JUNIPER NETWORKS

# DAY ONE: DEPLOY CLOUD-NATIVE CONTRAIL NETWORKING AS A CNI FOR KUBERNETES


By Krishna Kishore, Rahul Verma, Tayib Ahmed

# DAY ONE: DEPLOY CLOUD-NATIVE CONTRAIL NETWORKING AS A CNI FOR KUBERNETES

One of Contrail Networking's (CN2) key benefits is its simplicity and cloud-native design, making it easy to operate and consistent across different clouds. Additionally, CN2 is DevOps-friendly with your existing workflows and processes. Advanced security features such as micro-segmentation, multi-tenant and namespace network isolation, and label-based security policies provide pervasive security for your network. CN2's ability to manage multiple clusters with one CN2 instance, and its multi-cluster policy federation for network security and BGP cluster-to-cluster peering, enables scaling of your network across multiple clusters.

This book shows you how to deploy CN2 as a container network interface (CNI) in Kubernetes (K8s) and how to toggle these features. Step-by-step you'll learn how to deploy applications and add distributed clusters. It's a tour de force for CN2 and the authors make it easy with lots of verified examples and tips from their long-time Technical Support careers.

*"This new book on CN2 is an excellent resource that provides comprehensive coverage of key technical terms followed by step-by-step instructions for installation and verification. It also includes valuable insights from Juniper's Customer Focused Technical Support (CFTS) and Product Line Management (PLM) teams."*
***Payum Moussavi, GVP, Customer Service, Juniper Networks***

*"A thorough up-and-running book devoted to Cloud-Native Contrail Networking (CN2). Build a new cloud on day one! Everything you need plus insights and tips from Rahul, Kishore, and Tayib."*
***Raghupathi C., Senior Director of Technical Support, Juniper Networks***

## IT'S DAY ONE AND YOU HAVE A JOB TO DO:

- Understand Kubernetes Networking and the role of CNIs in Kubernetes.
- Learn about the advanced networking features and capabilities provided by CN2.
- Follow the CN2 deployment steps: installation of required components, configuring the network, and creating objects such as virtual networks.
- Learn how CN2 objects work and how they are used: virtual networks, BGP routers, isolated namespaces, security policies, etc.
- Deploy a multi-tier microservice-based enterprise application in CN2.
- Learn how to peer a CN2 cluster with DC gateway router.

Juniper Networks Books are focused on network reliability and efficiency. Peruse the complete library at www.juniper.net/dayone.

JUNIPER NETWORKS

# Day One: Deploy Cloud-Native Contrail Networking as a CNI for Kubernetes

By Krishna Kishore, Rahul Verma, Tayib Ahmed



JUNIPER NETWORKS

**Published by Juniper Networks Books**
Authors: Krishna Kishore, Rahul Verma, Tayib Ahmed
Technical Reviewers: Ashish Paul (JTAC), Sheetal Jangeed (CFTS), Aniruddh Amonker (CFTS), Aiman Iqbal
Editor in Chief: Patrick Ames
Printed in the USA by Vervante Corporation.
Version History: v1, April 2023
2 3 4 5 6 7 8 9 10

**About the Authors**

**Krishna Kishore** is an experienced Staff Engineer with Juniper CFTS, boasting a career spanning over 16 years in the field of networking. In his current role, he provides valuable support to Juniper customers in maintaining their private clouds and SD-WAN. With his wealth of knowledge and expertise, Krishna Kishore is an asset to Juniper CFTS and a reliable partner for customers seeking to optimize their networking infrastructure. He is the author of *Day One: Contrail Network Up and Running*. In addition to his professional pursuits, he also has a passion for design and 3D printing. He spends his free time creating and producing unique items that are not typically available for day-to-day needs.

**Rahul Verma** is a Staff Engineer at Juniper CFTS. With 13+ years of technical experience, he is an ambassador of Juniper values. He drives complex situations with passion and delivers excellence daily. In his current role, he possesses expertise across various networking domains, including security, SDN, SD-WAN, automation, and virtualization. Rahul is a technical writer (published a *Day One Book: vSRX on KVM*), an avid runner and loves to trek and breathe mountains.

**Tayib Ahmed** is a Senior Product Manager, Cloud Native Contrail Networking (CN2) at Juniper Networks. With 18 years in the networking industry, and over 10 years as an engineering leader, Tayib has built and lead multi-functional global teams from the ground up - creating and optimizing processes, building training plans, engaging with customers to remove friction points in the product and service offerings, and fostering innovation. As a Product Manager, he is responsible for the roadmap of CN2 - working closely with the engineering team for feature delivery, marketing and field for GTM, and customers for adoption. When not at work, Tayib can be found mentoring startups in India. He is an avid angel investor and mentor and

**Authors' Acknowledgments**

**Krishna Kishore** would like to express sincere gratitude to Payum Moussavi, Raghupathi C, and Brahmeswara Reddy Kauluru for their unwavering support in the office, and to my wife Rakshita Pandey for her constant encouragement and support throughout this journey. I am also grateful to my colleagues Ashish Paul, Aiman Iqbal, Sheetal Jangeed, and Aniruddh Amonker for their invaluable feedback and critical comments, which helped shape this book. I also extend Patrick Ames, for his guidance and expertise in bringing this book to fruition. I would also like to extend my heartfelt appreciation to Tayib Ahmed, co-author of this book, for his efforts and insights throughout the writing process.

**Rahul Verma** would like to express sincere gratitude to mentors Payum Moussavi, Raghupathi C, and Brahmeswara Reddy Kauluru, who kept me motivated, and my parents and my wife Isha S. Verma, who encouraged me for the pursuit. Peers and reviewers (Aiman, Sheetal, Ashish, and Aniruddh) played an important role in vetting the content. Delighted to have Patrick Ames as the Editor in Chief to spray his magic onto the work to give it the beautiful shape it is in today.

**Tayib Ahmed** would like to express sincere gratitude to Aiman Iqbal and Nick Davey for the help they extended while working on this book. Gratitude to co-authors is usually an afterthought, but I would like to call out the brilliance of Rahul Verma and Krishna Kishore. This book would have been a non-starter without their boundless enthusiasm, technical competence, and grit to get to the finish line despite countless obstacles. The support system at home can never be understated - my gratitude to my parents, my wife Nida Kauser and my three wonderful kids who put up with periods of my unavailability without complaint.

## Welcome to Day One

This book is part of the *Day One* library, produced and published by Juniper Networks Books. *Day One* books feature Juniper Networks technology with straightforward explanations, step-by-step instructions, and practical examples that are easy to follow.

- Download a free PDF edition at https://www.juniper.net/dayone.
- Purchase the paper edition at Vervante Corporation (www.vervante.com).

## Key CN2 Resources

The authors of this book highly recommend the following CN2 resources and documentation:

- CN2 Manifests:
  https://support.juniper.net/support/downloads/?p=contrail-networking
- CN2 Documentation:
  https://www.juniper.net/documentation/product/us/en/cloud-native-contrail-networking/#cat=set_up
- CN2 supported platforms:
  https://www.juniper.net/documentation/en_US/release-independent/contrail-cloud-native/topics/reference/cloud-native-contrail-supported-platforms.pdf
- CN2 Free trail:
  https://www.juniper.net/us/en/forms/cn2-free-trial.html
- CN2 Day One github repo for this book:
  https://github.com/Juniper/cn2dayone

## What You Need to Know Before Reading This Book

Before reading this book, you need to be familiar with the basic administrative functions of Contrail Networking, including the ability to work with operational commands and to read, understand, and change configurations.

This book makes a few assumptions about you, the reader:

- You are familiar with Linux CLI and virtualization.
- You are familiar using tools like VIM or VI.
- You are familiar and able to read YAML files, and you have high-level understanding of TCP/IP stack.

## What You Will Be Able To Do After Reading This Book

- Understand Kubernetes Networking and the role of CNI in Kubernetes.
- Discover how CN2 delivers more than CNI: Learn about the advanced networking features and capabilities provided by CN2 that go beyond the basic CNI, such as multi-cluster management, advanced security, and improved performance.
- Understand the CN2 deployment steps: Comprehend the step-by-step process of deploying CN2 in a Kubernetes cluster, including the installation of required components, configuring the network, and creating objects such as virtual networks and BGP routers.
- Learn about CN2 objects: Understand how CN2 objects such as virtual networks, BGP routers, isolated namespaces, and security policies work and how they are used to create a secure and scalable network environment.
- Deploy multi-tier microservice-based enterprise application in CN2: Understand how to deploy a multi-tier microservice-based enterprise application in a CN2 environment and how CN2 can be used to manage and secure the network for such applications.
- Learn how to peer CN2 cluster with DC gateway router: Understand how to peer a CN2 cluster with a DC gateway router to provide access to services and resources from outside the cluster.

## Glossary

BMS: Bare Metal Server

CIDR: Classless Inter-Domain Routing or super-netting

CLI: Command Line Interface

CRI: Container Runtime Interface

DPDK: The Data Plane Development Kit

eBPF: Extended Berkeley Packet Filter

FQDN: Fully Qualified Domain Name

K8s: Kubernetes

KLM: Kernel Loadable Module

KVM: Kernel-based Virtual Machine

LCM: Life Cycle Management

LXD: Linux Container

OCI: Open Container Initiative

RKT: Container engine (pronounced as rocket)

SR-IOV: Single Root - Input/Output virtualization

UEFI: Unified Extensible Firmware Interface

VM: Virtual Machine

YAML: Yet Another Markup language

Chapter 1

# Network Virtualization

This chapter provides an overview of how data center (DC) architecture evolved to keep up with trends in applications hosted from monolithic to microservices-based, evidenced by the emergence of containerization and the introduction of Kubernetes as a leading orchestration platform. The chapter provides an insight into Container and Kubernetes networking.

As Kubernetes has become the orchestrator of choice in data centers, it has created a need for software-defined networking solutions such as Cloud-Native Contrail Networking (CN2) to address new challenges and provide a more secure and scalable network environment. .

## Evolution of DCs from Monolithic to Microservices-based Architecture

The evolution of data centers has trended towards greater flexibility and efficiency of computing resources, rather than many physical servers, each running a separate application or service. The older approach has several drawbacks, including high costs for both hardware and maintenance, and limited scalability.

So, DCs began to adopt virtualization technologies that allowed multiple virtual machines (VMs) to run on a single bare metal server for better resource utilization and easier management of infrastructure.

The advent of virtualization added a new challenge: the VMs that were spun up inside the servers, in turn, needed to be connected, both within the same server and across servers. The network problem had now moved inside the servers, too!

In addition, server resources need to be monitored, the LCM (Lifecycle Management) of VMs was a necessity, and monitoring and troubleshooting were needed to maintain sanity of Day 2 operations.

To improve resources utilization, features like multi-tenancy were developed.

The added complexities of a virtualized world gave birth to an open-source platform called OpenStack. The OpenStack community developed many projects for managing the complexities of virtual infrastructure and the one used for managing the networking piece was Neutron.

While Neutron was good for demonstrating networking in a virtual world, it fell short of the rigors needed to operate a networking infrastructure in the real world.

This is where Juniper developed an overlay SDN solution, based off OpenStack called Contrail and it has been solving the real-world networking demands of leading telcos and ISPs for a decade.

However, virtualization still had its limitations: slow provisioning times, significant resources required to run each VM such as CPU, memory, and storage. CPU cycles are also used when hypervisors emulate several hardware functions and map actual memory addresses to memory addresses.

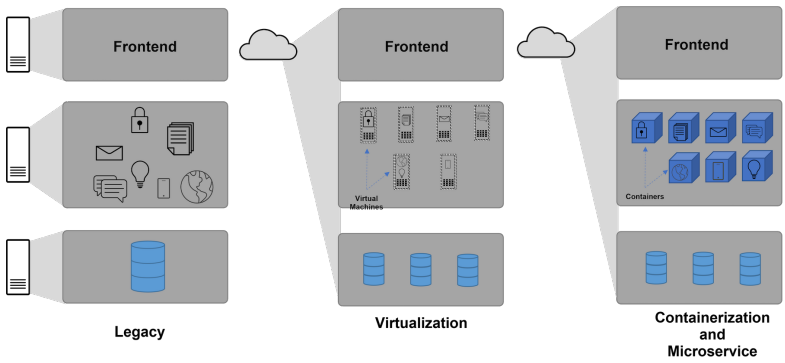



*Figure 1-1* *Evolution Of Data Centers From the Monolithic Architecture to Microservices-Based Architecture*

With microservices-based applications exploding in usage, containers became the de facto choice for implementing microservices-based applications. In recent years, containerization has emerged as a new approach to managing data centers. Containers are a lightweight alternative to VMs, allowing applications to be packaged in a self-contained way and run on any compatible host. This allows for faster deployment and easier scaling of applications. In addition, containers provide better isolation between applications, making it easier to manage their interactions. As seen with virtualization, introduction of new technologies to solve pain points in existing ones comes with a trade off – new and different problems for the older ones.

The adoption of microservices and container-based architecture has led to a significant increase in the number of network connections and traffic flows, making it difficult to manage and secure the network. Just like OpenStack emerged as the orchestrator of choice for virtual workloads, Kubernetes has emerged as the orchestrator of choice for containerized workloads.

The way Contrail stepped in to provide production-grade networking for VMs, Cloud-Native Contrail Networking, or CN2, has emerged as the CNI of choice for cloud-native workloads or containers. CN2 provides advanced networking services such as multi-cluster management, advanced security, and improved performance.

## Data Center Issues Today Are Essentially About Networking

Today, data centers face several challenges, one of which is the increasing complexity of managing and scaling the network to support the growing number of applications and services. As organizations increasingly rely on digital technologies, the number of applications and services being run in data centers is increasing exponentially, resulting in the need for more bandwidth, and perhaps more importantly, the ability to manage, scale, and secure the network to support these applications and services.

As the number of applications and services running in data centers increases, the network infrastructure needs to be able to scale to support them. Traditional networking solutions are often not able to keep up with this demand, making it difficult to add new applications and services without causing disruptions or creating bottlenecks.

Another issue is the lack of visibility and control over the network. As the number of devices and applications connected to the network increases, it becomes increasingly difficult to manage and troubleshoot the network. Without proper visibility and control, it can be challenging to identify and resolve issues, which can lead to network downtime and decreased productivity.

In addition, the complexity of the network infrastructure has increased with the adoption of cloud, virtualization, and containerized workloads. This has made it challenging to monitor and troubleshoot network issues and to manage security policies across the infrastructure.

In summary, today's data centers are facing networking challenges related to the management, scalability, and visibility of the network. Traditional networking solutions are not able to keep up with the growing complexity in data centers, and lack the scalability, visibility, and control needed to support these applications and services effectively.

## Overview of Containers

Now let's take a brief historical journey through containerization and review key terms essential to understanding the content of this book and where DCs are evolving.

Containers, in their rudimentary form, have existed since 1979 as CHROOT and BSD Jail. However, modern-day containers started to appear in 2006 after Google introduced cgroups and Kernel Namespaces.

Containers package the application, libraries, and runtime dependencies into a single file. They are portable, which means containers created on one machine can move to any other machine provided the target machine is also running a similar kernel (see Figure 1.2). For example, all containers created on Linux can run on any other Linux machine provided the destination has compatible container runtime.

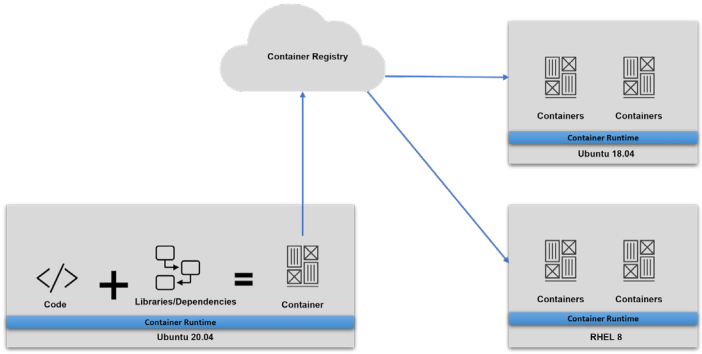



*Figure 1.2* *How Containers Are Created and Delivered*

Efforts were made by many different organizations and vendors to popularize containers, with Docker becoming synonymous for container management since 2013. Docker offers tools to create, publish, and operate containers.

### Container Runtime

Low-level software, which consumes mount point, meta data provided by the container engine and communicates with the kernel to start containerized processes, setting up cgroups, etc.

Though there are different container runtimes, most of them follow the specifications provided by OCI (Open Container Initiative). OCI was established by Docker and several other organizations in 2015 to standardize the runtimes and image specifications, see: https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction#containers_101 .

### Container Engine

Container engine is software that accepts user commands, including those from the command line. It fetches images and executes the containers from the viewpoint of the end user. There are numerous container engines: LXD, RKT, CRI-O, and Docker, for instance.

### Container Image

It is a file used as a local mount point when starting a container. Usually, these are stored on a container registry and pulled when the container is started for the first time on a host.

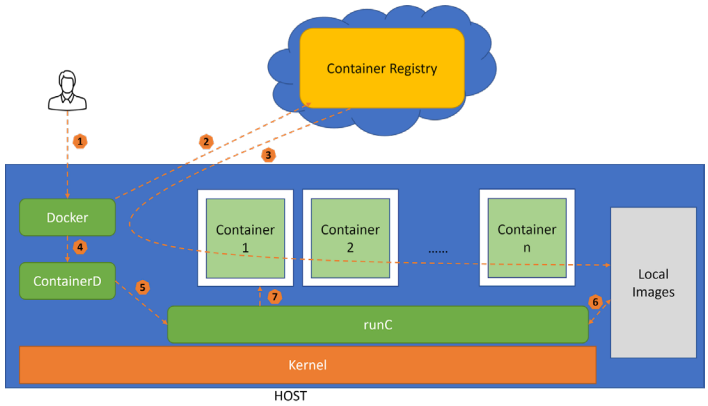



*Figure 1.3* *Overview of Container Engine and Container Runtime*

Running containers on a bare metal server or a VM requires a container runtime or engine, which manages all the steps in the life cycle of containers, viz, getting the image, mapping volumes, starting, and stopping the containers.

Here's a list of container runtimes that you might come across:

- Rkt
- Docker
- Podman

## Kubernetes: Container Orchestrator

Kubernetes, also known as K8s, is an open-source container orchestration engine for automating deployment, scaling, and management of container application workloads. It helps organize and manage containers across multiple hosts in a cluster and provides

features such as self-healing, rolling updates, replica sets, desired state of clusters, and service discovery. Kubernetes provides a unified solution for deploying and managing containers at scale, making it an essential tool for modern cloud-native application development and deployment.

A typical deployment of Kubernetes in a cluster setup consists of two primary components: a control plane residing on control node and worker nodes running the workload pods. The control plane of Kubernetes is responsible for managing the overall state of the cluster, including scheduling, and deploying containers, monitoring their health, and handling failovers and scaling. A worker node in Kubernetes is responsible for running the containers that make up an application and providing the necessary resources and services to support those containers, such as networking and storage. It communicates with the control plane to receive instructions on what containers to run and how to manage them.

Let's delve into the terminology used in K8s so the rest of this book makes sense.

### Nodes and Cluster

In K8s, *Node* is a VM or a physical machine. A collection of nodes working with a common control plane is referred to as *Cluster*.

### Pod

Pod is the smallest entity that can run on a node. A pod usually encapsulates a container inside it. However, there can be special requirements where developers may want to run more than one container per pod.

TIP Ideally a pod runs one container but the same isn't a rule. Kubernetes also supports running more than one container in a single pod. Read more about the multiple containers in a pod here: https://kubernetes.io/docs/concepts/workloads/pods/

Pod hosts the application or a part of the application and exposes the intended service on a specific port for other pods within the cluster or to the outside world to communicate with.

For a complete list of K8s terminologies please refer to the K8s documentation here: https://kubernetes.io/docs/reference/glossary/?fundamental=true .

### Kubernetes Cluster and its components

Figure 1.4 shows the basic building blocks of the Kubernetes ecosystem. It is majorly divided into a control node and multiple worker nodes forming a cluster. The control node and worker node run their respective components in the form of pods in a namespace called *kube-system*.

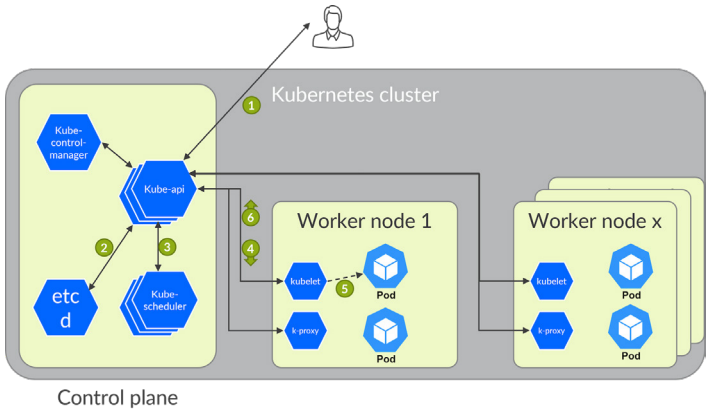



*Figure 1.4* *K8s Cluster and Components*

Control node runs a minimum of the following components:

- kube API-Server acts as a gateway for RESTful clients to communicate with. It also acts as focal point of communication for all other components of control and worker nodes.
- kube-scheduler fetches the state of the nodes from the etcd and calculates where to schedule the next pod.
- kube-control-manager runs controller processes to regulate the shared state of the cluster.
- etcd is the key-value store for the K8s ecosystem, all information about nodes, workload configuration and the intents are stored in the etcd store.

