JUNIPER NETWORKS | Engineering Simplicity

Day One Million

DAY ONE: BUILDING CONTAINERS WITH KUBERNETES AND CONTRAIL

Learn Kubernetes fundamentals and understand its integration with Juniper Contrail®

By Ping Song, Ayman Aborabh, and Yuvaraja Mariappan

DAY ONE: BUILDING CONTAINERS WITH KUBERNETES AND CONTRAIL

This *Day One* book details the long list of Juniper Contrail features that can enrich Kubernetes implementations. Starting with the basic concepts of containers and moving through virtual networks and Contrail architecture, the authors review the basic foundation and key components of Kubernetes, including several different Kubernetes features without Contrail integration. But the core of the book is devoted to detailed labs and use cases of Contrail and Kubernetes together. Contrail can build and manage virtual networks that integrate containers, VMs, and bare metal servers of all types, so the authors focus on how to integrate a popular pair: Kubernetes and Contrail.

“SDN and Kubernetes are two of the hottest technologies and this Day One book covers the concepts of Kubernetes as well as the native networking model inside the Kubernetes, and then demonstrates how Contrail enhances the capability of network functions as well as security in Kubernetes. Recommended.” - Lin zhang, CTO, CStack Technologies

“A must read for anyone exploring how to integrate Contrail’s virtual networking into Kubernetes containerized platform. You will find answers for common questions and practices of Kubernetes from Contrail’s perspective, including, but not limited to, container/kubernetes basics, packet flow, and much more.” – Kevin Yang, Staff Engineer, SDDCaaS, VMware

“Great book for those who want to understand how Contrail can be integrated into Kubernetes as a container network provider. Topics range from basic concepts to advance features and implementation details with lots of examples.” - Yan Chen, Network Engineer, Google

IT’S DAY ONE AND YOU HAVE A JOB TO DO, SO LEARN ABOUT:

- Container technology and different Kubernetes features using YAML.
- Kubernetes integration with Contrail.
- Kubernetes network policy and Contrail firewall security.
- Configuring isolated Kubernetes Namespaces using Contrail.
- Configuring Floating IP in Contrail to provide container’s outside connectivity.
- Configuring load balancer and cluster IP services in Kubernetes using Contrail.
- Configuring different types of Kubernetes ingress using Contrail.
- Configuring and building multi-interfaces/multi-network containers using Contrail.
- Deploying Contrail service chaining using the Juniper cSRX in Kubernetes.

ISBN 978-1941441916
52500
9 781941 441916

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

JUNIPER NETWORKS

Day One: Building Containers Using Kubernetes and Contrail

by Ping Song, Ayman Aborabh, and Yuvaraja Mariappan

Chapter 1: Foundation Principles 8

Chapter 2: Kubernetes Basics 15

Chapter 3: Kubernetes in Practice 31

Chapter 4: Kubernetes and Contrail Integration 92

Chapter 5: Contrail Services 116

Chapter 6: Contrail Ingress 153

Chapter 7: Packet Flow in Contrail: End-to-End View 195

Chapter 8: Contrail Firewall Policy 212

Chapter 9: Contrail Multiple Interface Pod 246

Chapter 10: Contrail Service Chaining with cSRX 255

Appendix 278

© 2019 by Juniper Networks, Inc. All rights reserved. Juniper Networks and Junos are registered trademarks of Juniper Networks, Inc. in the United States and other countries. The Juniper Networks Logo and the Junos logo, are trademarks of Juniper Networks, Inc. All other trademarks, service marks, registered trademarks, or registered service marks are the property of their respective owners. Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

Published by Juniper Networks Books

Author: Ping Song, Ayman Aborabh,Yuvaraja Mariappan

Editor in Chief: Patrick Ames

Copyeditor: Nancy Koerbel

Technical Reviewers: Yuvaraja Mariappan, Vincent Zhang

ISBN: 978-1-941441-96-1

Version History: v1, November. 2019

2 3 4 5 6 7 8 9 10

http://www.juniper.net/dayone

About the Authors

Ping Song is a technical support engineer at Juniper Networks. Ping has been working for 20 years in the networking industry. He is certified double CCIE#26084 (R&S, SP) and triple JNCIE (SP#2178, ENT#775, DC#239). Ping is an enthusiastic Linux and VIM user and he enjoys being a power user of Linux tool chains like asciidoc/tmux/git/shell/jupyter and others, as well as being a tclexpect/python script lover. He currently supports customers building and maintaining their data centers with Juniper contrail networking and SD-WAN solutions

Ayman Aborabh has over 15 years of experience in the networking industry, and is a certified CCIE (R&S, SP) #24287 and JNCIE (SP # 1261, ENT # 294, DC #29).In his current role as a Senior Proctor/Trainer at Juniper Networks, he builds, customizes, or directly delivers training sessions for almost every major Enterprise, Service Provider, and mobile operator in Europe. Only IP networks can keep Ayman away from his hobby, writing in political philosophy.

Yuvaraja Mariappan is a software engineer in the Contrail group at Juniper Networks. He has over 16 years of experience in software development and has worked within various Layers in the system from kernel to application. He is one of the core contributors for the Contrail Kubernetes solution.

Authors' Acknowledgments

We'd all like to thank Patrick Ames for his encouragement and support during the time of writing this book. And thank you to Nancy Koerbel for the expert editing and proofing.

Ping Song: Writing this book was not an easy process, especially considering the day-to-day work load coming from Juniper's amazing Contrail customers, and the fact that this book is entirely a 'volunteer in your spare time job' thing and the 'spare time' is very limited! So I'd like to thank Venkatraman Venkatapahty, Venkatesh Velpula, and Qasim Arham for sharing some of their YAML files in the beginning of our testing. Thanks Yuvaraja for the deep level explanations of our implementations. And thank you to my manager, Siew Ng, who has been supportive and encouraging on this entire project. I'd like also thank my wife Sandy for her understanding and support of my work, including this project. Thanks to my lovely kids XiXi and Jeremy for the joy (and sometimes extra work...) you guys bring to me. Special thanks to XiXi for enriching all my diagrams with her coloring skills.

Ayman Aborabh: I would love to thank my partners in this book, Ping Song for his dedication and never saying enough to extensions and adjustment, and Yuvaraja Mariappan for his deep inside-out knowledge that smoothed our way. Thanks to Lisa Watts for her support and encouragement from the start until the end. Giving support makes you one step ahead, but continuous support makes you fly. After my first book I promised to give Nada Affara back the time I spent on writing, and here I am finishing another book and relying on her support, now in double debt to her. I would love to thank Ahlam, Omnia, and Ahmed for the motivation they they gave to me. This book is dedicated to the soul of my father – as you once read novels to me before your siesta, I wish I could read this book to you now.

Yuvaraja Mariappan: I thought writing a book would not be easy. Ping Song and Ayman Aborabh have made it very easy. I really thank Ping and Ayman for the wonderful experience. I would also like to thank all the Contrail folks that got me here. I also want to thank my manager Sachchidanand Vaidya for supporting me on this entire project. Finally, I thank my parents, grandmother, sister, and my sister's family and friends for everything.

Feedback? Comments? Error reports? Email them to dayone@juniper.net.

Welcome to Day One

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

- Download a free PDF edition at http://www.juniper.net/dayone
- PDF books are available on the Juniper app: Junos Genius
- Purchase the paper edition at Vervante Corporation (www.vervante.com) for between $15-$40, depending on page length.

Key Juniper Contrail Resources

The Juniper TechLibrary has been supporting Contrail with its excellent documentation for years. The Contrail selection is thorough, and it's kept up-to-date with the latest technologies and GUI changes. This book is no substitution for that body of information. The authors assume that you have some familiarity with Juniper Contrail documentation: https://www.juniper.net/documentation/product/en_US/contrail-networking/5.0.

The authors keep a GitHub website at https://github.com/pinggit/kubernetes-contrail-day-one , where you can find the book's content, all the YAML file source code used for the examples, figures, etc. Add comments, suggestions or questions regarding the book, too."

What You Need to Know Before Reading This Book

This book details the long list of Juniper Contrail features that can enrich Kubernetes implementations. It starts with the basics and builds from there to cover more complex setups. It's structured as follows:

- *Chapter 1:* Provides a basic understanding of containers, virtual networks, and Contrail architecture.
- *Chapter 2:* Lays down the basic foundation and key components of Kubernetes.
- *Chapter 3:* Explains different Kubernetes features using labs and without any Contrail integration.
- *Chapters 4 through 10:* These chapters are the core of the book. They begin by explaining Contrail integration with Kubernetes, then continue on to cover a number of detailed labs and use cases using Contrail/Kubernetes.

Contrail Command and This Book

Contrail Command(CC) is the new user interface (UI) starting with Contrail 5.0.1. Throughout this book we use both the new CC and the old UI to demonstrate the lab studies. The publication date for this book is November 2019, so depending on when you are reading it, keep in mind that CC will soon be the only UI; the legacy one is slated to be discontinued at some time.

Detailed information about CC is available from the Juniper documentation website, so we don't elaborate on it here. To access CC use this URL in your web browser: *https://Contrail-Command-Server-IP-Address:9091*. The CC server can be the same as, or different from, the Kubernetes master server or the Contrail Controller node. In this book, we've installed them in same server.

The functions and settings in CC are grouped in a main menu. This makes a great entry point where you can navigate through different Contrail functions. To get the CC main menu, click on the group name right next to the Contrail Command logo on the upper left corner of the UI as shown in Figure P.1.

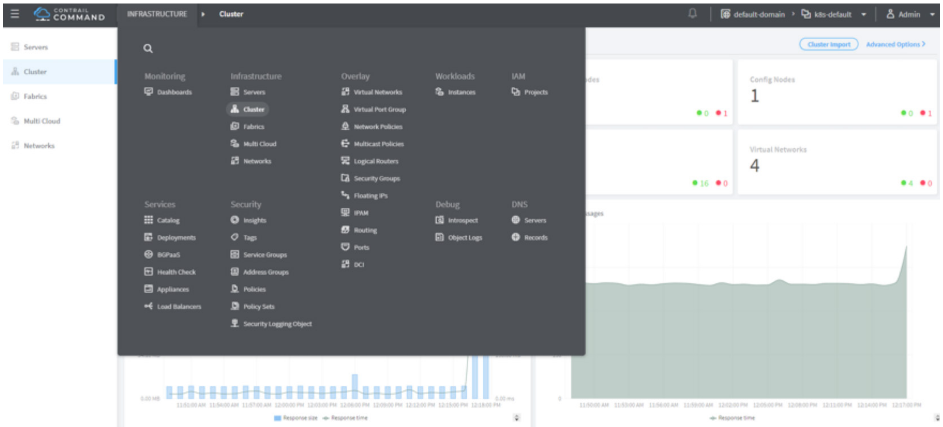


Figure P.1 Contrail Command Main Menu

While Figure P.1 shows a group in Infrastructure, it could be any group. Just click it and you will get the main menu, then from there you can select and jump into all kinds of different settings.

Remember, our focus is not on CC but on giving you some basic insights into CC, which will be helpful to you as you build containers using Kubernetes.

Chapter 1

Foundation Principles

Several years ago, virtualization was the most fashionable keyword in IT because it revolutionized the way servers were built. Virtualization was about the adoption of virtual machines (VMs) instead of dedicated, physical servers for hosting and building new applications. When it came to scaling, portability, capacity management, cost, and more, VMs were a clear winner (as they are today). You can find tons of comparisons between the two approaches.

If virtualization was the keyword then, the keywords now are cloud, SDN, and containers.

Today, the heavily discussed comparisons are between VMs and *containers,* and how containers promise a new way to build and scale applications. While many small organizations are thinking of containers as something too wild, or too early, to adopt, the simple fact is that from Gmail to YouTube to Search, everything at Google runs in containers, and they run two billion containers a week. This might give you a clue as to where the industry is heading.

But what is a container and how is it comparable to a VM? Let's start this *Day One* book with a comparison.

Containers Overview

From a technical perspective, the concept of a container is rooted in the Namespaces and Cgroups concept in Linux, but the term is also inspired by the actual metal cargo shipping containers that you see on seafaring ships. Both kinds of containers share the ability to isolate contents, maintain carrier independence, offer portability, and much more.

Containers are a logical packaging mechanism. YYou can think of containers as a lightweight virtualization that runs an application and its dependencies in the same operating system, but in different contexts that remove the need to replicate an entire OS as shown in Figure 1.1. By doing this the application is confined in a lightweight package that can be developed and tested individually, then implemented and scaled much faster than the traditional VM. Developers just need to build and configure this lightweight piece of software so that most of the application is containerized and publicly available without the need to manage and support the application per OS.

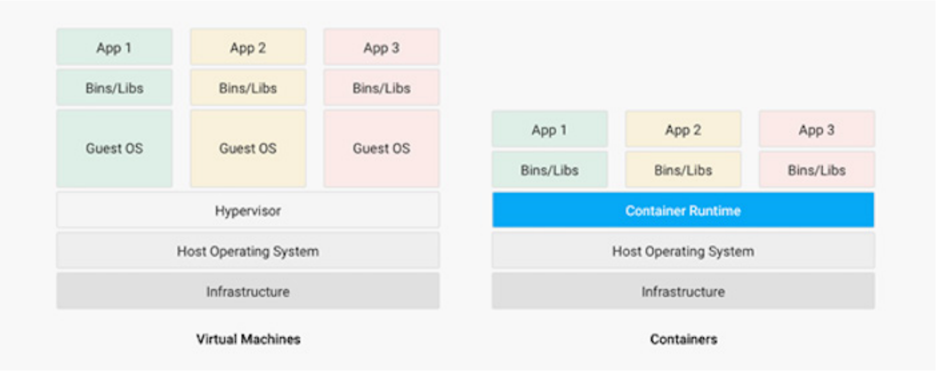


Figure 1.1 *VM versus Containers*

Many developers would call the container runtime shown in Figure 1.1 as the *Hypervisor of Containers*. Although this term is not technically correct, it may be useful in visualizing the hierarchy.

As in many VM technologies, the most common hypervisors are KVM and VMware ESX/ESXi. In container technologies, Docker and Rkt are the most common, with Docker being the most widely deployed. Let's review some useful numbers in comparing VMs with containers.

Juniper vSRX versus cSRX

Currently most common applications such as Redis, Ngnix, Mongo, MySQL, WordPress, Jenkins , Kibana, and Perl have been containerized and are offered publicly at https://hub.docker.com allowing developers to quickly build and test their applications.

There are lots of available tests that compare performance and scaling for any given application while running in containers versus VM. The comparisons all dwell on the benefits of running your application in containers, but what about network function virtualizations (NFV) such as firewall, NAT, routing, and more?

When it comes to VM-based NFV, most network vendors already implement a virtualized flavor of the hardware equipment that could be run on the hypervisor of a standard x86 hardware. Built on Junos, vSRX is a Juniper Networks SRX Series Services Gateway in a virtualized form factor that delivers networking and security features similar to those available for the physical SRX just as it does for the containerized based NFV. It's the new trend. Juniper cSRX is the industry's first containerized firewall offering a compact footprint with a high-density firewall for virtualized and cloud environments. Table 1.1 lists a comparison between vSRX and cSRX in which you can see the idea of the cSRX being a lightweight NFV.

Table 1.1 *vSRX versus cSRX*

	vSRX	cSRX
Use Cases	Integrated routing, security, NAT, VPN, High Performance	L4-L7 Security, Low Footprint
Memory Requirement	4GB Minimum	In MBs
NAT	Yes	Yes
IPSec VPN	Yes	No
Boot-up Time	~minutes	<1second
Image size	In GBs	In MBs

NOTE Using micro services techniques, the application can be split into smaller services with each part (a container in this case) doing a specific job.

Understanding Docker

As discussed, containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship them all out as one package. Docker is software that facilitates creating, deploying, and running containers.

The starting point is the source code for the Docker image file, and from there you can build the image to be stored and distributed to any registry – most commonly a Docker hub – and use this image to run the containers.

Docker uses the client-server architecture shown in Figure 1.2. The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker daemon does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon communicate using a REST API over UNIX sockets or a network interface.

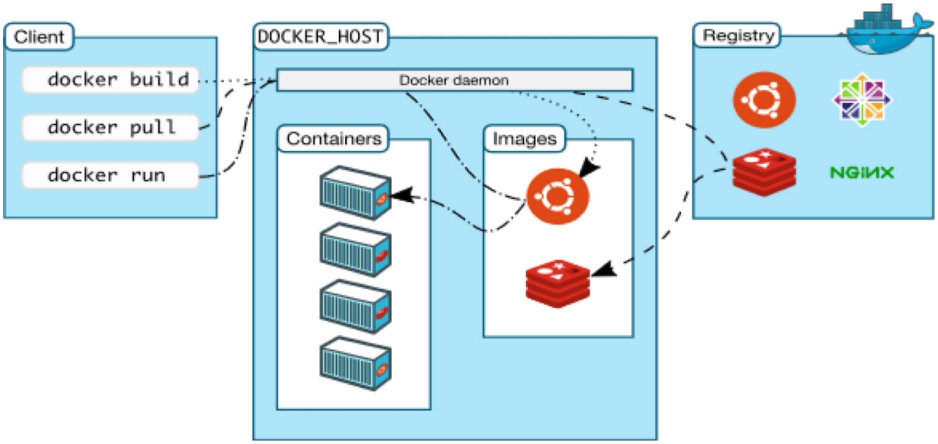


Figure 1.2 Docker Architecture

Containers don't exist in a vacuum, and in production environments you won't have just one host with multiple containers, but rather multiple hosts running hundreds, if not thousands, of containers, which raises two important questions:

- How do these containers communicate with each other on the same host or in different hosts, as well as with the outside world? (Basically, the networking parts of containers.)
- Who determines which containers get launched on which host? Based on what? Upgrade? Number of containers per application? Basically, who orchestrates that?

These two questions are answered, in detail, throughout the rest of the book, but if you want a quick answer, just think Juniper Contrail and Kubernetes!

Let's start with the basic foundations of the Juniper Contrail Platform.

Contrail Platform Overview

The Juniper Contrail Platform provides dynamic end-to-end networking, networking policy, and control for any cloud, any workload, and any deployment, all from a single user interface. Although the focus of this book is building a secure container network orchestrated by Kubernetes, Contrail can build virtual networks that integrate containers, VMs, and bare metal servers.

Virtual networks are a key concept in the Contrail system. Virtual networks are logical constructs implemented on top of physical networks. They are used to replace VLAN-based isolation and provide multi-tenancy in a virtualized data center. Each tenant or an application can have one or more virtual networks. Each virtual network is isolated from all the other virtual networks unless explicitly allowed by network policy. Virtual networks can be extended to physical networks using a gateway. Finally, virtual networks are used to build service-chaining.

As shown in Figure 1.3, the network operator only deals with the logical abstraction of the network, then Contrail does the heavy lifting which includes, but is not limited to, building polices, exchanging routes, and building tunnels on the physical topology.

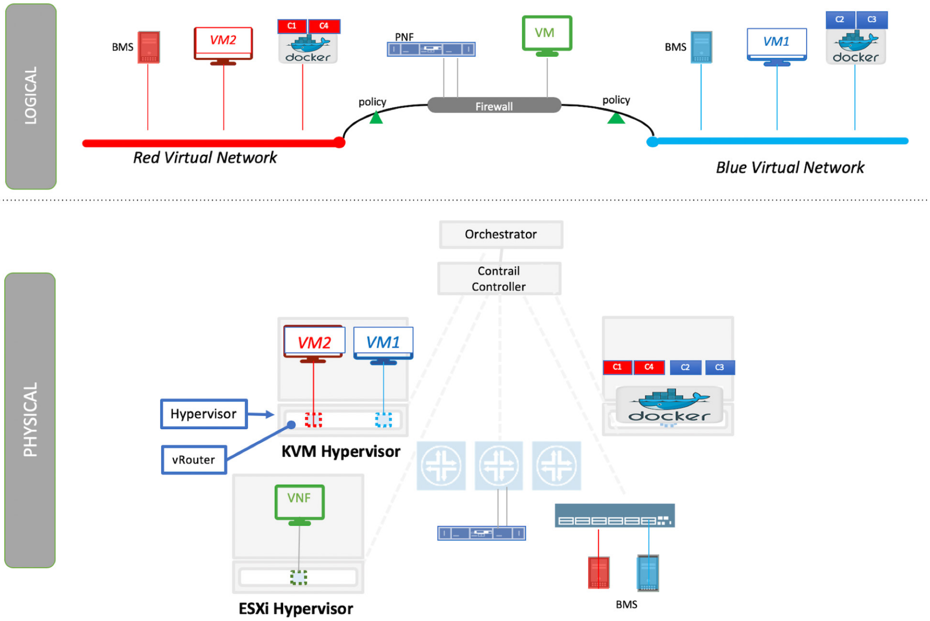


Figure 1.3 Virtual Networks

Contrail Architecture Fundamentals

Contrail runs in a logically centralized, physically-distributed model with its two main components, Contrail controller and Contrail vRouter. The controller is the control and management plane that manages and configures the vRouter, collecting and presenting analytics. The Contrail vRouter is the forwarding plane that provides Layer 2 and Layer 3 services, and distributed firewall capabilities, while implementing policies between virtual networks.

Contrail integrates with many orchestrators such as OpenStack, VMware, Kubernetes, OpenShift, and Mesos. It uses multiple protocols to provide SDN to these orchestrators, as shown in Figure 1.4 where Extensible Messaging and Presence Protocol (XMPP) is an open XML technology for real-time communication, defined in RFC 6120. In Contrail, XMPP offers two main functionalities: distributing routing information and pushing configurations, which are similar to what IBGP does in MPLS VPNs models, plus NETCONF in device management.

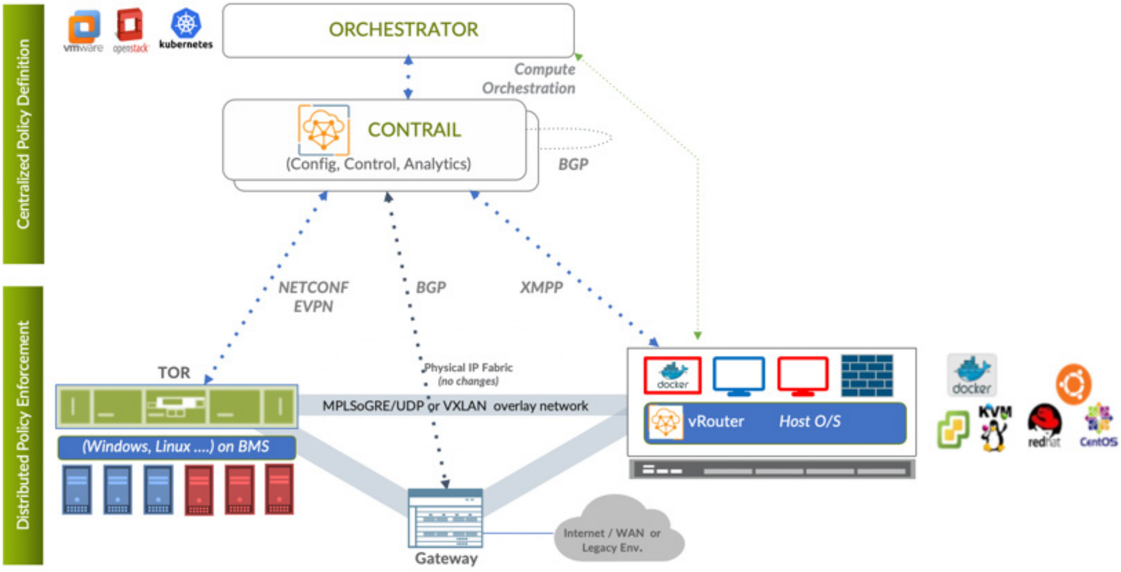

Centralized Policy Definition
Distributed Policy Enforcement
vmware
openstack
kubernetes
ORCHESTRATOR
Compute Orchestration
CONTRAIL
(Config, Control, Analytics)
BGP
NETCONF EVPN
BGP
XMPP
TOR
Physical IP Fabric (no changes)
MPLSoGRE/UDP or VXLAN overlay network
(Windows, Linux ....) on BMS
docker
vRouter
Host O/S
docker
KVM
redhat
CentOS
Gateway
Internet / WAN or Legacy Env.

Figure 1.4 Contrail Architecture

Figure 1.4 also illustrates:

- BGP is used to exchange routes with physical routers and Contrail device manager can use NETCONF to configure this Gateway.
- Ethernet VPN (EVPN) is a standards-based technology, RFC 7432, that provides virtual multipoint bridged connectivity between different Layer 2 domains over an IP network. Contrail Controller exchanges EVPN routes with TOR switches (acting as a Layer 2 VXLAN gateway) to offer faster recovery with active-active VXLAN forwarding.
- MPLSoGRE, MPLSoUDP, or VXLAN, are three different kind of overlay tunnels to carry traffic over IP networks. They are all IP packets, but in VXLAN you use the VNI values in the VXLAN header for segmentation, whereas in MPLSoGRE and MPLSoUDP you use the MPLS label value for segmentation.

To simplify the relationship between Contrail vRouter, Contrail Controller, and the IP fabric from an architectural prospective, compare it to the MPLS VPN model whereas any service provider's vrouter is like a PE router and the VM/container is like CE, but the vRouter is just a tool of the Contrail Controller, and when it comes to bare metal servers, the top of rack would be the PE.

NOTE This *Day One* book uses the words *compute node* and *host* interchangeably. Both mean the entity hosts the containers that need a compute node to host it. This host could be a physical server in your DC, or a VM in either your data center or the public cloud.

Contrail vRouter

Contrail vRouter is composed of the Contrail components on the compute node/host shown in Figure 1.5. For a compute node in the default Docker setup, containers on the same host communicate with each other, as well as with other containers and services hosted on the other host with a Docker bridge. In Contrail networking, on each compute node the vRouter creates a VRF table per virtual network, offering a long list of features.

From the perspective of the control plane, the Contrail vRouter:

- Receives low-level configuration (routing instances and forwarding policy).
- Exchanges routes.
- Installs forwarding state into the forwarding plane.
- Reports analytics (logs, statistics, and events).

From the prospective of the data plane, the Contrail vRouter:

- Assigns received packets from the overlay network to a routing instance based on the MPLS label or Virtual Network Identifier (VNI).
- Proxies DHCP, ARP, and DNS.
- Applies forwarding policy for the first packet of each new flow then programs the action to the flow entry in the flow table of the forwarding plane.
- Forwards the packetst after a destination address lookup (IP or MAC) in the Forwarding Information Base (FIB), encapsulating/decapsulating packets sent to or received from the overlay network.

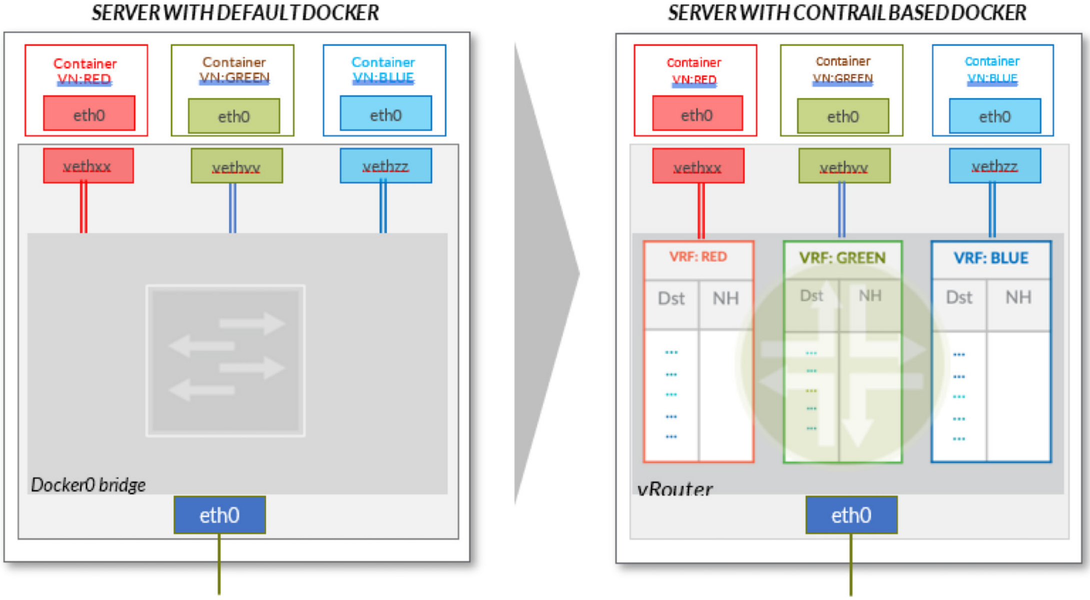


Figure 1.5 Docker and Contrail vRouter

Chapter 2

Kubernetes Basics

This chapter introduces Kubernetes, and the basic terminologies, key concepts, and most of the frequently referred components in Kubernetes architecture. This chapter also provides some examples in a Kubernetes cluster environment to demonstrate the key ideas about basic Kubernetes objects.

What is Kubernetes?

You can find the official definition of Kubernetes here (https://kubernetes.io/):

"Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Kubernetes builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community."

Here are a few important facts about Kubernetes:

- it's an open-source project initiated by Google
- it's a mature and stable product
- it's an orchestration tool
- it's a platform dealing with containers at a higher level

Kubernetes was created by a group of engineers at Google in 2014, with a design and development model influenced by Google's internal system, Borg. Kubernetes defines a set of building objects that collectively provide mechanisms that orchestrate containerized applications across a distributed cluster of nodes, based on system resources (CPU, memory, or other custom metrics). Kubernetes masks the complexity of managing a group of containers by providing REST APIs for the required functionalities.

In simple terms, container technologies like Docker provide you with the capability to package and distribute containerized applications, while an orchestration system like Kubernetes allows you to deploy and manage the containers at a relatively higher level and in a much easier way.

NOTE Many Kubernetes documents frequently abbreviate the technology as k8s (or K - eight characters - S), and the current major release (as of the writing of this book) is v1.14.

Chapter 1 stated that Docker is a prevailing and mature container technology, so why do you need Kubernetes? Technically speaking, Kubernetes works at a relatively higher level than Dockers, so what does that mean, exactly?

Well, when you compare Kubernetes with Docker, a helpful analogy is comparing Python with C language. C is powerful enough to build almost everything, including a whole bunch of fundamental OS components and APIs, but in practice you probably would prefer to write scripts to automate tasks in your workload, which means using Python much more than using C. With Python you only need to think of which existing module already provides the necessary functions, import it in your application, and then quickly focus on how to use the feature to accomplish what you need. You rarely need to worry about the low-level system API calls and hardware details.

A network analogy is the TCP/IP Internet protocols. When you develop a file transfer tool like FTP, naturally you prefer to start your work based on a TCP socket instead of a raw socket. With the TCP socket you are sitting on top of the TCP protocol, which provides a much more solid foundation that has all of the built-in reliability features like error detection, flow and congestion control, retransmission, and so on. What you need to consider is how to deliver the data from one end and receive it on the other end. With a raw socket you are working on the IP protocol and an even lower layer, so you have to consider and implement all of the reliability features before you can even start to work on the file transfer features of your tool.

So, back to Kubernetes. Assuming that you want to run multiple containers across multiple machines, you will have a lot of work to do if you interact with Docker directly. The following tasks should, at least minimally, be on your list of things to worry about:

- Loging in on different machines, and spawning containers, across the network
- Scaling up or down when demand changes by adding or removing containers

- Keeping storage consistent with multiple instances of an application
- Distributing load between the containers running in different nodes
- Launching new containers on different machines if something fails

You will quickly find that doing all of these manually with Docker will be overwhelming. With the high-level abstractions and the objects representing them in the Kubernetes API, all of these tasks become much easier.

NOTE Kubernetes is not the only tool of its kind, Docker has its own orchestration tool named Swarm. But that's a discussion for another book. This book focuses on Kubernetes.

Kubernetes Architecture and Components

There are two type of nodes in a Kubernetes cluster, and each one runs a well-defined set of processes:

- head node: also called master, or master node, it is the head and brain that does all the thinking and makes all the decisions; all of the intelligence is located here.
- worker node: also called node, or minion, it's the hands and feet that conducts the workforce.

The nodes are controlled by the master and in most cases, you only need to talk to the master.

One of the most common interfaces between you and the cluster is the command-line tool kubectl. It is installed as a client application, either in the same master node or in a separate machine, like in your PC. Regardless of where it is, it can talk to the master via the REST-API exposed by the master.

Later in this book you can see an example of using kubectl to create Kubernetes objects. But for now, just remember, whenever you work with the kubectl command, you're communicating with the cluster's master.

NOTE The term node may sound semantically ambiguous – it could mean two things in the context of this book. Usually a node refers to a logical unit in a cluster, like a server, which can be either be physical or virtual. In context of Kubernetes clusters, a node usually refers specifically to a worker node.

NOTE You rarely need to bypass the master and work with nodes, but you can log in to a node and run all Docker commands to check running status of the containers. An example of this appears later in this chapter.

Kubernetes Master

A Kubernetes master node, or master, is the brain. The cluster master provides the control plane that makes all of the global decisions about the cluster. For example, when you need the cluster to spawn a container, the master will decide which node to dispatch the task and spawn a new container. This procedure is called *scheduling*.

The master is responsible for maintaining the desired state for the cluster. When you give an order for this web server, make sure there are always two containers backing each other up! The master monitors the running status, and spawns a new container any time fewer than two web server containers are running due to any failures.

Typically you only need a single master node in the cluster, however, the master can also be replicated for higher availability and redundancy. The master's functions are implemented by a collection of processes running in the master node:

- `kube-apiserver`: Is the front-end of the control plane, and provides REST APIs.
- `kube-scheduler`: Does the scheduling and decides where to place the containers depending on system requirements (CPU, memory, storage, etc.) and other custom parameters or constraints (e.g., affinity specifications).
- `kube-controller-manager`: The single process that controls most of the different types of controllers, ensuring that the state of the system is what it should be. Controller examples might be:
  - Replication Controller
  - ReplicaSet
  - Deployment
  - Service Controller
- `etcd`: The database to store the state of the system.

NOTE For the sake of simplicity, some components are not listed (e.g., cloud-controller-manager, DNS server, kubelet). They are not trivial or negligible components, but skipping them for now helps us get past the Kubernetes basics.

Kubernetes Node

Kubernetes nodes in a cluster are the machines that run the user end applications. In production environments, there can be dozens or hundreds of nodes in one cluster, depending on the designed scales as they work under the hood provided by a cluster. Usually all of the containers and workloads are running on nodes. A node runs the following processes:

- kubelet: The Kubernetes agent process that runs on master and all the nodes. It interacts with master (through the kube-apiserver process) and manages the containers in the local host.
- kube-proxy: This process implements the Kubernetes service (introduced in Chapter 3) using Linux iptable in the node.
- container-runtime: Or the local container – mostly Docker in today's market, holding all of the running Dockerized applications.

NOTE The term proxy may sound confusing for Kubernetes beginners since it's not really a proxy in current Kubernetes architecture. Kube-proxy is a system that manipulates Linux IP tables in the node so the traffic between pods and nodes flows correctly.

Kubernetes Workflow

So far you've been reading about the master and node and the main processes running in each. Now it's time to visualize how things work together, as shown in Figure 2.1.

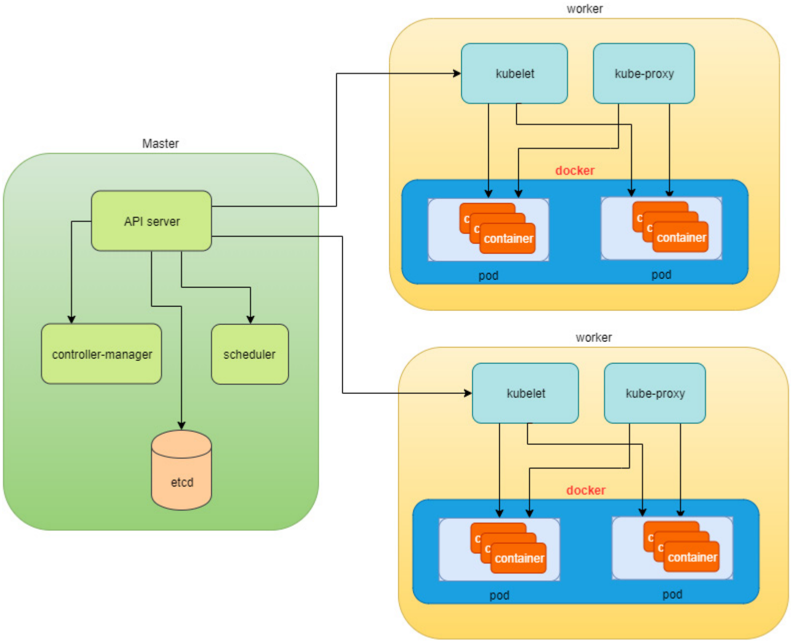


Figure 2.1 Kubernetes Architecture

At the top of Figure 2.1, via `kubectl` commands, you talk to the Kubernetes master, which manages the two node boxes on the right. Kubectl interacts with the master process kube-apiserver via its REST-API exposed to the user and other processes in the system.

Let's send some kubectl commands – something like `kubectl create x`, to spawn a new container. You can provide details about the container to be spawned along with its running behaviors, and those specifications can be provided either as `kubectl` command line parameters, or options and values defined in a configuration file (an example on this appears shortly). The workflow would be:

1. The *kubectl* client will first translate your CLI command to one more REST-API call(s) and send it to kube-apiserver.
2. After validating these REST-API calls, kube-apiserver understands the task and calls kube-scheduler process to select one node from the available ones to execute the job. This is the scheduling procedure.
3. Once kube-scheduler returns the target node, and kube-apiserver will dispatch the task with all of the details describing the task.
4. The kubelet process in the target node receives the task and talks to the container engine, for example, the Docker engine in Figure 2.1, to spawn a container with all provided parameters.
5. This job and its specification will be recorded in a centralized database etcd. Its job is to preserve and provide access to all data in the cluster.

NOTE Actually a master can also be a fully-featured node and carry pods workforce just like a node does. Therefore, kubelet and kube proxy components existing in node can also exist in the master. In Figure 2.1, we didn't include these components in the master, in order to provide a simplified conceptual separation of master and node. In your setup you can use command `kubectl get pods --all-namespaces -o wide` to list all pods with their location. Pods spawned in the master are usually running as part of the Kubernetes system itself – typically within kube-system namespace. The Kubernetes namespace is discussed in Chapter 3.

Of course this is a simplified workflow, but you should get the basic idea. In fact, with the power of Kubernetes, you rarely need to work directly with containers. You work with higher level objects that tend to hide most of the low level operation details.

For example, in Figure 2.1 when you give the task to spawn containers, instead of saying: create two containers and make sure to spawn new ones if either one would fail, in practice you just say: create a RC object (replication controller) with replica two.

Once the two Docker containers are up and running, kubeapiserver will interact with kube-controller-manager to keep monitoring the job status and take all necessary actions to make sure the running status is what it was defined as. For example, if any of the Docker containers go down, a new container will automatically be spawned and the broken one will be removed.

The RC in this example is one of the objects that is provided by the Kubernetes kube-controller-manager process. Kubernetes objects provide an extra layer of abstraction that gets the same (and usually more) work done under the hood, in a simpler and cleaner way. And because you are working at a higher level and staying away from the low-level details, Kubernetes objects sharply reduce your overall deployment time, brain effort, and troubleshooting pains. Let's examine.

Kubernetes Objects

Now that you understand the role of master and node in a Kubernetes cluster, and understand the workflow model in Figure 2.1, let's look at more objects in the Kubernetes architecture.

Kubernetes's objects represent:

- deployed containerized applications and workloads
- their associated network and disk resources
- other information about what the cluster is doing.

The most frequently used objects are:

- Pod
- Service
- Volume
- Namespace

The higher-level objects (Controllers) are:

- ReplicationController
- ReplicaSet
- Deployment
- StatefulSet
- DaemonSet
- Job

NOTE High-level objects are built upon basic objects, providing additional functionality and convenience features.

On the front end, Kubernetes gets things done via a group of objects, so with Kubernetes you only need to think about how to describe your task in the configuration file of objects, you don't have to worry about how it will be implemented in container level. Under the hood, Kubernetes interacts with the container engine to coordinate the scheduling and execution of containers on Kubelets. The container engine itself is responsible for running the actual container image (for example, by Docker build).

There are more examples about each object and its magic power in Chapter 3. First, let's look at the most fundamental object: pod.

Building a Kubernetes Pod

Pod is the first Kubernetes object you will learn. The Kubernetes website describes a pod as:

A pod (as in a pod of whales or pea pod) is a group of one or more containers (such as Docker containers), with shared storage/network, and a specification for how to run the containers.

Meaning:

- a pod is essentially a group of containers
- all containers in a pod share storage and network resources.

So what's the benefit of using pod compared to the old way of dealing with each individual container? Let's consider a simple use case: you are deploying a web service with Docker and you need not only the frontend service, for example an Apache server, but also some supporting services like a database server, a logging server, a monitoring server, and so forth. Each of these supporting services needs to be running in its own container. So essentially you find yourself always working with a group of docks whenever a web service container is needed. In production, the same scenario applies to most of the other services as well. Eventually you ask: is there a way to group a bunch of Docker containers in a higher-level unit, so you only need to worry once about the low-level inter-container interaction details?

Pod gives the exact higher-level abstraction you need by wrapping one or more containers into one object. If your web service becomes too popular and a single pod instance can't carry the load, you can replicate and scale the same group of containers (now in the form of one pod object) up and down very easily with the help of other objects (RC, deployment) - normally in a few seconds. This sharply increases deployment and maintenance efficiency.

In addition, containers in the same pod share the same network space, so containers can easily communicate with other containers in the same pod as though they were on the same machine, while maintaining a degree of isolation from others. You can read more about these advantages later in this book.

Now, let's get our feet wet and learn how to use a configuration file to launch a pod in a Kubernetes cluster.

YAML File for Kubernetes

Along with many other many ways of configuring Kubernetes, YAML is the standard format used in a Kubernetes configuration file. YAML is widely used, so mostly likely you are already familiar with it. If not, it's not a big deal because YAML is a pretty easy language to learn. Each line of the YAML configuration of a pod is detailed and you should understand the YAML format as a byproduct of your pod learning process.

The pod configuration file in YAML format is:

```
#pod-2containers-do-one.yaml    <1>
apiVersion: v1                  <2>
kind: Pod                       <3>
metadata:                       <4>
  name: pod-1                   <5>
  labels:                       <6>
      name: pod-1               <7>
spec:                           <8>
  containers:                   <9>
  - name: server                <10>
    image: contrailk8sdayone/contrail-webserver         <11>
    ports:                      <12>
    - containerPort: 80         <13>
  - name: client                <14>
    image: contrailk8sdayone/ubuntu                     <15>
```

YAML uses three basic data types:

1. scalars (strings/numbers): atom data item, strings like pod-1, port number 80.
2. mappings (hashes/dictionaries): key-value pairs, can be nested. apiVersion: v1 is a mapping. key apiVersion has a value of v1.
3. sequences (arrays/lists): collection of ordered values, without a key. List items are indicated by a - sign. The value of key containers is a list including two containers.

In this example you are also seeing nested YAML data structure:

- The mapping of a map: spec is the key of a map, where you define a pod's specification. In this example you only define the behavior of the containers to be launched in the pod. The value is another map with the key being containers.

- The mapping of a list. The values of the key containers are a list of two items: server and client container, each of which, again, are a mapping describing the individual container with a few attributes like name, image, and ports to be exposed.

Other characteristics you should know about YAML:

- It is case sensitive
- Elements in the same level share the same left indentation, the amount of indentation does not matter
- Tab characters are not allowed to be used as indentation
- Blank lines do not matter
- Use # to comment a line
- Use a single quote `'` to escape the special meaning of any character

Before diving into more details about the YAML file, let's finish the pod creation:

```
$ kubectl create -f pod-2containers-do-one.yaml
pod/pod-1 created

$ kubectl get pod -o wide
NAME   READY  STATUS             RESTARTS  AGE  IP             NODE     NOMINATED  NODE
pod-1  0/2    ContainerCreating  0         18s  10.47.255.237  cent333  <none>

$ kubectl get pod -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP              NODE      NOMINATED NODE
pod-1   2/2     Running   0          18s   10.47.255.237   cent333   <none>
```

There. We have created our first Kubernetes object – a pod named pod-1. But where are the containers? The output offers the clues: a pod pod-1 (NAME), containing two containers (READY /2), has been launched in the Kubernetes worker node cent333 with an assigned IP address of 10.47.255.237. Both containers in the pod are up (READY 2/) and it has been in running STATUS for 27s without any RESTARTS.

Here's a brief line-by-line commentary about what the YAML configuration is doing:

- Line 1: This is a comment line using # ahead of the text, you can put any comment in the YAML file. (Throughout this book we use this first line to give a filename to the YAML file. The filename is used later in the command when creating the object from the YAML file.)

- Lines 2, 3, 4, 8: The four YAML mappings are the main components of the pod definition:
  - ApiVersion: There are different versions, for example, v2. Here specifically, it is version 1.
  - Kind: Remember there are different type of Kubernetes objects, and here we want Kubernetes to create a pod object. Later, you will see the Kind being ReplicationController, or Service, in our examples of other objects.
  - Metadata: To identify the created objects. Besides the name of the object to be created, another important meta data are labels. And you will read more about that in Chapter 3.
  - Spec: This gives the specification about pod behavior.
- Lines 9-15: The pod specification here is just about the two containers. The system downloads the images, launches each container with a name, and exposes the specified ports, respectively.

Here's what's running inside of the pod:

```
  $ kubectl describe pod pod-1 | grep -iC1 container
  IP:                 10.47.255.237
  Containers:
    server:
      Container ID:   docker://9f8032f4fbe2f0d5f161f76b6da6d7560bd3c65e0af5f6e8d3186c6520cb3b7d
      Image:          contrailk8sdayone/contrail-webserver
  --
    client:
      Container ID:   docker://d9d7ffa2083f7baf0becc888797c71ddba78cd951f6724a10c7fec84aefce988
      Image:          contrailk8sdayone/ubuntu
  --
    Ready             True
    ContainersReady   True
    PodScheduled      True
  --
    Normal  Pulled     3m2s   kubelet, cent333   Successfully pulled image "contrailk8sdayone/
contrail-webserver"
    Normal  Created    3m2s   kubelet, cent333   Created container
    Normal  Started    3m2s   kubelet, cent333   Started container
    Normal  Pulling    3m2s   kubelet, cent333   pulling image "contrailk8sdayone/ubuntu"
    Normal  Pulled     3m1s   kubelet, cent333   Successfully pulled image "contrailk8sdayone/
ubuntu"
    Normal  Created    3m1s   kubelet, cent333   Created container
    Normal  Started    3m1s   kubelet, cent333   Started container
```

Not surprisingly, pod-1 is composed of two containers declared in the YAML file, server and client, respectively, with an IP address assigned by Kubernetes cluster and shared between all containers as shown in Figure 2.2:

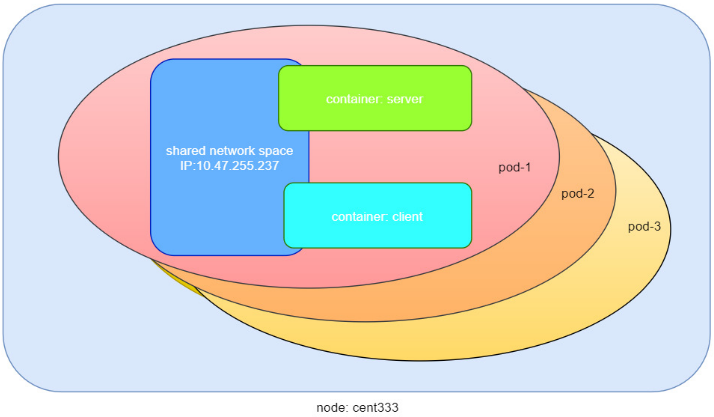


Figure 2.2 *Node, Pod, and Containers*

Pause Container

If you log in to node cent333, you'll see the Docker containers running inside of the pod:

```
$ docker ps | grep -E "ID|pod-1"
CONTAINER ID  IMAGE                             COMMAND              ... PORTS  NAMES
d9d7ffa2083f  contrailk8sdayone/ubuntu          "/sbin/init"         ...
k8s_client_pod-1_default_f8b42343-d87a-11e9-9a1e-0050569e6cfc_0
9f8032f4fbe2  contrailk8sdayone/contrail-webserver  "python app-dayone.py"  ...
k8s_server_pod-1_default_f8b42343-d87a-11e9-9a1e-0050569e6cfc_0
969ec6d93683  k8s.gcr.io/pause:3.1              "/pause"             ...
k8s_POD_pod-1_default_f8b42343-d87a-11e9-9a1e-0050569e6cfc_0
```

The third container with image name k8s.gcr.io/pause is a special container that was created for each pod by the Kubernetes system. The pause container is created to manage the network resources for the pod, which is shared by all the containers of that pod.

Figure 2.3 shows a pod including a few user containers and a pause container.

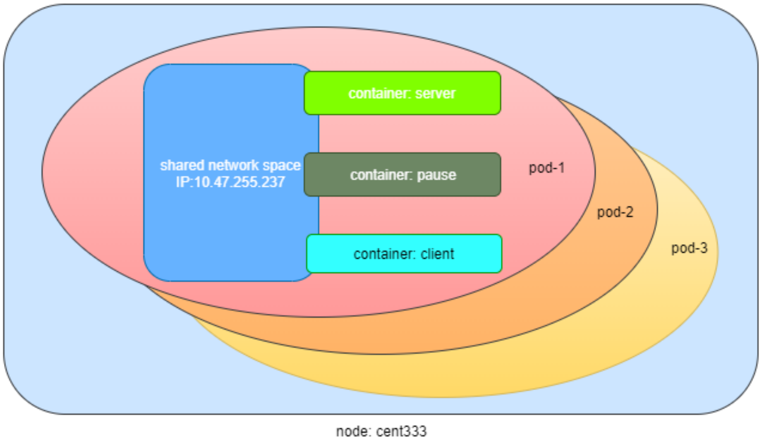


Figure 2.3 *Pod, User Containers, and the Special Pause Container*

Intra-pod Communication

In the Kubernetes master, let's log in to a container from the master:

```
----
#login to pod-1's container client
$ kubectl exec -it pod-1 -c client bash
root@pod-1:/#

#login to pod-1's container server
$ kubectl exec -it pod-1 -c server bash
root@pod-1:/app-dayone#
```

NOTE If you ever played with Docker you will immediately realize that this is pretty neat. Remember, the containers were launched at one of the nodes, so if you use Docker you will have to first log in to the correct remote node, and then use a similar docker exec command to log in to each container. Kubernetes hides these details. It allows you to do everything from one node – the master.

