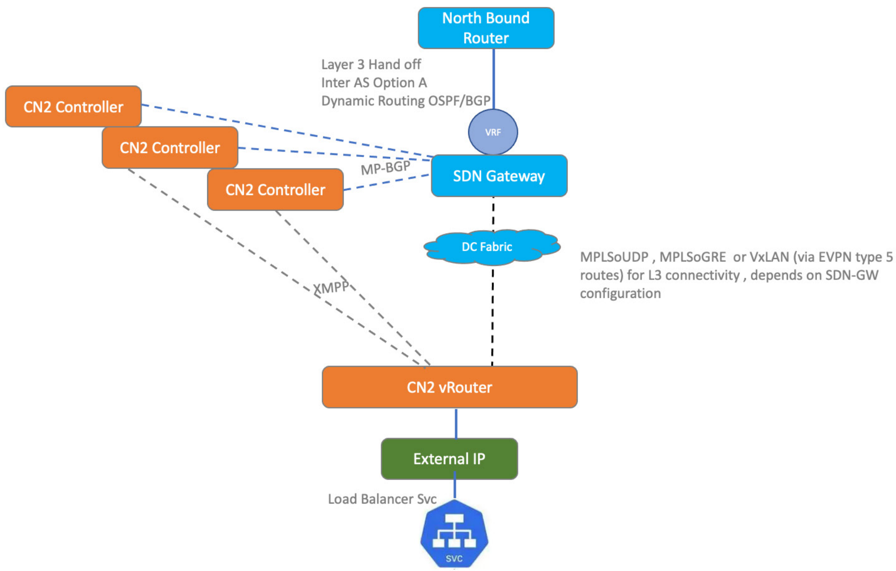


Figure 25 Load Balancer Service Type Extension to SDN Gateway Router

There are multiple ways to define LBR service and bind it with external network. We will describe two methods here.

LBR Service Using Default-External Network

In this approach a default-external network is created by assigning it a label "service.contrail.juniper.net/externalNetworkSelector: default-external" and this VN will be automatically selected to serve an IP address to a LBR Service:a

```
cat > loadbalacer_svc_default_external_network.yaml <<END_OF_SCRIPT
---
apiVersion: v1
```

```
kind: Namespace
metadata:
  name: dev-ns
---
apiVersion: “K8s.cni.cncf.io/v1”
kind: NetworkAttachmentDefinition
metadata:
  name: dev-vn
  namespace: dev-ns
  annotations:
    juniper.net/networks: ‘{
      “ipamV4Subnet”: “10.10.10.0/24”,
    }’
spec:
  config: ‘{
  “cniVersion”: “0.3.1”,
  “name”: “dev-vn”,
  “type”: “contrail-K8s-cni”
}’
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-1
  namespace: dev-ns
  labels:
    app: frontend-dev
spec:
  containers:
    - name: frontend-dev-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        [“bash”, “-c”, “python3 -m http.server 80 --directory /tmp/”]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
```

```
  nodeName: worker1
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-2
  namespace: dev-ns
  labels:
    app: frontend-dev
spec:
  containers:
    - name: frontend-dev-2
      image:  deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 80 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker2
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: ecmp-default
  namespace: dev-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.30.1.0/24",
      "routeTargetList": ["target:64562:2101"]
    }'
  labels:
    service.contrail.juniper.net/externalNetworkSelector: default-
external
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "ecmp-default",
```

```
  "type": "contrail-K8s-cni"
}'

---
apiVersion: v1
kind: Service
metadata:
  name: frontend-dev
  namespace: dev-ns
spec:
  ports:
  - name: port-443
    targetPort: 443
    protocol: TCP
    port: 443
  - name: port-80
    targetPort: 80
    protocol: TCP
    port: 80
  selector:
    app: frontend-dev
  type: LoadBalancer
---
END_OF_SCRIPT
```

Create NADs, Pods, LBR service, and then verify. We can see that an "EXTERNAL-IP, 10.30.1.2" is assigned to LBR service from the subnet assigned to NAD (ecmp-default):

```
kubectl create -f loadbalacer_svc_default_external_network.yaml
namespace/dev-ns created
networkattachmentdefinition.K8s.cni.cncf.io/dev-vn created
pod/frontend-dev-1 created
pod/frontend-dev-2 created
networkattachmentdefinition.K8s.cni.cncf.io/ecmp-default created
service/frontend-dev created

kubectl get all -n dev-ns
NAME                 READY   STATUS              RESTARTS   AGE
pod/frontend-dev-1   0/1     ContainerCreating   0          10s
```

```
pod/frontend-dev-2   0/1     ContainerCreating   0          10s

NAME                    TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)
AGE
service/frontend-dev   LoadBalancer   10.233.4.18   10.30.1.2
443:31926/TCP,80:31918/TCP   10s

kubectl get all -n dev-ns
NAME                   READY   STATUS    RESTARTS   AGE
pod/frontend-dev-1   1/1     Running   0          2m13s
pod/frontend-dev-2   1/1     Running   0          2m13s

NAME                    TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)
AGE
service/frontend-dev   LoadBalancer   10.233.4.18   10.30.1.2
443:31926/TCP,80:31918/TCP   2m13s

kubectl exec -it  -n dev-ns frontend-dev-1 /bin/bash
hostname | tee -a /tmp/index.html && exit

kubectl exec -it  -n dev-ns frontend-dev-2 /bin/bash
hostname | tee -a /tmp/index.html && exit
```

Test the connectivity from an external host which has reachability to the SDN-GW VRF and then further to, CN2 LBR External IP.

```
curl 10.30.1.2
frontend-dev-1
```

LBR Service Using Custom-External Network

In this approach a custom external network is created which needs to be referred to in the LBR service definition using the annotation "service.contrail.juniper.net/externalNetwork: namespace-name/VN-name:" to serve an external IP to LBR service:

```
cat > loadbalacer_svc_custom_external_network.yaml <<END_OF_SCRIPT
apiVersion: v1
kind: Namespace
metadata:
  name: ecmp-project
---
apiVersion: "K8s.cni.cncf.io/v1"
```

```
kind: NetworkAttachmentDefinition
metadata:
  name: ecmp-default
  namespace: ecmp-project
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.30.1.0/24",
      "routeTargetList": ["target:64562:2101"]
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "ecmp-default",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-01
  namespace: ecmp-project
  labels:
    run: ecmp
spec:
  containers:
    - name: frontend-01
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 8080 --directory /tmp/"]
      securityContext:
        privileged: true
  nodeName: worker1
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-02
  namespace: ecmp-project
```

```
  labels:
    run: ecmp
spec:
  containers:
    - name: frontend-02
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 8080 --directory /tmp/"]
      securityContext:
        privileged: true
  nodeName: worker2
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-03
  namespace: ecmp-project
  labels:
    run: ecmp
spec:
  containers:
    - name: frontend-03
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 8080 --directory /tmp/"]
      securityContext:
        privileged: true
  nodeName: worker3
---
apiVersion: v1
kind: Service
metadata:
  name: test-lb-default
  namespace: ecmp-project
  annotations:
    service.contrail.juniper.net/externalNetwork: ecmp-project/ecmp-
default
spec:
  type: LoadBalancer
```

```
  selector:
    run: ecmp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 8080
END_OF_SCRIPT
```

Create NAD, Pods, Service, and connectivity verification. You can see that an "EXTERNAL-IP, 10.30.1.2" is assigned to the LBR service from the subnet assigned to NAD (ecmp-default):

```
kubectl apply -f loadbalacer_svc_custom_external_network.yaml
namespace/ecmp-project created
networkattachmentdefinition.K8s.cni.cncf.io/ecmp-default created
pod/frontend-01 created
pod/frontend-02 created
pod/frontend-03 created
service/test-lb-default created


kubectl get all -n ecmp-project
NAME              READY   STATUS    RESTARTS   AGE
pod/frontend-01   1/1     Running   0          47s
pod/frontend-02   1/1     Running   0          47s
pod/frontend-03   1/1     Running   0          46s

NAME                      TYPE           CLUSTER-IP    EXTERNAL-IP
PORT(S)        AGE
service/test-lb-default   LoadBalancer   10.233.52.5   10.30.1.2
80:30994/TCP   46s


kubectl exec -it  -n ecmp-project frontend-01 /bin/bash
hostname | tee -a /tmp/index.html && exit

kubectl exec -it  -n ecmp-project frontend-02 /bin/bash
hostname | tee -a /tmp/index.html && exit

kubectl exec -it  -n ecmp-project frontend-03 /bin/bash
```

```
hostname | tee -a /tmp/index.html && exit
```

Test connectivity from an external host which has reachability to the SDN-GW VRF and then further to CN2 LBR External IP.

```
curl 10.30.1.2
frontend-01
```

On the SDN GW you can see that the route 10.30.1.2/32 is being received from three BGP neighbors (therefore, 192.168.5.51, 192.168.5.52, and 192.168.5.53 which are CN2 controllers). You can also see three different next hops (therefore, 192.168.5.54, 192.168.5.55, and 192.168.5.56 which are CN2 worker node IPs) are also installed in the routing table. It confirms that one floating IP is created and installed on each worker node where target Pods are instantiated:

```
show route table l3_use_case.inet.0 10.30.1.2


l3_use_case.inet.0: 8 destinations, 13 routes (6 active, 0 holddown, 2 
hidden)
+ = Active Route, - = Last Active, * = Both


10.30.1.2/32       *[BGP/170] 00:03:00, MED 100, localpref 200, from 
192.168.5.51
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172 
dest 192.168.5.54), Push 25
                    [BGP/170] 00:03:01, MED 100, localpref 200, from 
192.168.5.51
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172 
dest 192.168.5.55), Push 37
                    [BGP/170] 00:03:00, MED 100, localpref 200, from 
192.168.5.51
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172 
dest 192.168.5.56), Push 25
                    [BGP/170] 00:03:00, MED 100, localpref 200, from 
192.168.5.52
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172 
dest 192.168.5.54), Push 25
                    [BGP/170] 00:03:01, MED 100, localpref 200, from 
192.168.5.52
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
```

```
dest 192.168.5.55), Push 37
                    [BGP/170] 00:03:00, MED 100, localpref 200, from
192.168.5.53
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.56), Push 25
```

LBR Service Using Custom-Pod Network

In these two examples we have covered LBR Service being attached to end points on the default-Pod Network but in this next example, we will cover how to attach LoadBalancer Service with end points on the custom-PodNetwork.

```
cat > lbr-custom-pod-network.yaml <<END_OF_SCRIPT
---
apiVersion: v1
kind: Namespace
metadata:
  name: dev-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: ecmp-default
  namespace: dev-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.30.1.0/24",
      "routeTargetList": ["target:64562:2101"]
    }'
  labels:
    service.contrail.juniper.net/externalNetworkSelector: default-
external
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "ecmp-default",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
```

```
metadata:
  name: dev-vn
  namespace: dev-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.10.10.0/24",
      "podNetwork": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "dev-vn",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-1
  namespace: dev-ns
  labels:
    app: frontend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: frontend-dev-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 80 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker4
---
apiVersion: v1
```

```
kind: Pod
metadata:
  name: frontend-dev-2
  namespace: dev-ns
  labels:
    app: frontend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: frontend-dev-2
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 80 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker5
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-dev
  namespace: dev-ns
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  ports:
  - name: port-443
    targetPort: 443
    protocol: TCP
    port: 443
  - name: port-80
    targetPort: 80
    protocol: TCP
    port: 80
```

```
  type: LoadBalancer
  selector:
    app: frontend-dev
---
END_OF_SCRIPT
```

Let's create NADs, Pods, LBR Service ,and test connectivity. You can see that an "EXTERNAL-IP, 10.30.1.2" is assigned to LBR service from the subnet assigned to NAD (ecmp-default).

```
kubectl apply -f lbr-custom-pod-network.yaml
namespace/dev-ns created
networkattachmentdefinition.K8s.cni.cncf.io/ecmp-default created
networkattachmentdefinition.K8s.cni.cncf.io/dev-vn created
pod/frontend-dev-1 created
pod/frontend-dev-2 created
service/frontend-dev created

kubectl get all -n dev-ns
NAME                 READY   STATUS    RESTARTS   AGE
pod/frontend-dev-1   1/1     Running   0          115s
pod/frontend-dev-2   1/1     Running   0          115s

NAME                    TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)
AGE
service/frontend-dev   LoadBalancer   10.233.6.42   10.30.1.2
443:30214/TCP,80:32359/TCP   114s



kubectl exec -n dev-ns frontend-dev-1 /bin/bash -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
248: eth0@if249: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:06:aa:78:46:2b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.10.10.3/24 brd 10.10.10.255 scope global eth0
```

```
       valid_lft forever preferred_lft forever
    inet6 fe80::4004:18ff:fe91:53e/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec -n dev-ns frontend-dev-2 /bin/bash -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
272: eth0@if273: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:80:93:56:c7:1b brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.10.10.2/24 brd 10.10.10.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::c436:bfff:fe42:213b/64 scope link
       valid_lft forever preferred_lft forever

show route 10.30.1.2 table l3_use_case.inet.0

l3_use_case.inet.0: 8 destinations, 11 routes (6 active, 0 holddown, 2
hidden)
+ = Active Route, - = Last Active, * = Both

10.30.1.2/32       *[BGP/170] 00:02:27, MED 100, localpref 200, from
192.168.5.51
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.57), Push 25
                    [BGP/170] 00:02:27, MED 100, localpref 200, from
192.168.5.52
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.57), Push 25
                    [BGP/170] 00:02:27, MED 100, localpref 200, from
192.168.5.52
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.58), Push 25
```

```
                    [BGP/170] 00:02:27, MED 100, localpref 200, from 
192.168.5.53
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172 
dest 192.168.5.58), Push 25

kubectl exec -it  -n dev-ns frontend-dev-1 /bin/bash
hostname | tee -a /tmp/index.html && exit

kubectl exec -it  -n dev-ns frontend-dev-2 /bin/bash
hostname | tee -a /tmp/index.html && exit

kubectl exec -it  -n dev-ns frontend-dev-3 /bin/bash
hostname | tee -a /tmp/index.html && exit
```

Test connectivity from an external host which has reachability to the SDN-GW VRF and then further to the CN2 LBR External IP.

```
curl 10.30.1.2
frontend-dev-1Ingress
```

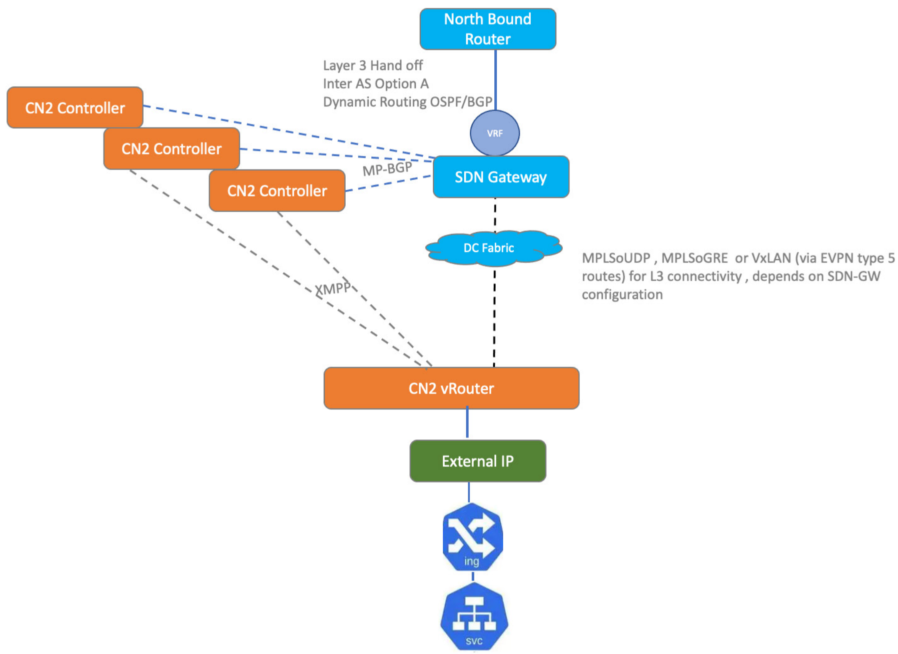


Figure 26 *K8s Ingress extension to SDN Gateway Router*

Ingress allows exposure of http and https services in K8s clusters while managing access to those services via ingress rules coupled with back-end services. In CN2-based K8s clusters, an external IP is required in default-namespace before deploying the Ingress controller. The external IP is used to access services controlled by Ingress Rules. We will use Nginx Ingress controller in this write up, however, HAProxy and Contour Ingress Controllers are also tested with CN2.