Worker nodes by default run the following components. However, this may differ based on the deployment variants:

- kubelet is the captain of the ship (worker node) and makes sure pods and containers are healthy and in a desired state. It is the one that updates the etcd with the state of cluster via kubeAPI
- kube-proxy maintains the network rules and the proxy information for data plane communication for the workloads/pods.

The process of deploying intent in a Kubernetes cluster can be visualized as follows: When a REST request is made via kubectl or any other API client, it submits the pod spec to the kube-api-server. (1) The API server writes the Pod object to the etcd. (2) All K8s components use watches to keep checking API server for changes. In this case, KubeScheduler (using its watchers) sees that a new pod object is created but isn't yet bound to any compute nodes.

KubeScheduler assigns a node to the pod and updates the API server which is further propagated to etcd. (3) Kubelet, on every node (using its watchers), keep watching API server. (4) On reading about a new pod assignment to itself, kubelet starts the pod on its node. (5) Kubelet updates API server about the pod status. (6) API server persists the pod state into etcd. Once etcd sends an ack, API server sends ack to kubelet indicating the event status.

NOTE A K8s control node is also a worker node in itself and can run pods. It also has a kubelet, kube-proxy, and runs a container runtime engine of itself. We did not include these components on purpose in the diagram to simplify the learning curve.

### K8s Networking

Networking inside Kubernetes is based on the following network model:

- Containers within a pod share network namespaces (i.e., share same IP & MAC). Hence, the containers within a pod can communicate with each other using a loopback address.
- Every pod gets its own IP address and there is no need to map ports from container to the host.
- NAT is not required, pods on one node can communicate with pods on all other nodes.

### Understanding Pod Networking

In the previous section, we looked at container networking and how it is connected to the host. With the understanding that Kubernetes is a de-facto orchestration and container management system which works at scale, let's focus on pod networking.

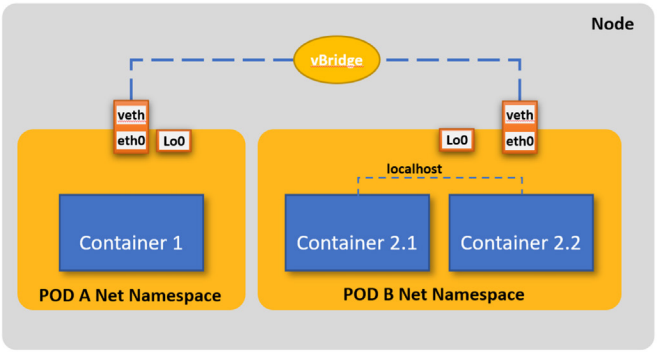



*Figure 1.5 Understanding Pod Networking*

A pod is a logical wrapper for a container to run in a K8s cluster. A pod can be termed as a group of one or more containers with shared network and storage resources.

In Figure 1.5, you can see a node running two pods: Pod A running one container and Pod B running two containers inside the pod.

Now, let us understand how a pod looks at the network.

```
root@Jumphost:~# kubectl get pods -A -o wide | grep PODA
default         PODA      1/1    Running    0        51s    10.233.65.1    centralworker1   <none>
<none>
root@Jumphost:~# kubectl get pods -A -o wide | grep PODB
default         PODB         2/2    Running          0        20m    10.233.66.1    centralworker2
<none>         <none>
```

Here, the kubectl command is listing all pods in a specific namespace.

Now, let us look at how a pod looks at the underlying network. On inspecting the pod running single container, you can see one network interface defined as eth0@if20 with IP 10.233.65.1/18 and the other as a lo0 aka loopback interface:

```
root@Jumphost:~# kubectl exec --stdin --tty PODA -- /bin/bash
root@PODA:/# 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
19: eth0@if20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
   link/ether 02:bd:d6:d6:00:59 brd ff:ff:ff:ff:ff:ff link-netnsid 0
   inet 10.233.65.1/18 brd 10.233.127.255 scope global eth0
      valid_lft forever preferred_lft forever
   inet6 fd85:ee78:d8a6:8607::1:101/112 scope global
      valid_lft forever preferred_lft forever
   inet6 fe80::246c:55ff:fed0:f255/64 scope link
      valid_lft forever preferred_lft forever
```

Likewise, if we inspect networking inside a multi-container pod, we will see an output of command *ip addr* showing the same IP and mac address. This is because they share the same storage and network resources and can communicate with each other using the loopback address, i.e., 127.0.0.1.

```
root@Jumphost:~# kubectl exec --stdin --tty PODB -c 1st -- /bin/bash
root@PODB:/# 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
13: eth0@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
   link/ether 02:52:2f:6b:db:17 brd ff:ff:ff:ff:ff:ff link-netnsid 0
   inet 10.233.66.1/18 brd 10.233.127.255 scope global eth0
      valid_lft forever preferred_lft forever
   inet6 fd85:ee78:d8a6:8607::1:201/112 scope global
```

```
      valid_lft forever preferred_lft forever
   inet6 fe80::b86f:14ff:fea2:553e/64 scope link
      valid_lft forever preferred_lft forever

root@Jumphost:~# kubectl exec --stdin --tty PODB -c 2nd -- /bin/bash
root@PODB:/# 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
13: eth0@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
   link/ether 02:52:2f:6b:db:17 brd ff:ff:ff:ff:ff:ff link-netnsid 0
   inet 10.233.66.1/18 brd 10.233.127.255 scope global eth0
      valid_lft forever preferred_lft forever
   inet6 fd85:ee78:d8a6:8607::1:201/112 scope global
      valid_lft forever preferred_lft forever
   inet6 fe80::b86f:14ff:fea2:553e/64 scope link
      valid_lft forever preferred_lft forever
```

Veth interface consists of two ends – one inside the pod and the other on the host.

Looking at the single container pod named "PODA", the eth0@if20 is the veth pair side of the pod and is tunneled into the host namespace where the other side of the veth pair lies.

To verify the host side of the veth, identify the host on which the pod is running by executing the command kubectl get pods -A -o wide. Using the information from the output, login to the host and execute the command ip -c link show up.

```
root@centralworker1:~# ip -c link show up
~
20: tapeth0-908665@if19: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode
DEFAULT group default
   link/ether 56:1c:23:6f:75:8d brd ff:ff:ff:ff:ff:ff link-netnsid 1
```

The ping from PODA container pod towards the multi-container pod can be captured on the tap interface of the host for troubleshooting purposes:

```
root@single:/# ping 10.233.66.1
PING 10.233.66.1 (10.233.66.1) 56(84) bytes of data.
64 bytes from 10.233.66.1: icmp_seq=1 ttl=63 time=2.49 ms
^C
--- 10.233.66.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.494/2.494/2.494/0.000 ms

root@centralworker1:~# tcpdump -ni tapeth0-908665
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on tapeth0-908665, link-type EN10MB (Ethernet), capture size 262144 bytes
16:46:36.326100 ARP, Request who-has 10.233.64.1 (00:00:5e:00:01:00) tell 10.233.64.1, length 28
16:46:38.326579 ARP, Request who-has 10.233.64.1 (00:00:5e:00:01:00) tell 10.233.64.1, length 28
16:46:38.348125 IP 10.233.65.1 > 10.233.66.1: ICMP echo request, id 382, seq 1, length 64
16:46:38.350581 IP 10.233.66.1 > 10.233.65.1: ICMP echo reply, id 382, seq 1, length 64
```

The traffic, when passed from pod to the host via veth pair, checks the Linux IP tables and once allowed, it is forwarded to the corresponding veth pair of the destination pod.

This also shows the flat architecture of K8s where all pods can communicate with other pods irrespective of where they are spawned. Similarly, if the destination pod is on a separate node, the CNI plugin implemented in K8s will learn the destination pod IP subnet and route the traffic to said node. Now, let us review what a CNI is.

### Container Network Interface (CNI)

By default, the container or pod does not have a network interface. The runtime or orchestrator calls a CNI to provide it.

CNI is used by container runtimes or orchestrators such as K8s to ADD or DEL interfaces to pods and CHECK the status of those interfaces.

When the Kubernetes control plane determines the node on which a pod should be created, the kube-api then informs corresponding kubelet to execute the task. The kubelet then executes the task on the worker node.

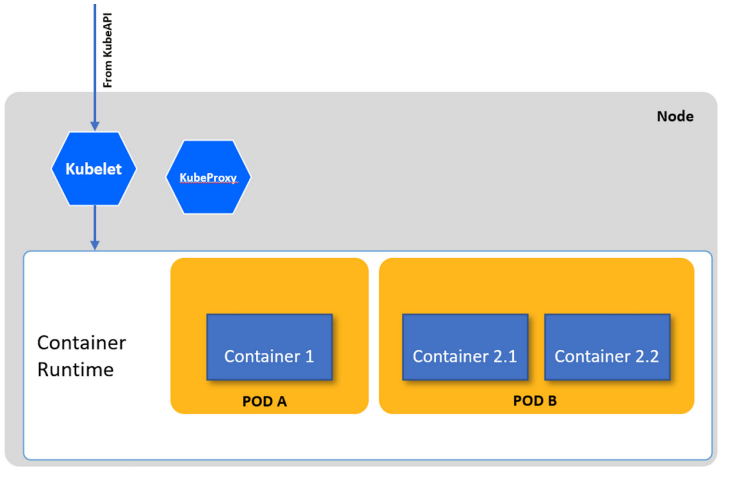



*Figure 1.6* *Kubelet and Its High-level Interaction*

Kubelet is the primary worker node agent, and it works in terms of PodSpec, which defines and describes a pod. Kubelet considers the definition and makes sure that containers are running and healthy as per the spec. Each worker node needs to install a container runtime at the time of cluster deployment as this is the component that would accept the intent definition from Kubelet and perform all low-level tasks. Container runtime is also sometimes termed as container engine, as it takes the responsibility of managing individual containers. In a nutshell, container engine undertakes the tasks to:

- Load the container image from a repository
- Monitor the local system resources
- Manage the lifecycle of a container

Kubernetes is designed to support modular cloud-native applications. Hence, the platform is equally flexible and modular. It provides interfaces at each layer to allow interchangeability. To incorporate multiple runtimes into Kubernetes, the community specified a CRI (Container Runtime Interface), which allows a user to switch between runtimes at any point of the cluster deployment. CRI-O is the most popular CRI plugin as it is known for being extremely light and nimble.

Likewise, CRI and its corresponding runtime, we have CNI (Container Network Interface) defined for this layer to be flexible for *n* number of plugins to be used for different functionalities. There are several open source and closed source CNI plugins available. You must use at least one CNI plugin for the pods in various nodes to communicate with each other. For example, Calico uses the standard L3 approach while Flannel alongside L3 fabric builds VXLAN out of the box. Cilium built a VXLAN-based networking solution leveraging the existing CNI plugins like Calico. Juniper's CN2 provides connectivity to the pod workloads using overlay tunnels across the IP fabric. These tunnels could be VXLAN, MPLSoUDP, or MPLSoGRE.

To conclude, Kubelet receives the intent from the Kube API server and connects to installed container runtime to run the required task. Container runtime performs ADD, DELETE or CHECK function to perform the required task and updates the Kubelet, which in turn updates the Kube API server with the result of the function execution. The process is termed as a JOB and the details are written into the etcd store.

### Kubernetes Services

Pods are the smallest execution unit in K8s cluster. Pods encapsulate containers that comprise of application code. Pods are ephemeral resources; they get their own IP address. If a pod goes down, K8s checks etcd, realizes that the current state does not meet the desired state, and spins up a new pod. This leads to a problem: if a set of pods, say "frontend," uses functionality from some set of pods, let's say "backend," how do the frontend pods keep track of IP addresses being used by pods running the backend application?

Kubernetes service is the answer. It is an abstraction which defines a logical set of pods and a policy to access them. It uses a Virtual IP as a stable IP. When a client sends a request to a service's VIP, the request is redirected to one of the pods that is associated with that service.

The set of pods targeted by a service are usually determined by the label selector.

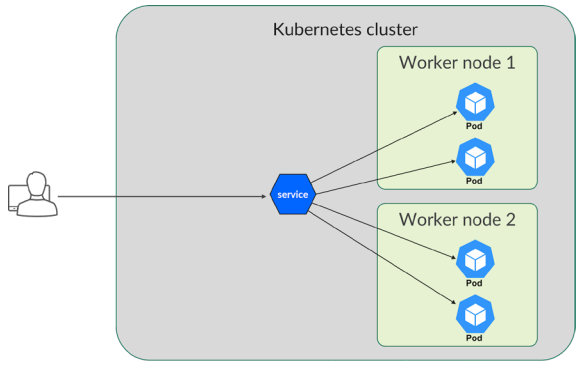



*Figure 1.7 Kubernetes Service*

K8s supports three types of services:

- Cluster IP: is the usual way to access a service from within a Kubernetes cluster. It is used for service communication within a cluster like frontend talking to backend of an application.
- Node Port: is the most basic way to access a service from outside the cluster. It is an extension of clusterIP that exposes the service to entities outside the cluster by adding a cluster-wide port on top of clusterIP. Node Port is used when you want external connectivity to a service.
- Load Balancer: a more sophisticated way to expose a service outside the cluster, using an external load balancer. It builds on top of NodePort and ClusterIP and exposes the service externally through a cloud provider's load balancer. Typically used when the Kubernetes cluster is hosted on a cloud provider.

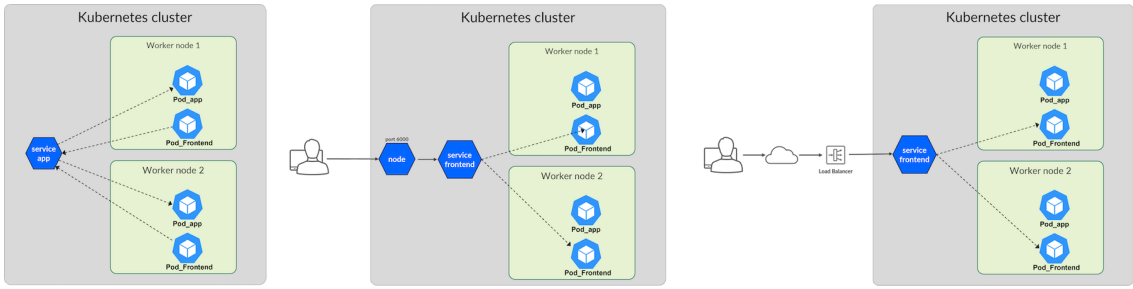



*Figure 1.8 Service Types: 1. Cluster IP, 2. Node Port, 3. Load-Balancer*

### Kubernetes Service Discovery

In modern distributed systems, services can be added, removed, or scaled dynamically, and their IP addresses can change frequently. Without a way to dynamically discover these services, it would be difficult to ensure that the clients can always connect to the correct service instance.

Kubernetes has an in-built service discovery mechanism that makes it easier for applications to find each other in a K8s cluster as pods and services are dynamically created, updated, and shifted between nodes.

Service discovery is done using Kubernetes DNS service called CoreDNS. It is a single process container which resolves and caches DNS queries, responds to health checks, and provides metrics.

Kubelet sets each new pod's /etc/resolv.conf nameserver option to the cluster IP of the DNS service, with appropriate search options to allow shorter hostnames.

The below output shows the DNS service IP address, which is in turn mapped to a pod or a set of pods running coreDNS under the namespace kube-system. When the user spawns pods, their DNS setting will point to this IP address.

```
root@single:~# kubectl exec –ti dnsutils –– cat /etc/resolv.conf
search default.svc.single.cluster svc.single.cluster single.cluster
nameserver 10.233.0.3  <<<<
options ndots:5
root@single:~#
```

## Summary

This chapter covered the history of data center development and the current challenges related to networking. It also delved into containerization and its role in the modern data center. Additionally, the chapter presented an introduction to Kubernetes as a leading container orchestration platform, including its key concepts and terminology.

Chapter 2

# CN2 as a Kubernetes CNI

Juniper's Cloud Native Contrail Networking or CNCN/CN2, or simply *CN2*, is an overlay cloud native SDN solution. It is an evolution of a popular SDN solution called Contrail Networking, built from ground up to be cloud native. It offers a range of key capabilities, including cloud native networking, SDN for OpenStack and Kubernetes, NetOps-driven automation, multi-cluster management and scale, edge and remote compute, and unmatched advanced networking services. It integrates into K8s as a foundational piece of the cluster by utilizing the extension framework of custom resources. One can make use of all the popular tools of K8s such as kubectl, K9s, and its own custom GUI to monitor and manage a K8s+CN2 cluster.

## Advantages of CN2 When Used as a CNI in K8s Environments

One of CN2's key benefits is its simplicity and cloud native design, which makes it easy to operate and consistent across different clouds. Additionally, CN2 is DevOps-friendly and integrates seamlessly with existing workflows and processes. Advanced security features such as micro-segmentation, multi-tenant and namespace network isolation, and label-based security policies provide unparalleled security for your network.

Another benefit of CN2 is its ability to manage *multiple clusters* with one CN2 instance, and its multi-cluster policy federation for network/security and BGP cluster-to-cluster peering. This allows for easy management and scaling of your network across multiple clusters.

CN2 also utilizes ultra-fast, high-performance technologies such as kernel vRouter, DPDK vRouter, and SmartNIC to provide unmatched performance and speed. Overall, CN2 is a powerful, versatile, and secure solution for managing and scaling your network in a hybrid and multi-cloud environment.

To understand more about CN2 and its features, please use the following link: https://www.juniper.net/us/en/products/sdn-and-orchestration/contrail/cloud-native-contrail-networking-datasheet.html

## CN2 Architecture in K8s Environments

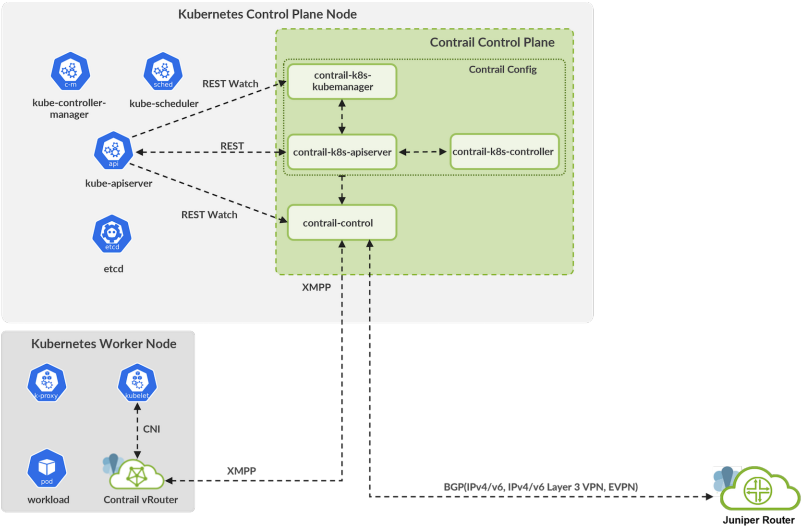



*Figure 2.1* *Cloud Native Contrail Networking with Kubernetes components*

As an SDN product, CN2 can be visualized in planes: viz, config, control, data, UI and analytics. Each plane consists of horizontally scalable nodes and each node consists of one or more microservices to perform its function.

Config and control functions are together termed as CN2 control plane, and it controls the networking on one or more compute nodes running in a cluster or across clusters.

Let's review the various microservices created by CN2 deployment on a K8s cluster.

### Contrail-k8s-apiserver

Contrail-k8s-apiserver extends the Kubernetes API to support contrail custom resources. It exposes REST and watches for any configuration resource changes. Pods, service, etc., are K8s native resources taken care of by kube apiserver and virtualnetwork, routinginstance, etc., and are custom resources whose definition is provided under contrail-k8s-server. The regular kube-apiserver forwards all network-related requests to the contrail-k8s-apiserver for handling.

### Contrail-k8s-controller

Contrail-k8s-controller applies business logic and converts the user intent into reality. The same is implemented as a reconciliation loop and constantly compares the intent with actual state, and in case of a mismatch, initiates tasks to fix it.

### Contrail-k8s-kubemanager

Contrail-k8s-kubemanager watches for any changes to regular Kubernetes resources such as service and namespace and interacts with contrail-k8s-controller to act upon any changes that affect the networking resources.

### Contrail-Control

The Contrail-control node performs two major functions:

- Configuration distribution: It receives the config from the config node via REST watch and creates a configuration graph. It further passes on the required config (partial graph) as per the need of respective compute nodes which can also be referred to as worker nodes in K8s terminology. The configuration graph available at the control node contains the entire cluster view whereas the partial graph refers to what an individual node should have.
- Route learning and distribution: This function of Contrail-control distributes routes between compute nodes and gateway router. It uses iBGP between the control nodes and XMPP with computes. It uses eBGP to peer with GW/routers to exchange routes with the external world. Refer to Figure 2.1

### Compute node or worker node

Compute node consists of a vRouter-agent and vRouter forwarding component which together form the data plane of the CN2.