And now check processes running in the container:

Server Container

```
  root@pod-1:/app-dayone# ps aux
  USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
  root         1  0.0  0.0  55912 17356 ?        Ss   12:18   0:00 python app-dayo
  root         7  0.5  0.0 138504 17752 ?        Sl   12:18   0:05 /usr/bin/python
  root        10  0.0  0.0  18232  1888 pts/0    Ss   12:34   0:00 bash
  root        19  0.0  0.0  34412  1444 pts/0    R+   12:35   0:00 ps aux

  root@pod-1:/app-dayone# ss -ant
```

```
State      Recv-Q Send-Q Local Address:Port               Peer Address:Port
LISTEN     0      128          *:80                       *:*
LISTEN     0      128          *:22                       *:*
LISTEN     0      128         :::22                      :::*

root@pod-1:/app-dayone# 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
116: eth0@if117: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:f8:e6:63:7e:d8 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.47.255.237/12 scope global eth0
       valid_lft forever preferred_lft forever
```

The Client Container

```
$ kubectl exec -it pod-1 -c client bash
root@pod-1:/# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  32716  2088 ?        Ss   12:18   0:00 /sbin/init
root        41  0.0  0.0  23648   888 ?        Ss   12:18   0:00 cron
root        47  0.0  0.0  61364  3064 ?        Ss   12:18   0:00 /usr/sbin/sshd
syslog     111  0.0  0.0 116568  1172 ?        Ssl  12:18   0:00 rsyslogd
root       217  0.2  0.0  18168  1916 pts/0    Ss   12:45   0:00 bash
root       231  0.0  0.0  15560  1144 pts/0    R+   12:45   0:00 ps aux

root@pod-1:/# ss -ant
State      Recv-Q Send-Q        Local Address:Port          Peer Address:Port
LISTEN     0      128                       *:80                       *:*
LISTEN     0      128                       *:22                       *:*
LISTEN     0      128                      :::22                      :::*

root@pod-1:/# 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
116: eth0@if117: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:f8:e6:63:7e:d8 brd ff:ff:ff:ff:ff:ff
    inet 10.47.255.237/12 scope global eth0
       valid_lft forever preferred_lft forever
```

This ps command output shows that each container is running its own process. However, the ss and ip command output indicate that both containers share the same exact network environment, so both see the port exposed by each other. Therefore, communication between containers in a pod can happen simply by using localhost. Let's test this out by starting a TCP connection using the curl command.

Suppose from the client container, you want to get a web page from the server container. You can simply start curl using the localhost IP address:

```
  root@pod-1:/# curl localhost

  <html>
  <style>
    h1   {color:green}
    h2   {color:red}
  </style>
    <div align="center">
    <head>
      <title>Contrail Pod</title>
    </head>
    <body>
      <h1>Hello</h1><br><h2>This page is served by a <b>Contrail</b> pod</
h2><br><h3>IP address = 10.47.255.237<br>Hostname = pod-1</h3>
      <img src="/static/giphy.gif">
    </body>
    </div>
  </html>
```

You can see that the connection is established, and the web page has downloaded successfully.

Now let’s monitor the TCP connection state: the connection was established successfully:

```
root@pod-1:/# ss -ant
State      Recv-Q Send-Q        Local Address:Port          Peer Address:Port
LISTEN     0      128                       *:80                       *:*
LISTEN     0      128                       *:22                       *:*
TIME-WAIT  0      0                 127.0.0.1:80               127.0.0.1:34176 #<---
LISTEN     0      128                      :::22                      :::*
```

And the same exact connection can be seen from the server container:

```
$ kubectl exec -it pod-1 -c server bash
root@pod-1:/app-dayone# ss -ant
State      Recv-Q Send-Q Local Address:Port               Peer Address:Port
LISTEN     0      128          *:80                       *:*
LISTEN     0      128          *:22                       *:*
TIME-WAIT  0      0      127.0.0.1:80                 127.0.0.1:34176   #<---
LISTEN     0      128         :::22                      :::*
```

Kubectl Tool

So far you’ve seen the object created by the kubectl command. This command, just like the docker command in Docker world, is the interface in the Kubernetes world to talk to the cluster, or more precisely, the Kubernetes master, via Kubernetes API. It’s a versatile tool that provides options to fulfill all kinds of tasks you would need to deal with Kubernetes.

As a quick example, assuming you have enabled the auto-completion feature for kubectl, you can list all of the options supported in your current environment by logging into the master and typing kubectl, followed by two tab keystrokes:

```
root@test1:~# kubectl<TAB><TAB>
alpha          attach         completion     create         exec
logs           proxy          set            wait annotate  auth
config         delete         explain        options        replace
taint          api-resources  autoscale      convert        describe
patch          rollout        top            api-versions   certificate
drain          get            plugin         run            uncordon apply
cluster-info   cp             edit           label          port-forward
scale          version        expose         cordon
```

NOTE To set up auto-completion for the kubectl command, follow the instruction from the help of completion option:
kubectl completion -h

Rest assured, you'll see and learn some of these options in the remainder of this book.

Chapter 3

Kubernetes in Practice

This chapter introduces some of the fundamental objects and features of Kubernetes.

Imagine you have a pod that needs to be hosted on a machine with certain specifications (SSD HD, physical location, processing power, etc.) or you want to search or group your pods for easier administration. What do you do? Labels are the way to go. In Kubernetes, labels are attached to an object.

Let's use labels to launch a pod on a certain machine.

Labels

In Kubernetes, any object can be identified using a label.

You can assign multiple labels per object, but you should avoid using too many labels, or too few; too many will get you confused and too few won't give the real benefits of grouping, selecting, and searching.

Best practice is to assign labels to indicate:

- application/program ID using this pod
- owner (who manages this pod/application)
- stage (the pod/application in development/testing/production version)
- resource requirements (SSD, CPU, storage)
- location (preferred location/zone/data center to run this pod/application)

Okay, let's assign labels for (`stage: testing`) and (`zone: production`) to two nodes, respectively, then try to launch a pod in a node that has the label (`stage: testing`):

```
$kubectl get nodes --show-labels

NAME      STATUS     ROLES     AGE       VERSION   LABELS
cent222   Ready      <none>    2h        v1.9.2    <none>
cent111   NotReady   <none>    2h        v1.9.2    <none>
cent333   Ready      <none>    2h        v1.9.2    <none>

$kubectl label nodes cent333 stage=testing
$kubectl label nodes cent222 stage=production

$kubectl get nodes --show-labels

NAME         STATUS    ROLES   AGE  VERSION  LABELS
cent222  Ready     <none>  2h   v1.9.2   stage=production
cent111  NotReady  <none>  2h   v1.9.2   <none>
cent333  Ready     <none>  2h   v1.9.2   stage=testing
```

Now let's launch a basic Nginx pod tagged with `stage: testing` in the nodeSelector and confirm it will land on a node tagged with `stage: testing`. Kube-scheduler uses labels mentioned in the nodeSelector section of the pod YAML to select the node to launch the pod:

NOTE Kube-scheduler picks the node based on various factors like individual and collective resource requirements, hardware, software, or policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

```
#pod-webserver-do-label.yaml
apiVersion: v1
kind: Pod
metadata:
  name: contrail-webserver
  labels:
    app: webserver
spec:
  containers:
  - name: contrail-webserver
    image: contrailk8sdayone/contrail-webserver
  nodeSelector:
    stage: testing
$
$ kubectl create -f pod-webserver-do-label.yaml
pod "contrail-webserver" created

$ kubectl get pods --output=wide
NAME                 READY     STATUS    RESTARTS   AGE       IP              NODE
contrail-webserver   1/1       Running   0          48s       10.47.255.238   cent333
```

NOTE You can assign a pod to a certain node without labels by adding the argument `nodeName: nodeX` under the spec in the YAML file where `nodeX` is the name of the node.

Namespace

As in many other platforms, there is normally more than one user (or team) working on a Kubernetes cluster. Suppose a pod named *webserver1* has been built by a devops department, but when sales department attempts to launch a pod with the same name, the system will give an error:

```
Error from server (AlreadyExists): error when creating "webserver1.
yaml": pods "webserver1" already exists
```

Kubernetes won't allow the same object name for the Kubernetes resources to appear more than once in the same scope.

Namespaces provide the scope for the Kubernetes resource like project/tenant in OpenStack. Names of resources need to be unique within a namespace, but not across namespaces. It's a natural way to divide cluster resources between multiple users.

Kubernetes starts with three initial namespaces:

- *default*: The default namespace for objects with no other namespace.
- *kube-system*: The namespace for objects created by the Kubernetes system.
- *kube-public*: Initially created by kubeadm tool when deploying a cluster. By convention the purpose of this namespace is to make some resources readable by all users without authentication. It exists mostly in Kubernetes clusters boot-stapped with the kubeadm tool only.

Create a Namespace

Creating a namespace is pretty simple. The `kubectl` command does the magic. You don't need to have a YAML file:

```
root@test3:~# kubectl create ns dev
namespace/dev created
```

And the new namespace *dev* is now created:

```
root@test3:~# kubectl get ns
NAME          STATUS    AGE
default       Active    15d
dev           Active    5s  #<-----
```

Now the webserver1 pod in *dev* namespace won't conflict with webserver1 pod in the *sales* namespace:

```
$ kubectl get pod --all-namespaces -o wide
NAMESPACE  NAME        READY  STATUS   RESTARTS  AGE   IP             NODE     NOMINATED  NODE
dev        webserver1  1/1    Running  4         2d4h  10.47.255.249  cent222  <none>
sales      webserver1  1/1    Running  4         2d4h  10.47.255.244  cent222  <none>
```

Quota

You can now apply constraints that limit resource consumption per namespace, similar to the OpenStack tenant. For example, you can limit the quantity of objects that can be created in a namespace, the total amount of compute resources that may be consumed by resources, etc. The constraint in k8s is called *quota*. Here's an example:

```
kubectl -n dev create quota quota-onepod --hard pods=1
```

There, we just created `quota quota-onepod`, and the constraint we gave is `pods=1` – so only one pod is allowed to be created in this namespace:

```
$ kubectl get quota -n dev
NAME            CREATED AT
quota-onepod    2019-06-14T04:25:37Z

$ kubectl get quota -o yaml
apiVersion: v1
items:
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    creationTimestamp: 2019-06-14T04:25:37Z
    name: quota-onepod
    namespace: dev
    resourceVersion: "823606"
    selfLink: /api/v1/namespaces/dev/resourcequotas/quota-onepod
    uid: 76052368-8e5c-11e9-87fb-0050569e6cfc
  spec:
    hard:
      pods: "1"
  status:
    hard:
      pods: "1"
    used:
      pods: "1"
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

And now create a pod within it:

```
$ kubectl create -f pod-webserver-do.yaml -n dev
pod/contrail-webserver created
```

That works fine, so now let's create a second pod in it:

```
$ kubectl create -f pod-2containers-do.yaml -n dev
Error from server (Forbidden): error when creating
"pod/pod-2containers-do.yaml": pods "pod-1" is forbidden: exceeded quota:
quota-onepod, requested: pods=1, used: pods=1, limited: pods=1
```

Immediately we run into the error `exceeded quota`. Let's delete the `quota quota-onepod`. This new pod will be created after the quota is removed:

```
$ kubectl delete quota quota-onepod -n dev
resourcequota "quota-onepod" deleted
$ kubectl create -f pod/pod-2containers-do.yaml -n dev
pod/pod-1 created
```

ReplicationController

You learned how to launch a pod representing your containers from its YAML file in Chapter 2. One question might arise in your container-filled mind: what if I need three pods that are exactly the same (each runs an Apache container) to make sure the web service appears more robust? Do I change the name in the YAML file then repeat the same commands to create the required pods? Or maybe with a shell script? Kubernetes already has the objects to address this demand with RC - `ReplicationController`, or RS – `ReplicaSet`.

A *ReplicationController* (rc) ensures that a specified number of pod replicas are running at any one time. In other words, a replication controller makes sure that a pod or a homogeneous set of pods is always up and available.

Creating an rc

Let's create an rc with an example. First create a YAML file for an rc object named webserver:

```
#rc-webserver-do.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: webserver
  labels:
    app: webserver
spec:
  replicas: 3
  selector:
    app: webserver
  template:
    metadata:
      name: webserver
      labels:
        app: webserver
    spec:
      containers:
      - name: webserver
        image: contrailk8sdayone/contrail-webserver
        securityContext:
           privileged: true
        ports:
        - containerPort: 80
contrailK8sdayone
```

Remember that kind indicates the object type that this YAML file defines, here it is an rc instead of a pod. In metadata it is showing the rc's name as webserver. The spec is the detail specification of this rc object, and replicas: 3 indicates the same pod will be cloned to make sure the total number of pods created by the rc is always three. Finally, the template provides information about the containers that will run in the pod, the same as what you saw in a pod YAML file. Now use this YAML file to create the rc object:

```
kubectl create -f rc-webserver-do.yaml
replicationcontroller "webserver" created
```

If you are quick enough, you may capture the intermediate status when the new pods are being created:

```
$ kubectl get pod
NAME              READY     STATUS              RESTARTS   AGE
webserver-5ggv6   1/1       Running             0          9s
webserver-lbj89   0/1       ContainerCreating   0          9s
webserver-m6nrx   0/1       ContainerCreating   0          9s
```

Eventually you will see three pods launched:

```
$ kubectl get rc
NAME            DESIRED   CURRENT   READY   AGE
webserver          3         3         3       3m29s
$ kubectl get pod
NAME              READY     STATUS    RESTARTS   AGE
webserver-5ggv6   1/1       Running   0          21m
webserver-lbj89   1/1       Running   0          21m
webserver-m6nrx   1/1       Running   0          21m
```

Rc works with the pod directly. The workflows are shown in Figure 3.1.

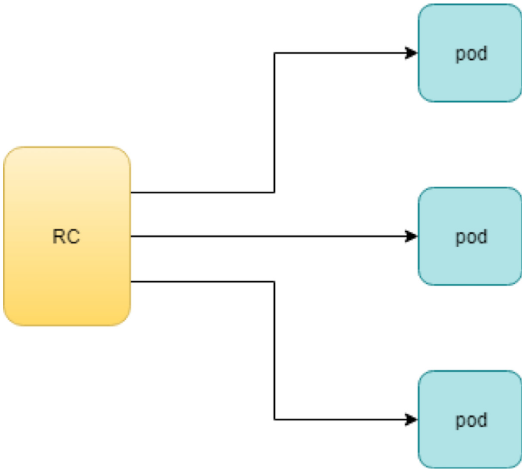


Figure 3.1 rc Workflows

With the replicas parameter specified in the rc object YAML file, the Kubernetes replication controller, running as part of kube-controller-manager process in the master node, will keep monitoring the number of running pods spawned by the rc and automatically launch new ones should any of them run into failure. The key thing to learn is that individual pods may die any time, but the pool as a whole is always up and running, making a robust service. You will understand this better when you learn Kubernetes service.

Test Rc

You can test an rc's impact by deleting one of the pods. To delete a resource with kubectl, use the kubectl delete sub-command:

```
$ kubectl delete pod webserver-5ggv6
pod "webserver-5ggv6" deleted
$ kubectl get pod
NAME          READY     STATUS        RESTARTS   AGE
webserver-5ggv6   0/1       Terminating   0          22m        #<---
webserver-5v9w6   1/1       Running       0          2s         #<---
webserver-lbj89   1/1       Running       0          22m
webserver-m6nrx   1/1       Running       0          22m
$ kubectl get pod
NAME          READY     STATUS        RESTARTS   AGE
webserver-5v9w6   1/1       Running       0          5s
webserver-lbj89   1/1       Running       0          22m
webserver-m6nrx   1/1       Running       0          22m
```

As you can see, when one pod is being terminated, a new pod is immediately spawned. Eventually the old pod will go away and the new pod will be up and running. The total number of running pods will remain unchanged.

You can also scale up or down replicas with rc. For example, to scale up from number of 3 to 5:

```
$ kubectl scale rc webserver --replica=5
replicationcontroller/webserver scaled
$ kubectl get pod
NAME              READY     STATUS              RESTARTS   AGE
webserver-5v9w6   1/1       Running             0          8s
webserver-lbj89   1/1       Running             0          22m
webserver-m6nrx   1/1       Running             0          22m
webserver-hnnlj   0/1       ContainerCreating   0          2s
webserver-kbgwm   1/1       ContainerCreating   0          2s
$ kubectl get pod
NAME              READY     STATUS        RESTARTS   AGE
webserver-5v9w6   1/1       Running       0          10s
webserver-lbj89   1/1       Running       0          22m
webserver-m6nrx   1/1       Running       0          22m
webserver-hnnlj   1/1       Running       0          5s
webserver-kbgwm   1/1       Running       0          5s
```

There are other benefits with rc. Actually, since this abstraction is so popular and heavily used, two very similar objects, rs - ReplicaSet and Deploy – Deployment, have been developed with more powerful features. Generally speaking, you can call them next generation rc. For now, let's stop exploring more rc features and move our focus to these two new objects.

Before moving to the next object, you can delete the rc:

```
$ kubectl delete rc webserver
replicationcontroller/webserver deleted
```

ReplicaSet

ReplicaSet, or rs object, is pretty much the same thing as an rc object, with just one major exception – the looks of selector:

```
#rs-webserver-do.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: webserver
  labels:
    app: webserver
spec:
  replicas: 3
  selector:
    matchLabels:                                     #<---
      app: webserver                                 #<---
    matchExpressions:                                #<---
      - {key: app, operator: In, values: [webserver]}   #<---
  template:
    metadata:
      name: webserver
      labels:

    spec:
      containers:
      - name: webserver
        image: contrailk8sdayone/contrail-webserver
        securityContext:
           privileged: true
        ports:
        - containerPort: 80webservercontrailK8sdayone
```

Rc uses *equally-based* selectors only, while rs supports an extra selector format, *set-based*. Functionally the two forms of selectors do the same job – that is—select the pod with a matching label:

```
#RS selector
matchLabels:
  app: webserver
webserver
  matchExpressions:
  - {key: app, operator: In, values: [webserver]}

#RC selector
app: webserver
webserver
$ kubectl create -f rs-webserver.yaml
replicaset.extensions/webserver created
$ kubectl get pod
NAME                             READY   STATUS    RESTARTS   AGE
webserver-lkwvt                  1/1     Running   0          8s
```

An rs is created and it launches a pod, just the same as what an rc would do. If you compare the kubectl `describe` on the two objects:

```
$ kubectl describe rs webserver
......
Selector:     app=webserver,app in (webserver)      #<---
......
  Type    Reason            Age   From                   Message
  ----    ------            ----  ----                   -------
  Normal  SuccessfulCreate  15s   replicaset-controller  Created pod: webserver-lkwvt
$ kubectl describe rc webserver
......
Selector:     app=webserver                     #<---
......
  Type    Reason            Age   From                    Message
  ----    ------            ----  ----                    -------
  Normal  SuccessfulCreate  19s   replication-controller  Created pod: webserver-lkwvt
```

As you can see, for the most part the outputs are the same, with the only exception of the selector format. You can also scale the rs the same way as you would do with rc:

```
$ kubectl scale rs webserver --replicas=5
replicaset.extensions/webserver scaled
$ kubectl get pod
NAME                         READY   STATUS    RESTARTS   AGE
webserver-4jvvx                  1/1     Running   0          3m30s
webserver-722pf                  1/1     Running   0          3m30s
webserver-8z8f8                  1/1     Running   0          3m30s
webserver-lkwvt                  1/1     Running   0          4m28s
webserver-ww9tn                  1/1     Running   0          3m30s
```

Before moving to the next object, delete the rs:

```
$ kubectl delete rs webserver
replicaset.extensions/webserver deleted
```

Deployment

You may wonder why Kubernetes has different objects to do almost the same job. As mentioned earlier, the features of rc have been extended through the rs and deployment. We've seen the rs, which has done the same job of rc, only with a different selector format. Now we'll check out the other new object, DEPLOY – deployment, and explore the features coming from it.

Create a Deployment

If you simply change the kind attribute from ReplicaSet to Deployment you'll get the YAML file of a deployment object:

```
#deploy-webserver-do.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver
  labels:
    app: webserver
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webserver
    matchExpressions:
      - {key: app, operator: In, values: [webserver]}
  template:
    metadata:
      name: webserver
      labels:
        app: webserver
    spec:
      containers:
      - name: webserver
        image: contrailk8sdayone/contrail-webserver
        securityContext:
           privileged: true
        ports:
        - containerPort: 80
```

Create a deployment with the `kubectl` command:

```
  ----
  $ kubectl create -f deploy-webserver-do.yaml
  deployment.extensions/webserver created

  $ kubectl get deployment
  NAME                       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
  deployment.apps/webserver  1         1         1            1           21s
  ----
```

Actually, the deployment is a relatively higher-level of abstraction than rc and rs. Deployment does not create a pod directly, and the describe command reveals this:

```
$ kubectl describe deployments
Name:                   webserver
Namespace:              default
CreationTimestamp:      Sat, 14 Sep 2019 23:17:17 -0400
Labels:                 app=webserver
Annotations:            deployment.kubernetes.io/revision: 5
                        kubectl.kubernetes.io/last-applied-configuration:
                          {"apiVersion":"apps/v1","kind":"Deployment",
                          "metadata":{"annotations":{},"labels":{"app":"webserver"},
                          "name":"webserver","namespace":"defa...
Selector:               app=webserver,app in (webserver)
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=webserver
  Containers:
   webserver:
    Image:        contrailk8sdayone/contrail-webserver
    Port:         80/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   webserver-7c7c458cc5 (1/1 replicas created)    #<---
Events:          <none>
```

Deployment Workflow

When you create a deployment a replica set is automatically created. The pods defined in the deployment object are created and supervised by the deployment's replicaset.

The workflow is shown in Figure 3.2:

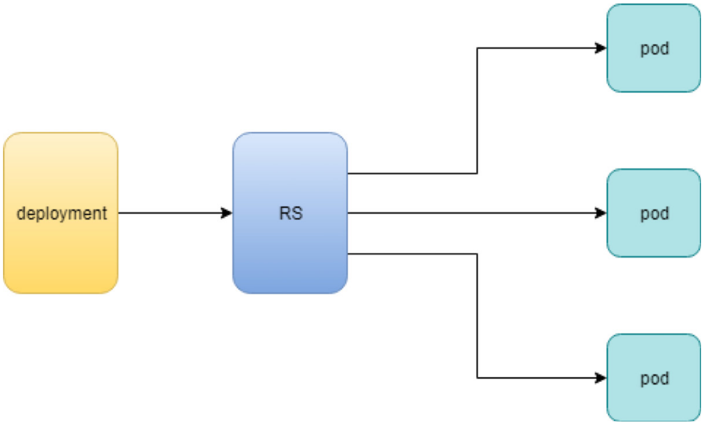


Figure 3.2 Deployment Workflow

You might still be wondering why you need rs as one more layer sitting between deployment and pod and that's answered next.

Rolling Update

The rolling update feature is one of the more powerful features that comes with the deployment object. Let's demonstrate the feature with a test case to explain how it works.

NOTE In fact, a similar *rolling update* feature exists for the old rc object. The implementation has quite a few drawbacks compared with the new version supported by Deployment. In this book we focus on the new implementation with Deployment.

Test Case: Rolling Update

Suppose you have a `nginx-deployment`, with replica=3 and pod image 1.7.9. We want to upgrade the image from version 1.7.9 to the new image version 1.9.1. With kuberctl you can use the set image option and specify the new version number to trigger the update:

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1
deployment.extensions/nginx-deployment image updated
```

Now check the deployment information again:

```
$ kubectl describe deployment/nginx-deployment
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Tue, 11 Sep 2018 20:49:45 -0400
Labels:                 app=nginx
Annotations:            deployment.Kubernetes.io/revision=2
Selector:               app=nginx
Replicas:               3 desired | 1 updated | 4 total | 3 available | 1 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        nginx:1.9.1       #<------
    Port:         80/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    ReplicaSetUpdated
OldReplicaSets:  nginx-deployment-67594d6bf6 (3/3 replicas created)
NewReplicaSet:   nginx-deployment-6fdbb596db (1/1 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  4m    deployment-controller  Scaled up replica
  set nginx-deployment-67594d6bf6 to 3  #<---
  Normal  ScalingReplicaSet  7s    deployment-controller  Scaled up replica
  set nginx-deployment-6fdbb596db to 1  #<---
```

There are two changes you can observe here:

- the image version in deployment is updated
- a new rs nginx-deployment-6fdbb596db is created, with a replica set to 1

And with the new rs with replica being 1, a new pod (the fourth one) is now generated:

```
$ kubectl get pods
NAME                                READY     STATUS              RESTARTS   AGE
nginx-deployment-67594d6bf6-88wqk   1/1       Running             0          4m
nginx-deployment-67594d6bf6-m4fbj   1/1       Running             0          4m
nginx-deployment-67594d6bf6-td2xn   1/1       Running             0          4m
nginx-deployment-6fdbb596db-4b8z7   0/1       ContainerCreating   0          17s        #<------
```

The new pod is with the new image:

```
$ kubectl describe pod/nginx-deployment-6fdbb596db-4b8z7 | grep Image:
...(snipped)...
    Image:          nginx:1.9.1     #<---
...(snipped)...
```

While the old pod is still with the old image:

```
$ kubectl describe pod/nginx-deployment-67594d6bf6-td2xn | grep Image:
...(snipped)...
    Image:          nginx:1.7.9     #<------
...(snipped)...
```

Let's wait, and keep checking the pods status… eventually all old pods are terminated, and three new pods are running – the pod names confirm they are new ones:

```
$ kubectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-6fdbb596db-4b8z7   1/1       Running   0          1m
nginx-deployment-6fdbb596db-bsw25   1/1       Running   0          18s
nginx-deployment-6fdbb596db-n9tpg   1/1       Running   0          21s
```

So the update is done, and all pods are now running with the new version of the image.

How It Works

Hold on, you might argue, this is not updated, this should be called a replacement because Kubernetes used three new pods with new images to replace the old pods! Precisely speaking, this is true. But this is how it works. Kubernetes's philosophy is that pods are cheap, and replacement is easy – imagine how much work it will be when you have to log in to each pod, uninstall old images, clean up the environment, only to install a new image. Let's look at more details about this process and understand why it is called a *rolling update*.

When you update the pod with new software, the deployment object introduces a new rs that will start the pod update process. The idea here is *not* to log in to the existing pod and do the image update in -place, instead, the new rs just creates a new pod equipped with the new software release in it. Once this new (and additional) pod is up and running, the original rs will be scaled down by one, so the total number of running pods remains unchanged. The new rs will continue to scale up by one and the original rs scales down by one. This process repeats until the number of pods created by the new rs reaches the original replica number defined in the deployment, and that is when all of the original rs pods are terminated. The process is depicted in Figure 3.3.

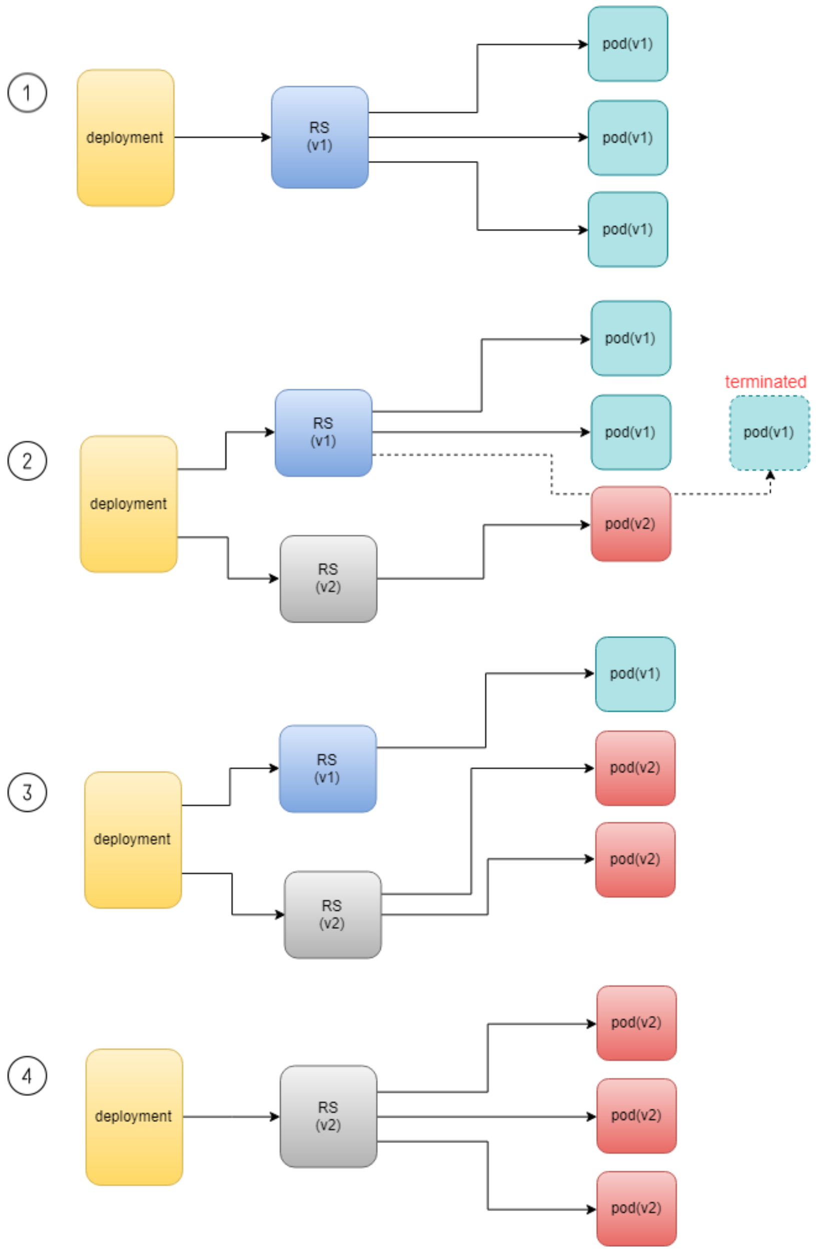


Figure 3.3 Deployment Overview

As you can see, the whole process of creating a new rs, scaling up the new rs, and scaling down the old one simultaneously, is fully automated and taken care of by the deployment object. It is deployment that is deploying and driving the ReplicaSet object, which, in this sense, is working merely as a backend.

This is why deployment is considered a higher-layer object in Kubernetes, and also the reason why it is officially recommended that you never use ReplicaSet alone, without deployment.

Record

Deployment also has the ability to record the whole process of rolling updates, so in case it is needed, you can review the update history after the update job is done:

```
$ kubectl describe deployment/nginx-deployment
Name:                   nginx-deployment
...(snipped)...
NewReplicaSet:   nginx-deployment-6fdbb596db (3/3 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  28m   deployment-controller  Scaled up replica set nginx-deployment-
67594d6bf6 to 3
  Normal  ScalingReplicaSet  24m   deployment-controller  Scaled up replica set nginx-deployment-
6fdbb596db to 1
  Normal  ScalingReplicaSet  23m   deployment-controller  Scaled down replica set nginx-deployment-
67594d6bf6 to 2
  Normal  ScalingReplicaSet  23m   deployment-controller  Scaled up replica set nginx-deployment-
6fdbb596db to 2
  Normal  ScalingReplicaSet  23m   deployment-controller  Scaled down replica set nginx-deployment-
67594d6bf6 to 1
  Normal  ScalingReplicaSet  23m   deployment-controller  Scaled up replica set nginx-deployment-
6fdbb596db to 3
  Normal  ScalingReplicaSet  23m   deployment-controller  Scaled down replica set nginx-deployment-
67594d6bf6 to 0
```

Pause/Resume/Undo

Additionally, you can also pause/resume the update process to verify the changes before proceeding:

```
$ kubectl rollout pause deployment/nginx-deployment
$ kubectl rollout resume deployment/nginx-deployment
```

You can even undo the update when things are going wrong during the maintenance window:

```
$ kubectl rollout undo deployment/nginx-deployment
$ kubectl describe deployment/nginx-deployment
Name:                   nginx-deployment
...(snipped)...
```

```
NewReplicaSet:   nginx-deployment-6fdbb596db (3/3 replicas created)
NewReplicaSet:   nginx-deployment-67594d6bf6 (3/3 replicas created)
Events:
  Type    Reason              Age From                  Message
  ----    ------              --- ----                  -------
  Normal  DeploymentRollback  8m  deployment-controller  Rolled back deployment "nginx-
deployment" to revision 1  #<------
  Normal  ScalingReplicaSet   8m  deployment-controller  Scaled up replica set nginx-deployment-
67594d6bf6 to 1   #<------
  Normal  ScalingReplicaSet   8m  deployment-controller  Scaled down replica set nginx-deployment-
6fdbb596db to 2 #<------
  Normal  ScalingReplicaSet   8m  deployment-controller  Scaled up replica set nginx-deployment-
67594d6bf6 to 2   #<------
  Normal  ScalingReplicaSet   8m  deployment-controller  Scaled up replica set nginx-deployment-
67594d6bf6 to 3   #<------
  Normal  ScalingReplicaSet   8m  deployment-controller  Scaled down replica set nginx-deployment-
6fdbb596db to 1 #<------
  Normal  ScalingReplicaSet   8m  deployment-controller  Scaled down replica set nginx-deployment-
6fdbb596db to 0 #<------
```

Typically you do this when something is broken in your deployment. Compared with how much work it takes to prepare for the software upgrade during maintenance windows in the old days, this is an amazing feature for anyone who suffered from software upgrade!

TIP This is pretty similar to the Junos rollback magic command that you probably use every day when you need to quickly revert the changes you make to your router.

Secrets

All modern network systems need to deal with sensitive information, such as username, passwords, SSH keys, etc. in the platform. The same applies to the pods in a Kubernetes environment. However, exposing this information in your pod specs as cleartext may introduce security concerns and you need a tool or method to resolve the issue – at least to avoid the cleartext credentials as much as possible.

The Kubernetes secrets object is designed specifically for this purpose – it encodes all sensitive data and exposes it to pods in a controlled way.

The official definition of Kubernetes secrets is:

"A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image; putting it in a secret object allows for more control over how it is used and reduces the risk of accidental exposure."

Users can create secrets, and the system also creates secrets. To use a secret, a pod needs to reference the secret.

There are many different types of secrets, each serving a specific use case, and there are also many methods to create a secret and a lot of different ways to refer to it in a pod. A complete discussion of secrets is beyond the scope of this book, so please refer to the official documentation to get all of the details and track all up-to-date changes.

Here, we'll look at some commonly used secret types. You will also learn several methods to create a secret and how to refer to it in your pods. And once you get to the end of the section, you should understand the main benefits of a Kubernetes secrets object and how it can help improve your system security.

Let's begin with a few secret terms:

- *Opaque*: This type of secret can contain arbitrary key-value pairs, so it is treated as unstructured data from Kubernetes' perspective. All other types of secret have constant content.
- *Kubernetes.io/Dockerconfigjson*: This type of secret is used to authenticate with a private container registry (for example, a Juniper server) to pull your own private image.
- *TLS*: A TLS secret contains a TLS private key and certificate. It is used to secure an ingress. You will see an example of an ingress with a TLS secret in Chapter 4.
- *Kubernetes.io/service-account-token*: When processes running in containers of a pod access the API server, they have to be authenticated as a particular account (for example, account default by default). An account associated with a pod is called a *service-account*. Kubernetes.io/service-account-token type of secret contains information about Kubernetes service-account. We won't elaborate on this type of secret and service-account in this book.
- *Opaque secret*: The secret of type opaque represents arbitrary user-owned data – usually you want to put some kind of sensitive data in secret, for example, username, password, security pin, etc., just about anything you believe is sensitive and you want to carry into your pod.

Define Opaque Secret

First, to make our sensitive data looks less sensitive, let's encode it with the base64 tool:

```
$ echo -n 'username1' | base64
dXNlcm5hbWUx
$ echo -n 'password1' | base64
cGFzc3dvcmQx
```

Then put the encoded version of the data in a secret definition YAML file:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-opaque
type: Opaque
data:
  username: dXNlcm5hbWUx
  password: cGFzc3dvcmQx
```

Alternatively, you can define the same secret from kubectl CLI directly, with the --from-literal option:

```
kubectl create secret generic secret-opaque     \
    --from-literal=username='username1'         \
    --from-literal=password='password1'
```

Either way, a secret will be generated:

```
$ kubectl get secrets
NAME                   TYPE                                  DATA   AGE
secret-opaque          Opaque                                2      8s

$ kubectl get secrets secret-opaque -o yaml
apiVersion: v1
data:
  password: cGFzc3dvcmQx
  username: dXNlcm5hbWUx
kind: Secret
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"password":"cGFzc3dvcmQx","username":"dXNlcm5hbWUx"},"kind":"Secret
","metadata":{"annotations":{},"name":"secret-opaque","namespace":"ns-user-1"},"type":"Opaque"}
  creationTimestamp: 2019-08-22T22:51:18Z
  name: secret-opaque
  namespace: ns-user-1
  resourceVersion: "885702"
  selfLink: /api/v1/namespaces/ns-user-1/secrets/secret-opaque
  uid: 5a78d9d4-c52f-11e9-90a3-0050569e6cfc
type: Opaque
```

Refer Opaque Secret

Next you will need to use the secret in a pod, and the user information contained in the secret will be carried into the pod. As mentioned, there are different ways to refer the opaque secret in a pod, and correspondingly, the result will be different.

Typically, user information carried from a secret can appear in a container in one of these forms:

- files
- environmental variables

Now let's demonstrate using secret to generate environmental variables in a container:

```
#pod-webserver-do-secret.yaml
apiVersion: v1
kind: Pod
metadata:
  name: contrail-webserver-secret
  labels:
    app: webserver
spec:
  containers:
  - name: contrail-webserver-secret
    image: contrailk8sdayone/contrail-webserver
    #envFrom:
    #- secretRef:
    #    name: test-secret
    env:
    - name: SECRET_USERNAME
      valueFrom:
        secretKeyRef:
          name: secret-opaque
          key: username
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: secret-opaque
          key: password
```

Spawn the pod and container from this YAML file:

```
$ kubectl apply -f pod/pod-webserver-do-secret.yaml
pod/contrail-webserver-secret created
```

Log in the container and verify the generated environmental variables:

```
$ kubectl exec -it contrail-webserver-secret -- printenv | grep SECRET
SECRET_USERNAME=username1
SECRET_PASSWORD=password1
```

The original sensitive data encoded with base64 is now present in the container!

Dockerconfigjson Secret

The dockerconfigjson secret, as the name indicates, carries the Docker account credential information that is typically stored in a .docker/config.json file. The image in a Kubernetes pod may point to a private container registry. In that case, Kubernetes needs to authenticate it with that registry in order to pull the image. The dockerconfigjson type of secret is designed for this very purpose.

Docker Credential Data

The most straightforward method to create a kubernetes.io/dockerconfigjson type of secret is to provide login information directly with the kubectl command and let it generate the secret:

```
$ kubectl create secret docker-registry secret-jnpr1 \
    --docker-server=hub.juniper.net                 \
    --docker-username=JNPR-FieldUser213             \
    --docker-password=CLJd2kpMsVc9zrAuTFPn
secret/secret-jnpr created
```

Verify the secret creation:

```
$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
secret-jnpr           kubernetes.io/dockerconfigjson        1      6s    #<---
default-token-hkkzr   kubernetes.io/service-account-token   3      62d
```

NOTE Only the first line in the output is the secret you have just created. The second line is a kubernetes.io/service-account-token type of secret that the Kubernetes system creates automatically when the contrail setup is up and running.

Now inspect the details of the secret:

```
$ kubectl get secrets secret-jnpr -o yaml
apiVersion: v1
data:
  .dockerconfigjson: eyJhdXRocyI6eyJodWIuanVuaXBlci5uZXQvc2...<snipped>...
kind: Secret
metadata:
  creationTimestamp: 2019-08-14T05:58:48Z
  name: secret-jnpr
  namespace: ns-user-1
  resourceVersion: "870370"
  selfLink: /api/v1/namespaces/ns-user-1/secrets/secret-jnpr
  uid: 9561cdc3-be58-11e9-9367-0050569e6cfc
type: kubernetes.io/dockerconfigjson
```

Not surprisingly, you don't see any sensitive information in the form of cleartext. There is a data portion of the output where you can see a very long string as the value of key: dockerconfigjson. Its appearance seems to have transformed from the original data, but at least it does not contain sensitive information anymore – after all one purpose of using a secret is to improve the system security.

However, the transformation is done by encoding, not encryption, so there is still a way to manually retrieve the original sensitive information: just pipe the value of key .dockerconfigjson into the base64 tool, and the original username and password information is viewable again:

```
$ echo "eyJhdXRocyI6eyJodWIuanVua..." | base64 -d | python -mjson.tool
{
    "auths": {
        "hub.juniper.net": {
            "auth": "Sj5QUi1GaWVsZFVzZXIyMTM6Q0xKZDJqcE1zVmM5enJBdVRGUG4=",
            "password": "CLJd2kpMsVc9zrAuTFPn",
            "username": "JNPR-FieldUser213"
        }
    }
}
```

Some highlights in this output are:

- The python -mjson.tool is used to format the decoded json data before displaying to the terminal.
- There is an auth key-value pair. It is the token generated based on the authentication information you gave (username and password).
- Later on, when equipped with this secret, a pod will use this token, instead of the username and password to authenticate itself towards the private Docker registry hub.juniper.net in order to pull a Docker image.

TIP Here's another way to decode the data directly from the secret object:

```
$ kubectl get secret secret-jnpr1 \
    --output="jsonpath={.data.\.dockerconfigjson}" \
    | base64 --decode | python -mjson.tool
{
    "auths": {
        "hub.juniper.net/security": {
            "auth": "Sj5QUi1GaWVsZFVzZXIyMTM6Q0xKZDJqcE1zVmM5enJBdVRGUG4=",
            "password": "CLJd2kpMsVc9zrAuTFPn",
            "username": "JNPR-FieldUser213"
        }
    }
}
```

The `--output=xxxx` option filters the kubectl get output so only the value of .dockerconfigjson under data is displayed. The value is then piped into base64 with option --decode (alias of -d) to get it decoded.

A docker-registry secret created manually like this will only work with a single private registry. To support multiple private container registries you can create a secret from the Docker credential file.

Docker Credential File (~/.Docker/config.json)

As the name of the key .dockerconfigjson in the secret we created indicates, it serves a similar role as the Docker config file: .docker/config.json. Actually, you can generate the secret directly from the Docker configuration file.

To generate the Docker credential information, first check the Docker config file:

```
$ cat .docker/config.json
{
    ......
    "auths": {},
    ......
}
```

There's nothing really here. Depending on the usage of the set up you may see different output, but the point is that this Docker config file will be updated automatically every time you docker login a new registry:

```
$ cat mydockerpass.txt | \
    docker login hub.juniper.net \
        --username JNPR-FieldUser213 \
        --password-stdin
Login Succeeded
```

The file mydockerpass.txt is the login password for username JNPR-FieldUser213. Saving the password in a file and then piping it to the docker login command with --password-stdin option has an advantage of not exposing the password cleartext in the shell history.

TIP If you want you can write the password directly, and you will get a friendly warning that this is insecure.