The workflow to deploy Ingress and ingress backed services is as follows:

- Create a default-external network in default-namespace.
- Extend the newly created default-external network to the SDN GW described in the preceding section "Extending Layer 3 Connectivity to SDN GW)."
- Deploy the Ingress controller.
- Create the Pod, Service, and Ingress Rule

First let's create default-external in the default-namespace.

```
cat > default.external.yaml <<END_OF_SCRIPT
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: external-default
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.30.1.0/24",
      "routeTargetList": ["target:64562:2101"]
    }'
  labels:
    service.contrail.juniper.net/externalNetworkSelector: default-
external
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "external-default",
  "type": "contrail-K8s-cni"
}'
END_OF_SCRIPT
kubectl apply -f default.external.yaml
```

Now install the Ingress controller (in our case we are using Nginx):

```
curl 'https://get.helm.sh/helm-v3.11.3-linux-amd64.tar.gz' -o helm-
v3.11.3-linux-amd64.tar.gz
```

```
tar xzf helm-v3.11.3-linux-amd64.tar.gz

sudo mv linux-amd64/helm  /usr/local/bin/helm

helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

helm repo update

#Install nginx controller

helm install nginx-ingress ingress-nginx/ingress-nginx

To check if LB is available

kubectl --namespace default get services -o wide -w nginx-ingress-
ingress-nginx-controller


kubectl get  pod | grep ingress

nginx-ingress-ingress-nginx-controller-58ff69979c-2lps6   1/1     Running
2 (66s ago)   107s

kubectl get ValidatingWebhookConfiguration |grep ingress

nginx-ingress-ingress-nginx-admission   1          3m8s


kubectl get svc

NAME                                                TYPE
CLUSTER-IP      EXTERNAL-IP   PORT(S)                       AGE

kubernetes                                          ClusterIP
10.233.0.1      <none>        443/TCP                       2d19h

nginx-ingress-ingress-nginx-controller             LoadBalancer
10.233.47.127   10.30.1.2     80:31859/TCP,443:31007/TCP   3m37s

nginx-ingress-ingress-nginx-controller-admission   ClusterIP
10.233.7.201    <none>        443/TCP                       3m37s



show route 10.30.1.2 table l3_use_case.inet.0


l3_use_case.inet.0: 8 destinations, 9 routes (6 active, 0 holddown, 2
hidden)

+ = Active Route, - = Last Active, * = Both


10.30.1.2/32       *[BGP/170] 00:01:00, MED 100, localpref 200, from
192.168.5.51

                      AS path: ?, validation-state: unverified

                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.57), Push 25

                    [BGP/170] 00:01:00, MED 100, localpref 200, from
192.168.5.52
```

```
                      AS path: ?, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.57), Push 25
```

Define NAD, Pod, Service, and Ingress object with ingress rules in the definition file:

```
cat > lbr-pod-ingress-ns.yaml <<END_OF_SCRIPT
---
apiVersion: v1
kind: Namespace
metadata:
  name: dev-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: dev-vn
  namespace: dev-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.10.10.0/24",
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "dev-vn",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-1
  namespace: dev-ns
  labels:
    app: frontend-dev
spec:
  containers:
    - name: frontend-dev-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
```

```
      command:
        [“bash”, “-c”, “python3 -m http.server 80 --directory /tmp/”,
“hostname | tee -a /tmp/index.html”]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker1
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-2
  namespace: dev-ns
  labels:
    app: frontend-dev
spec:
  containers:
    - name: frontend-dev-2
      image:  deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        [“bash”, “-c”, “python3 -m http.server 80 --directory /tmp/”,
“hostname | tee -a /tmp/index.html”]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker3
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-dev
  namespace: dev-ns
spec:
  ports:
  - name: port-443
```

```
    targetPort: 443
    protocol: TCP
    port: 443
  - name: port-80
    targetPort: 80
    protocol: TCP
    port: 80
  selector:
    app: frontend-dev
---
apiVersion: networking.K8s.io/v1
kind: Ingress
metadata:
  name: hello-kubernetes-ingress
  namespace: dev-ns
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
  - host: "frontend.dev.svc"
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: frontend-dev
            port:
              number: 80
END_OF_SCRIPT
```

Create NAD, Pod, Service, and Ingress Rules and verify connectivity.

```
kubectl apply -f lbr-pod-ingress-ns.yaml
namespace/dev-ns unchanged
networkattachmentdefinition.K8s.cni.cncf.io/dev-vn unchanged
pod/frontend-dev-1 unchanged
pod/frontend-dev-2 unchanged
service/frontend-dev unchanged
ingress.networking.K8s.io/hello-kubernetes-ingress created
```

```
kubectl get all -n dev-ns
NAME                 READY   STATUS    RESTARTS   AGE
pod/frontend-dev-1   1/1     Running   0          13m
pod/frontend-dev-2   1/1     Running   0          13m

NAME                   TYPE          CLUSTER-IP     EXTERNAL-IP
PORT(S)                      AGE
service/frontend-dev   ClusterIP      10.233.24.54   <pending>
443:31001/TCP,80:32420/TCP   13m

kubectl get ingress -n dev-ns
NAME                       CLASS    HOSTS              ADDRESS     PORTS
AGE
hello-kubernetes-ingress   <none>   frontend.dev.svc   10.30.1.2   80
3m22s
```

Test connectivity from an external host which has reachability to the SDN-GW VRF and then further to the CN2 Ingress External IP:

```
curl -H "Host:frontend.dev.svc" 10.30.1.2
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
</ul>
<hr>
</body>
</html>
```

Since the "Host:frontend.dev" is not defined in the ingress rules thus request is rejected by the Ingress Controller.

```
curl -H "Host:frontend.dev" 10.30.1.2
<html>
<head><title>404 Not Found</title></head>
```

```
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

In this example we have configured an Ingress back-end service on the default-PodNetwork but in this next example we will use a custom PodNetwork for the Ingress back-end service:

```
cat > ingress-custom-pod-network.yaml <<END_OF_SCRIPT
---
apiVersion: v1
kind: Namespace
metadata:
  name: dev-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: dev-vn
  namespace: dev-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.10.10.0/24",
      "podNetwork": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "dev-vn",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-1
  namespace: dev-ns
  labels:
```

```
    app: frontend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: frontend-dev-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 80 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker4
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-2
  namespace: dev-ns
  labels:
    app: frontend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: frontend-dev-2
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 80 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker5
---
```

```
apiVersion: v1
kind: Service
metadata:
  name: frontend-dev
  namespace: dev-ns
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  ports:
  - name: port-443
    targetPort: 443
    protocol: TCP
    port: 443
  - name: port-80
    targetPort: 80
    protocol: TCP
    port: 80
  selector:
    app: frontend-dev
---
apiVersion: networking.K8s.io/v1
kind: Ingress
metadata:
  name: hello-kubernetes-ingress
  namespace: dev-ns
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
  - host: "frontend.dev.svc"
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: frontend-dev
            port:
```

```
              number: 80
END_OF_SCRIPT
```

Create NAD, Pod, Service, Ingress Rules and test connectivity.

```
kubectl apply -f ingress-custom-pod-network.yaml
namespace/dev-ns created
networkattachmentdefinition.K8s.cni.cncf.io/dev-vn created
pod/frontend-dev-1 created
pod/frontend-dev-2 created
service/frontend-dev created
ingress.networking.K8s.io/hello-kubernetes-ingress created



kubectl get all -n dev-ns
NAME                 READY   STATUS    RESTARTS   AGE
pod/frontend-dev-1   1/1     Running   0          108s
pod/frontend-dev-2   1/1     Running   0          108s

NAME                   TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
service/frontend-dev   ClusterIP   10.233.58.113   <none>        443/
TCP,80/TCP   108s

kubectl exec  -n dev-ns frontend-dev-1 /bin/bash  -- ip add
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
246: eth0@if247: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
    link/ether 02:9e:c7:9d:d1:f6 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.10.10.3/24 brd 10.10.10.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::c86a:eeff:fe8e:7579/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec  -n dev-ns frontend-dev-2 /bin/bash  -- ip add
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
270: eth0@if271: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:ac:5f:e3:2d:81 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.10.10.2/24 brd 10.10.10.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::e0ef:78ff:fe68:9be/64 scope link
       valid_lft forever preferred_lft forever
```

```
kubectl get ingress -n dev-ns
NAME                       CLASS    HOSTS              ADDRESS     PORTS   AGE
hello-kubernetes-ingress   <none>   frontend.dev.svc   10.30.1.2   80      2m12s
```

Test connectivity from an external host which has reachability to SDN-GW VRF and then further to the CN2 Ingress External IP.

```
curl -H "Host:frontend.dev.svc" 10.30.1.2
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
</ul>
<hr>
</body>
</html>
```

Since the "Host:frontend.dev" is not defined in the ingress rules thus request is rejected by the Ingress controller:

```
curl -H "Host:frontend.dev.com" 10.30.1.2
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

Chapter 8

Extending SRIOV Networking to Cloud-Native Work Loads

This chapter describes Single Root I/O Virtualization (SR-IOV) and how CN2 handles SRIOV VFs plumbing into K8s workloads.

SRIOV Introduction

Developed by the PCI-SIG (PCI Special Interest Group), the Single Root I/O Virtualization (SR-IOV) specification is a standard for a type of PCI device assignment that can share a single device to multiple virtual machines. SR-IOV improves device performance for virtual machines and containers. Reference (https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/virtualization_host_configuration_and_guest_installation_guide/chap-virtualization_host_configuration_and_guest_installation_guide-sr_iov).

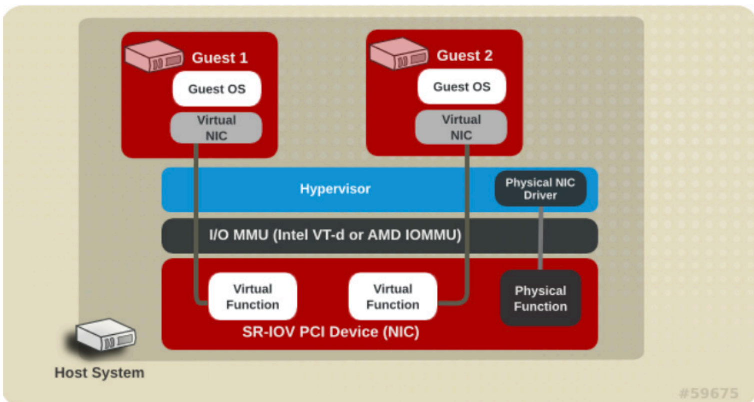


Figure 27 *Single Root Input Output Virtualization (SRIOV)*
Courtesy to RedHat for above diagram.

Deployment Workflow

- Identify SRIOV supported NIC on K8s worker nodes.
- Enable SRIOV support in grub.
- Create SRIOV VIFs on SRIOV supported NIC and make it persistent across machine reboot.
- Create SRIOV ConfigMap.

- Add SRIOV CNI Plugin into K8s Cluster.
- Label SRIOV worker nodes.
- Deploy SRIOV CNI and CN2-IPAM.

Identify SRIOV Supported NIC

```
sudo lshw -c network -businfo

Bus info          Device      Class         Description

=========================================================

pci@0000:01:00.0  eno1        network       Ethernet Controller
10-Gigabit X540-AT2

pci@0000:01:00.1  eno2        network       Ethernet Controller
10-Gigabit X540-AT2

pci@0000:07:00.0  eno3        network       I350 Gigabit Network
Connection

pci@0000:07:00.1  eno4        network       I350 Gigabit Network
Connection
```

Enable SRIOV Support in Grub

```
sed -i s/GRUB_CMDLINE_LINUX_DEFAULT=""/GRUB_CMDLINE_LINUX_DEFAULT="intel_
iommu=on iommu=pt"/ /etc/default/grub

update-grub

reboot
```

Create SRIOV VFs on SRIOV Supported NIC

```
cat << EOF > /etc/rc.local

#!/bin/bash

#echo 8 > /sys/class/net/eno1/device/sriov_numvfs

echo 8 > /sys/class/net/eno2/device/sriov_numvfs

EOF


cat << EOF > /etc/systemd/system/rc-local.service

[Unit]

Description=/etc/rc.local Compatibility

ConditionPathExists=/etc/rc.local


[Service]

Type=forking

ExecStart=/etc/rc.local start

TimeoutSec=0

StandardOutput=tty

RemainAfterExit=yes
```

```
SysVStartPriority=99

[Install]
WantedBy=multi-user.target
EOF

sudo chmod +x /etc/rc.local
sudo systemctl start rc-local.service
sudo systemctl status rc-local.service
sudo systemctl enable rc-local
```

Create SRIOV CNI ConfigMap

Note that to create ConfigMap you need to know detailed information about SRIOV VFs created over NIC of a particular worker node. Let's verify the SRIOV VFs:

```
sudo lshw -c network -businfo
Bus info          Device       Class          Description
==========================================================
pci@0000:01:00.0  eno1         network        Ethernet Controller
10-Gigabit X540-AT2
pci@0000:01:00.1  eno2         network        Ethernet Controller
10-Gigabit X540-AT2
pci@0000:01:10.1  eno2v0       network        X540 Ethernet Controller
Virtual Function
pci@0000:01:10.3  eno2v1       network        X540 Ethernet Controller
Virtual Function
pci@0000:01:10.5  eno2v2       network        X540 Ethernet Controller
Virtual Function
pci@0000:01:10.7  eno2v3       network        X540 Ethernet Controller
Virtual Function
pci@0000:01:11.1  eno2v4       network        X540 Ethernet Controller
Virtual Function
pci@0000:01:11.3  eno2v5       network        X540 Ethernet Controller
Virtual Function
pci@0000:01:11.5  eno2v6       network        X540 Ethernet Controller
Virtual Function
pci@0000:01:11.7  eno2v7       network        X540 Ethernet Controller
Virtual Function
pci@0000:07:00.0  eno3         network        I350 Gigabit Network
Connection
pci@0000:07:00.1  eno4         network        I350 Gigabit Network
Connection
```

The SRIOV VFs are created over eno2 NIC, and v0 to v7 virtual functions can be seen in the above snippet. Let's get some information about any VFs detected above and we'll refer that information to the SRIOV ConfigMap:

lspci -vmmkns 01:10.1

Slot: 01:10.1

Class: 0200

Vendor: 8086

Device: 1515

SVendor: 1028

SDevice: 1f61

Rev: 01

Driver: ixgbevf

Module: ixgbevf

NUMANode: 0

Here's the device code information for Intel Devices (https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/virtualization_host_configuration_and_guest_installation_guide/chap-virtualization_host_configuration_and_guest_installation_guide-sr_iov) . Now let's construct the SRIOV ConfigMap based on this information and create it with kubectl command:

```
cat <<EOF> sriovintel-config.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: sriovdp-config
  namespace: kube-system
data:
  config.json: |
    {
        "resourceList": [{
                "resourceName": "intel_sriov_netdevice",
                "selectors": {
                    "vendors": ["8086"],
                    "devices": ["1515"],
```

```
                    "drivers": ["ixgbevf"]
                }
            }
        ]
    }
EOF
kubectl create -f sriovintel-config.yaml
```

Add SRIOV CNI Plugin into K8s Cluster

Clone the git wiki sriov-network-device-plugin (https://github.com/K8snetworkplumbingwg/sriov-network-device-plugin) in your environment.

```
git clone https://github.com/K8snetworkplumbingwg/sriov-network-device-plugin.git
```

Create the SRIOV Plugin DaemonSet:

```
kubectl create -f ./sriov-network-device-plugin/deployments/sriovdp-daemonset.yaml
serviceaccount/sriov-device-plugin created
daemonset.apps/kube-sriov-device-plugin-amd64 created
daemonset.apps/kube-sriov-device-plugin-ppc64le created
daemonset.apps/kube-sriov-device-plugin-arm64 created
```

Verify the SRIOV Plugin status:

```
kubectl get pods -A -o wide |grep 'sriov-device-plugin'
kube-system       kube-sriov-device-plugin-amd64-hjlfh
1/1      Running     0              56s   192.168.24.92   worker3
<none>           <none>
kube-system       kube-sriov-device-plugin-amd64-p4tvx
1/1      Running     0              56s   192.168.24.90   worker1
<none>           <none>
kube-system       kube-sriov-device-plugin-amd64-zk46h
1/1      Running     0              56s   192.168.24.91   worker2
<none>           <none>
```

Verify if SRIOV VFs are available as an allocatable resource from a particular worker node:

```
kubectl get node worker1 -o json | jq '.status.allocatable'
{
  "cpu": "24",
  "ephemeral-storage": "529495978337",
  "hugepages-1Gi": "0",
  "hugepages-2Mi": "0",
```

```
  "intel.com/intel_sriov_netdevice": "8",
  "memory": "131929732Ki",
  "pods": "110"
}
kubectl get node worker2 -o json | jq '.status.allocatable'
{
  "cpu": "24",
  "ephemeral-storage": "516892523279",
  "hugepages-1Gi": "16Gi",
  "intel.com/intel_sriov_netdevice": "8",
  "memory": "115152516Ki",
  "pods": "110"
}

kubectl get node worker3 -o json | jq '.status.allocatable'
{
  "cpu": "32",
  "ephemeral-storage": "1061212961059",
  "hugepages-1Gi": "16Gi",
  "intel.com/intel_sriov_netdevice": "8",
  "memory": "115152516Ki",
  "pods": "110"
}
```

In the above snippet "intel.com/intel_sriov_netdevice": "8" means that in each worker node SRIOV Plugin has detected eight SRIOV VFs and marked those SRIOV VFs as allocatable resources.