The agent microservice performs control-related functions. It receives the config from the control node, converts the config, and sets up flow and interfaces with orchestration plug-ins like CNI for the forwarding component.

vRouter supports multiple forwarding modes like kernel loadable module, DPDK (at the time of writing this book, support for SmartNIC offload and eBPF is work-in-progress).

NOTE Telemetry node is an optional component that can be deployed based on the user's needs. It provides metrics, alarms info, logging, and flow analysis. It leverages services like Prometheus, Elastic, Fluentd, Kibana stack and Influx TSDB. All CN2 components produce telemetry data and the telemetry node exposes REST endpoints for users to plug-in their applications.

## CN2 as CNI in K8s cluster

CN2 is not just a CNI, it's a game-changing K8s SDN solution that brings next-level networking capabilities to your K8s clusters.

Before delving deeper into the topic, let's imagine CN2 in conjunction with Kubernetes components using Figure 2.2.

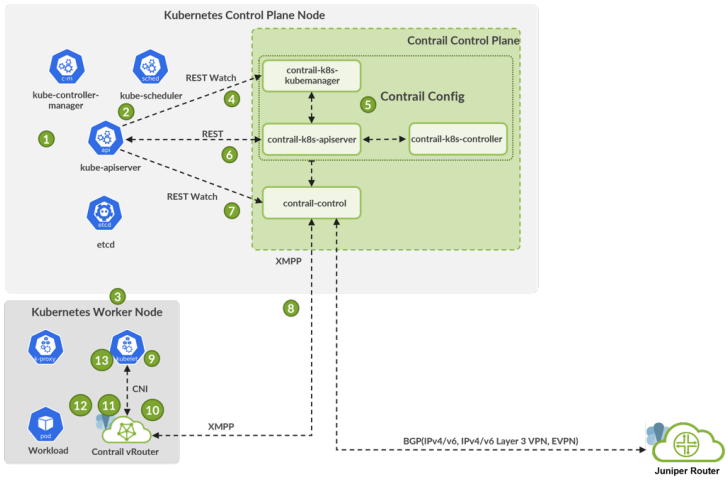



*Figure 2.2* *CN2 in Conjunction with Kubernetes*

A CNI, as discussed in Chapter 1, is a binary that implements networking interfaces to every pod created by the K8s cluster. Its functions involve creating network interfaces, assigning IP addresses, and attaching interfaces to the data plane.

As shown in Figure 2.2, the CN2 config and control components are installed alongside K8s components on the control node. The CN2 vRouter components are installed on each node alongside K8s components.

Let us understand what happens when an intent to create a pod is sent by a user, and how it is created:

1. A REST request is made via kubectl or any other API client. It submits the pod spec (or k8s object) to the kube-api-server which writes the object to etcd.

2. kube-scheduler watches for the changes and schedules the pod object on a node, updates kube-api-server which further updates the etcd store.

3. kube-api-server sends a request to Kubelet to create the said pod as per the user intent.

4/5. Contrail-k8s-kubemanager, using REST watch, listens for the intent i.e., the request/changes, and works with kube-api-server and kube-k8s-controller to translate the high-level intent into low-level details like VMI, IIP etc.

NOTE Contrail-k8s-controller watches for any changes to CN2 objects and reconciles until the desired intent is achieved for that object.

6. Contrail-k8s-apiserver write the translated low-level configuration to the etcd database via the kube-api-server.

7. Contrail-control, using REST watch, listens for changes from kube-apiserver. Once it notices the changes, it reads the etcd database to build the corresponding configuration map.

8. Contrail-control then pushes the required configuration map to the intended vRouter on the scheduled worker node.

9. From Step 3, Kubelet polls the configured CNI (CN2 in this case) for the networking information.

10. CNI (CN2 in this case) consults the vRouter agent for the pod config.

11. vRouter agent replies with VMI list as learnt from contrail-control. If the pod information is not already with vRouter, it polls the control node to fetch the required information.

12. CNI processes the VMI list and creates interfaces. It then attaches the interfaces to the pod.

13. CNI then requests VMI IP for each interface from vRouter agent and updates the kubelet about the same.

Refer to Juniper documentation using the link below. It explains the different ways CN2 can be deployed. For example, pod with DPDK workload, pod with classic workload, pod using multus to deploy a second interface like SRIOV. https://www.juniper.net/documentation/us/en/software/cn-cloud-native22.4/cn-cloud-native-k8s-install-and-lcm/index.html .

## CN2 Custom Resources

### Isolated Namespace

Kubernetes has a construct called *namespace* which is meant to isolate various resources of the cluster between a group of pods from others. However, default K8s NS does not prevent pods from one NS to communicate with the other NS. CN2 enhances traditional Kubernetes by providing isolated namespaces and custom default pod networks. Isolated namespaces allow network segmentation between pods by creating new default pod and service networks for each namespace. Isolated NS prevents communication between the pods spawned in isolated NS from other namespaces.

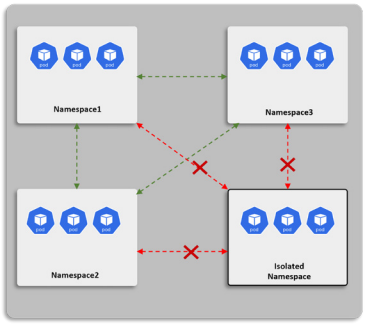



*Figure 2.3* *Communication of Pods Between Isolated Namespaces and Non-Isolated Namespaces*

## Virtual Network

By default, CN2 will use the CIDRs provided for pod and service networking in Kubernetes and create two virtual networks called default-pod-network and default-service-network. In the previous paragraph, we stated that isolated NS enables Kubernetes to create default pods and default service networks on a per NS basis. CN2 takes this further by enabling Kubernetes to create pods with their individual networks, based on the subnet parameters of a custom default pod network. This allows network isolation at both the namespace and pod level.

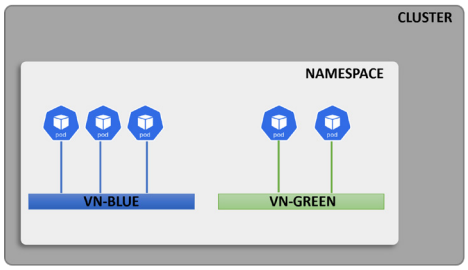



*Figure 2.4* *Pods in Their Custom Virtual Networks*

## VNR

A Virtual Network Router (VNR) in CN2 serves as a mechanism to establish communication between different virtual networks. This is achieved by a process known as route leaking, where routing instances and tables are imported into designated virtual networks.

Two types of network models are offered by CN2 VNR:

- Mesh: This model allows intercommunication between pods in all connected virtual networks.
- Hub-Spoke: In this model, pods in spoke virtual networks can communicate with all pods in the hub virtual network and vice versa. However, pods in different spoke virtual networks cannot communicate with each other.

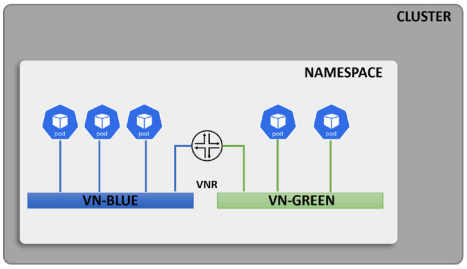
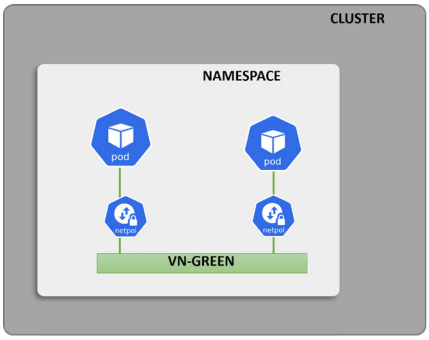



*Figure 2.5 Virtual network router enabling communication across VNs*

## Security Policy




Security policies in CN2 are used to block or allow specific ports between VNs when VNR is created. It can also be used within a VN to provide micro segmentation.

*Figure 2.6 Security policy providing micro-segmentation*

## BGPRouter

The BGPRouter, which is also referred to as the gateway router (GWR), is a device that is used to provide external connectivity for a cluster of devices; in this case, the CN2 cluster. The GWR acts as a gateway between the internal network of the CN2 cluster and the external networks, allowing devices within the cluster to communicate with devices outside the cluster.

### Floating IP

Floating IP is used to direct traffic towards pod(s) or a service from external networks. For example, a service-web-lb is a Kubernetes service, which is assigned an IP address from service-vn, and it is not reachable from external networks. To enable communication to this service from external hosts, create a VN called public-fip-vn and configure RT matching with external GW VRF to this VN. This will allow routes to be advertised between public-fip-vn and GW VRF. The IP address from this VN will be used to assign a floating IP to either a service or pod.

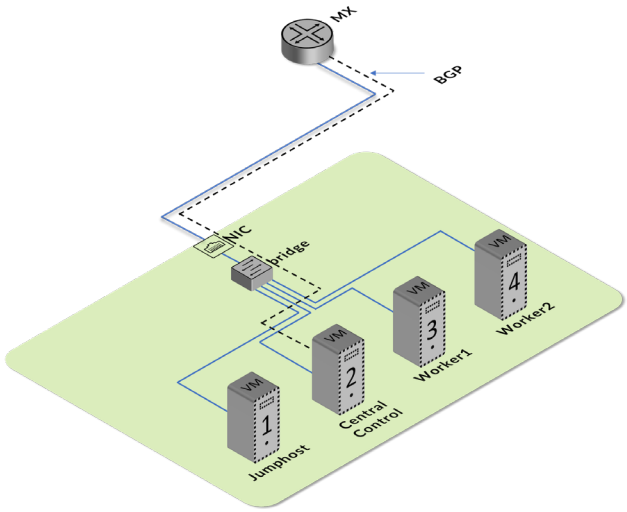



*Figure 2.7* *BGProuter: Session Between Contrail Controller and Gateway Router*

For this to work properly, it is necessary to configure matching route-targets on both the VN and the GWR. Route-targets are used to identify the routes that are exported or imported between different networks. By configuring matching route-targets on the VN and the GWR, you can ensure that the VN and the GWR are able to communicate with each other and exchange routing information. This allows devices within the VN to communicate with external devices through the GWR and vice versa.

To enable the exchange of routes between the CN2 cluster and the gateway router (GWR), it is necessary to create a configuration object called a BGPRouter. The BGPRouter object contains details such as the IP address and router ID of the GWR, the autonomous system (AS) number of the GWR, the type of router, and the address family (e.g., inet-vpn) that the GWR supports. The BGPRouter object also includes the route-target, which is used to identify the routes that are exported or imported between different networks.

In addition to creating the BGPRouter object on the CN2 cluster, it is also necessary to configure the GWR to peer with the CN2 control nodes. This can be done by specifying the IP addresses and other details of the CN2 control nodes in the GWR's configuration. Once this configuration is in place, the GWR and the CN2 control nodes should be able to establish a BGP (Border Gateway Protocol) session with each other.

Once the BGP session is established, you should be able to see the BGP status on both the CN2 cluster and the GWR as *established*. This indicates that the two devices can exchange routing information with each other and that they are able to communicate using BGP.

Overall, the process of configuring the CN2 cluster and the GWR to exchange routes involves creating a BGPRouter object on the CN2 cluster and configuring the GWR to peer with the CN2 control nodes. Once this configuration is in place, the two devices should be able to establish a BGP session and exchange routing information. Chapter 4 will be covering the configuration of both GWR and the CN2 cluster to peer with each other for exchanging routes.

### VMI

Virtual Machine Interface or VMI is a representation of a port or interface which belongs to a virtual network. It may or may not have an associated pod/virtual machine to it.

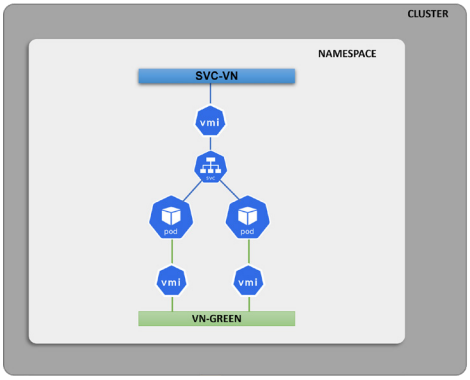



*Figure 2.8* *Virtual Machine Interface*

### IIP

Instance IP, or IIP, is an IP address associated with a subnet of a VN. This can be IPv4 or IPv6. IIP is further associated with the VMI.

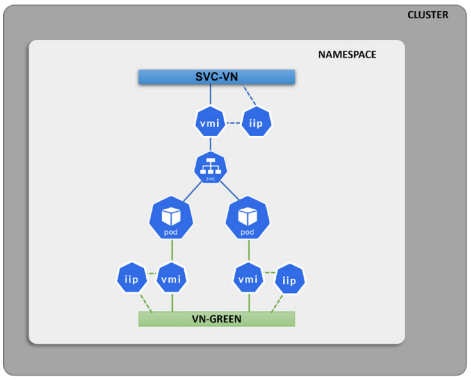



*Figure 2.9* *Instance IP*

## CN2 Deployment Models

CN2 deployment models can broadly be classified in two types: single cluster and multi-cluster.

In single cluster model, the control plane components of both K8s and CN2 are exclusive to this cluster. See Figure 2.10.

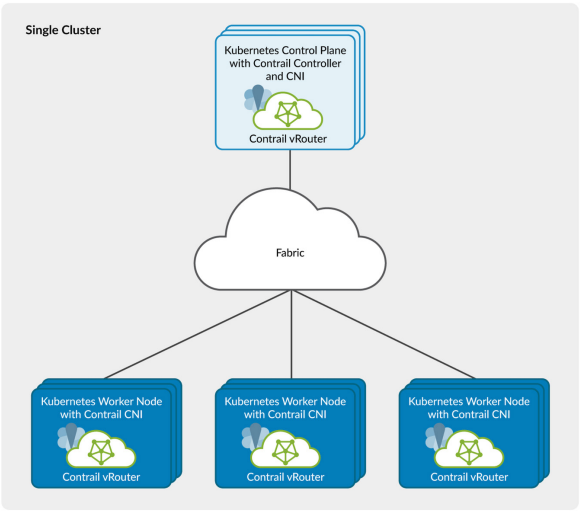



*Figure 2.10* *Kubernetes with CN2 as CNI in Single Cluster Deployment*

In the multi-cluster model, each cluster will have its own K8s control components running. However, the CN2 control plane components will reside only on the central cluster. Each worker residing in either central or workload cluster will be hosting vrouter-agent and vrouter to provide SDN capabilities.

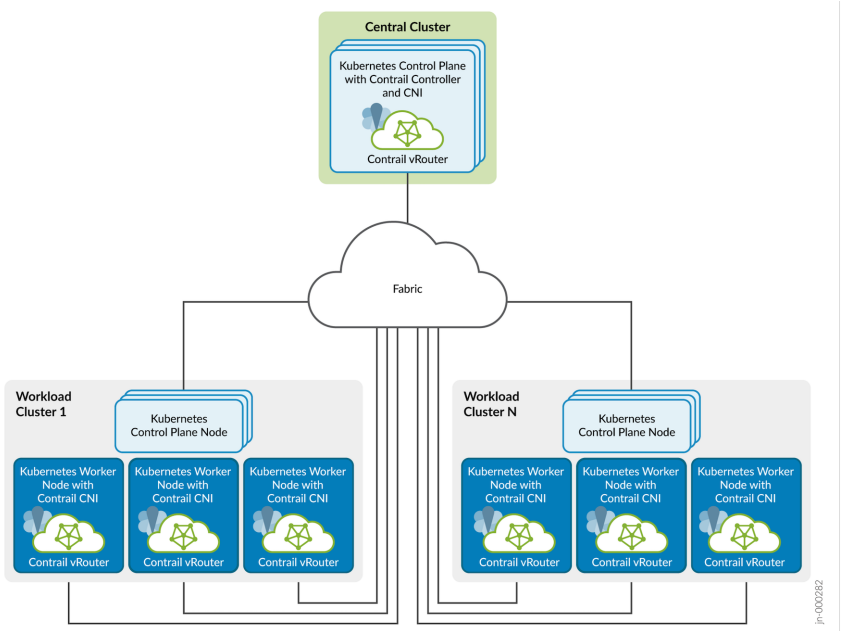



*Figure 2.11* *Kubernetes with CN2 as CNI in Multi-cluster Deployment*

Single cluster can also be deployed with two interfaces on each node – one for management and the other for data+control. Similar topology is not supported for multi- cluster deployments.

For more information on this, refer to: https://www.juniper.net/documentation/us/en/software/cn-cloud-native22.3/cn-cloud-native-k8s-install-and-lcm/topics/topic-map/cn-cloud-native-deployment-models.html .

# Chapter 3

# Installing and Getting Familiar with CN2

Here's a checklist to review before attempting to install CN2:

- Bare Metal Server(s) or BMSs to host VMs.
- Alternately, you would need four VMs for central site and four VMs for workload site spread across any number of BMSs.
- Supported Ubuntu VM image. Supported OS and Kernel versions are listed in the PDF file which can be accessed using this link: https://www.juniper.net/documentation/en_US/release-independent/contrail-cloud-native/topics/reference/cloud-native-contrail-supported-platforms.pdf.
- Subnet with at least four IPs for each cluster that can be assigned to VMs.
- The above-mentioned IPs should be able to access the Internet for downloading packages, manifests, and containers.
- Credentials to access the enterprise-hub.juniper.net for downloading container images – request a free demo: https://www.juniper.net/us/en/forms/cn2-free-trial.html
- Access to the git repo https://github.com/Juniper/cn2dayone to download installation playbooks and example manifests for application.
- Juniper MX series router or any other router which supports MP-BGP routing protocol and MPLSoGRE as tunneling encapsulation.

Figure 3.1 illustrates a central cluster setup using K8s and CN2. The VMs shown in the diagram represent the Jumphost, Central, Worker1, and Worker2 nodes that make up the cluster. All these VMs are connected to a bridge, which in turn, is connected to a network segment of a particular subnet. This subnet is part of the interface of a router, which facilitates communication between the cluster and the rest of the network.

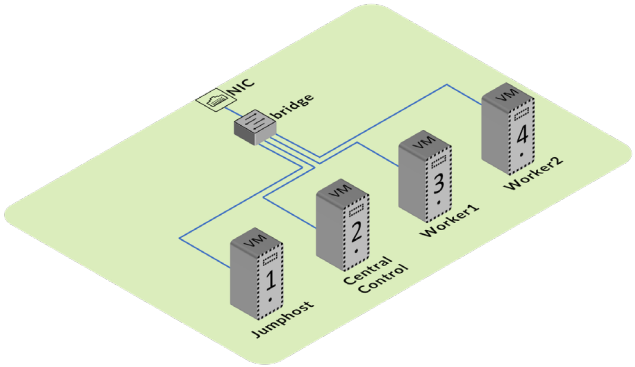



*Figure 3.1* *CN2 Central Cluster Basic Setup*

Additionally, there is another bare metal server that contains similar VMs. However, this set of nodes or cluster will be referred to as the workload cluster.

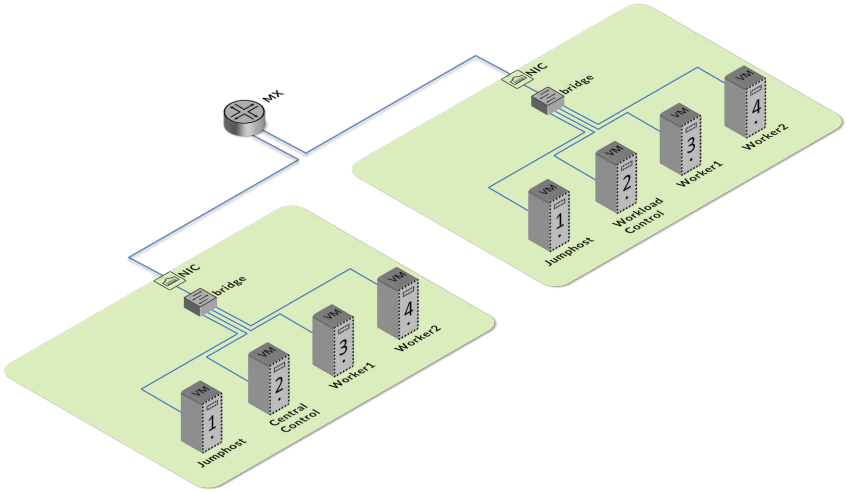



*Figure 3.2* *The Workload Cluster*

# Setting Up Infrastructure for CN2 Deployment

To resemble a setup which would be as close as a production one, you need to install two clusters residing on two individual BMSs. You can also create the two clusters on a single BMS provided that each set of VM/nodes have their own bridge for networking to connect to external devices. The BMSs should be installed with Ubuntu 20.04 and have enough storage in the root partition to host VMs; around 400 GB of storage should be sufficient. Note that the exact steps and functioning of the installation playbooks might differ depending on your specific installation and configuration. So, it would be preferable to use the same installation method as we have used in the example.

Before attempting to install the CN2 as CNI for K8s, we will set up the physical cabling based on Figure 3.3.