```
$ docker login hub.juniper.net --username XXXXXXXXXXXXXXX --password XXXXXXXXXXXXXX
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
Login Succeeded
```

Now the Docker credential information is generated in the updated `config.json` file:

```
$ cat .docker/config.json
{
    ......
    "auths": {    #<---
        "hub.juniper.net": {
            "auth": "Sj5QUi1GaWVsZFVzZXIyMTM6Q0xKZDJqcE1zVmM5enJBdVRGUG4="
        }
    },
    ......
}
```

The login process creates or updates a `config.json` file that holds the authorization token. Let's create a secret from the `.docker/config.json` file:

```
$ kubectl create secret generic secret-jnpr2 \
    --from-file=.dockerconfigjson=/root/.docker/config.json \
    --type=kubernetes.io/dockerconfigjson
secret/secret-jnpr2 created

$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
secret-jnpr2          kubernetes.io/dockerconfigjson        1      8s   #<---
default-token-hkkzr   kubernetes.io/service-account-token   3      63d
secret-jnpr           kubernetes.io/dockerconfigjson        1      26m
$ kubectl get secrets secret-jnpr2 -o yaml
apiVersion: v1
data:
  .dockerconfigjson: ewoJImF1dGhzIjoIlNrNVFVaTFHYVdWc1pGVnpaWEl5TVRNNlEweEtaREpxY0UxelZtTTVlbkpCZ
FZSR1VHND0iCgkJfQoJfSwKCSJIdHRwSGVhZGVycyI6IHsKCQkiVXNlci1BZ2VudCI6ICJEb2NrZXItQ2xpZW50LzE4LjAzLjE
tY2UgKGxpbnV4KSIKCX0sCgkiZGV0YWNoS2V5cyI6ICJjdHJsLUAiCn0=
kind: Secret
metadata:
  creationTimestamp: 2019-08-15T07:35:25Z
  name: csrx-secret-dr2
  namespace: ns-user-1
  resourceVersion: "878490"
  selfLink: /api/v1/namespaces/ns-user-1/secrets/secret-jnpr2
  uid: 3efc3bd8-bf2f-11e9-bb2a-0050569e6cfc
type: kubernetes.io/dockerconfigjson

$ kubectl get secret secret-jnpr2 --output="jsonpath={.data.\.dockerconfigjson}" | base64 --decode
{
    ......
    "auths": {
        "hub.juniper.net": {
            "auth": "Sj5QUi1GaWVsZFVzZXIyMTM6Q0xKZDJqcE1zVmM5enJBdVRGUG4="
        }
    },
    ......
}
```

YAML File

You can also create a secret directly from a YAML file the same way you create other objects like service or ingress.

To manually encode the content of the `.docker/config.json` file:

```
$ cat .docker/config.json | base64
ewoJImF1dGhzIjogewoJCSJodWIuanVuaXBlci5uZXQiOiB7CgkJCSJhdXRoIjogIlNrNVFVaTFH
YVdWc1pGVnpaWEl5TVRNNlEweEtaREpxY0UxelZtTTVlbkpCZFZSR1VHND0iCgkJfQoJfSwKCSJI
dHRwSGVhZGVycyI6IHsKCQkiVXNlci1BZ2VudCI6ICJEb2NrZXItQ2xpZW50LzE4LjAzLjEtY2Ug
KGxpbnV4KSIKCX0sCgkiZGV0YWNoS2V5cyI6ICJjdUAiCn0=
```

Then put the base64 encoded value of the `.docker/config.json` file as data in below the YAML file:

```
#secret-jnpr.yaml
apiVersion: v1
kind: Secret
type: kubernetes.io/dockerconfigjson
metadata:
  name: secret-jnpr3
  namespace: ns-user-1
data:
  .dockerconfigjson: ewoJImF1dGhzIjogewoJCSJodW......
$ kubectl apply -f secret-jnpr.yaml
secret/secret-jnpr3 created

$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-hkkzr   kubernetes.io/service-account-token   3      64d
secret-jnpr1          kubernetes.io/dockerconfigjson        1      9s
secret-jnpr2          kubernetes.io/dockerconfigjson        1      6m12s
secret-jnpr3          kubernetes.io/dockerconfigjson        1      78s
```

Keep in mind that base64 is all about encoding instead of encryption – it is considered the same as plain text. So sharing this file compromises the secret.

Refer Secret in Pod

After a secret is created, it can be referred to by a pod/rc or deployment in order to pull an image from the private registry. There are many ways to refer to secrets. This section will examine using `imagePullSecrets` under pod spec to refer to the secret.

An `imagePullSecret` is a way to pass a secret that contains a Docker (or other) image registry password to the kubelet so it can pull a private image on behalf of your pod.

Create a pod pulling the Juniper cSRX container from the private repository:

```
apiVersion: v1
kind: Pod
metadata:
  name: csrx-jnpr
  labels:
    app: csrx
  annotations:
   k8s.v1.cni.cncf.io/networks: '[
       { "name": "vn-left-1" },
       { "name": "vn-right-1" }
   ]'
spec:
  containers:
  #- name: csrx
  #  image: csrx
  - name: csrx
    image: hub.juniper.net/security/csrx:18.1R1.9
    ports:
    - containerPort: 22
    #imagePullPolicy: Never
    imagePullPolicy: IfNotPresent
    stdin: true
    tty: true
    securityContext:
      privileged: true
  imagePullSecrets:
  - name: secret-jnpr
```

Now, generate the pod:

```
$ kubectl apply -f csrx/csrx-with-secret.yaml
pod/csrx-jnpr created
```

The cSRX is up and running:

```
$ kubectl get pod
NAME                   READY   STATUS    RESTARTS   AGE
csrx-jnpr              1/1     Running   0          20h
```

And behind the scenes, the pod authenticates itself towards the private registry, pulls the image, and launches the cSRX container:

```
$ kubectl describe pod csrx
......
Events:
19h  Normal  Scheduled  Pod   Successfully assigned ns-user-1/csrx to cent333
19h  Normal  Pulling    Pod   pulling image "hub.juniper.net/security/csrx:18.1R1.9"
19h  Normal  Pulled     Pod   Successfully pulled image "hub.juniper.net/security/csrx:18.1R1.9"
19h  Normal  Created    Pod   Created container
19h  Normal  Started    Pod   Started container
```

As you saw from our test, the secret objects are created independently of the pods, and inspecting the object spec does not provide the sensitive information directly on the screen.

Secrets are not written to the disk, but are instead stored in a tmpfs file system, only on nodes that need them. Also, secrets are deleted when the pod that is dependent on them is deleted.

On most native Kubernetes distributions, communication between users and the API server is protected by SSL/TLS. Therefore, secrets transmitted over these channels are properly protected.

Any given pod does not have access to the secrets used by another pod, which facilitates encapsulation of sensitive data across different pods. Each container in a pod has to request a secret volume in its *volumeMounts* for it to be visible inside the container. This feature can be used to construct security partitions at the pod level.

Service

When a pod gets instantiated, terminated, and moved from one node to another, and in so doing changes its IP address, how do we keep track of that to get uninterrupted functionalities from the pod? Even if the pod isn't moving, how does traffic reach groups of pods via a single entity?

The answer to both questions is Kubernetes *service*.

Service is an abstraction that defines a logical set of pods and a policy, by which you can access them. Think of services as your waiter in a big restaurant – this waiter isn't cooking, instead he's an abstraction of everything happing in the kitchen and you only have to deal with this single waiter.

Service is a Layer 4 load balancer and exposes pod functionalities via a specific IP and port. The service and pods are linked via labels like rs. And there's three different types of services:

- ClusterIP
- NodePort
- LoadBalancer

ClusterIP Service

The clusterIP service is the simplest service, and the default mode if the ServiceType is not specified. Figure 3.4 illustrates how clusterIP service works.

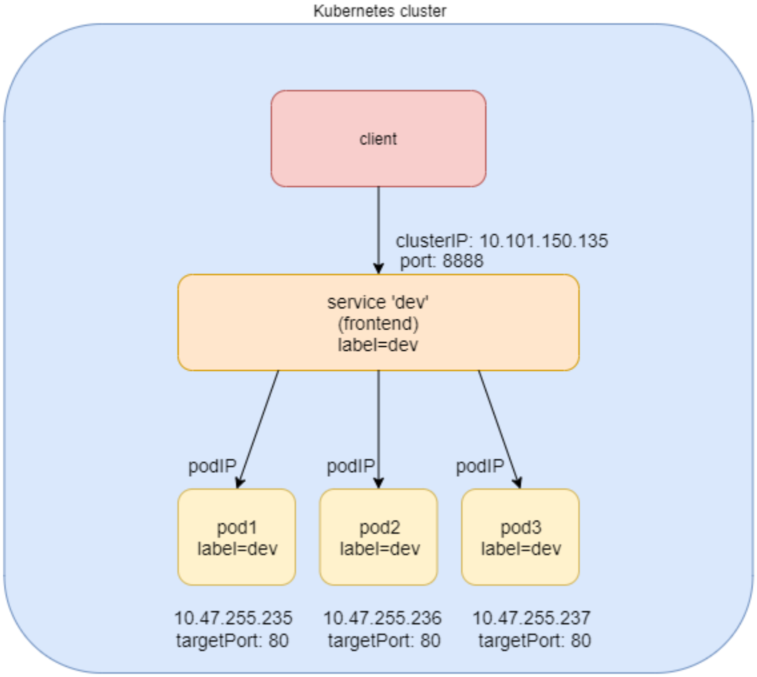


Figure 3.4 ClusterIP Service

You can see that the ClusterIP Service is exposed on a clusterIP and a service port. When client pods need to access the service it sends requests towards this clusterIP and service port. This model works great if all requests are coming from inside of the same cluster. The nature of the clusterIP limits the scope of the service to only be within the cluster. Overall, by default, the clusterIP is not externally reachable .

Create ClusterIP Service

Let's create our first service, with service type clusterIP:

```
#service-web-clusterip.yaml
apiVersion: v1
kind: Service
metadata:
  name: service-web-clusterip
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver
```

The YAML file looks pretty simple and self-explanatory. It defines a `service/service-web-clusterip` with the service port 8888, mapping to targetPort, which means container port 80 in some pod. The selector indicates that whichever pod with a label and app: webserver will be the backend pod responding service request.

Okay, now generate the service:

```
$ kubectl apply -f service-web-clusterip.yaml
service/service-web-clusterip created
```

Use `kubectl` commands to quickly verify the service and backend pod objects:

```
$ kubectl get svc -o wide
NAME                   TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE    SELECTOR
service-web-clusterip  ClusterIP   10.101.150.135   <none>        8888/TCP   9m10s  app=webserver

$ kubectl get pod -o wide -l 'app=webserver'
No resources found.
```

The service is created successfully, but there are no pods for the service. This is because there is no pod with a label matching the selector in the service. So you just need to create a pod with the proper label.

Now, you can define a pod directly but given the benefits of rc and the deployment over pods, as discussed earlier, using rc or deployment is more practical (you'll soon see why).

As an example, let's define a Deployment object named webserver:

```
#deploy-webserver-do.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver
  labels:
    app: webserver
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webserver
    matchExpressions:
      - {key: app, operator: In, values: [webserver]}
  template:
    metadata:
      name: webserver
      labels:
        app: webserver
    spec:
      containers:
      - name: webserver
        image: contrailk8sdayone/contrail-webserver
        securityContext:
           privileged: true
        ports:
        - containerPort: 80
```

The Deployment webserver has a label app: webserver, matching the `selector` defined in our service. The `replicas: 1` instructs the controller to launch only one pod at the moment. Let's see:

```
$ kubectl apply -f deployment-webserver-do.yaml
deployment.extensions/webserver created

$ kubectl get pod -o wide -l 'app=webserver'
NAME                         READY  STATUS   RESTARTS  AGE  IP             NODE     NOMINATED  NODE
webserver-7c7c458cc5-vl6zs   1/1    Running  0         24s  10.47.255.238  cent333  <none>
```

And immediately the pod is chosen to be the backend.

Other brief summaries about the previous `kubectl get svc` command output are:

- the service got a clusterIP, or service IP, of 10.101.150.135 allocated from the service IP pool.
- the service port is 8888 as what is defined in YAML.
- by default, the protocol type is TCP if not declared in the YAML file. You can use protocol: UDP to declare a UDP service.
- the backend pod can be located with the label selector.

TIP The example shown here uses an equality-based selector (-l) to locate the backend pod, but you can also use a set-based syntax to archive the same effect. For example: `kubectl get pod -o wide -l 'app in (webserver)'`.

Verify ClusterIP Service

To verify that the service actually works, let's start another pod as a client to initiate a HTTP request toward the service. For this test, we'll launch and log in to a client pod and use the curl command to send an HTTP request toward the service. You'll see the same pod being used as a client to send requests throughout this book:

```
#pod-client-do.yaml
apiVersion: v1
kind: Pod
metadata:
  name: client
  labels:
    app: client
spec:
  containers:
  - name: contrail-webserver
    image: contrailk8sdayone/contrail-webserver
```

Create the client pod:

```
$ kubectl apply -f pod-client-do.yaml
pod/client created
```

TIP The client pod is just another spawned pod based on the exact same image whatever the webserver Deployment and its pods do. This is the same as with physical servers and VMs: nothing stops a server from doing the client's job:

```
$ kubectl exec -it client -- curl 10.101.150.135:8888
<html>
<style>
  h1   {color:green}
  h2   {color:red}
</style>
  <div align="center">
  <head>
    <title>Contrail Pod</title>
  </head>
  <body>
    <h1>Hello</h1><br><h2>This page is served by a <b>Contrail</b>
    pod</h2><br><h3>IP address = 10.47.255.238<br>Hostname =
    webserver-7c7c458cc5-vl6zs</h3>
    <img src="/static/giphy.gif">
  </body>
  </div>
</html>
```

The HTTP request toward the service reaches a backend pod running the web server application, which responds with a HTML page.

To better demonstrate which pod is providing the service, let's set up a customized pod image that runs a simple web server. The web server is configured in such a way that when receiving a request it will return a simple HTML page with local pod IP and hostname embedded. This way the curl returns something more meaningful in our test.

The returned HTML looks relatively okay to read, but there is a way to make it easier to see, too:

```
$ kubectl exec -it client -- curl 10.101.150.135:8888 | w3m -T text/html | head
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.238
                         Hostname = webserver-7c7c458cc5-vl6zs
```

The w3m tool is a lightweight console-based web browser installed in the host.

With w3m you can render a HTML webpage into text, which is more readable than the HTML page.

Now that service is verified, requests to service have been redirected to the correct backend pod, with a pod IP of 10.47.255.238 and a pod name of webserver-7c7c458cc5-vl6zs.

Specify a ClusterIP

If you want to have a specific clusterIP, you can mention it in the spec. IP addresses should be in the service IP pool.

Here's some sample YAML with specific `clusterIP`:

```
#service-web-clusterip-static.yaml
apiVersion: v1
kind: Service
metadata:
  name: service-web-clusterip-static
spec:
  clusterIP: 10.101.150.150      #<---
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver
```

NodePort Service

The second general type of service, NodePort, exposes a service on each node's IP at a static port. It maps the static port on each node with a port of the application on the pod as shown in Figure 3.5.

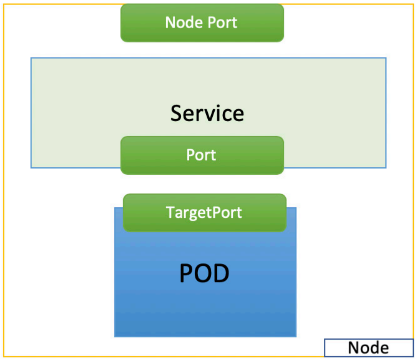


Figure 3.5 NodePort Service

```
#service-web-nodeport
apiVersion: v1
kind: Service
metadata:
  name: service-web-nodeport
spec:
  selector:
    app: webserver
  type: NodePort
  ports:
  - targetPort: 80
    port: 80
    nodePort: 32001 #<--- (optional)
```

Here are some highlights in this services YAML file:

- `selector`: The label selector that determines which set of pods is targeted by this service; here, any pod with the label `app: webserver` will be selected by this service as the backend.
- `port`: This is the service port.
- `targetPort`: The actual port used by the application in the container. Here, it's port 80, as we are planning to run a web server.
- `nodePort`: The port on the host of each node in the cluster.

Let's create the service:

```
$ kubectl create -f service-web-nodeport.yaml
service "service-web-nodeport" created

$ kubectl describe service web-app
Name:                     service-web-nodeport
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=webserver
Type:                     NodePort
IP:                       10.98.108.168
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  32001/TCP
Endpoints:                10.47.255.228:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

- Type: The default service type is `ClusterIP`. In this example, we set the type to `NodePort`.

- `NodePort`: By default, Kubernetes allocates node ports in the 30000-32767 range, if it is not mentioned in the spec. This can be changed using the flag `--service-node-port-range`. The `NodePort` value can also be set, but make sure it's in the configured range.
- `Endpoints`: The podIP and the exposed container port. The request toward service IP and service port will be directed here, and 10.47.255.252:80 indicates that we have created a pod that has a matching label with the service, so its IP is selected as one of the backends.

NOTE For this test, make sure there is at least one pod with the label `app:webserver` running. Pods in previous sections are all created with this label. Recreating the client pod suffices if you've removed them already.

Now we can test this by using the `curl` command to trigger an HTTP request toward any node IP address:

```
$kubectl get pod -o wide

NAME    READY   STATUS    RESTARTS   AGE   IP              NODE     NOMINATED NODE
client  1/1     Running   0          20m   10.47.255.252   cent222  <none>
```

With the power of the `NodePort` service, you can access the web server running in the pod from any node via the nodePort 32001:

```
$ kubectl get node -o wide
NAME      STATUS     ROLES    AGE    VERSION   INTERNAL-IP    ... KERNEL-
VERSION              CONTAINER-RUNTIME
cent111   NotReady   master   100d   v1.12.3   10.85.188.19   ... 3.10.0-957.10.1.el7.
x86_64   docker://18.3.1
cent222   Ready      <none>   100d   v1.12.3   10.85.188.20   ... 3.10.0-957.10.1.el7.
x86_64   docker://18.3.1
cent333   Ready      <none>   100d   v1.12.3   10.85.188.21   ... 3.10.0-957.10.1.el7.
x86_64   docker://18.3.1

$ curl 10.85.188.20:32001

<html>
<style>
  h1   {color:green}
  h2   {color:red}
</style>
  <div align="center">
  <head>
    <title>Contrail Pod</title>
  </head>
  <body>
    <h1>Hello</h1><br><h2>This page is served by a <b>Contrail</b>
    pod</h2><br><h3>IP address = 10.47.255.228<br>Hostname =
    client</h3>
    <img src="/static/giphy.gif">
  </body>
  </div>
</html>
```

Load Balancer Service

The third service, the *load balancer* service, goes one step beyond the NodePort service by exposing the service externally using a cloud provider's load balancer. The load balancer service by its nature automatically includes all the features and functions of NodePort and ClusterIP services.

Kubernetes clusters running on cloud providers support the automatic provision of a load balancer. The only difference between the three services is the type value. To reuse the same NodePort service YAML file, and create a load balancer service, just set the type to LoadBalancer:

```
#service-web-lb.yaml
apiVersion: v1
kind: Service
metadata:
  name: service-web-lb
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver
  type: LoadBalancer    #<---
```

The cloud will see this keyword and a load balancer will be created. Meanwhile, an external public load balancerIP is allocated to serve as the frontend virtual IP. Traffic coming to this loadbalancerIP will be redirected to the service backend pod. Please keep in mind that this redirection process is solely a transport layer operation. The loadbalancerIP and port will be translated to private backend clusterIP and it's targetPort. It does not involve any application layer activities. There is nothing like parsing a URL, proxy HTTP request, and etc., like what happens in the HTTP proxying process. Because the loadbalancerIP is publicly reachable, any Internet host that has access to it (and the service port) can access the service provided by the Kubernetes cluster.

From an Internet host's perspective, when it requests service, it refers this public external loadbalancerIP plus service port, and the request will reach the backend pod. The loadbalancerIP is acting as a gateway between service inside of the cluster and the outside world.

Some cloud providers allow you to specify the loadBalancerIP. In those cases, the load balancer is created with the user-specified `loadBalancerIP`. If the loadBalancerIP field is not specified, the load balancer is set up with an ephemeral IP address. If you specify a loadBalancerIP but your cloud provider does not support the feature, the loadbalancerIP field that you set is ignored.

How a load balancer is implemented in the load balancer service is vendor-specific. A GCE load balancer may work in a totally different way with an AWS load balancer. There is a detailed demonstration of how the load balancer service works in a Contrail Kubernetes environment in Chapter 4.

External IPs

Exposing service outside of the cluster can also be achieved via the `externalIPs` option. Here's an example:

```
#service-web-externalips.yaml
apiVersion: v1
kind: Service
metadata:
  name: service-web-externalips
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver
  externalIPs:          #<---
  - 101.101.101.1       #<---
```

In the `Service` spec, externalIPs can be specified along with any of the service types. External IPs are not managed by Kubernetes and are the responsibility of the cluster administrator.

NOTE External IPs are different from loadbalancerIP, which is the IP assigned by the cluster administrator, while external IPs come with the load balancer created by the cluster that supports it.

Service Implementation: Kube-proxy

By default, Kubernetes uses the kube-proxy module for services, but CNI providers can have their own implementations for services.

Kube-proxy can be deployed in one of the three modes:

- user-space proxy-mode
- iptables proxy-mode
- ipvs proxy-mode

When traffic hits the node, it's forwarded to one of the backend pods via a deployed kube-proxy forwarding plane. Detailed explanations and comparisons of these three modes will not be covered in this book, but you can check Kubernetes official website for more information. Chapter 4 illustrates how Juniper Contrail as a Container Network Interface (CNI) provider implements the service.

Endpoints

There is one object we haven't explored so far: *EP*, or *endpoint*. We've learned that a particular pod or group of pods with matching labels are chosen to be the backend through label selector, so the service request traffic will be redirected to them. The IP and port information of the matching pods are maintained in the endpoint object. The pods may die and spawn any time, the mortal nature of the pod will most likely cause the new pods be respawned with new IP addresses. During this dynamic process the endpoints will always be updated accordingly, to reflect the current backend pod IPs, so the service traffic redirection will act properly. (CNI providers who have their own service implementation update the backend of the service based on the endpoint objects.)

Here is an example to demonstrate some quick steps to verify the service, corresponding endpoint, and the pod, with matching labels.

To create a service:

```
#service-web-clusterip.yaml
apiVersion: v1
kind: Service
metadata:
  name: service-web-clusterip
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver

$kubectl apply -f service-web-clusterip.yaml
service/service-web-clusterip created
```

To list the endpoint:

```
$ kubectl get ep
NAME             ENDPOINTS          AGE
service-web-lb   10.47.255.252:80   5d17h
```

To locate the pod with the label that is used by the selector in service:

```
$ kubectl get pod -o wide -l 'app=webserver'
NAME                          READY  STATUS    RESTARTS  AGE    IP              NODE     ... LABELS
webserver-7c7c458cc5-
```

```
rjlgr  1/1    Running  4         5d17h  10.47.255.252  cent333  ... app=webserver
```

And finally, scale the backend pods:

```
$ kubectl scale deploy webserver --replicas=3
$ kubectl get pod -o wide -l 'app=webserver'
NAME                           READY  STATUS   RESTARTS  AGE    IP             NODE     ... LABELS
rc-webserver-7c7c458cc5-
rjlgr  1/1    Running  4         5d17h  10.47.255.252  cent333  ... app=webserver
rc-webserver-7c7c458cc5-
45skv  1/1    Running  0         5s     10.47.255.251  cent222  ... app=webserver
rc-webserver-7c7c458cc5-
m2cp5  1/1    Running  0         5s     10.47.255.250  cent111  ... app=webserver
```

Now check the endpoints again, and you will see that they are updated accordingly:

```
$ kubectl get ep
NAME             ENDPOINTS                                            AGE
service-web-lb   10.47.255.250:80,10.47.255.251:80,10.47.255.252:80   5d17h
```

Service Without Selector

In the preceding example, the endpoints object is automatically generated by the Kubernetes system whenever a service is created and at least one pod with a matching label exists. But another endpoint use case is a service that has *no* label selector defined in which you can manually map the service to the network address and the port where it's running by manually adding an endpoint object. Then you can connect the endpoint with the service. This can be very useful in some scenarios, for example, in a setup where you have a backend web server running in a physical server, and you still want to integrate it into a Kubernetes service. In that case, you just create the service as usual, and then create an endpoint with an address and port pointing to the web server. That's it! The service does not care about the backend type, it just redirects the service request traffic exactly the same way as if all backend is a pod.

Ingress

Now that you've now seen ways of exposing a service to clients outside the cluster, another method is Ingress. In the service section, service works in transport layer. In reality, you access all services via URLs.

Ingress, or *ing* for short, is another core concept of Kubernetes that allows HTTP/HTTPS routing that does not exist in `service`. Ingress is built on top of service. With ingress, you can define URL-based rules to distribute HTTP/HTTPS routes to multiple different backend services, therefore, ingress exposes services via HTTP/HTTPS routes. After that the requests will be forwarded to each service's corresponding backend pods.

Ingress Versus Service

There are similarities between load balancer service and ingress. Both can expose service outside of the cluster, but there are some significant differences.

Operation Layer

Ingress operates at the application layer of the OSI network model, while service only operates at the transport layer. Ingress understands the HTTP/HTTPS protocol, service only enacts forwarding based on the IP and the port, which means it does not care about the application layer protocol (HTTP/HTTPS) details. Ingress can operate at the transport layer, but service does the same thing, so it doesn't make sense for ingress to do it as well, unless there is a special reason to do so.

Forwarding Mode

Ingress does the application layer proxy pretty much in the same way a traditional web load balancer does. A typical web load balancer proxy sitting between machine A (client) and B (server), works at the application layer. It is aware of the application layer protocols (HTTP/HTTPS) so the client-server interaction does *not* look transparent to the load balancer. Basically it creates two connections each with the source, (A), and the destination, (B), machine. Machine A does not even know about the existence of machine B. For machine A, the proxy is the only thing it talks to and it does not care how and where the proxy gets its data.

Number of Public IPs.

Each service of the ingress needs a public IP if it is exposed directly to the outside of the cluster. When ingress is a frontend to all these services, one public IP is sufficient, which makes life easy for cloud administrators.

Ingress Object

Before going into detail about the ingress object, the best way to get a feel for it is to look at the YAML definition:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-sf
spec:
  rules:
  - host: www.juniper.net
    http:
```

```
paths:
- path: /dev
  backend:
    serviceName: webservice-1
    servicePort: 8888
- path: /qa
  backend:
    serviceName: webservice-2
    servicePort: 8888
```

You can see it looks pretty simple. The spec defines only one item – that is the rules. The rules say a host, which is the Juniper URL here, may have two possible paths in the URL string. The path is whatever follows the host in the URL, in this case they are /dev and /qa. Each path is then associated to a different service. When ingress sees HTTP requests arrive, it proxies the traffic to each URL path associated backend service. Each service, as we've learned in this service section, will deliver the request to its corresponding backend path. That's it. Actually this is one of the three types of ingress that Kubernetes supports today – simple *fan-out* ingress. The other two types of ingress will be discussed later in this chapter.

About URL, Host, and Path

The terms host and path are used frequently in Kubernetes Ingress documentation. The host is a fully qualified domain name of the server. The path, or url-path is the rest of the string part after the host in a URL. If the case is one of having a port in the URL, then it is the string *after* the port.

Take a look at the following URL:

```
http://www.juniper.net:1234/my/resource
       --------------- ---- -----------
       host            port path

http://www.juniper.net/my/resource
       --------------- ------------
       host            path
```

The host is www.juniper.net, whatever follows port 1234 is called path, my/resource in this example. If a URL has no port, then the strings following host are the path. For more details you can read RFC 1738, but for the purpose of this book, understanding what is introduced here will suffice.

If you now think Kubernetes Ingress just defines some rules and the rules are just to instruct the system to direct incoming request to different services, based on the URLs, you are basically right at a high level. Figure 3.6 illustrates the interdependency between the three Kubernetes objects: ingress, service, and pod.

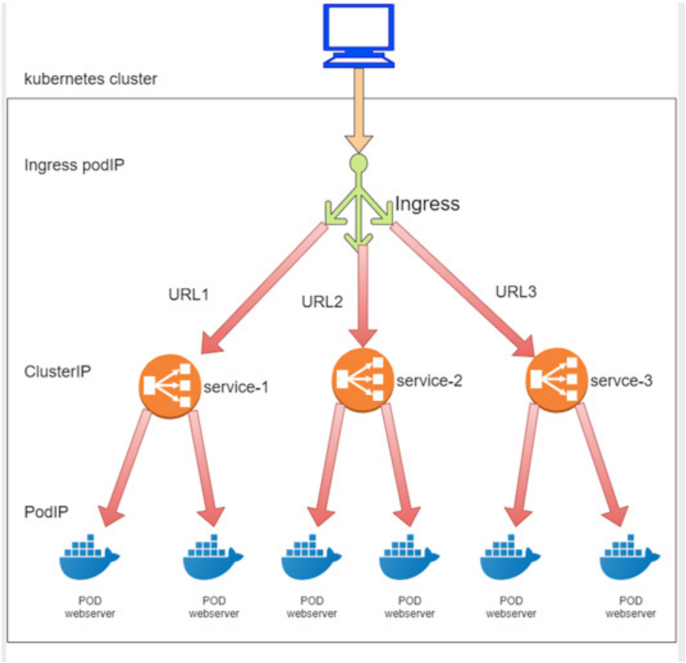


Figure 3.6 *Ingress, Service, and Pod*

In practice there are other things you need to understand, to handle the ingress rules, you need at least one more component called the ingress controller.

Ingress Controller

An ingress controller is responsible for reading the ingress rules and then programming the rules into the proxy, which does the real work – dispatching traffic based on the host / URL.

Ingress controllers are typically implemented by third-party vendors. Different Kubernetes environments have different ingress controllers based on the need of the cluster. Each ingress controller has its own implementations to program the ingress rules. The bottom line is, there has to be an ingress controller running in the cluster.

Some ingress controller providers are:

- nginx
- gce
- haproxy
- avi

- f5
- istio
- contour

You may deploy any number of ingress controllers within a cluster. When you create an ingress, you should annotate each ingress with the appropriate ingress.class to indicate which ingress controller should be used (if more than one exists within your cluster).

The annotation used in ingress objects will be explained in the annotation section.

Ingress Examples

There are three types of ingresses:

- Single Service ingress
- Simple Fanout ingress
- Name-based Virtual Hosting ingress

We've looked at the simple fanout ingress, so now let's look at a YAML file example for the other two types of ingress.

Single Service Ingress

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-single-service
spec:
  backend:
    serviceName: webservice
    servicePort: 80
```

This is the simplest form of ingress. The ingress will get an external IP so the service can be exposed to the public, however, it has no rules defined, so it does not parse host or path in the URLs. All requests go to the same service.

Simple Fanout Ingress

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-sf
spec:
  rules:
  - host: www.juniper.net
    http:
      paths:
      - path: /dev
```

```
          backend:
            serviceName: webservice-1
            servicePort: 8888
        - path: /qa
          backend:
            serviceName: webservice-2
            servicePort: 8888
```

We checked this out at the beginning of this section. Compared to single service ingress, simple fanout ingress is more practical. It's not only able to expose service via a public IP, but it is also able to do URL routing or fan out based on the path. This is a very common usage scenario when a company wants to direct traffic to each of its department's dedicated servers based on the suffix of URL after the domain name.

Virtual Host Ingress

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-virutal-host
spec:
  rules:
  - host: www.juniperhr.com
    http:
      paths:
      - backend:
          serviceName: webservice-1
          servicePort: 80
  - host: www.junipersales.com
    http:
      paths:
      - backend:
          serviceName: webservice-2
          servicePort: 80
```

The name-based virtual host is similar to simple fanout ingress in that it is able to do rule-based URL routing. The unique power of this type of ingress is that it supports routing HTTP traffic to multiple host names at the same IP address. The example here may not be practical (unless one day the two domains merge!) but it is good enough to showcase the idea. In the YAML file two hosts are defined, the "*juniperhr*" and "*junipersales*" URL respectively. Even though ingress will be allocated with only one public IP, based on the host in the URL, requests toward that same public IP will still be routed to different backend services. That's why it is called a *virtual hosting* ingress and there's a very detailed case study in Chapter 4 for you to explore.

NOTE It is also possible to merge a simple fanout ingress and a virtual host ingress into one, but the details are not covered here.

Multiple Ingress Controller

You can have multiple ingress controllers in one cluster but the cluster needs to know which one to choose. For example, in Chapter 4 we'll talk about Contrail's built-in ingress controller, which does not stop us from installing another third-party ingress controller like the nginx ingress controller. Instead you end up having two ingress controllers in the same cluster with the names:

- opencontrail (default)
- nginx

Contrail's implementation is the default one, so you don't have to specifically select it. To select nginx as ingress controller, use this annotation. Kubernetes.io/ingress.class:

```
metadata:
  name: foo
  annotations:
    Kubernetes.io/ingress.class: "nginx"
```

This will tell Contrail's ingress controller opencontrail to ignore the ingress configuration.

Kubernetes Network Policy

The Kubernetes networking model requires all pods to be able to access all other pods by default. This is called a *flat network* because it follows an *allow-any-any* model. It significantly simplifies the design and implementation of Kubernetes networking and makes it much more scalable.

NOTE Chapter 4 details the requirements that Kubernetes enforces on network implementations.

Security is an important concern. In reality, in many cases a certain level of network segmentation methods is required to ensure that only certain pods can talk to each other, and that is when Kubernetes network policy comes into the picture. A Kubernetes network policy defines the access permissions for groups of pods the same way a security group in the cloud is used to control access to VM instances.

Kubernetes supports network policy via the NetworkPolicy object, which is a Kubernetes resource just like pod, service, ingress, and many others you've learned about earlier in this chapter. The role of the NetworkPolicy object is to define how groups of pods are allowed to communicate with each other.

Let's examine how Kubernetes network policy works:

1. Initially, in a Kubernetes cluster, all pods are non-isolated by default and they work in an allow-any-any model so any pod can talk to any other pod.

2. Now apply a network policy named policy1 to pod A. In policy policy1 you define a rule to explicitly allow pod A to talk to pod B. In this case let's call pod A a *target* pod because it is the pod that the network policy will act on.

3. From this moment on, a few things happen:

- Target pod A can talk to pod B, and can talk to *pod B only*, because B is the only pod you allowed in the policy. Due to the nature of the policy rules, you can call the rule a *whitelist*.

- For target pod A only, any connections that are not explicitly allowed by the whitelist of this network policy policy1 will be rejected. You don't need to explicitly define this in policy1, because it will be enforced by the nature of Kubernetes network policy. Let's call this *implicit* policy the deny all policy.

- As for other non-targeted pods, for example, pod B or pod C, which are *not* applied with policy1, nor to any other network policies, will continue to follow the allow-any-any model. Therefore they are not affected and can continue to communicate to all other pods in the cluster. This is another implicit policy, an allow all policy.

4. Assuming you also want pod A to be able to communicate to pod C, you need to update the network policy policy1 and its rules to *explicitly* allow it. In other words, you need to keep updating the whitelist to allow more traffic types.

As you can see, when you define a policy, at least three policies will be applied in the cluster:

- Explicit policy1: This is the network policy you defined, with the whitelist rules allowing certain types of traffic for the selected (target) pod.

- An implicit deny all network policy: This denies all other traffic that is not in the whitelist of the target pod.

- An implicit allow all network policy: This allows all other traffic for the other non-targeted pods that are not selected by policy1. We'll see *deny all* and *allow all* policies again in Chapter 8.

Here are some highlights of the Kubernetes network policy.

- Pod specific: Network policy specification applies to one pod or a group of pods based on label, same way as rc or Deploy do.

- Whitelist-based rules: explicit rules that compose a whitelist, and each rule describes a certain type of traffic to be allowed. All other traffic not described by any rules in the whitelist will be dropped for the target pod.

- Implicit allow all: A pod will be affected only if it is selected as the target by a network policy, and it will be affected only by the selecting network policy. The absence of a network policy applied on a pod indicates an implicit allow all policy to this pod. In other words, if a non-targeted pod continues its allow-any-any networking model.
- Separation of ingress and egress: Policy rules need to be defined for a specific direction. The direction can be Ingress, Egress, none, or both.
- Flow-based (vs. packet-based): Once the initiating packet is allowed, the return packet in the same flow will also be allowed. For example, suppose an ingress policy applied on pod A allows an ingress HTTP request, then the whole HTTP interaction will be allowed for pod A. This includes the three-way TCP connection establishment and all data and acknowledgments in both directions.

NOTE Network policies are implemented by the network component, so you must be using a network solution that supports network policy. Simply creating the NetworkPolicy resource without a controller to implement it will have no effect. In this book Contrail is such a network component with network policy implemented. In Chapter 8, you'll see how these network policies work in Contrail.

Network Policy Definition

Like all other objects in Kubernetes, network policy can be defined in a YAML file. Let's look at an example (the same example will be used in Chapter 8):

```
#policy1-do.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: policy1
  namespace: dev
spec:
  podSelector:
    matchLabels:
      app: webserver-dev
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 10.169.25.20/32
    - namespaceSelector:
        matchLabels:
          project: jtac
    - podSelector:
        matchLabels:
          app: client1-dev
    ports:
    - protocol: TCP
      port: 80
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: dbserver-dev
    ports:
    - protocol: TCP
      port: 80
```

Let's look at the `spec` part of this YAML file since the other sections are somewhat self-explanatory. The spec has the following structure:

```
spec:
  podSelector:
    ......
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    ......
  egress:
  - to:
    ......
```

Here you can see that a network policy definition YAML file can logically be divided into four sections:

- podSelector: This defines the pods selection. It identifies the pods to which the current network policy will be applied.
- policyTypes: Specifies the type of policy rules: Ingress, Egress or both.
- ingress: Defines the ingress policy rules for the target pods.
- egress: Defines the egress policy rules for the target pods.

Next we'll look at each section in more detail.

Selecting Target Pods

When you define a network policy, Kubernetes needs to know which pods you want this policy to act on. Similar to how service selects its backend pods, the network policy selects pods to which it will be applied based on labels:

```
podSelector:
  matchLabels:
    app: webserver-dev
```

Here, all pods that have the label `app: webserver-dev` are selected to be the target pods by the network policy. All of the following content in `spec` will apply to only the target pods.

Policy Types

The second section defines the `policyTypes` for the target pods:

```
policyTypes:
  - Ingress
  - Egress
```

`PolicyTypes` can either be ingress, egress, or both. And both types define specific traffic types in the form of one or more rules, as discussed next.

Policy Rules

The ingress and egress sections define the direction of traffic, from the selected target pods' perspective. For example, consider the following simplified example:

```
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: client1-dev
    ports:
    - protocol: TCP
      port: 80
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: client1-dev
    ports:
    - protocol: TCP
      port: 8080
```

Assuming the target pod is webserver-dev pod, and there is only one pod client1-dev in the cluster having a matching label client1-dev, two things will happen:

1. The ingress direction: the pod webserver-dev can accept a TCP session with a destination port 80, initiated from pod client1-dev. This explains why we said Kubernetes network policy is flow-based instead of packet-based. The TCP connection could not be established if the policy would have been packet-based designed because on receiving the incoming TCP sync, the returning outgoing TCP sync-ack would have been rejected without a matching egress policy.
2. The egress direction: pod webserver-dev can initiate a TCP session with a destination port 8080, towards pod client1-dev.

TIP For the egress connection to go through, the other end needs to define an ingress policy to allow the incoming connection.

Network Policy Rules

Each from or to statement defines a rule in the network policy:

- A from statement defines an ingress policy rule.
- A to statement defines an egress policy rule.
- Both rules can optionally have ports statements, which will be discussed later.

So you can define multiple rules to allow complex traffic modes for each direction:

```
ingress:
INGRESS RULE1
INGRESS RULE2
egress:
EGRESS RULE1
EGRESS RULE2
```

Each rule identifies the network endpoints where the target pods can communicate. Network endpoints can be identified by different methods:

- ipBlock: Selects pods based on an IP address block.
- namespaceSelector: Selects pods based on the label of the namespace.
- podSelector: Selects pods based on label of the pod.

NOTE The podSelector selects different things when it is used in different places of a YAML file. Previously (under spec) it selected pods that the network policy applies to, which we've called *target pods*. Here, in a rule (under from or to), it selects which pods the target pod is communicating with. Sometimes we call these pods *peering pods*, or *endpoints*.

So the YAML structure for a rule can look like this:

```
ingress:
- from:
  - ipBlock:
    .....
  - namespaceSelector:
    .....
  - podSelector:
    .....
  ports:
    ......
```

For example:

```
ingress:
- from:
  - ipBlock:
      cidr: 10.169.25.20/32
  - namespaceSelector:
      matchLabels:
        project: jtac
  - podSelector:
      matchLabels:
        app: client1-dev
  ports:
  - protocol: TCP
    port: 80
egress:
- to:
  - podSelector:
      matchLabels:
        app: dbserver-dev
  ports:
  - protocol: TCP
    port: 80
```

Here, the ingress network endpoints are subnet 10.169.25.20/32; *or* all pods in namespaces that have the label project: jtac; *or* pods which have the label app: client1-dev in current namespace (namespace of target pod), and the egress network point is pod dbserver-dev. We'll come to the ports part soon.

AND versus OR

It's also possible to specify only a few pods from namespaces, instead of communicating with all pods. In our example, podSelector is used all along, which assumes the same namespace as the target pod. Another method is to use podSelector along with a namespaceSelector. In that case, the namespaces that the pods belong to are those with matching labels with namespaceSelector, instead of the same as the target pod's namespace.

For example, assuming that the target pod is webserver-dev and its namespace is dev, and only namespace qa has a label `project=qa` matching to the namespaceSelector:

```
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        project: qa
    podSelector:
      matchLabels:
        app: client1-qa
```

Here, the target pod can only communicate with those pods that are in namespace `qa`, AND (not OR) with the label `app: client1-qa`.

Be careful here because it is totally different than the definition below, which allows the target pod to talk to those pods that are: in namespaces `qa`, OR (not AND) with label `app: client1-qa` in the target pod's namespace dev:

```
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        project: qa
  - podSelector:
      matchLabels:
        app: client1-qa
```

Protocol and Ports

It is also possible to specify ports for an ingress and egress rule. The protocol type can also be specified along with a protocol port. For example:

```
egress:
- to:
  - podSelector:
      matchLabels:
        app: dbserver-dev
  ports:
  - protocol: TCP
    port: 80
```

The ports in ingress say that the target pods can allow incoming traffic for the specified ports and protocol. Ports in egress say that target pods can initiate traffic to specified ports and protocol. If port are not mentioned, all ports and protocols are allowed.

Line-By-Line Explanation

Let's look at our example again in detail:

```
  podSelector:
    matchLabels:
      app: webserver-dev
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 10.169.25.20/32
    - namespaceSelector:
        matchLabels:
          project: jtac
    - podSelector:
        matchLabels:
```

```
        app: client1-dev
  ports:
  - protocol: TCP
    port: 80
egress:
- to:
  - podSelector:
      matchLabels:
        app: dbserver-dev
  ports:
  - protocol: TCP
    port: 80
```

You should now know exactly what the network policy is trying to enforce.

Lines 1-3: pod webserver-dev is selected by the policy, so it is the target pod; all following policy rules will apply on it, and on it alone.

Lines 4-6: the policy will define rules for both Ingress and Egress traffic.

Lines 7-19: ingress: section defines the ingress policy.

Line 8: from: and line 17: ports, these two sections define one policy rule on ingress policy.

Lines 9-16: these eight lines under the from: section compose an ingress whitelist:

- Lines 9-10: any incoming data with source IP being 10.169.25.20/32 can access the target pod webserver-dev.
- Lines 11-13: any pods under namespace jtac can access target pod webserver-dev.
- Lines 14-16: any pods with label client1-dev can access target pod webserver-dev.

Lines 17-19: ports section is second (and optional) part of the same policy rule. Only TCP port 80 (web service) on target pod webserver-dev is exposed and accessible. Access to all other ports will be denied.

Lines 20-26: egress: section defines the egress policy.

Lines 21: to: and line 24: ports, these two sections define one policy rule in egress policy.

- Lines 21-24: these four lines under to: section compose an egress whitelist, here the target pod can send egress traffic to pod dbserver-dev.

Line 25: ports section is second part of the same policy rule. The target pod webserver-pod can only start TCP session with a destination port of 80 to other pods.

And that's not all. If you remember at the beginning of this chapter, we talked about the Kubernetes default *allow-any-any* network model and the implicit *deny-all*, *allow-all* policies, you will realize that so far we just explained the explicit part of it (policy1 in our network policy introduction section). After that, there are two more implicit policies:

The deny all network policy: for the target pod webserver-dev, deny all other traffic that is other than what is explicitly allowed in the above whitelists, this implies at least two rules:

- ingress: deny all incoming traffic destined to the target pod webserver-dev, other than what is defined in the ingress whitelist.
- egress: deny all outgoing traffic sourcing from the target pod webserver-dev, other than what is defined in the egress whitelist.

An allow all network policy allows all traffic for other pods that are not target of this network policy, on both ingress and egress direction.

NOTE In Chapter 8 we'll take a more in depth look at these implicit network policies and their rules in Contrail implementation.

Create Network Policy

You can create and verify the network policy the same way that you create other Kubernetes objects:

```
$ kubectl apply -f policy1-do.yaml
networkpolicy.networking.k8s.io/policy1-do created

$ kubectl get netpol -n dev
NAME           POD-SELECTOR       AGE
policy1        app=webserver-dev   6s

$ kubectl describe netpol policy -n dev
Name:         policy1
Namespace:    dev
Created on:   2019-10-01 11:18:19 -0400 EDT
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     app=webserver-dev
  Allowing ingress traffic:
    To Port: 80/TCP
    From:
      IPBlock:
        CIDR: 10.169.25.20/32
        Except:
    From:
      NamespaceSelector: project=jtac
    From:
      PodSelector: app=client1-dev
  Allowing egress traffic:
    To Port: 80/TCP
    To:
      PodSelector: app=dbserver-dev
  Policy Types: Ingress, Egress
```

In Chapter 8 we'll set up a test environment to verify the effect of this network policy in more detail.

Liveness Probe

What happens if the application in the pod is running but it can't serve its main purpose, for whatever reason? Also applications that run for a long time might transition to broken states, and if this is the case the last thing you want is a call reporting a problem in an application that could be easily fixed with restarting the pod. Liveness probes are a Kubernetes feature made specifically for this kind of situation. Liveness probes send a pre-defined request to the pod on a regular basis then restart the pod if the request fails. The most commonly used liveness probe is HTTP GET request, but it can also open the TCP socket or even issue a command.

Next is an HTTP GET request probe example, where the `initialDelaySeconds` is the waiting time before the first try to HTTP GET request to port 80, then it will run the probe every 20 seconds as specified in `periodSeconds`. If this fails the pod will restart automatically. You have the option to specify the path, which here is just the main website. Also you can send the probe with a customized header. Take a quick look:

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-pod
  labels:
    app: tcpsocket-test
spec:
  containers:
    - name: liveness-pod
      image: contrailk8sdayone/ubuntu
      ports:
       - containerPort: 80
      securityContext:
          privileged: true
          capabilities:
           add:
             - NET_ADMIN
      livenessProbe:
        httpGet:
           path: /
           port: 80
           httpHeaders:
           - name: some-header
```

```
             value: Running
        initialDelaySeconds: 15
        periodSeconds: 20
```

Now let's launch this pod then log in to it to terminate the process that handles the HTTP GET request:

```
[root@cent11 ~]# kubectl get pod
NAME           READY   STATUS    RESTARTS   AGE
liveness-pod   1/1     Running   0          114s


[root@cent11 ~]# kubectl exec -it liveness-pod bash
root@liveness-pod:/# sudo netstat -tulpn

Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      111/apache2
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      45/sshd
tcp6       0      0 :::22                   :::*                    LISTEN      45/sshd

root@liveness-pod:/# service apache2 stop
 * Stopping web server apache2                    *

root@liveness-pod:/# sudo netstat -tulpn
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      45/sshd
tcp6       0      0 :::22                   :::*                    LISTEN      45/sshd

[root@cent11 ~]# kubectl get pod
NAME           READY   STATUS    RESTARTS   AGE
liveness-pod   1/1     Running   1          5m33s
```

You can see that the pod was automatically restarted, and you can also see the reason for that restart in the event:

```
Killing container with id docker://liveness-
pod:Container failed liveness probe. Container will be killed and recreated.
[root@cent11 ~]# kubectl describe pod liveness-pod
Name:               liveness-pod
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               cent22/10.85.188.17
Start Time:         Fri, 05 Jul 2019 16:39:12 -0400
Labels:             app=tcpsocket-test
Annotations:        k8s.v1.cni.cncf.io/network-status:
                      [
                          {
                              "ips": "10.47.255.249",
                              "mac": "02:c2:59:4a:82:9f",
                              "name": "cluster-wide-default"
                          }
                      ]
Status:             Running
IP:                 10.47.255.249
Containers:
  liveness-pod:
    Container ID:   docker://01969f51d32f38a15baab18487b85c54cee4125f55c8c7667236722084e4df06
    Image:          virtualhops/ato-ubuntu:latest
    Image ID:       docker-pullable://virtualhops/ato-ubuntu@sha256:fa2930cb8f4b766e5b335dfa42de510e
cd30af6433ceada14cdaae8de9065d2a
    Port:           80/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Fri, 05 Jul 2019 16:41:35 -0400
    Last State:     Terminated
      Reason:       Error
      Exit Code:    137
      Started:      Fri, 05 Jul 2019 16:39:20 -0400
      Finished:     Fri, 05 Jul 2019 16:41:34 -0400
    Ready:          True
    Restart Count:  1
    Liveness:       http-get http://:80/ delay=15s timeout=1s period=20s #success=1 #failure=3
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-m75c5 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-m75c5:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-m75c5
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type     Reason     Age                    From               Message
  ----     ------     ----                   ----               -------
 Normal   Scheduled  7m19s               default-scheduler  Successfully assigned default/liveness-pod
to cent22
 Warning  Unhealthy  4m6s (x3 over 4m46s)   kubelet, cent22   Liveness probe failed: Get
http://10.47.255.249:80/: dial tcp 10.47.255.249:80: connect: connection refused
 Normal   Pulling   3m36s (x2 over 5m53s)  kubelet, cent22   pulling image "virtualhops/ato-
ubuntu:latest"
 Normal   Killing   3m36s               kubelet, cent22   Killing container with id docker://
liveness-pod:Container failed liveness probe.. Container will be killed and recreated.
 Normal   Pulled    3m35s (x2 over 5m50s)  kubelet, cent22   Successfully pulled image "virtualhops/
ato-ubuntu:latest"
 Normal   Created   3m35s (x2 over 5m50s)  kubelet, cent22   Created container
 Normal   Started   3m35s (x2 over 5m50s)  kubelet, cent22   Started container
```

This is a TCP socket probe example. A TCP socket probe is similar to the HTTP GET request probes, but it will open the TCP socket:

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-pod
  labels:
    app: tcpsocket-test
spec:
  containers:
    - name: liveness-pod
      image: contrailk8sdayone/ubuntu
      ports:
        - containerPort: 80
      securityContext:
          privileged: true
          capabilities:
           add:
             - NET_ADMIN
      livenessProbe:
        tcpSocket:
          port: 80
       initialDelaySeconds: 15
       periodSeconds: 20
```

The command is like HTTP GET and TCP socket probes. But the probe will execute the command in the container:

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-pod
  labels:
    app: command-test
spec:
  containers:
    - name: liveness-pod
      image: k8s.gcr.io/busybox
      args:
      - /bin/sh
      - -c
      - touch /tmp/healthy; while true; do sleep 600;done;
      livenessProbe:
        exec:
          command:
          - cat
          - /tmp/healthy
        initialDelaySeconds: 5
        periodSeconds: 5
```

Readiness Probe

A liveness probe makes sure that your pod is in good health, but for some applications it isn't enough. Some applications need to load large files before starting. You might think if you set a higher `initialDelaySeconds` value then the problem is solved but this is not an efficient solution. The readiness probe is a solution especially for Kubernetes services, as the pod will not receive the traffic until it is ready. Whenever the readiness probe fails, the endpoint for the pod is removed from the service and it will be added back when the readiness probe succeeds. The readiness probe is configured in the same way as the liveness probe:

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-readiness
  labels:
    app: tcpsocket-test
spec:
  containers:
    - name: liveness-readiness-pod
      image: virtualhops/ato-ubuntu:latest
      ports:
       - containerPort: 80
      securityContext:
          privileged: true
          capabilities:
           add:
             - NET_ADMIN
      livenessProbe:
        httpGet:
           path: /
           port: 80
           httpHeaders:
           - name: some-header
             value: Running
        initialDelaySeconds: 15
        periodSeconds: 20
      readinessProbe:
        tcpSocket:
          port: 80
        initialDelaySeconds: 5
        periodSeconds: 10
```

NOTE It's recommended to use both the readiness probe and the liveness probe whereby the liveness probe restarts the pod if it failed and the readiness probe makes sure the pod is ready before it gets traffic.

Probe Parameters

Probes have a number of parameters that you can use to more precisely control the behavior of liveness and readiness checks.

- initialDelaySeconds: Number of seconds after the container has started before liveness or readiness probes are initiated.
- periodSeconds: How often (in seconds) to perform the probe. Default is 10 seconds. Minimum value is 1.
- timeoutSeconds: Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.
- successThreshold: Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness. Minimum value is 1.
- failureThreshold: When a pod starts and the probe fails, Kubernetes will try failureThreshold times before giving up. Giving up in case of a liveness probe means restarting the pod. In case of a readiness probe the pod will be marked Unready. Defaults to 3. Minimum value is 1.

And HTTP probes have additional parameters that can be set on `httpGet`:

- `host`: The host name to connect to, which defaults to the pod IP. You probably want to set "Host" in `httpHeaders` instead.
- `scheme`: The scheme to use for connecting to the host (HTTP or HTTPS). Defaults to HTTP.
- `path`: Path to access on the HTTP server.
- `httpHeaders`: Custom headers to set in the request. HTTP allows repeated headers.
- `port`: Name or number of the port to access on the container. Number must be in the range 1 to 65535.

Annotation

You have already seen how labels in Kubernetes are used for identifying, selecting, and organizing objects. But labels are just one way to attach metadata to Kubernetes objects.

Another way is *annotations,* which is a key/value map that attaches non-identifying metadata to objects. Annotation has a lot of use cases, such as attaching:

- pointers for logging and analytics
- phone numbers, directory entries, and web sites
- timestamps, image hashes, and registry addresses
- network, namespaces
- and, types of ingress controller.

Here's an example for annotations:

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-demo
  annotations:   #<---
    imageregistry: https://hub.docker.com
spec:
  containers:
    - name: annotation-pod
      image: contrailk8sdayone/ubuntu
      ports:
       - containerPort: 80
```

Annotations can be used to assign network information to pods, and in Chapter 9, you'll see how a Kubernetes annotation can instruct Juniper Contrail to attach an interface to a certain network. Cool.

Before seeing annotations in action, let's first create a network with a minimum configuration based on the de facto Kubernetes network custom resource definition. `NetworkAttachmentDefinition` is used here to indicate the CNI as well as the parameters of the network to which we will attach to the interface pod:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: net-a
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "awesome-plugin"
  }'
```

The type, awesome-plugin, is the name of the CNI which could be Flannel, Calico, Contrail-K8s-cni, etc.

Create a pod and use annotations to attach its interface to a network called net-a:

```
kind: Pod
metadata:
  name: my-pod
  namespace: my-namespace
  annotations:
    k8s.v1.cni.cncf.io/networks: net-a
```

NOTE According to the official Kubernetes network custom resource definition, the annotation k8s.v1.cni.cncf.io/networks is used to represent NetworkAttachment-Definition and has two formats:

```
Network
   k8s.v1.cni.cncf.io/networks: net-a
Namespace/network name
   k8s.v1.cni.cncf.io/networks: my-namespace/net-a
```

NOTE To maintain compatibility with existing Kubernetes deployments, all pods must attached to the cluster-wide default network, which means even if you have attached one pod interface to a specific network, this pod would have two interfaces: one attached to the cluster-wide default network, and the other attached to the network specified in the annotation argument (net-a in this case).

Chapter 4

Kubernetes and Contrail Integration

This chapter takes a deep dive into Contrail's role in Kubernetes. It starts with a section about Contrail Kubernetes integration architecture, where you will learn how Kubernetes objects such as NS, pod, service, ingress, network policy, and more are handled in Contrail. Then it looks into the implementation of each of the objects in detail.

Whenever needed the chapter will introduce Contrail objects. As a Kubernetes network, CNI multiple interface pods are one of Contrail's advantages over other implementations, so this chapter details such advantages.

The chapter concludes with a demonstration of service chaining using Juniper's cSRX container. Let's get started with the integration architecture.

Contrail-Kubernetes Architecture

After witnessing the main concepts of Kubernetes in Chapters 2 and 3, what could be the benefit of adding Contrail to standard Kubernetes deployment?

In brief, and please refer to the Contrail product pages on www.juniper.net for the latest offerings, Contrail offers common deployment for multiple environments (OpenStack, Kubernetes, etc.) and enriches Kubernetes' networking and security capabilities.