Deploy SRIOV CNI and CN2-IPAM

You need to have the SRIOV CNI binary available on all the worker nodes that have SRIOV-enabled NICs. CN2 simplifies the installation of this binary by packaging it along with the IPAM binary installation.

The network plugin is responsible for the creation of a network interface for the container as well as making any necessary networking changes to enable the plumbing. The IP address is then assigned to the network interface, as well as routes and gateways being set up based on the results obtained by invoking the IPAM plugin. The available CNI IPAMs (https://www.cni.dev/plugins/current/ipam/) have some shortcomings. For example, consider the host-local IPAM plugin; when there are multiple Kubernetes worker nodes (which would typically be the case) the IPs that are allocated are local only to that worker node , and in case of multiple worker nodes, will end up with duplicate IPs being allocated to the Pods across different worker nodes. Hence, Juniper CN2 offers a centralized IPAM solution and plugin via the cn2-ipam executable.

```
Add the label sriov:"true" for each worker node with SR-IOV-enabled NICs:
kubectl edit nodes <node name>
labels:
  beta.kubernetes.io/arch: amd64
  sriov: "true"
```

Add the sriovLabelSelector on the contrail-vrouters-nodes CRD:

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

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

```
spec:
  agent:
    default:
  sriovLabelSelector:
    matchLabels:
      sriov: "true"
```

Verify the plug-in installation:

```
root@worker1:~# ls /opt/cni/bin/ | egrep 'sriov|cn2-ipam'
cn2-ipam
sriov
```

Note that the default location of the executable files depends on whether you use Kubernetes or OpenShift

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

Create NAD with SRIOV Support

```
cat > sriov-nad-201.yaml <<END_OF_SCRIPT
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: sriov-201
  namespace: default
  annotations:
```

```
    K8s.v1.cni.cncf.io/resourceName: intel.com/intel_sriov_netdevice
    juniper.net/networks: '{
      "vlanID": 201,
      "ipamV4Subnet": "192.168.201.0/24"
    }'
spec:
  config: |
    {
        "type": "sriov",
        "cniVersion": "0.3.1",
        "vlan": 201,
        "name": "sriov-201",
        "ipam": {
          "type": "cn2-ipam"
        }
    }
END_OF_SCRIPT
kubectl create -f  sriov-nad-201.yaml
networkattachmentdefinition.K8s.cni.cncf.io/sriov-201 created
```

Ensure that the switches and fabric connecting to the SRIOV-enabled NICs are configured to provide the required network connectivity:

- If it is a single switch, you need to configure the switch ports (connecting to the SRIOV-enabled NICs) with the right VLAN membership.
- If it is a fabric, you need to configure VXLAN-based overlays and VXLAN to VLAN mapping on the leaf switches to stretch the subnet across the IP boundary. This work can also be automated by leveraging the CN2-Apstra integration feature available in CN2 R23.1 release.

Create Pod with SRIOV VF Plumbed In

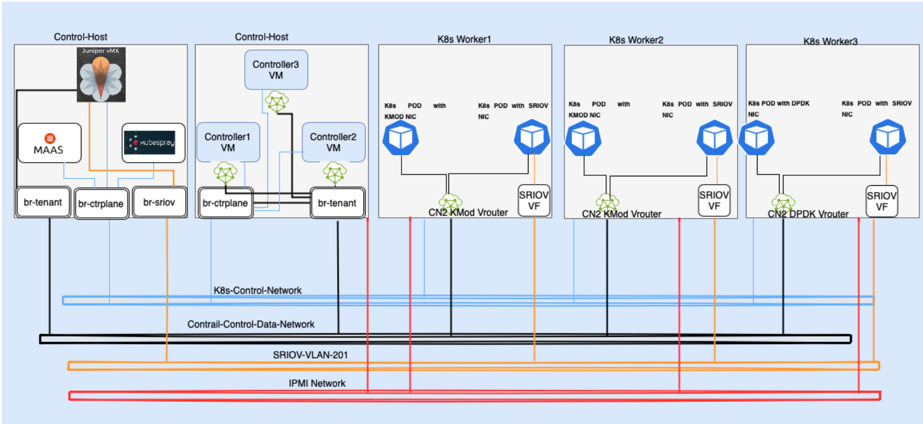


Figure 28 *K8S Pods with SRIOV Virtual Functions along with Native InterfaceA Pod is created on each worker node by referring to the SRIOV network created via the above-described NAD file. The Pod definition files are listed here:*

```
cat > sriov-pod-201-1.yaml <<END_OF_SCRIPT
apiVersion: v1
kind: Pod
metadata:
  name: sriov-pod-201-1
  annotations:
    K8s.v1.cni.cncf.io/networks: sriov-201

spec:
  containers:
  - name: sriov-pod-200-1c
    image: busybox:1.28
    imagePullPolicy: IfNotPresent
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
    resources:
     requests:
       intel.com/intel_sriov_netdevice: '1'
     limits:
       intel.com/intel_sriov_netdevice: '1'
    securityContext:
       privileged: true
  nodeName: worker1
```

```
END_OF_SCRIPT

cat > sriov-pod-201-2.yaml <<END_OF_SCRIPT
apiVersion: v1
kind: Pod
metadata:
  name: sriov-pod-201-2
  annotations:
    K8s.v1.cni.cncf.io/networks: sriov-201
spec:
  containers:
  - name: sriov-pod201-2c
    image: busybox:1.28
    imagePullPolicy: IfNotPresent
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
    resources:
     requests:
       intel.com/intel_sriov_netdevice: '1'
     limits:
       intel.com/intel_sriov_netdevice: '1'
    securityContext:
       privileged: true
  nodeName: worker2
END_OF_SCRIPT

cat > sriov-pod-201-3.yaml <<END_OF_SCRIPT
apiVersion: v1
kind: Pod
metadata:
  name: sriov-pod-201-3
  annotations:
    K8s.v1.cni.cncf.io/networks: sriov-201
spec:
  containers:
  - name: sriov-pod-201-3c
    image: busybox:1.28
    imagePullPolicy: IfNotPresent
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
```

```
    resources:
     requests:
       intel.com/intel_sriov_netdevice: '1'
     limits:
       intel.com/intel_sriov_netdevice: '1'
    securityContext:
       privileged: true
  nodeName: worker3
END_OF_SCRIPT
```

Okay, let's create the SRIOV Pods:

```
kubectl create -f sriov-pod-201-1.yaml
pod/sriov-pod-201-1 created

kubectl create -f sriov-pod-201-2.yaml
pod/sriov-pod-201-2 created

kubectl create -f sriov-pod-201-3.yaml
pod/sriov-pod-201-3 created
```

SRIOV Pods Verification

Verify the Pods creation:

```
kubectl get pods -o wide | grep sriov
sriov-pod-201-1   1/1     Running   0          8m      10.233.68.2
worker1   <none>           <none>
sriov-pod-201-2   1/1     Running   0          7m1s    10.233.69.0
worker2   <none>           <none>
sriov-pod-201-3   1/1     Running   0          6m38s   10.233.67.0
worker3   <none>           <none>
```

Log in into the container to check if SRIOV VF is attached to it or not:

```
kubectl exec sriov-pod-201-1 -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
7: net1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq qlen 1000
    link/ether 06:53:76:16:d8:b2 brd ff:ff:ff:ff:ff:ff
```

```
    inet 192.168.201.2/24 brd 192.168.201.255 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::453:76ff:fe16:d8b2/64 scope link
       valid_lft forever preferred_lft forever
160: eth0@if161: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
    link/ether 02:e3:d7:4e:0c:b0 brd ff:ff:ff:ff:ff:ff
    inet 10.233.68.2/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::1c63:35ff:fe45:22bf/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec sriov-pod-201-2 -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
9: net1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq qlen 1000
    link/ether ce:b5:7a:f8:f3:42 brd ff:ff:ff:ff:ff:ff
    inet 192.168.201.3/24 brd 192.168.201.255 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::ccb5:7aff:fef8:f342/64 scope link
       valid_lft forever preferred_lft forever
39: eth0@if40: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
    link/ether 02:3b:e5:01:65:8a brd ff:ff:ff:ff:ff:ff
    inet 10.233.69.0/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42f:bcff:fe89:3bcf/64 scope link
       valid_lft forever preferred_lft forever

kubectl exec sriov-pod-201-3 -- ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
```

```
       valid_lft forever preferred_lft forever
13: net1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq qlen 1000
    link/ether de:92:c6:6e:05:23 brd ff:ff:ff:ff:ff:ff
    inet 192.168.201.4/24 brd 192.168.201.255 scope global net1
       valid_lft forever preferred_lft forever
    inet6 fe80::dc92:c6ff:fe6e:523/64 scope link
       valid_lft forever preferred_lft forever
39: eth0@if40: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
    link/ether 02:94:9b:80:e7:c9 brd ff:ff:ff:ff:ff:ff
    inet 10.233.67.0/18 brd 10.233.127.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::4c9a:48ff:fe38:fdce/64 scope link
       valid_lft forever preferred_lft forever
```

SRIOV VF attachment Verification on Worker Nodes

The SRIOV CNI will not only attach the VFs with the K8s Pods but will also dynamically configure the VLAN ID over the corresponding VF if a VLAN ID was referred to in the NAD file:

```
ssh worker1 sudo ip link show eno2
5: eno2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP
mode DEFAULT group default qlen 1000
    link/ether bc:30:5b:f2:87:52 brd ff:ff:ff:ff:ff:ff
    vf 0     link/ether 06:53:76:16:d8:b2 brd ff:ff:ff:ff:ff:ff, vlan
201, spoof checking on, link-state auto, trust off, query_rss off
    vf 1     link/ether 7e:46:6e:7a:24:9b brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 2     link/ether 46:06:e8:b6:87:07 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 3     link/ether d2:6d:32:2a:b2:bb brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 4     link/ether 66:e6:da:2c:8d:a5 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 5     link/ether d6:1b:ce:7c:b5:eb brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 6     link/ether 1e:67:4e:c8:72:99 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 7     link/ether e6:4c:e8:52:9a:57 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off


ssh worker2 sudo ip link show eno2
5: eno2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP
mode DEFAULT group default qlen 1000
```

```
    link/ether bc:30:5b:f2:3f:72 brd ff:ff:ff:ff:ff:ff
    vf 0     link/ether d2:73:b6:a8:61:83 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 1     link/ether 0e:7f:d4:47:e0:67 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 2     link/ether ce:b5:7a:f8:f3:42 brd ff:ff:ff:ff:ff:ff, vlan
201, spoof checking on, link-state auto, trust off, query_rss off
    vf 3     link/ether c6:c1:fd:61:5e:c3 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 4     link/ether be:39:54:bd:3f:87 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 5     link/ether 7e:f2:94:74:43:b1 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 6     link/ether 5a:d8:27:b1:56:62 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 7     link/ether 0a:58:d2:30:22:61 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off


ssh worker3 sudo ip link show eno2
5: eno2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP
mode DEFAULT group default qlen 1000
    link/ether bc:30:5b:f1:c2:02 brd ff:ff:ff:ff:ff:ff
    vf 0     link/ether f6:64:24:0c:9d:c8 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 1     link/ether b6:1b:56:f8:17:01 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 2     link/ether 0a:45:14:e0:cd:76 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 3     link/ether 56:2a:45:5d:22:2c brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 4     link/ether 8e:45:ae:9f:a0:29 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 5     link/ether 5a:03:b5:b3:88:9d brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
    vf 6     link/ether de:92:c6:6e:05:23 brd ff:ff:ff:ff:ff:ff, vlan
201, spoof checking on, link-state auto, trust off, query_rss off
    vf 7     link/ether 82:fa:a2:f0:8d:78 brd ff:ff:ff:ff:ff:ff, spoof
checking on, link-state auto, trust off, query_rss off
```

End-to-End Connectivity Verification

Send an ICMP ping toward VLAN-201 (subnet 192.168.201.0/24 gateway, therefore 192.168.201.1) from each Pod:

```
kubectl exec sriov-pod-201-1 -- ping 192.168.201.1 -c1
PING 192.168.201.1 (192.168.201.1): 56 data bytes
```

```
64 bytes from 192.168.201.1: seq=0 ttl=64 time=9.151 ms

--- 192.168.201.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 9.151/9.151/9.151 ms

kubectl exec sriov-pod-201-2 -- ping 192.168.201.1 -c1
PING 192.168.201.1 (192.168.201.1): 56 data bytes
64 bytes from 192.168.201.1: seq=0 ttl=64 time=7.637 ms

--- 192.168.201.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 7.637/7.637/7.637 ms

kubectl exec sriov-pod-201-3 -- ping 192.168.201.1 -c1
PING 192.168.201.1 (192.168.201.1): 56 data bytes
64 bytes from 192.168.201.1: seq=0 ttl=64 time=10.007 ms

--- 192.168.201.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 10.007/10.007/10.007 ms
```

Check the MAC and ARP tables of network switches to verify if the Pods MACs and IPs for VLAN-201 and subnet 192.168.201.0/24 are learned on the switch or not:

```
show ethernet-switching table vlan SRIOV_201
Ethernet-switching table: 4 unicast entries
  VLAN              MAC address       Type         Age Interfaces
  SRIOV_201          *                 Flood          - All-members
  SRIOV_201          06:53:76:16:d8:b2 Learn       2:48 ge-0/0/9.0
  SRIOV_201          ac:4b:c8:2b:77:c1 Static         - Router
  SRIOV_201          ce:b5:7a:f8:f3:42 Learn       2:39 ge-0/0/1.0
  SRIOV_201          de:92:c6:6e:05:23 Learn       2:17 ge-0/0/19.0

show arp no-resolve | match 201.
56:7f:f1:5e:ac:52 192.168.201.2   vlan.201            none
0a:43:80:e7:80:15 192.168.201.3   vlan.201            none
a2:1f:bc:be:1c:7e 192.168.201.4   vlan.201            none
```

Chapter 9

Cloud-Native Persistent Storage for Cloud-Native Workloads

This chapter describes the requirements for cloud-native persistent storage and its implementation via ROOK (a K8s Operator).

Persistent Storage Use Case in K8s Cluster

Cloud-native persistent storage capabilities must be added to the K8s cluster so that CNF application data can survive the failure of a worker node. Application data storage should be accessible in case a worker node is broken, or an application or container is broken. It implies that data storage should be central so that failure of any component should not impact its availability, but data storage should also be distributed as well for robust access and to achieve resiliency. To provide persistent storage to workloads in case of worker node failures (Ceph (https://ceph.io/en/discover/technology/) is the first choice. Ceph is Software Defined Storagealready widely deployed in Telco Cloud solutions.