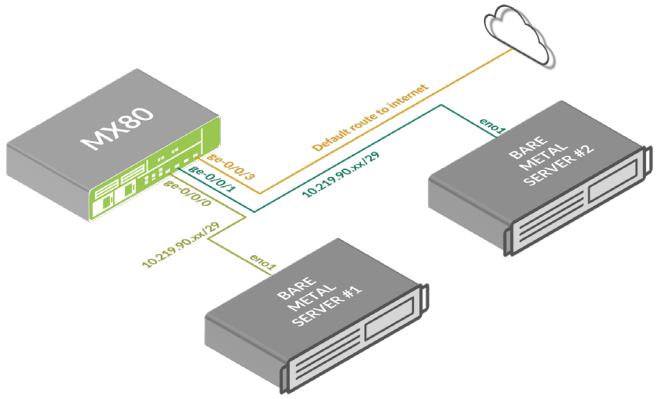



*Figure 3.3* *BMS and MX Physical Connectivity*

On BMS#1, create virtual machines to be used for central cluster. This cluster will host the Kubernetes control and worker nodes as well as the CN2 control plane. The worker nodes in the central cluster will run the vrouter and vrouter-agent pods to provide software-defined networking (SDN) capabilities.

On BMS#2, create virtual machines to set up a distributed cluster, also known as *workload cluster*. This cluster will have its own Kubernetes control node but will not include any Contrail Networking control components. The worker nodes in the distributed cluster will also run the vrouter and vrouter-agent pods to provide SDN capabilities.

### Preparing the Network

The networking infrastructure for this deployment requires the creation of two subnets, connected to a common routing domain, to allow communication between the CIDRs and the rest of the internet for downloading container images and other dependencies. Routing between the CIDRs is also required for communication between clusters placed on each of the bare metal servers. It's important to note that these requirements are not strict guidelines and can be adapted as needed to fit the specific needs of the deployment. However, one must understand the desired state to make any necessary changes.

### Preparing the BMS to Support Virtualization

Ensure that the two bare metal servers (BMSs) hosting the various nodes of K8s and CN2 cluster as virtual machines (VMs) are properly installed with all required packages and dependencies. This will ensure smooth deployment and operation of the cluster.

Check if the BMSs BIOS/UEFI is enabled to support KVM:

```
root@cftsbmsr2003:~# kvm-ok
INFO: /dev/kvm exists
KVM acceleration can be used
root@cftsbmsr2003:~#
```

Check if the packages required to support virtualization are installed:

```
root@cftsbmsr2003:~# apt list | egrep "qemu-kvm|libvirt-daemon/|libvirt-clients|virtinst|virt-
manager|bridge-util|libguestfs-tools"

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

bridge-utils/focal,now 1.6-2ubuntu1 amd64 [installed]
libguestfs-tools/focal,now 1:1.40.2-7ubuntu5 amd64 [installed]
libvirt-clients/focal-updates,focal-security,now 6.0.0-0ubuntu8.16 amd64 [installed]
libvirt-daemon/focal-updates,focal-security,now 6.0.0-0ubuntu8.16 amd64 [installed,automatic]
qemu-kvm/focal-updates,focal-security,now 1:4.2-3ubuntu6.23 amd64 [installed]
virt-manager/focal-updates,now 1:2.2.1-3ubuntu2.1 all [installed]
virtinst/focal-updates,now 1:2.2.1-3ubuntu2.1 all [installed]
root@cftsbmsr2003:~#
```

If any of the packages or none of them are installed during your BMS installation, you can install them using the following procedure or *apt-get install <package-name>*:

```
root@cftsbmsr2003:~# apt install qemu-kvm libvirt-daemon-system libvirt-clients bridge-
utils virtinst virt-manager libguestfs-tools
root@cftsbmsr2003:~# systemctl is-active libviwrtd
root@cftsbmsr2003:~# usermod -aG libvirt $USER
root@cftsbmsr2003:~# usermod -aG kvm $USER
```

NOTE This verification step must be performed on both bare metal servers hosting the VMs.

### Creating bridges on BMS to provide network connectivity for the virtual machines

Bridges on Ubuntu can be created with *brctl* command. Bridges created with this command will disappear once the server is restarted. To hardcode the server to maintain the required bridges, you can either use *virsh edit net-edit* option or just add the bridge to the yaml file under */etc/netplan*.

Below are the contents of the */etc/netplan/00-installer-config.yaml* used in the setup for demonstration:

```
network:
  version: 2
  renderer: networkd

  ethernets:
    enp3s0f0:
      dhcp4: false
      dhcp6: false


  bridges:
    mgmt:
      interfaces: [enp3s0f0]
      addresses: [10.219.90.79/26]
      gateway4: 10.219.90.65
      mtu: 1500
      nameservers:
        addresses: [66.129.233.81]
      parameters:
        stp: true
        forward-delay: 4
      dhcp4: no
      dhcp6: no
```

Observe that we are configuring the static IP address to the bridge called *mgmt* and using the interface *enp3s0f0* as a child interface of the bridge.

## Installing K8s Cluster with CN2 as a CNI

Let's begin deploying the first cluster with K8s with CN2 as CNI. This cluster will later act as our central cluster.

On BMS#1, four virtual machines will be created and referred to as *nodes*. These nodes will be connected to each other and the outside world through a Linux bridge. One of these nodes will be used for orchestrating the cluster and is referred to as the KubeSprayHost. The remaining three nodes will be referred to as k8s-control, k8s-worker1, and k8s-worker2, respectively.

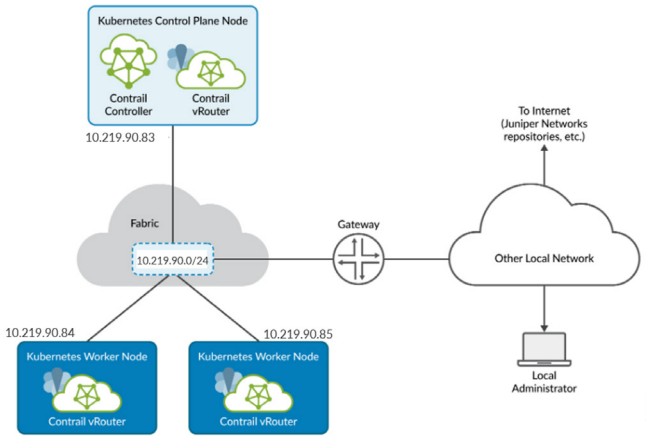



*Figure 3.4 Network Plan for Installing CN2 with Single Interface*

Depending on the CN2 version you are attempting to install, the recommended OS and version needs to be installed in nodes for proper functioning. Here's a link to CN2-supported platforms: https://www.juniper.net/documentation/en_US/release-independent/contrail-cloud-native/topics/reference/cloud-native-contrail-supported-platforms.pdf .

At the time of writing this document, the latest version of CN2 is CN2 22.4 and the recommended OS and version for upstream Kubernetes is Ubuntu 20.04.3 with kernel version 5.4.0-97-generic/5.4.0-135-generic.

Download the KVM image of Ubuntu 20.04.3 with kernel 5.4.0-97-generic using the following command on the BMS:

```
root@cftsbmsr2003:~# wget -qO ubuntu-20.04-server-cloudimg-amd64.img http://cloud-images-archive.
ubuntu.com/releases/focal/release-20220131/ubuntu-20.04-server-cloudimg-amd64.img
root@cftsbmsr2003:~# wget -qO MD5SUMS http://cloud-images-archive.ubuntu.com/releases/focal/
release-20220131/MD5SUMS
Verify that the downloaded VM image has been successfully downloaded by verifying the MD5 checksum of
image against the original checksum provided in the MD5SUMS file
root@cftsbmsr2003:~# md5sum ubuntu-20.04-server-cloudimg-amd64.img
5a487f53da7c9f1a5b59769b3463282e  ubuntu-20.04-server-cloudimg-amd64.img
root@cftsbmsr2003:~# cat MD5SUMS | grep ubuntu-20.04-server-cloudimg-amd64-disk-kvm.img
5a487f53da7c9f1a5b59769b3463282e *ubuntu-20.04-server-cloudimg-amd64-disk-kvm.img
```

Now that the baremetal servers (BMSs) are ready and the KVM image from Ubuntu archives is sourced, we can start deploying Kubernetes and CN2 in the central cluster.

The process of deploying a K8s cluster with CN2 as CNI involves several steps to define networking information like CIDR, gateway, and DNS for the cluster, modifying the Ubuntu image to increase its size and configure the root credentials, hostname, SSH, and network settings, spawning VMs, increasing the partition size from the default 8GB to the maximum set in the image, and building a host file for accessing the nodes using their FQDN/hostname. The jumphost is prepared to act as the host from which the

Kubespray will deploy the K8s cluster and trigger the installation of K8s and CN2 as the K8s CNI for the cluster named Central. To simplify the process, we have automated the deployment process into easy ansible playbooks that can be cloned from a git repository, and a workflow diagram is provided to explain the installation process of Kubernetes and CN2 as CNI.

NOTE Some stages of the diagram could be a simple execution of the playbook, while some may require two or more commands to be executed before moving to the next stage.

Navigate to your bare metal server shell prompt and git clone the following repository:

```
git clone https://github.com/Juniper/cn2dayone.git
```

This git repo has ansible playbooks and .MD files containing troubleshooting links and other content. We will update this repo frequently to make this day one book content relevant for the upcoming release of CN2.

When you execute the git clone command, a folder with the name cn2dayone gets created and it contains folders named Release and the litmustest. Deployment playbooks for each release can be found under the Release folder.

For example, we will use folder 22.4 as it was the latest CN2 release at the time of authoring this book.

Navigate to the folder 22.4 using cd command

```
cn2dayone/Release# cd 22.4
```

Navigate to cn2_central_ansible under 22.4 folder.

```
root@ubuntu:~/cn2dayone/Release/22.4#
```

The cn2_central_ansible contains the following files:

```
-rw-r--r-- 1 root root 3968 Jan 13 16:17 1Play_VM_Creation.yaml
-rw-r--r-- 1 root root 3351 Jan 13 16:13 2Play_VM_Disk_Resize.yaml
-rw-r--r-- 1 root root 3794 Jan 13 16:13 3Play_jumphost.yaml
-rw-r--r-- 1 root root 1670 Jan 13 16:13 4Play_CNI_less_Cluster.yaml
-rw-r--r-- 1 root root 3798 Jan 13 16:13 5Play_CN2_Cluster.yaml
-rwxr-xr-x 1 root root  165 Jan 13 16:14 destroy_cluster.sh
-rw-r--r-- 1 root root 1136 Jan 13 16:13 inventory.yaml
-rw-r--r-- 1 root root 1381 Jan 13 16:13 k8s_inventory.yaml
-rwxr-xr-x 1 root root  556 Jan 13 16:14 network_yaml_create.sh
-rw-r--r-- 1 root root 1872 Jan 11 08:41 README.md
```

The first five files are the playbooks to automate the installation of K8s CN2 cluster. Also, there are two inventory files, inventory.yaml used by custom playbooks and k8s_inventory.yaml for the Kubernetes cluster. You need to edit these files to match your CN2 deployment topology. There is a shell script which takes input from inventory.yaml and generates network yaml(ubuntu) for the VMs. This script is executing as part of first playbook. Lastly, we have a destroy_cluster.sh used in an event cluster that needs to be cleaned up for attempting installation once again.

## Knowing the Five Deployment Playbooks

These five playbooks would be your best companion for your journey to deploy/redeploy CN2:

- Playbook 1 creates the VM images, customizes them and sets up the network. It then spawns the VMs and starts them.
- Playbook_2 resizes the VM disk size. This is required as the default image size is only 2GB.
- Playbook 3 installs all the required packages on first VM (node) to enable execution of playbooks 4 and 5. After the execution of this playbook, the VM turns into a KubeSpray host. This node will be used by other playbooks to install Kubernetes on the remaining hosts.
- Playbook 4 uses kubespray and the k8s inventory.yaml file to deploy a Kubernetes cluster without a CNI.
- Playbook 5 requests the user to enter the enterprise-hub.juniper.net credentials to pull the required CN2 images from the repository. It then clones and updates the yaml manifests with the token code created using the credentials. Finally, it deploys CN2 using the updated manifests.

So, five ansible command executions using these five yaml playbooks, is all it takes to install a CN2 cluster. Now, let's install CN2 by going through each step in detail.

### Populate inventory.yaml to reflect network infrastructure

Edit the inventory.yaml file on Linux host using an editor of your choice.

Example: vi

The inventory.yaml should contain the subnet you have identified. This would be used to assign IP addresses to the VMs. Remember to update the password and set it as per your deployment. If you intend to deploy a distributed cluster as well, it is advisable to add the IPs for the distributed cluster too. Example:

```
allcentral:
    hosts:
      centraljumphost:
        ansible_host: 10.219.90.82
      centralctrl:
        ansible_host: 10.219.90.83
      centralworker1:
        ansible_host: 10.219.90.84
      centralworker2:
        ansible_host: 10.219.90.85
    vars:
      root: juniper123
```

```
alldsi:
    hosts:
      ds1jumphost:
        ansible_host: 10.219.90.88
      ds1ctrl:
        ansible_host: 10.219.90.89
      ds1worker1:
        ansible_host: 10.219.90.90
      ds1worker2:
        ansible_host: 10.219.90.91



jumphost:
    hosts:
      centraljumphost:
        ansible_host: 10.219.90.82
      ds1jumphost:
        ansible_host: 10.219.90.88
    vars:
      root: juniper123

centralk8svm:
    hosts:
      centralctrl:
        ansible_host: 10.219.90.83
      centralworker1:
        ansible_host: 10.219.90.84
      centralworker2:
        ansible_host: 10.219.90.85

    vars:
      root: juniper123

ds1k8svm:
    hosts:
      ds1ctrl:
        ansible_host: 10.219.90.89
      ds1worker1:
        ansible_host: 10.219.90.90
      ds1worker2:
        ansible_host: 10.219.90.91

    vars:
      root: juniper123
```

### Updating Subnet and DNS Details

Open the network_yaml_create.sh and confirm the subnet and DNS of your environment. Change if required to match your network.

```
#!/bin/bash
#Script to generate network.yaml file for the nodes being configured.
node_list=( centraljumphost centralcontrol centralworker1 centralworker2 )

node_gw=$(/sbin/ip route | awk '/default/ { print $3 }')
env_dns=66.129.233.81
```

```
for node in ${node_list[@]}; do
node_ip=$( host $node | awk '/has address/ { print $4 }')
cat > $(pwd)/$node.yaml <<FILE
 network:
  ethernets:
    enp1s0:
      dhcp4: no
      dhcp6: no
      addresses: [$node_ip/26]
      gateway4:  $node_gw
      nameservers:
        addresses: [$env_dns]
  version: 2
FILE
done
```

## Executing Playbook 1 to Create VMs

This playbook would take inputs from the inventory file and build VMs to be used as cluster nodes.

Run the Playbook_1: 1Play_VM_Creation.yaml

```
# ansible-playbook -i inventory.yaml 1Play_VM_Creation.yaml
```

Output:

```
PLAY [Playbook to prepare VMs images, network, and spawn the VMs] ***********************************
*******************************

TASK [Gathering Facts] ********************************************************************
ok: [localhost]
--SNIP--

PLAY RECAP ********************************************************************************************************
*****************************************
localhost              : ok=9    changed=5    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
Verification Task#1 : Run command "virsh list –all | grep central"
Output :-
473   centraljumphost    running
 474   centralcontrol     running
 475   centralworker1     running
 476   centralworker2     running
```

### Verification of VM Creation

```
# ssh centraljumphost"
```

Output:

The user should be able to login without getting prompted for a password and kernel version should be 5.4.0-97-generic.

## Resizing VM disks

This playbook resizes the disks of all the VMs created in the previous step. This is necessary because the downloaded cloud image has only 2GB of disk space.

Run the Playbook_2:

```
#ansible-playbook -i 2Play_VM_Disk_Resize.yaml
```

Output:

```
PLAY [Update host file among all VMs and resize disk] ***********************************************
******************

TASK [Gathering Facts] ******************************************************************
ok: [centraljumphost]
ok: [centralcontrol]
ok: [centralworker1]
ok: [centralworker2]
--SNIP--
ok: [centraljumphost] => (item={'mount': '/boot/efi', 'device': '/dev/vda15', 'fstype': 'vfat',
'options': 'rw,relatime,fmask=0077,dmask=0077,codepage=437,iocharset=iso8859-
1,shortname=mixed,errors=remount-ro', 'size_total': 109422592, 'size_available': 103973888, 'block_
size': 512, 'block_total': 213716, 'block_available': 203074, 'block_used': 10642, 'inode_total': 0,
'inode_available': 0, 'inode_used': 0, 'uuid': 'BA5D-627F'}) => {
~
    "msg": "All assertions passed"
}
failed: [centralcontrol] (item={'mount': '/', 'device': '/dev/vda1', 'fstype': 'ext4', 'options':
'rw,relatime', 'size_total': 2107494400, 'size_available': 688410624, 'block_size': 4096, 'block_
total': 514525, 'block_available': 168069, 'block_used': 346456, 'inode_total': 274176, 'inode_
available': 201558, 'inode_used': 72618, 'uuid': '00b72c14-e32b-4e33-988c-002ba91aafec'}) => {
~
    "msg": "Assertion failed"
}

PLAY RECAP ******************************************************************
centralcontrol           : ok=5    changed=3    unreachable=0    failed=0    skipped=0    rescued=0
ignored=1
centraljumphost          : ok=5    changed=3    unreachable=0    failed=0    skipped=0    rescued=0
ignored=1
centralworker1           : ok=5    changed=3    unreachable=0    failed=0    skipped=0    rescued=0
ignored=1
centralworker2           : ok=5    changed=3    unreachable=0    failed=0    skipped=0    rescued=0
ignored=1
```

### Verify VM Disk Resizing

Verification: Log in to centraljumphost and confirm disk expansion.

```
ssh centraljumphost
root@centraljumphost:~# df -H
Filesystem      Size  Used Avail Use% Mounted on
udev             17G     0   17G   0% /dev
tmpfs           3.4G  1.2M  3.4G   1% /run
/dev/vda1        55G  1.5G   53G   3% /  <<<<<<<<
```

```
tmpfs            17G     0   17G   0% /dev/shm
tmpfs           5.3M     0  5.3M   0% /run/lock
tmpfs            17G     0   17G   0% /sys/fs/cgroup
/dev/vda15      110M  5.5M  104M   5% /boot/efi
/dev/loop1       66M   66M     0 100% /snap/core20/1328
/dev/loop0       71M   71M     0 100% /snap/lxd/21835
/dev/loop2       46M   46M     0 100% /snap/snapd/14549
tmpfs           3.4G     0  3.4G   0% /run/user/0
```

## Installing Packages on Jumphost

With the execution of this playbook, Jumphost is installed with various packages like Ansible, Kubespray, Kubectl, Helm, etc.

Run the Playbook_3:

```
ansible-playbook -i  3Play_jumphost.yaml
ansible-playbook -i inventory.yaml 3Play_JumpHost.yaml
```

Output:

```
PLAY [Install required packages on Jumpshost] ***************************************************
***********

TASK [Gathering Facts] ******************************************************************
ok: [centraljumphost]

TASK [Add apt-key to update kubernetes repo] ****************************************************
**********
changed: [centraljumphost]

--SNIP---
PLAY RECAP ******************************************************************
centraljumphost          : ok=16   changed=13   unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
centralcontrol           : ok=4    changed=3    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
centralworker1           : ok=4    changed=3    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
centralworker2           : ok=4    changed=3    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

### Verification Task #1: Confirm Ansible is running

```
root@centraljumphost:~# ansible --version
ansible [core 2.12.5]
  config file = None
  configured module search path = ['/root/.ansible/plugins/modules', '/usr/share/ansible/plugins/
modules']
  ansible python module location = /usr/local/lib/python3.8/dist-packages/ansible
  ansible collection location = /root/.ansible/collections:/usr/share/ansible/collections
  executable location = /usr/local/bin/ansible
  python version = 3.8.10 (default, Nov 14 2022, 12:59:47) [GCC 9.4.0]
  jinja version = 2.11.3
  libyaml = True
```

### Verification Task #2

Log in to Jumphost and confirm if you can login to control and worker nodes without password.