When it comes to deployment for multiple environments, yes, *containers* are the current trend in building applications (not to mention the nested approach, where containers are hosted in VM). But don't expect everyone to migrate from VMs to containers that fast. Add a workload, fully or partially run in the public cloud, and you can see the misery for network and security administrators where Kubernetes becomes just one more thing to manage.

Administrators in many organizations manage individual orchestrators/managers for each environment. OpenStack or VMware NSX for VM, Kubernetes or Mesos for Containers, AWS console. WContrail alleviates this misery by providing dynamic end-to-end networking policy and control for any cloud, any workload, any deployment.

From a single user interface Contrail translates abstract workflows into specific policies and simplifies the orchestration of virtual overlay connectivity across all environments. It does this by building and securing virtual networks connecting BMS, VM, and containers located in a private or public cloud.

You can deploy Kubernetes to launch its pod in VMs orchestrated by OpenStack and others, but this chapter focuses only on Kubernetes. Many features discussed here can be extended for other environments, but Contrail simply enriches standard Kubernetes deployment.

Even though Kubernetes does not provide the networking, it imposes the fundamental requirements of the network implementation and that is taken care of by all CNI (Container Network Interface) providers. Juniper Contrail is one of the CNI providers. Refer to: https://kubernetes.io/docs/concepts/cluster-administration/networking/ for more information.

Kubernetes has some well-defined requirements for its networking implementation:

- pods on a node can communicate with all pods on all nodes without NAT,
- agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node, and
- pods in the host network of a node can communicate with all pods on all nodes without NAT.

Kubernetes offers flat network connectivity with some security features confined in a cluster, but on top of that, Contrail can offer:

- namespaces and services customized isolations for segmentations and multi-tenancy,
- distributed load balancing and firewall with extensive centralized flow and logs insight,
- rich security policy using tags that can extend to other environments (OpenStack, VMWare, BMS, AWS, etc.), and
- service chaining.

This chapter covers some of these features, but first let's talk about Contrail architecture and object mapping.

Contrail-Kube-Manager

A new module of Contrail has been added called `contrail-kube-manager`, abbreviated as `KM`. It watches the Kubernetes API server for interested Kubernetes resources, and translates them into a Contrail controller object. Figure 4.1 illustrates the basic workflow.

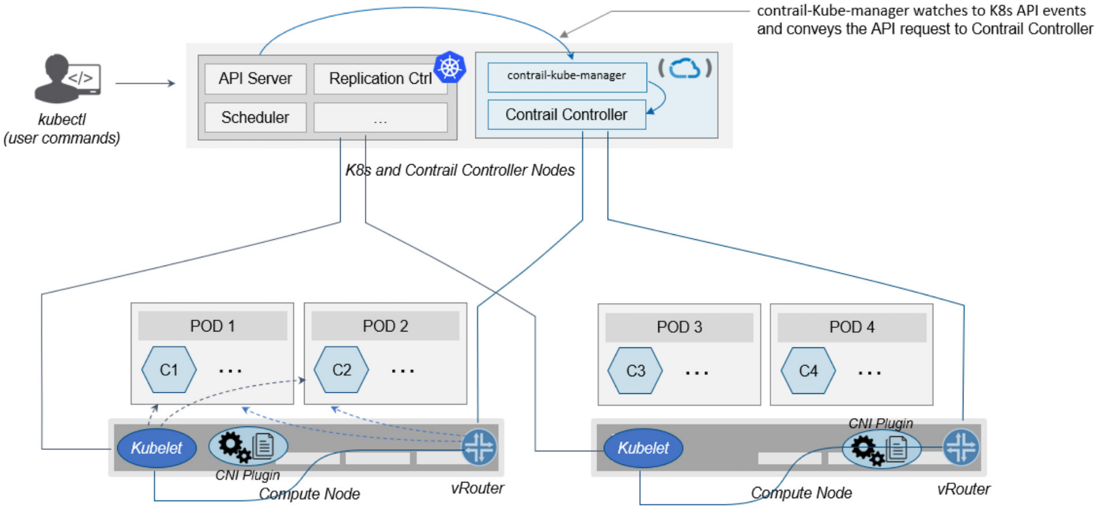


Figure 4.1 *Contrail Kubernetes Architecture*

Kubernetes to Contrail Object Mapping

This is not much of a change from the regular Contrail that we know and love, but there's a lot happening behind the scenes. Be aware that dealing with Kubernetes/ Contrail is all about object mapping. That's because Contrail is a single interface managing multiple environments, and each environment has its own acronyms and terms and hence the need for this mapping, which is done by a plugin. In Kubernetes, `contrail-kube-manager` does this.

NOTE Juniper Contrail has specific plugins for each environment/orchestrator. To learn more and get up-to-date release information, go to Juniper Contrail at https://www.juniper.net/us/en/products-services/sdn/contrail/.

For example, namespaces in Kubernetes are intended for segmentation between multiple teams, or projects, as if creating virtual clusters. In Contrail the similar concept is named *project* so when you create a namespace in Kubernetes it will automatically create an equivalent project in Contrail. More on that will come later, but for now familiarizing yourself with the list of objects shown in Figure 4.2 will help you understand the architecture.

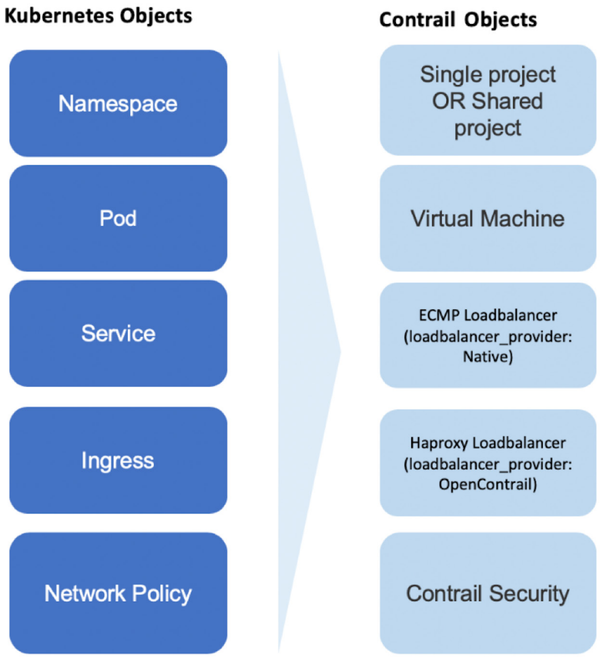

Kubernetes Objects
Contrail Objects
Namespace
Pod
Service
Ingress
Network Policy
Single project OR Shared project
Virtual Machine
ECMP Loadbalancer (loadbalancer_provider: Native)
Haproxy Loadbalancer (loadbalancer_provider: OpenContrail)
Contrail Security

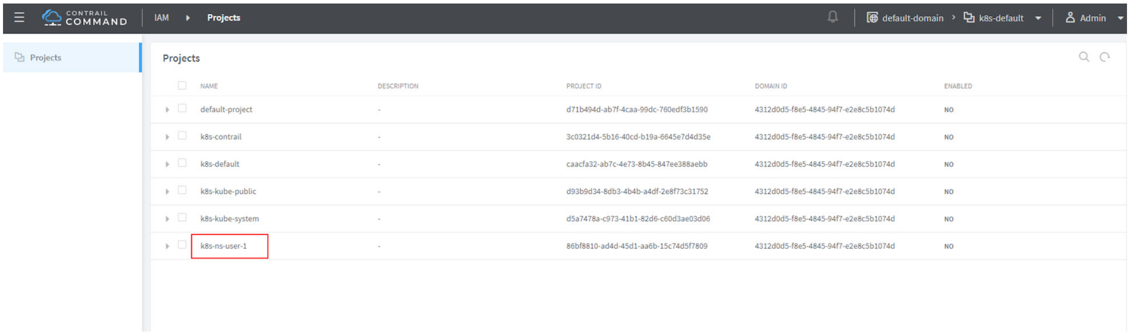
Figure 4.2 *Contrail Kubernetes Object Mapping*

Contrail Namespaces and Isolation

In Chapter 3 you read about `namespace` or `NS` in Kubernetes, and at the beginning of this chapter we mentioned object mappings between Kubernetes and Contrail. In this section you'll see how namespace works in Contrail environments and how Contrail extends the feature set even further.

One analogy given when introducing the `namespace` concept is OpenStack `project`, or `tenant`. And that is exactly how Contrail is looking at it. Whenever a new `namespace` object is created, `contrail-kube-manager` (KM) gets a notice about the object creation event and it will create the corresponding `project` in Contrail.

To differentiate between multiple Kubernetes clusters in Contrail, a Kubernetes cluster name will be added to the Kubernetes namespace or project name. The default Kubernetes cluster name is `k8s`. So if you create a Kubernetes namespace `ns-user-1`, a `k8s-ns-user-1` project will be created in Contrail just as you can see here in Figure 4.3, which shows the Contrail GUI.



Projects

NAME	DESCRIPTION	PROJECT ID	DOMAIN ID	ENABLED
default-project	-	d71b494d-ab7f-4caa-99dc-760edf3b1590	4312d0d5-f8e5-4845-94f7-e2e8c5b1074d	NO
k8s-contrail	-	3c0321d4-5b16-40cd-b19a-6645e7d4d35e	4312d0d5-f8e5-4845-94f7-e2e8c5b1074d	NO
k8s-default	-	caacfa32-ab7c-4e73-8b45-847ee388aebb	4312d0d5-f8e5-4845-94f7-e2e8c5b1074d	NO
k8s-kube-public	-	d93b9d34-8db3-4b4b-a4df-2e8f73c31752	4312d0d5-f8e5-4845-94f7-e2e8c5b1074d	NO
k8s-kube-system	-	d5a7478a-c973-41b1-82d6-c60d3ae03d06	4312d0d5-f8e5-4845-94f7-e2e8c5b1074d	NO
k8s-ns-user-1	-	86bf8810-ad4d-45d1-aa6b-15c74d5f7809	4312d0d5-f8e5-4845-94f7-e2e8c5b1074d	NO

Figure 4.3 Contrail Command: Projects

The Kubernetes `cluster name` is configurable, but only during the deployment process. If you don't configure it `k8s` will be the default. Once the cluster is created, the name cannot be changed. To view the `cluster name`, you have to go to the `contrail-kube-manager` (KM) docker and check its configuration file.

To locate the `KM` docker container:

```
$ docker ps -a | grep  kubemanager
2260c7845964  ...snipped...  ago  Up  2  minutes  kubemanager_kubemanager_1
```

To log in to the `KM` container:

```
$ docker exec -it kubemanager_kubemanager_1 bash
```

To find the `cluster_name` option:

```
$ grep cluster /etc/contrail/contrail-kubernetes.conf
cluster_name=k8s        #<---
cluster_project={}
cluster_network={}
```

NOTE The rest of this book will refer to all these terms `namespace`, `NS`, `tenant`, and `project` interchangeably.

Non-Isolated Namespaces

You should be aware that one Kubernetes basic networking requirement is for a flat/NAT-less network – any pod can talk to any pod in any namespace – and any CNI provider must ensure that. Consequently, in Kubernetes, by default, all namespaces are *not* isolated:

NOTE The term isolated and non-isolated are in the context of (Contrail) networking only.

`k8s-default-pod-network and k8s-default-service-network`

To provide networking for all non-isolated namespaces, there should be a common VRF (virtual routing and forwarding) table or routing instance. In the Contrail Kubernetes environment, two default virtual networks are pre-configured in k8s' default namespace, for pod and for service, respectively. Correspondingly, there are two VRF tables, each with the same name as their corresponding virtual network.

The name of the two virtual networks/VRF tables is in this format:

```
<k8s-cluster-name>-<namespace name>-[pod|service]-network
```

So, for the `default` namespace with a default cluster name, `k8s`, the two Virtual network/VRF table names are:

- `k8s-default-pod-network`: the pod virtual network/VRF table, with the default subnet `10.32.0.0/12`
- `k8s-default-service-network`: the service virtual network /VRF table, with a default subnet `10.96.0.0/12`

NOTE The default subnet for pod or service is configurable.

It is important to know that these two default virtual networks are shared between all of the non-isolated namespaces. What that means is that they will be available for any new non-isolated namespace that you create, implicitly. That's why pods from all non-isolated namespaces, including default namespaces, can talk to each other.

On the other hand, any virtual networks that you create will be isolated with other virtual networks, regardless of the same or different namespaces. Communication between pods in two different virtual networks requires Contrail network policy.

NOTE Later, when you read about Kubernetes service , you may wonder why packets destined for the service virtual network/VRF table can reach the backend pod in pod virtual network/VRF table. Again, the good news is because of Contrail network policy. By default, Contrail network policy is enabled between the service and pod networks, which allows packets arriving to the service virtual network/VRF table to reach the pod, and vice versa.

Isolated Namespaces

In contrast, isolated namespaces have their own default pod-network and service-network, and accordingly, two new VRF tables are also created for each isolated namespace. The same flat-subnets `10.32.0.0/12` and `10.96.0.0/12` are shared by the pod and service networks in the isolated namespaces. However, since the networks are with a different VRF table, by default it is isolated with another namespace. Pods launched in isolated namespaces can only talk to service and pods on the same namespace. Additional configurations, for example, policy, are required to make the pod able to reach the network outside of the current namespace.

To illustrate this concept, let's use an example. Suppose you have three namespaces: the `default` namespace, and two user namespaces: `ns-non-isolated` and `ns-isolated`. In each namespace you can create one user virtual network: `vn-left-1`. You will end up following virtual network/VRF tables in Contrail:

- default-domain:k8s-default:k8s-default-pod-network
- default-domain:k8s-default:k8s-default-service-network
- default-domain:k8s-default:k8s-vn-left-1-pod-network
- default-domain:k8s-ns-non-isolated:k8s-vn-left-1-pod-network
- default-domain:k8s-ns-isolated:k8s-ns-isolated-pod-network
- default-domain:k8s-ns-isolated:k8s-ns-isolated-service-network
- default-domain:k8s-ns-isolated:k8s-vn-left-1-pod-network

NOTE The above names are listed in FQDN format. In Contrail, domain is the top-level object, followed by project/tenant, and then followed by virtual networks.

Figure 4.4 expertly illustrates all this.

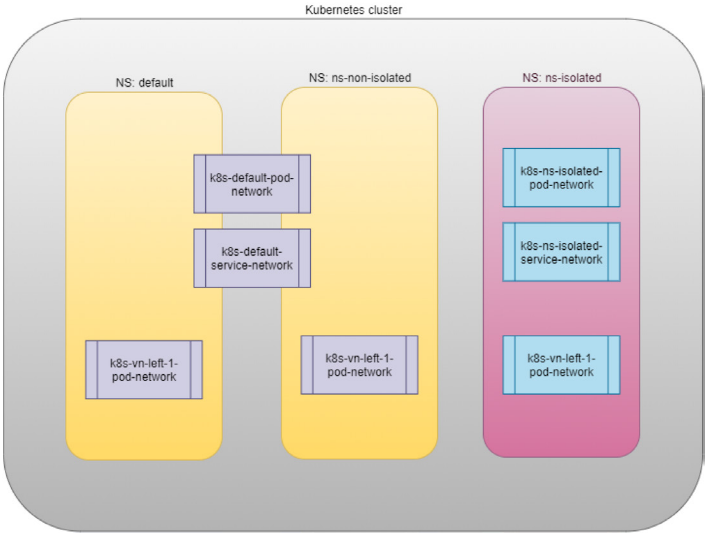

Kubernetes cluster
NS: default
NS: ns-non-isolated
NS: ns-isolated
k8s-default-pod-network
k8s-default-service-network
k8s-ns-isolated-pod-network
k8s-ns-isolated-service-network
k8s-vn-left-1-pod-network
k8s-vn-left-1-pod-network
k8s-vn-left-1-pod-network

Figure 4.4 *NS and Virtual Network*

Here is the YAML file to create an isolated namespace:

```
$ cat ns-isolated.yaml
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    "opencontrail.org/isolation" : "true"
  name: ns-isolated
```

And to create the NS:

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

$ kubectl get ns
NAME          STATUS    AGE
contrail      Active    8d
default       Active    8d
ns-isolated   Active    1d   #<---
kube-public   Active    8d
kube-system   Active    8d
```

The annotations under metadata are an additional way to compare standard (non-isolated) k8s namespace. The value of `true` indicates this is an isolated namespace:

```
annotations:
  "opencontrail.org/isolation" : "true"
```

You can see that this part of the definition is Juniper's extension. The `contrail-kube-manager` (`KM`) reads the namespace `metadata` from `kube-apiserver`, parses the information defined in the `annotations` object, and sees that the `isolation` flag is set to `true`. It then creates the tenant with the corresponding routing instance (one for pod and one for service) instead of using the default namespace routing instances for the isolated namespace. Fundamentally that is how the isolation is implemented.

The following sections will verify that the routing isolation is working.

Pods Communication Across NS

Create a non-isolated namespace and an isolated namespace:

```
$ cat ns-non-isolated.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: ns-non-isolated

$ cat ns-isolated.yaml
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    "opencontrail.org/isolation": "true"
  name: ns-isolated

$ kubectl apply -f ns-non-isolated.yaml
namespace/ns-non-isolated created

$ kubectl apply -f ns-isolated.yaml
namespace/ns-isolated created

$ kubectl get ns | grep isolate
ns-isolated       Active   79s
ns-non-isolated   Active   73s
```

In both namespaces and the default namespace, create a deployment to launch a webserver pod:

```
#deploy-webserver-do.yaml
apiVersion: apps/v1
- {key: app, operator: In, values: [webserver]}

$ kubectl apply -f deploy-webserver-do.yaml -n default
deployment.extensions/webserver created

$ kubectl apply -f deploy-webserver-do.yaml -n ns-non-isolated
deployment.extensions/webserver created
```

```
$ kubectl apply -f deploy-webserver-do.yaml -n ns-isolated
deployment.extensions/webserver created

$ kubectl get pod -o wide -n default
NAME                        READY  STATUS   ... IP             NODE     ...
webserver-85fc7dd848-tjfn6  1/1    Running  ... 10.47.255.242  cent333  ...
$ kubectl get pod -o wide -n ns-non-isolated...
NAME                        READY  STATUS   ... IP             NODE     ...
webserver-85fc7dd848-nrxq6  1/1    Running  ... 10.47.255.248  cent222  ...

$ kubectl get pod -o wide -n ns-isolated
NAME                        READY  STATUS   ... IP             NODE     ...
webserver-85fc7dd848-6l7j2  1/1    Running  ... 10.47.255.239  cent222  ...
```

Ping between all pods in three namespaces:

```
#default ns to non-isolated new ns: succeed
$ kubectl -n default exec -it webserver-85fc7dd848-tjfn6 -- ping 10.47.255.248
PING 10.47.255.248 (10.47.255.248): 56 data bytes
64 bytes from 10.47.255.248: seq=0 ttl=63 time=1.600 ms
^C
--- 10.47.255.248 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 1.600/1.600/1.600 ms

#default ns to isolated new ns: fail
$ kubectl -n default exec -it webserver-85fc7dd848-tjfn6 -- ping 10.47.255.239
PING 10.47.255.239 (10.47.255.239): 56 data bytes
^C
--- 10.47.255.239 ping statistics ---
3 packets transmitted, 0 packets received, 100% packet loss
```

The test result shows that bidirectional communication between two non-isolated namespaces (namespace `ns-non-isolated` and `default`, in this case) works, but traffic from a non-isolated namespace (`default` ns) toward an isolated namespace does not pass through. What about traffic within the same isolated namespace?

With the power of `deployment` you can quickly test it out: in isolated namespace `ns-isolated`, clone one more pod by `scale` the deployment with `replicas=2` and ping between the two pods:

```
$ kubectl scale deployment webserver --replicas=2
$ kubectl get pod -o wide -n ns-isolated
NAME                        READY  STATUS   RESTARTS  AGE  IP             NODE
webserver-85fc7dd848-6l7j2  1/1    Running  0         8s   10.47.255.239  cent222
webserver-85fc7dd848-215k8  1/1    Running  0         8s   10.47.255.238  cent333

$ kubectl -n ns-isolated exec -it webserver-85fc7dd848-6l7j2 -- ping 10.47.255.238
PING 10.47.255.238 (10.47.255.238): 56 data bytes
64 bytes from 10.47.255.238: seq=0 ttl=63 time=1.470 ms
^C
--- 10.47.255.238 ping statistics ---
```

```
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 1.470/1.470/1.470 ms
```

The ping packet passes through now. To summarize the test results:

- Traffic is not isolated between non-isolated namespace.
- Traffic is isolated between an isolated namespace and all other tenants in the cluster.
- Traffic is not isolated in the same namespace.

NOTE Pod-level isolation can be achieved via Kubernetes network policy, or security groups in Contrail, all covered later in this chapter.

Contrail Floating IP

Communication has been discussed and tested between pods in the same or different namespace, but so far, it's been inside of the same cluster. What about communication with devices outside of the cluster?

You may already know that in the traditional (OpenStack) Contrail environment, there are many ways for the overlay entities (typically a VM) to access the Internet. The three most frequent methods are:

- floating IP
- fabric SNAT
- logical router

The preferred Kubernetes solution is to expose any service via `service` and `Ingress` objects, which you've read about in Chapter 3. In the Contrail Kubernetes environment, floating IP is used in the service and ingress implementation to expose them to what's outside of the cluster. Later this chapter discusses each of these two objects. But first, let's review the floating IP basis and look at how it works with Kubernetes.

NOTE The `fabric SNAT` and `logical router` are used by overlay workloads (VM and pod) to reach the Internet but initializing communication from the reverse direction is not possible. However `floating IP` supports traffic initialized from both directions – you can configure it to support ingress traffic, egress traffic, or both, and the default is bi-directional. This book focuses only on `floating IP`. Refer to Contrail documentation for detailed information about the fabric SNAT and logical router: https://www.juniper.net/documentation/en_US/contrail5.0/information-products/pathway-pages/contrail-feature-guide-pwp.html.

Floating IP and Floating IP Pool

The `floating IP`, or `FIP` for short, is a traditional concept that Contrail has supported since its very early releases. Essentially, it's an OpenStack concept to map a VM IP, which is typically a private IP address, to a public IP (the floating IP in this context) that is reachable from the outside of the cluster. Internally, the one-to-one mapping is implemented by NAT. Whenever a vRouter receives packets from outside of the cluster destined to the floating IP, it will translate it to the VM's private IP and forward the packet to the VM. Similarly, it will do the translation on the reverse direction. Eventually both VM and Internet host can talk to each other, and both can initiate the communication.

NOTE The vRouter is a Contrail forwarding plane that resides in each compute node handling workload traffic.

Figure 4.5 illustrates the basic workflow of floating IP.

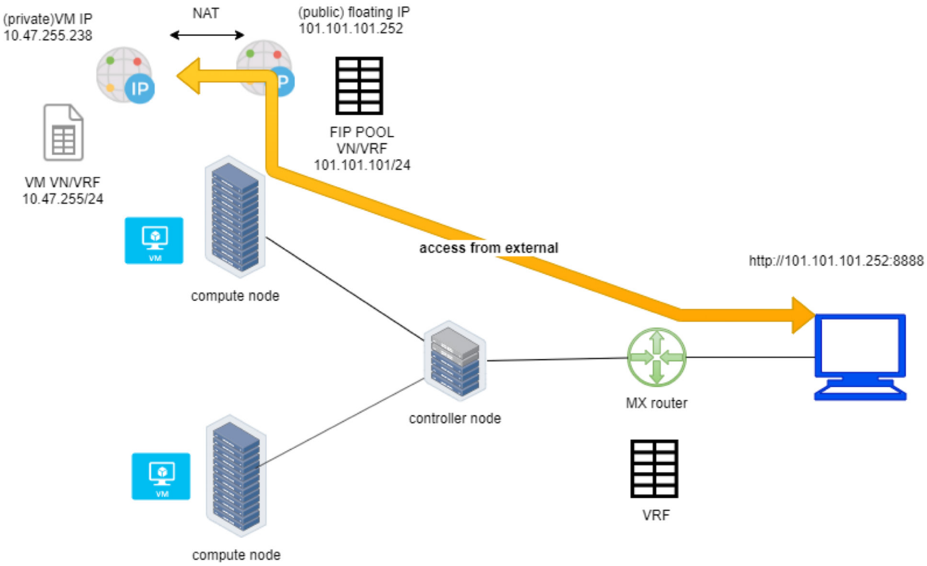


Figure 4.5 *Floating IP Workflow*

Here are some highlights regarding floating IP to keep in mind:

- A floating IP is associated with a VM's `port`, or a `VMI` (Virtual Machine Interface).
- A floating IP is allocated from a `FIP pool`.

- A floating IP pool is created based on a virtual network (FIP-VN).
- The FIP-VN will be available to outside of the cluster, by setting matching route-target (RT) attributes of the gateway router's VRF table.
- When a gateway router sees a match with its route import policy in the RT, it will load the route into its VRF table. All remote clients connected to the VRF table will be able to communicate with the floating IP.

There is nothing new in the Contrail Kubernetes environment regarding the floating IP concept and role. But the use of floating IP has been extended in Kubernetes service and ingress object implementation, and it plays an important role for accessing Kubernetes service and ingress externally. You can check later sections in this chapter for more details.

Create FIP Pool

Let's create a floating IP pool in a three-step process:

1. Create a public floating IP-VN.
2. Set RT (route-target) for the virtual network so it can be advertised and imported into the gateway router's VRF table.
3. Create a floating IP pool based on the public floating IP-virtual network.

Again, there is nothing new here. The same steps would be required in other Contrail environments without Kubernetes. However, as you've learned in previous sections, with Contrail Kubernetes integration a floating IP-virtual network can now be created in Kubernetes style.

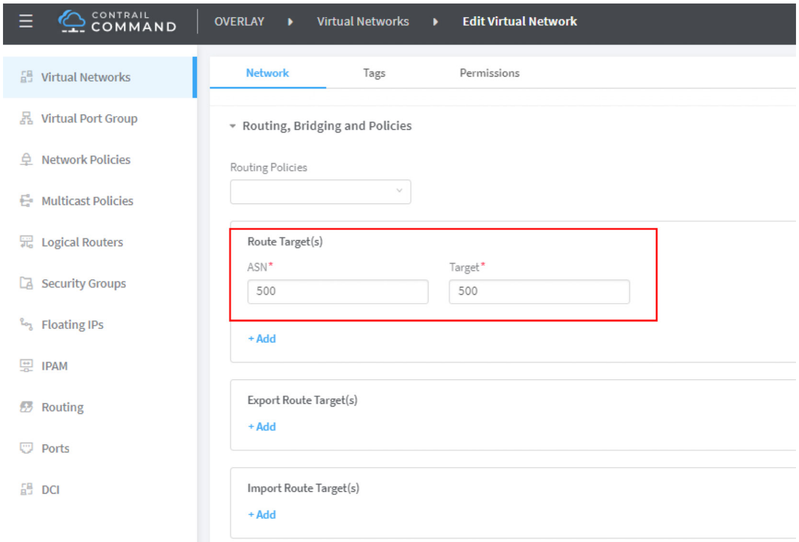
Create a Public Floating IP-Virtual Network Named vn-ns-default

```
# vn-ns-default.yaml
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  annotations:
    "opencontrail.org/cidr": "101.101.101.0/24"
  name: vn-ns-default
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "contrail-k8s-cni"
  }'

$ kubectl apply -f vn-ns-default.yaml
networkattachmentdefinition.k8s.cni.cncf.io/vn-ns-default created

$ kubectl get network-attachment-definitions.k8s.cni.cncf.io
NAME            AGE
vn-ns-default   22d
```

Now set the routing target.

If you need the floating IP to be reachable from the Internet through the gateway router, you'll need to set a route target for the virtual network prefix getting imported in the gateway router's VRF table (see Figure 4.6). This step is necessary whenever Internet access is required.

The UI navigation path to set the RT is: Contrail Command > Main Menu > Overlay > Virtual Networks > k8s-vn-ns-default-pod-network > Edit > Routing, Bridging and Policies.



Route Target(s)

ASN* 500

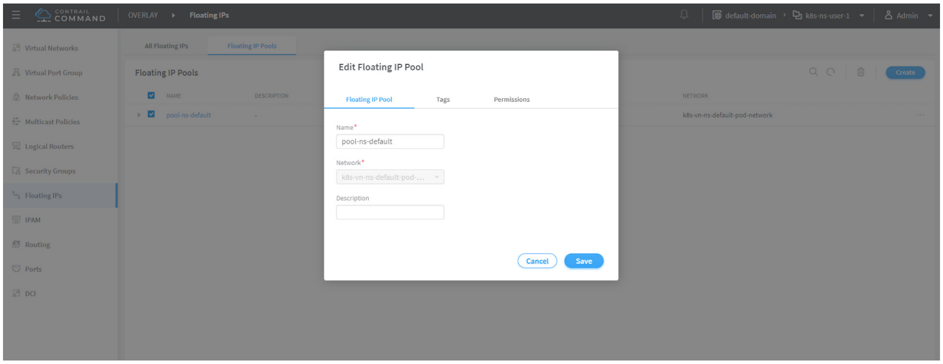
Target* 500

Figure 4.6 Contrail Command: Setting RT

Now let's create a floating IP pool based on the public virtual network.

This is the final step. From the Contrail Command UI, create a floating IP pool based on the public virtual network. The UI navigation path for this setting shown in Figure 4.7 is: Contrail Command > Main Menu > Overlay > Floating IP > Create.

TIP The Contrail UI also allows you to set the external flag in virtual network advanced options, so that a floating IP pool named *public* will automatically be created.



Floating IP Pools

☑ NAME	DESCRIPTION	NETWORK
☑ pool-ns-default	-	k8s-vn-ns-default-pod-network

Edit Floating IP Pool

Name* pool-ns-default

Network* k8s-vn-ns-default-pod-...

Description

Figure 4.7 *Contrail Command: Create a Floating IP Pool*

Floating IP Pool Scope

There are different ways you can refer to a floating IP pool in the Contrail Kubernetes environment, and correspondingly the scope of the pools will also be different. The three possible levels with descending priority are:

- object specific
- namespace level
- global level

Object Specific

This is the most specific level of scope. An object specific floating IP pool binds itself only to the object that you specified, it does not affect any other objects in the same namespace or cluster. For example, you can specify a service object `web` to get floating IP from the floating IP pool `pool1`, a service object `dns` to get floating IP from another floating IP pool `pool2`, etc. This gives the most granular control of where the floating IP will be allocated from for an object – the cost is that you need to explicitly specify it in your YAML file for every object.

Namespace Level

In a multi-tenancy environment each namespace would be associated to a tenant, and each tenant would have a dedicated floating IP pool. In that case, it is better to have an option to define a floating IP pool at the NS level, so that all objects created in that namespace will get floating IP assignment from that pool. With the namespace level pool defined (for example, `pool-ns-default`), there is no need to specify the floating IP-pool name in each object's YAML file any more. You can still give a different pool name, say `my-webservice-pool` in an object `webservice`. In that case, object `webservice` will get the floating IP from `my-webservice-pool` instead of from the namespace level pool `pool-ns-default`, because the former is more specific.

Global Level

The scope of the global level pool would be the whole cluster. Objects in any namespaces can use the global floating IP pool.

You can combine all three methods to take advantage of their combined flexibility. Here's a practical example:

- Define a global pool `pool-global-default`, so any objects in a namespace that has no namespace-level or object-level pool defined, will get a floating IP from this pool.
- For ns `dev`, define a floating IP pool `pool-dev`, so all objects created in ns `dev` will by default get floating IP from `pool-dev`.
- For ns `sales`, define a floating IP pool `pool-sales`, so all objects created in ns `sales` will by default get floating IP from `pool-sales` .
- For ns `test-only`, do *not* define any namespace-level pool, so by default objects created in it will get floating IP from the `pool-global-default`.
- When a service `dev-webservice` in ns `dev` needs a floating IP from `pool-sales` instead of `pool-dev`, specifying `pool-sales` in `dev-webservice` object YAML file will achieve this goal.

NOTE Just keep in mind the rule of thumb – the most specific scope will always prevail.

Object Floating IP Pool

Let's first take a look at the object-specific floating IP pool:

```
#service-web-lb-pool-public-1.yaml
apiVersion: v1
kind: Service
metadata:
  name: service-web-lb-pool-public-1
  annotations:
    "opencontrail.org/fip-pool": "{'domain': 'default-domain', 'project': 'k8s-ns-
user-1', 'network': 'vn-public-1', 'name': 'pool-public-1'}"
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver
  type: LoadBalancer
```

In this example, service service-web-lb-pool-public-1 will get an floating IP from pool pool-public-1, which is created based on virtual network vn-public-1 under current project k8s-ns-user-1. The corresponding Kubernetes namespace is ns-user-1. Since object-level floating IP pool is assigned for this specific object only, with this method each new object needs to be explicitly assigned a floating IP pool.

NS Floating IP Pool

The next floating IP pool scope is in the namespace level. Each namespace can define its own floating IP pool. In the same way as a Kubernetes annotations object is used to give a subnet to a virtual network, it is also used to specify a floating IP pool. The YAML file looks like this:

```
#ns-user-1-default-pool.yaml
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    opencontrail.org/isolation: "true"
    opencontrail.org/fip-pool: "{'domain': 'default-domain', 'project': 'k8s-ns-
user-1', 'network': 'vn-ns-default', 'name': 'pool-ns-default'}"
  name: ns-user-1
```

Here ns-user-1 is given a namespace-level floating IP pool named pool-ns-default, and the corresponding virtual network is vn-ns-default. Once the ns-user-1 is created with this YAML file, any new service which requires a floating IP, if not created with the object-specific pool name in its YAML file, will get a floating IP allocated from this pool. In practice, most namespaces (especially those isolated namespaces) will need their own namespace default pool so you will see this type of configuration very often in the field.

Global floating IP Pool

To specify a global level floating IP pool, you need to give the fully-qualified pool name (domain > project > network > name) in contrail-kube-manager (KM) Docker's configuration file(/etc/contrail/contrail-kubernetes.conf). This file is automatically generated by the Docker during its bootup based on its ENV parameters, which can be found in the /etc/contrail/common_kubemanager.env file in the master node:

```
$ cat /etc/contrail/common_kubemanager.env
VROUTER_GATEWAY=10.169.25.1
CONTROLLER_NODES=10.85.188.19
KUBERNETES_API_NODES=10.85.188.19
RABBITMQ_NODE_PORT=5673
CLOUD_ORCHESTRATOR=kubernetes
KUBEMANAGER_NODES=10.85.188.19
CONTRAIL_VERSION=master-latest
KUBERNETES_API_SERVER=10.85.188.19
```

```
TTY=True
ANALYTICS_SNMP_ENABLE=True
STDIN_OPEN=True
ANALYTICS_ALARM_ENABLE=True
ANALYTICSDB_ENABLE=True
CONTROL_NODES=10.169.25.19
```

As you can see, this `.env` file contains important environmental parameters about the setup. To specify a `global FIP pool`, add the following line:

```
KUBERNETES_PUBLIC_FIP_POOL={'domain': 'default-domain','name': 'pool-global-default','network': 'vn-
global-default','project': 'k8s-ns-user-1'}
```

It reads: the global floating IP pool is called `pool-global-default` and it is defined based on a virtual network `vn-global-default` under project `k8s-ns-user-1`. This indicates that the corresponding Kubernetes namespace is `ns-user-1`.

Now with that piece of configuration placed, you can re-compose the `contrail-kube-manager` Docker container to make the change take effect. Essentially you need to tear it down and then bring it back up:

```
$ cd /etc/contrail/kubemanager/
$ docker-compose down;docker-compose up -d
Stopping kubemanager_kubemanager_1 ... done
Removing kubemanager_kubemanager_1 ... done
Removing kubemanager_node-init_1   ... done
Creating kubemanager_node-init_1 ... done
Creating kubemanager_kubemanager_1 ... done
```

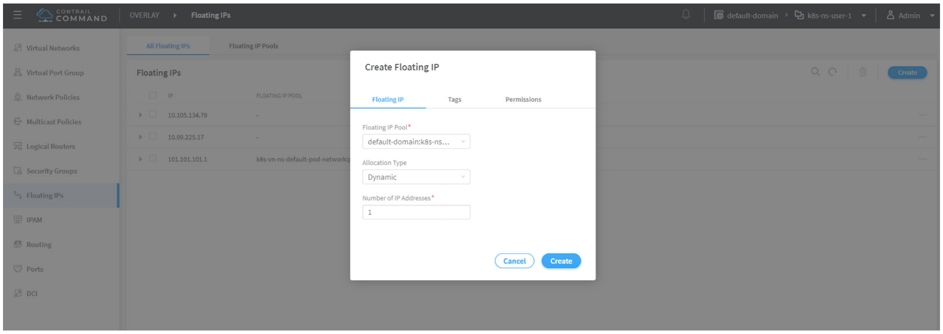
Now the global floating IP pool is specified for the cluster.

NOTE In all three scopes, floating IP is automatically allocated and associated only to service and ingress objects. If the floating IP has to be associated to a pod it has to be done manually. We'll talk about this in the next section.

Floating IP for Pods

Once floating IP pool is created and available, a floating IP can be allocated from the floating IP pool for the pods that require one. This can be done by associating a floating IP to a VMI (VM, or pod, interface),

You can manually create a floating IP out of a floating IP pool in Contrail UI, and then associate it with a pod VMI as in Figures 4.8 and 4.9.



Floating IPs

IP	FLOATING IP POOL
10.105.134.79	-
10.99.225.17	-
101.101.101.1	k8s-vn-ns-default-pod-network:

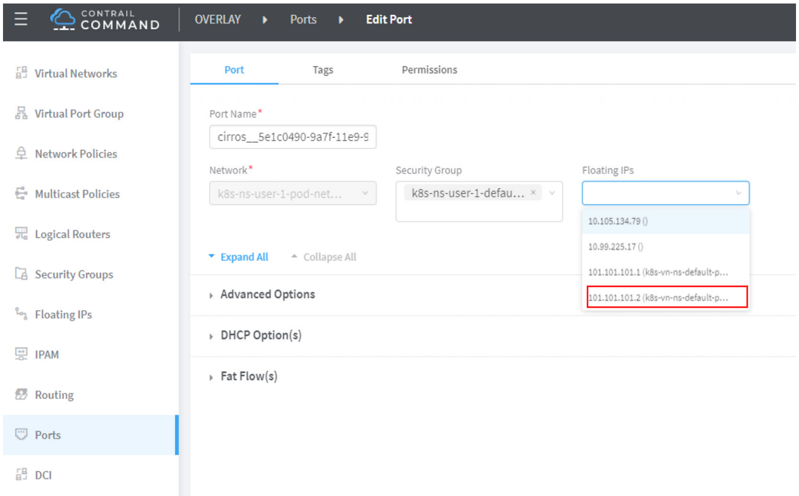
Create Floating IP

Floating IP Pool*: default-domain:k8s-ns...

Allocation Type: Dynamic

Number of IP Addresses*: 1

Figure 4.8 *Create Floating IP*



Port Name*: cirros__5e1c0490-9a7f-11e9-9

Network*: k8s-ns-user-1-pod-net...

Security Group: k8s-ns-user-1-defau... ×

Floating IPs:

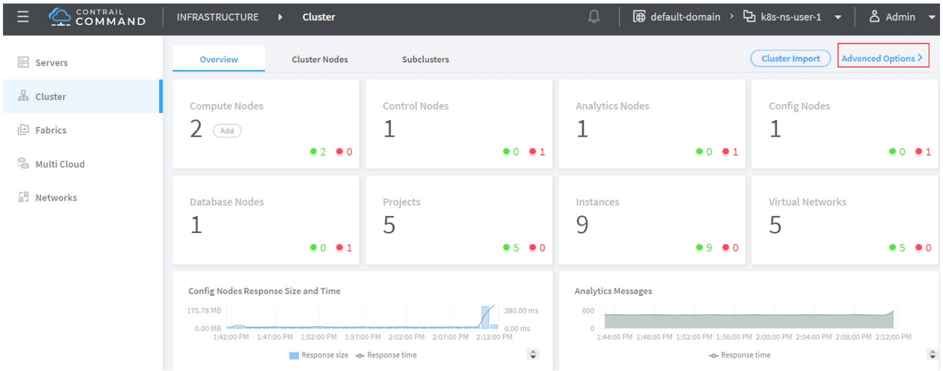
10.105.134.79 ()
10.99.225.17 ()
101.101.101.1 (k8s-vn-ns-default-p...
101.101.101.2 (k8s-vn-ns-default-p...

Figure 4.9 *Associate a Floating IP in a Pod Interface*

NOTE Make sure the floating IP pool is shared to the project where floating IP is going to be created.

Advertising Floating IP

Once a floating IP is associated to a pod interface, it will be advertised to the MP-BGP peers, which are typically gateway routers. The following Figures, 4.10, 4.11, and 4.12, show how to add and edit a BGP peer.



Compute Nodes 2 ● 2 ● 0

Control Nodes 1 ● 0 ● 1

Analytics Nodes 1 ● 0 ● 1

Config Nodes 1 ● 0 ● 1

Database Nodes 1 ● 0 ● 1

Projects 5 ● 5 ● 0

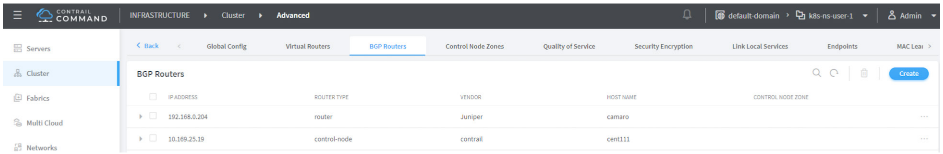
Instances 9 ● 9 ● 0

Virtual Networks 5 ● 5 ● 0

Config Nodes Response Size and Time

Analytics Messages

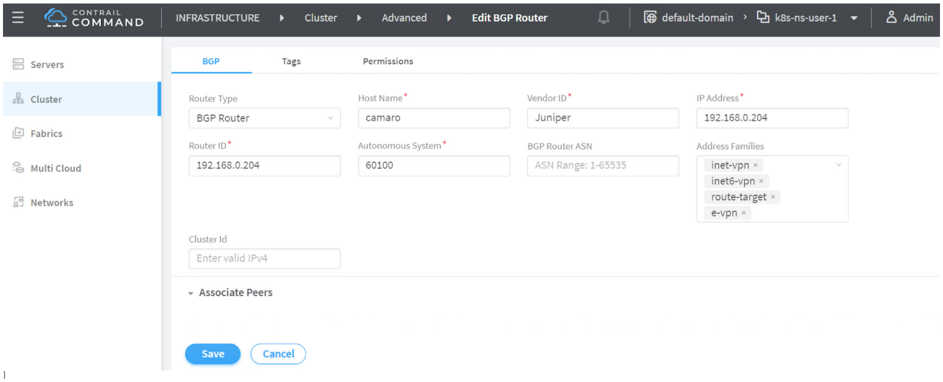
Figure 4.10 *Contrail Command: Select Main-Menu > INFRASTRUCTURE: Cluster > Advanced Options*



BGP Routers

IP ADDRESS	ROUTER TYPE	VENDOR	HOST NAME	CONTROL NODE ZONE
192.168.0.204	router	Juniper	camaro	
10.169.25.19	control-node	contrail	cent111	

Figure 4.11 *Contrail Command: Select BGP Router > Create*



Router Type: BGP Router

Host Name*: camaro

Vendor ID*: Juniper

IP Address*: 192.168.0.204

Router ID*: 192.168.0.204

Autonomous System*: 60100

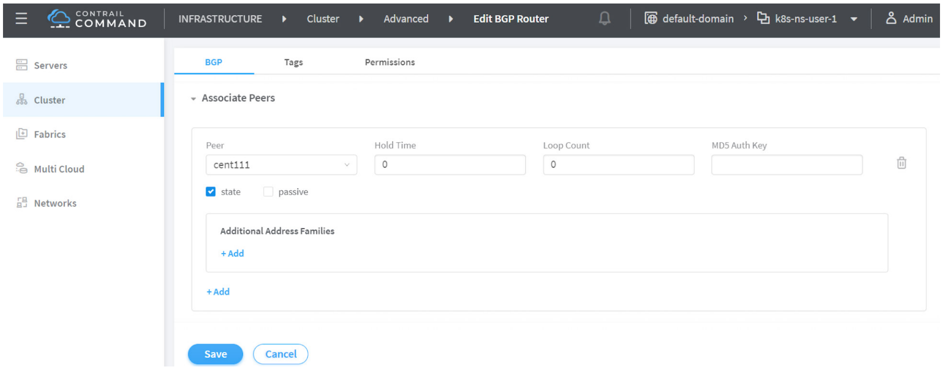
BGP Router ASN: ASN Range: 1-65535

Address Families: inet-vpn × inet6-vpn × route-target × e-vpn ×

Cluster Id: Enter valid IPv4

Figure 4.12 *Edit BGP Peer Parameters*

Input all the BGP peer information and don't forget to associate the controller(s), which is shown next in Figure 4.13.



Peer: cent111
Hold Time: 0
Loop Count: 0
MD5 Auth Key:

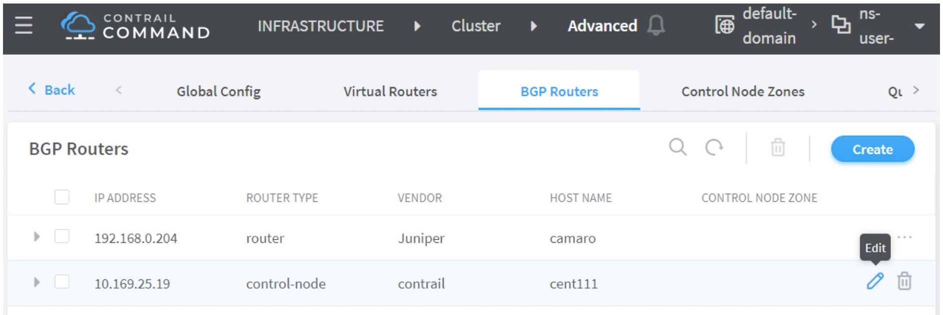
☑ state ☐ passive

Additional Address Families
+ Add

+ Add

Figure 4.13 Associate the Peer to a Controller

From the dropdown of `peer` under `Associated Peers`, select the controller(s) to peer with this new BGP router that you are trying to add. Click `save` when done. A new BGP peer with ROUTER TYPE router will pop up.

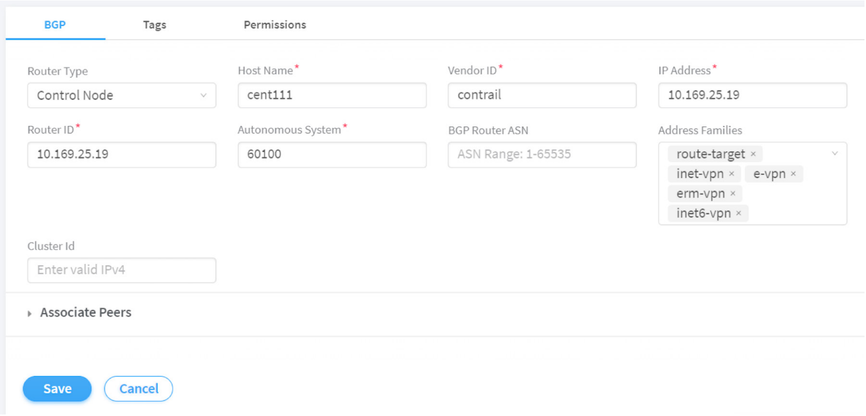


BGP Routers

	IP ADDRESS	ROUTER TYPE	VENDOR	HOST NAME	CONTROL NODE ZONE
☐	192.168.0.204	router	Juniper	camaro	
☐	10.169.25.19	control-node	contrail	cent111	

Figure 4.14 A New BGP Router in the BGP Router List

Now we've added a peer BGP router as type router. For the local BGP speaker, which is with type control-node, you just need to double-check the parameters by clicking the Edit button. In this test we want to build an MP-IBGP neighborship between Contrail Controller and the gateway router, so make sure the ASN and Address Families fields match on both ends, refer to Figure 4.15.



Router Type: Control Node
Host Name*: cent111
Vendor ID*: contrail
IP Address*: 10.169.25.19
Router ID*: 10.169.25.19
Autonomous System*: 60100
BGP Router ASN: ASN Range: 1-65535
Address Families: route-target × inet-vpn × e-vpn × erm-vpn × inet6-vpn ×
Cluster Id: Enter valid IPv4
Associate Peers

Figure 4.15 *Contrail Controller BGP Parameters: ASN*

Now you can check BGP neighborship status in the gateway router:

```
labroot@camaro> show bgp summary | match 10.169.25.19
10.169.25.19          60100       2235       2390       0      39     18:19:34 Establ
```

Once the neighborship is established, BGP routes will be exchanged between the two speakers, and that is when we'll see that the floating IP assigned to the Kubernetes object is advertised by the master node (`10.169.25.19`) and learned in the gateway router:

```
labroot@camaro> show route table k8s-test.inet.0 101.101.101.2
Jul 11 01:18:31

k8s-test.inet.0: 8 destinations, 8 routes (8 active, 0 holddown, 0 hidden)
@ = Routing Use Only, # = Forwarding Use Only
+ = Active Route, - = Last Active, * = Both

101.101.101.2/32   *[BGP/170] 00:01:42, MED 200, localpref 100, from 10.169.25.19
                      AS path: ?
                    validation-state: unverified, > via gr-2/3/0.32771, Push 47
```

The `detail` version of the same command tells more: the floating IP route is reflected from the Contrail Controller, but `Protocol next hop` being the compute node (`10.169.25.20`) indicates that the floating IP is assigned to a compute node. One entity currently running in that compute node owns the floating IP:

```
labroot@camaro> show route table k8s-test.inet.0 101.101.101.2 detail | match "next hop"
Jul 11 01:19:18
                Next hop type: Indirect, Next hop index: 0
                Next hop type: Router, Next hop index: 1453
                Next hop: via gr-2/3/0.32771, selected
                Protocol next hop: 10.169.25.20
                Indirect next hop: 0x900e640 1048601 INH Session ID: 0x70f
```

The dynamic soft GRE configuration makes the gateway router automatically create a soft GRE tunnel interface:

```
labroot@camaro> show interfaces gr-2/3/0.32771
Jul 11 01:19:53
  Logical interface gr-2/3/0.32771 (Index 432) (SNMP ifIndex 1703)
    Flags: Up Point-To-Point SNMP-Traps 0x4000
    IP-Header 10.169.25.20:192.168.0.204:47:df:64:0000000800000000 Encapsulation: GRE-NULL
    Copy-tos-to-outer-ip-header: Off, Copy-tos-to-outer-ip-header-transit: Off
    Gre keepalives configured: Off, Gre keepalives adjacency state: down
    Input packets : 0
    Output packets: 0
    Protocol inet, MTU: 9142
    Max nh cache: 0, New hold nh limit: 0, Curr nh cnt: 0, Curr new hold cnt: 0, NH drop cnt: 0
      Flags: None
    Protocol mpls, MTU: 9130, Maximum labels: 3
      Flags: None
```

The `IP-Header` indicates a GRE outer IP header, so the tunnel is built from the current gateway router whose BGP local address is `192.168.0.204`, to the remote node `10.169.25.20`, in this case it's one of the Contrail compute nodes. The floating IP advertisement process is illustrated in Figure 4.16.

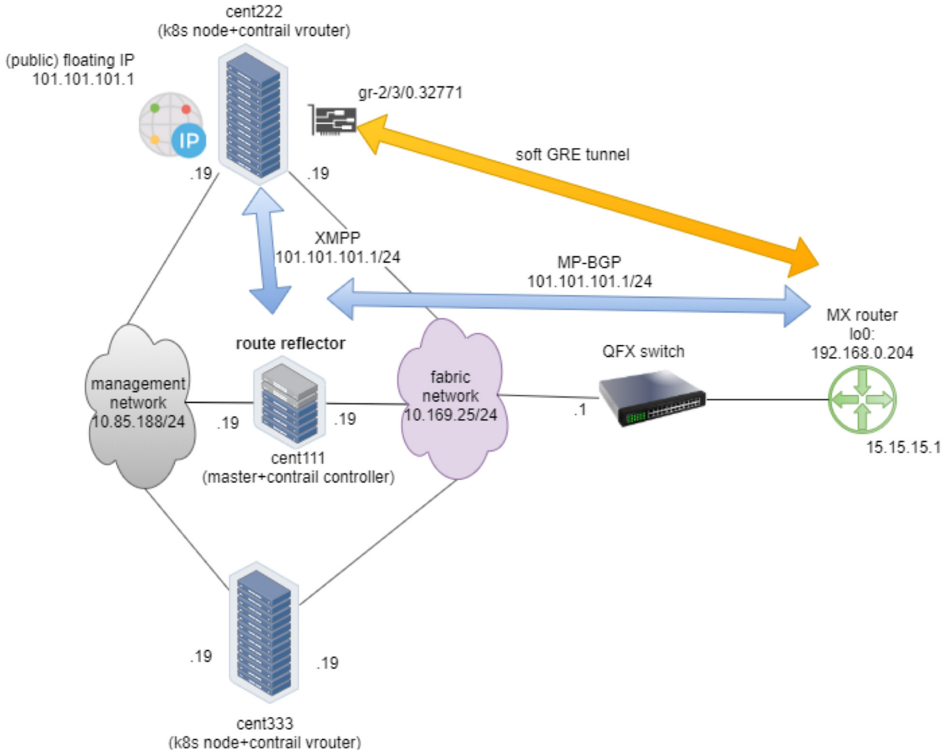


Figure 4.16 Floating IP Advertisement

Summary

In this chapter we created the following objects:

- Ns: ns-user-1
- FIP VN: vn-ns-default
- FIP pool: pool-ns-default

The `ns-user-1` ns project, which refers to a namespace-level pool `pool-ns-default` that is to be created manually, will hold all of our test objects. The namespace-level pool is based on the virtual network `vn-ns-default` that has subnet 101.101.101/24. The floating IP for objects created in namespace ns-user-1 will be assigned from this subnet.

NOTE Once you have YAML files (given earlier) ready for the namespace and floating IP-virtual network, you can create these objects:

```
$ kubectl apply -f ns/ns-user-1-default-pool.yaml
namespace/ns-user-1 created
$ kubectl apply -f vn/vn-ns-default.yaml
networkattachmentdefinition.k8s.cni.cncf.io/vn-ns-default created
```

The floating IP-pool needs to be created separately in Contrail's UI. Refer to the *Contrail Floating IP* section for the details.

With these objects there is a namespace associated with a floating IP pool. From inside of this namespace you can proceed to create and study other Kubernetes objects, such as Service.

NOTE All tests in this book that demonstrate service and ingress will be created under this `ns-user-1` namespace.

Chapter 5

Contrail Services

This chapter looks at Kubernetes `service` in the Contrail environment. Specifically, it will focus on `clusterIP` and `loadbalancer` type of services that are commonly used in practice. Contrail uses its `loadbalancer` object to implement these two type of services. First, we'll review the concept of legacy Contrail neutron load balancer, then we'll look into the extended ECMP load balancer object, which is the object that these two types of services are based on in Contrail. The final part of this chapter explores how the `clusterIP` and `loadbalancer` service works, in detail, each with a test case built in the book's testbed.

Kubernetes Service

Service is the core object in Kubernetes. In Chapter 3 you learned what Kubernetes service is and how to create a `service` object with a YAML file. Functionally, a service is running as a Layer 4 (transport layer) load balancer that is sitting between clients and servers. Clients can be anything requesting a service. The server in our context is the backend pods responding to the request. The client only sees the frontend - a service IP and service port exposed by the service, and it does not (and does not need to) care about which backend pods (and what pod IP) actually responds to the service request. Inside of the cluster, that `service IP`, also called `cluster IP`, is a kind of virtual IP (`VIP`).

NOTE In the Contrail environment it is implemented through floating IP.

This design model is very powerful and efficient in the sense that it covers the fragility of the possible single point failure that may be caused by failure of any individual pod providing the service, therefore making a `service` much more robust from client's perspective.

In the Contrail Kubernetes integrated environment, all three types of services are supported:

- clusterIP
- nodePort
- loadbalancer

Now let's see how the service is implemented in Contrail.

Contrail Service

Chapter 3 introduced Kubernetes' default implementation of service through `kube-proxy`. In Chapter 3 we mentioned that CNI providers can have their own implementations. Well, in Contrail, `nodePort` service is implemented by kube-proxy. However, `clusterIP` and `loadbalancer` services are implemented by Contrail's `loadbalancer` (`LB`).

Before diving into the details of Kubernetes service in Contrail, let's review the legacy OpenStack-based load balancer concept in Contrail.

TIP For brevity, sometimes `loadbalancer` is also referred to as `LB`.

Contrail Openstack Load Balancer

Contrail load balancer is a foundation feature supported since its first release. It enables the creation of a pool of VMs serving applications, sharing one `virtual-ip` (`VIP`) as the front-end IP towards clients. Figure 5.1 illustrates Contrail load balancer and its components.

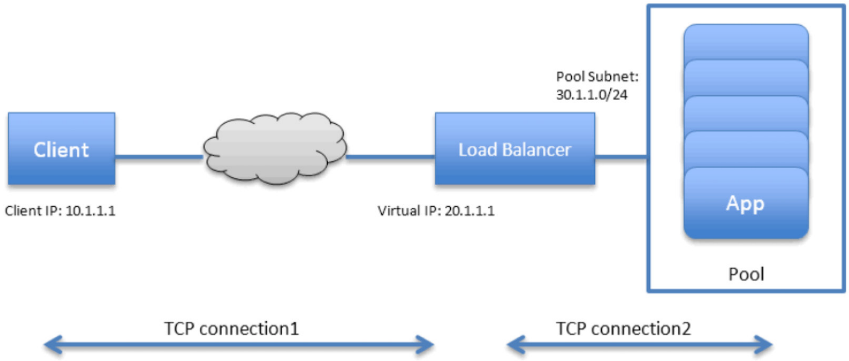


Figure 5.1 Contrail OpenStack Load Balancer

Some highlights of Figure 5.1 are:

- The LB is created with an internal VIP 30.1.1.1. An LB listener is also created for each listening port.
- Together, all back-end VMs compose a pool which is with subnet 30.1.1.0/24, the same as LB's internal VIP.
- Each back-end VM in the pool, also called a member, is allocated an IP from the pool subnet 30.1.1.0/24.
- To expose the LB to the external world, it has allocated another VIP which is external VIP 20.1.1.1.
- A client only sees one external VIP 20.1.1.1, representing the whole service.
- When LB sees a request coming from the client, it does TCP connection proxying. That means it establishes the TCP connection with the client, extracts the client's HTTP/HTTPS requests, creates a new TCP connection towards one of the back-end VMs from the pool, and sends the request in the new TCP connection.
- When LB gets its response from the VM, it forwards the response to the client.
- And when the client closes the connection to the LB, the LB may also close its connection with the back-end VM.

TIP When the client closes its connection to the LB, the LB may or may not close its connection to the back-end VM. Depending on the performance, or other considerations, it may use a timeout before it tears down the session.

You can see that this load balancer model is very similar to the Kubernetes service concept:

- VIP is the service IP
- backend VM becomes backend pods
- members are added by Kubernetes instead of OpenStack.

In fact, Contrail re-uses a good part of this model in its Kubernetes service implementation. To support service load balancing, Contrail extends the load balancer with a new driver. Along with the driver, service will be implemented as an equal cost multiple path (ECMP) load balancer working in Layer 4 (transport layer). This is the primary difference when compared with the proxy mode used by the OpenStack load balancer type.

- Actually any load balancer can be integrated with Contrail via the Contrail component conrail-svc-monitor.
- Each load balancer has a load balancer driver that is registered to Contrail with a loadbalancer_provider type.

- The `contrail-svc-monitor` listens to Contrail `loadbalancer`, `listener`, `pool`, and `member` objects. It also calls the registered load balancer driver to do other necessary jobs based on the `loadbalancer_provider` type.
- Contrail by default provides ECMP load balancer (`loadbalancer_provider` is `native`) and haproxy load balancer (`loadbalancer_provider` is `opencontrail`).
- The OpenStack load balancer is using haproxy load balancer.
- Ingress, on the other hand, is conceptually even closer to the OpenStack load balancer in the sense that both are Layer 7 (Application Layer) proxy-based. More about ingress will be discussed in later sections.

Contrail Service Load Balancer

Let's take a look at service load balancer and related objects.

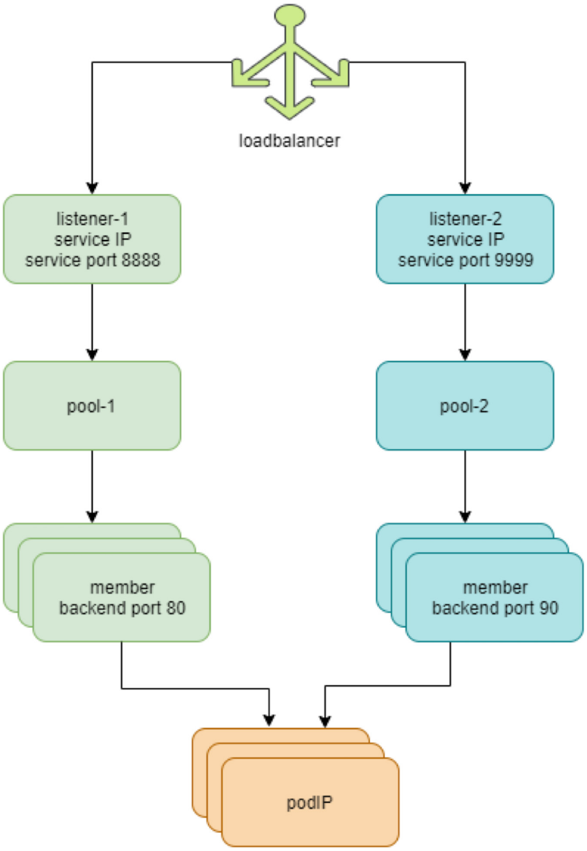


Figure 5.2 Service Load Balancer

The highlights in Figure 5.2 are:

- Each service is represented by a `loadbalancer` object.
- The load balancer object comes with a `loadbalancer_provider` property. For service implementation, a new `loadbalancer_provider` type called `native` is implemented.
- For each service port a `listener` object is created for the same service `loadbalancer`.
- For each `listener` there will be a `pool` object.
- The `pool` contains `members`, depending on the number of back-end pods, one pool may have multiple `members`.
- Each member object in the pool will map to one of the back-end pods.
- `Contrail-kube-manager` listens `kube-apiserver` for k8s service and when a `custerIP` or `loadbalancer` type of `service` is created, a `loadbalancer` object with `loadbalancer_provider` property `native` is created.
- `Loadbalancer` will have a virtual IP `VIP`, which is the same as the `serviceIP`.
- The `service-ip`/`VIP` will be linked to the interface of each back-end pod. This is done with an ECMP load balancer driver.
- The linkage from `service-ip` to the interfaces of multiple back-end pods creates an ECMP next-hop in Contrail, and traffic will be load balanced from the source pod towards one of the back-end pods directly. Later we'll show the ECMP prefix in the pod's VRF table.
- The `contrail-kube-manager` continues to listen to `kube-apiserver` for any changes, based on the pod list in `Endpoints`, it will know the most current back-end pods, and update members in the pool.

The most important thing to understand in Figure 5.2, as mentioned before, is that in contrast to the legacy neutron load balancer (and the ingress load balancer which we'll discuss later), there is no application layer proxy in this process. Contrail service implementation is based on Layer 4 (transport layer) ECMP-based load balancing.

Contrail Load Balancer Objects

We've talked a lot about the Contrail load balancer object and you may wonder what exactly it looks like. Well it's time to dig a little bit deeper to look at the load balancers and the supporting objects: listener, pool, and members.

In a Contrail setup you can pull the object data either from Contrail UI, CLI (`curl`), or third-party UI tools based on REST API. In production, depending on which one is available and handy, you can select your favorite.

Let's explore load balancer object with `curl`. With the `curl` tool you just need a FQDN of the URL pointing to the object.

For example, to find the load balancer object URL for the service `service-web-clusterip` from load balancers list:

```
$ curl http://10.85.188.19:8082/loadbalancers | \
    python -mjson.tool | grep -C4 `service-web-clusterip`
        {
            "fq_name": [
                "default-domain",
                "k8s-ns-user-1",
                "service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc"
            ],
            "href": "http://10.85.188.19:8082/loadbalancer/99fe8ce7-9e75-11e9-b485-0050569e6cfc",
            "uuid": "99fe8ce7-9e75-11e9-b485-0050569e6cfc"
        },
```

Now with one specific load balancer URL, you can pull the specific LB object details:

```
$ curl \
    http://10.85.188.19:8082/loadbalancer/99fe8ce7-9e75-11e9-b485-0050569e6cfc \
    | python -mjson.tool
{
    "loadbalancer": {
        "annotations": {
            "key_value_pair": [
                {
                    "key": "namespace",
                    "value": "ns-user-1"
                },
                {
                    "key": "cluster",
                    "value": "k8s"
                },
                {
                    "key": "kind",
                    "value": "Service"
                },
                {
                    "key": "project",
                    "value": "k8s-ns-user-1"
                },
                {
                    "key": "name",
                    "value": "service-web-clusterip"
                },
                {
                    "key": "owner",
                    "value": "k8s"
                }
            ]
        },
        "display_name": "ns-user-1__service-web-clusterip",
        "fq_name": [
            "default-domain",
            "k8s-ns-user-1",
            "service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc"
        ],
        "href": "http://10.85.188.19:8082/loadbalancer/99fe8ce7-9e75-11e9-b485-0050569e6cfc",
        "id_perms": {
            ...<snipped>...
        },
        "loadbalancer_listener_back_refs": [    #<---
            {
                "attr": null,
                "href": "http://10.85.188.19:8082/loadbalancer-listener/3702fa49-f1ca-4bbb-87d4-
22e1a0dc7e67",
                "to": [
                    "default-domain",
                    "k8s-ns-user-1",
                    "service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc-TCP-8888-3702fa49-
f1ca-4bbb-87d4-22e1a0dc7e67"
                ],
                "uuid": "3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67"
            }
        ],
        "loadbalancer_properties": {
            "admin_state": true,
            "operating_status": "ONLINE",
            "provisioning_status": "ACTIVE",
            "status": null,
            "vip_address": "10.105.139.153",    #<---
            "vip_subnet_id": null
        },
        "loadbalancer_provider": "native",      #<---
        "name": "service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc",
        "parent_href": "http://10.85.188.19:8082/project/86bf8810-ad4d-45d1-aa6b-15c74d5f7809",
        "parent_type": "project",
        "parent_uuid": "86bf8810-ad4d-45d1-aa6b-15c74d5f7809",
        "perms2": {
            ...<snipped>...
        },
        "service_appliance_set_refs": [
            ...<snipped>...
        ],
        "uuid": "99fe8ce7-9e75-11e9-b485-0050569e6cfc",
        "virtual_machine_interface_refs": [
            {
                "attr": null,
                "href": "http://10.85.188.19:8082/virtual-machine-interface/8d64176c-9fc7-491a-a44d-
430e187d6b52",
                "to": [
                    "default-domain",
                    "k8s-ns-user-1",
                    "k8s__Service__service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc"
                ],
                "uuid": "8d64176c-9fc7-491a-a44d-430e187d6b52"
            }
        ]
    }
}
```

The output is very extensive and includes many details that are not of interest to us at the moment, except for a few details worth mentioning:

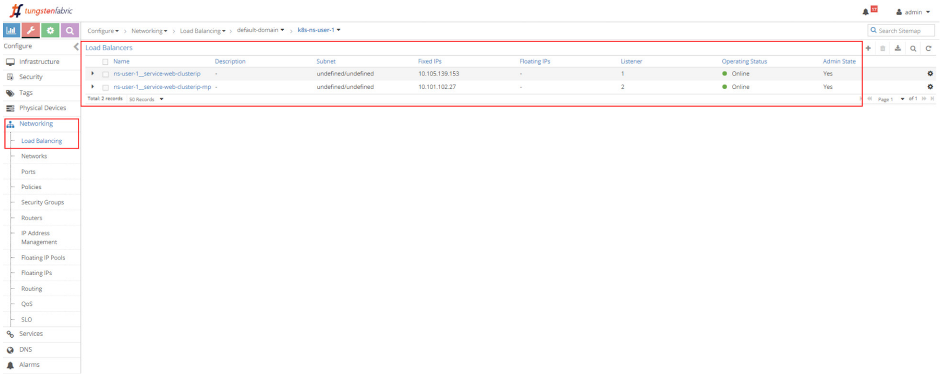
- In loadbalancer_properties, the LB use service IP as its VIP.
- The LB is connected to a listener by a reference.
- The `loadbalancer_provider` attribute is `native`, a new extension to implement Layer 4 (transport layer) ECMP for Kubernetes service.

In the rest of the exploration of LB and its related objects, let's use the legacy Contrail UI.

TIP You can also easily use the new Contrail Command UI to do the same.

For each service there is an LB object, in Figure 5.3 the screen capture shows two LB objects:

- `ns-user-1-service-web-clusterip`
- `ns-user-1-service-web-clusterip-mp`



Load Balancers

Name	Description	Subnet	Fixed IPs	Floating IPs	Listener	Operating Status	Admin State
ns-user-1__service-web-clusterip	-	undefined/undefined	10.105.139.153	-	1	● Online	Yes
ns-user-1__service-web-clusterip-mp	-	undefined/undefined	10.101.102.27	-	2	● Online	Yes

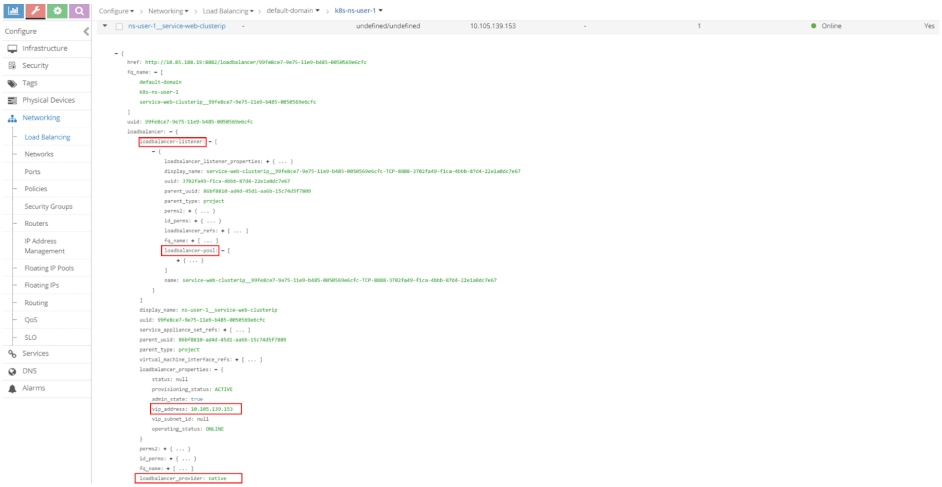
Total: 2 records 50 Records Page 1 of 1

Figure 5.3 Load Balancer Object List

This indicates two services were created. The service load balancer object's name is composed by connecting the NS name with the service name, hence you can tell the names of the two services:

- `service-web-clusterip`
- `service-web-clusterip-mp`

Click on the small triangle icon in the left of the first load balancer object `ns-user-1-service-web-clusterip` to expand it, then click on `advanced json view` icon on the right, and you will see detailed information similar to what you've seen in `curl` capture. For example, the `VIP`, `loadbalancer_provider`, `loadbalancer_listener` object that refers it, etc.



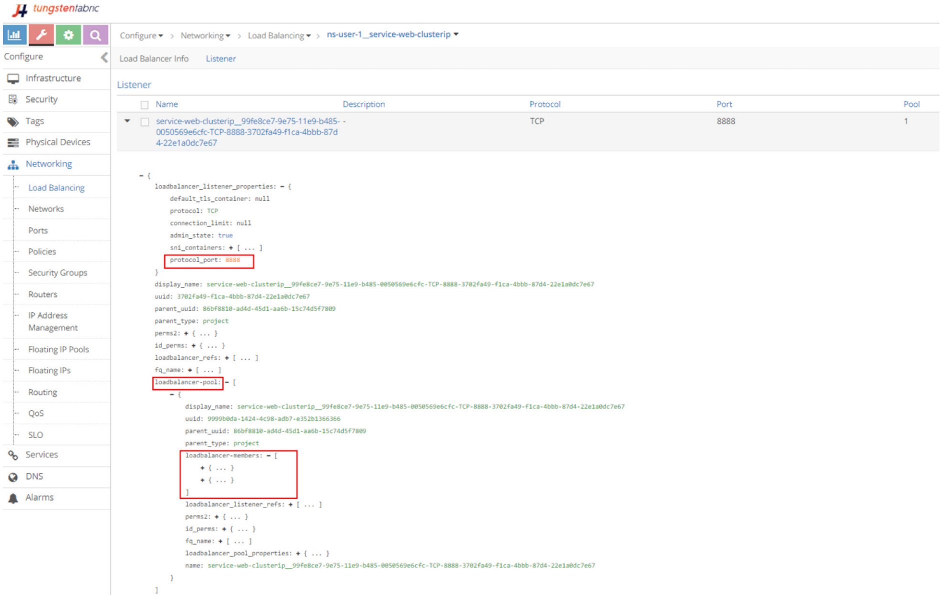
```
▾ □ ns-user-1__service-web-clusterip   -   undefined/undefined   10.105.139.153   -   1   ● Online   Yes

- {
    href: http://10.85.188.19:8082/loadbalancer/99fe8ce7-9e75-11e9-b485-0050569e6cfc
    fq_name: - [
        default-domain
        k8s-ns-user-1
        service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc
    ]
    uuid: 99fe8ce7-9e75-11e9-b485-0050569e6cfc
    loadbalancer: - {
        loadbalancer-listener: - [
            - {
                loadbalancer_listener_properties: + { ... }
                display_name: service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
                uuid: 3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
                parent_uuid: 86bf8810-ad4d-45d1-aa6b-15c74d5f7809
                parent_type: project
                perms2: + { ... }
                id_perms: + { ... }
                loadbalancer_refs: + [ ... ]
                fq_name: + [ ... ]
                loadbalancer-pool: - [
                    + { ... }
                ]
                name: service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
            }
        ]
        display_name: ns-user-1__service-web-clusterip
        uuid: 99fe8ce7-9e75-11e9-b485-0050569e6cfc
        service_appliance_set_refs: + [ ... ]
        parent_uuid: 86bf8810-ad4d-45d1-aa6b-15c74d5f7809
        parent_type: project
        virtual_machine_interface_refs: + [ ... ]
        loadbalancer_properties: - {
            status: null
            provisioning_status: ACTIVE
            admin_state: true
            vip_address: 10.105.139.153
            vip_subnet_id: null
            operating_status: ONLINE
        }
        perms2: + { ... }
        id_perms: + { ... }
        fq_name: + [ ... ]
        loadbalancer_provider: native
```

Figure 5.4 Contrail Load Balancer

From here you can keep expanding the `loadbalancer_listener` object by clicking the + character to see the detail as shown in Figure 5.4. You'll then see a `loadbalancer_pool`; expand it again and you will see `member`. You can repeat this process to explore the object data.

Listener

Click on the LB name and select *listener*, then expand it and display the details with JSON format and you will get the listener details. The listener is listening on service port 8888, and it is referenced by a pool in Figure 5.5.



Listener

	Name	Description	Protocol	Port	Pool
▾ □	service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67	-	TCP	8888	1

```
- {
    loadbalancer_listener_properties: - {
        default_tls_container: null
        protocol: TCP
        connection_limit: null
        admin_state: true
        sni_containers: + [ ... ]
        protocol_port: 8888
    }
    display_name: service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
    uuid: 3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
    parent_uuid: 86bf8810-ad4d-45d1-aa6b-15c74d5f7809
    parent_type: project
    perms2: + { ... }
    id_perms: + { ... }
    loadbalancer_refs: + [ ... ]
    fq_name: + [ ... ]
    loadbalancer-pool: - [
        - {
            display_name: service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
            uuid: 9999b0da-1424-4c98-adb7-e352b1366366
            parent_uuid: 86bf8810-ad4d-45d1-aa6b-15c74d5f7809
            parent_type: project
            loadbalancer-members: - [
                + { ... }
                + { ... }
            ]
            loadbalancer_listener_refs: + [ ... ]
            perms2: + { ... }
            id_perms: + { ... }
            fq_name: + [ ... ]
            loadbalancer_pool_properties: + { ... }
            name: service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
        }
    ]
```

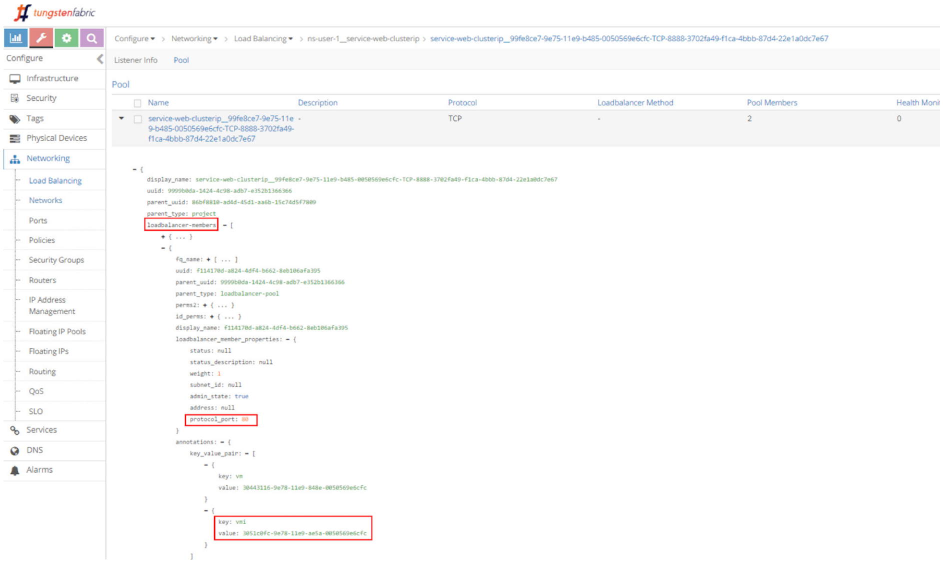
Figure 5.5 *Listener*

TIP In order to see the detailed parameters of an object in JSON format, click the triangle in the left of the load balancer name to expand it, then click on the Advanced JSON view icon </> on the upper right corner in the expanded view. The JSON view is used a lot in this book to explore different Contrail objects.

Pool and Member

Just repeating this explorative process will get you down to the pool and the two `members` in it. The member is with a port of `80`, which maps to the container targetPort in pod as shown in Figures 5.6 and 5.7.

Next we'll examine the vRouter VRF table for the pod to show Contrail service load balancer ECMP operation details. To better understand the 1-to-N mapping between load balancer and listener shown in the load balancer object figure, we'll also give an example of a multiple port service in our setup. We'll conclude the clusterIP service section by inspecting the vRouter flow table to illustrate the service packet workflow.

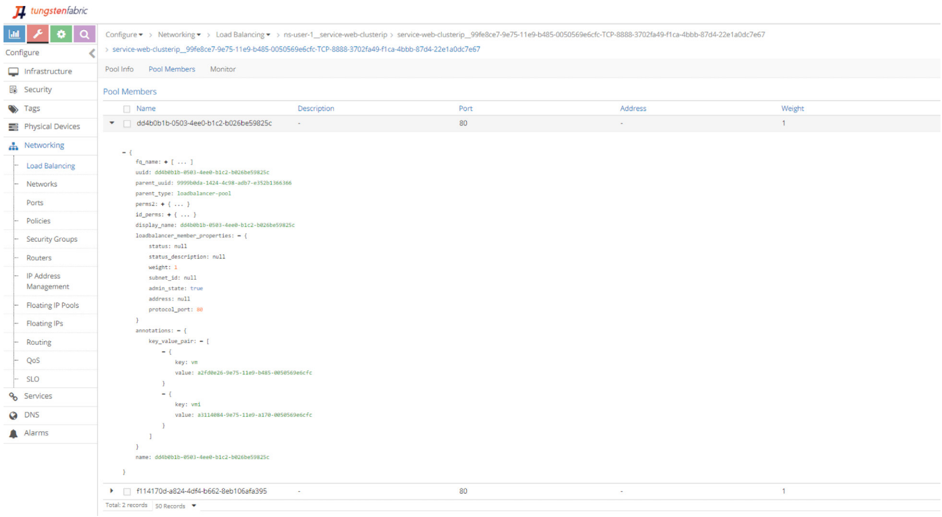


Pool

□	Name	Description	Protocol	Loadbalancer Method	Pool Members	Health Moni
□	service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67	-	TCP	-	2	0

```
- {
    display_name: service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
    uuid: 9999b0da-1424-4c98-adb7-e352b1366366
    parent_uuid: 86bf8810-ad4d-45d1-aa6b-15c74d5f7809
    parent_type: project
    loadbalancer-members - [
        + { ... }
        - {
            fq_name: + [ ... ]
            uuid: f114170d-a824-4df4-b662-8eb106afa395
            parent_uuid: 9999b0da-1424-4c98-adb7-e352b1366366
            parent_type: loadbalancer-pool
            perms2: + { ... }
            id_perms: + { ... }
            display_name: f114170d-a824-4df4-b662-8eb106afa395
            loadbalancer_member_properties: - {
                status: null
                status_description: null
                weight: 1
                subnet_id: null
                admin_state: true
                address: null
                protocol_port: 80
            }
            annotations: - {
                key_value_pair: - [
                    - {
                        key: vm
                        value: 30443116-9e78-11e9-848e-0050569e6cfc
                    }
                    - {
                        key: vmi
                        value: 3051c0fc-9e78-11e9-ae5a-0050569e6cfc
                    }
                ]
```

Figure 5.6 Pool


tungstenfabric
Configure ▾ › Networking ▾ › Load Balancing ▾ › ns-user-1__service-web-clusterip › service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67 › service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
Configure
Infrastructure
Security
Tags
Physical Devices
Networking
Load Balancing
Networks
Ports
Policies
Security Groups
Routers
IP Address Management
Floating IP Pools
Floating IPs
Routing
QoS
SLO
Services
DNS
Alarms
Pool Info
Pool Members
Monitor

Pool Members

□	Name	Description	Port	Address	Weight
□	dd4b0b1b-0503-4ee0-b1c2-b026be59825c	-	80	-	1

```
- {
    fq_name: + [ ... ]
    uuid: dd4b0b1b-0503-4ee0-b1c2-b026be59825c
    parent_uuid: 9999b0da-1424-4c98-adb7-e352b1366366
    parent_type: loadbalancer-pool
    perms2: + { ... }
    id_perms: + { ... }
    display_name: dd4b0b1b-0503-4ee0-b1c2-b026be59825c
    loadbalancer_member_properties: - {
        status: null
        status_description: null
        weight: 1
        subnet_id: null
        admin_state: true
        address: null
        protocol_port: 80
    }
    annotations: - {
        key_value_pair: - [
            - {
                key: vm
                value: a2fd0e26-9e75-11e9-b485-0050569e6cfc
            }
            - {
                key: vmi
                value: a3114084-9e75-11e9-a170-0050569e6cfc
            }
        ]
    }
    name: dd4b0b1b-0503-4ee0-b1c2-b026be59825c
}
```

□	Name	Description	Port	Address	Weight
□	f114170d-a824-4df4-b662-8eb106afa395	-	80	-	1

Total: 2 records 50 Records ▾

Figure 5.7 Members

Contrail Service Setup

Before starting our investigation, let’s look at the setup. In this book we built a setup including the following devices, most of our case studies are based on it:

- one centos server running as k8s `master` and Contrail Controllers
- two centos servers, each running as a k8s `node` and Contrail vRouter
- one Juniper QFX switch running as the underlay leaf
- one Juniper MX router running as a gateway router, or a spine
- one centos server running as an Internet host machine

Figure 5.8 depicts the setup.

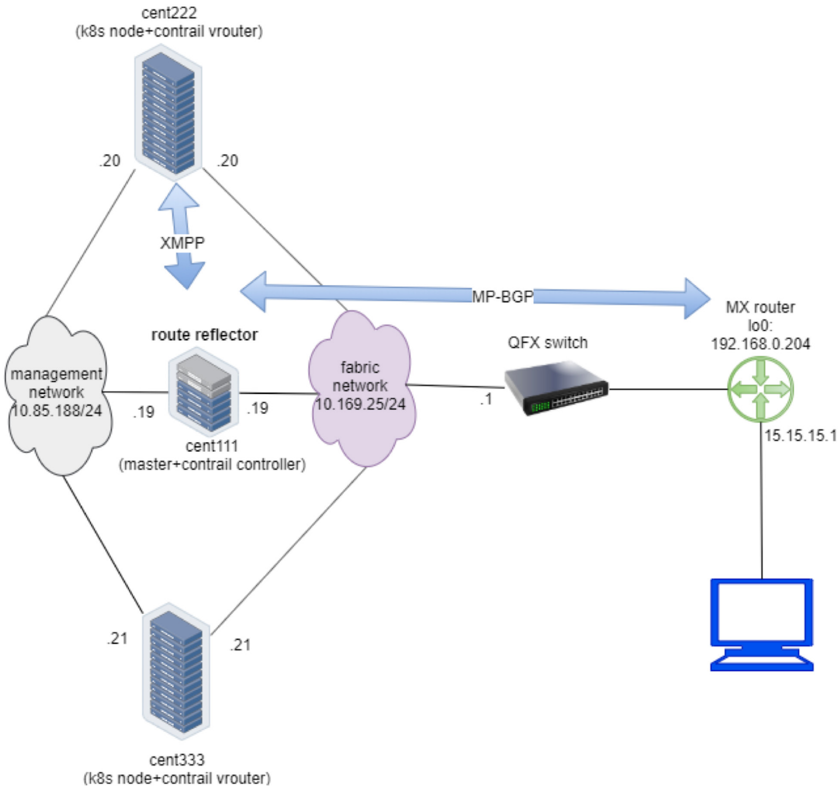


Figure 5.8 Contrail Service Setup

NOTE To minimize the resource utilization, all servers are actually centos VMs created by a VMware ESXI hypervisor running in one physical HP server. This is also the same testbed for ingress. The Appendix of this book has details about the setup.

Contrail ClusterIP Service

Chapter 3 demonstrated how to create and verify a clusterIP service. This section revisits the lab to look at some important details about Contrail's specific implementations. Let's continue on and add a few more tests to illustrate the Contrail service load balancer implementation details.

ClusterIP as Floating IP

Here is the YAML file used to create a `clusterIP` service:

```
#service-web-clusterip.yaml
apiVersion: v1
kind: Service
metadata:
  name: service-web-clusterip
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver
```

And here's a review of what we got from the service lab in Chapter 3:

```
$ kubectl get svc -o wide
NAME                   TYPE       CLUSTER-IP      EXTERNAL-IP  PORT(S)   AGE  SELECTOR
service-web-clusterip  ClusterIP  10.105.139.153  <none>       8888/TCP  45m  app=webserver
$ kubectl get pod -o wide --show-labels
NAME                        READY  STATUS   ...  IP             NODE     ...  LABELS
client                      1/1    Running  ...  10.47.255.237  cent222  ...  app=client
webserver-846c9ccb8b-g27kg  1/1    Running  ...  10.47.255.238  cent333  ...  app=webserver
```

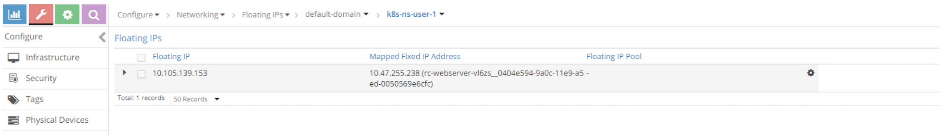
You can see one service is created, with one pod running as its backend. The label in the pod matches the SELECTOR in service. The pod name also indicates this is a deploy-generated pod. Later we can scale the deploy for the ECMP case study, but for now let's stick to one pod and examine the clusterIP implementation details.

In Contrail, a `ClusterIP` is essentially implemented in the form of a floating IP. Once a service is created, a floating IP will be allocated from the service subnet and associated to all the back-end pod VMIs to form the ECMP load balancing. Now all the back-end pods can be reached via cluserIP (along with the pod IP). This clusterIP (floating IP) is acting as a VIP to the client pods inside of the cluster.

TIP Why does Contrail choose floating IP to implement clusterIP? In the previous section, you learned that Contrail does NAT for floating IP and service also needs NAT. So, it is natural to use the floating IP for clusterIP.

For load balancer type of service, Contrail will allocate a second floating IP - the EXTERNAL-IP as the VIP, and the external VIP is advertised outside of the cluster through the gateway router. You will get more details about these later.

From the UI you can see the automatically allocated floating IP as clusterIP in Figure 5.9.

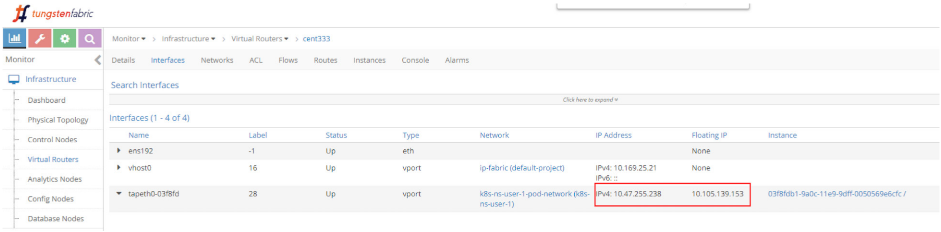

Configure ▾ › Networking ▾ › Floating IPs ▾ › default-domain ▾ › k8s-ns-user-1 ▾
Configure
Infrastructure
Security
Tags
Physical Devices
Floating IPs

	Floating IP	Mapped Fixed IP Address	Floating IP Pool
▸ □	10.105.139.153	10.47.255.238 (rc-webserver-vl6zs__0404e594-9a0c-11e9-a5 - ed-0050569e6cfc)	

Total: 1 records 50 Records ▾

Figure 5.9 ClusterIP as Floating IP

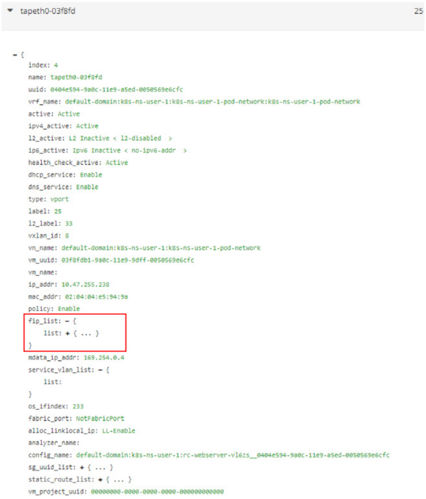
And the floating IP is also associated with the pod VMI and pod IP, in this case the VMI is representing the pod interface shown in Figure 5.10.



Name	Label	Status	Type	Network	IP Address	Floating IP	Instance
▸ ens192	-1	Up	eth			None	
▸ vhost0	16	Up	vport	ip-fabric (default-project)	IPv4: 10.169.25.21 IPv6: ::	None	
▾ tapeth0-03f8fd	28	Up	vport	k8s-ns-user-1-pod-network (k8s-ns-user-1)	IPv4: 10.47.255.238	10.105.139.153	03f8fdb1-9a0c-11e9-9dff-0050569e6cfc /

Figure 5.10 Pod Interface

The interface can be expanded to display more details as in the next screen capture, shown in Figure 5.11.



```
- {
    index: 4
    name: tapeth0-03f8fd
    uuid: 0404e594-9a0c-11e9-a5ed-0050569e6cfc
    vrf_name: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network:k8s-ns-user-1-pod-network
    active: Active
    ipv4_active: Active
    l2_active: L2 Inactive < l2-disabled  >
    ip6_active: Ipv6 Inactive < no-ipv6-addr  >
    health_check_active: Active
    dhcp_service: Enable
    dns_service: Enable
    type: vport
    label: 25
    l2_label: 33
    vxlan_id: 8
    vn_name: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network
    vm_uuid: 03f8fdb1-9a0c-11e9-9dff-0050569e6cfc
    vm_name:
    ip_addr: 10.47.255.238
    mac_addr: 02:04:04:e5:94:9a
    policy: Enable
    fip_list: - {
        list: + { ... }
    }
    mdata_ip_addr: 169.254.0.4
    service_vlan_list: - {
        list:
    }
    os_ifindex: 233
    fabric_port: NotFabricPort
    alloc_linklocal_ip: LL-Enable
    analyzer_name:
    config_name: default-domain:k8s-ns-user-1:rc-webserver-vl6zs__0404e594-9a0c-11e9-a5ed-0050569e6cfc
    sg_uuid_list: + { ... }
    static_route_list: + { ... }
    vm_project_uuid: 00000000-0000-0000-0000-000000000000
```

Figure 5.11 Pod Interface Detail

Expand the `fip_list`, any you'll see the information here:

```
fip_list:  {
    list:  {
        FloatingIpSandeshList:  {
            ip_addr: 10.105.139.153
            vrf_name: default-domain:k8s-ns-user-1:k8s-ns-user-1-service-network:k8s-ns-user-1-
service-network
            installed: Y
            fixed_ip: 10.47.255.238
            direction: ingress
            port_map_enabled: true
            port_map:  {
                list:  {
                    SandeshPortMapping:  {
                    protocol: 6
                    port: 80
                    nat_port: 8888
                    }
                }
            }
        }
    }
}
```

Service/clusterIP/FIP 10.105.139.153 maps to podIP/fixed_ip 10.47.255.238. The `port_map` says that port `8888` is a `nat_port`, `6` is the protocol number so it means protocol TCP. Overall, clusterIP:port `10.105.139.153:8888` will be translated to podIP:targetPort `10.47.255.238:80`, and vice versa.

Now you understand with floating IP representing clusterIP, NAT will happen in service. NAT will be examined again in the flow table.

Scaling Backend Pods

In Chapter 3's clusterIP service example, we created a service and a backend pod. To verify the ECMP, let's increase the replica unit to 2 to generate a second backend pod. This is a more realistic model: each of the pods will now be backing each other up to avoid a single point failure.

Instead of using a YAML file to manually create a new webserver pod, with the Kubernetes spirit in mind, think of `scale` to a `deployment`, as was done earlier in this book. In our service example we've been using a `deployment` object on purpose to spawn our webserver pod:

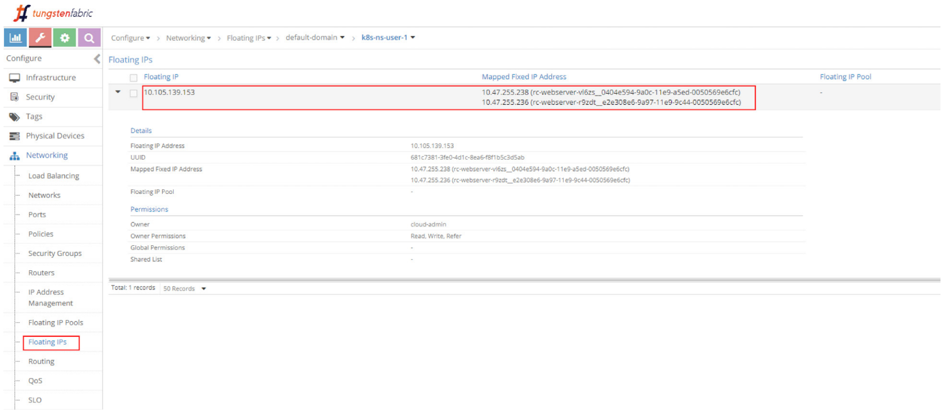
```
$ kubectl scale deployment webserver --replicas=2
deployment.extensions/webserver scaled

$ kubectl get pod -o wide --show-labels
NAME                        READY  STATUS   ... IP             NODE     ... LABELS
client                      1/1    Running  ... 10.47.255.237  cent222  ... app=client
webserver-846c9ccb8b-7btnj  1/1    Running  ... 10.47.255.236  cent222  ... app=webserver
webserver-846c9ccb8b-g27kg  1/1    Running  ... 10.47.255.238  cent333  ... app=webserver

$ kubectl get svc -o wide
NAME                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE   SELECTOR
service-web-clusterip   ClusterIP   10.105.139.153   <none>        8888/TCP   45m   app=webserver
```

Immediately after you create a new webserver pod by scaling the deployment with `replicas 2`, a new pod is launched. You end up having two backend pods, one is running in the same node `cent222` as the client pod, or a local node for client pod; the other one is running in the other node `cent333`, the remote node from client pod's perspective. And the `endpoint` objects get updated to reflect the current set of backend pods behind the `service`.