ROOK K8s Operator Introduction

ROOK is a K8s operator (https://github.com/rook/rook)which provides a very easy methodology to deploy Ceph in K8s Clusters.

Deployment Workflow

- Any prerequisites.
- Identify spare HDD in K8s worker nodes.
- Deploy ceph-operator.
- Create Ceph Shared file system.
- Create Storage Class (SC).
- Create Persistent Volume Claim (PVC).
- Create a Pod with CephFS backed PVC.

Prerequisite

It is assumed that the K8s cluster is running.

Verify Spare HDD Availability in K8s worker Nodes

In bare metal worker nodes, add the secondary HDD by following the hardware addition and replacement procedure from the respective vendor. If the K8s worker nodes are virtual machines, then the following sequence can be followed by adding a secondary HDD in the worker nodes:

```
sudo virsh list | grep worker
12    worker1                        running
13    worker2                        running
14    worker3                        running
Login to worker nodes and verify HDD
lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sr0     11:0    1  368K  0 rom
vda    253:0    0  100G  0 disk
`-vda1 253:1    0  100G  0 part /
```

Create the HDD and mount to VM-based K8s worker nodes:

```
for domain in worker{1..3}; do
sudo virsh vol-create-as images ${domain}-disk-2.qcow2 50G
done

for domain in worker{1..3}; do
sudo virsh attach-disk --domain ${domain}  --source /var/lib/libvirt/
images/${domain}-disk-2.qcow2      --persistent --target vdb;
done
```

Verify the second HDD in worker nodes. For VMs it would appear as vdX and for bare metal worker nodes it would appear as sdX:

```
lsblk
  NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
  sr0     11:0    1  368K  0 rom
  vda    253:0    0  100G  0 disk
  `-vda1 253:1    0  100G  0 part /
  vdb    253:16   0   50G  0 disk

 lsblk
  NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
  sr0     11:0    1  368K  0 rom
  sda    253:0    0  100G  0 disk
  `-sda1 253:1    0  100G  0 part /
  sdb    253:16   0   50G  0 disk
```

Deploy ROOK Operator

Clone the ROOK Git repo and amend the relevant files as per your setup:

```
git clone --single-branch --branch v1.10.4 https://github.com/rook/rook.
git
 cd rook/deploy/examples
 kubectl create -f crds.yaml -f common.yaml -f operator.yaml
```

Verify if ceph-operator is deployed.

Do not proceed until the following are completed:

```
kubectl get all -n rook-ceph | grep 'ceph-operator'
NAME                                                   READY   STATUS
RESTARTS   AGE
pod/rook-ceph-operator-758ddc869f-5lk94                1/1     Running
0            2m


NAME                                                  READY   UP-TO-DATE
AVAILABLE   AGE
deployment.apps/rook-ceph-operator                     1/1     1
1            2m


NAME                                                   DESIRED   CURRENT
READY      AGE
replicaset.apps/rook-ceph-operator-758ddc869f          1         1
1            2m
```

Create the Ceph Cluster on K8s:

```
kubectl config set-context --current --namespace rook-ceph
 Context "kubernetes-admin@cluster.local" modified.
```

Amend the cluster cluster.yaml file to match your setup or leave it as it is if you want to deploy Ceph on all worker nodes:

```
vim ~/rook/deploy/examples/cluster.yaml


storage: # cluster level storage configuration and selection
  useAllNodes: false
  useAllDevices: false
  #deviceFilter:
  nodes:
  - name: "node2"
    devices: # specific devices to use for storage can be specified for
```

```
each node
    - name: "sdb"
  - name: "node3"
    devices:
    - name: "sdb"
  - name: "node4"
    devices:
    - name: "sdb"
kubectl apply -f cluster.yaml
```

Wait for the Ceph deployment:

```
kubectl get pods -n rook-ceph --watch
Wait until all pods are in running state
```

Verify if the Ceph Pods are deployed on the K8s worker nodes:

```
kubectl get -n rook-ceph jobs.batch
NAME                           COMPLETIONS   DURATION   AGE
rook-ceph-osd-prepare-node2    1/1           6s         170m
rook-ceph-osd-prepare-node3    1/1           22s        170m
rook-ceph-osd-prepare-node4    1/1           20s        170m

kubectl describe job.batch rook-ceph-osd-prepare-node2
Name:              rook-ceph-osd-prepare-node2
Namespace:         rook-ceph
Selector:          controller-uid=2fcbcbf8-990f-4632-81f2-326199d0755c
Labels:            app=rook-ceph-osd-prepare
                ceph-version=16.2.9-0
                rook-version=v1.8.10
                rook_cluster=rook-ceph
Annotations:       batch.kubernetes.io/job-tracking:
Controlled By:     CephCluster/rook-ceph
Parallelism:       1
Completions:       1
Completion Mode:   NonIndexed
Start Time:        Mon, 24 Oct 2022 07:31:24 +0000
Completed At:      Mon, 24 Oct 2022 07:31:30 +0000
Duration:          6s
Pods Statuses:     0 Active / 1 Succeeded / 0 Failed
Pod Template:
```

```
Labels:           app=rook-ceph-osd-prepare
                  ceph.rook.io/pvc=
                  controller-uid=2fcbcbf8-990f-4632-81f2-326199d0755c
                  job-name=rook-ceph-osd-prepare-node2
                  rook_cluster=rook-ceph
Service Account:  rook-ceph-osd

kubectl -n rook-ceph get cephcluster
NAME         DATADIRHOSTPATH   MONCOUNT   AGE   PHASE   MESSAGE
HEALTH       EXTERNAL
rook-ceph    /var/lib/rook     3          21h   Ready   Cluster created
successfully   HEALTH_OK
```

Deploy the Ceph toolbox to interact with Ceph directly:

```
kubectl apply -f ~/rook/deploy/examples/toolbox.yaml
```

Wait until the toolbox is deployed. Log in to the toolbox Pod and verify Ceph status:

```
kubectl -n rook-ceph exec -it deploy/rook-ceph-tools -- bash
[rook@rook-ceph-tools-6f7bb4b67-gvvrt /]$ceph status
cluster:
  id:     f1fbdc34-b88e-4187-bf5c-028777547eab
  health: HEALTH_OK

services:
  mon: 3 daemons, quorum a,b,c (age 82m)
  mgr: a(active, since 81m)
  osd: 3 osds: 3 up (since 80m), 3 in (since 80m)

data:
  pools:   1 pools, 1 pgs
  objects: 0 objects, 0 B
  usage:   15 MiB used, 150 GiB / 150 GiB avail
  pgs:     1 active+clean

[rook@rook-ceph-tools-6f7bb4b67-gvvrt /]$ ceph osd status
 ID  HOST    USED  AVAIL  WR OPS  WR DATA  RD OPS  RD DATA  STATE
 0   node4  5160k  49.9G     0        0       0        0   exists,up
 1   node2  5096k  49.9G     0        0       0        0   exists,up
 2   node3  5096k  49.9G     0        0       0        0   exists,up
```

```
[rook@rook-ceph-tools-6f7bb4b67-gvvrt /]$ ceph df
--- RAW STORAGE ---
CLASS     SIZE    AVAIL    USED  RAW USED  %RAW USED
hdd    150 GiB  150 GiB  15 MiB    15 MiB          0
TOTAL  150 GiB  150 GiB  15 MiB    15 MiB          0

--- POOLS ---
POOL                   ID  PGS  STORED  OBJECTS  USED  %USED  MAX AVAIL
device_health_metrics   1    1     0 B        0   0 B      0     47 GiB
[rook@rook-ceph-tools-6f7bb4b67-gvvrt /]$ rados df
POOL_NAME                USED  OBJECTS  CLONES  COPIES  MISSING_ON_PRIMARY
UNFOUND  DEGRADED  RD_OPS   RD  WR_OPS   WR  USED COMPR  UNDER COMPR
device_health_metrics   0 B        0       0       0                    0
0          0         0   0 B       0   0 B         0 B           0 B

total_objects    0
total_used       15 MiB
total_avail      150 GiB
total_space      150 GiB

[rook@rook-ceph-tools-6f7bb4b67-gvvrt /]$ ceph fs ls
No filesystems enabled
```

Create Ceph Shared File System

With ROOK you can either deploy a shared file system, block storage, or object storage as per use case requirements:

- Shared file system (https://rook.io/docs/rook/v1.7/ceph-filesystem.html)
- Block storage (https://rook.io/docs/rook/v1.7/ceph-block.html)
- Object storage (https://rook.io/docs/rook/v1.7/ceph-object.html)

In this chapter we are only covering how to create a shared file system and how to use it. See also (https://rook.io/docs/rook/v1.10/Storage-Configuration/Shared-Filesystem-CephFS/filesystem-storage/#quotas).

Create the Ceph Filesystem:

```
vim ~/rook/deploy/examples/filesystem.yaml
apiVersion: ceph.rook.io/v1
kind: CephFilesystem
metadata:
  name: K8sfs #Name changed from myfs to K8sfs
```

```
kubectl apply -f  ~/rook/deploy/examples/filesystem.yaml
```

Log in to the Ceph toolbox and verify that metadata and data pools are created:

```
kubectl -n rook-ceph exec -it deploy/rook-ceph-tools -- bash
[rook@rook-ceph-tools-6f7bb4b67-gvvrt /]$ ceph fs ls
name: K8sfs, metadata pool: K8sfs-metadata, data pools: [K8sfs-replicated
]

[rook@rook-ceph-tools-6f7bb4b67-gvvrt /]$ ceph osd lspools
1 device_health_metrics
2 K8sfs-metadata
3 K8sfs-replicated
```

Create K8s StorageClass (SC)

Provision Storage using Rook. See also (https://rook.io/docs/rook/v1.10/Storage-Configuration/Shared-Filesystem-CephFS/filesystem-storage/#provision-storage).

```
vim ~/rook/deploy/examples/csi/cephfs/storageclass.yaml

fsName: K8sfs #changed fsName from myfs to K8sf
pool: K8sfs-replicated #changed pool name from myfs-replicated to K8sfs-
replicated

kubectl apply -f ~/rook/deploy/examples/csi/cephfs/storageclass.yaml

kubectl get sc
NAME              PROVISIONER                     RECLAIMPOLICY
VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
rook-cephfs       rook-ceph.cephfs.csi.ceph.com   Delete
Immediate           true                   20h
```

Create Persistent Volume Claim

Persistent Volume (PV) will be created dynamically:

```
kubectl apply -f  ~/rook/deploy/examples/csi/cephfs/pvc.yaml

kubectl get pvc
NAME             STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS      AGE
cephfs-pvc       Bound    pvc-1816c01a-a05a-4a49-bad0-2c19db47c0ea   1Gi
RWO            rook-cephfs       19h
```

```
kubectl get pv

NAME                                       CAPACITY   ACCESS MODES
RECLAIM POLICY   STATUS   CLAIM                        STORAGECLASS
REASON   AGE

pvc-1816c01a-a05a-4a49-bad0-2c19db47c0ea   1Gi        RWO
Delete           Bound    rook-ceph/cephfs-pvc         rook-cephfs
19h
```

PersistentVolume (PV) is dynamically provisioned over storage class. Create the Pod by referring to the above defined PresistentVolumeClaim (PVC).

Create Pod with CephFS Backed PVC

```
vim ~/rook/deploy/examples/csi/cephfs/pod.yaml


kubectl create -f  ~/rook/deploy/examples/csi/cephfs/pod.yaml


[contrail@deployer examples]$ kubectl get pods csicephfs-demo-pod

NAME                 READY   STATUS    RESTARTS   AGE

csicephfs-demo-pod   1/1     Running   0          19h


[contrail@deployer examples]$ kubectl describe pods csicephfs-demo-pod |
grep Volumes -A4

 Volumes:

   mypvc:

     Type:       PersistentVolumeClaim (a reference to a
PersistentVolumeClaim in the same namespace)

     ClaimName:  cephfs-pvc

    ReadOnly:   false
```

Chapter 10

Virtual Machines as Cloud-Native workload

This chapter describes VM use cases in K8s and their implementation via kubevirt.

Virtual Machine Use Case in K8s Cluster

Although a majority of workloads are containerized the need for Virtual Machines cannot be ruled out. Kubevirt is a K8s operator which provides the capability to create and manage Virtual Machines in a K8s Cluster. Juniper CN2 has an enhanced Kubevirt Operator by adding support for Data Plane Development Kit (DPDK).

For more see these links: What is Kubevirt and CN2_Kubevirt_DPDK.

Kubevirt Operator Deployment

```
Kubevirt v0.58.0 (current) is tested in this setup. Download the virtctl
binary:

export VERSION=v0.58.0

wget https://github.com/kubevirt/kubevirt/releases/download/${VERSION}/
virtctl-${VERSION}-linux-amd64


sudo mv virtctl-v0.58.0-linux-amd64 /usr/local/bin/virtctl

sudo chmod +x /usr/local/bin/virtctl
```

Download required container images, re-tag, and push those images in local containers images registry:

```
sudo wget -O /tmp/virt-api_v0.58.0-jnpr.tar.gz https://github.com/
Juniper/kubevirt/releases/download/v0.58.0-jnpr/virt-api_v0.58.0-jnpr.
tar.gz

sudo wget -O /tmp/virt-controller_v0.58.0-jnpr.tar.gz https://github.com/
Juniper/kubevirt/releases/download/v0.58.0-jnpr/virt-controller_v0.58.0-
jnpr.tar.gz

sudo wget -O /tmp/virt-handler_v0.58.0-jnpr.tar.gz https://github.com/
Juniper/kubevirt/releases/download/v0.58.0-jnpr/virt-handler_v0.58.0-
jnpr.tar.gz

sudo wget -O /tmp/virt-launcher_v0.58.0-jnpr.tar.gz https://github.com/
Juniper/kubevirt/releases/download/v0.58.0-jnpr/virt-launcher_v0.58.0-
jnpr.tar.gz

sudo wget -O /tmp/virt-operator_v0.58.0-jnpr.tar.gz https://github.com/
Juniper/kubevirt/releases/download/v0.58.0-jnpr/virt-operator_v0.58.0-
jnpr.tar.gz


sudo docker load < /tmp/virt-api_v0.58.0-jnpr.tar.gz
```

```
sudo docker load < /tmp/virt-controller_v0.58.0-jnpr.tar.gz
sudo docker load < /tmp/virt-handler_v0.58.0-jnpr.tar.gz
sudo docker load < /tmp/virt-launcher_v0.58.0-jnpr.tar.gz
sudo docker load < /tmp/virt-operator_v0.58.0-jnpr.tar.gz


sudo docker tag svl-artifactory.juniper.net/atom-docker/kubevirt/virt-api:v0.58.0-jnpr deployer-node.maas:5000/virt-api:v0.58.0-jnpr
sudo docker push deployer-node.maas:5000/virt-api:v0.58.0-jnpr
sudo docker tag svl-artifactory.juniper.net/atom-docker/kubevirt/virt-controller:v0.58.0-jnpr deployer-node.maas:5000/virt-controller:v0.58.0-jnpr
sudo docker push deployer-node.maas:5000/virt-controller:v0.58.0-jnpr
sudo docker tag svl-artifactory.juniper.net/atom-docker/kubevirt/virt-handler:v0.58.0-jnpr deployer-node.maas:5000/virt-handler:v0.58.0-jnpr
sudo docker push deployer-node.maas:5000/virt-handler:v0.58.0-jnpr
sudo docker tag svl-artifactory.juniper.net/atom-docker/kubevirt/virt-launcher:v0.58.0-jnpr deployer-node.maas:5000/virt-launcher:v0.58.0-jnpr
sudo docker push deployer-node.maas:5000/virt-launcher:v0.58.0-jnpr
sudo docker tag svl-artifactory.juniper.net/atom-docker/kubevirt/virt-operator:v0.58.0-jnpr deployer-node.maas:5000/virt-operator:v0.58.0-jnpr
sudo docker push deployer-node.maas:5000/virt-operator:v0.58.0-jnpr


curl http://deployer-node.maas:5000/v2/_catalog | jq .repositories[]
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   112  100   112    0     0   5090      0 --:--:-- --:--:-- --:--:--  5333
"ubuntu-traffic"
"virt-api"
"virt-controller"
"virt-handler"
"virt-launcher"
"virt-operator"
```