```
ssh centraljumphost
ssh centralcontrol
ssh centralworker1
ssh centralworker2
```

## Install K8s Without CNI

With the completion of this step, Kubespray will install a cluster on the remaining three VMs.

Run the 4Play_CNI_less_Cluster.yaml

```
ansible-playbook -i inventory.yaml 4Play_CNI_less_Cluster.yaml
Output:
PLAY [Install k8s cluster without CNI] ***************************************************************
*********************
--SNIP--
PLAY RECAP *******************************************************************************************
centraljumphost          : ok=9    changed=8    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

### Verification of K8s Installation

```
Login to centraljumphost:- ssh centraljumphost
Run command : kubectl get pods -A
root@centraljumphost:~# kubectl get pods -A
NAMESPACE     NAME                                      READY   STATUS    RESTARTS      AGE
kube-system   coredns-74d6c5659f-z8bqs                  0/1     Pending   0             70s
kube-system   dns-autoscaler-59b8867c86-jbqmf           0/1     Pending   0             66s
kube-system   kube-apiserver-centralmaster              1/1     Running   1             2m54s
kube-system   kube-controller-manager-centralmaster     1/1     Running   2 (29s ago)   2m54s
kube-system   kube-proxy-76kjb                          1/1     Running   0             102s
kube-system   kube-proxy-fhst5                          1/1     Running   0             102s
kube-system   kube-proxy-p7rnj                          1/1     Running   0             102s
kube-system   kube-scheduler-centralmaster              1/1     Running   2 (29s ago)   2m54s
kube-system   nginx-proxy-centralworker1                1/1     Running   0             90s
kube-system   nginx-proxy-centralworker2                1/1     Running   0             90s
root@centraljumphost:~# kubectl get nodes -A -o wide
NAME           STATUS    ROLES          AGE    VERSION   INTERNAL-IP    EXTERNAL-IP   OS-IMAGE
KERNEL-VERSION    CONTAINER-RUNTIME
centralmaster   NotReady   control-plane   3m40s   v1.24.6   10.219.90.83   <none>       Ubuntu 20.04.3
LTS   5.4.0-97-generic   cri-o://1.24.4
centralworker1   NotReady   <none>         2m26s   v1.24.6   10.219.90.84   <none>       Ubuntu 20.04.3
LTS   5.4.0-97-generic   cri-o://1.24.4
centralworker2   NotReady   <none>         2m26s   v1.24.6   10.219.90.85   <none>       Ubuntu 20.04.3
LTS   5.4.0-97-generic   cri-o://1.24.4
```

## Installing CN2 as CNI for K8s Cluster

The execution of this playbook deploys CN2 22.4 as a CNI on the K8s cluster installed in the previous step.

Run the 5Play_CN2_Cluster.yaml

```
ansible-playbook -i inventory.yaml 5Play_CN2_Cluster.yaml
Enter Docker Username?: cn2dayone@juniper.net
Enter Password?:

PLAY [Install required packages on Jumpshost] ***********************************************************
******************************
--SNIP--
PLAY RECAP ***************************************************************************************
centraljumphost          : ok=15   changed=4    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

### Verification of CN2 Installation in K8s Cluster

```
Run command: kubectl get pods -A
root@centraljumphost:~# kubectl get pods -A
NAMESPACE       NAME                                          READY   STATUS     RESTARTS     AGE
cert-manager     cert-manager-745fb764f4-jd9zk                    1/1    Running    0           17m
cert-manager     cert-manager-cainjector-5654f68b7b-v96j5         1/1    Running    0           17m
cert-manager     cert-manager-webhook-fff46dd94-bmzg6             1/1    Running    0           17m
contrail-deploy   contrail-k8s-deployer-58c49b55d5-8nzz9          1/1    Running    0           18m
contrail-system   contrail-k8s-apiserver-54949d5455-zk27n         1/1    Running    0           17m
contrail-system   contrail-k8s-cert-gen-job-create-8qb8c          0/1    Completed   0
18m
contrail-system   contrail-k8s-controller-79cc75db9-849jh         1/1    Running    0           16m
contrail        contrail-control-0                             2/2    Running    0           16m
contrail        contrail-k8s-contrailstatusmonitor-fd97d69-bblwk   1/1    Running    0
16m
contrail        contrail-k8s-kubemanager-6ddbcd597d-qrs99        1/1    Running    0           16m
contrail        contrail-vrouter-masters-ntfbj                  3/3    Running    0           16m
contrail        contrail-vrouter-nodes-62ppm                    3/3    Running    0           16m
contrail        contrail-vrouter-nodes-ldqp4                    3/3    Running    0           16m
kube-system      coredns-74d6c5659f-fl8c7                       1/1    Running    0           12m
kube-system      coredns-74d6c5659f-z8bqs                       1/1    Running    0           22m
kube-system      dns-autoscaler-59b8867c86-jbqmf                 1/1    Running    0           22m
kube-system      kube-apiserver-centralmaster                    1/1    Running    1           24m
kube-system      kube-controller-manager-centralmaster           1/1    Running    2 (21m ago)
24m
kube-system      kube-proxy-76kjb                               1/1    Running    0           23m
kube-system      kube-proxy-fhst5                               1/1    Running    0           23m
kube-system      kube-proxy-p7rnj                               1/1    Running    0           23m
kube-system      kube-scheduler-centralmaster                    1/1    Running    2 (21m ago)   24m
kube-system      nginx-proxy-centralworker1                      1/1    Running    0           22m
kube-system      nginx-proxy-centralworker2                      1/1    Running    0           22m
```

# Getting Familiar with the Cluster

Congratulations on deploying the K8s cluster alongside CN2 as a CNI. Before we start deploying the application as a workload on this cluster, let us get familiar with it and check the basic constructs of K8s and Contrail.

## Accessing the jump server

Log into the jump server by using ssh or virsh console <vm id>.

To confirm that you are on the right machine, enter the command *kubectl get nodes*:

```
root@centraljumphost:~# kubectl get nodes -A -o wide
NAME           STATUS   ROLES          AGE    VERSION   INTERNAL-IP    EXTERNAL-IP   OS-IMAGE
KERNEL-VERSION     CONTAINER-RUNTIME
centralctrl     Ready    control-plane   4h6m   v1.24.6   10.219.90.83   <none>       Ubuntu 20.04.5 LTS
5.4.0-135-generic   cri-o://1.24.4
centralworker1   Ready    <none>         4h5m   v1.24.6   10.219.90.84   <none>       Ubuntu 20.04.5 LTS
5.4.0-135-generic   cri-o://1.24.4
centralworker2   Ready    <none>         4h5m   v1.24.6   10.219.90.85   <none>       Ubuntu 20.04.5 LTS
5.4.0-135-generic   cri-o://1.24.4
```

Expected outcome: One should see three nodes with names k8s-cp0, k8s-worker0, and k8s-worker1, with STATUS in Ready state.

## Verify the cluster status

All K8s pods would run in the kube-system namespace and the contrail pods would run in Contrail and contrail-system namespace. We need to verify that all pods are in running state.

```
root@centraljumphost:~# kubectl get pods -A
NAMESPACE       NAME                                          READY   STATUS     RESTARTS      AGE
cert-manager     cert-manager-745fb764f4-fj48s                  1/1    Running    0            4h
cert-manager     cert-manager-cainjector-5654f68b7b-4vvzk        1/1    Running    0            4h
cert-manager     cert-manager-webhook-fff46dd94-947b6            1/1    Running    0           4h
contrail-deploy   contrail-k8s-deployer-58c49b55d5-5vzz7          1/1    Running    0           4h1m
contrail-system   contrail-k8s-apiserver-785ffdf895-gmr8q         1/1    Running    0
3h59m
contrail-system   contrail-k8s-cert-gen-job-create-rz9wr          0/1    Completed   0           4h
contrail-system   contrail-k8s-controller-79cc75db9-ncqk8         1/1    Running    0
3h58m
contrail        contrail-control-0                            2/2    Running    0           3h58m
contrail        contrail-k8s-contrailstatusmonitor-fd97d69-nf546   1/1    Running    0
3h58m
contrail        contrail-k8s-kubemanager-6ddbcd597d-xv7zm        1/1    Running    0           3h58m
contrail        contrail-vrouter-masters-v9kjz                 3/3    Running    0           3h58m
contrail        contrail-vrouter-nodes-58528                   3/3    Running    0           3h58m
contrail        contrail-vrouter-nodes-wsrct                   3/3    Running    0           3h58m
kube-system      coredns-74d6c5659f-96hmr                      1/1    Running    0           3h56m
kube-system      coredns-74d6c5659f-rs7vb                      1/1    Running    0           4h6m
kube-system      dns-autoscaler-59b8867c86-4m2rh                1/1    Running    0           4h6m
kube-system      kube-apiserver-centralctrl                    1/1    Running    1           4h8m
```

```
kube-system      kube-controller-manager-centralctrl            1/1    Running    2 (4h6m ago)
4h8m
kube-system      kube-proxy-62k9k                              1/1    Running    0           4h7m
kube-system      kube-proxy-6nf8v                              1/1    Running    0           4h7m
kube-system      kube-proxy-vsvql                              1/1    Running    0           4h7m
kube-system      kube-scheduler-centralctrl                      1/1    Running    2 (4h6m ago)   4h8m
kube-system      nginx-proxy-centralworker1                      1/1    Running    0           4h7m
kube-system      nginx-proxy-centralworker2                      1/1    Running    0           4h7m
```

### Monitoring the K8s cluster status

To list the pods, their status and uptime, execute the command *kubectl get pods -A:*

```
root@centraljumphost :~# kubectl get pods -A
NAMESPACE         NAME                                       READY   STATUS    RESTARTS   AGE
contrail-deploy   contrail-k8s-deployer-77f978c8f5-5m9v5      1/1     Running   0          7m17s
contrail-system   contrail-k8s-apiserver-758588f78c-sgr8k     1/1     Running   0          6m11s
contrail-system   contrail-k8s-controller-c6465c47b-k74cv     1/1     Running   0          5m20s
contrail          contrail-control-0                         2/2     Running   0          5m20s
contrail          contrail-k8s-kubemanager-79899489f8-w2nkk  1/1     Running   0          5m20s
contrail          contrail-vrouter-masters-4vz69             3/3     Running   0          5m19s
contrail          contrail-vrouter-nodes-gkjt2               3/3     Running   0          5m19s
contrail          contrail-vrouter-nodes-pr7j6               3/3     Running   0          5m19s
kube-system       coredns-657959df74-5fh4v                   1/1     Running   0          3m4s
kube-system       coredns-657959df74-p4qpl                   1/1     Running   0          7d22h
kube-system       dns-autoscaler-b5c786945-kz5z7             1/1     Running   0          7d22h
kube-system       kube-apiserver-k8s-cp0                     1/1     Running   0          7d22h
kube-system       kube-controller-manager-k8s-cp0            1/1     Running   0          7d22h
kube-system       kube-proxy-7k9kl                           1/1     Running   0          7d22h
kube-system       kube-proxy-nd97f                           1/1     Running   0          7d22h
kube-system       kube-proxy-w4gm4                           1/1     Running   0          7d22h
kube-system       kube-scheduler-k8s-cp0                     1/1     Running   0          7d22h
kube-system       nginx-proxy-k8s-worker0                    1/1     Running   0          7d22h
kube-system       nginx-proxy-k8s-worker1                    1/1     Running   0          7d22
```

You can observe the K8s microservices running in kube-system namespace and Contrail microservices running in contrail, contrail-deploy, and contrail-system namespaces.

### Namespaces

To list all the namespaces, execute the command *kubectl get ns -A*

```
root@centraljumphost :~# kubectl get ns -A
NAME                                                  STATUS   AGE
contrail                                              Active   11m
contrail-analytics                                    Active   65m
contrail-deploy                                       Active   11m
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail   Active   9m12s
contrail-system                                       Active   11m
default                                               Active   7d22h
kube-node-lease                                       Active   7d22h
kube-public                                           Active   7d22h
kube-system                                           Active   7d22h
```

### Deployments

Deployment in Kubernetes allows you to describe an application's life cycle, such as the image to be used, and number of pods that should be there in it. It provides declarative updates of pods and replica sets.

To list all the deployments and their state, execute the command *kubectl get deployments -A*

```
root@centraljumphost:~# kubectl get deployments -A
NAMESPACE          NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
contrail-deploy    contrail-k8s-deployer       1/1     1            1           86m
contrail-system    contrail-k8s-apiserver      1/1     1            1           85m
contrail-system    contrail-k8s-controller     1/1     1            1           84m
contrail           contrail-k8s-kubemanager    1/1     1            1           84m
kube-system        coredns                     2/2     2            2           7d23h
kube-system        dns-autoscaler              1/1     1            1           7d23h
```

The output shows deployments of both K8s and Contrail. It also describes the desired state of each subsystem. We can also print a more detailed output of each deployment by executing command kubectl describe deployment <deployment name> -n <namespace>. As an example: kubectl describe deployment contrail-k8s-apiserver -n=contrail-system to see details about the deployment, such as number of replicas it is configured to run, etc.

### Service

A service in K8s is an abstraction which defines a logical set of pods and a policy to access them. Run command *kubectl get service -A:*

```
root@centraljumphost:~# kubectl get service -A
NAMESPACE       NAME         TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)              AGE
contrail-system   contrail-api   ClusterIP   10.233.37.213   <none>       19443/TCP            92m
default         kubernetes    ClusterIP   10.233.0.1     <none>       443/TCP              7d23h
kube-system     coredns       ClusterIP   10.233.0.3     <none>       53/UDP,53/TCP,9153/TCP
7d23h
```

The output describes three services of type Cluster-IP. Each service also defines IP and a port on which it is reachable. As no service is exposed to external users, external-ip is empty for all entries.

### DNS

K8s adds a DNS entry for each pod and service it creates. As pods get terminated and recreated, the DNS entry is updated with the IP address assigned.

To determine the cluster's DNS service clusterIP, execute command *kubectl get service -n=kube-system:*

```
root@centraljumphost:~# kubectl get service -n=kube-system
NAME     TYPE       CLUSTER-IP   EXTERNAL-IP   PORT(S)              AGE
coredns   ClusterIP   10.233.0.3   <none>       53/UDP,53/TCP,9153/TCP   8d
```

The output describes the type of service, the IP address, and the ports on which DNS is exposed. To debug DNS, you can download the dnsutils utility in the form of a pod from:

```
https://kubernetes.io/docs/tasks/administer-cluster/dns-debugging-resolution/ .

root@centraljumphost :~# kubectl exec -ti dnsutils -- cat /etc/resolv.conf
search default.svc.mycluster.contrail.lan svc.mycluster.contrail.lan mycluster.contrail.lan
nameserver 10.233.0.3
options ndots:5
```

The output here defines the search path and the nameserver set for the cluster. This utility can also be used for debugging url to IP resolution for an application service.

### CN2 status

Contrailstatus is a K8s binary written by Juniper to run on K8s system to show the contrail system status. Run command *kubectl contrailstatus* to get the details about contrail subsystem running on K8s.

```
root@centraljumphost :~# kubectl contrailstatus --all
PODNAME(CONFIG)                                 STATUS  NODE    IP              MESSAGE
contrail-k8s-apiserver-5dc9bccf4-dcsjc          ok      k8s-cp0 10.219.90.74
contrail-k8s-controller-c6465c47b-m4s64         ok      k8s-cp0 10.219.90.74
contrail-k8s-kubemanager-79899489f8-qr6r9       ok      k8s-cp0 10.219.90.74


PODNAME(CONTROL)        STATUS  NODE    IP              MESSAGE
contrail-control-0      ok      k8s-cp0 10.219.90.74


LOCAL BGPROUTER NEIGHBOR BGPROUTER      ENCODING        STATE           POD
k8s-cp0         k8s-cp0                 XMPP            Established ok  contrail-control-0
k8s-cp0         k8s-worker0             XMPP            Established ok  contrail-control-0
k8s-cp0         k8s-worker1             XMPP            Established ok  contrail-control-0


PODNAME(DATA)                   STATUS  NODE            IP              MESSAGE
contrail-vrouter-masters-6fr5h  ok      k8s-cp0         10.219.90.74
contrail-vrouter-nodes-4dr95    ok      k8s-worker1     10.219.90.76
contrail-vrouter-nodes-srrzc    ok      k8s-worker0     10.219.90.75
```

## Exploring Various CN2 Objects Using Kubectl

### Subnet

The Subnet is a block of IP addresses and the configurations associated with those addresses. A Subnet is based on a single address family (IPv4, IPv6) at a time. You must create separate IPv4 and IPv6 Subnets. Run *kubectl get subnet -A* to list all subnets.

```
root@centraljumphost:~# kubectl get subnet -A
NAMESPACE                                       NAME                            CIDR
USAGE   STATE    AGE
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail   default-podnetwork-pod-v4-subnet
```

```
10.233.64.0/18              0.04%   Success   2d23h
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail  default-podnetwork-pod-v6-subnet
fd85:ee78:d8a6:8607::1:0/112   0.01%   Success   2d23h
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail  default-servicenetwork-pod-v4-subnet
10.233.0.0/18               0.03%   Success   2d23h
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail  default-servicenetwork-pod-v6-subnet
fd85:ee78:d8a6:8607::1000/116  0.07%   Success   2d23h
```

This output shows two subnets defined for IPv4 and IPv6, one each for pods and services.

### VN

Virtual Networks are the basic building blocks for a CN2 subsystem. It forms a collection of VMIs, IPs, and MACs that can aid in communication between entities. VN is namespace scoped. Each namespace can have one or more VNs and each VN is isolated unless they are explicitly connected via a policy or VNR. Run command *kubectl get vn -A*:

```
root@centraljumphost:~# kubectl get vn -A
NAMESPACE                                        NAME                VNI   IP FAMILIES   STATE
AGE
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail  default-podnetwork      1    v6,v4
Success   2d23h
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail  default-servicenetwork  4    v6,v4
Success   2d23h
contrail                                         ip-fabric           2               Success
2d23h
contrail                                         link-local          3               Success
2d23h
```

The output shows two virtual networks built for the two K8s default networks, namely pod and service. Besides these, there are two more networks created by default in contrail namespace.

### VMI

VMI represents an interface (port) in a virtual network. VMI objects are namespace scoped. Contrail controller and Kubemanager reconcile and create VMIs for pods. Run *kubectl get vmi -A*:

```
root@centraljumphost:~# kubectl get vmi -A
NAMESPACE    CLUSTERNAME                              NAME              NETWORK           PODNAME
IFCNAME   STATE    AGE
contrail                                          k8s-cp0-vhost0      ip-fabric
Success   2d23h
contrail                                          k8s-worker0-vhost0   ip-fabric
Success   2d23h
contrail                                          k8s-worker1-vhost0   ip-fabric
Success   2d23h
kube-system   contrail-k8s-kubemanager-mycluster-contrail-lan  coredns-657959df74-5fh4v-3245903c
default-podnetwork   coredns-657959df74-5fh4v       eth0     Success   2d23h
kube-system   contrail-k8s-kubemanager-mycluster-contrail-lan  coredns-657959df74-p4qpl-447e3644
default-podnetwork   coredns-657959df74-p4qpl       eth0     Success   2d23h
kube-system   contrail-k8s-kubemanager-mycluster-contrail-lan  dns-autoscaler-b5c786945-kz5z7-
5f94d62c   default-podnetwork   dns-autoscaler-b5c786945-kz5z7   eth0     Success   2d23h
```

This output shows one VMI created for vhost0 on each node of the cluster. Also, one VMI is created for the DNS system running on each node.

NOTE Without a CNI deployed on the K8s cluster, the core DNS pods would be in Pending state, as they require a VMI.

### IIP

InstanceIP is an endpoint that automatically gets created when a service or a pod is created in a K8s system.

To list the InstanceIPs, execute the command *kubectl get iip -A*.

```
root@centraljumphost:~# kubectl get iip -A
NAME                                                                    IPADDRESS
NETWORK                                                              STATE    AGE
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail-api-49e3fdb5                10.233.37.213
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail/default-servicenetwork   Success   2d23h
contrail-k8s-kubemanager-mycluster-contrail-lan-coredns-657959df74-5fh4v-387c815f
fd85:ee78:d8a6:8607::1:2    contrail-k8s-kubemanager-mycluster-contrail-lan-contrail/default-
podnetwork      Success   2d23h
contrail-k8s-kubemanager-mycluster-contrail-lan-coredns-657959df74-5fh4v-3a7c8485       10.233.64.2
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail/default-podnetwork      Success   2d23h
contrail-k8s-kubemanager-mycluster-contrail-lan-coredns-657959df74-p4qpl-90263557
fd85:ee78:d8a6:8607::1:100   contrail-k8s-kubemanager-mycluster-contrail-lan-contrail/default-
podnetwork      Success   2d23h
contrail-k8s-kubemanager-mycluster-contrail-lan-coredns-657959df74-p4qpl-9226387d       10.233.65.0
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail/default-podnetwork      Success   2d23h
contrail-k8s-kubemanager-mycluster-contrail-lan-coredns-8560b829                     10.233.0.3
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail/default-servicenetwork   Success   2d23h
contrail-k8s-kubemanager-mycluster-contrail-lan-dns-autoscaler-b5c786945-kz5z7-891fc867
fd85:ee78:d8a6:8607::1:200   contrail-k8s-kubemanager-mycluster-contrail-lan-contrail/default-
podnetwork      Success   2d23h
contrail-k8s-kubemanager-mycluster-contrail-lan-dns-autoscaler-b5c786945-kz5z7-8b1fcb8d   10.233.66.0
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail/default-podnetwork      Success   2d23h
```

### Floating IP

FloatingIP is a non-namespaced object created alongside a service in a CN2-powered K8s system. To list floatingIPs, execute the command *kubectl get fip -A*.

```
root@centraljumphost:~# kubectl get fip -A
NAME                                                        IPADDRESS      INTERFACES   PORTMAPPING
STATE    AGE
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail-api-49e3fdb5   10.233.37.213   0
TCP/19443->19443                  Success   2d23h
contrail-k8s-kubemanager-mycluster-contrail-lan-coredns-8560b829      10.233.0.3     2
UDP/53->53,TCP/53->53,TCP/9153->9153   Success   2d23h
```

### VNR

Virtual Network Router is an object that connects one virtual network to another. The VNs can either be in the same Namespace or in different Namespaces. To list the VNRs, execute command kubectl get vnr -A.