```
$ kubectl get ep -o wide
NAME             ENDPOINTS                           AGE
service-web-lb   10.47.255.236:80,10.47.255.238:80   20m
```

NOTE Without the `-o wide` option, only the first endpoint will be displayed properly.

Let's check the floating IP again.



Configure ▾ › Networking ▾ › Floating IPs ▾ › default-domain ▾ › k8s-ns-user-1 ▾

Floating IPs

Floating IP	Mapped Fixed IP Address	Floating IP Pool
10.105.139.153	10.47.255.238 (rc-webserver-vl6zs__0404e594-9a0c-11e9-a5ed-0050569e6cfc) 10.47.255.236 (rc-webserver-r9zdt__e2e308e6-9a97-11e9-9c44-0050569e6cfc)	-

Details

Floating IP Address	10.105.139.153
UUID	681c7381-3fe0-4d1c-8ea6-f8f1b5c3d5ab
Mapped Fixed IP Address	10.47.255.238 (rc-webserver-vl6zs__0404e594-9a0c-11e9-a5ed-0050569e6cfc) 10.47.255.236 (rc-webserver-r9zdt__e2e308e6-9a97-11e9-9c44-0050569e6cfc)
Floating IP Pool	-

Permissions

Owner	cloud-admin
Owner Permissions	Read, Write, Refer
Global Permissions	-
Shared List	-

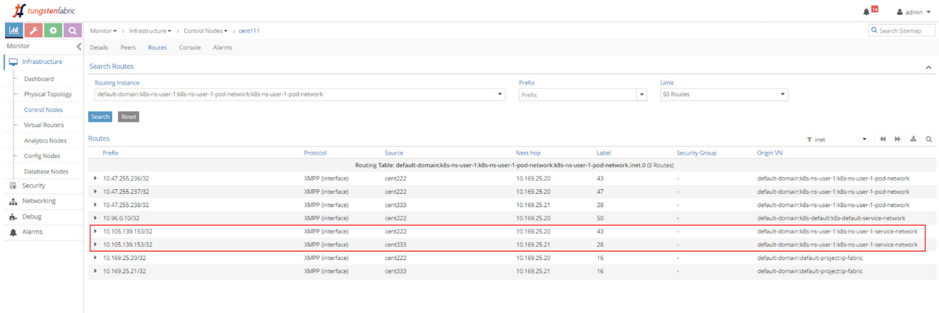
Total: 1 records 50 Records

Figure 5.12 ClusterIP as Floating IP (ECMP)

In Figure 5.12 you can see the same floating IP, but now it is associated with two podIPs, each representing a separate pod.

ECMP Routing Table

First let's examine the ECMP. Let's take a look at the routing table in the controller's routing instance in the screen capture seen in Figure 5.13.



Monitor ▾ › Infrastructure ▾ › Control Nodes ▾ › cent111

Search Routes

Routing Instance: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network:k8s-ns-user-1-pod-network
Prefix: Prefix
Limit: 50 Routes

Routes

Routing Table: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network:k8s-ns-user-1-pod-network.inet.0 (8 Routes)

Prefix	Protocol	Source	Next hop	Label	Security Group	Origin VN
10.47.255.236/32	XMPP (interface)	cent222	10.169.25.20	43	-	default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network
10.47.255.237/32	XMPP (interface)	cent222	10.169.25.20	47	-	default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network
10.47.255.238/32	XMPP (interface)	cent333	10.169.25.21	28	-	default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network
10.96.0.10/32	XMPP (interface)	cent222	10.169.25.20	50	-	default-domain:k8s-default:k8s-default-service-network
10.105.139.153/32	XMPP (interface)	cent222	10.169.25.20	43	-	default-domain:k8s-ns-user-1:k8s-ns-user-1-service-network
10.105.139.153/32	XMPP (interface)	cent333	10.169.25.21	28	-	default-domain:k8s-ns-user-1:k8s-ns-user-1-service-network
10.169.25.20/32	XMPP (interface)	cent222	10.169.25.20	16	-	default-domain:default-project:ip-fabric
10.169.25.21/32	XMPP (interface)	cent333	10.169.25.21	16	-	default-domain:default-project:ip-fabric

Figure 5.13 Control Node Routing Instance Table

The routing instance (RI) has a full name with the following format:

```
<DOMAIN>:<PROJECT>:<VN>:<RI>
```

In most cases the RI inherits the same name from its virtual network, so in this case the full IPv4 routing table has this name:

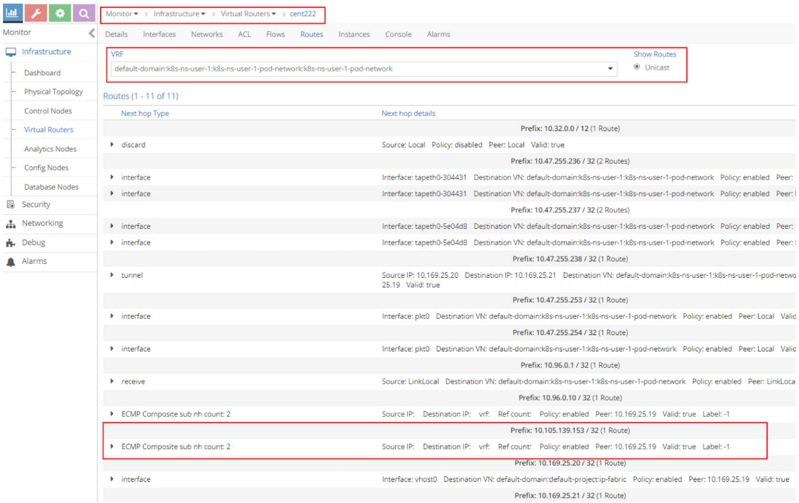
```
default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network:k8s-ns-user-1-pod-network.inet.0
```

The .inet.0 indicates the routing table type is unicast IPv4. There are many other tables that are not of interest to us right now.

Two routing entries with the same exact prefixes of the clusterIP show up in the routing table, with two different next hops, each pointing to a different node. This gives us a hint about the route propagation process: both nodes (compute) have advertised the same clusterIP toward the master (Contrail Controller), to indicate the running backend pods are present in it. This route propagation is via XMPP. The master (Contrail Controller) then reflects the routes to all the other compute nodes.

Compute Node Perspective

Next, starting from the client pod node `cent222`, let's look at the pod's VRF table to understand how the packets are forwarded towards the backend pods in the screen capture in Figure 5.14.



VRF
default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network:k8s-ns-user-1-pod-network ▾

● Unicast

Routes (1 - 11 of 11)

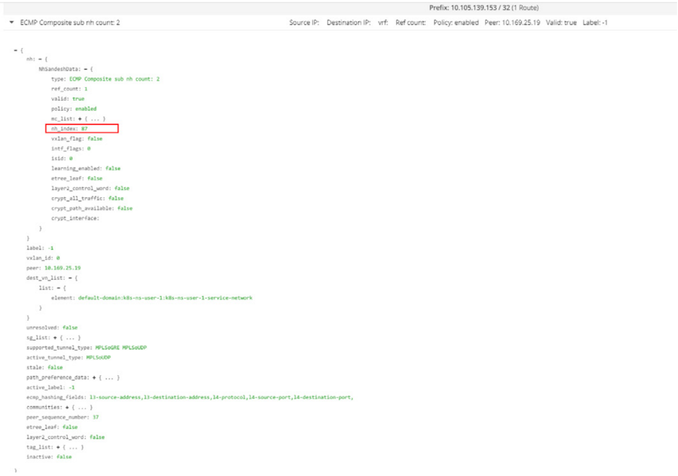
```
Next hop Type                   Next hop details
                                Prefix: 10.32.0.0 / 12 (1 Route)
discard                         Source: Local   Policy: disabled   Peer: Local   Valid: true
                                Prefix: 10.47.255.236 / 32 (2 Routes)
interface                       Interface: tapeth0-304431   Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network   Policy: enabled   Peer:
interface                       Interface: tapeth0-304431   Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network   Policy: enabled   Peer:
                                Prefix: 10.47.255.237 / 32 (2 Routes)
interface                       Interface: tapeth0-5e04d8   Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network   Policy: enabled   Peer:
interface                       Interface: tapeth0-5e04d8   Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network   Policy: enabled   Peer:
                                Prefix: 10.47.255.238 / 32 (1 Route)
tunnel                          Source IP: 10.169.25.20   Destination IP: 10.169.25.21   Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-netwo
                                25.19   Valid: true
                                Prefix: 10.47.255.253 / 32 (1 Route)
interface                       Interface: pkt0   Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network   Policy: enabled   Peer: Local   Valid
                                Prefix: 10.47.255.254 / 32 (1 Route)
interface                       Interface: pkt0   Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network   Policy: enabled   Peer: Local   Valid
                                Prefix: 10.96.0.1 / 32 (1 Route)
receive                         Source: LinkLocal   Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network   Policy: enabled   Peer: LinkLoca
                                Prefix: 10.96.0.10 / 32 (1 Route)
ECMP Composite sub nh count: 2  Source IP:   Destination IP:   vrf:   Ref count:   Policy: enabled   Peer: 10.169.25.19   Valid: true   Label: -1
                                Prefix: 10.105.139.153 / 32 (1 Route)
ECMP Composite sub nh count: 2  Source IP:   Destination IP:   vrf:   Ref count:   Policy: enabled   Peer: 10.169.25.19   Valid: true   Label: -1
                                Prefix: 10.169.25.20 / 32 (1 Route)
interface                       Interface: vhost0   Destination VN: default-domain:default-project:ip-fabric   Policy: enabled   Peer: 10.169.25.19   Valid: true
                                Prefix: 10.169.25.21 / 32 (1 Route)
```

Figure 5.14 vRouter VRF Table

The most important part of the screenshot in FIgure 5.14 is the routing entry `Prefix: 10.105.139.153 / 32 (1 Route)`, as it is our clusterIP address. Underneath the prefix there is the statement `ECMP Composite sub nh count: 2`. This indicates the prefix has multiple possible next hops to reach.

Now, expand the ECMP statement by clicking on the small triangle icon in the left and you will be given a lot more details about this prefix as shown in the next screen capture in Figure 5.15.

The most import of all the details in this output is that of our focus, `nh_index: 87`, which is the next hop ID (`NHID`) for the clusterIP prefix. From the vRouter agent Docker, you can further resolve the Composite NHID to the `sub-NHs`, which are the member next hops under the Composite next hop.



Prefix: 10.105.139.153 / 32 (1 Route)

ECMP Composite sub nh count: 2 Source IP: Destination IP: vrf: Ref count: Policy: enabled Peer: 10.169.25.19 Valid: true Label: -1

```
- {
    nh: - {
        NhSandeshData: - {
            type: ECMP Composite sub nh count: 2
            ref_count: 1
            valid: true
            policy: enabled
            mc_list: + { ... }
            nh_index: 87
            vxlan_flag: false
            intf_flags: 0
            isid: 0
            learning_enabled: false
            etree_leaf: false
            layer2_control_word: false
            crypt_all_traffic: false
            crypt_path_available: false
            crypt_interface:
        }
    }
    label: -1
    vxlan_id: 0
    peer: 10.169.25.19
    dest_vn_list: - {
        list: - {
            element: default-domain:k8s-ns-user-1:k8s-ns-user-1-service-network
        }
    }
    unresolved: false
    sg_list: + { ... }
    supported_tunnel_type: MPLSoGRE MPLSoUDP
    active_tunnel_type: MPLSoUDP
    stale: false
    path_preference_data: + { ... }
    active_label: -1
    ecmp_hashing_fields: l3-source-address,l3-destination-address,l4-protocol,l4-source-port,l4-destination-port,
    communities: + { ... }
    peer_sequence_number: 37
    etree_leaf: false
    layer2_control_word: false
    tag_list: + { ... }
    inactive: false
}
```

Figure 5.15 vRouter ECMP Next Hop

TIP Don't forget to execute the vRouter commands from the vRouter Docker container. Doing it directly from the host may not work:

```
[2019-07-04 12:42:06]root@cent222:~
$ docker exec -it vrouter_vrouter-agent_1 nh --get 87
Id:87         Type:Composite      Fmly: AF_INET  Rid:0  Ref_cnt:2         Vrf:2
              Flags:Valid, Policy, Ecmp, Etree Root,
              Valid Hash Key Parameters: Proto,SrcIP,SrcPort,DstIp,DstPort
              Sub NH(label): 51(43) 37(28)              #<---

Id:51         Type:Tunnel         Fmly: AF_INET  Rid:0  Ref_cnt:18        Vrf:0
              Flags:Valid, MPLSoUDP, Etree Root,        #<---
              Oif:0 Len:14 Data:00 50 56 9e e6 66 00 50 56 9e 62 25 08 00
              Sip:10.169.25.20 Dip:10.169.25.21

Id:37         Type:Encap          Fmly: AF_INET  Rid:0  Ref_cnt:5         Vrf:2
              Flags:Valid, Etree Root,
              EncapFmly:0806 Oif:8 Len:14               #<---
              Encap Data: 02 30 51 c0 fc 9e 00 00 5e 00 01 00 08 00
```

Some important information to highlight from this output:

- NHID 87 is an ECMP composite next hop.
- The ECMP next hop contains two sub-next hops: next hop 43 and next hop 28, each represents a separate path towards the backend pods.
- Next hop 51 represents a MPLSoUDP tunnel toward backend pod in the remote node, the tunnel is established from current node `cent222`, with source IP being local fabric IP `10.169.25.20`, to the other node `cent333` whose fabric IP is `10.169.25.21`. If you recall where our two backend pods are located, this is the forwarding path between the two nodes.

- Next hop 37 represents a local path, towards vif 0/8 (`Oif:8`), which is the local backend pod's interface.

To resolve the vRouter `vif` interface, use the `vif --get 8` command:

```
$ vif --get 8
Vrouter Interface Table
......
vif0/8      OS: tapeth0-304431
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:10.47.255.236  #<---
            Vrf:2 Mcast Vrf:2 Flags:PL3DEr QOS:-1 Ref:6
            RX packets:455  bytes:19110 errors:0
            TX packets:710  bytes:29820 errors:0
            Drops:455
```

The output displays the corresponding local pod interface's name, IP, etc.

ClusterIP Service Workflow

The clusterIP service's load balancer ECMP workflow is illustrated in Figure 5.16.

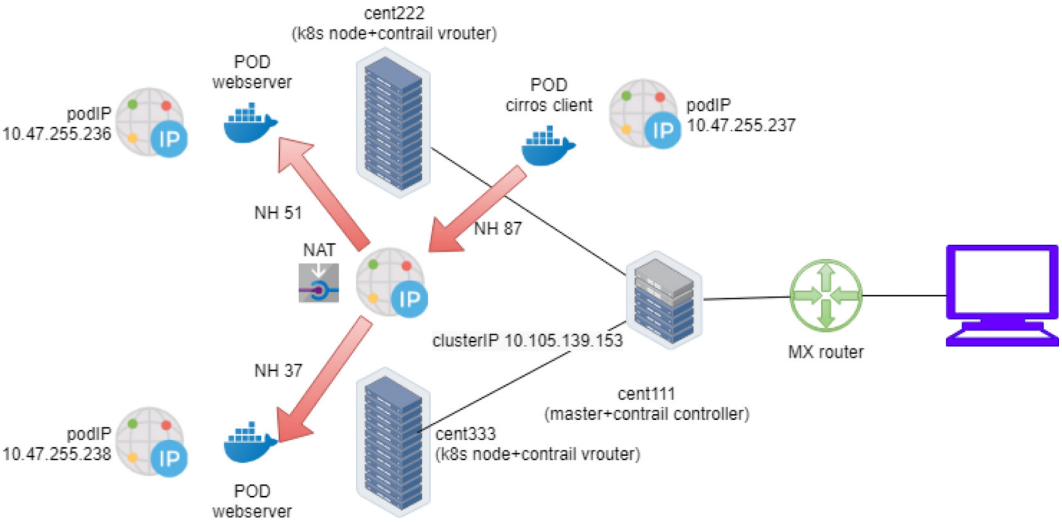


Figure 5.16 *Contrail ClusterIP Service Load Balancer ECMP Forwarding*

This is what happened in the forwarding plane:

- A pod `client` located in node `cent222` needs to access a service `service-web-clusterip`. It sends a packet towards the service's clusterIP `10.105.139.153` and port `8888`.

- The pod `client` sends the packet to node `cent222` vRouter based on the default route.
- The vRouter on node `cent222` gets the packet, checks its corresponding VRF table, gets a Composite next hop ID `87`, which resolves to two sub-next hops `51` and `37`, representing a remote and local backend pod, respectively. This indicates ECMP.
- The vRouter on node `cent222` starts to forward the packet to one of the pods based on its ECMP algorithm. Suppose the remote backend pod is selected, the packet will be sent through the MPLSoUDP tunnel to the remote pod on node `cent333`, after establishing the flow in the flow table. All subsequent packets belonging to the same flow will follow this same path. The same applies to the local path towards the local backend pod.

Multiple Port Service

You should now understand how the service Layer 4 ECMP and the LB objects in the lab work. Figure 5.17 shows the LB and relevant objects, and you can see that one LB may be having two or more LB listeners. Each listener has an individual backend pool that has one or multiple member(s).

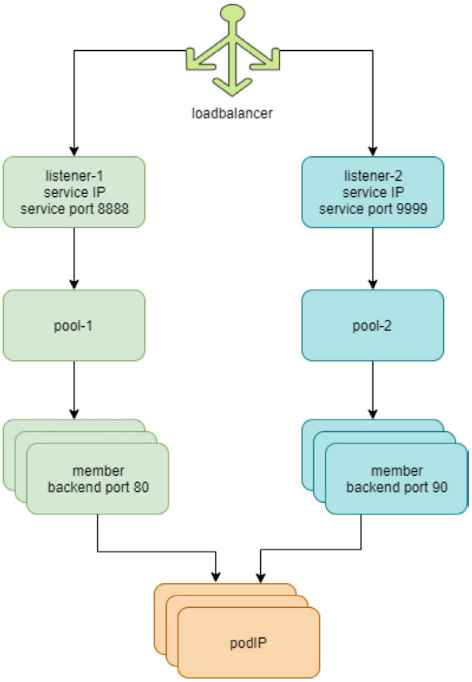


Figure 5.17 Service Load Balancer

In Kubernetes, this 1:N mapping between load balancer and listeners indicates a `multiple port service`, one service with multiple ports. Let's look at the YAML file of it: *svc/service-web-clusterip-mp.yaml*:

```
apiVersion: v1
kind: Service
metadata:
  name: service-web-clusterip-mp
spec:
  ports:
  - name: port1
    port: 8888
    targetPort: 80
  - name: port2         #<---
    port: 9999
    targetPort: 90
  selector:
    app: webserver
```

What has been added is another item in the `ports` list: a new service port `9999` that maps to the container's `targetPort 90`. Now, with two port mappings, you have to give each port a name, say, `port1` and `port2`, respectively.

NOTE Without a port `name` the multiple ports' YAML file won't work.

Now apply the YAML file. A new service `service-web-clusterip-mp` with two ports is created:

```
$ kubectl apply -f svc/service-web-clusterip-mp.yaml
service/service-web-clusterip-mp created

$ kubectl get svc
NAME                      TYPE       CLUSTER-IP      EXTERNAL-IP  PORT(S)            AGE
service-web-clusterip     ClusterIP  10.105.139.153  <none>       8888/TCP           3h8m
service-web-clusterip-mp  ClusterIP  10.101.102.27   <none>       8888/TCP,9999/TCP  4s

$ kubectl get ep
NAME                       ENDPOINTS                           AGE
service-web-clusterip      10.47.255.238:80                    4h18m
service-web-clusterip-mp   10.47.255.238:80,10.47.255.238:90   69m
```

NOTE To simplify the case study, the backend deployment's replicas number has been scaled down to one.

Everything looks okay, doesn't it? The new service comes up with two service ports exposed, `8888`, the old one we've tested in previous examples, and the new `9999` port, should work equally well. But it turns out that is not the case. Let's investigate.

Service port 8888 works:

```
$ kubectl exec -it client -- curl 10.101.102.27:8888 | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.238
                         Hostname = webserver-846c9ccb8b-g27kg
```

Service port 9999 doesn't work:

```
$ kubectl exec -it client -- curl 10.101.102.27:9999 | w3m -T text/html | cat
command terminated with exit code 7
curl: (7) Failed to connect to 10.101.102.27 port 9999: Connection refused
```

The request towards port 9999 is rejected. The reason is the `targetPort` is not running in the pod container, so there is no way to get a response from it:

```
$ kubectl exec -it webserver-846c9ccb8b-g27kg -- netstat -lnap
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      1/python
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State         I-Node   PID/Program name    Path
```

The `readinessProbe` introduced in Chapter 3 is the official Kubernetes tool to detect this situation, so in case the pod is not ready, it will be restarted and you will catch the events.

To resolve this let's start a new server in the pod to listen on a new port `90`. One of the easiest ways to start a HTTP server is to use the `SimpleHTTPServer` module that comes with the `python` package. In this test we only need to set its listening port to `90` (the default value is 8080):

```
$ kubectl exec -it webserver-846c9ccb8b-g27kg -- python -m SimpleHTTPServer 90
Serving HTTP on 0.0.0.0 port 90 ...
```

The `targetPort` is on. Now you can again start the request towards service port `9999` from the `cirros` pod. This time it succeeds and gets the returned webpage from Python's SimpleHTTPServer:

```
$ kubectl exec -it client -- curl 10.101.102.27:9999 | w3m -T text/html | cat

Directory listing for /
 ━━━━━━━━━━━━━━━━━━━━━
  • app.py
  • Dockerfile
  • file.txt
  • requirements.txt
  • static/
 ━━━━━━━━━━━━━━━━━━━━━
```

Next, for each incoming request, the `SimpleHTTPServer` logs one line of output with an IP address showing where the request came from. In this case, the request is coming from the `client` pod with the IP address: `10.47.255.237`:

```
10.47.255.237 - - [04/Jul/2019 23:49:44] "GET / HTTP/1.1" 200 -
```

Contrail Flow Table

So far, we've tested the clusterIP service and we've seen client requests are sent towards the service IP. In Contrail, `vrouter` is the module that does all of the packet forwarding. When the `vrouter` gets a packet from the client pod, it looks up the corresponding VRF table in the vRouter module for the client pod (`client`), gets the next hop, and resolves the correct egress interface and proper encapsulation. So far, the client and backend pods are in two different nodes, and the source `vrouter` decides the packets needed to be sent in the MPLSoUDP tunnel, towards the node where the backend pod is running. What interests us the most?

- How are the service IP and backend pod IP translated to each other?
- Is there a way to capture and see the two IPs in a flow, before and after the translations, for comparison purposes?

The most straightforward method you would think of is to capture the packets, decode, and then see the results. Doing that, however, may not be as easy as what you expect. First you need to capture the packet at different places:

- At the pod interface, this is after the address is translated, and that's easy.
- At the fabric interface, this is before packet is translated and reaches the pod interface. Here the packets are with MPLSoUDP encapsulation since data plane packets are tunneled between nodes.

Then you need to copy the pcap file out and load with Wireshark to decode. You probably also need to configure Wireshark to recognize the MPLSoUDP encapsulation.

An easier way to do this is to check the vRouter flow table, which records IP and port details about a traffic flow. Let's test it by preparing a big file, `file.txt,` in the backend webserver pod and try to download it from the client pod.

TIP You may wonder: in order to trigger a flow why we don't simply use the same `curl` test to pull the webpage? That's what we did in an early test. In theory, that is fine. The only problem is that the TCP flow follows the TCP session. In our previous test with `curl`, the TCP session starts and stops immediately after the webpage is retrieved, then the vRouter clears the flow right away. You won't be fast enough to capture the flow table at the right moment. Instead, downloading a big file will hold the TCP session – as long as the file transfer is ongoing the session will remain – and you can take time to investigate the flow. Later on, the Ingress section will demonstrate a different method with a one-line shell script.

So, in the `client` pod `curl` URL, instead of just giving the root path `/` to list the files in folder, let's try to pull the file: `file.txt`:

```
$ kubectl exec -it client -- curl 10.101.102.27:9999/file.txt
```

And in the server pod we see the log indicating the file downloading starts:

```
10.47.255.237 - - [05/Jul/2019 00:41:21] "GET /file.txt HTTP/1.1" 200 -
```

Now, with the file transfer going on, there's enough time to collect the flow table from both the client and server nodes, in the vRouter container:

Client node flow table:

```
(vrouter-agent)[root@cent222 /]$ flow --match 10.47.255.237
Flow table(size 80609280, entries 629760)

Entries: Created 1361 Added 1361 Deleted 442 Changed 443Processed 1361 Used Overflow entries 0
(Created Flows/CPU: 305 342 371 343)(oflows 0)

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

Listing flows matching ([10.47.255.237]:*)

    Index                    Source:Port/Destination:Port                      Proto(V)
 ----------------------------------------------------------------------------------
    40100<=>340544       10.47.255.237:42332                                 6 (3)
                         10.101.102.279999
(Gen: 1, K(nh):59, Action:F, Flags:, TCP:SSrEEr, QOS:-1, S(nh):59,  Stats:7878/520046,
 SPort 65053, TTL 0, Sinfo 6.0.0.0)

   340544<=>40100        10.101.102.279999                                  6 (3)
                         10.47.255.237:42332
(Gen: 1, K(nh):59, Action:F, Flags:, TCP:SSrEEr, QOS:-1, S(nh):68,  Stats:142894/205180194,
 SPort 63010, TTL 0, Sinfo 10.169.25.21)
```

Highlights in this output are:

- The `client` pod starts the TCP connection from its pod IP `10.47.255.237` and a random source port, towards the service IP `10.101.102.27` and server port `9999`.
- The flow TCP flag `SSrEEr` indicates the session is established bi-directionally.
- The `Action: F` means forwarding. Note that there is no special processing like `NAT` happening here.

NOTE When using a filter such as `--match 15.15.15.2` only flow entries with Internet Host IPs are displayed.

We can conclude, from the client node's perspective, that it only sees the service IP and is not aware of any backend pod IP at all.

Let's look at the server node flow table in the server node vRouter Docker container:

```
(vrouter-agent)[root@cent333 /]$ flow --match 10.47.255.237
Flow table(size 80609280, entries 629760)

Entries: Created 1116 Added 1116 Deleted 422 Changed 422Processed 1116 Used Overflow entries 0
(Created Flows/CPU: 377 319 76 344)(oflows 0)

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

Listing flows matching ([10.47.255.237]:*)

    Index                Source:Port/Destination:Port                      Proto(V)
 ----------------------------------------------------------------------------------
   238980<=>424192       10.47.255.238:90                                    6 (2->3)
                         10.47.255.237:42332
(Gen: 1, K(nh):24, Action:N(SPs), Flags:, TCP:SSrEEr, QOS:-1, S(nh):24,
 Stats:8448/202185290,  SPort 62581, TTL 0, Sinfo 3.0.0.0)

   424192<=>238980       10.47.255.237:42332                                 6 (2->2)
                         10.101.102.279999
(Gen: 1, K(nh):24, Action:N(DPd), Flags:, TCP:SSrEEr, QOS:-1, S(nh):26,
 Stats:8067/419582,  SPort 51018, TTL 0, Sinfo 10.169.25.20)
```

Look at the second flow entry first – the IPs look the same as the one we just saw in the client side capture. Traffic lands the vRouter fabric interface from the remote `client` pod node, across the MPLSoUDP tunnel. Destination IP and the port are service IP and the service port, respectively. Nothing special here.

However, the flow `Action` is now set to `N(DPd)`, not `F`. According to the header lines in the `flow` command output, this means NAT, or specifically, `DNAT` (Destination address translation) with `DPAT` (Destination port translation) – both the service IP and service port are translated to the backend pod IP and port.

Now look at the first flow entry. The source IP `10.47.255.238` is the backend pod IP and the source port is the Python server port `90` opened in backend container. Obviously, this is returning traffic indicating the downloading of the file is still ongoing. The `Action` is also NAT(`N`), but this time it is the reverse operation – source NAT (`SNAT`) and source PAT(`SPAT`).
The vRouter will translate the backend's source IP source port to the service IP and port, before putting it into the MPLSoUDP tunnel and returning back to the client pod in remote node.

The complete end-to-end traffic flow is illustrated in Figure 5.18.

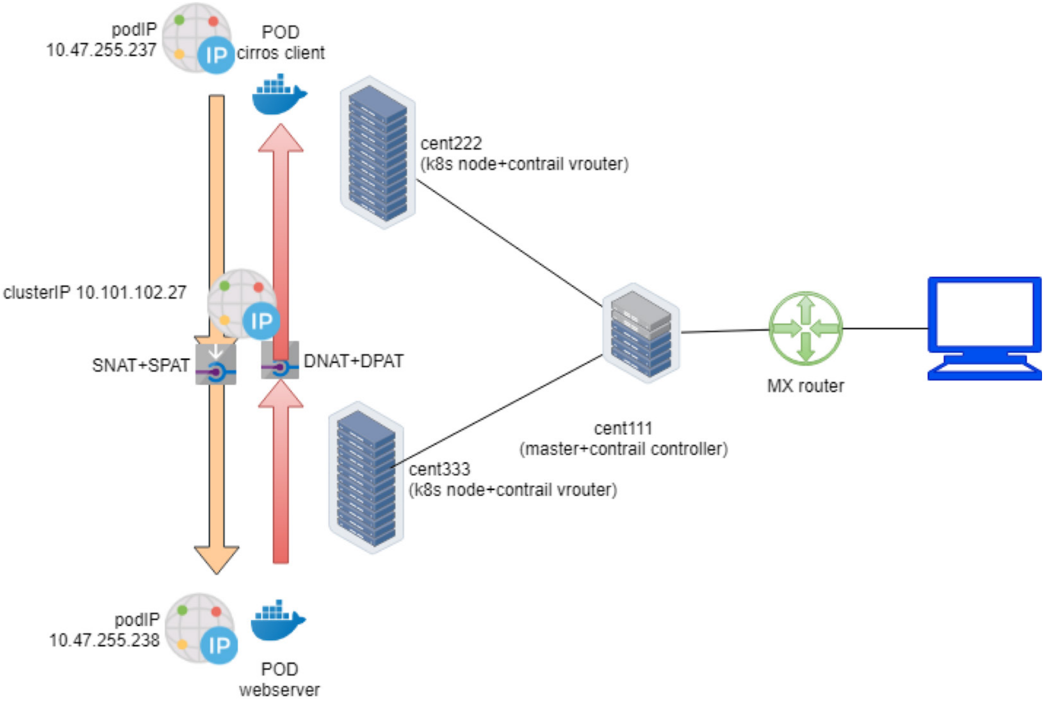


Figure 5.18 ClusterIP Service Traffic Flow (NAT)

Contrail Load Balancer Service

Chapter 3 briefly discussed load balancer service. It mentioned that if the goal is to expose the service to the external world outside of the cluster, then just specify `ServiceType` as the `LoadBalancer` in the service YAML file.

In Contrail, whenever a service of `type: LoadBalancer` gets created, not only will a `clusterIP` be allocated and exposed to other pods within the cluster, but also a `floating IP` from the public floating IP pool will be assigned to the load balancer instance as an external IP and exposed to the public world outside of the cluster.

While the `clusterIP` is still acting as a `VIP` to the client *inside* of the cluster, the `floating ip` or `external ip` will essentially act as a `VIP` facing those clients sitting *outside* of the cluster, for example, a remote Internet host which sends requests to the service across the gateway router.

The next section demonstrates how the `LoadBalancer` type of service works in our end-to-end lab setup, which includes the Kubernetes cluster, fabric switch, gateway router, and Internet host.

External IP as Floating IP

Let's look at the YAML file of a `LoadBalancer` service. It's the same as the clusterIP service except just one more line declaring the service `type`:

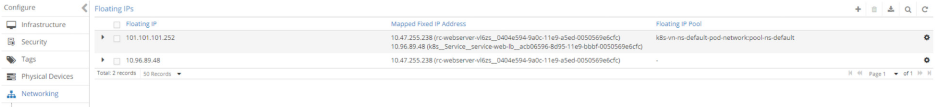
```
#service-web-lb.yaml
apiVersion: v1
kind: Service
metadata:
  name: service-web-lb
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver
  type: LoadBalancer    #<---
```

Create and verify the service:

```
$ kubectl apply -f service-web-lb.yaml
service/service-web-lb created

$ kubectl get svc -o wide
NAME            TYPE          CLUSTER-IP   EXTERNAL-IP      PORT(S)         AGE   SELECTOR
service-web-lb  LoadBalancer  10.96.89.48  101.101.101.252  8888:32653/TCP  10s   app=webserver
```

Compare the output with the `clusterIP` service type, this time there is an IP allocated in the EXTERNAL-IP column. If you remember what we've covered in the floating IP pool section, you should understand this EXTERNAL-IP is actually another `FIP`, allocated from the `NS FIP pool` or `global FIP pool`. We did not give any specific floating IP pool information in the service object YAML file, so based on the algorithm the right floating IP pool will be used automatically.

From the UI you can see that for the `loadbalancer` service we now have two floating IPs: one as a clusterIP (internal VIP) and the other one as EXTERNAL-IP (external VIP), as can be seem in Figure 5.19:

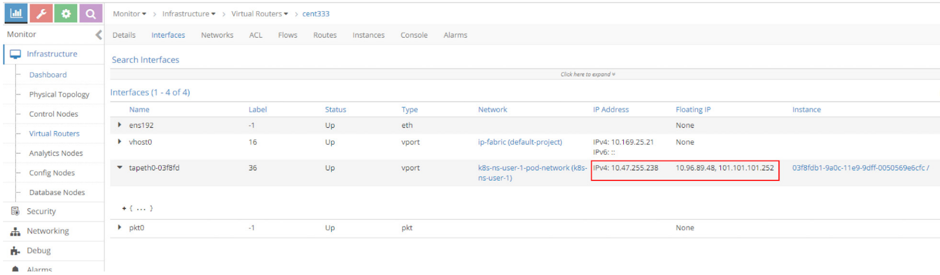


□ Floating IP	Mapped Fixed IP Address	Floating IP Pool
□ 101.101.101.252	10.47.255.238 (rc-webserver-vl6zs__0404e594-9a0c-11e9-a5ed-0050569e6cfc) 10.96.89.48 (k8s__Service__service-web-lb__acb06596-8d95-11e9-bbbf-0050569e6cfc)	k8s-vn-ns-default-pod-network:pool-ns-default
□ 10.96.89.48	10.47.255.238 (rc-webserver-vl6zs__0404e594-9a0c-11e9-a5ed-0050569e6cfc)	-

Total: 2 records 50 Records Page 1 of 1

Figure 5.19 Two Floating IPs for a Load Balancer Service

Both floating IPs are associated with the pod interface shown in the next screen capture, Figure 5.20.

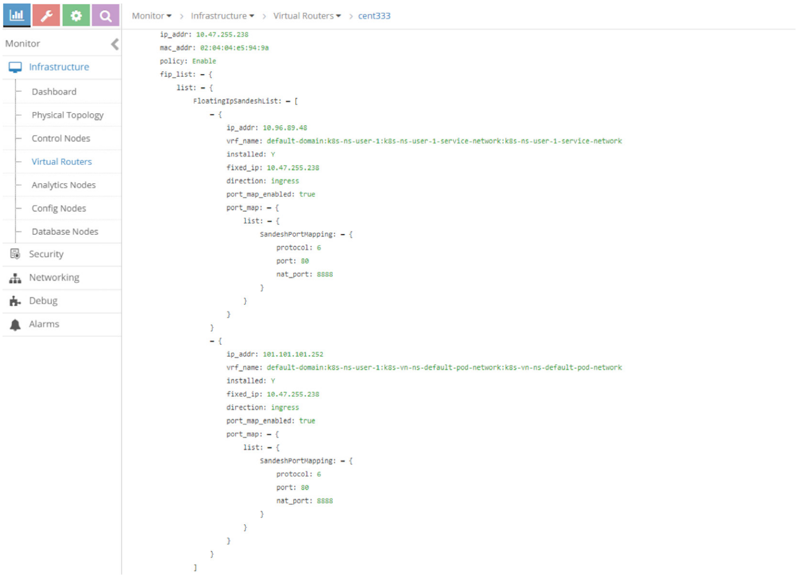


Interfaces (1 - 4 of 4)

Name	Label	Status	Type	Network	IP Address	Floating IP	Instance
ens192	-1	Up	eth			None	
vhost0	16	Up	vport	ip-fabric (default-project)	IPv4: 10.169.25.21 IPv6: ::	None	
tapeth0-03f8fd	36	Up	vport	k8s-ns-user-1-pod-network (k8s-ns-user-1)	IPv4: 10.47.255.238	10.96.89.48, 101.101.101.252	03f8fdb1-9a0c-11e9-9dff-0050569e6cfc /
pkt0	-1	Up	pkt			None	

Figure 5.20 Pod Interface

Expand the tap interface and you will see two floating IPs listed in the `fip_list`:



```
ip_addr: 10.47.255.238
mac_addr: 02:04:04:e5:94:9a
policy: Enable
fip_list: - {
    list: - {
        FloatingIpSandeshList: - [
            - {
                ip_addr: 10.96.89.48
                vrf_name: default-domain:k8s-ns-user-1:k8s-ns-user-1-service-network:k8s-ns-user-1-service-network
                installed: Y
                fixed_ip: 10.47.255.238
                direction: ingress
                port_map_enabled: true
                port_map: - {
                    list: - {
                        SandeshPortMapping: - {
                            protocol: 6
                            port: 80
                            nat_port: 8888
                        }
                    }
                }
            }
            - {
                ip_addr: 101.101.101.252
                vrf_name: default-domain:k8s-ns-user-1:k8s-vn-ns-default-pod-network:k8s-vn-ns-default-pod-network
                installed: Y
                fixed_ip: 10.47.255.238
                direction: ingress
                port_map_enabled: true
                port_map: - {
                    list: - {
                        SandeshPortMapping: - {
                            protocol: 6
                            port: 80
                            nat_port: 8888
                        }
                    }
                }
            }
        ]
```

Figure 5.21 Pod Interface Detail

Now you should understand the only difference here between the two types of services is that for the load balancer service, an extra FIP is allocated from the public FIP pool, which is advertised to the gateway router and acts as the outside-facing VIP. That is how the `loadbalancer` service exposes itself to the external world.

Gateway Router VRF Table

In the Contrail floating IP section you've learned how to advertise floating IP. But now let's review the main concepts to understand how it works in Contrail `service` implementation.

The `route-target` community setting in the floating IP VN makes it reachable by the Internet host, so effectively our service is now also exposed to the Internet instead of only to pods inside of the cluster. Examining the gateway router's VRF table reveals this:

```
labroot@camaro> show route table k8s-test.inet.0 101.101.101/24
Jun 19 03:56:11

k8s-test.inet.0: 23 destinations, 40 routes (23 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

101.101.101.252/32 *[BGP/170] 00:01:11, MED 100, localpref 200, from 10.169.25.19
                      AS path: ?, validation-state: unverified
                    > via gr-2/2/0.32771, Push 44
```

The floating IP host route is learned by the gateway router from the Contrail controller – more specifically, Contrail control node – which acts as a standard MP-BGP VPN RR reflecting routes between compute nodes and the gateway router. A further look at the detailed version of the same route displays more information about the process:

```
labroot@camaro> show route table k8s-test.inet.0 101.101.101/24 detail
Jun 20 11:45:42

k8s-test.inet.0: 23 destinations, 41 routes (23 active, 0 holddown, 0 hidden)
101.101.101.252/32 (2 entries, 1 announced)
        *BGP    Preference: 170/-201
                Route Distinguisher: 10.169.25.20:9
                ......
                Source: 10.169.25.19                    #<---
                Next hop type: Router, Next hop index: 1266
                Next hop: via gr-2/2/0.32771, selected  #<---
                Label operation: Push 44
                Label TTL action: prop-ttl
                Load balance label: Label 44: None;
                ......
                Protocol next hop: 10.169.25.21         #<---
                Label operation: Push 44
                Label TTL action: prop-ttl
                Load balance label: Label 44: None;
                Indirect next hop: 0x900c660 1048574 INH Session ID: 0x690
                State: <Secondary Active Int Ext ProtectionCand>
                Local AS: 13979 Peer AS: 60100
                Age: 10:15:38   Metric: 100     Metric2: 0
                Validation State: unverified
                Task: BGP_60100_60100.10.169.25.19
                Announcement bits (1): 1-KRT
                AS path: ?
                Communities: target:500:500 target:64512:8000016
                    ......
```

```
Import Accepted
VPN Label: 44
Localpref: 200
Router ID: 10.169.25.19
Primary Routing Table bgp.l3vpn.0
```

Highlights of the output here are:

- The `source` indicates from which BGP peer the route is learned, `10.169.25.19` is the Contrail Controller (and Kubernetes master) in the book's lab.
- The `protocol next hop` tells who generates the route. And `10.169.25.20` is node `cent222` where the backend webserver pod is running.
- The `gr-2/2/0.32771` is an interface representing the (MPLS over) GRE tunnel between the gateway router and node `cent333`.

Load Balancer Service Workflow

To summarize, the floating IP is given to the service as its external IP is advertised to the gateway router and gets loaded into the router's VRF table. When the Internet host sends a request to the floating IP through the MPLSoGRE tunnel, the gateway router will forward it to the compute node where the backend pod is located.

The packet flow is illustrated in Figure 5.22.

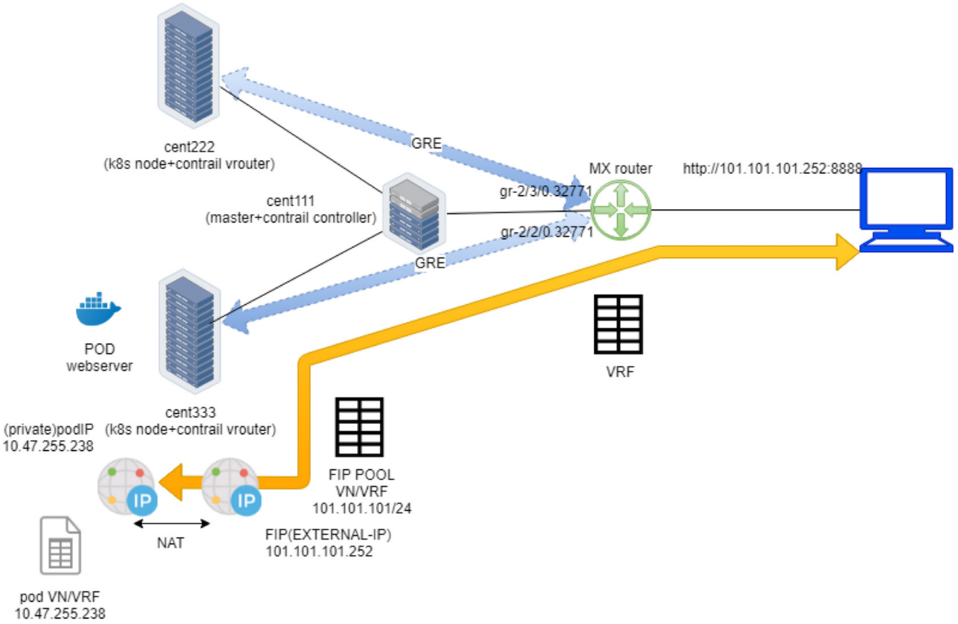


Figure 5.22 Load Balancer Service Workflow

Here is the full story of Figure 5.22:

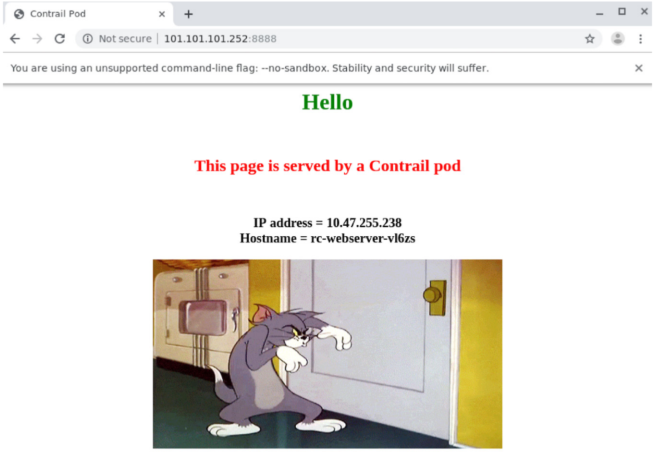
- Create a `FIP pool` from a public VN, with `route-target`. The VN is advertised to the remote gateway router via MP-BGP.
- Create a pod with a label `app: webserver`, and Kubernetes decides the pod will be created in node `cent333`. The node publishes the pod IP via XMPP.
- Create a `loadbalancer` type of service with `service port` and label selector `app=webserver`. Kubernetes allocates a service IP.
- Kubernetes finds the pod with the matching label and updates the `endpoint` with the pod IP and port information.
- Contrail creates a `loadbalancer` instance and assigns a floating IP to it. Contrail also associates that floating IP with the pod interface, so there will be a one-to-one NAT operation between the floating IP and the pod IP.
- Via XMPP, node `cent333` advertises this floating IP to Contrail Controller `cent111`, which then advertises it to the gateway router.
- On receiving the floating IP prefix, the gateway router checks and sees that the RT of the prefix matches what it's expecting, and it will import the prefix in the local VRF table. At this moment the gateway learns the next hop of the floating IP is `cent333`, so it generates a soft GRE tunnel toward `cent333`.
- When the gateway router sees a request coming from the Internet toward the floating IP, it will send the request to the node `cent333` through the MPLS over GRE tunnel.
- The vRouter in the node sees the packets destined to the floating IP, it will perform NAT so the packets will be sent to the right backend pod.

Verify Load Balancer Service

To verify end-to-end service access from Internet host to the backend pod, let's log in to the Internet host desktop and launch a browser, with URL pointing to `http://101.101.101.252:8888`.

TIP Keep in mind that the Internet host request has to be sent to the public floating IP, not to the service IP (clusterIP) or backend pod IP which are only reachable from inside the cluster!

You can see the returned web page on the browser below in Figure 5.23.



You are using an unsupported command-line flag: --no-sandbox. Stability and security will suffer.

Hello

This page is served by a Contrail pod

IP address = 10.47.255.238
Hostname = rc-webserver-vl6zs

Figure 5.23 Verify End-to-End Service

TIP This book's lab installed a Centos desktop as an Internet host.

To simplify the test, you can also SSH into the Internet host and test it with the `curl` tool:

```
[root@cent-client ~]# curl http://101.101.101.252:8888 | w3m -T text/html | cat
             Hello
This page is served by a Contrail pod
  IP address = 10.47.255.238
  Hostname = webserver-846c9ccb8b-vl6zs
```

And the Kubernetes service is available from the Internet!

Load Balancer Service ECMP

You've seen how the load balancer type of service is exposed to the Internet and how the floating IP did the trick. In the clusterIP service section, you've also seen how the service load balancer ECMP works. But what you haven't seen yet is how the ECMP processing works under the load balancer type of service. To demonstrate this we again scale the RC to generate one more backend pod behind the `service`:

```
$ kubectl scale deployment webserver --replicas=2
deployment.extensions/webserver scaled

$ kubectl get pod -l app=webserver -o wide
NAME                    READY  STATUS   RESTARTS  AGE  IP             NODE     NOMINATED  NODE
webserver-846c9ccb8b-r9zdt  1/1    Running  0         25m  10.47.255.238  cent333  <none>
webserver-846c9ccb8b-xkjpw  1/1    Running  0         23s  10.47.255.236  cent222  <none>
```

Here's the question: with two pods on different nodes, and both as backend now, from the gateway router's perspective, when it gets the service request, which node does it choose to forward the traffic to?

Let's check the gateway router's VRF table again:

```
labroot@camaro> show route table k8s-test.inet.0 101.101.101.252/32
Jun 30 00:27:03

k8s-test.inet.0: 24 destinations, 46 routes (24 active, 0 holddown, 0 hidden)
@ = Routing Use Only, # = Forwarding Use Only
+ = Active Route, - = Last Active, * = Both

101.101.101.252/32 *[BGP/170] 00:00:25, MED 100, localpref 200, from 10.169.25.19
                       AS path: ?
                    validation-state: unverified, > via gr-2/3/0.32771, Push 26
                    [BGP/170] 00:00:25, MED 100, localpref 200, from 10.169.25.19
                       AS path: ?
                    validation-state: unverified, > via gr-2/2/0.32771, Push 26
```

The same floating IP prefix is imported, as we've seen in the previous example, except that now the same route is learned twice and an additional MPLSoGRE tunnel is created. Previously, in the clusterIP service example, the `detail` option was used in the `show route` command to find the tunnel endpoints. This time we examine the soft GRE `gr-` interface to find the same:

```
labroot@camaro> show interfaces gr-2/2/0.32771
Jun 30 00:56:01
  Logical interface gr-2/2/0.32771 (Index 392) (SNMP ifIndex 1801)
    Flags: Up Point-To-Point SNMP-Traps 0x4000
    IP-Header 10.169.25.21:192.168.0.204:47:df:64:0000000800000000      #<---
    Encapsulation: GRE-NULL
    Copy-tos-to-outer-ip-header: Off, Copy-tos-to-outer-ip-header-transit: Off
    Gre keepalives configured: Off, Gre keepalives adjacency state: down
    Input packets : 0
    Output packets: 0
    Protocol inet, MTU: 9142
    Max nh cache: 0, New hold nh limit: 0, Curr nh cnt: 0, Curr new hold cnt: 0, NH drop cnt: 0
      Flags: None
    Protocol mpls, MTU: 9130, Maximum labels: 3
      Flags: None

labroot@camaro> show interfaces gr-2/3/0.32771
  Logical interface gr-2/3/0.32771 (Index 393) (SNMP ifIndex 1703)
    Flags: Up Point-To-Point SNMP-Traps 0x4000
    IP-Header 10.169.25.20:192.168.0.204:47:df:64:0000000800000000      #<---
    Encapsulation: GRE-NULL
    Copy-tos-to-outer-ip-header: Off, Copy-tos-to-outer-ip-header-transit: Off
    Gre keepalives configured: Off, Gre keepalives adjacency state: down
    Input packets : 11
    Output packets: 11
    Protocol inet, MTU: 9142
    Max nh cache: 0, New hold nh limit: 0, Curr nh cnt: 0, Curr new hold cnt: 0, NH drop cnt: 0
      Flags: None
    Protocol mpls, MTU: 9130, Maximum labels: 3
      Flags: None
```

The `IP-Header` of the `gr-` interface indicates the two end points of the GRE tunnel:

- `10.169.25.20:192.168.0.204`: Here the tunnel is between node `cent222` and the gateway router.
- `10.169.25.21:192.168.0.204`: Here the tunnel is between node `cent333` and the gateway router

We end up needing two tunnels in the gateway router, each pointing to a different node where a backend pod is running. Now we believe the router will perform ECMP load balancing between the two GRE tunnels, whenever it gets a service request toward the same floating IP. Let's check it out.

Verify Load Balancer Service ECMP

To verify ECMP let's just pull the webpage a few more times and we can expect to see both pod IPs eventually displayed.

Turns out this never happens!