Get kubevirt-operator.yaml and kubevirt-cr.yaml:

```
mkdir ~/kubevirt && cd ~/kubevirt
wget https://github.com/Juniper/kubevirt/releases/download/v0.58.0-jnpr/kubevirt-operator.yaml
wget https://github.com/Juniper/kubevirt/releases/download/v0.58.0-jnpr/kubevirt-cr.yaml
```

Modify kubevirt-operator.yaml so that it can refer to local container image registry:

```
sed -i  'sed -i  's/<INSECURE_REGISTERY>/deployer-node.maas:5000/g' kubevirt-operator.yaml

grep -rni deployer-node.maas:5000 kubevirt-operator.yaml
6578:          value: deployer-node.maas:5000/virt-operator@sha256:2280b3277cbbc3c51be3fee77204baa8d45190cf9aac8bf8be87fcec8c1327e5
6599:         image:  deployer-node.maas:5000/virt-operator@sha256:2280b3277cbbc3c51be3fee77204baa8d45190cf9aac8bf8be87fcec8c1327e5
```

Deploy the kubevirt operator:

```
kubectl create -f kubevirt-operator.yaml

namespace/kubevirt created
customresourcedefinition.apiextensions.K8s.io/kubevirts.kubevirt.io created
priorityclass.scheduling.K8s.io/kubevirt-cluster-critical created
clusterrole.rbac.authorization.K8s.io/kubevirt.io:operator created
serviceaccount/kubevirt-operator created
role.rbac.authorization.K8s.io/kubevirt-operator created
rolebinding.rbac.authorization.K8s.io/kubevirt-operator-rolebinding created
clusterrole.rbac.authorization.K8s.io/kubevirt-operator created
clusterrolebinding.rbac.authorization.K8s.io/kubevirt-operator created
deployment.apps/virt-operator created

kubectl create -f kubevirt-cr.yaml

kubevirt.kubevirt.io/kubevirt created
```

Verify Kubevirt deployment:

```
kubectl get pods -n kubevirt
NAME                               READY   STATUS    RESTARTS   AGE
virt-api-78d49589b4-n6wwl          1/1     Running   0          105s
virt-api-78d49589b4-zl5l6          1/1     Running   0          105s
virt-controller-5c489cf7cc-j75pg   1/1     Running   0          59s
virt-controller-5c489cf7cc-sdkdg   1/1     Running   0          59s
virt-handler-76lmj                 0/1     Running   0          59s
virt-handler-lmdvw                 0/1     Running   0          59s
virt-handler-mv7rd                 0/1     Running   0          59s
```

```
virt-operator-db5ccfdb9-g486v      1/1     Running   0          3m8s
virt-operator-db5ccfdb9-rz64m      1/1     Running   0          3m8s
```

One important part is to disable the checksum on the physical interfaces of the Master and Controller VMs, otherwise Kubevirt deployment might face some challenges:

```
kubectl get apiservices | grep kubevirt
v1.kubevirt.io                                Local
True                         11m


v1.subresources.kubevirt.io                   kubevirt/virt-api
False (FailedDiscoveryCheck)   4m38s


v1alpha1.clone.kubevirt.io                    Local
True                         4m42s


v1alpha1.export.kubevirt.io                   Local
True                         4m42s


v1alpha1.instancetype.kubevirt.io             Local
True                         4m42s


v1alpha1.migrations.kubevirt.io               Local
True                         4m42s


v1alpha1.pool.kubevirt.io                     Local
True                         4m42s


v1alpha1.snapshot.kubevirt.io                 Local
True                         4m42s


v1alpha2.instancetype.kubevirt.io             Local
True                         4m42s


v1alpha3.kubevirt.io                          Local
True                         11m


v1alpha3.subresources.kubevirt.io             kubevirt/virt-api
False (FailedDiscoveryCheck)   4m38s


for node in 1 2 3; do ssh controller${node} "sudo /sbin/ethtool -K ens4
tx-checksum-ip-generic off";done
```

```
kubectl get apiservices | grep kubevirt
v1.kubevirt.io                                   Local
True        40m
v1.subresources.kubevirt.io                      kubevirt/virt-api
True        33m
v1alpha1.clone.kubevirt.io                       Local
True        33m
v1alpha1.export.kubevirt.io                      Local
True        33m
v1alpha1.instancetype.kubevirt.io                Local
True        33m
v1alpha1.migrations.kubevirt.io                  Local
True        33m
v1alpha1.pool.kubevirt.io                        Local
True        33m
v1alpha1.snapshot.kubevirt.io                    Local
True        33m
v1alpha2.instancetype.kubevirt.io                Local
True        33m
v1alpha3.kubevirt.io                             Local
True        40m
v1alpha3.subresources.kubevirt.io                kubevirt/virt-api
True        33m
```

Custom Image to Create kubevirt VMs

Let's first create a Centos image which will be used in the next step to create VMs via kubevirt:

```
mkdir docker_images
cd docker_images
cat << END > Dockerfile
FROM scratch
ADD --chown=107:107 https://cloud.centos.org/centos/7/images/CentOS-7-
x86_64-GenericCloud.qcow2 /disk/
END

docker build -t deployer-node.maas:5000/centos7:latest .
docker push deployer-node.maas:5000/centos7:latest
```

Instantiate kubevirt VMs

Now let's create a couple of VMs on different VirtualNetworks and allow communication between VMs using the VirtualNetworkRouter:

```
sudo cat > kubevirt-cluster.yaml <<END_OF_SCRIPT
```

```
apiVersion: v1
kind: Namespace
metadata:
  labels:
    name: vm-ns
    core.juniper.net/isolated-namespace: 'true'
  name: vm-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: blue-vn
  namespace: vm-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "40.40.40.0/24"
    }'
  labels:
    vn: mesh_vnr
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "blue-vn",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: green-vn
  namespace: vm-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "50.50.50.0/24"
    }'
  labels:
    vn: mesh_vnr
spec:
```

```
  config: '{
  "cniVersion": "0.3.1",
  "name": "green-vn",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: kubevirt.io/v1alpha3
kind: VirtualMachine
metadata:
  name: vm-blue
  namespace: vm-ns
spec:
  running: true
  template:
    metadata:
      labels:
        app: vm-blue
    spec:
      domain:
        devices:
          disks:
          - disk:
              bus: virtio
            name: containerdisk
          - disk:
              bus: virtio
            name: cloudinitdisk
          interfaces:
          - name: default
            bridge: {}
          - name: blue-vn
            bridge: {}
        resources:
          requests:
            memory: 1024M
      networks:
      - name: default
        pod: {}
```

```
    - name: blue-vn
      multus:
        networkName: blue-vn
    volumes:
    - containerDisk:
        image: deployer-node.maas:5000/centos7:latest
      name: containerdisk
    - cloudInitNoCloud:
        userData: |-
          #cloud-config
          password: centos
          ssh_pwauth: True
          chpasswd: { expire: False }
      name: cloudinitdisk
---
apiVersion: kubevirt.io/v1alpha3
kind: VirtualMachine
metadata:
  name: vm-green
  namespace: vm-ns
spec:
  running: true
  template:
    metadata:
      labels:
        app: vm-green
    spec:
      domain:
        devices:
          disks:
          - disk:
              bus: virtio
            name: containerdisk
          - disk:
              bus: virtio
            name: cloudinitdisk
          interfaces:
          - name: default
```

```
            bridge: {}
          - name: green-vn
            bridge: {}
        resources:
          requests:
            memory: 1024M
      networks:
      - name: default
        pod: {}
      - name: green-vn
        multus:
          networkName: green-vn
      volumes:
      - containerDisk:
          image: deployer-node.maas:5000/centos7:latest
        name: containerdisk
      - cloudInitNoCloud:
          userData: |-
            #cloud-config
            password: centos
            ssh_pwauth: True
            chpasswd: { expire: False }
        name: cloudinitdisk
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: vm-ns
  name: vnr01
  annotations:
    core.juniper.net/display-name: vnr01
  labels:
      vnr: mesh_vnr
spec:
  type: mesh
  virtualNetworkSelector:
    matchLabels:
      vn: mesh_vnr
```

```
END_OF_SCRIPT
kubectl create -f kubevirt-cluster.yaml
```

Let's verify if the VMs are created:

```
kubectl get all -n vm-ns
NAME                                READY   STATUS    RESTARTS   AGE
pod/virt-launcher-vm-blue-hmbvs     2/2     Running   0          110m
pod/virt-launcher-vm-green-m58sm    2/2     Running   0          110m

NAME                                   AGE    STATUS    READY
virtualmachine.kubevirt.io/vm-blue     110m   Running   True
virtualmachine.kubevirt.io/vm-green    110m   Running   True

NAME                                           AGE    PHASE     IP
NODENAME   READY
virtualmachineinstance.kubevirt.io/vm-blue     110m   Running
10.233.68.4   worker1    True
virtualmachineinstance.kubevirt.io/vm-green    110m   Running
10.233.69.3   worker3    True
```

Now let's ensure communication is allowed between the VMs created on the separate network; the username and password is Centos:

```
virtctl ssh centos@vn-blue -n vm-ns
[centos@vm-blue ~]$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
    link/ether 02:fb:ac:e0:fb:58 brd ff:ff:ff:ff:ff:ff
    inet 10.233.68.4/18 brd 10.233.127.255 scope global dynamic eth0
       valid_lft 86313151sec preferred_lft 86313151sec
    inet6 fe80::fb:acff:fee0:fb58/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group
default qlen 1000
    link/ether 02:1c:18:db:14:a3 brd ff:ff:ff:ff:ff:ff
```

```
[centos@vm-blue ~]$ sudo ip link set up eth1
[centos@vm-blue ~]$ sudo dhclient eth1
[centos@vm-blue ~]$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
    link/ether 02:fb:ac:e0:fb:58 brd ff:ff:ff:ff:ff:ff
    inet 10.233.68.4/18 brd 10.233.127.255 scope global dynamic eth0
       valid_lft 86313123sec preferred_lft 86313123sec
    inet6 fe80::fb:acff:fee0:fb58/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
    link/ether 02:1c:18:db:14:a3 brd ff:ff:ff:ff:ff:ff
    inet 40.40.40.2/24 brd 40.40.40.255 scope global dynamic eth1
       valid_lft 86313599sec preferred_lft 86313599sec
    inet6 fe80::1c:18ff:fedb:14a3/64 scope link
       valid_lft forever preferred_lft forever
sudo ip r add 50.50.50.0/24 dev eth1 via 40.40.40.1
```

```
[centos@vm-green ~]$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
    link/ether 02:0f:95:84:ae:49 brd ff:ff:ff:ff:ff:ff
    inet 10.233.69.3/18 brd 10.233.127.255 scope global dynamic eth0
       valid_lft 86306554sec preferred_lft 86306554sec
```

```
    inet6 fe80::f:95ff:fe84:ae49/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast state DOWN group
default qlen 1000
    link/ether 02:4f:78:70:33:0f brd ff:ff:ff:ff:ff:ff
sudo ip link set up eth1
sudo dhclient eth1

[centos@vm-green ~]$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
    link/ether 02:0f:95:84:ae:49 brd ff:ff:ff:ff:ff:ff
    inet 10.233.69.3/18 brd 10.233.127.255 scope global dynamic eth0
       valid_lft 86306574sec preferred_lft 86306574sec
    inet6 fe80::f:95ff:fe84:ae49/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
UP group default qlen 1000
    link/ether 02:4f:78:70:33:0f brd ff:ff:ff:ff:ff:ff
    inet 50.50.50.2/24 brd 50.50.50.255 scope global dynamic eth1
       valid_lft 86306685sec preferred_lft 86306685sec
    inet6 fe80::4f:78ff:fe70:330f/64 scope link
       valid_lft forever preferred_lft forever

sudo ip r add 40.40.40.0/24 dev eth1 via 50.50.50.1

[centos@vm-green ~]$ ping 40.40.40.2 -c1
PING 40.40.40.2 (40.40.40.2) 56(84) bytes of data.
64 bytes from 40.40.40.2: icmp_seq=1 ttl=64 time=2.47 ms

--- 40.40.40.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.475/2.475/2.475/0.000 ms
```

Chapter 11

Real World Use Cases and Implementation in CN2

This chapter covers two use cases, Micro Segmentation Simulation in IT Cloud environment and LTE Traffic Flows Simulation in telco cloud environment, both using CN2 constructs.

Micro Segmentation Simulation

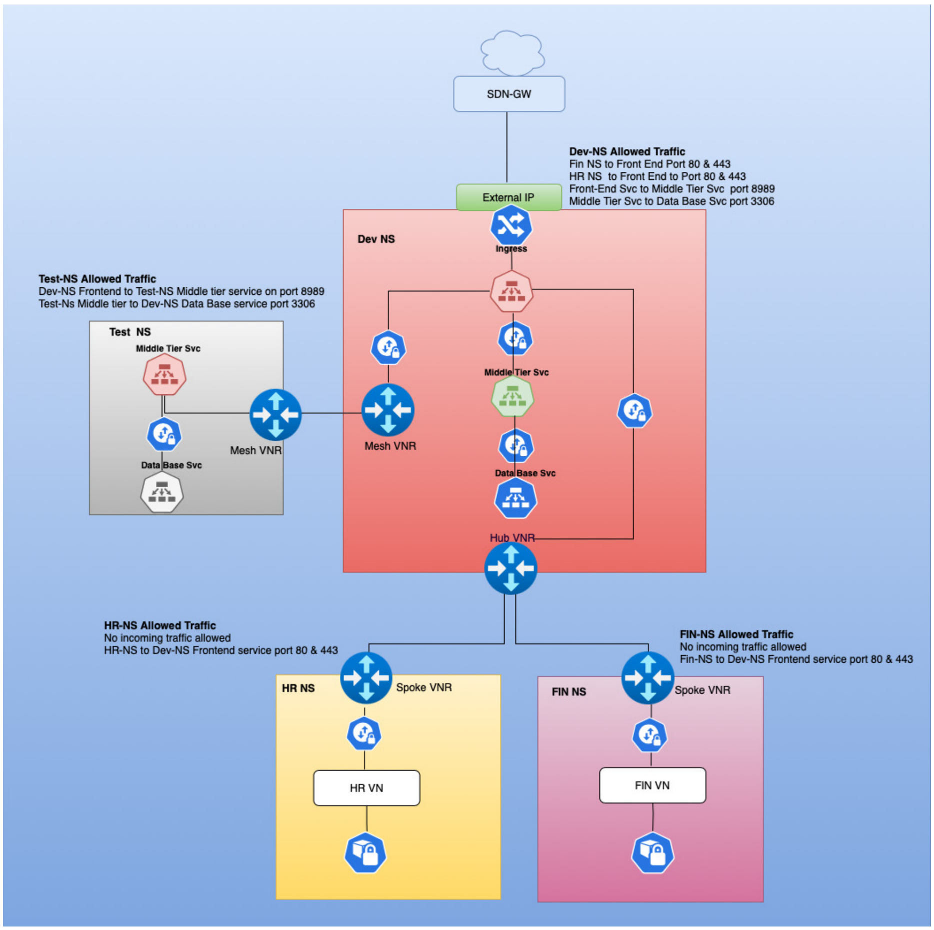


Figure 29 Micro Segmentation Use Case

Use Case Description

In this use case we will demonstrate capabilities of CN2 to implement micro segmentation in IT Cloud enterprise environment. A 3-tier web application will be deployed in the dev name space (dev-ns) and the frontend web application will be extended to the SDN GW by using Ingress and an external IP. Clients should be able to access Ingress-backed web service. Within dev-ns the frontend service should be able to access the middle tier application on port 8989 and the middle tier service should be able to access database service on port 3306. Pods deployed in HR (hr-ns) and Fin (fin-ns) name spaces should only be able to access the dev-ns frontend application on port 80 and no communication is allowed between fin-ns and hr-ns. Moreover, any incoming traffic to hr-ns and fin-ns from dev-ns is not allowed.

In the test namespace (test-ns) the middle tier and database services are deployed and dev-ns frontend-svc should only be able to access middle tier service in test-ns on port 8989. Inside test-ns, middle tier service should be able to access database service on port 3306. Besides the above described flows, no other flow is allowed. All Pods will be deployed using custom Pod networks instead of default Pod-Network. Wherever inter virtual networks traffic is required we will use Virtual Network Routers (VNRs) but strict traffic compliance rules will also be applied using Contrail Security Policies (CSP).