```
root@centraljumphost:~# kubectl get vnr -A
NAMESPACE                                          NAME                          TYPE    STATE    AGE
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail  DefaultPodServiceIPFabricNetwork   spoke
Success   3d
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail  DefaultPodServiceNetwork         mesh
Success   3d
contrail-k8s-kubemanager-mycluster-contrail-lan-contrail  DefaultServiceNetwork            hub
Success   3d
contrail                                          DefaultIPFabricNetwork          hub    Success   3d
```

This output shows three VNRs created by default for contrail cluster and one for contrail namespace. Take a look at the types of VNRs created.

Mesh type VNR means all VNs connected in type mesh should be able to communicate with each other barring the labels match. In case of hub and spoke types, VNs connected as spoke can only communicate with hub VN and hub VN can communicate with all spoke VNs. No spoke VNs can communicate with each other, again barring labels match.

### Gateway router

A gateway router or bgprouter is a BGP speaker to which a CN2 cluster can be connected. The gateway router also builds dynamic tunnel with vRouter agent running on compute. Once deployed, a bgprouter can be inspected using the following commands.

```
root@centraljumphost:~# kubectl get bgprouter -A -o wide
contrailstatus.contrail.juniper.net/v1
NAMESPACE   NAME          TYPE         IDENTIFIER     STATE    AGE
contrail    SDN-GW        router       10.219.90.133   Success   34d  <<<<
contrail    centralmaster   control-node   10.219.90.136   Success   45d
```

Another command to get more details about a specific bgprouter is mentioned below.

```
kubectl describe -n contrail bgprouter SDN-GW.
```

NOTE Next in Chapter 4, when we deploy a sample application and expose it via a Loadbalancer service, a bgprouter will be configured and used to expose the external IP outside the cluster.

Chapter 4

# Deploy a 3-tier Application

This chapter starts with deploying a 3-tier application using a single kubectl apply command. The demo application will be running diagnostic containers in three environments, namely Prod, Staged, and Test. This resembles the environments a typical application development team uses.

We access the service IP, on which the application is exposed, to replicate how isolation is created for the three environments without the use of a network policy. Later, we roll back the deployment and then re-deploy the application step-by-step to better understand how it is built.

At this stage, you should be familiar with how the application is deployed and how isolation is achieved. Next, we implement Network Policy to add micro segmentation and to secure the application. In the end, we will summarize the deployment by implementing one last construct around how the external IP, on which the application is exposed, can be advertised to a gateway router.

## The 3-tier Application to Be Deployed

In this book, we will deploy an application called *thelitmustest*, which is a 3-tier application running diagnostic containers in generic tiers, namely frontend, middleware, and the backend.

The frontend is exposed on port 80 to the outside world. The frontend pods poll the middleware pods of the three environments in which the application is deployed, namely prod, staged, test and confirms the connectivity. The middleware pods listen on port 90 and poll their respective environment's backend pods to confirm the connectivity. The backend pods are exposed on port 80.

To conclude, once we deploy the application and access it using the exposed ExternalIP, we get a clear understanding of pods and their inter-connectivity.

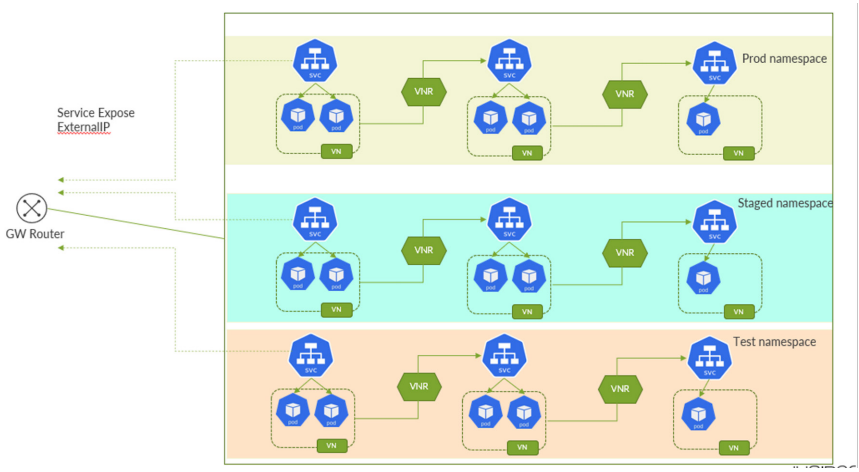



*Figure 4.1* *Application Deployment in Multi-tier Multi-environment*

## Deploying the 3-tier Application in Multiple Environments

Let us deploy the *litmustest* application in all three environments with their respective tiers along with all required components of the application.

If you are not already on AnsibleHost shell prompt, ssh to it or connect using virsh console <AnsibleHost>.

Execute the command *kubectl get nodes -A -o wide* to confirm the CN2 cluster health status.

```
root@centraljumphost:~# kubectl get nodes -A -o wide
NAME           STATUS   ROLES                  AGE   VERSION   INTERNAL-IP     EXTERNAL-IP   OS-IMAGE
KERNEL-VERSION     CONTAINER-RUNTIME
centralmaster    Ready    control-plane,master   21d   v1.20.7   10.219.90.136   <none>        Ubuntu
20.04.3 LTS   5.4.0-97-generic   cri-o://1.20.7
centralworker1   Ready    <none>                 21d   v1.20.7   10.219.90.137   <none>        Ubuntu
20.04.3 LTS   5.4.0-97-generic   cri-o://1.20.7
centralworker2   Ready    <none>                 21d   v1.20.7   10.219.90.138   <none>        Ubuntu
20.04.3 LTS   5.4.0-97-generic   cri-o://1.20.7
```

Now, deploy thelitmustest application in production environment using the following yaml manifest on git.

NOTE We need to clone the repo and navigate to thelitmustest application folder that contains the yamls necessary.

```
root@centraljumphost:~# git clone https://github.com/Juniper/cn2dayone.git
Cloning into 'cn2dayone'...
remote: Enumerating objects: 66, done.
remote: Total 66 (delta 0), reused 0 (delta 0), pack-reused 66
Unpacking objects: 100% (66/66), 21.05 KiB | 615.00 KiB/s, done.

root@centraljumphost:~# ls -la
total 12
drwxr-xr-x  3 root root 4096 Mar 14 10:25 .
drwx------ 35 root root 4096 Mar 14 10:24 ..
drwxr-xr-x  6 root root 4096 Mar 14 10:25 cn2dayone

root@centraljumphost:~# cd cn2dayone/
.git/                cn2_central_ansible/ cn2_ds1_ansible/     thelitmustest/

root@centraljumphost:~# cd cn2dayone/thelitmustest/

root@centraljumphost:~/cn2dayone/thelitmustest# ls -la
total 20
drwxr-xr-x 5 root root 4096 Mar 14 10:25 .
drwxr-xr-x 6 root root 4096 Mar 14 10:25 ..
drwxr-xr-x 2 root root 4096 Mar 14 10:25 prod_app1
drwxr-xr-x 2 root root 4096 Mar 14 10:25 staged_app1
drwxr-xr-x 2 root root 4096 Mar 14 10:25 test_app1
```

Apply all the yamls of this folder using the command *kubectl apply -f prod_aap1*.

```
root@centraljumphost:~/cn2dayone/thelitmustest# kubectl apply -f prod_app1
virtualnetwork.core.contrail.juniper.net/t2-ext-svc-vn created
subnet.core.contrail.juniper.net/t2-ext-svc-sn created
namespace/t2-prod-app1 created
subnet.core.contrail.juniper.net/t2-prod-app1-frontend-sn created
subnet.core.contrail.juniper.net/t2-prod-app1-middleware-sn created
subnet.core.contrail.juniper.net/t2-prod-app1-backend-sn created
virtualnetwork.core.contrail.juniper.net/t2-prod-app1-frontend-vn created
virtualnetwork.core.contrail.juniper.net/t2-prod-app1-backend-vn created
virtualnetwork.core.contrail.juniper.net/t2-prod-app1-middleware-vn created
virtualnetworkrouter.core.contrail.juniper.net/vnr-spoke created
virtualnetworkrouter.core.contrail.juniper.net/vnr-hub created
deployment.apps/t2-prod-app1-frontend-deployment created
service/t2-prod-app1-frontend-service created
service/t2-prod-app1-frontend-service-external created
deployment.apps/t2-prod-app1-middleware-deployment created
service/t2-prod-app1-middleware-service created
deployment.apps/t2-prod-app1-backend-deployment created
service/t2-prod-app1-backend-service created
```

Confirm the frontend deployment status

```
root@centraljumphost:~# kubectl rollout status -n t2-prod-app1 deployment/t2-prod-app1-frontend-
deployment
deployment "t2-prod-app1-frontend-deployment" successfully rolled out
```

Confirm the middleware deployment status

```
root@centraljumphost:~# kubectl rollout status -n t2-prod-app1 deployment/t2-prod-app1-middleware-
deployment
deployment "t2-prod-app1-middleware-deployment" successfully rolled out
```

Confirm the backend deployment status

```
root@centraljumphost:~# kubectl rollout status -n t2-prod-app1 deployment/t2-prod-app1-backend-
deployment
deployment "t2-prod-app1-backend-deployment" successfully rolled out
```

Or one can look at all the deployments per namespace using *kubectl get deployment -n <namespace>*

```
root@centraljumphost:~# kubectl get deployment -n=t2-prod-app1
NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
t2-prod-app1-backend-deployment     2/2     2            2           57m
t2-prod-app1-frontend-deployment    2/2     2            2           57m
t2-prod-app1-middleware-deployment  2/2     2            2           57m
```

Now, let us visualize the services the deployment has exposed. Run the following command to check it.

```
root@centraljumphost:~# kubectl get svc -n=t2-prod-app1
NAME                              TYPE         CLUSTER-IP     EXTERNAL-IP    PORT(S)      AGE
t2-prod-app1-backend-service          ClusterIP     10.233.46.97    <none>        80/TCP       6h22m
t2-prod-app1-frontend-service         ClusterIP     10.233.52.38    <none>        80/TCP       6h17m
t2-prod-app1-frontend-service-external  LoadBalancer  10.233.49.77    10.219.90.162  80:32355/TCP
6h17m
t2-prod-app1-middleware-service       ClusterIP     10.233.50.220   <none>        90/TCP       6h24m
```

The output above shows that the frontend service is externally exposed using a service type Loadbalancer. It has an ExternalIP assigned as 10.219.90.162.

Log in to the master node and access this application using the externalIP command: *lynx 10.219.90.162*.

```
Front End Pod Information

   Pod Name: t2-prod-app1-frontend-deployment-cd676f4cf-2zx2m

   Pod Namespace: t2-prod-app1

   Pod IP: 21.1.1.2

   Node Name: centralworker1

Middleware/Application pod information per environment

DEV ENV

{
  "pod_name": "unknown",
  "pod_namespace": "unknown",
  "pod_node_name": "unknown",
  "pod_ip": "unknown",
  "pod_backend_connection": false
}

   Connection to backend tier

   connection failure
```

```
TEST ENV

{
  “pod_name”: “unknown”,
  “pod_namespace”: “unknown”,
  “pod_node_name”: “unknown”,
  “pod_ip”: “unknown”,
  “pod_backend_connection”: false
}

   Connection to backend tier

   connection failure

PROD ENV

{
  “pod_name”: “t2-prod-app1-middleware-deployment-f9dff6cb6-sqx89”,
  “pod_namespace”: “t2-prod-app1”,
  “pod_nodename”: “centralworker1”,
  “pod_ip”: “21.1.2.1”,
  “pod_backend_connection”: true
}

   Connection to backend tier

   connection success
```

To conclude, the frontend pod with IP 21.1.1.2 got the request, the middleware pod with IP 21.1.2.1 received the poll from frontend, and the connection to backend from middleware is a success. Connection from prod frontend to staged and test environments is unknown as the application has not been deployed in them yet.

Now, let us deploy the application in staged environment by running the following yaml manifest.

```
root@centraljumphost:~/cn2dayone/thelitmustest# kubectl apply -f staged_app1
deployment.apps/t2-staged-app1-frontend-deployment created
service/t2-staged-app1-frontend-service created
service/t2-staged-app1-frontend-service-external created
deployment.apps/t2-staged-app1-middleware-deployment created
service/t2-staged-app1-middleware-service created
deployment.apps/t2-staged-app1-backend-deployment created
service/t2-staged-app1-backend-service created
virtualnetwork.core.contrail.juniper.net/t2-ext-svc-vn created
subnet.core.contrail.juniper.net/t2-ext-svc-sn created
namespace/t2-staged-app1 created
subnet.core.contrail.juniper.net/t2-staged-app1-frontend-sn created
subnet.core.contrail.juniper.net/t2-staged-app1-middleware-sn created
subnet.core.contrail.juniper.net/t2-staged-app1-backend-sn created
virtualnetwork.core.contrail.juniper.net/t2-staged-app1-frontend-vn created
virtualnetwork.core.contrail.juniper.net/t2-staged-app1-backend-vn created
virtualnetwork.core.contrail.juniper.net/t2-staged-app1-middleware-vn created
virtualnetworkrouter.core.contrail.juniper.net/vnr-spoke created
virtualnetworkrouter.core.contrail.juniper.net/vnr-hub created
```

Rollout of the deployment can be confirmed as shown below.

```
root@centraljumphost:~# kubectl get deployment -n=t2-staged-app1
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
t2-staged-app1-backend-deployment       2/2     2            2           52s
t2-staged-app1-frontend-deployment      2/2     2            2           52s
t2-staged-app1-middleware-deployment    2/2     2            2           52s
```

Similarly, let us check the service IP on which the application is exposed.

```
root@centraljumphost:~# kubectl get svc -n=t2-staged-app1
NAME                                TYPE          CLUSTER-IP     EXTERNAL-IP    PORT(S)       AGE
t2-staged-app1-backend-service           ClusterIP     10.233.9.84    <none>         80/TCP
2m18s
t2-staged-app1-frontend-service          ClusterIP     10.233.30.178   <none>         80/TCP
2m18s
t2-staged-app1-frontend-service-external   LoadBalancer   10.233.10.17    10.219.90.163   80:31848/TCP
2m18s
t2-staged-app1-middleware-service         ClusterIP     10.233.50.231   <none>         90/TCP
2m18s
```

Now, let us execute the Lynx command from master node to access the application using its ExternalIP, i.e. 10.219.90.163

Output:

```
Front End Pod Information

   Pod Name: t2-staged-app1-frontend-deployment-c445cc5ff-gwwb6

   Pod Namespace: t2-staged-app1

   Pod IP: 22.1.1.1

   Node Name: centralworker2

Middleware/Application pod information per environment

DEV ENV

{
  “pod_name”: “t2-staged-app1-middleware-deployment-679c895db8-k5pcl”,
  “pod_namespace”: “t2-staged-app1”,
  “pod_nodename”: “centralworker1”,
  “pod_ip”: “22.1.2.2”,
  “pod_backend_connection”: true
}

   Connection to backend tier

   connection success

TEST ENV

{
  “pod_name”: “unknown”,
  “pod_namespace”: “unknown”,
  “pod_node_name”: “unknown”,
```

```
  “pod_ip”: “unknown”,
  “pod_backend_connection”: false
}

   Connection to backend tier

   connection failure

PROD ENV

{
  “pod_name”: “Connection timeout”,
  “pod_backend_connection”: false
}

   Connection to backend tier

   connection failure
```

To summarize, using external-IP, we connected to frontend pod running in staged environment with IP 22.1.1.1. The middleware pod from the same environment with IP 22.1.2.2 polled the status of the backend pod and served the combined connection status to frontend.

If you notice, the result for test middleware remains unknown as the said environment isn't deployed yet, but the production environment shows timeout as the DNS got resolved, but the CN2 isolation kicked in and denied staged frontend pod from accessing production middleware pod.

Lastly, let us deploy the application in the test environment.

```
root@centraljumphost:~/cn2dayone/thelitmustest# kubectl apply -f test_app1
deployment.apps/t2-test-app1-frontend-deployment created
service/t2-test-app1-frontend-service created
service/t2-test-app1-frontend-service-external created
deployment.apps/t2-test-app1-middleware-deployment created
service/t2-test-app1-middleware-service created
deployment.apps/t2-test-app1-backend-deployment created
service/t2-test-app1-backend-service created
virtualnetwork.core.contrail.juniper.net/t2-ext-svc-vn unchanged
subnet.core.contrail.juniper.net/t2-ext-svc-sn configured
namespace/t2-test-app1 created
subnet.core.contrail.juniper.net/t2-test-app1-frontend-sn created
subnet.core.contrail.juniper.net/t2-test-app1-middleware-sn created
subnet.core.contrail.juniper.net/t2-test-app1-backend-sn created
virtualnetwork.core.contrail.juniper.net/t2-test-app1-frontend-vn created
virtualnetwork.core.contrail.juniper.net/t2-test-app1-backend-vn created
virtualnetwork.core.contrail.juniper.net/t2-test-app1-middleware-vn created
virtualnetworkrouter.core.contrail.juniper.net/vnr-spoke created
virtualnetworkrouter.core.contrail.juniper.net/vnr-hub created



root@centraljumphost:~# kubectl get deployment -n=t2-test-app1
```

```
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
t2-test-app1-backend-deployment       2/2     2            2           36s
t2-test-app1-frontend-deployment      2/2     2            2           36s
t2-test-app1-middleware-deployment    2/2     2            2           36s

root@centraljumphost:~# kubectl get svc -n=t2-test-app1
NAME                                TYPE         CLUSTER-IP     EXTERNAL-IP    PORT(S)       AGE
t2-test-app1-backend-service          ClusterIP     10.233.42.143   <none>        80/TCP        57s
t2-test-app1-frontend-service         ClusterIP     10.233.32.27    <none>        80/TCP        57s
t2-test-app1-frontend-service-external  LoadBalancer  10.233.39.118   10.219.90.164  80:32215/TCP
57s
t2-test-app1-middleware-service       ClusterIP     10.233.31.147   <none>        90/TCP        57s
```

To perform the test, first access the K8s control node using SSH. Now, run an access test using the browser application Lynx. Access the IP address 10.219.90.164, which is the IP address for frontend-service-external listening on port 80, by entering lynx 10.219.90.164 in the command line.

```
Front End Pod Information

   Pod Name: t2-test-app1-frontend-deployment-8c56867ff-qt8jt

   Pod Namespace: t2-test-app1

   Pod IP: 23.1.1.2

   Node Name: centralworker1

Middleware/Application pod information per environment

DEV ENV

{
  “pod_name”: “Connection timeout”,
  “pod_backend_connection”: false
}

   Connection to backend tier

   connection failure

TEST ENV

{
  “pod_name”: “t2-test-app1-middleware-deployment-64678d548-knn4s”,
  “pod_namespace”: “t2-test-app1”,
  “pod_nodename”: “centralworker2”,
  “pod_ip”: “23.1.2.1”,
  “pod_backend_connection”: true
}

   Connection to backend tier

   connection success
```

```
PROD ENV

{
  “pod_name”: “Connection timeout”,
  “pod_backend_connection”: false
}

   Connection to backend tier

   connection failure
```

To summarize the output, the frontend pod in the test environment with IP 23.1.1.2, responded to the request on the ExternalIP service. The request from the frontend pod to the test middleware was serviced by the pod with IP 23.1.2.1, and the request from the middleware to the test backend pod was successful. However, while the DNS query for both the production and staging environments was successful, access from the test middleware pod to either environment failed and resulted in a timeout.

## Visualizing Application Isolation Without Network Policy

If you have noticed the deployed manifests for each application deployment in any tier, you will see that no network policy manifests have been deployed yet. Despite this, traffic can still be segmented/isolated in each tier. This is made possible by using CN2. By default, Kubernetes uses a single CIDR for the pod network, which provides no network layer segmentation between pods in the cluster. However, CN2's latest release 22.4 introduces isolated namespaces, which creates a default pod and service network per namespace to ensure isolation for pods and services within a namespace. Additionally, CN2 allows for the creation of a custom pod network, which enables the creation of a pod with a different pod network per namespace or even per pod basis, by defining an annotation on either the pod or the namespace definition.

NOTE The above customization not only applies to the second interface of a pod, but also to the first.

### Rollback the deployment

Now that we have learned the outcome, let us take a step back and rollback the deployment. Let us deploy the application one step at a time and understand the constructs that achieve the isolation.

To rollback, apply the following command.