```
[root@cent-client ~]# curl http://101.101.101.252:8888 | lynx -stdin --dump
                                     Hello
This page is served by a Contrail pod
  IP address = 10.47.255.236
  Hostname = webserver-846c9ccb8b-xkjpw
```

TIP Lynx is another terminal web browser similar to the w3m program that we used earlier.

The only webpage is from the first backend pod `10.47.255.236`, `webserver-846c9ccb8b-xkjpw`, running in node `cent222`. The other one never shows up. So the expected ECMP does not happen yet. But when you examine the routes using the `detail` or `extensive` keyword, you'll find the root cause:

```
labroot@camaro> show route table k8s-test.inet.0 101.101.101.252/32 detail | match state
Jun 30 00:48:29
                State: <Secondary Active Int Ext ProtectionCand>
                Validation State: unverified
                State: <Secondary NotBest Int Ext ProtectionCand>
                Validation State: unverified
```

This reveals that even if the router learned the same prefix from both nodes, only one is `Active` and the other won't take effect because it is `NotBest`. Therefore, the second route and the corresponding GRE interface `gr-2/2/0.32771` will never get loaded into the forwarding table:

```
labroot@camaro> show route forwarding-table table k8s-test destination 101.101.101.252
Jun 30 00:53:12
Routing table: k8s-test.inet
Internet:
Enabled protocols: Bridging, All VLANs,
Destination         Type  RtRef  Next  hop      Type  Index  NhRef  Netif
101.101.101.252/32  user  0      indr  1048597  2
                                Push 26     1272     2 gr-2/3/0.32771
```

This is the default Junos BGP path selection behavior, but a detailed discussion of that is beyond the scope of this book.

MORE? For the Junos BGP path selection algorithm, go to the Juniper TechLibrary: https://www.juniper.net/documentation/en_US/junos/topics/topic-map/bgp-path-selection.html.

The solution is to enable the `multipath vpn-unequal-cost` knob under the VRF table:

```
labroot@camaro# set routing-instances k8s-test routing-options multipath vpn-unequal-cost
```

Now let's check the VRF table again:

```
labroot@camaro# run show route table k8s-test.inet.0 101.101.101.252/32
Jun 26 20:09:21

k8s-test.inet.0: 27 destinations, 54 routes (27 active, 0 holddown, 0 hidden)
@ = Routing Use Only, # = Forwarding Use Only
+ = Active Route, - = Last Active, * = Both

101.101.101.252/32 @[BGP/170] 00:00:04, MED 100, localpref 200, from 10.169.25.19
                      AS path: ?
                    validation-state: unverified, > via gr-2/3/0.32771, Push 72
                    [BGP/170] 00:00:52, MED 100, localpref 200, from 10.169.25.19
                      AS path: ?
                    validation-state: unverified, > via gr-2/2/0.32771, Push 52
                   #[Multipath/255] 00:00:04, metric 100, metric2 0
                      via gr-2/3/0.32771, Push 72
                    > via gr-2/2/0.32771, Push 52
```

A `Multipath` with both GRE interfaces will be added under the floating IP prefix, and the forwarding table reflects the same:

```
labroot@camaro> show route forwarding-table table k8s-test destination 101.101.101.252
Jun 30 01:12:36
Routing table: k8s-test.inet
Internet:
Enabled protocols: Bridging, All VLANs,
Destination        Type RtRef Next hop    Type Index    NhRef Netif
101.101.101.252/32 user     0             ulst  1048601     2
                                          indr  1048597     2
                                         Push 26     1272     2 gr-2/3/0.32771
                                          indr  1048600     2
                                         Push 26     1277     2 gr-2/2/0.32771
```

Now, try to pull the webpage from the Internet host multiple times with `curl` or a web browser and you'll see the random result – both backend pods get the request and responses back:

```
[root@cent-client ~]# curl http://101.101.101.252:8888 | lynx -stdin --dump
                                     Hello
This page is served by a Contrail pod
  IP address = 10.47.255.236
  Hostname = webserver-846c9ccb8b-xkjpw

[root@cent-client ~]# curl http://101.101.101.252:8888 | lynx -stdin --dump
                                     Hello
This page is served by a Contrail pod
  IP address = 10.47.255.238
  Hostname = webserver-846c9ccb8b-r9zdt
```

The end-to-end packet flow is illustrated in Figure 5.24.

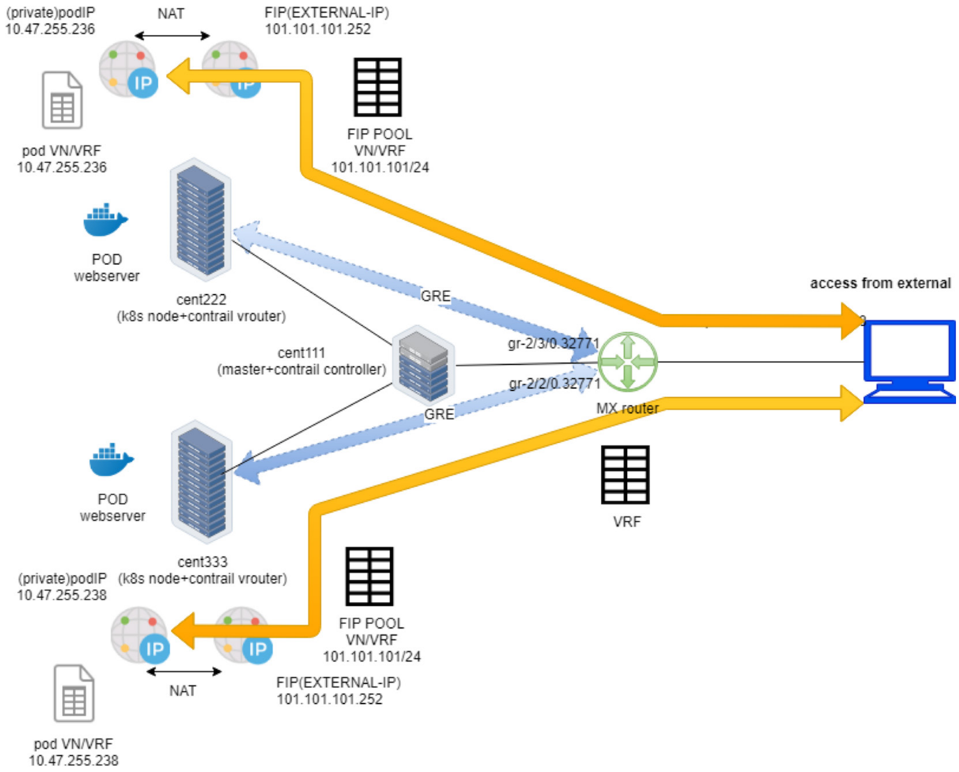


Figure 5.24 Load Balancer Service ECMP

Chapter 6

Contrail Ingress

Chapter 3 contained ingress basics, the relation to service, ingress types, and the YAML file of each type.

This chapter introduces the details of ingress workflow in Contrail implementation, then uses a few test cases to demonstrate and verify ingress in the Contrail environment.

Contrail Ingress Load Balancer

Like Contrail's service implementation, Contrail Ingress is also implemented through load balancer, but with a different `loadbalancer_provider` attribute. Accordingly, the `contrail-svc-monitor` component takes different actions to implement Ingress in Contrail environments.

Remember that way back in the Contrail/Kubernetes architecture section we gave the object mapping between Kubernetes and Contrail, and that Kubernetes service maps to ECMP load balancer (native) and Ingress maps to Haproxy load balancer.

In the service section when we were exploring the load balancer and the relevant objects (listener, pool, and member), we noticed the load balancer's loadbalancer_provider type is native.

```
"loadbalancer_provider": "native",
```

In this section we'll see the `loadbalancer_provider` type is `opencontrail` for ingress's load balancer. We'll also look into the similarities and differences between service load balancer and Ingress load balancer.

Contrail Ingress Workflow

When an Ingress is configured in a Contrail Kubernetes environment, the event will be noticed by other system components and a lot of actions will be triggered. The deep level implementation is beyond the scope of this book, but at a high level this is the workflow:

- The contrail-kube-manager keeps listening to the events of the `kube-apiserver`.
- User creates an ingress object (rules).
- The contrail-kube-manager gets the event from kube-apiserver.
- The contrail-kube-manager creates a loadbalancer object in contrail DB, and sets the loadbalancer_provider type as opencontrail for ingress (where as it is native for service).

As mentioned earlier, the contrail-service-monitor component sees the loadbalancer creation event, based on the loadbalancer_provider type, and it invokes the registered load balancer driver for the specified loadbalancer_provider type:

- If the loadbalancer_provider type is native, it will invoke the ECMP loadbalancer driver for ECMP loadbalancing, which we reviewed in the previous section.
- If the loadbalancer_provider type is opencontrail, it will invoke the haproxy loadbalancer driver, which triggers the haproxy processes to be launched in Kubernetes nodes.

As you can see, Contrail implements Ingress with haproxy load balancer, as you also read in the section on Contrail Kubernetes object mapping. Chapter 3 described the ingress controller, and how multiple ingress controllers can coexist in Contrail. In the Contrail environment, the contrail-kube-manager plays the Ingress controller role. It reads the ingress rules that users input, and programs them into the load balancer. Furthermore:

- For each Ingress object, one load balancer will be created;
- Two haproxy processes will be created for Ingress and they are working in active-standby mode:
  - one compute node runs the active haproxy process
  - the other compute node runs the standby haproxy process
- Both haproxy processes are programmed with appropriate configuration, based on the rules defined in the Ingress object.

Contrail Ingress Traffic Flow

Client requests, such as a type of overlay traffic, may come from two different sources, depending on who initiates the request:

- An *internal* request: requests coming from another pod inside of the cluster.
- An *external* request: requests coming from an Internet host outside of the cluster.

The only difference between the two is how the traffic hits the active haproxy. An ingress will be allocated two IPs: a cluster-internal virtual IP and an external virtual IP.

Here is the traffic flow for the client request:

1. For internal requests, it hits Ingress's internal VIP directly.
2. For external requests, it first hits Ingress's external VIP – the floating IP, which is the one exposed to external, and that's when NAT starts to play as we've explained in the FIP section. After NAT processing, traffic is forwarded to the internal Ingress VIP.
3. From this moment on, both types of requests are processed exactly the same way.
4. The requests will be proxied to the corresponding service IP.
5. Based on the availability of the backend pods, it will be sent to the node where one of the backend pods are located and eventually reaches the target pods.
6. In case the backend pods are running in a different compute node than the one running active haproxy, a MPLS over UDP tunnel is created between the two compute nodes.

Figure 6.1 and 6.2 illustrate the end-to-end service request flow when accessing from a pod in the cluster and when accessing from an Internet host.

Contrail supports all three types of ingress:

- HTTP-based single-service ingress
- simple-fanout ingress
- name-based virtual hosting ingress

Next we'll look into each type of ingress.

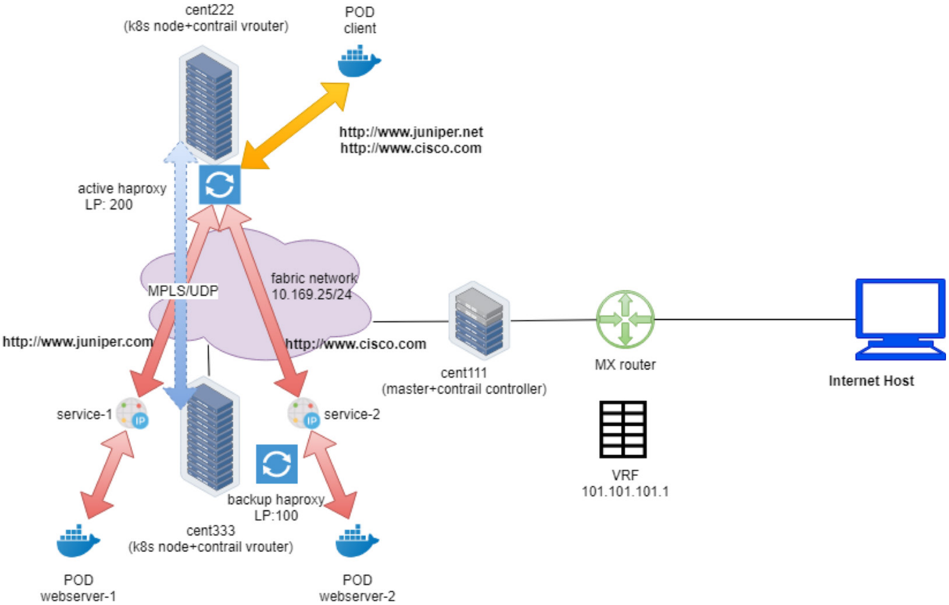


Figure 6.1 Ingress Traffic Flow: Access from Internal

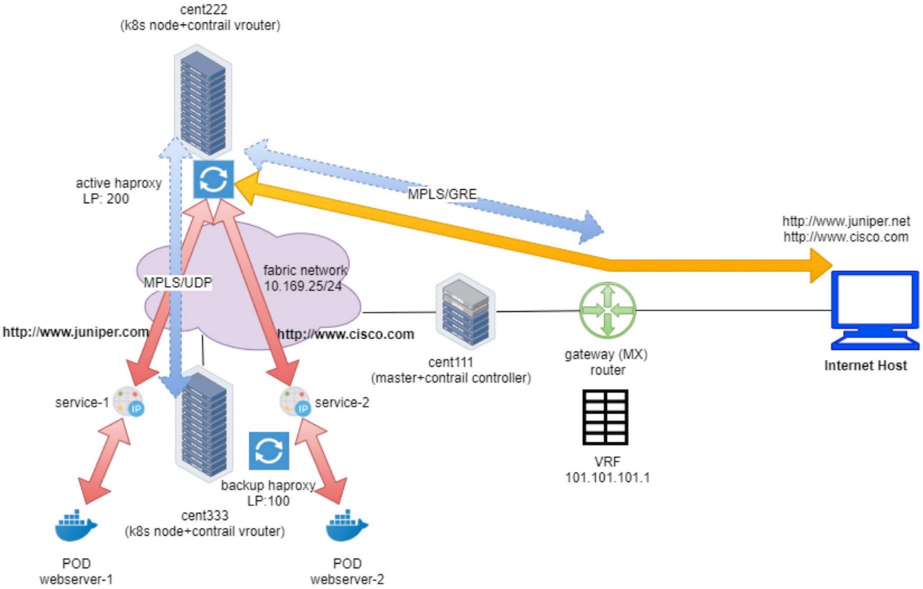


Figure 6.2 Ingress Traffic Flow: Access from External

Ingress Setup

This book's lab uses the same testbed as used for the service test, shown in Figure 6.3.

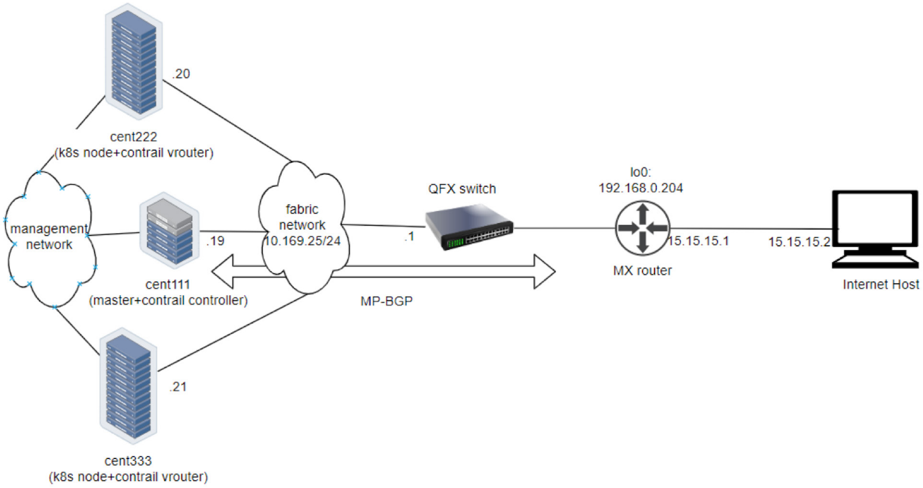


Figure 6.3 Ingress Testbed

Single Service Ingress

Single service ingress is the most basic form of ingress. It does not define any rules and its main use is to expose service to the outside world. It proxies all incoming service requests to the same single backend service:

```
www.juniper.net --|                  |
www.cisco.com   --|  101.101.101.1   |-> webservice
www.google.com  --|                  |
```

To demonstrate single service type of ingress, the objects that we need to create are:

- an Ingress object that defines the backend service
- a backend service object
- at least one backend pod for the service

Ingress Definition

In the single service ingress test lab, we want to request that any URLs are directed to service-web-clusterip with servicePort 8888. Here is the corresponding YAML definition file:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-ss
spec:
  backend:
    serviceName: service-web-clusterip
    servicePort: 8888
```

This does not look fancy. Basically there is nothing else but a reference to a single service webserver-1 as its backend. All HTTP requests will be dispatched to this service, and from there the request will reach a backend pod. Simple enough. Let's look at the backend service.

Backend Service Definition

You can use the exact same service as introduced in the service example:

```
apiVersion: v1
kind: Service
metadata:
  name: service-web-clusterip
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver
  #type: LoadBalancer
```

NOTE The service type is optional. With Ingress, service does not need to be exposed externally anymore. Therefore, the `LoadBalancer` type of service is not required.

Backend Pod Definition

Just as in the service example, you can use exactly the same webserver deployment to launch backend pods:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver
  labels:
    app: webserver
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webserver
  template:
    metadata:
      name: webserver
      labels:
        app: webserver
    spec:
      containers:
      - name: webserver
        image: contrailk8sdayone/contrail-webserver
        securityContext:
           privileged: true
        ports:
        - containerPort: 80
```

All in One YAML File

As usual, you can create an individual YAML file for each of the objects, but considering that these objects always need to be created and removed together in Ingress, it's better to merge definitions of all these objects into one YAML file. YAML syntax supports this by using document delimiters (a --- line between each object definition):

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-ss
spec:
  backend:
    serviceName: service-web-clusterip
    servicePort: 8888
---
apiVersion: v1
kind: Service
metadata:
  name: service-web-clusterip
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver
  #type: LoadBalancer
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver
  selector:
    matchLabels:
      app: webserver
spec:
  replicas: 1
  selector:
    app: webserver
  template:
    metadata:
      name: webserver
      labels:
        app: webserver
    spec:
      containers:
      - name: webserver
        image: contrailk8sdayone/contrail-webserver
        securityContext:
           privileged: true
        ports:
        - containerPort: 80
```

The benefits of the all-in-one YAML file are:

- You can create/update all objects in the YAML file in one go, using just one `kubectl apply` command.
- Similarly, if anything goes wrong and you need to clean up, you can delete all objects created with the YAML file in one `kubectl delete` command.
- Whenever needed, you can still delete each individual object independently, by providing the object name.

TIP During test processing, you may need to create and delete all objects as a whole very often, so grouping multiple objects in one YAML file can be very convenient.

Deploy the Single Service Ingress

Before applying the YAML file to get all the objects created, let's take a quick look at our two nodes. You want to see if there is any haproxy process running without ingress, so later, after you deploy ingress, you can compare:

```
$ ps aux | grep haproxy
$
```

So the answer is no, there is no haproxy process in either of the nodes. Haproxy will be created only after you create Ingress and the corresponding load balancer object is seen by the contrail-service-monitor. We'll check this again after an Ingress is created:

```
$ kubectl apply -f ingress/ingress-single-service.yaml
ingress.extensions/ingress-ss created
service/service-web-clusterip created
deployment.extensions/webserver created
```

The ingress, one service, and one deployment object have now been created.

Ingress Object

Let's examine the ingress object:

```
$ kubectl get ingresses.extensions -o wide
NAME         HOSTS   ADDRESS                        PORTS   AGE
ingress-ss   *       10.47.255.238,101.101.101.1    80      29m
```

```
$ kubectl get ingresses.extensions -o yaml
apiVersion: v1
items:
- apiVersion: extensions/v1beta1
  kind: Ingress
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
        {"apiVersion":"extensions/v1beta1", "kind":"Ingress",
        "metadata":{"annotations":{},"name":"ingress-ss","namespace":"ns-user-1"},
        "spec":{"backend":{"serviceName":"service-web-clusterip ", "servicePort":80}}}
    creationTimestamp: 2019-07-18T04:06:29Z
    generation: 1
    name: ingress-ss
    namespace: ns-user-1
    resourceVersion: "845969"
    selfLink: /apis/extensions/v1beta1/namespaces/ns-user-1/ingresses/ingress-ss
    uid: 6b48bd8f-a911-11e9-8112-0050569e6cfc
  spec:
    backend:
      serviceName: service-web-clusterip
      servicePort: 80
  status:
    loadBalancer:
      ingress:
      - ip: 101.101.101.1
      - ip: 10.47.255.238
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

As expected, the backend service is properly applied to the ingress. In this single-service ingress there are no explicit rules defined to map a certain URL to a different service – all HTTP requests will be dispatched to the same backend service.

TIP In the items metadata annotations kubectl.kubernetes.io/last-applied-configuration section of the output…

```
{"apiVersion":"extensions/v1beta1", "kind":"Ingress",
"metadata":{"annotations":{},"name":"ingress-ss","namespace":"ns-user-1"},
"spec":{"backend":{"serviceName":"service-web-clusterip", "servicePort":80}}}
```

…Actually contains the configuration information that you provided. You can format it (with a JSON formatting tool like Python's json.tool module) to get a better view…

```
{
    "apiVersion": "extensions/v1beta1",
    "kind": "Ingress",
    "metadata": {
        "annotations": {},
        "name": "ingress-ss",
        "namespace": "ns-user-1"
    },
    "spec": {
        "backend": {
            "serviceName": "service-web-clusterip",
            "servicePort": 80
        }
    }
}
```

…And you can do the same formatting for all other objects to make it more readable.

But what may confuse you are the two IP addresses shown here:

```
loadBalancer:
  ingress:
  - ip: 101.101.101.1
  - ip: 10.47.255.238
```

We've seen these two subnets in service examples:

- 10.47.255.x is a cluster-internal podIP allocated from the pod's default subnet, and
- 101.101.101.x is the public FIP associated with an internal IP.

The question is: Why does ingress even require a podIP and FIP?

Let's hold off on the answer for now and continue to check the service and pod object created from the all-in-one YAML file. We'll come back to this question shortly.

Service Objects

Let's check on services:

```
$ kubectl get svc -o wide
NAME                   TYPE       CLUSTER-IP      EXTERNAL-IP  PORT(S)   AGE  SELECTOR
service-web-clusterip  ClusterIP  10.97.226.91  <none>       8888/TCP  28m  app=webserver
```

The service is created and allocated a clusterIP. We've seen this before and it looks like nothing special. Now, let's look at the backend and client pods:

```
$ kubectl get pod -o wide --show-labels
NAME                        READY  STATUS   ... IP             NODE     ... LABELS
client                      1/1    Running  ... 10.47.255.237  cent222  ... app=client
webserver-846c9ccb8b-9nfdx  1/1    Running  ... 10.47.255.236  cent333  ... app=webserver
```

Everything looks fine, here. There is a backend pod running for the service. You have already learned how selector and label works in service-pod associations. Nothing new here. So let's examine the haproxy and try to make some sense out of the two IPs allocated to the ingress object.

Haproxy Processes

Earlier, before the ingress was created, we were looking for the haproxy process in nodes but could not see anything. Let's check it again and see if any magic happens:

On node cent222:

```
$ ps aux | grep haproxy
188  23465  0.0  0.0  55440  852  ?  Ss  00:58  0:00  haproxy
  -f  /var/lib/contrail/loadbalancer/haproxy/5be035d8-a918-11e9-8112-0050569e6cfc/haproxy.conf
  -p  /var/lib/contrail/loadbalancer/haproxy/5be035d8-a918-11e9-8112-0050569e6cfc/haproxy.pid
  -sf  23447
```

On node cent333:

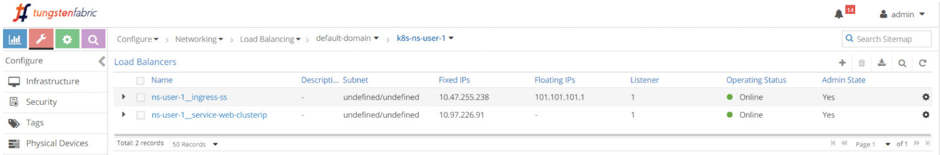
```
$ ps aux | grep haproxy
188   16335  0.0  0.0  55440  2892  ?  Ss  00:58  0:00  haproxy
  -f  /var/lib/contrail/loadbalancer/haproxy/5be035d8-a918-11e9-8112-0050569e6cfc/haproxy.conf
  -p  /var/lib/contrail/loadbalancer/haproxy/5be035d8-a918-11e9-8112-0050569e6cfc/haproxy.pid
  -sf  16317
```

And right after ingress is created, you can see a haproxy process created in each of our two nodes!

Previously we stated that Contrail Ingress is also implemented through load balancer (just like service). Since ingress's loadbalancer_provider type is opencontrail, 'contrail-svc-monitor invokes the haproxy load balancer driver. The haproxy driver generates the required haproxy configuration for the ingress rules and triggers haproxy processes to be launched (in active-standby mode) with the generated configuration in Kubernetes nodes.

Ingress Load Balancer Objects

We've mentioned the ingress load balancer a few times but haven't looked at it yet. In the service section, we've looked into the service load balancer object in the UI, and we inspected some details about the object data structure. Now, after creating the ingress object, let's check the list of load balancer objects again and see what ingress brings in here starting with the UI at Configure>Load Balancers.



Load Balancers

Name	Descripti...	Subnet	Fixed IPs	Floating IPs	Listener	Operating Status	Admin State
ns-user-1__ingress-ss	-	undefined/undefined	10.47.255.238	101.101.101.1	1	● Online	Yes
ns-user-1__service-web-clusterip	-	undefined/undefined	10.97.226.91	-	1	● Online	Yes

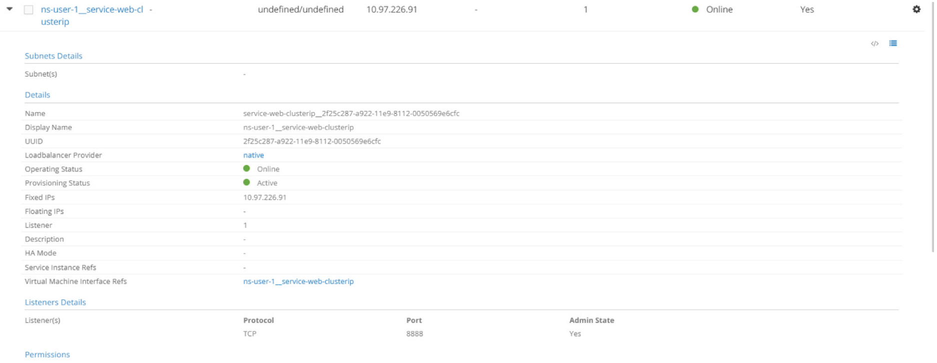
Total: 2 records 50 Records Page 1 of 1

Figure 6.4 UI: Load Balancers Configuration

Two load balancers are generated after applying the all-in-one YAML file:

- Load balancer ns-user-1__ingress-ss for ingress ingress-ss
- Load balancer ns-user-1__webservice-clusterip for service webserver-clusterip

We've been through the service load balancer object previously, and if you expand the service you will see lots of detail but nothing should surprise you.



□ ns-user-1__service-web-cl usterip	-	undefined/undefined	10.97.226.91	-	1	● Online	Yes

Subnets Details

Subnet(s)	-

Details

Name	service-web-clusterip__2f25c287-a922-11e9-8112-0050569e6cfc
Display Name	ns-user-1__service-web-clusterip
UUID	2f25c287-a922-11e9-8112-0050569e6cfc
Loadbalancer Provider	native
Operating Status	● Online
Provisioning Status	● Active
Fixed IPs	10.97.226.91
Floating IPs	-
Listener	1
Description	-
HA Mode	-
Service Instance Refs	-
Virtual Machine Interface Refs	ns-user-1__service-web-clusterip

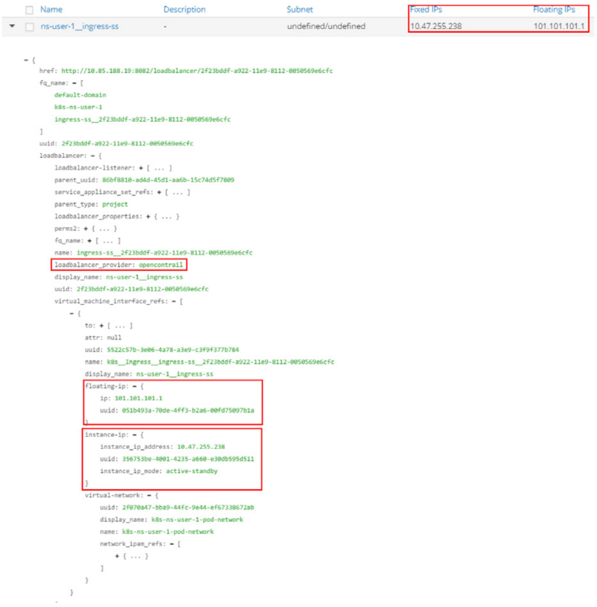
Listeners Details

Listener(s)	Protocol	Port	Admin State
	TCP	8888	Yes

Permissions

Figure 6.5 *Service Load Balancer Object (click the triangle in the left of the load balancer)*

As you can see, the service load balancer has a clusterIP and a listener object that is listening on port 8888. One thing to highlight is the loadbalancer_provider. This type is `native`, so the action contrail-svc-monitor takes is the Layer 4 (Application Layer) ECMP process, which is explored extensively in the service section. Let's expand the ingress load balancer and glance at the details.



	Name	Description	Subnet	Fixed IPs	Floating IPs
▾ □	ns-user-1__ingress-ss	-	undefined/undefined	10.47.255.238	101.101.101.1

```
- {
    href: http://10.85.188.19:8082/loadbalancer/2f23bddf-a922-11e9-8112-0050569e6cfc
    fq_name: - [
        default-domain
        k8s-ns-user-1
        ingress-ss__2f23bddf-a922-11e9-8112-0050569e6cfc
    ]
    uuid: 2f23bddf-a922-11e9-8112-0050569e6cfc
    loadbalancer: - {
        loadbalancer-listener: + [ ... ]
        parent_uuid: 86bf8810-ad4d-45d1-aa6b-15c74d5f7809
        service_appliance_set_refs: + [ ... ]
        parent_type: project
        loadbalancer_properties: + { ... }
        perms2: + { ... }
        fq_name: + [ ... ]
        name: ingress-ss__2f23bddf-a922-11e9-8112-0050569e6cfc
        loadbalancer_provider: opencontrail
        display_name: ns-user-1__ingress-ss
        uuid: 2f23bddf-a922-11e9-8112-0050569e6cfc
        virtual_machine_interface_refs: - [
            - {
                to: + [ ... ]
                attr: null
                uuid: 5522c57b-3e06-4a78-a3e9-c3f9f377b784
                name: k8s__Ingress__ingress-ss__2f23bddf-a922-11e9-8112-0050569e6cfc
                display_name: ns-user-1__ingress-ss
                floating-ip: - {
                    ip: 101.101.101.1
                    uuid: 051b493a-70de-4ff3-b2a6-00fd75097b1a
                }
                instance-ip: - {
                    instance_ip_address: 10.47.255.238
                    uuid: 356753be-4001-4235-a660-e30db595d511
                    instance_ip_mode: active-standby
                }
                virtual-network: - {
                    uuid: 2f070a47-bba9-44fc-9e44-ef67338672ab
                    display_name: k8s-ns-user-1-pod-network
                    name: k8s-ns-user-1-pod-network
                    network_ipam_refs: - [
                        + { ... }
                    ]
                }
            }
        -
```

Figure 6.6 Ingress Load Balancer Object

The highlights in Figure 6.6 are:

- The loadbalancer_provider is opencontrail
- The ingress load balancer has a reference to a virtual-machine-interface (VMI) object
- And the VMI object is referred to by an instance-ip object with a (fixed) IP 10.47.255.238 and a floating-ip object with a (floating) IP 101.101.101.1.

And you can explain the ingress IP 10.47.255.238 seen in the ingress as:

- It is a cluster-internal IP address allocated from the default pod network as a load balancer VIP
- It is the frontend IP that the ingress load balancer will listen to for HTTP requests
- And it is also what the public floating IP 101.101.101.1 maps to with NAT.

TIP This book refers to this private IP by different names that are used interchangeably, namely: ingress internal IP, ingress internal VIP, ingress private IP, ingress load balancer interface IP, etc., to differentiate it from the ingress public floating IP. You can also name it as ingress pod IP since the internal VIP is allocated from the pod network. Similarly, it refers to the ingress public floating IP as ingress external IP.

Now to compare the different purposes of these two IPs:

- *Ingress pod IP* is the VIP facing other pods inside of a cluster. To reach ingress from inside of the cluster, requests coming from other pods will have their destination IP set to `Ingress` podIP.
- *Ingress floating IP* is the VIP facing the Internet host outside world. To reach ingress from outside of the cluster, requests coming from Internet hosts need to have their destinations IP set to `Ingress FIP`. When the node receives traffic destined to the ingress floating IP from outside of the cluster, the vRouter will translate it into the `Ingress podIP`.

The detailed ingress load balancer object implementation refers to a service instance, and the service instance includes other data structures or references to other objects (VMs, VMIs, etc.). Overall it is more complicated and involves more details than what's been covered in this book. We've tailored some of the details into a high-level overview so that important concepts like *haproxy* and the *two ingress IPs* can at least be understood.

Once an HTTP/HTTPS request arrives at the ingress podIP, internally or externally, the ingress load balancer will do HTTP/HTTPS proxy operations through the haproxy process, and dispatch the requests towards the service and eventually to the backend pod.

We've seen that the haproxy process is running, to examine more details of this proxy operation, let's check its configuration file for details on the running parameters.

Haproxy.conf File

In each (compute) node, under the /var/lib/contrail/Loadbalancer/haproxy/ folder there will be a subfolder for each load balancer UUID. The file structure looks like this:

```
8fd3e8ea-9539-11e9-9e54-0050569e6cfc
├── haproxy.conf
├── haproxy.pid
└── haproxy.sock
```

You can check the haproxy.conf file for the haproxy configuration:

```
$ cd /var/lib/contrail/loadbalancer/haproxy/8fd3e8ea-9539-11e9-9e54-0050569e6cfc/
$ cat haproxy.conf
global
        daemon
        user haproxy
        group haproxy
        log /var/log/contrail/lbaas/haproxy.log.sock local0
        log /var/log/contrail/lbaas/haproxy.log.sock local1 notice
        tune.ssl.default-dh-param 2048
        ......
        ulimit-n 200000
        maxconn 65000
        ......
        stats socket
        /var/lib/contrail/loadbalancer/haproxy/6b48bd8f-a911-11e9-8112-0050569e6cfc/haproxy.sock
            mode 0666 level user

defaults
        log global
        retries 3
        option redispatch
        timeout connect 5000
        timeout client 300000
        timeout server 300000

frontend f3a7a6a6-5c6d-4f78-81fb-86f6f1b361cf
        option tcplog
        bind 10.47.255.238:80                                   #<---
        mode http                                               #<---
        option forwardfor
        default_backend b45fb570-bec5-4208-93c9-ba58c3a55936    #<---

backend b45fb570-bec5-4208-93c9-ba58c3a55936                    #<---
        mode http                                               #<---
        balance roundrobin
        option forwardfor
        server 4c3031bb-e2bb-4727-a1c7-95afc580bc77 10.97.226.91:8888 weight 1
                                                    ^^^^^^^^^^^^^^^^^
```

The configuration is simple, and Figure 6.7 illustrates it. The highlights of Figure 6.7 are:

- The haproxy frontend represents the frontend of an ingress, facing clients.
- The haproxy backend represents the backend of an ingress, facing services.
- The haproxy frontend defines a bind to the ingress podIP and mode http. These knobs indicate what the frontend is listening to.
- The haproxy backend section defines the server, which is a backend service in our case. It has a format of serviceIP:servicePort, which is the exact service object we've created using the all-in-one YAML file.
- The default_backend in the frontend section defines which backend is the default: it will be used when a haproxy receives a URL request that has no explicit match anywhere else in the frontend section. In this case the default_backend refers to only the backend service 10.97.226.91:8888 This is due to the fact that there are no rules defined in single service Ingress, so all HTTP requests will go to the same default_backend service, regardless of what URL the client sent.

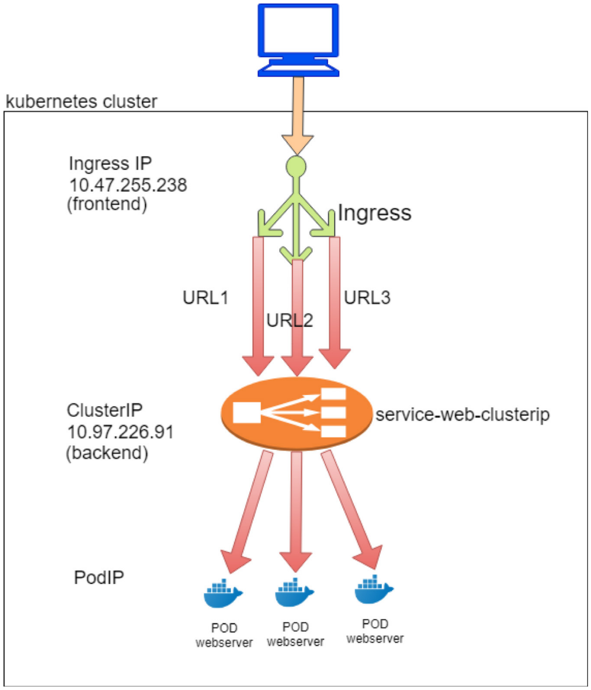


Figure 6.7 Single Service Ingress

NOTE Later, in the simple fanout Ingress and name-based virtual hosting Ingress examples, you will see another type of configuration statement `use_backend…if…` that can be used to force each URL to go to a different backend.

Throughout this configuration, the haproxy implemented our single service ingress.

Gateway Router VRF Table

We've explored a lot inside of the cluster, so now let's look at the gateway router's VRF table:

```
labroot@camaro> show route table k8s-test protocol bgp

k8s-test 7 destinations, 7 routes (7 active, 0 holddown, 0 hidden)
@ = Routing Use Only, # = Forwarding Use Only
+ = Active Route, - = Last Active, * = Both

101.101.101.1/32   *[BGP/170] 02:46:13, MED 100, localpref 200, from 10.169.25.19
                      AS path: ?
                    validation-state: unverified, > via gr-2/2/0.32771, Push 61
```

Same as in the service example, from outside of the cluster, only floating IP is visible. Running the detailed version of the `show` command conveys more information:

```
labroot@camaro> show route table k8s-test 101.101.101.1 detail

k8s-test 24 destinations, 49 routes (24 active, 0 holddown, 0 hidden)
101.101.101.1/32 (1 entry, 1 announced)
        *BGP    Preference: 170/-201
                Route Distinguisher: 10.169.25.20:5     #<---
                ......
                Source: 10.169.25.19
                Next hop: via gr-2/2/0.32771, selected
                Label operation: Push 61
                Label TTL action: prop-ttl
                Load balance label: Label 61: None;
                ......
                Protocol next hop: 10.169.25.20         #<---
                Label operation: Push 61
                Label TTL action: prop-ttl
                Load balance label: Label 61: None;
                Indirect next hop: 0x900d320 1048597 INH Session ID: 0x6f9
                State: <Secondary Active Int Ext ProtectionCand>
                Local AS: 13979 Peer AS: 60100
                Age: 34         Metric: 100     Metric2: 0
                Validation State: unverified
                Task: BGP_60100_60100.10.169.25.19
                Announcement bits (1): 1-KRT
                AS path: ?
                Communities: target:500:500 target:64512:8000016
                Import Accepted
                VPN Label: 61
                Localpref: 200                          #<---
                Router ID: 10.169.25.19
```

The show detail reveals:

- The vRouter advertises the floating IP prefix to Contrail Controller through XMPP. At least two pieces of information from the output indicate who represents the floating IP in this example - node cent222:
  - The Protocol next hop being 10.169.25.20
  - The Route Distinguisher being 10.169.25.20:5
- And through MP-BGP, Contrail Controller reflects the floating IP prefix to the gateway router. Source: 10.169.25.19 indicates this fact.

So, it looks like cent222 is selected to be the active haproxy node and the other node, cent333, is the standby one. Therefore you should expect a client request coming from the Internet host to go to node cent222 first. Of course, the overlay traffic will be carried in MPLS over the GRE tunnel, same as what you've seen from the service example.

The floating IP advertisement towards the gateway router is exactly the same in all types of ingresses.

Another fact that we've somewhat skipped on purpose is the different local preference value used by the active and standby node when advertising the floating IP prefix. A complete examination involves other complex topics, like the active node selection algorithm, and so on, but it is worth it to understand this from a high level.

Both nodes have load balancer and haproxy running, so both will advertise the floating IP prefix 101.101.101.1 to the gateway router. However, they are advertised with different local preference values. The active node will advertise with a value of 200 and the standby node with 100. Contrail Controller both have routes from the two nodes, but only the winning one will be advertised to the gateway router. That is why the other BGP route is dropped and only one is displayed. Localpref being 200 proves it is coming from the active compute node. This applies to both the ingress public floating IP route and the internal VIP route advertisement.

Ingress Verification: Internal

After exploring a lot about ingress load balancer and the related service, pod objects, etc., it's time to verify the end-to-end test result. Since the Ingress serves both inside and outside of the cluster, our verification will start from the client pod inside of the cluster and then from an Internet host outside of it. First, from the inside of cluster:

```
$ kubectl exec -it client-- \
    curl -H 'Host:www.juniper.net' 10.47.255.238 | w3m -T text/html | cat
                                     Hello
                    This page is served by a Contrail pod
                         IP address = 10.47.255.236
                        Hostname = webserver-846c9ccb8b-9nfdx


$ kubectl exec -it client -- \
    curl -H 'Host:www.cisco.com' 10.47.255.238 | w3m -T text/html | cat
                                     Hello
                    This page is served by a Contrail pod
                         IP address = 10.47.255.236
                        Hostname = webserver-846c9ccb8b-9nfdx


$ kubectl exec -it client -- \
    curl -H 'Host:www.google.com' 10.47.255.238 | w3m -T text/html | cat
                                     Hello
                    This page is served by a Contrail pod
                         IP address = 10.47.255.236
                        Hostname = webserver-846c9ccb8b-9nfdx

$ kubectl exec -it client -- \
    curl 10.47.255.238:80 | w3m -T text/html | cat
                                     Hello
                    This page is served by a Contrail pod
                         IP address = 10.47.255.236
                        Hostname = webserver-846c9ccb8b-9nfdx
```

You still use the `curl` command to trigger HTTP requests towards the ingress's private IP. The return proves our Ingress works: requests towards different URLs are all proxied to the same backend pods, through the default backend service, service-web-clusterip.

In the fourth request we didn't give a URL via -H, so curl will fill host with the request IP address, 10.47.255.238 in this test, and again it goes to the same backend pod and gets the same returned response.

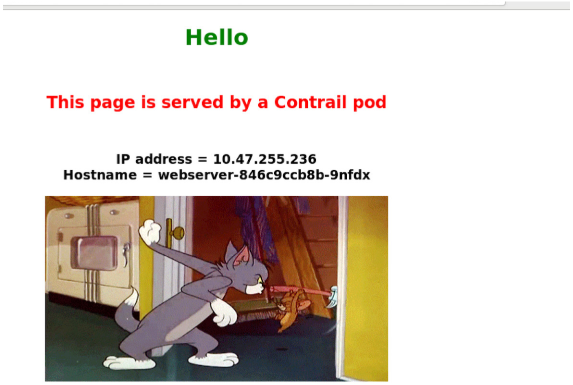
NOTE The -H option is important in ingress tests with curl. It carries the full URL in HTTP payloads that the ingress load balancer is waiting for. Without it the HTTP header will carry Host: 10.47.255.238, which has no matching rule, so it will be treated the same as with an unknown URL.

Ingress Verification: External (Internet host)

The more exciting part of the test is to externally visit the URLs. Overall, we're hoping ingress meant to expose services to the Internet host, even though it does not have to. To make sure the URL resolves to the right floating IP address, you need to update the /etc/hosts file by adding one line at the end – you probably don't want to just end up with a nice webpage from an official website as your test result:

```
# echo "101.101.101.1  www.juniper.net www.cisco.com www.google.com" >> /etc/hosts
# cat /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
101.101.101.1  www.juniper.net www.cisco.com www.google.com     #<---
```

Now, from the Internet host's desktop, launch your browser, and input one of the three URLs. By refreshing the pages you can confirm all HTTP requests are returned by the same backend pod, as shown in Figure 6.8.



Hello

This page is served by a Contrail pod

IP address = 10.47.255.236
Hostname = webserver-846c9ccb8b-9nfdx

Figure 6.8 Accessing www.juniper.net from the Internet Host

The same result can also be seen from curl. The command is exactly the same as what we've been using when testing from a pod, except this time you send requests to the ingress *external* floating IP, instead of the ingress internal podIP. From the Internet host machine:

```
$ curl -H 'Host:www.juniper.net' 101.101.101.1 | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.236
                         Hostname = webserver-846c9ccb8b-9nfdx
                                    [giphy]

$ curl -H 'Host:www.cisco.com' 101.101.101.1 | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.236
                         Hostname = webserver-846c9ccb8b-9nfdx
                                    [giphy]

$ curl -H 'Host:www.google.com' 101.101.101.1 | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.236
                         Hostname = webserver-846c9ccb8b-9nfdx
                                    [giphy]

$ curl 101.101.101.1 | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.236
                         Hostname = webserver-846c9ccb8b-9nfdx
                                    [giphy]
```

Everything works! Okay, next we'll look at the second ingress type simple fanout Ingress. Before going forward, you can take advantage of the all-in-one YAML file and everything can be cleared with one `kubectl delete` command using the same all-in-one YAML file:

```
$ kubectl delete -f ingress/ingress-single-service.yaml
ingress.extensions "ingress-ss" deleted
service "service-web-clusterip" deleted
deployment "webserver" deleted
```

Simple Fanout Ingress

Both the simple fanout Ingress and name-based virtual host Ingress support URL routing, the only difference is that the former is based on path and the latter is based on host.

With simple fanout Ingress, based on the URL path and rules, an ingress load balancer directs traffic to different backend services like so:

```
www.juniper.net/qa --|                 |-> webservice-1
                     |  101.101.101.1  |
www.juniper.net/dev -|                 |-> webservice-2
```

To demonstrate simple fan-out type of ingress, the objects that we need to create are:

- An Ingress object: it defines the rules, mapping two paths to two backend services
- Two backend services objects
- Each service requires at least one pod as a backend
- The same client pod as the cluster-internal client used in previous examples.

Ingress Objects Definition

The goals of the simple fanout Ingress test lab for host www.juniper.net are:

- Requests toward path /dev will be directed to a service webservice-1 with servicePort 8888.
- Requests toward path /qa will be directed to a service webservice-2 with servicePort 8888.

Here is the corresponding YAML file to implement these goals:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-sf
spec:
  rules:
  - host: www.juniper.net
    http:
      paths:
      - path: /dev
        backend:
          serviceName: webservice-1
          servicePort: 8888
      - path: /qa
        backend:
          serviceName: webservice-2
          servicePort: 8888
```

In contrast to single service Ingress, in simple fanout Ingress object (and `name-based virtual host Ingress`) you can see rules defined – here it is the mappings from multiple paths to different backend services.

Backend Service Definition

Since we defined two rules each for a path, you need two services, accordingly. You can clone the previous service in the single service Ingress example and just change that service's name and selector to generate the second service. For example, this is definition of webservice-1 and webservice-2 service:

```
apiVersion: v1
kind: Service
metadata:
  name: webservice-1
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver-1
  #type: LoadBalancer
apiVersion: v1
kind: Service
metadata:
  name: webservice-2
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver-2
  #type: LoadBalancer
```

Backend Pod Definition

Because there are two backend services now, you also need at least two backend pods each with a label matching to a service. You can clone the previous Deployment into two and just change the name and label of the second Deployment.

The `Deployment` for webserver-1:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver-1
  labels:
    app: webserver-1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webserver-1
  template:
    metadata:
      name: webserver-1
      labels:
        app: webserver-1
    spec:
      containers:
      - name: webserver-1
        image: contrailk8sdayone/contrail-webserver
        securityContext:
           privileged: true
        ports:
        - containerPort: 80
```

And the Deployment for webserver-2:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver-2
  labels:
    app: webserver-2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webserver-2
  template:
    metadata:
      name: webserver-2
      labels:
        app: webserver-2
    spec:
      containers:
      - name: webserver-2
        image: contrailk8sdayone/contrail-webserver
        securityContext:
           privileged: true
        ports:
        - containerPort: 80
```

Deploy Simple Fanout Ingress

Just as in the single service Ingress, you put everything together to get an all-in-one YAML file to test simple fanout Ingress:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-sf
spec:
  rules:
  - host: www.juniper.net
    http:
      paths:
      - path: /dev
        backend:
          serviceName: webservice-1
          servicePort: 8888
      - path: /qa
        backend:
          serviceName: webservice-2
          servicePort: 8888
---
apiVersion: v1
kind: Service
metadata:
  name: webservice-1
spec:
  ports:
  - port: 8888
```

```
    targetPort: 80
  selector:
    app: webserver-1
  #type: LoadBalancer
---
apiVersion: v1
kind: Service
metadata:
  name: webservice-2
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver-2
  #type: LoadBalancer
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver-1
  labels:
    app: webserver-1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webserver-1
  template:
    metadata:
      name: webserver-1
      labels:
        app: webserver-1
    spec:
      containers:
      - name: webserver-1
        image: contrailk8sdayone/contrail-webserver
        securityContext:
           privileged: true
        ports:
        - containerPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver-2
  labels:
    app: webserver-2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webserver-2
  template:
    metadata:
      name: webserver-2
      labels:
        app: webserver-2
    spec:
      containers:
      - name: webserver-2
        image: contrailk8sdayone/contrail-webserver
        securityContext:
           privileged: true
        ports:
        - containerPort: 80
```

Now apply the all-in-one YAML file to create all objects:

```
$ kubectl apply -f ingress/ingress-simple-fanout.yaml
ingress.extensions/ingress-sf created
service/webservice-1 created
service/webservice-2 created
deployment.extensions/webserver-1 created
deployment.extensions/webserver-2 created
```

The `ingress`, two service, and two Deployment objects are now created.

Ingress Post Examination

Let's look at the Kubernetes objects created from the all-in-one YAML file. First, the ingress objects:

```
$ kubectl get ingresses.extensions
NAME        HOSTS            ADDRESS                      PORTS  AGE
ingress-sf  www.juniper.net  10.47.255.238,101.101.101.1  80     7s

$ kubectl get ingresses.extensions -o yaml
apiVersion: v1
items:
- apiVersion: extensions/v1beta1
  kind: Ingress
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
        {"apiVersion":"extensions/v1beta1","kind":"Ingress","metadata":{"annotations":{},"name":"ing
ress-sf","namespace":"ns-user-1"},"spec":{"rules":[{"host":"www.juniper.net","http":{"paths":[{"back
end":{"serviceName":"webservice-1","servicePort":8888},"path":"/dev"},{"backend":{"serviceName":"web
service-2","servicePort":8888},"path":"/qa"}]}}]}}
    creationTimestamp: 2019-08-13T06:00:28Z
    generation: 1
    name: ingress-sf
    namespace: ns-user-1
    resourceVersion: "860530"
    selfLink: /apis/extensions/v1beta1/namespaces/ns-user-1/ingresses/ingress-sf
    uid: a6e801fd-bd8f-11e9-9072-0050569e6cfc
  spec:
    rules:
    - host: www.juniper.net
      http:
        paths:
        - backend:
            serviceName: webservice-1
            servicePort: 8888
          path: /dev
        - backend:
            serviceName: webservice-2
            servicePort: 8888
          path: /qa
  status:
    loadBalancer:
      ingress:
      - ip: 101.101.101.1
      - ip: 10.47.255.238
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

The rules are defined properly, and within each rule there is a mapping from a path to the corresponding service. You can see the same ingress internal podIP and external floating IP as seen in the previous single service Ingress example:

```
loadBalancer:
  ingress:
  - ip: 101.101.101.1
  - ip: 10.47.255.238
```

That is why, from the gateway router's perspective, there are no differences between all the types of ingress. In all cases, a public floating IP will be allocated to the ingress and it is advertised to the gateway router:

```
labroot@camaro> show route table k8s-test protocol bgp

k8s-test 7 destinations, 7 routes (7 active, 0 holddown, 0 hidden)
@ = Routing Use Only, # = Forwarding Use Only
+ = Active Route, - = Last Active, * = Both

101.101.101.1/32   *[BGP/170] 02:46:13, MED 100, localpref 200, from 10.169.25.19
                      AS path: ?
                   validation-state: unverified, > via gr-2/2/0.32771, Push 61
```

Now, check the backend services and pods. First the service objects:

```
$ kubectl get svc -o wide
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE   SELECTOR
webservice-1   ClusterIP   10.96.51.227    <none>        8888/TCP   68d   app=webserver-1
webservice-2   ClusterIP   10.100.156.38   <none>        8888/TCP   68d   app=webserver-2
```

Then the backend and client pods:

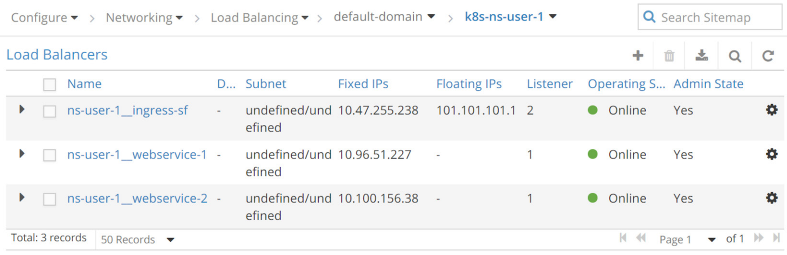
```
$ kubectl get pod -o wide
NAME                          READY  STATUS   ... AGE  IP             NODE    ..
client                        1/1    Running  ... 44d  10.47.255.237  cent222 ..
webserver-1-846c9ccb8b-wns77  1/1    Running  ... 13m  10.47.255.236  cent333 ..
webserver-2-846c9ccb8b-t75d8  1/1    Running  ... 13m  10.47.255.235  cent333 ..

$ kubectl get pod -o wide -l app=webserver-1
NAME                          READY  STATUS   ... AGE  IP             NODE    ..
webserver-1-846c9ccb8b-wns77  1/1    Running  ... 156m 10.47.255.236  cent333 ..
```

```
$ kubectl get pod -o wide -l app=webserver-2
NAME                          READY  STATUS    ... AGE  IP             NODE    ..
webserver-2-846c9ccb8b-t75d8  1/1    Running  ... 156m 10.47.255.235  cent333 ..
```

Two services are created, each with a different allocated clusterIP. For each service there is a backend pod. Later, when we verify ingress from the client, we'll see these podIPs in the returned web pages.