VNRs Description

Dev_ns_hub_vnr_hr_ns_fin_ns_spokes_vnr.yaml will create the following VNRs:

- Hub VNR in dev-ns attached to the Default-Service Network.
- Spoke VNR in fin-ns attached with custom Pod Network.
- Spoke VNR in hr-ns attached with cthe ustom Pod Network. Dev_ns_test_ns_mesh_vnr.yaml will create the following VNRs.
- Mesh VNR in dev-ns attached with the custom Pod Network.
- Mesh VNI in test-ns attached with the Default Service Network.

Contrail Security Polices Description

Dev-ns-csp.yaml will configure a CSP in dev-ns with the following rules: -

- Allows incoming traffic from cidr (192.168.5.0/24) to depict incoming traffic from internet) on port 80, therefore the vRouter subnet, as incoming traffic from the SDN-GW that will hit Ingress. Ingress will re-direct that traffic to one of Pods backed by the frontend-dev service. During that process the source IP will be changed to the vRouter IP of worker node where corresponding Pod is running..
- Allows incoming traffic from hr-ns, fin-ns destined to the frontend-dev service on port 80.
- Allows incoming traffic from fin-ns destined to the frontend-dev service on port 80.

- Allows traffic from the frontend-dev service to middletier-dev service on port 8989.
- Allows the middletier-dev service to backend-dev service on port 3306.
- Allows outgoing traffic from dev-ns (frontend-dev service/ Pods) to test-ns (middletier-test service) on port 8989.

Hr-ns-csp.yaml will configure a CSP in hr-ns with following rules: -

- Allows outgoing traffic from hr-ns destined to the dev-ns frontend-dev service on port 80.
- Block any incoming traffic to hr-ns.

Fin-ns-csp.yaml will configure a CSP in fin-ns with the following rules:-

- Allow outgoing traffic from fin-ns destined to the dev-ns frontend-dev service on port 80.
- Block any incoming traffic to fin-ns.
- Test-ns-csp.yaml will configure a CSP in with following rules:-
- Allow incoming traffic from dev-ns (the frontend-dev service and Pods) to test-ns (the frontend-test service) on port 8989.
- Allow traffic from the frontend-test service to the backend-test service on port 3306.
- Block any other flow.

Components Used

To achieve these use cases, the following CN2 constructs will be used: -

- Custom Pod-Network
- VNR between Custom Pod-Network and Default Service-Network
- Cross Namespace Hub & Spoke VNR
- Cross Namespace Mesh VNR
- Contrail Security Policy (CSP)
- Ingress
- SDN GW L3 Connectivity

Definition file to create name spaces, Pods, Services, and Ingress:

```
sudo cat > ingress-custom-pod-network.yaml <<END_OF_SCRIPT
---
apiVersion: v1
kind: Namespace
metadata:
```

```
  name: dev-ns
  labels:
    name: dev-ns
---
apiVersion: “K8s.cni.cncf.io/v1”
kind: NetworkAttachmentDefinition
metadata:
  name: dev-vn
  namespace: dev-ns
  labels:
    vn: dev-vn
  annotations:
    juniper.net/networks: ‘{
      “ipamV4Subnet”: “10.10.10.0/24”,
      “podNetwork”: true
    }’
spec:
  config: ‘{
  “cniVersion”: “0.3.1”,
  “name”: “dev-vn”,
  “type”: “contrail-K8s-cni”
}’
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-1
  namespace: dev-ns
  labels:
    app: frontend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: frontend-dev-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        [“bash”, “-c”, “python3 -m http.server 80 --directory /tmp/”]
```

```
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker4
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend-dev-2
  namespace: dev-ns
  labels:
    app: frontend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: frontend-dev-2
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 80 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker5
---
apiVersion: v1
kind: Service
metadata:
  name: frontend-dev
  namespace: dev-ns
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  ports:
```

```
  - name: port-443
    targetPort: 443
    protocol: TCP
    port: 443
  - name: port-80
    targetPort: 80
    protocol: TCP
    port: 80
  selector:
    app: frontend-dev
---
apiVersion: v1
kind: Pod
metadata:
  name: middleware-dev-1
  namespace: dev-ns
  labels:
    app: middleware-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: middleware-dev-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 8989 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker4
---
apiVersion: v1
kind: Pod
metadata:
  name: middleware-dev-2
  namespace: dev-ns
```

```
  labels:
    app: middleware-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: middleware-dev-2
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 8989 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker5
---
apiVersion: v1
kind: Service
metadata:
  name: middleware-dev
  namespace: dev-ns
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  ports:
  - name: port-8989
    targetPort: 8989
    protocol: TCP
    port: 8989
  selector:
    app: middleware-dev
  type: ClusterIP
---
apiVersion: v1
kind: Pod
metadata:
  name: backend-dev-1
```

```
  namespace: dev-ns
  labels:
    app: backend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: backend-dev-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 3306 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker4
---
apiVersion: v1
kind: Pod
metadata:
  name: backend-dev-2
  namespace: dev-ns
  labels:
    app: backend-dev
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  containers:
    - name: backend-dev-2
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 3306 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
```

```
  nodeName: worker5
---
apiVersion: v1
kind: Service
metadata:
  name: backend-dev
  namespace: dev-ns
  annotations:
    net.juniper.contrail.podnetwork: dev-ns/dev-vn
spec:
  ports:
  - name: port-3306
    targetPort: 3306
    protocol: TCP
    port: 3306
  selector:
    app: backend-dev
  type: ClusterIP
---
apiVersion: networking.K8s.io/v1
kind: Ingress
metadata:
  name: hello-kubernetes-ingress
  namespace: dev-ns
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
  - host: "frontend.dev.svc"
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: frontend-dev
            port:
              number: 80
```

```
---
apiVersion: v1
kind: Namespace
metadata:
  labels:
    name: hr-ns
  name: hr-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: hr-vn
  namespace: hr-ns
  labels:
    vngroup: spoke
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.10.20.0/24",
      "podNetwork": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "hr-vn",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: hr-1
  namespace: hr-ns
  labels:
    dept: hr
  annotations:
    net.juniper.contrail.podnetwork: hr-ns/hr-vn
spec:
  containers:
```

```
    - name: hr-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker4
---
apiVersion: v1
kind: Namespace
metadata:
  labels:
    name: fin-ns
  name: fin-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: fin-vn
  namespace: fin-ns
  labels:
    vngroup: spoke
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.10.30.0/24",
      "podNetwork": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "fin-vn",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
```

```
  name: fin-1
  namespace: fin-ns
  labels:
    dept: fin
  annotations:
    net.juniper.contrail.podnetwork: fin-ns/fin-vn
spec:
  containers:
    - name: fin-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker5
---
apiVersion: v1
kind: Namespace
metadata:
  name: test-ns
  labels:
    name: test-ns
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: test-vn
  namespace: test-ns
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "10.10.40.0/24",
      "podNetwork": true
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "test-vn",
```

```
  "type": "contrail-K8s-cni"
}'
---
apiVersion: v1
kind: Pod
metadata:
  name: middleware-test-1
  namespace: test-ns
  labels:
    app: middleware-test
  annotations:
    net.juniper.contrail.podnetwork: test-ns/test-vn
spec:
  containers:
    - name: middleware-test-1
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 8989 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker4
---
apiVersion: v1
kind: Pod
metadata:
  name: middleware-test-2
  namespace: test-ns
  labels:
    app: middleware-test
  annotations:
    net.juniper.contrail.podnetwork: test-ns/test-vn
spec:
  containers:
    - name: middleware-test-2
      image: deployer-node.maas:5000/ubuntu-traffic:latest
```

```
      command:
        ["bash", "-c", "python3 -m http.server 8989 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker5
---
apiVersion: v1
kind: Service
metadata:
  name: middleware-test
  namespace: test-ns
  annotations:
    net.juniper.contrail.podnetwork: test-ns/test-vn
spec:
  ports:
  - name: port-8989
    targetPort: 8989
    protocol: TCP
    port: 8989
  selector:
    app: middleware-test
  type: ClusterIP
---
apiVersion: v1
kind: Pod
metadata:
  name: backend-test-1
  namespace: test-ns
  labels:
    app: backend-test
  annotations:
    net.juniper.contrail.podnetwork: test-ns/test-vn
spec:
  containers:
    - name: backend-test-1
```

```
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 3306 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker4
---
apiVersion: v1
kind: Pod
metadata:
  name: backend-test-2
  namespace: test-ns
  labels:
    app: backend-test
  annotations:
    net.juniper.contrail.podnetwork: test-ns/test-vn
spec:
  containers:
    - name: backend-test-2
      image: deployer-node.maas:5000/ubuntu-traffic:latest
      command:
        ["bash", "-c", "python3 -m http.server 3306 --directory /tmp/"]
      securityContext:
        privileged: true
        capabilities:
          add:
          - NET_ADMIN
  nodeName: worker5
---
apiVersion: v1
kind: Service
metadata:
  name: backend-test
  namespace: test-ns
  annotations:
```

```
    net.juniper.contrail.podnetwork: test-ns/test-vn
spec:
  ports:
  - name: port-3306
    targetPort: 3306
    protocol: TCP
    port: 3306
  selector:
    app: backend-test
  type: ClusterIP
---
END_OF_SCRIPT
```

Okay, now let's create NAD, Pods , Services, and Ingress:

```
kubectl apply -f ingress-custom-pod-network.yaml
```

Let's create patches for the test-ns ServiceNetwork and dev-ns ServiceNetwork:

```
sudo cat > dev-svc-nw-vn-label.yaml <<END_OF_SCRIPT
metadata:
  labels:
    vngroup: hub
END_OF_SCRIPT

sudo cat > test-svc-nw-vn-label.yaml <<END_OF_SCRIPT
metadata:
  labels:
    vn: test-vn
END_OF_SCRIPT
```

Now apply patches to the test-ns ServiceNetwork and dev-ns ServiceNetwork:

```
kubectl patch vn dev-vn-servicenetwork -n dev-ns --patch-file dev-svc-nw-vn-label.yaml
kubectl patch vn test-vn-servicenetwork -n test-ns --patch-file test-svc-nw-vn-label.yaml
```

Use the definition file to create Hub and Spoke VNRs:

```
sudo cat > dev_ns_hub_vnr_hr_ns_fin_ns_spokes_vnr.yaml <<END_OF_SCRIPT
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
```

```
metadata:
  namespace: hr-ns
  name: hr-vnr-spoke
  annotations:
    core.juniper.net/display-name: hr-vnr-spoke
  labels:
    vnr: hr-spoke
spec:
  type: spoke
  virtualNetworkSelector:
    matchLabels:
      vngroup: spoke
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: hub
        namespaceSelector:
          matchLabels:
            name: dev-ns
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: fin-ns
  name: fin-vnr-spoke
  annotations:
    core.juniper.net/display-name: fin-vnr-spoke
  labels:
    vnr: fin-spoke
spec:
  type: spoke
  virtualNetworkSelector:
    matchLabels:
      vngroup: spoke
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
```

```
          matchLabels:
            vnr: hub
        namespaceSelector:
          matchLabels:
            name: dev-ns
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: dev-ns
  name: vnr-hub
  annotations:
    core.juniper.net/display-name: vnr-hub
  labels:
    vnr: hub
spec:
  type: hub
  virtualNetworkSelector:
    matchLabels:
      vngroup: hub
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: hr-spoke
        namespaceSelector:
          matchLabels:
            name: hr-ns
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: fin-spoke
        namespaceSelector:
          matchLabels:
            name: fin-ns
END_OF_SCRIPT
```

And the definition file to create Mesh VNRs:

```
sudo cat > dev_ns_test_ns_mesh_vnr.yaml <<END_OF_SCRIPT
---
```

```
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: test-ns
  name: test-vnr-mesh
  annotations:
    core.juniper.net/display-name: test-vnr-mesh
  labels:
    vnr: test-mesh
spec:
  type: mesh
  virtualNetworkSelector:
    matchLabels:
      vn: test-vn
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: dev-mesh
        namespaceSelector:
          matchLabels:
            name: dev-ns
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: dev-ns
  name: vnr-mesh
  annotations:
    core.juniper.net/display-name: vnr-mesh
  labels:
    vnr: dev-mesh
spec:
  type: mesh
  virtualNetworkSelector:
    matchLabels:
      vn: dev-vn
  import:
```

```
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: test-mesh
        namespaceSelector:
          matchLabels:
            name: test-ns
END_OF_SCRIPT
```

The definition file to create CSP in dev-ns:

```
sudo cat > dev-ns-csp.yaml <<END_OF_SCRIPT
---
apiVersion: core.contrail.juniper.net/v3
kind: ContrailSecurityPolicy
metadata:
  name: dev-ns-csp
  namespace: dev-ns
spec:
  rules:
    - srcEP:
        endPoints:
          - ipBlock:
              cidr: 192.168.5.0/24

      dstEP:
        endPoints:
          - podSelector:
              matchLabels:
                app: frontend-dev
      ports:
          - protocol: TCP
            port: 80
            endPort: 80
    - srcEP:
        endPoints:
          - namespaceSelector:
              matchLabels:
                name: hr-ns
          - podSelector:
```

```
              matchLabels:
                dept: hr
      dstEP:
        endPoints:
          - podSelector:
              matchLabels:
                app: frontend-dev
      ports:
          - protocol: TCP
            port: 80
            endPort: 80
    - srcEP:
        endPoints:
          - namespaceSelector:
              matchLabels:
                name: fin-ns
          - podSelector:
              matchLabels:
                dept: fin
      dstEP:
        endPoints:
          - podSelector:
              matchLabels:
                app: frontend-dev
      ports:
          - protocol: TCP
            port: 80
            endPort: 80
    - srcEP:
        endPoints:
          - podSelector:
              matchLabels:
                app: frontend-dev
      dstEP:
        endPoints:
          - podSelector:
              matchLabels:
                app: middleware-dev
```

```
      ports:
          - protocol: TCP
            port: 8989
            endPort: 8989
    - srcEP:
        endPoints:
          - podSelector:
              matchLabels:
                app: middleware-dev
      dstEP:
        endPoints:
          - podSelector:
              matchLabels:
                app: backend-dev
      ports:
          - protocol: TCP
            port: 3306
            endPort: 3306
    - srcEP:
        endPoints:
          - podSelector:
              matchLabels:
                app: frontend-dev
      dstEP:
        endPoints:
          - namespaceSelector:
              matchLabels:
                name: test-ns
          - podSelector:
              matchLabels:
                app: middleware-test
      ports:
          - protocol: TCP
            port: 8989
            endPort: 8989
  action: pass
END_OF_SCRIPT
---
```

And the definition file to create CSP in test-ns:

```
sudo cat > test-ns-csp.yaml <<END_OF_SCRIPT
---
apiVersion: core.contrail.juniper.net/v3
kind: ContrailSecurityPolicy
metadata:
  name: test-ns-csp
  namespace: test-ns
spec:
  rules:
    - srcEP:
        endPoints:
          - namespaceSelector:
              matchLabels:
                name: dev-ns
          - podSelector:
              matchLabels:
                app: frontend-dev
      dstEP:
        endPoints:
          - podSelector:
              matchLabels:
                app: middleware-test
      ports:
          - protocol: TCP
            port: 8989
            endPort: 8989
    - srcEP:
        endPoints:
          - podSelector:
              matchLabels:
                app: middleware-test
      dstEP:
        endPoints:
          - podSelector:
              matchLabels:
                app: backend-test
      ports:
```

```
          - protocol: TCP
            port: 3306
            endPort: 3306
  action: pass
---
END_OF_SCRIPT
```

The definition file to create CSP in hr-ns:

```
sudo cat > hr-ns-csp.yaml <<END_OF_SCRIPT
---
apiVersion: core.contrail.juniper.net/v3
kind: ContrailSecurityPolicy
metadata:
  name: hr-ns-csp
  namespace: hr-ns
spec:
  rules:
    - srcEP:
        endPoints:
          - podSelector:
              matchLabels:
                dept: hr

      dstEP:
        endPoints:
          - namespaceSelector:
              matchLabels:
                name: dev-ns
          - podSelector:
              matchLabels:
                app: frontend-dev
      ports:
          - protocol: TCP
            port: 80
            endPort: 80
  action: pass
---
END_OF_SCRIPT
```