```
kubectl delete -f ~/cn2dayone/thelistmustest/test_app1

deployment.apps “t2-test-app1-backend-deployment” deleted
service “t2-test-app1-backend-service” deleted
virtualnetwork.core.contrail.juniper.net “t2-ext-svc-vn” deleted
subnet.core.contrail.juniper.net “t2-ext-svc-sn” deleted
```

```
deployment.apps “t2-test-app1-frontend-deployment” deleted
service “t2-test-app1-frontend-service” deleted
service “t2-test-app1-frontend-service-external” deleted
deployment.apps “t2-test-app1-middleware-deployment” deleted
service “t2-test-app1-middleware-service” deleted
namespace “t2-test-app1” deleted
subnet.core.contrail.juniper.net “t2-test-app1-frontend-sn” deleted
subnet.core.contrail.juniper.net “t2-test-app1-middleware-sn” deleted
subnet.core.contrail.juniper.net “t2-test-app1-backend-sn” deleted
virtualnetwork.core.contrail.juniper.net “t2-test-app1-frontend-vn” deleted
virtualnetwork.core.contrail.juniper.net “t2-test-app1-backend-vn” deleted
virtualnetwork.core.contrail.juniper.net “t2-test-app1-middleware-vn” deleted
virtualnetworkrouter.core.contrail.juniper.net “vnr-spoke” deleted
virtualnetworkrouter.core.contrail.juniper.net “vnr-hub” deleted
```

## Deep-Dive into the Deployment [Stepwise Approach]

Let’s deploy the application in a test environment from scratch in a step-by-step manner. This will help you understand the modifications in manifests that led to the segmentation of application tier and environment without the use of network policy.

Navigate to the test_app1 folder and you should be able to see the following yaml files:

```
root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# ls -la
total 36
drwxr-xr-x 2 root root 4096 Jan 13 03:55 .
drwxr-xr-x 5 root root 4096 Jan 11 16:52 ..
-rw-r--r-- 1 root root  855 Jan 13 03:53 back_dpl.yaml
-rw-r--r-- 1 root root  632 Jan 10 16:42 ext-net.yaml
-rw-r--r-- 1 root root 1924 Jan 13 03:55 front_dpl.yaml
-rw-r--r-- 1 root root 1687 Jan 13 03:54 mid_dpl.yaml
-rw-r--r-- 1 root root 2811 Jan 12 19:38 ns_vn_sn.yaml
-rw-r--r-- 1 root root 1560 Jan 13 03:53 svc_all.yaml
-rw-r--r-- 1 root root  980 Jan 12 19:37 vnr.yaml
```

The five files are:

- ns_vn_sn.yaml defines the namespace, subnet and virtual-network.
- vnr.yaml defines the virtualnetworkrouter.
- front_dpl.yaml, mid_dpl.yaml and back_dpl.yaml define the deployment of the three tiers of the application.
- svc.yaml defines the service for all the tiers.
- ext-net.yaml defines the external network which is mapped to a subnet. ExternalIP is assigned to the frontend tier service from this subnet.

The workflow is conceptualized and implemented in which we define a namespace for the environment, subnet and virtual network (VN) per tier.

NOTE A VN is a construct, which defines isolation by default, i.e., any pods within the same VN can talk to each other but not to pods outside the VN.

Following this, we define a VNR that connects the two VNs together.

Once the objects mentioned above are created, we can start deploying application pod to spawn and reside within VNs. Pods are assigned IPs from the defined subnet attached to the VN in which they are deployed. However, services are assigned an IP from the default service network for Kubernetes service discovery.

Open file ns_vn_sn.yaml and study the different yaml definitions for different constructs like namespace, subnet and virtual network. Observe the spec section under the virtual network. It contains a Boolean field "podnetwork" set to true.

```
apiVersion: core.contrail.juniper.net/v2
kind: VirtualNetwork
metadata:
  namespace: t2-test-app1
  name: t2-test-backend-vn
  annotations:
    core.juniper.net/display-name: t2-test-backend-vn
  labels:
    vn: spoke
spec:
  podNetwork: true
  v4SubnetReference:
    apiVersion: core.contrail.juniper.net/v2
    kind: Subnet
    namespace: t2-test-app1
    name: t2-test-backend-sn
```

NOTE This knob defines the virtual network type as a custom pod network. This will be examined in the next section.

Let us deploy the yaml file ns_vn_sn.yaml.

```
root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl apply -f ns_vn_sn.yaml
namespace/t2-test-app1 created
subnet.core.contrail.juniper.net/t2-test-app1-frontend-sn created
subnet.core.contrail.juniper.net/t2-test-app1-middleware-sn created
subnet.core.contrail.juniper.net/t2-test-app1-backend-sn created
virtualnetwork.core.contrail.juniper.net/t2-test-app1-frontend-vn created
virtualnetwork.core.contrail.juniper.net/t2-test-app1-backend-vn created
virtualnetwork.core.contrail.juniper.net/t2-test-app1-middleware-vn created
```

Here are the commands to check if the deployment is a success:

```
root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl get namespace | grep t2-test-app1
t2-test-app1                                 Active   44s

root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl get sn -n=t2-test-app1
NAME                    CIDR         USAGE   STATE     AGE
t2-test-app1-backend-sn     23.1.3.0/24   1.17%   Success   90s
t2-test-app1-frontend-sn    23.1.1.0/24   1.17%   Success   90s
```

```
t2-test-app1-middleware-sn   23.1.2.0/24   1.17%   Success   90s

root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl get vn -n=t2-test-app1
NAME                          VNI    IP FAMILIES    STATE      AGE
t2-test-app1-backend-vn       15     v4             Success    2m20s
t2-test-app1-frontend-vn      7      v4             Success    2m20s
t2-test-app1-middleware-vn    16     v4             Success    2m20s
```

Let us examine the pod yaml files for the three tiers. Open the frontend tier file and analyze the difference from a standard Kubernetes pod deployment manifest. Look for the annotations defined in the metadata section. These settings define the virtual network to be used and sets the podnetwork Boolean as true under the cni-args section.

Here's an extract from the frontend pod definition manifest:

```
    annotations:
      k8s.v1.cni.cncf.io/networks: |
        [
          {
            "name": "t2-test-app1-frontend-vn",
            "namespace": "t2-test-app1",
            "cni-args": {
            "net.juniper.contrail.podnetwork": true
             }
          }
        ]
```

This annotation sets the pod's first interface mapping to the user-defined virtual network and not the default pod network.

Apply the three yaml files for the 3-tier application pods.

```
root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl apply -f front_dpl.yaml
deployment.apps/t2-test-app1-frontend-deployment created
service/t2-test-app1-frontend-service created
service/t2-test-app1-frontend-service-external created

root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl apply -f mid_dpl.yaml
deployment.apps/t2-test-app1-middleware-deployment created
service/t2-test-app1-middleware-service created


root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl apply -f back_dpl.yaml
deployment.apps/t2-test-app1-backend-deployment created
service/t2-test-app1-backend-service created
```

Check the pods that are deployed for the 3-tier application.

```
root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl get pods -n=t2-test-app1 -o wide
NAME                                       READY   STATUS    RESTARTS   AGE   IP        NODE
NOMINATED NODE   READINESS GATES
t2-test-app1-backend-deployment-59ff5f8549-dl5tz    1/1    Running   0        62s   23.1.3.1
centralworker2   <none>         <none>
t2-test-app1-backend-deployment-59ff5f8549-kts2c    1/1    Running   0        62s   23.1.3.2
centralworker1   <none>         <none>
t2-test-app1-frontend-deployment-8c56867ff-cb5gj    1/1    Running   0        76s   23.1.1.1
centralworker2   <none>         <none>
```

```
t2-test-app1-frontend-deployment-8c56867ff-qvnlr    1/1    Running   0        76s   23.1.1.2
centralworker1   <none>         <none>
t2-test-app1-middleware-deployment-64678d548-28kf5   1/1    Running   0        69s   23.1.2.1
centralworker2   <none>         <none>
t2-test-app1-middleware-deployment-64678d548-pfqv5   1/1    Running   0        69s   23.1.2.2
centralworker1   <none>         <none>
```

Kubectl description for one of the pods would show that only one interface is attached to the container.

```
root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl describe pod t2-test-app1-
frontend-deployment-8c56867ff-cb5gj -n=t2-test-app1
Name:             t2-test-app1-frontend-deployment-8c56867ff-cb5gj
Namespace:        t2-test-app1
Priority:         0
Service Account:  default
Node:             centralworker2/10.219.90.138
Start Time:       Thu, 12 Jan 2023 19:47:32 +0000
Labels:           env=t2-test-app1
                  pod-template-hash=8c56867ff
                  svc=t2-test-app1-frontend-deployment
                  vn=t2-test-app1-frontend-vn
Annotations:      k8s.v1.cni.cncf.io/network-status:
                    [{
                        “name”: “t2-test-app1/t2-test-app1-frontend-vn”,
                        “interface”: “eth0”,
                        “ips”: [
                            “23.1.1.1”
                        ],
                        “mac”: “02:6c:71:53:0c:c0”,
                        “default”: true,
                        “dns”: {}
                    }]
                  k8s.v1.cni.cncf.io/networks:
                    [
                      {
                        “name”: “t2-test-app1-frontend-vn”,
                        “namespace”: “t2-test-app1”,
                        “cni-args”: {
                        “net.juniper.contrail.podnetwork”: true
                         }
                      }
                    ]
                  kube-manager.juniper.net/vm-uuid: d6920791-bf6e-4c71-a584-3b2ad0a44380
Status:           Running
IP:               23.1.1.1
```

So, we deployed the namespace, subnet, virtual network and pods. Service’s definitions in this lab are generic clusterIP type per tier, besides the frontend tier, which has a definition of a second Loadbalancer service. This will get an ExternalIP to expose the application outside.

This service has a special annotation that defines the knob externalnetwork and points it to the network definition.

Open the file front_dpl.yaml to verify the annotation as below:

```
service.contrail.juniper.net/externalNetwork: default/t2-ext-svc-vn
```

Let us deploy the service manifest file svc_all.yaml.

Run the following commands to deploy and check the status of services:

```
root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl apply -f svc_all.yaml
service/t2-test-app1-frontend-service created
service/t2-test-app1-frontend-service-external created
service/t2-test-app1-middleware-service created
service/t2-test-app1-backend-service created

root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl get svc -n=t2-test-app1
NAME                                TYPE         CLUSTER-IP     EXTERNAL-IP   PORT(S)       AGE
t2-test-app1-backend-service           ClusterIP     10.233.28.170   <none>       80/TCP        9m38s
t2-test-app1-frontend-service          ClusterIP     10.233.25.244   <none>       80/TCP        9m52s
t2-test-app1-frontend-service-external   LoadBalancer   10.233.51.151   <pending>    80:31116/TCP
9m52s
t2-test-app1-middleware-service        ClusterIP     10.233.27.63    <none>       90/TCP        9m45s
```

The output shows ClusterIP per tier and an extra Loadbalancer service for the frontend pod. But doesn't the ExternalIP status show <pending>?

This is because we defined the virtual networks for the pods but not for the external network. Let us deploy the file named ext-net.yaml.

```
root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl apply -f ext-net.yaml
virtualnetwork.core.contrail.juniper.net/t2-ext-svc-vn created
subnet.core.contrail.juniper.net/t2-ext-svc-sn created

root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl get svc -n=t2-test-app1
t2-test-app1       t2-test-app1-backend-service            ClusterIP     10.233.28.170   <none>
80/TCP                  16m
t2-test-app1       t2-test-app1-frontend-service           ClusterIP     10.233.25.244   <none>
80/TCP                  16m
t2-test-app1       t2-test-app1-frontend-service-external    LoadBalancer   10.233.53.31
10.219.90.164      80:31791/TCP               76s
t2-test-app1       t2-test-app1-middleware-service         ClusterIP     10.233.27.63    <none>
90/TCP                  16m
```

Now, access the service using the ClusterIIP/ExternalIP of the Loadbalancer service.

```
root@centralmaster:~# lynx 10.233.53.31
Output:-
Front End Pod Information

  Pod Name: t2-test-app1-frontend-deployment-8c56867ff-cb5gj

  Pod Namespace: t2-test-app1

  Pod IP: 23.1.1.1

  Node Name: centralworker2

Middleware/Application pod information per environment
```

```
DEV ENV

{
  “pod_name”: “Connection timeout”,
  “pod_backend_connection”: false
}

   Connection to backend tier

   connection failure

TEST ENV

{
  “pod_name”: “Connection timeout”,
  “pod_backend_connection”: false
}

   Connection to backend tier

   connection failure

PROD ENV

{
  “pod_name”: “Connection timeout”,
  “pod_backend_connection”: false
}

   Connection to backend tier

   connection failure
```

Let us examine the output above. We accessed the ClusterIP service. The frontend pod of the test environment with the IP address 23.1.1.1 replied to this request. However, the middleware access is timed out for all three environments. Prod and stage timeout is expected as they reside in other namespaces, but timeout to the test environment middleware is not expected.

The reason for the timeout is that the said pods are mapped to custom pod networks for each tier. This feature of CN2 provides an additional layer of isolation other than namespaces.

To enable communication selectively, we will use another feature of CN2 called VNRs.

The file vnr.yaml defines VNRs. These will bind the VNs together. As explained earlier in the book, VNRs can be configured either as hub-and-spoke or mesh. This can be configured by using the type field in the spec section of the manifest.

```
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: t2-test-app1
```

```
  name: vnr-spoke
  annotations:
    core.juniper.net/display-name: vnr-spoke
  labels:
    vnr: spoke
spec:
  type: spoke
  virtualNetworkSelector:
    matchLabels:
      vn: spoke
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: hub
        namespaceSelector:
          matchLabels:
            ns: t2-test-app1
---
apiVersion: core.contrail.juniper.net/v1alpha1
kind: VirtualNetworkRouter
metadata:
  namespace: t2-test-app1
  name: vnr-hub
  annotations:
    core.juniper.net/display-name: vnr-hub
  labels:
    vnr: hub
spec:
  type: hub
  virtualNetworkSelector:
    matchLabels:
      vn: hub
  import:
    virtualNetworkRouters:
      - virtualNetworkRouterSelector:
          matchLabels:
            vnr: spoke
        namespaceSelector:
          matchLabels:
            ns: t2-test-app1
```

We will use the hub-and-spoke type to enable communication between middleware to frontend and backend. We are not using mesh type considering that the communication from frontend to backend tier is not required.

The procedure will require us to create two VNRs — one would be vnr-hub and the other one vnr-spoke. Vnr-hub is mapped to middleware tier and vnr-spoke to frontend and backend tier using the labels.

Let us deploy the final yaml manifest for this lab.

```
root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl apply -f vnr.yaml
virtualnetworkrouter.core.contrail.juniper.net/vnr-spoke created
virtualnetworkrouter.core.contrail.juniper.net/vnr-hub created
```

```
root@centraljumphost:~/cn2dayone/thelitmustest/test_app1# kubectl get vnr -n=t2-test-app1
NAME                                                        TYPE    STATE     AGE
CustomPodNetDefaultSvcNetwork-t2-test-app1-backend-vn       spoke   Success   8h
CustomPodNetDefaultSvcNetwork-t2-test-app1-frontend-vn      spoke   Success   8h
CustomPodNetDefaultSvcNetwork-t2-test-app1-middleware-vn    spoke   Success   8h
CustomPodNetIPFabricNetwork-t2-test-app1-backend-vn         spoke   Success   8h
CustomPodNetIPFabricNetwork-t2-test-app1-frontend-vn        spoke   Success   8h
CustomPodNetIPFabricNetwork-t2-test-app1-middleware-vn      spoke   Success   8h
vnr-hub                                                     hub     Success   27s
vnr-spoke                                                   spoke   Success   27s
```

Now that the VNR has been deployed, let us re-check the application using the ClusterIP /ExternalIP from the master nodes.

```
root@centralmaster:~# lynx 10.233.53.31
Output:-
Front End Pod Information

   Pod Name: t2-test-app1-frontend-deployment-8c56867ff-cb5gj

   Pod Namespace: t2-test-app1

   Pod IP: 23.1.1.1

   Node Name: centralworker2

Middleware/Application pod information per environment

DEV ENV

{
  “pod_name”: “Connection timeout”,
  “pod_backend_connection”: false
}

   Connection to backend tier

   connection failure

TEST ENV

{
  “pod_name”: “t2-test-app1-middleware-deployment-64678d548-pfqv5”,
  “pod_namespace”: “t2-test-app1”,
  “pod_nodename”: “centralworker1”,
  “pod_ip”: “23.1.2.2”,
  “pod_backend_connection”: true
}

   Connection to backend tier

   connection success

PROD ENV

{
  “pod_name”: “Connection timeout”,
  “pod_backend_connection”: false
```

```
}

   Connection to backend tier

   connection failure
```

You can see that the frontend pod's poll towards middleware is a success and the pod with ID 23.1.2.2 accepted the request. The middleware pod's poll to backend is also a success. Hence, our VNR deployment is accomplished.

So, in this section we deployed the namespace, subnet and virtual network for the 3-tier application. After verification of these resources, we deployed the pod and service on each tier. Finally, we connected the VNs to each other by deploying VNRs, which eventually allowed communication between the middleware and other tiers. We achieved isolation between the tiers without defining a network policy.

Let us now understand micro-segmentation by using network policy.

## Network Policy for Micro-segmentation

A Kubernetes network policy defines access permissions for groups of pods by defining ingress and egress rules. CN2 implements this using Contrail Firewall Constructs.

The primary question is whether we need to deploy a network policy when we already have virtual networks (VNs) that isolate pods. This depends on how we want to structure our application architecture. By deploying a NetPol, we can add an extra layer of isolation for pod communication, effectively enabling micro-segmentation. NetPol provides security between pods belonging to a virtual network.

To deploy network policy on the 3-tier application, apply the following yaml file.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: t2-prod-app1-net-pol
  namespace: t2-prod-app1
spec:
  egress:
  - ports:
    - port: 53
      protocol: UDP
    - port: 53
      protocol: TCP
  - ports:
    - port: 90
      protocol: TCP
  ingress:
  - ports:
    - port: 443
      protocol: TCP
    - port: 80
```

```
      protocol: TCP
  podSelector:
    matchLabels:
      svc: t2-prod-app1-frontend-deployment
  policyTypes:
  - Ingress
  - Egress

---

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: t2-prod-app1-middleware-net-pol
  namespace: t2-prod-app1
spec:
  ingress:
  - from:
    - podSelector:
        matchLabels:
          svc: t2-prod-app1-frontend-deployment
    ports:
    - port: 90
      protocol: TCP
  egress:
  - ports:
    - port: 80
      protocol: TCP
    - port: 53
      protocol: UDP
    - port: 53
      protocol: TCP
  podSelector:
    matchLabels:
      svc: t2-prod-app1-middleware-deployment
  policyTypes:
  - Ingress
  - Egress
---

---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: t2-prod-app1-backend-net-pol
  namespace: t2-prod-app1
spec:
  ingress:
  - from:
    - podSelector:
        matchLabels:
          svc: t2-prod-app1-middleware-deployment
    ports:
    - port: 80
      protocol: TCP
  podSelector:
    matchLabels:
```

```
    svc: t2-prod-app1-backend-deployment
  policyTypes:
  - Ingress

root@centraljumphost:~/vnr_book# kubectl apply -f Net_Pol.yaml
networkpolicy.networking.k8s.io/t2-test-app1-net-pol created
networkpolicy.networking.k8s.io/t2-test-app1-middleware-net-pol created
networkpolicy.networking.k8s.io/t2-prod-app1-backend-net-pol created

root@centraljumphost:~/vnr_book# kubectl get netpol -n=t2-test-app1
NAME                          POD-SELECTOR                          AGE
t2-test-app1-backend-net-pol     svc=t2-test-app1-backend-deployment     36h
t2-test-app1-middleware-net-pol   svc=t2-test-app1-middleware-deployment   36h
t2-test-app1-net-pol            svc=t2-test-app1-frontend-deployment    36h
```

## Exposing Applications Externally Using BGP Peering with an SDNGW

Exposing applications to the outside world is another important aspect to pull user traffic to the application. This task is tied to the cluster's SDN controller in which the application is running.

CN2 natively supports peering with the gateway router using BGP protocol. The CN2 controller can peer with a BGP router and form dynamic GRE tunnels towards each compute node to route and load balance traffic towards services or pods.

Here, in this example, we peer the CN2 cluster with an MX80.

NOTE CN2 can peer with any vendor or software BGP speaker, acting as a gateway router for the cluster environment.