The definition file to create CSP in fin-ns:

```
sudo cat > fin-ns-csp.yaml <<END_OF_SCRIPT
---
apiVersion: core.contrail.juniper.net/v3
kind: ContrailSecurityPolicy
metadata:
  name: fin-ns-csp
  namespace: fin-ns
spec:
  rules:
    - srcEP:
        endPoints:
          - podSelector:
              matchLabels:
                dept: fin

      dstEP:
        endPoints:
          - namespaceSelector:
              matchLabels:
                name: dev-ns
          - podSelector:
              matchLabels:
                app: frontend-dev
      ports:
          - protocol: TCP
            port: 80
            endPort: 80
  action: pass
---
END_OF_SCRIPT
```

Okay! Let's create the VNRs:

```
kubectl apply -f dev_ns_test_ns_mesh_vnr.yaml
kubectl apply -f dev_ns_hub_vnr_hr_ns_fin_ns_spokes_vnr.yaml

kubectl get all -n dev-ns -o wide
NAME                       READY    STATUS    RESTARTS    AGE    IP
```

```
NODE      NOMINATED NODE   READINESS GATES
pod/backend-dev-1      1/1     Running   0          24h   10.10.10.5
worker4   <none>           <none>
pod/backend-dev-2      1/1     Running   0          24h   10.10.10.7
worker5   <none>           <none>
pod/frontend-dev-1     1/1     Running   0          24h   10.10.10.3
worker4   <none>           <none>
pod/frontend-dev-2     1/1     Running   0          24h   10.10.10.4
worker5   <none>           <none>
pod/middleware-dev-1   1/1     Running   0          24h   10.10.10.2
worker4   <none>           <none>
pod/middleware-dev-2   1/1     Running   0          24h   10.10.10.6
worker5   <none>           <none>

NAME                         TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)          AGE   SELECTOR
service/backend-dev      ClusterIP   10.233.2.15     <none>        3306/
TCP          24h   app=backend-dev
service/frontend-dev     ClusterIP   10.233.57.82    <none>        443/
TCP,80/TCP   24h   app=frontend-dev
service/middleware-dev   ClusterIP   10.233.63.214   <none>        8989/
TCP          24h   app=middleware-dev


kubectl get all -n test-ns -o wide
NAME                        READY   STATUS    RESTARTS   AGE   IP
NODE      NOMINATED NODE   READINESS GATES
pod/backend-test-1      1/1     Running   0          24h   10.10.40.4
worker4   <none>           <none>
pod/backend-test-2      1/1     Running   0          24h   10.10.40.5
worker5   <none>           <none>
pod/middleware-test-1   1/1     Running   0          24h   10.10.40.2
worker4   <none>           <none>
pod/middleware-test-2   1/1     Running   0          24h   10.10.40.3
worker5   <none>           <none>

NAME                         TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)     AGE   SELECTOR
service/backend-test      ClusterIP   10.233.30.92    <none>        3306/
TCP    24h   app=backend-test
service/middleware-test   ClusterIP   10.233.21.135   <none>        8989/
TCP    24h   app=middleware-test

kubectl get all -n hr-ns -o wide
```

```
NAME       READY   STATUS    RESTARTS   AGE   IP           NODE
NOMINATED NODE   READINESS GATES

pod/hr-1   1/1     Running   0          24h   10.10.20.2   worker4
<none>           <none>


kubectl get all -n fin-ns -o wide

NAME        READY   STATUS    RESTARTS   AGE   IP           NODE
NOMINATED NODE   READINESS GATES

pod/fin-1   1/1     Running   0          24h   10.10.30.2   worker5
<none>           <none>
```

Once the VNRs are created, traffic will be allowed between name spaces in compliance with VNR rules:Hub VNR to Spoke VNR and vice-versa traffic is allowed.

- Spoke VNR to Spoke VNR traffic is not allowed.
- Mesh VNR to Mesh VNR traffic is allowed.
- Mesh VNR to Hub VNR traffic is not allowed.

```
kubectl exec -n fin-ns fin-1 /bin/bash -- curl 10.233.63.214:8989

% Total    % Received % Xferd  Average Speed   Time    Time     Time
Current

                                 Dload  Upload   Total   Spent    Left
Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--
0<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">

<title>Directory listing for /</title>

</head>

<body>

<h1>Directory listing for /</h1>

<hr>

<ul>

</ul>

<hr>

</body>

</html>

100   297  100   297    0     0  21214      0 --:--:-- --:--:-- --:--:--
21214
```

```
kubectl exec -n fin-ns fin-1 /bin/bash -- curl 10.233.2.15:3306

% Total    % Received % Xferd  Average Speed   Time    Time     Time
Current

                                 Dload  Upload   Total   Spent    Left
Speed

100   297  100   297    0     0  33000      0 --:--:-- --:--:-- --:--:--
37125

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">

<title>Directory listing for /</title>

</head>

<body>

<h1>Directory listing for /</h1>

<hr>

<ul>

</ul>

<hr>

</body>

</html>



kubectl exec -n hr-ns hr-1 /bin/bash -- curl 10.233.2.15:3306

% Total    % Received % Xferd  Average Speed   Time    Time     Time
Current

                                 Dload  Upload   Total   Spent    Left
Speed

100   297  100   297    0     0  49500      0 --:--:-- --:--:-- --:--:--
59400

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">

<title>Directory listing for /</title>

</head>

<body>

<h1>Directory listing for /</h1>
```

```
<hr>
<ul>
</ul>
<hr>
</body>
</html>

kubectl exec -n hr-ns hr-1 /bin/bash -- curl 10.233.63.214:8989
 % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--
0<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
</ul>
<hr>
</body>
</html>
100   297  100   297    0     0  29700      0 --:--:-- --:--:-- --:--:--
29700

kubectl exec -n dev-ns frontend-dev-1 /bin/bash  -- curl
10.233.30.92:3306
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100   297  100   297    0     0  59400      0 --:--:-- --:--:-- --:--:--
59400
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">
<html>
```

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
</ul>
<hr>
</body>
</html>
```

Let's create the CSP:

```
kubectl apply -f dev-ns-csp.yaml
contrailsecuritypolicy.core.contrail.juniper.net/dev-ns-csp created

kubectl apply -f hr-ns-csp.yaml
contrailsecuritypolicy.core.contrail.juniper.net/hr-ns-csp created

kubectl apply -f fin-ns-csp.yaml
contrailsecuritypolicy.core.contrail.juniper.net/fin-ns-csp created

kubectl apply -f test-ns-csp.yaml
contrailsecuritypolicy.core.contrail.juniper.net/test-ns-csp created
```

Undesired traffic flows, which worked earlier, but now will not work as required CSPs have been applied:

```
kubectl exec -n dev-ns frontend-dev-1 /bin/bash  -- curl
10.233.30.92:3306
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
  0     0    0     0    0     0      0      0 --:--:--  0:02:09 --:--:--
0
curl: (28) Failed to connect to 10.233.30.92 port 3306: Connection timed
out
command terminated with exit code 28

kubectl exec -n hr-ns hr-1 /bin/bash -- curl 10.233.63.214:8989
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
  0     0    0     0    0     0      0      0 --:--:--  0:02:09 --:--:--
0
curl: (28) Failed to connect to 10.233.63.214 port 8989: Connection timed
out
command terminated with exit code 28

kubectl exec -n hr-ns hr-1 /bin/bash -- curl 10.233.2.15:3306
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
  0     0    0     0    0     0      0      0 --:--:--  0:02:10 --:--:--
0
curl: (28) Failed to connect to 10.233.2.15 port 3306: Connection timed
out
command terminated with exit code 28

 kubectl exec -n fin-ns fin-1 /bin/bash -- curl 10.233.2.15:3306
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
  0     0    0     0    0     0      0      0 --:--:--  0:02:10 --:--:--
0
curl: (28) Failed to connect to 10.233.2.15 port 3306: Connection timed
out
command terminated with exit code 28
```

The CSPs should allow desired traffic flow:

```
kubectl get ingress -n dev-ns
NAME                       CLASS    HOSTS              ADDRESS     PORTS
AGE
hello-kubernetes-ingress   <none>   frontend.dev.svc   10.30.1.2   80
25h

curl -H "Host:frontend.dev.svc" 10.30.1.2
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">
<html>
```

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
</ul>
<hr>
</body>
</html>

kubectl get all -n dev-ns
NAME                    READY   STATUS    RESTARTS   AGE
pod/backend-dev-1       1/1     Running   0          25h
pod/backend-dev-2       1/1     Running   0          25h
pod/frontend-dev-1      1/1     Running   0          25h
pod/frontend-dev-2      1/1     Running   0          25h
pod/middleware-dev-1    1/1     Running   0          25h
pod/middleware-dev-2    1/1     Running   0          25h

NAME                      TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)           AGE
service/backend-dev       ClusterIP   10.233.2.15     <none>         3306/
TCP          25h
service/frontend-dev      ClusterIP   10.233.57.82    <none>         443/
TCP,80/TCP   25h
service/middleware-dev    ClusterIP   10.233.63.214   <none>         8989/
TCP          25h

kubectl exec -n hr-ns hr-1 /bin/bash -- curl 10.233.57.82
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100   297  100   297    0     0  74250      0 --:--:-- --:--:-- --:--:--
99000
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">
<html>
```

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
</ul>
<hr>
</body>
</html>
```

```
kubectl exec -n fin-ns fin-1 /bin/bash -- curl 10.233.57.82
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100   297  100   297    0     0  37125      0 --:--:-- --:--:-- --:--:--
37125
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
</ul>
<hr>
</body>
</html>
```

```
kubectl exec -it -n dev-ns frontend-dev-1 /bin/bash -- curl
10.233.63.214:8989
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
```

```
html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
</ul>
<hr>
</body>
</html>


kubectl exec -it -n dev-ns middleware-dev-1 /bin/bash -- curl
10.233.2.15:3306
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
</ul>
<hr>
</body>
</html>

kubectl get all -n test-ns
NAME                    READY   STATUS    RESTARTS   AGE
pod/backend-test-1      1/1     Running   0          25h
pod/backend-test-2      1/1     Running   0          25h
pod/middleware-test-1   1/1     Running   0          25h
```

```
pod/middleware-test-2   1/1     Running   0          25h

NAME                         TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)    AGE
service/backend-test         ClusterIP   10.233.30.92    <none>        3306/
TCP   25h
service/middleware-test      ClusterIP   10.233.21.135   <none>        8989/
TCP   25h

kubectl exec -n dev-ns frontend-dev-1 /bin/bash -- curl
10.233.21.135:8989
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100   297  100   297    0     0  2<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
</ul>
<hr>
</body>
</html>
9700      0 --:--:-- --:--:-- --:--:-- 33000LTE Traffic Flows Simulation
```

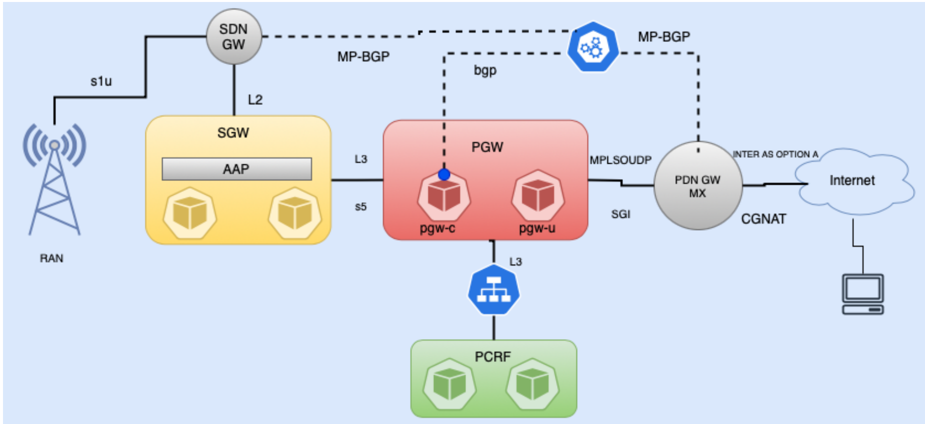


Figure 30 *LTE Traffic Simulation*

Use Case Description

The second use case of this chapter simulates LTE PGW-C, PGW-U PCRF, and SGW containerized network functions. This simulation will not demonstrate actual functional specs of LTE components but will demonstrate traffic flows using CN2 constructs. PThe GW-C Pod will serve control plane functionality. PGW-C Pods will establish eBGP sessions using its loop back IP towards BGPaaS IPs created by CN2, and the UE simulated IP will be advertised over that eBGP sessions. The PGW-C Pod loop back should be accessible over BGPaaS virtual Network, and we'll use the CN2 routing table construct to add a static route towards the PGW-C loop back, 1.1.1.100, via the PGW-C Pod physical interface IP, 31.31.31.12. Once an eBGP session is established, the inside PGW-C Pod will advertise a simulated User Equipment (UE) IP (for example,100.100.100.100) which will be active on the PGW-U Pods.

To simulate a mobile user on the PGW-U Pod, we will add the IP address 100.100.100.100 to its loop back interface. We will also add default route pointing to the PGW-U interface with subnet (31.31.31.0/24) as this subnet is extended to SDN-GW and this arrangement will allow UE simulated IP,, 100.100.100.100 accessibility via the PDN/SDN GW. SGW Pods will have a L2 network that will be extended to the SDN-GW and the Radio Network should be able to access a high availability IP address configured on the SGW Pods using active-active allowed address pair. PCRF Pods will be bound to a service on port 3386 and that service will be accessible from the PGW-C Pod.

Definition File

To achieve this functionality definition the file test-profile_nad.yaml will be used to create the following constructs:

- Create a NAD, sgw-l2-network with forwarding mode l2, that will be used for connectivity towards the Radio Unit via the SDN-GW.
- An allowed address pair (AAP) with active-active mode will be configured over the sgw-l2-network NAD.
- Create a NAD, gw-pgw-l3-network, with forwarding mode l3 and this subnet will be SGW-PGW connectivity.
- Create a vlan2-sub NAD network with forwarding mode default.
- Create a vlan3-sub NAD network with forwarding mode default.
- BGPaaS IPs will be created over subnets associated with vlan2-sub and vlan3-sub NAD.
- Create deployment for SGW.
- Create 1 PGW-C Pod.
- Create 1 PGW-U Pod.
- Create PCRF deployment.
- Create a service for PCRF pods.
- Create Routing table construct.

Here's the definition file to create these described components:

```
sudo cat > test-profile_nad.yaml <<END_OF_SCRIPT
---
apiVersion: v1
kind: Namespace
metadata:
  labels:
    name: telco-ns
    core.juniper.net/isolated-namespace: 'true'
  name: telco-ns
---
apiVersion: core.contrail.juniper.net/v3
kind: RouteTable
metadata:
  name: static-rt
  namespace: telco-ns
spec:
  routes:
    route:
    - nextHop: 31.31.31.12
```

```
      nextHopType: ip-address
      prefix: 1.1.1.100/32
      communityAttributes:
        communityAttribute:
          - accept-own
---
apiVersion: “K8s.cni.cncf.io/v1”
kind: NetworkAttachmentDefinition
metadata:
  name: sgw-l2-network-pf4fhmacys6j
  namespace: telco-ns
  annotations:
    juniper.net/networks: ‘{
      “ipamV4Subnet”: “15.15.15.0/24”,
      “virtualNetworkNetworkID”: 10001,
      “forwardingMode”: “l2”,
      “routeTargetList”: [“target:64562:1001”]
    }’
spec:
  config: ‘{
  “cniVersion”: “0.3.1”,
  “name”: “sgw-l2-network-pf4fhmacys6j”,
  “type”: “contrail-K8s-cni”
}’
---
apiVersion: “K8s.cni.cncf.io/v1”
kind: NetworkAttachmentDefinition
metadata:
  name: sgw-pgw-l3-network-pf4fhmacys6j
  namespace: telco-ns
  labels:
    vn: sgw-pgw-l3-network-pf4fhmacys6j
  annotations:
    juniper.net/networks: ‘{
      “ipamV4Subnet”: “25.25.25.0/24”
    }’
spec:
  config: ‘{
```

```
  "cniVersion": "0.3.1",
  "name": "sgw-pgw-l3-network-pf4fhmacys6j",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vlan2-pf4fhmacys6j
  namespace: telco-ns
  labels:
    vn: sgix-network-pf4fhmacys6j
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "31.31.31.0/24",
      "routeTargetList": ["target:64562:2101"],
      "exportRouteTargetList": ["target:65291:38113"],
      "importRouteTargetList": ["target:52555:6855"],
      "routeTableReferences": [{"name": "static-rt", "namespace": "telco-
ns"}]
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "vlan2-pf4fhmacys6j",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: "K8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vlan3-pf4fhmacys6j
  namespace: telco-ns
  labels:
    vn: sgix-network-pf4fhmacys6j
  annotations:
    juniper.net/networks: '{
      "ipamV4Subnet": "32.32.32.0/24",
      "exportRouteTargetList": ["target:65291:38113"],
```

```
      "importRouteTargetList": ["target:52555:6855"]
    }'
spec:
  config: '{
  "cniVersion": "0.3.1",
  "name": "vlan3-pf4fhmacys6j",
  "type": "contrail-K8s-cni"
}'
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sgw-pf4fhmacys6j
  namespace: telco-ns
spec:
  replicas: 2
  selector:
    matchLabels:
      app: sgw-pf4fhmacys6j
  template:
    metadata:
      labels:
        app: sgw-pf4fhmacys6j
      annotations:
        K8s.v1.cni.cncf.io/networks: '[{"name": "sgw-l2-network-
pf4fhmacys6j", "cni-args":
          {"net.juniper.contrail.allowedAddressPairs": "[{\"ip\":
\"15.15.15.254/32\",
          \"addressMode\": \"active-active\"}, {\"ip\":
\"32d2:ec3c:164d:a8d7:4e06:bc15:ffff:fffe/128\",
          \"addressMode\": \"active-standby\"}]"}}, {"name": "sgw-pgw-l3-
network-pf4fhmacys6j",
          "cni-args": {}}]
          '
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
```

```
              - key: app
                operator: In
                values:
                - sgw-pf4fhmacys6j
            topologyKey: kubernetes.io/hostname
      containers:
      - name: sgw-pf4fhmacys6j
        image: deployer-node.maas:5000/ubuntu-traffic:latest
        imagePullPolicy: IfNotPresent
        securityContext:
          privileged: true
          capabilities:
            add:
            - NET_ADMIN
---
apiVersion: v1
kind: Pod
metadata:
  annotations:
    K8s.v1.cni.cncf.io/networks: '[{"name": "sgw-pgw-l3-network-
pf4fhmacys6j", "cni-args":
      {}}, {"name": "vlan2-pf4fhmacys6j", "cni-args":
{},"ips":["31.31.31.12"]}, {"name": "vlan3-pf4fhmacys6j", "cni-args":{},"
ips":["32.32.32.12"]}]'
    core.juniper.net/bgpaas-networks: 'vlan2-pf4fhmacys6j,vlan3-
pf4fhmacys6j'
  labels:
    app: pgw-c-pf4fhmacys6j
  name: pgw-c-pf4fhmacys6j
  namespace: telco-ns
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - pgw-c-pf4fhmacys6j
```

```
        topologyKey: kubernetes.io/hostname
  containers:
  - name: pgw-c-pf4fhmacys6j
    image: deployer-node.maas:5000/ubuntu-traffic:latest
    imagePullPolicy: IfNotPresent
    securityContext:
      privileged: true
      capabilities:
        add:
        - NET_ADMIN
---
apiVersion: v1
kind: Pod
metadata:
  annotations:
    K8s.v1.cni.cncf.io/networks: '[{"name": "sgw-pgw-l3-network-
pf4fhmacys6j", "cni-args":
      {}}, {"name": "vlan2-pf4fhmacys6j", "cni-args":
{},"ips":["31.31.31.13"]}, {"name": "vlan3-pf4fhmacys6j", "cni-args":{},"
ips":["32.32.32.13"]}]'
    core.juniper.net/bgpaas-networks: 'vlan2-pf4fhmacys6j,vlan3-
pf4fhmacys6j'
  labels:
    app: pgw-u-pf4fhmacys6j
  name: pgw-u-pf4fhmacys6j
  namespace: telco-ns
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: app
            operator: In
            values:
            - pgw-u-pf4fhmacys6j
        topologyKey: kubernetes.io/hostname
  containers:
  - name: pgw-u-pf4fhmacys6j
    image: deployer-node.maas:5000/ubuntu-traffic:latest
```

```
    imagePullPolicy: IfNotPresent
    securityContext:
      privileged: true
      capabilities:
        add:
        - NET_ADMIN
---
apiVersion: core.contrail.juniper.net/v1
kind: BGPAsAService
metadata:
  annotations:
    core.juniper.net/display-name: Sample BGP as a Service
    core.juniper.net/description: BGP as a Service (BGPaaS) feature
allows a guest
      virtual machine (VM) to place routes in its own virtual routing and
forwarding
      (VRF) instance using BGP.
  name: bgpaas-pgw-vlan2-pf4fhmacys6j
  namespace: telco-ns
spec:
  autonomousSystem: 65291
  shared: false
  virtualMachineInterfacesSelector:
  - matchLabels:
      core.juniper.net/bgpaasVN: vlan2-pf4fhmacys6j
---
apiVersion: core.contrail.juniper.net/v1
kind: BGPAsAService
metadata:
  annotations:
    core.juniper.net/display-name: Sample BGP as a Service
    core.juniper.net/description: BGP as a Service (BGPaaS) feature
allows a guest
      virtual machine (VM) to place routes in its own virtual routing and
forwarding
      (VRF) instance using BGP.
  name: bgpaas-pgw-vlan3-pf4fhmacys6j
  namespace: telco-ns
spec:
  autonomousSystem: 65291
```

```
  shared: false
  virtualMachineInterfacesSelector:
  - matchLabels:
      core.juniper.net/bgpaasVN: vlan3-pf4fhmacys6j
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pcrf-pf4fhmacys6j
  namespace: telco-ns
spec:
  replicas: 2
  selector:
    matchLabels:
      app: pcrf-pf4fhmacys6j
  template:
    metadata:
      labels:
        app: pcrf-pf4fhmacys6j
    spec:
      containers:
      - name: pcrf-pf4fhmacys6j
        image: deployer-node.maas:5000/ubuntu-traffic:latest
        command:
          ["bash", "-c", "python3 -m http.server 3386 --directory /tmp/"]
        imagePullPolicy: IfNotPresent
        securityContext:
          privileged: true
          capabilities:
            add:
            - NET_ADMIN
---
apiVersion: v1
kind: Service
metadata:
  name: pcrf-pf4fhmacys6j
  namespace: telco-ns
spec:
```

```
  ports:
  - name: port-3386
    targetPort: 3386
    protocol: TCP
    port: 3386
  selector:
    app: pcrf-pf4fhmacys6j
  type: ClusterIP
END_OF_SCRIPT
```

Deployment Verification

```
kubectl get pods -A | grep pf4fhmacys6j

telco-ns          pcrf-pf4fhmacys6j-5db49d5f-6jngv
1/1     Running       0             23m

telco-ns          pcrf-pf4fhmacys6j-5db49d5f-jnv5n
1/1     Running       0             23m

telco-ns          pgw-c-pf4fhmacys6j
1/1     Running       0             19m

telco-ns          pgw-u-pf4fhmacys6j
1/1     Running       0             15m

telco-ns          sgw-pf4fhmacys6j-7749c94fcf-52b9w
1/1     Running       0             23m

telco-ns          sgw-pf4fhmacys6j-7749c94fcf-ps57v
1/1     Running       0             23m


kubectl get bgpaas -A | grep pf4fhmacys6j

telco-ns    bgpaas-pgw-vlan2-pf4fhmacys6j   65291               false
Success   23m

telco-ns    bgpaas-pgw-vlan3-pf4fhmacys6j   65291               false
Success   23m


kubectl get svc -A | grep pf4fhmacys6j

telco-ns          pcrf-pf4fhmacys6j      ClusterIP   10.233.40.187
<none>        3386/TCP                 23m
```

Static Routing Configuration Inside the PGW-U Pod

```
kubectl get pods -n telco-ns | grep pgw-u

pgw-u-pf4fhmacys6j                  1/1     Running   0          97m


kubectl exec -it -n telco-ns pgw-u-pf4fhmacys6j /bin/bash

ip r delete default

ip r add default dev eth2 via 31.31.31.1
```

```
ip addr add 100.100.100.100 dev lo
```

HA IP Configuration Inside the SGW Pods

```
kubectl get pods -n telco-ns | grep sgw
sgw-pf4fhmacys6j-7749c94fcf-52b9w   1/1     Running   0          17h
sgw-pf4fhmacys6j-7749c94fcf-ps57v   1/1     Running   0          17h


kubectl exec -it -n telco-ns sgw-pf4fhmacys6j-7749c94fcf-52b9w /bin/bash
-- ip addr add 15.15.15.254 dev eth1
kubectl exec -it -n telco-ns sgw-pf4fhmacys6j-7749c94fcf-ps57v /bin/bash
-- ip addr add 15.15.15.254 dev eth1
```

PGW-C BGP Configuration

```
kubectl exec -it -n telco-ns pgw-c-pf4fhmacys6j /bin/bash -- ip addr add
1.1.1.100 dev lo


kubectl get bgpaas -n telco-ns
NAME                            AS      IPADDRESS   SHARED   STATE
AGE
bgpaas-pgw-vlan2-pf4fhmacys6j   65291               false    Success
7h29m
bgpaas-pgw-vlan3-pf4fhmacys6j   65291               false    Success
7h29m


kubectl describe bgpaas bgpaas-pgw-vlan2-pf4fhmacys6j -n telco-ns | grep
Bgpaas
      Bgpaas Primary IP:    31.31.31.2
      Bgpaas Secondary IP:  31.31.31.3


 kubectl describe bgpaas bgpaas-pgw-vlan3-pf4fhmacys6j -n telco-ns | grep
Bgpaas
      Bgpaas Primary IP:    32.32.32.2
      Bgpaas Secondary IP:  32.32.32.3
```

BGP configuration inside the PGW-C Pod:

```
cat > /etc/bird/bird.conf <<END_OF_SCRIPT
router id 1.1.1.100;
protocol kernel {
   scan time 60;
   ipv4 {
       import none;
```

```
        export all;
    };
}
protocol device {
    scan time 60;
}
protocol static static_vlan2 {
    ipv4;
    route 100.100.100.100/32 via 31.31.31.13;
}

protocol bgp bgp_vlan2_primary {
    description “BGP - vlan2 Primary”;
    connect retry time 10;
    error wait time 10,30;
    ipv4 {
        export where proto = “static_vlan2”;
        import all;
    };
    ipv6 {
        export all;
        import all;
    };
    local 1.1.1.100 as 65291;
    neighbor 31.31.31.2 as 64512;
}
protocol bgp bgp_vlan2_secondary {
    description “BGP - vlan2 secondary”;
    connect retry time 10;
    error wait time 10,30;
    ipv4 {
        export where proto = “static_vlan2”;
        import all;
    };
    ipv6 {
        export all;
        import all;
    };
```

```
    local 1.1.1.100 as 65291;
    neighbor 31.31.31.3 as 64512;
}
END_OF_SCRIPT
```

Start Bird service and verify BGP sessions status:

```
service bird restart
 * Restarting BIRD Internet Routing Daemon bird
[ OK ]
birdc show protocols all

birdc show protocols all | egrep -i 'BGP state|Neighbor address'
  BGP state:          Established
    Neighbor address: 31.31.31.2
  BGP state:          Established
    Neighbor address: 31.31.31.3
```

Traffic Flows Testing

- User equipment simulated IP address, 100.100.100.100, inside the PGW-U should be accessible from outside destinations via the PDN-GW or SDN-GW:

```
contrail@DCG> show route 100.100.100.100

l3_use_case.inet.0: 12 destinations, 17 routes (9 active, 0 holddown, 3
hidden)
+ = Active Route, - = Last Active, * = Both

100.100.100.100/32 *[BGP/170] 17:23:31, localpref 100, from 192.168.5.52
                      AS path: 65291 I, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.58), Push 72
                    [BGP/170] 17:23:31, localpref 100, from 192.168.5.53
                      AS path: 65291 I, validation-state: unverified
                    >  via Tunnel Composite, UDP (src 172.172.172.172
dest 192.168.5.58), Push 72

lab@internet_router> show route 100.100.100.100

inet.0: 24 destinations, 24 routes (24 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both
```

```
100.100.100.100/32 *[OSPF/150] 17:24:16, metric 0, tag 3489660928
                    > to 192.168.201.254 via vlan.201
```

Test connectivity from an external host which has reachability to the SDN-GW VRF and then further to the CN2 Pods:

```
ip r
default via 192.168.24.1 dev ens3 proto static
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
192.168.24.0/24 dev ens3 proto kernel scope link src 192.168.24.82

ping 100.100.100.100 -c1
PING 100.100.100.100 (100.100.100.100) 56(84) bytes of data.
64 bytes from 100.100.100.100: icmp_seq=1 ttl=61 time=8.30 ms

--- 100.100.100.100 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 8.302/8.302/8.302/0.000 ms
```

The SGW HA IP address,, 15.15.15.254, should be accessible from a simulated device in the Radio Unit over a Layer 2 network via the SDN-GW:

```
ip -4 -br a | grep 15.15.15
br-siov-200      UP             15.15.15.253/24

ping 15.15.15.254 -c1
PING 15.15.15.254 (15.15.15.254) 56(84) bytes of data.
64 bytes from 15.15.15.254: icmp_seq=1 ttl=64 time=4.01 ms

--- 15.15.15.254 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 4.019/4.019/4.019/0.000 ms
contrail@control-host:~$ ping 15.15.15.2 -c1
PING 15.15.15.2 (15.15.15.2) 56(84) bytes of data.
64 bytes from 15.15.15.2: icmp_seq=1 ttl=64 time=113 ms

--- 15.15.15.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 113.356/113.356/113.356/0.000 ms
contrail@control-host:~$ ping 15.15.15.3 -c1
PING 15.15.15.3 (15.15.15.3) 56(84) bytes of data.
```

```
64 bytes from 15.15.15.3: icmp_seq=1 ttl=64 time=4.51 ms

--- 15.15.15.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 4.519/4.519/4.519/0.000 ms
```

The PGW-C Pods should be able to access a service running inside the PCRF-Pods:

```
kubectl get svc -n telco-ns
NAME                TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
pcrf-pf4fhmacys6j   ClusterIP   10.233.11.89   <none>        3386/TCP
56m

kubectl describe svc pcrf-pf4fhmacys6j -n telco-ns
Name:              pcrf-pf4fhmacys6j
Namespace:         telco-ns
Labels:            back-reference.core.juniper.
net/080c781a751b3ea9e204f32c0daf2cbd7eef70c7f3b00773064cec13=InstanceIP_
contrail-K8s-kubemanager-cluster-local-pcrf-pf4fhmac
                   back-reference.core.juniper.
net/20bf5b77a65073cd5f7e68da6745cf20b86732b408baa09a37222f3c=FloatingIP_
contrail-K8s-kubemanager-cluster-local-pcrf-pf4fhmac
Annotations:       <none>
Selector:          app=pcrf-pf4fhmacys6j
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.233.11.89
IPs:               10.233.11.89
Port:              port-3386  3386/TCP
TargetPort:        3386/TCP
Endpoints:         10.233.71.17:3386,10.233.72.26:3386
Session Affinity:  None
Events:            <none>

kubectl get pods -n telco-ns | grep pgw-c
pgw-c-pf4fhmacys6j                  1/1     Running   0          44m


kubectl exec -it -n telco-ns pgw-c-pf4fhmacys6j  /bin/bash -- curl
10.233.11.89:3386
```

```
<!DOCTYPE HTML PUBLIC “-//W3C//DTD HTML 4.01//EN” “http://www.w3.org/TR/
html4/strict.dtd”>
<html>
<head>
<meta http-equiv=”Content-Type” content=”text/html; charset=utf-8”>
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
</ul>
<hr>Glossary
```

https://www.juniper.net/documentation/en_US/release-independent/glossary/topic-122763.html#symbols