To configure CN2 to start peering with the BGP router, deploy the following yaml file:

```
apiVersion: core.contrail.juniper.net/v1alpha1
kind: BGPRouter
metadata:
  namespace: contrail
  name: sdn-gw
spec:
  parent:
    apiVersion: core.contrail.juniper.net/v1alpha1
    kind: RoutingInstance
    namespace: contrail
    name: default
  bgpRouterParameters:
    vendor: juniper
    routerType: router
    address: 10.219.90.133
    identifier: 10.219.90.133
    autonomousSystem: 64513
    addressFamilies:
      family:
        - inet
  bgpRouterReferences:
```

```
    - apiVersion: core.contrail.juniper.net/v1alpha1
      kind: bgpouter
      namespace: contrail
      name: controller.central.cluster

kubectl apply -f bgprouter.yaml

root@centraljumphost:~# kubectl get bgprouter -A
NAMESPACE   NAME           TYPE          IDENTIFIER      STATE     AGE
contrail    SDN-GW         router        10.219.90.133   Success   1d
contrail    centralmaster  control-node  10.219.90.136   Success   1d
```

On the gateway side, with the following configuration in place, we should see BGP neighborship established and dynamic GRE tunnels created towards the compute nodes:

```
set protocols bgp group clustercentral type internal
set protocols bgp group clustercentral local-address 10.219.90.133
set protocols bgp group clustercentral keep all
set protocols bgp group clustercentral family inet-vpn unicast
set protocols bgp group clustercentral neighbor 10.219.90.136
set routing-options dynamic-tunnels dynamic_overlay_tunnels source-address 10.219.90.133
set routing-options dynamic-tunnels dynamic_overlay_tunnels gre
set routing-options dynamic-tunnels dynamic_overlay_tunnels destination-networks 10.219.90.0/24
set interfaces lt-0/0/0 unit 0 encapsulation frame-relay
set interfaces lt-0/0/0 unit 0 dlci 1
set interfaces lt-0/0/0 unit 0 peer-unit 1
set interfaces lt-0/0/0 unit 0 family inet
set interfaces lt-0/0/0 unit 1 encapsulation frame-relay
set interfaces lt-0/0/0 unit 1 dlci 1
set interfaces lt-0/0/0 unit 1 peer-unit 0
set interfaces lt-0/0/0 unit 1 family inet
set interfaces irb unit 100 family inet address 10.219.90.133/26
set interfaces lo0 unit 0 family inet
set interfaces lo0 unit 1 family inet address 192.0.2.1/24
set routing-options route-distinguisher-id 10.219.90.133
set routing-instances Intranet instance-type vrf
set routing-instances Intranet interface lt-0/0/0.1
set routing-instances Intranet interface lo0.1
set routing-instances Intranet vrf-target target:64512:10000
set routing-instances Intranet routing-options static route 0.0.0.0/0 next-hop lt-0/0/0.1

root@jtac-mx80-r2026> show bgp summary
Threading mode: BGP I/O
Groups: 1 Peers: 1 Down peers: 0
Table          Tot Paths  Act Paths Suppressed    History Damp State    Pending
bgp.l3vpn.0
                      95         95          0          0          0          0
inet.0
                       0          0          0          0          0          0
Peer                     AS      InPkt     OutPkt    OutQ   Flaps Last Up/Dwn State|#Active/
Received/Accepted/Damped...
10.219.90.136         64512       8391       7162       0       4  2d 5:43:22 Establ
  bgp.l3vpn.0: 95/95/95/0

root@jtac-mx80-r2026> show dynamic-tunnels database
*- Signal Tunnels #- PFE-down
Table: inet.3
```

```
Destination-network: 10.0.0.0/24

Destination-network: 10.219.90.0/24
Tunnel to: 10.219.90.136/32 State: Up
  Reference count: 1
  Next-hop type: gre
    Source address: 10.219.90.133
    Next hop: gr-0/0/10.32770
      State: Up
Tunnel to: 10.219.90.137/32 State: Up
  Reference count: 1
  Next-hop type: gre
    Source address: 10.219.90.133
    Next hop: gr-0/0/10.32774
      State: Up
Tunnel to: 10.219.90.138/32 State: Up
  Reference count: 1
  Next-hop type: gre
    Source address: 10.219.90.133
    Next hop: gr-0/0/10.32773
      State: Up

root@jtac-mx80-r2026> show route 10.219.90.162/32

Intranet.inet.0: 3 destinations, 3 routes (3 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.219.90.162/32      *[BGP/170] 00:00:32, MED 100, localpref 200, from 10.219.90.136
                       AS path: ?, validation-state: unverified
                     >  via gr-0/0/10.32774, Push 140

bgp.l3vpn.0: 76 destinations, 76 routes (76 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

10.219.90.137:8:10.219.90.162/32
                    *[BGP/170] 00:00:32, MED 100, localpref 200, from 10.219.90.136
                       AS path: ?, validation-state: unverified
                     >  via gr-0/0/10.32774, Push 140
```

From the output, we can observe that route 10.219.90.162/32 is advertised by CN2 controller. The output also indicates that the route is installed in Intranet.inet.0 table.

The mentioned destination in CN2 cluster is reachable from the gateway using the gr-0/0/10.32774 tunnel using push label as 140.

This route can be propagated by the gateway router to upstream networks. This enables communication from external entities towards the service using IP 10.219.90.162.

This concludes the chapter of deploying the application and enabling communication from external entities.

# Appendix

## Add a Distributed Cluster to the Setup

The central cluster contains all the contrail components required to control the Kubernetes constructs and the CN2 objects. One can add multiple distributed clusters running their own K8s control plane and CN2 data plane, i.e., the vRouter and vRouter agent in each of their nodes. Distributed clusters are attached to the central cluster via a listener in the form of a Kubemanager. This reconciles any CRUD operation on the distributed cluster.

With the following steps, one should be able to build the distributed cluster and attach it to the central cluster with only five ansible playbook executions.

### Knowing the five deployment playbooks

The cloned cn2dayone git repo contains folders cn2_deploy_central and cn2_deploy_ds1. Navigate to the cn2_deploy_ds1 directory and analyze it. Listing the folder will show you the following files:

```
root@CN2demo1:~/cn2dayone/cn2_ds1_ansible# ls -la
total 64
drwxr-xr-x 2 root root 4096 Jan 21 18:05 .
drwxr-xr-x 6 root root 4096 Jan 14 13:03 ..
-rw-r--r-- 1 root root 4143 Jan 19 15:36 1Play_VM_Creation.yaml
-rw-r--r-- 1 root root 1378 Jan 19 15:36 2Play_VM_Disk_Resize.yaml
-rw-r--r-- 1 root root 3976 Jan 19 15:36 3Play_KubeSprayHost.yaml
-rw-r--r-- 1 root root 1950 Jan 19 15:36 4Play_CNI_less_Cluster.yaml
-rw-r--r-- 1 root root 7597 Jan 19 15:36 5Play_CN2_Cluster.yaml
-rwxr-xr-x 1 root root  146 Jan 19 15:36 destroy_cluster.sh
-rw-r--r-- 1 root root 1124 Jan 19 15:36 inventory.yaml
-rw-r--r-- 1 root root 1317 Jan 19 16:07 k8s_inventory.yaml
-rw-r--r-- 1 root root  657 Jan 19 15:36 kubemanager_ds1.yaml
-rwxr-xr-x 1 root root  537 Jan 19 15:36 network_yaml_create.sh
-rw-r--r-- 1 root root 1872 Jan 14 13:03 README.md
```

The yaml files starting with numbers 1 to 5 are the ones used to deploy the distributed cluster. The file inventory.yaml defines the inventory used for cluster deployment. It contains the VM names and corresponding IP addresses for this cluster. The K8s_inventory.yaml defines the K8s cluster inventory file which would be pushed to the jumphost while executing Playbook 3. This will eventually be used to build the CN2 K8s cluster. The file kubemanager_ds1.yaml defines parameters used to deploy the kubemanager for distributed cluster on the central cluster. This file will be copied to the central cluster during the execution of Playbook 5. Lastly, shell script network_yaml_create.sh is a file that is used during the Playbook 1 execution to generate the four network yaml files to be pushed to individual qcow images. These qcow images are disks mapped to the VMs.

## VM Creation

The first playbook builds the network file, customizes the qcow2 images, spawns and initializes the VMs to be used in the deployment.

```
root@CN2demo1:~/cn2dayone/cn2_ds1_ansible# ansible-playbook -i inventory.yaml 1Play_VM_Creation.yaml

PLAY [Playbook to prepare vms image, network, customize and lastly spawn the vms]
************************

TASK [Gathering Facts] *********************************************************************
ok: [localhost]


PLAY RECAP *********************************************************************
localhost              : ok=9    changed=5    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

Verification task #1: Enter command *virsh list –all | grep ds1* to confirm four VMs are in the running state.

```
root@CN2demo1:~/cn2dayone/cn2_ds1_ansible# virsh list --all | grep ds1
 665   ds1jumphost       running
 666   ds1ctrl           running
 667   ds1worker1        running
 668   ds1worker2        running
```

Verification task #2: You must be able to ssh to the VMs without entering password.

```
root@CN2demo1:~/cn2dayone/cn2_ds1_ansible# ssh ds1jumphost
Warning: Permanently added 'ds1jumphost' (ECDSA) to the list of known hosts.
Warning: the ECDSA host key for 'ds1jumphost' differs from the key for the IP address '10.219.90.88'
Offending key for IP in /root/.ssh/known_hosts:12
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.4.0-97-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Sat Jan 21 18:18:35 UTC 2023
```

```
  System load:  0.07            Processes:             183
  Usage of /:   3.0% of 50.42GB  Users logged in:       0
  Memory usage: 0%              IPv4 address for enp1s0: 10.219.90.88
  Swap usage:   0%


192 updates can be applied immediately.
141 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

New release '22.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


Last login: Sat Jan 21 18:18:21 2023 from 10.219.90.79
root@ds1jumphost:~#
```

### Resizing VM disks

The execution of this playbook expands the disk to the required size for the deployment and installs NTP on the VMs.

```
root@CN2demo1:~/cn2dayone/cn2_ds1_ansible# ansible-playbook -i inventory.yaml 2Play_VM_Disk_Resize.
yaml

PLAY [Update /etc/hosts among all Vms and resize the VMs Disk] ************************************
***

TASK [Wait for system to become reachable] ********************************************************
****
ok: [ds1jumphost]
ok: [ds1ctrl]
ok: [ds1worker2]
ok: [ds1worker1]

TASK [gather Facts] *******************************************************************************
****
ok: [ds1worker2]
ok: [ds1worker1]
ok: [ds1jumphost]
ok: [ds1ctrl]

--SNIP--

PLAY RECAP ****************************************************************
ds1ctrl                : ok=10   changed=7    unreachable=0    failed=0    skipped=0    rescued=0
ignored=1
ds1jumphost             : ok=10   changed=7    unreachable=0    failed=0    skipped=0    rescued=0
ignored=1
ds1worker1              : ok=10   changed=7    unreachable=0    failed=0    skipped=0    rescued=0
ignored=1
ds1worker2              : ok=10   changed=7    unreachable=0    failed=0    skipped=0    rescued=0
ignored=1
```

Verification task #1: Log in to a VM and execute df -H to confirm if the disk has been resized to 50G+

```
root@ds1jumphost:~# df -H
Filesystem      Size  Used Avail Use% Mounted on
udev             17G     0   17G   0% /dev
tmpfs           3.4G  1.2M  3.4G   1% /run
/dev/vda1        55G  1.7G   53G   3% /
tmpfs            17G     0   17G   0% /dev/shm
tmpfs           5.3M     0  5.3M   0% /run/lock
tmpfs            17G     0   17G   0% /sys/fs/cgroup
/dev/vda15      110M  5.5M  104M   5% /boot/efi
/dev/loop2       46M   46M     0 100% /snap/snapd/14549
/dev/loop0       71M   71M     0 100% /snap/lxd/21835
/dev/loop1       66M   66M     0 100% /snap/core20/1328
tmpfs           3.4G     0  3.4G   0% /run/user/0
```

### Packages' installation on ds1jumphost

The execution of this playbook will perform a makeover of ds1jumphost. At the completion of this playbook, the VM should be installed with all the packages required to deploy Playbook 4 and 5.

```
root@CN2demo1:~/cn2dayone/cn2_ds1_ansible# ansible-playbook -i inventory.yaml 3Play_KubeSprayHost.
yaml

PLAY [Install required packages on Jumpshost] ***********************************************************
****

TASK [Gathering Facts] ***********************************************************************************
****
ok: [ds1jumphost]

--SNIP--

PLAY RECAP ***********************************************************************************************
ds1ctrl                : ok=4    changed=3    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
ds1jumphost             : ok=22   changed=17   unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
ds1worker1              : ok=4    changed=3    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
ds1worker2              : ok=4    changed=3    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

### Install distributed K8s cluster

The execution of this playbook makes the jumphost a kubespray node. The jumphost, in turn, installs the K8s cluster on the remaining nodes sans CNI. This may require patience as it involves a "Deploy cluster" step that may take time to complete successfully.

```
root@CN2demo1:~/cn2dayone/cn2_ds1_ansible# ansible-playbook -i inventory.yaml 4Play_CNI_less_Cluster.
yaml
```

```
PLAY [Install k8s cluster without CNI] ****************************************************************
***

TASK [Gathering Facts] ********************************************************************
ok: [ds1jumphost]

--SNIP--

PLAY RECAP ********************************************************************
ds1jumphost              : ok=9    changed=8    unreachable=0    failed=0    skipped=0    rescued=0
ignored=0
```

Verification task #1: Check that cluster has been deployed without a CNI and the DNS is in pending state. This is an expected outcome.

```
root@ds1jumphost:~# kubectl get pods -A
NAMESPACE     NAME                                 READY   STATUS    RESTARTS      AGE
kube-system   coredns-74d6c5659f-fdv5m             0/1     Pending   0             65s
kube-system   dns-autoscaler-59b8867c86-cxc6q      0/1     Pending   0             61s
kube-system   kube-apiserver-ds1ctrl               1/1     Running   1             2m49s
kube-system   kube-controller-manager-ds1ctrl      1/1     Running   2 (25s ago)   2m49s
kube-system   kube-proxy-62vbp                     1/1     Running   0             98s
kube-system   kube-proxy-cdgxq                     1/1     Running   0             98s
kube-system   kube-proxy-mktkq                     1/1     Running   0             98s
kube-system   kube-scheduler-ds1ctrl               1/1     Running   2 (23s ago)   2m49s
kube-system   nginx-proxy-ds1worker1               1/1     Running   0             98s
kube-system   nginx-proxy-ds1worker2               1/1     Running   0             96s
```

Verification task #2: Observe the output of command *kubectl get svc -A* to confirm that the cluster-ip for kubernetes is unique and not in conflict with the central cluster.

```
root@ds1jumphost:~# kubectl get service -A | egrep -i “namespace|Kubernetes”
NAMESPACE    NAME         TYPE       CLUSTER-IP     EXTERNAL IP   PORT(S)     AGE
default    kubernetes     ClusterIP   10.234.0.1      <none>        443/TCP    1d

root@centraljumphost:~# kubectl get svc -A | egrep -i “namespace|Kubernetes”
NAMESPACE    NAME         TYPE       CLUSTER-IP     EXTERNAL IP   PORT(S)     AGE
default    Kubernetes   ClusterIP      10.233.0.1      <none>     443/TCP     1d
```

### Deploy CN2 in distributed cluster

The execution of this play will perform all the necessary steps to install CN2 components and plug the distributed cluster to central cluster.

```
root@CN2demo1:~/cn2dayone/cn2_ds1_ansible# ansible-playbook -i inventory.yaml 5Play_CN2_Cluster.yaml
[WARNING]: While constructing a mapping from
/root/cn2_deploy_ansible/cn2_ds1_ansible/5Play_CN2_Cluster.yaml, line 197, column 7, found a
duplicate
dict key (register). Using last defined value only.
Enter Enterprise_Hub.juniper.net Username?: rahulverma@juniper.net
Enter Password?:

PLAY [Install required packages on Jumpshost] *********************************************************
****

TASK [Gathering Facts] ********************************************************************************
```

```
*****
ok: [ds1jumphost]


--SNIP--

TASK [Pause for 1 minutes to build app cache] ************************************************************
***
Pausing for 60 seconds
(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)
ok: [ds1jumphost]

TASK [Reset DNS resolution] ******************************************************************
changed: [ds1jumphost -> 10.219.90.89] => (item=ds1ctrl)
changed: [ds1jumphost -> 10.219.90.90] => (item=ds1worker1)
changed: [ds1jumphost -> 10.219.90.91] => (item=ds1worker2)

PLAY RECAP ***********************************************************************
centralctrl             : ok=2    changed=0    unreachable=0    failed=0    skipped=3    rescued=0
ignored=0
centraljumphost          : ok=10   changed=5    unreachable=0    failed=0    skipped=2    rescued=0
ignored=0
centralworker1           : ok=2    changed=0    unreachable=0    failed=0    skipped=3    rescued=0
ignored=0
centralworker2           : ok=2    changed=0    unreachable=0    failed=0    skipped=3    rescued=0
ignored=0
ds1ctrl                 : ok=2    changed=0    unreachable=0    failed=0    skipped=3    rescued=0
ignored=0
ds1jumphost              : ok=28   changed=13   unreachable=0    failed=0    skipped=1    rescued=0
ignored=0
ds1worker1              : ok=2    changed=0    unreachable=0    failed=0    skipped=3    rescued=0
ignored=0
ds1worker2              : ok=2    changed=0    unreachable=0    failed=0    skipped=3    rescued=0
ignored=0
```

Verification task #1: Access the ds1jumphost and confirm pod status.

```
root@ds1jumphost:~# kubectl get pods -A
NAMESPACE         NAME                                       READY   STATUS      RESTARTS      AGE
cert-manager      cert-manager-745fb764f4-49mqc              1/1     Running     0             8m53s
cert-manager      cert-manager-cainjector-5654f68b7b-bkn7t   1/1     Running     0             8m53s
cert-manager      cert-manager-webhook-fff46dd94-5s4kc       1/1     Running     0             8m52s
contrail-deploy   contrail-k8s-deployer-5cb67969d8-pddz2     1/1     Running     0             10m
contrail-system   contrail-k8s-cert-gen-job-create-95xnh     0/1     Completed   0             9m10s
contrail          contrail-vrouter-masters-qlmtl             3/3     Running     0             6m58s
contrail          contrail-vrouter-nodes-6n299               3/3     Running     0             6m58s
contrail          contrail-vrouter-nodes-d8dg5               3/3     Running     0             6m58s
kube-system       coredns-74d6c5659f-fdv5m                   1/1     Running     0             18m
kube-system       coredns-74d6c5659f-grjgq                   1/1     Running     0             4m10s
kube-system       dns-autoscaler-59b8867c86-cxc6q            1/1     Running     0             18m
kube-system       kube-apiserver-ds1ctrl                     1/1     Running     1             20m
kube-system       kube-controller-manager-ds1ctrl            1/1     Running     2 (17m ago)   20m
kube-system       kube-proxy-62vbp                           1/1     Running     0             19m
kube-system       kube-proxy-cdgxq                           1/1     Running     0             19m
kube-system       kube-proxy-mktkq                           1/1     Running     0             19m
kube-system       kube-scheduler-ds1ctrl                     1/1     Running     2 (17m ago)   20m
kube-system       nginx-proxy-ds1worker1                     1/1     Running     0             19m
kube-system       nginx-proxy-ds1worker2                     1/1     Running     0             19m
```

The output suggests that the distributed cluster components are in running state

```
root@centraljumphost:~# kubectl get pods -A
NAMESPACE          NAME                                                READY   STATUS     RESTARTS   AGE
cert-manager       cert-manager-7d6d9465db-5zv8r                       1/1     Running    0          95d
cert-manager       cert-manager-cainjector-6f567667c6-5ch8d            1/1     Running    0
95d
cert-manager       cert-manager-webhook-f947d7c68-dz54j                1/1     Running    0
95d
contrail-deploy    contrail-k8s-deployer-6447484fc7-4gpxb              1/1     Running    0
95d
contrail-system    contrail-k8s-apiserver-59db79d795-nrq6b             1/1     Running    0
95d
contrail-system    contrail-k8s-cert-gen-job-create-cf5zx              0/1     Completed  0
95d
contrail-system    contrail-k8s-controller-7dbdbdd799-lz49d            1/1     Running    0
95d
contrail           contrail-control-0                                  2/2     Running    0          95d
contrail           contrail-k8s-contrailstatusmonitor-7d746f44f5-xldwz 1/1     Running    0
95d
contrail           contrail-k8s-kubemanager-659fc566d6-rt49c           1/1     Running    0
95d
contrail           contrail-vrouter-masters-zfr7b                      3/3     Running    0          95d
contrail           contrail-vrouter-nodes-qjcdw                        3/3     Running    0          95d
contrail           contrail-vrouter-nodes-rltgq                        3/3     Running    0          95d
contrail           kubemanager-ds1cluster-6d6996d784-4fd7w             1/1     Running    0
95d
kube-system        coredns-657959df74-mqlng                            1/1     Running    0          95d
kube-system        coredns-657959df74-qq4pk                            1/1     Running    0          95d
kube-system        dns-autoscaler-b5c786945-s69qh                      1/1     Running    0          95d
kube-system        kube-apiserver-centralmaster                        1/1     Running    0          95d
kube-system        kube-controller-manager-centralmaster               1/1     Running    0
95d
kube-system        kube-proxy-bpqfs                                    1/1     Running    0          95d
kube-system        kube-proxy-h229g                                    1/1     Running    0          95d
kube-system        kube-proxy-nqlmb                                    1/1     Running    0          95d
kube-system        kube-scheduler-centralmaster                        1/1     Running    0          95d
kube-system        nginx-proxy-centralworker1                          1/1     Running    0          95d
kube-system        nginx-proxy-centralworker2                          1/1     Running    0          95d
```

From the above output, we can confirm that the central cluster is deployed with a pod called kubemanager-ds1cluster-6d6996d784 which acts as the kubemanger for distributed cluster.

This concludes the deployment of distributed cluster and connecting its CN2 data plane with CN2 control plane residing in central cluster.

## What to Do Next and Where to Go

As you conclude this book, we hope you found the above chapters helpful in understanding the concepts and technologies explained. However, this is just the beginning of your journey towards mastering these topics. As technology continues to evolve rapidly, we encourage you to stay up to date with the latest developments and best practices.

To assist you in your learning journey, we have shared a git link that will be updated periodically with additional resources such as troubleshooting guides, case studies, and more. We hope you will continue to explore and learn more about these exciting technologies.

Git link: https://github.com/Juniper/cn2dayone