Contrail Ingress Load Balancer Object

Compared with single service Ingress, the only difference is one more service load balancer as shown in the next screen capture.


Configure ▾ > Networking ▾ > Load Balancing ▾ > default-domain ▾ > k8s-ns-user-1 ▾
Search Sitemap
Load Balancers

	□	Name	D...	Subnet	Fixed IPs	Floating IPs	Listener	Operating S...	Admin State	
▸	□	ns-user-1__ingress-sf	-	undefined/undefined	10.47.255.238	101.101.101.1	2	● Online	Yes	⚙
▸	□	ns-user-1__webservice-1	-	undefined/undefined	10.96.51.227	-	1	● Online	Yes	⚙
▸	□	ns-user-1__webservice-2	-	undefined/undefined	10.100.156.38	-	1	● Online	Yes	⚙

Total: 3 records
50 Records ▾
Page 1 ▾ of 1

Figure 6.9 Simple Fanout Ingress Load Balancers (UI: configuration > Networking > Floating IPs)

The three load balancers generated in this test are:

- Load balancer ns-user-1__ingress-sf for ingress ingress-sf
- Load balancer ns-user-1__webservice-1 for service webserver-1
- Load balancer ns-user-1__webservice-2 for service webserver-2

We won't explore the details of the objects again since we've investigated the key parameters of service and Ingress load balancers in single service Ingress and there is really nothing new here.

Haproxy Process and Haproxy.cfg File

In the single service Ingress example, we demonstrated two haproxy processes invoked by contrail-svc-monitor when it sees loadbalancer appearing with loadbalancer_provider and set to opencontrail. At the end of that example, after we removed the single service Ingress, since there is no more Ingress left in the cluster, the two haproxy processes will be ended. Now, with a new ingress creation, the two new haproxy processes are invoked again:

Node `cent222`:

```
$ ps aux | grep haproxy
188   29706  0.0  0.0  55572   2940  ?      Ss  04:04  0:00  haproxy
    -f /var/lib/contrail/loadbalancer/haproxy/b32780cd-ae02-11e9-9c97-002590a54583/haproxy.conf
    -p /var/lib/contrail/loadbalancer/haproxy/b32780cd-ae02-11e9-9c97-002590a54583/haproxy.pid
    -sf  29688
```

Node `cent333`:

```
[root@b4s42 ~]# ps aux | grep haproxy
188   1936  0.0  0.0  55572   896  ?      Ss  04:04  0:00  haproxy
    -f /var/lib/contrail/loadbalancer/haproxy/b32780cd-ae02-11e9-9c97-002590a54583/haproxy.conf
    -p /var/lib/contrail/loadbalancer/haproxy/b32780cd-ae02-11e9-9c97-002590a54583/haproxy.pid
    -sf  1864
```

This time what interests us is how the simple fanout Ingress rules are programmed in the haproxy.conf file. Let’s look at the haproxy configuration file:

```
$ cd /var/lib/contrail/loadbalancer/haproxy/b32780cd-ae02-11e9-9c97-002590a54583
$ cat haproxy.conf
global
    daemon
    user haproxy
    group haproxy
    log /var/log/contrail/lbaas/haproxy.log.sock local0
    log /var/log/contrail/lbaas/haproxy.log.sock local1 notice
    tune.ssl.default-dh-param 2048
    ssl-default-bind-ciphers ECDH+AESGCM:DH+AESGCM:ECDH+AES256:DH+AES256:......
    ulimit-n 200000
    maxconn 65000
    stats socket
        /var/lib/contrail/loadbalancer/haproxy/b32780cd-ae02-11e9-9c97-002590a54583/haproxy.sock
        mode 0666 level user

defaults
    log global
    retries 3
    option redispatch
    timeout connect 5000
    timeout client 300000
    timeout server 300000

frontend acd9cb38-30a7-4eb1-bb2e-f7691e312625
    option tcplog
    bind 10.47.255.238:80
    mode http
    option forwardfor


    acl 46f7e7da-0769-4672-b916-21fdd15b9fad_host hdr(host) -i www.juniper.net
    acl 46f7e7da-0769-4672-b916-21fdd15b9fad_path path /dev
    use_backend 46f7e7da-0769-4672-b916-21fdd15b9fad if
        46f7e7da-0769-4672-b916-21fdd15b9fad_host
        46f7e7da-0769-4672-b916-21fdd15b9fad_path

    acl 020e371c-e222-400f-b71f-5909c93132de_host hdr(host) -i www.juniper.net
    acl 020e371c-e222-400f-b71f-5909c93132de_path path /qa
    use_backend 020e371c-e222-400f-b71f-5909c93132de if
        020e371c-e222-400f-b71f-5909c93132de_host
        020e371c-e222-400f-b71f-5909c93132de_path

backend 46f7e7da-0769-4672-b916-21fdd15b9fad
    mode http
    balance roundrobin
    option forwardfor
    server d58689c2-9e59-494b-bffd-fb7a62b4e17f 10.96.51.227:8888 weight 1

backend 020e371c-e222-400f-b71f-5909c93132de
    mode http
    balance roundrobin
    option forwardfor
    server c13b0d0d-6e4a-4830-bb46-2377ba4caf23 10.100.156.38:8888 weight 1
```

NOTE The configuration file is formatted slightly to make it fit to a page width.

The configuration looks a little bit more complicated than the one for single service Ingress, but the most important part of it looks pretty straightforward:

- The haproxy frontend section: It now defines URLs. Each URL is represented by a pair of acl statements, one for the host, and the other for the path. In a nutshell, host is the domain name and path is what follows the host in the URL string. Here, for simple fanout Ingress there is the host www.juniper.net with two different paths: \dev and \qa.
- The haproxy backend section: Now there are two of them. For each path there is a dedicated service.
- The `use_backend…if…` command in the frontend section: This statement declares the ingress rules – if the URL request includes a specified path that matches to what is programmed in one of the two ACL pairs, use the corresponding backend (that is a service) to forward the traffic.

For example, acl 020e371c-e222-400f-b71f-5909c93132de_path path /qa defines path /qa. If the URL request contains such a path, haproxy will `use_backend` 020e371c-e222-400f-b71f-5909c93132de, which you can find in the backend section. The backend is a UUID referring to server c13b0d0d-6e4a-4830-bb46-2377ba4caf23 10.100.156.38:8888 weight 1, which is essentially a service. You can identify this by looking at the serviceIP:port: 10.100.156.38:8888.

The configuration file is illustrated in Figure 6.10.

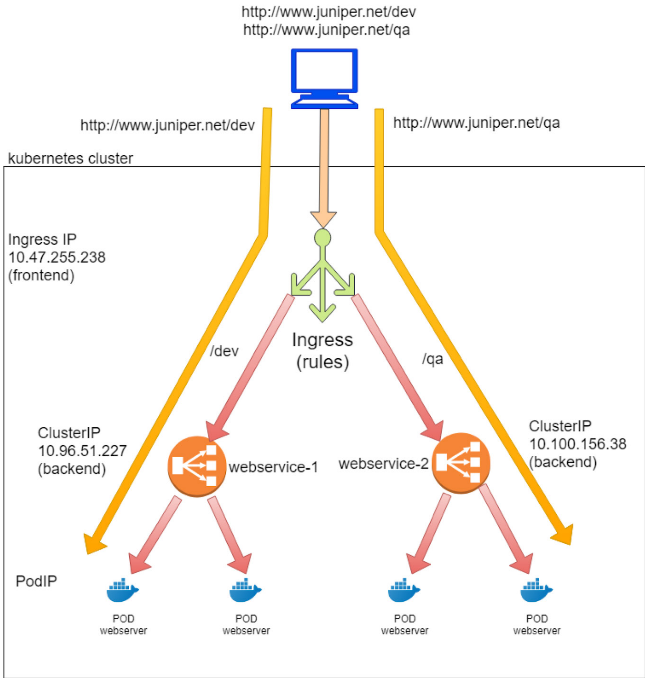


Figure 6.10 Simple Fanout Service

With this proxy.conf file, the haproxy implements our simple fanout Ingress:

- If the full URL is composed of host www.juniper.net and path /dev, the request will be dispatched to webservice-1 (`10.96.51.227:8888`).
- If the full URL is composed of host www.juniper.net and path /qa, the request will be dispatched to webservice-2 (`10.100.156.38:8888`).
- For any other URLs the request will be dropped because there is no corresponding backend service defined for it.

NOTE In practice, you often need the default_backend service to process all those HTTP requests with no matching URLs in the rules. We've seen it in the previous example of single service Ingress. Later in the name-based virtual hosting Ingress section we'll combine the use_backend and default_backend together to provide this type of flexibility.

Ingress Verification: from Internal

Let’s test the URL with different paths:

```
$ kubectl exec -it client -- \
    curl -H 'Host:www.juniper.net' 10.47.255.238/dev | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.236
                         Hostname = webserver-1-846c9ccb8b-wns77


$ kubectl exec -it client -- \
    curl -H 'Host:www.juniper.net' 10.47.255.238/qa | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.235
                        Hostname = Vwebserver-2-846c9ccb8b-t75d8


$ kubectl exec -it client -- \
    curl -H 'Host:www.juniper.net' 10.47.255.238/abc | w3m -T text/html | cat
503 Service Unavailable
No server is available to handle this request.

$ kubectl exec -it client -- \
    curl -H 'Host:www.juniper.net' 10.47.255.238/ | w3m -T text/html | cat
503 Service Unavailable
No server is available to handle this request.

$ kubectl exec -it client -- \
    curl -H 'Host:www.cisco.com' 10.47.255.238/ | w3m -T text/html | cat
503 Service Unavailable
No server is available to handle this request.
```

The returned output shows the Ingress works: the two requests towards the /qa and /dev paths are proxied to two different backend pods through two backend services: webservice-1 and webservice-2, respectively.

The third request with a path abc composes an unknown URL which does not have a matching service in Ingress configuration, so it won’t be served. It’s the same for the last two requests. Without a path, or with a different host, the URLs become unknown to the ingress so they won’t be served.

You may think that you should be adding more rules to include these scenarios. Doing that works fine, but it’s not scalable – you can never cover all the possible paths and URLs that could come into your server. As we mentioned earlier, one solution is to use the default_backend service to process all other HTTP requests, which happens to be covered in the next example.

Ingress Verification: From External (Internet Host)

When you test simple fanout Ingress from outside of the cluster, the command is the same as what you've done for initiating the HTTP request from inside of a pod, except this time you are initiating from an Internet host. Let's send the HTTP requests to the ingress's public floating IP, instead of its internal podIP. So, from an Internet host machine:

```
$ curl -H 'Host:www.juniper.net' 101.101.101.1/qa | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.235
                        Hostname = Vwebserver-2-846c9ccb8b-t75d8
                                    [giphy]

$ curl -H 'Host:www.juniper.net' 101.101.101.1/dev | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.236
                         Hostname = webserver-846c9ccb8b-wns77
                                    [giphy]

$ curl -H 'Host:www.juniper.net' 101.101.101.1/ | w3m -T text/html | cat
503 Service Unavailable
No server is available to handle this request.

$ curl -H 'Host:www.juniper.net' 101.101.101.1/abc | w3m -T text/html | cat
503 Service Unavailable
No server is available to handle this request.

$ curl -H 'Host:www.cisco.com' 101.101.101.1/dev | w3m -T text/html | cat
503 Service Unavailable
No server is available to handle this request.
```

Virtual Hosting Ingress

Virtual hosting ingress supports routing HTTP traffic to multiple host names at the same IP address. Based on the URL and the rules, an ingress load balancer directs traffic to different backend services, and each service directs traffic to its backend pods, like in this diagram:

```
www.juniper.net --|                 |-> webservice-1
                  |  101.101.101.1  |
www.cisco.com   --|                 |-> webservice-2
```

To demonstrate the virtual host type of ingress, the objects that we need to create are same as the previous simple fanout Ingress:

- An Ingress object: the rules that map two URLs to two backend services
- Two backend services objects
- Each service requires at least one pod as a backend

Ingress Objects Definition

In the *virtual host ingress* test lab, we defined the following rules:

- A request toward URL www.juniper.net will be directed to a service webservice-1 with servicePort 8888.
- A request toward URL www.cisco.com will be directed to a service webservice-2 with servicePort 8888.
- A request toward any URLs other than these two, will be directed to webservice-1 with servicePort 8888. Effectively we want webservice-1 to become the default backend service.

And here is the corresponding YAML definition file:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-vh
spec:
  backend:
    serviceName: webservice-1
    servicePort: 8888
  rules:
    - host: www.juniper.net
      http:
        paths:
          - backend:
              serviceName: webservice-1
              servicePort: 8888
            path: /
    - host: www.cisco.com
      http:
        paths:
          - backend:
              serviceName: webservice-2
              servicePort: 8888
            path: /
```

Backend Service and Pod Definition.

The same exact service and Deployment definition that were used in simple fanout Ingress can be used here. And to be even briefer, here's the all-in-one YAML file:

```
$ cat ingress/ingress-test.yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-vh
spec:
  backend:
    serviceName: webservice-1
    servicePort: 8888
  rules:
    - host: www.juniper.net
      http:
        paths:
          - backend:
              serviceName: webservice-1
              servicePort: 8888
            path: /
    - host: www.cisco.com
      http:
        paths:
          - backend:
              serviceName: webservice-2
              servicePort: 8888
            path: /
---
apiVersion: v1
kind: Service
metadata:
  name: webservice-1
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver-1
---
apiVersion: v1
kind: Service
metadata:
  name: webservice-2
spec:
  ports:
  - port: 8888
    targetPort: 80
  selector:
    app: webserver-2
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver-1
  labels:
    app: webserver-1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webserver-1
  template:
    metadata:
      name: webserver-1
      labels:
        app: webserver-1
    spec:
      containers:
      - name: webserver-1
        image: contrailk8sdayone/contrail-webserver
        securityContext:
            privileged: true
        ports:
        - containerPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver-2
  labels:
    app: webserver-2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webserver-2
  template:
    metadata:
      name: webserver-2
      labels:
        app: webserver-2
    spec:
      containers:
      - name: webserver-2
        image: contrailk8sdayone/contrail-webserver
        securityContext:
           privileged: true
        ports:
        - containerPort: 80
```

Now let's apply the all-in-one YAML file to create `ingress` and the other necessary objects:

```
$ kubectl apply -f ingress/ingress-virtual-host-test.yaml
ingress.extensions/ingress-vh created
service/webservice-1 created
service/webservice-2 created
deployment.extensions/webserver-1 created
deployment.extensions/webserver-2 created
```

You can see that the Ingress, two services, and two Deployment objects have now been created.

Ingress Post Examination

Let's examine the Ingress object:

```
$ kubectl get ingresses.extensions -o wide
NAME        HOSTS                          ADDRESS                       PORTS   AGE
ingress-vh  www.juniper.net,www.cisco.com  10.47.255.238,101.101.101.1  80      8m27s
```

Compared to simple fanout Ingress, this time you can see two hosts instead of one. Each host represents a domain name:

```
$ kubectl get ingresses.extensions -o yaml
apiVersion: v1
items:
```

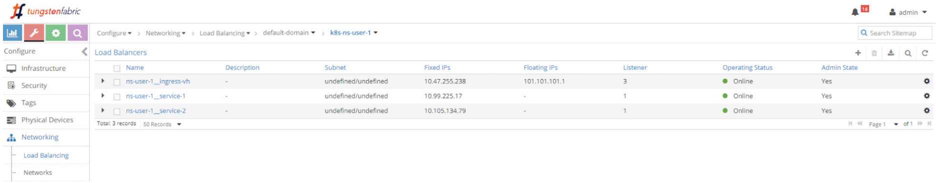
```
- apiVersion: extensions/v1beta1
  kind: Ingress
  metadata:
    ......
    generation: 1
    name: ingress-vh
    namespace: ns-user-1
    resourceVersion: "830991"
    selfLink: /apis/extensions/v1beta1/namespaces/ns-user-1/ingresses/ingress-vh
    uid: 8fd3e8ea-9539-11e9-9e54-0050569e6cfc
  spec:
    backend:
      serviceName: webservice-1
      servicePort: 8888
    rules:
    - host: www.juniper.net
      http:
        paths:
        - backend:
            serviceName: webservice-1
            servicePort: 8888
          path: /
    - host: www.cisco.net
      http:
        paths:
        - backend:
            serviceName: webservice-2
            servicePort: 8888
          path: /
  status:
    loadBalancer:
      ingress:
      - ip: 101.101.101.1
      - ip: 10.47.255.238
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

The rules are defined properly, and within each rule there is a mapping from a host to the corresponding service. Note that the services, pods, and floating IP prefix advertisement to gateway router behavior are all exactly the same as those in simple fanout Ingress.

Exploring Ingress Load Balancer Objects

Three load balancers were generated after we applied the all-in-one YAML file, one for ingress, and two for services.

Load balancers created in this test are almost the same as the ones created in simple fanout Ingress test, as shown in the next screen capture, Figure 6.11.



Load Balancers

	Name	Description	Subnet	Fixed IPs	Floating IPs	Listener	Operating Status	Admin State
□	ns-user-1__ingress-vh	-	undefined/undefined	10.47.255.238	101.101.101.1	3	● Online	Yes
□	ns-user-1__service-1	-	undefined/undefined	10.99.225.17	-	1	● Online	Yes
□	ns-user-1__service-2	-	undefined/undefined	10.105.134.79	-	1	● Online	Yes

Total: 3 records 50 Records Page 1 of 1

Figure 6.11 *Load Balancers*

Okay so let’s check the haproxy configuration file for name-based virtual host Ingress.

Here’s an examination of the `haproxy.conf` file:

```
$ cd /var/lib/contrail/loadbalancer/haproxy/8fd3e8ea-9539-11e9-9e54-0050569e6cfc/
$ cat haproxy.conf
global
        daemon
        user haproxy
        group haproxy
        log /var/log/contrail/lbaas/haproxy.log.sock local0
        log /var/log/contrail/lbaas/haproxy.log.sock local1 notice
        tune.ssl.default-dh-param 2048
        ssl-default-bind-ciphers ECDH+AESGCM:DH+AESGCM:ECDH+AES256:DH+AES256:ECDH+AES128:DH+AES:ECDH
+3DES:DH+3DES:RSA+AESGCM:RSA+AES:RSA+3DES:!aNULL:!MD5:!DSS
        ulimit-n 200000
        maxconn 65000
        stats socket /var/lib/contrail/loadbalancer/haproxy/8fd3e8ea-9539-11e9-9e54-0050569e6cfc/
haproxy.sock mode 0666 level user

defaults
        log global
        retries 3
        option redispatch
        timeout connect 5000
        timeout client 300000
        timeout server 300000

frontend acf8b96d-b322-4bc2-aa8e-0611baa43b9f

        option tcplog
        bind 10.47.255.238:80                  #<---Ingress loadbalancer podIP
        mode http
        option forwardfor

        #map www.juniper.net to backend "xxx4e6a681ec8e6", which maps to "webservice-1"
        acl 77c6ad05-e3cc-4be4-97b2-4e6a681ec8e6_host hdr(host) -i www.juniper.net
        acl 77c6ad05-e3cc-4be4-97b2-4e6a681ec8e6_path path /
        use_backend 77c6ad05-e3cc-4be4-97b2-4e6a681ec8e6 if
            77c6ad05-e3cc-4be4-97b2-4e6a681ec8e6_host
            77c6ad05-e3cc-4be4-97b2-4e6a681ec8e6_path
        #map URL www.cisco.net to backend "xxx44d1ca50a92f", which maps to "webservice-2"
        acl 1e1e9596-85b5-4b10-8e14-44d1ca50a92f_host hdr(host) -i www.cisco.net
        acl 1e1e9596-85b5-4b10-8e14-44d1ca50a92f_path path /
        use_backend 1e1e9596-85b5-4b10-8e14-44d1ca50a92f if
            1e1e9596-85b5-4b10-8e14-44d1ca50a92f_host
            1e1e9596-85b5-4b10-8e14-44d1ca50a92f_path

        #map other URLs, to default backend "xxx4e6a681ec8e6"
        default_backend cd7a7a5b-6c49-4c23-b656-e23493cf7f46

backend 77c6ad05-e3cc-4be4-97b2-4e6a681ec8e6    #<---webservice-1
        mode http
        balance roundrobin
        option forwardfor
        server 33339e1c-5011-4f2e-a276-f8dd37c2cc51 10.99.225.17:8888 weight 1

backend 1e1e9596-85b5-4b10-8e14-44d1ca50a92f    #<---webservice-2
        mode http
        balance roundrobin
        option forwardfor
        server aa0cde60-2526-4437-b943-6f4eaa04bb05 10.105.134.79:8888 weight 1

backend cd7a7a5b-6c49-4c23-b656-e23493cf7f46    #<---default
        mode http
        balance roundrobin
        option forwardfor
        server e8384ee4-7270-4272-b765-61488e1d3e9c 10.99.225.17:8888 weight 1
```

And here are the highlights:

- The haproxy frontend section defines each URL, or host, and its path. Here the two hosts are www.juniper.net and www.cisco.com and for both path is /.
- The haproxy backend section defines the servers, which is all service in our case. It has a format of serviceIP:servicePort, which is what the service created.
- The use_backend…`if`… command in the frontend section declares the ingress rules: if the request includes a specified URL and path, use the corresponding backend to forward the traffic.
- The default_backend defines the service that will act as the default: it will be used when a haproxy receives a URL request that has no explicit match in the defined rules.

The workflow of the configuration file is illustrated in Figure 6.12.

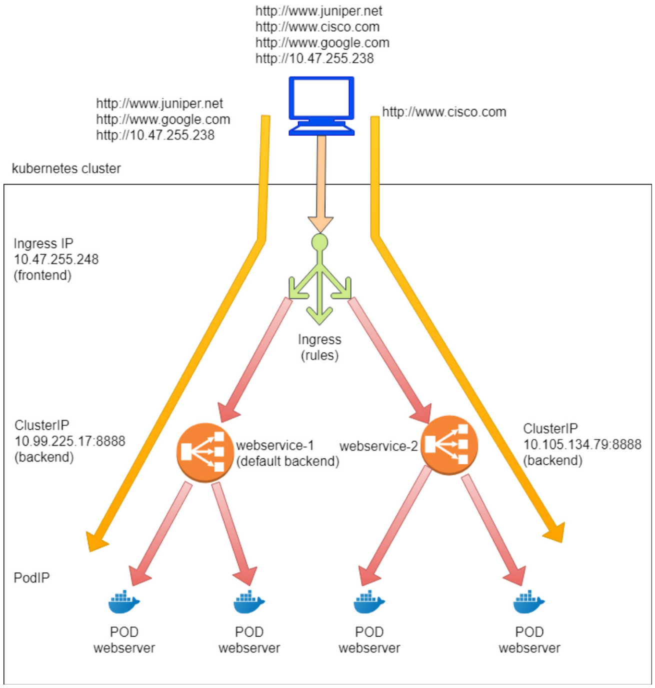


Figure 6.12 The Haproxy Configuration File's Workflow

Through the configuration, the haproxy implements our ingress:

- If www.juniper.net and / compose the full URL, request will be dispatched to webservice-1 (`10.99.225.17:8888`).
- If www.cisco.net and / compose the full URL, request will be dispatched to webservice-2 (`10.105.134.79:8888`).
- Other URLs go to the default backend, which is service webservice-1.

Let's go ahead and verify these behaviors.

Ingress Verification: From Internal

From inside of cluster:

```
$ kubectl exec -it client -- \
    curl -H 'Host:www.juniper.net' 10.47.255.238:80 | w3m -T text/html | cat
                                 Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.236
                        Hostname = Vwebserver-1-846c9ccb8b-g65dg
```

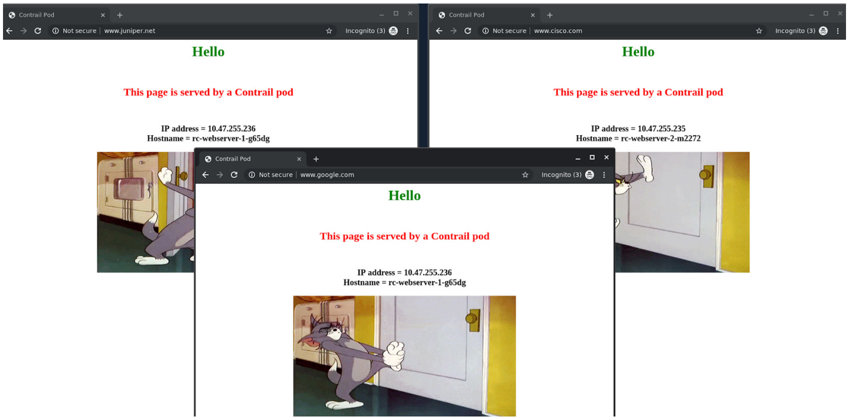
```
$ kubectl exec -it client -- \
    curl -H 'Host:www.cisco.com' 10.47.255.238:80 | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.235
                        Hostname = Vwebserver-2-846c9ccb8b-m2272

$ kubectl exec -it client -- \
    curl -H 'Host:www.google.com' 10.47.255.238:80 | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.236
                        Hostname = Vwebserver-1-846c9ccb8b-g65dg


$ kubectl exec -it client -- \
    curl 10.47.255.238:80 | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.236
                        Hostname = Vwebserver-1-846c9ccb8b-g65dg
```

The Ingress works. The two requests towards Juniper and Cisco are proxied to two different backend pods, through two backend services, webservice-1 and webservice-2, respectively. The third request towards Google is an unknown URL, which does not have a matching service in Ingress configuration, so it goes to the default backend service, webservice-1, and reaches the same backend pod.

The same rule applies to the fourth request. When not given a URL using -H, curl will fill the host with the request IP address, in this case 10.47.255.238. Since that URL doesn't have a defined backend service, the default backend service will be used. In our lab, we use backend pods for each service spawned by the same Deployment, so the podIP in a returned webpage tells us who is who. Except in the second test the returned podIP was 10.47.255.235, representing webservice-2, while the other three tests returned the podIP for webservice-1, as expected.

Ingress Verification: from External (Internet host)

From an Internet host's desktop, we launched two Chrome pages side-by-side and input www.juniper.net in one and www.cisco.com in the other, and kept refreshing the pages. We can confirm that the Juniper page is always returned by the Deployment webserver-1 pod 10.47.255.236, and the Cisco page is always returned by the Deployment webserver-2 pod 10.47.255.235. Then we launched a third Chrome page towards www.google.com, and it was returned by the same pod serving the Juniper instance, as shown by the screen shots in Figure 6.13.


Contrail Pod
Not secure | www.juniper.net
Incognito (3)
Contrail Pod
Not secure | www.cisco.com
Incognito (3)
Contrail Pod
Not secure | www.google.com
Incognito (3)

Hello

This page is served by a Contrail pod

IP address = 10.47.255.236
Hostname = rc-webserver-1-g65dg

Hello

This page is served by a Contrail pod

IP address = 10.47.255.235
Hostname = rc-webserver-2-m2272

Hello

This page is served by a Contrail pod

IP address = 10.47.255.236
Hostname = rc-webserver-1-g65dg

Figure 6.13 Internet Host

The same result can be seen from curl, too. Here it’s shown from the Internet host machine:

```
$ curl -H 'Host:www.juniper.net' 101.101.101.1 | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.236
                        Hostname = Vwebserver-1-846c9ccb8b-g65dg


$ curl -H 'Host:www.cisco.com' 101.101.101.1 | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.235
                        Hostname = Vwebserver-2-846c9ccb8b-m2272

$ curl -H 'Host:www.google.com' 101.101.101.1 | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.236
                        Hostname = Vwebserver-1-846c9ccb8b-g65dg


$ curl 101.101.101.1 | w3m -T text/html | cat
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.236
                        Hostname = Vwebserver-1-846c9ccb8b-g65dg
```

Service Versus Ingress Traffic Flow

Even though both service and ingress are implemented via load balancers (but with different loadbalancer_provider types), the forwarding modes for service and ingress are quite different. With service forwarding it's a one-hop process: the client sends the request to the clusterIP or the floating IP. With NAT, the request reaches the destination backend pod; while with Ingress forwarding, the traffic takes a two-hop process to arrive at the destination pod. The request first goes to the active haproxy, which then starts a HTTP/HTTPS level proxy procedure and does the service forwarding to reach the final pod. NAT processing happens in both forwarding processes, since both ingress and service public floating IP implementation relies on it. Chapter 7 provides a detailed view of this Contrail packet flow.

Chapter 7

Packet Flow in Contrail: End-to-End View

So far, we've looked at floating IP, service, and Ingress in detail, and examined how all these objects are related to each other. In Contrail, both service and ingress are implemented based on load balancers (but with different loadbalancer_provider types). Conceptually, Ingress is designed based on service. The VIP of both types of load balancers are implemented based on floating IP.

Packet Flow

In order to illustrate the detail packet flow in this Contrail Kubernetes environment, let's examine the end-to-end HTTP request from the external Internet host to the destination pod in our Ingress lab setup. We'll examine the forwarding state step-by-step: starting from the Internet host, through the gateway router, then through the active haproxy, backend service, and to the final destination pod.

NOTE Understanding packet flow will enable you to troubleshoot any future forwarding plane issues.

Setup, Utilities, and Tools

You’ve seen Figure 7.1 before in the Ingress section of Chapter 6.

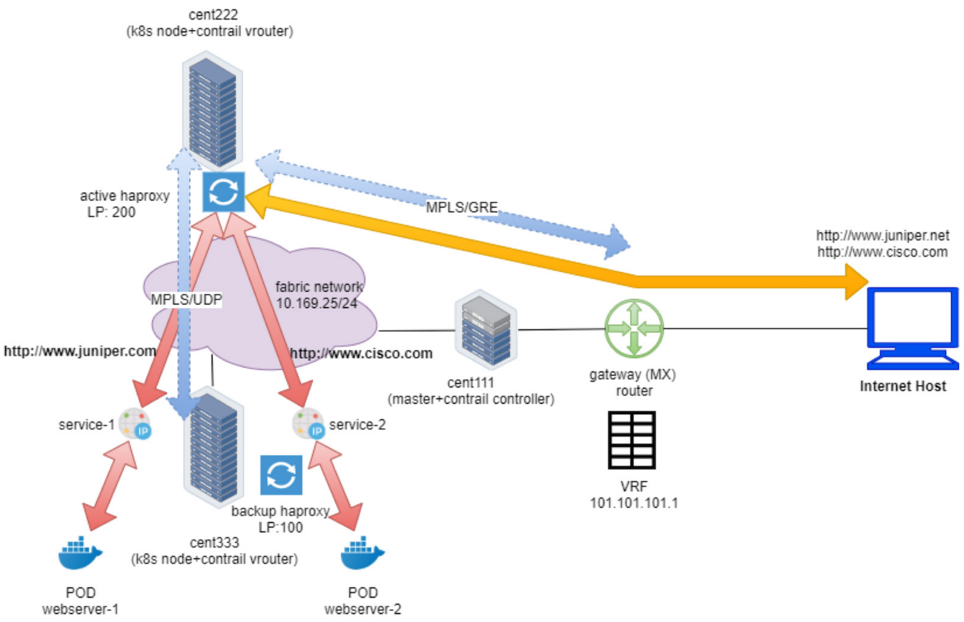


Figure 7.1 Ingress Traffic Flow: Access from External

Earlier, we looked at the external gateway router’s VRF routing table and used the protocol next hop information to find out which node gets the packet from the client. In practice, you need to find out the same from the cluster and the nodes themselves. A Contrail *cluster* typically comes with a group of built-in utilities that you can use to inspect the packet flow and forwarding state. In the service examples you saw the usage of `flow`, `nh`, `vif`, etc., and in this chapter we’ll revisit these utilities and introduce some more that can demonstrate additional information about packet flow.

Some of the available utilities/tools that are used:

- On any Linux machine:
  - curl (with debug option), telnet as HTTP client tool
  - tcpdump and wireshark as packet capture tool
  - shell script can be used to automate command line tasks
- On the vRouter: flow/rt/nh/vif and etc.

Curl

One behavior in the curl tool implementation is that it will always close the TCP session right after the HTTP response has been returned when running in a shell terminal. Although this is safe and clean behavior in practice, it may bring some difficulties to our test. So in this lab we actually held the TCP connection to look into the details. However, a TCP flow entry in Contrail vRouter is bound to the TCP connection, and when the TCP session closes the flow will be cleared. The problem is that curl gets its job done too fast. It establishes the TCP connection, sends the HTTP request, gets the response, and closes the session. Its process is too fast to allow us any time to capture anything with the vRouter utilities (e.g. flow command). As soon as you hit enter to start the curl command, the command returns in less than one or two seconds.

Some workarounds are:

- Large file transfer: One method is to install a large file in the webserver and try to pull it with `curl`, that way the file transfer process holds the TCP session. We've seen this method in the service section in Chapter 3.
- Telnet: You can also make use of the telnet protocol. Establish the TCP connection toward the URL's corresponding IP and port, and then manually input a few HTTP commands and headers to trigger the HTTP request. Doing this allows you some period of time before the haproxy times out and takes down the TCP connection toward the client.

However, please note that haproxy may still tear down its session immediately toward the backend pod. How haproxy behaves varies depending on its implementation and configurations.

From the Internet host, telnet to Ingress public FIP 101.101.101.1 and port 80:

```
[root@cent-client ~]# telnet 101.101.101.1 80
Trying 101.101.101.1...
Connected to 101.101.101.1.
Escape character is '^]'.
```

The TCP connection is established (we'll check what is at the other end in a while). Next, send the HTTP GET command and host header:

```
GET / HTTP/1.1
Host: www.juniper.net
```

This basically sends a HTTP GET request to retrieve data and the Host provides the URL of the request. One more return indicates the end of the request, which triggers an immediate response from the server:

```
HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 359
```

```
Server: Werkzeug/0.12.1 Python/2.7.12
Date: Mon, 02 Sep 2019 04:05:44 GMT
Connection: keep-alive


<html>
<style>
  h1   {color:green}
  h2   {color:red}
</style>
  <div align="center">
  <head>
    <title>Contrail Pod</title>
  </head>
  <body>
    <h1>Hello</h1><br><h2>This page is served by a <b>Contrail</b>
    pod</h2><br><h3>IP address = 10.47.255.236<br>Hostname =
    webserver-1-846c9ccb8b-g65dg</h3>
    <img src="/static/giphy.gif">
  </body>
  </div>
</html>
```

From now on you can collect the flow table in the active haproxy compute node for later analysis.

Shell Script

The third useful tool is a script with which you can automate the test process and repeat the `curl` and `flow` command at the same time over and over. With a small shell script in compute node to collect flow table periodically, and another script in the Internet host to keep sending request with curl, over time you will have a good chance to have the flow table captured in compute node at the right moment.

For instance, the Internet host side script can be:

```
while :; do curl -H 'Host:www.juniper.net' 101.101.101.1; sleep 3; done
```

And the compute side script may look like:

```
while :; do flow --match 10.47.255.238; sleep 0.2; done
```

First the shell one-liner starts a new test every three seconds, then the second one captures a specific flow entry every 0.2 seconds. Twenty tests can be done in two minutes to capture some useful information in a short time.

In this next section we'll use the script method to capture the required information from compute nodes.

Packet Flow Analysis

Earlier we used the curl tool to trigger HTTP requests for our test. It supports extensive options for various features. We've seen the -H option, which specifies the host field in a HTTP request. This time, for debugging purposes, we use another useful option, -v in the curl command:

```
[root@cent-client ~]# curl -vH 'Host:www.juniper.net' 101.101.101.1
* About to connect() to 101.101.101.1 port 80 (#0)
*   Trying 101.101.101.1...
* Connected to 101.101.101.1 (101.101.101.1) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.29.0
> Accept: */*
> Host:www.juniper.net
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: text/html; charset=utf-8
< Content-Length: 359
< Server: Werkzeug/0.12.1 Python/2.7.12
< Date: Tue, 02 Jul 2019 16:50:46 GMT
* HTTP/1.0 connection set to keep alive!
< Connection: keep-alive
<

<html>
<style>
  h1   {color:green}
  h2   {color:red}
</style>
  <div align="center">
  <head>
    <title>Contrail Pod</title>
  </head>
  <body>
    <h1>Hello</h1><br><h2>This page is served by a <b>Contrail</b>
    pod</h2><br><h3>IP address = 10.47.255.236<br>Hostname =
    webserver-1-846c9ccb8b-g65dg</h3>
    <img src="/static/giphy.gif">
  </body>
  </div>
</html>
* Connection #0 to host 101.101.101.1 left intact
```

This option displays more verbose information about the HTTP interaction:

- The > lines are the messages content that curl sent out, and
- The < lines are message content that it receives from remote.

From the interaction you can see:

- The curl sent a HTTP GET with path / to the FIP 101.101.101.1, and with Host filled with juniper URL.
- It gets the response with code 200 OK, indicating the request has succeeded.

- There are a bunch of other headers in the response that are not important for our test so we can skip them.
- The rest of the response is the HTML source code of a returned web page.
- The connection is immediately closed afterward.

Now you've seen the verbose interactions that `curl` performed under the hood, and you can understand the GET command and host header we sent in the telnet test. In that test we were just emulating what curl would do, but just now we did it manually!

Internet Host to Gateway Router

First let's start from the client – the Internet host.

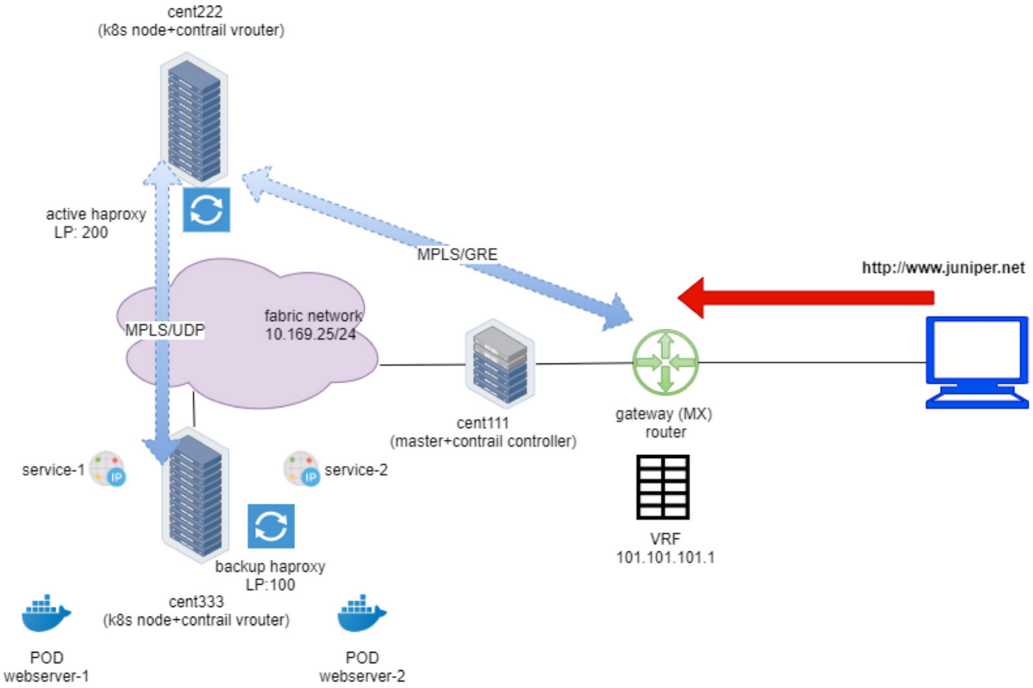


Figure 7.2 Internet Host: Send an HTTP Request

As in any host, the routing table is pretty simple. The static route, or more typically, a default route, pointing to the gateway route is all that it needs:

```
[root@cent-client ~]# ip r
default via 10.85.188.1 dev ens160 proto static metric 100
10.85.188.0/27 dev ens160 proto kernel scope link src 10.85.188.24 metric 100
15.15.15.0/24 dev ens192 proto kernel scope link src 15.15.15.2 metric 101
101.101.101.0/24 via 15.15.15.1 dev ens192      #<---
```

The last entry is the static route that we've manually configured, pointing to our gateway router.

NOTE In this setup, we configured a VRF table in the gateway router to connect the host machine into the same MPLS/VPN so that it can communicate with the overlay networks in Contrail cluster. In practice, there are other ways to achieve the same goal. For example, the gateway router can also choose to leak routes with policies between VPNs and the Internet routing table, so that an Internet host that is not part of the VPNs can also access the overlay networks in Contrail.

Gateway Router to Ingress Public Floating IP: MPLS over GRE

We've seen gateway router's routing table before. From the protocol next hop we can find out that the packet will be sent to active haproxy node cent222 via the MPLSoGRE tunnel.

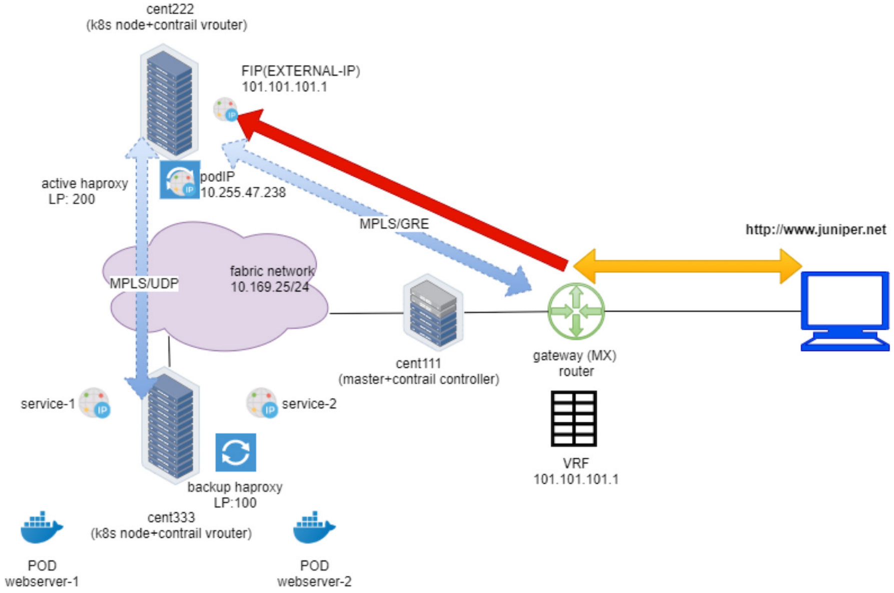


Figure 7.3 Gateway Router: Forward to Ingress Public Floating IP

Now with the flow table collected on both computes, we can find out the same information. Let's take a look at the flow entries of active proxy compute:

```
(vrouter-agent)[root@cent222 /]$ flow --match 15.15.15.2
Flow table(size 80609280, entries 629760)

Entries: Created 586803 Added 586861 Deleted 1308 Changed 1367Processed 586803
Used Overflow entries 0
(Created Flows/CPU: 147731 149458 144549 145065)(oflows 0)

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

Listing flows matching ([15.15.15.2]:*)

    Index                    Source:Port/Destination:Port                      Proto(V)
 -----------------------------------------------------------------------------------
   114272<=>459264       15.15.15.2:42786                                    6 (2->2)
                         101.101.101.1:80
(Gen: 3, K(nh):89, Action:N(D), Flags:, TCP:SSrEEr, QOS:-1, S(nh):61,
 Stats:2/112,  SPort 50985, TTL 0, Sinfo 192.168.0.204)

   459264<=>114272       10.47.255.238:80                                    6 (2->5)
                         15.15.15.2:42786
(Gen: 1, K(nh):89, Action:N(S), Flags:, TCP:SSrEEr, QOS:-1, S(nh):89,
 Stats:1/74,  SPort 60289, TTL 0, Sinfo 8.0.0.0)


(vrouter-agent)[root@cent222 /]$ nh --get 61
Id:61         Type:Tunnel         Fmly: AF_INET  Rid:0  Ref_cnt:5604       Vrf:0
              Flags:Valid, MPLSoGRE, Etree Root,
              Oif:0 Len:14 Data:f0 1c 2d 41 90 00 00 50 56 9e 62 25 08 00
              Sip:10.169.25.20 Dip:192.168.0.204
```

This flow reflects the state of the TCP connection originating from the Internet host client to active haproxy. Let's look at the first entry in the capture:

- The first flow entry displays the source and destination of the HTTP request; it is coming from Internet host (15.15.15.2) and lands the Ingress floating IP in current node cent222.
- The S(nh):61 is the next hop to the source of the request – the Internet host. This is similar to reverse path forwarding(RPF). The vRouter always maintains the path toward the source of the packet in the flow.
- The nh --get command resolves the nexthop 61 with more details. You can see a MPLSoGRE flag is set, Sip and Dip are the two ends of the GRE tunnel, and they are currently the node and the gateway router's loopback, IP respectively.
- The TCP:SSrEEr are TCP flags showing the state of this the TCP connection. The vRouter detects the SYN (S), SYN-ACK (Sr), so the bidirectional connection is established (EEr).

- Proto(V) field indicate the VRF number and protocol type. two VRF is involved here in current (isolated) NS ns-user-1.
  - VRF 2: the VRF of default pod network
  - VRF 5: the VRF of the FIP-VN
  - protocol 6 means TCP (HTTP packets).

TIP We'll use VRF 2 later when we query the nexthop for a prefix in the VRF routing table.

Overall, the first flow entry confirms that the request packet from the Internet host traverses the gateway router, and via the MPLSoGRE tunnel it hits the ingress external VIP 101.101.101.1. NAT will happen and we'll look into that next.

Ingress Public Floating IP to Ingress Pod IP: FIP(NAT)

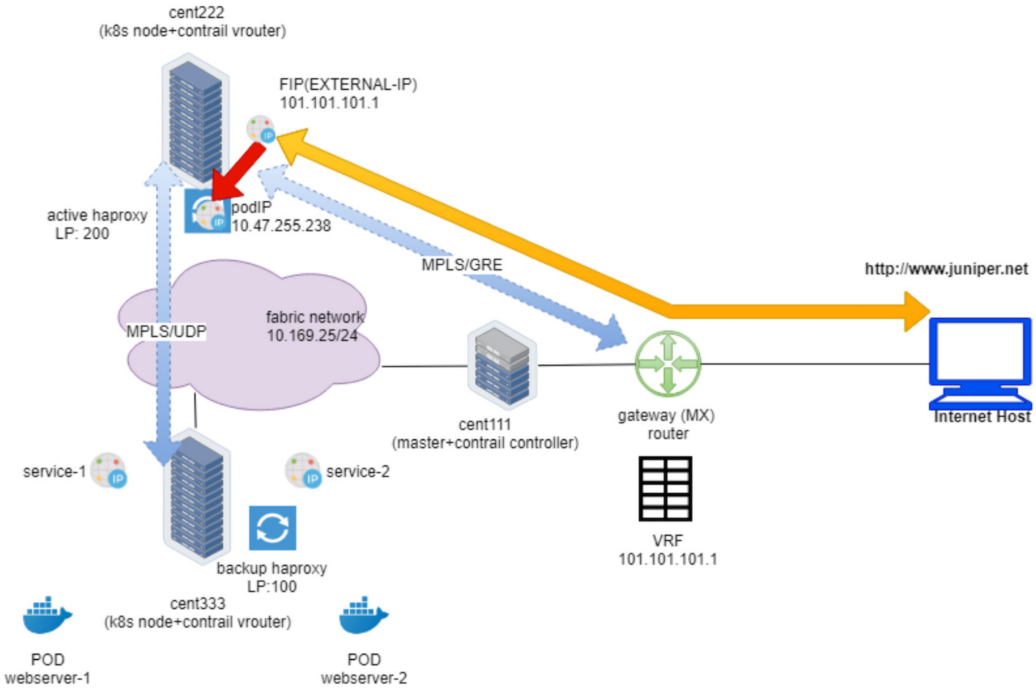


Figure 7.4 Active Haproxy Node: Ingress Public Floating IP to Ingress Pod IP

To verify the NAT operation, you only need to dig a little bit more out of the previous flow output:

- The Action flag, N(D), in the first entry indicates destination NAT or DNAT. Destination ingress external floating IP 101.101.101.1, which is the external ingress, will be translated to the ingress internal VIP.
- The Action flag, N(S), in the second entry, indicates source NAT or SNAT. This indicates source NAT source IP 10.47.255.238, which is the internal ingress, and the VIP will be translated to the ingress external VIP.

In summary, what the flow table of active haproxy node cent222 tells us is that on receiving the packet destined to the ingress floating IP, vRouter on node cent222 performs NAT operation and translates destination floating IP (101.101.101.1) to the ingress's internal VIP (10.47.255.238). After that the packet lands the ingress load balancer's VRF table and forwards it to the active haproxy's listening interface. The HTTP proxy operation will now happen, and we'll talk about it next.

NOTE In vRouter flow, the second flow entry is also called a *reverse flow of the first one*. It is the flow entry vRouter that sends the returning packet towards the Internet host. From the ingress load balancer's perspective it only uses 10.47.255.238, assigned from the default pod network as its source IP, it does not know anything about the floating IP. The same goes for the external Internet host, it only knows how to reach the floating IP and has no clues about the private ingress internal VIP. It is the vRouter that is doing the two-way NAT translations in between.

Ingress Pod IP to Service IP: MPLS over UDP

Now the packet lands in the ingress load balancer's VRF table and it is in the frontend of the haproxy. What happens is:

- The haproxy is listening in on the frontend IP (ingress internal podIP/VIP) and port 80 to see the packet.
- The haproxy checks the ingress rule programmed in its configuration file, decides that the requests need to be proxied to the service IP of webservice-1.
- The vRouter checks the ingress load balancer's VRF table and sees the prefix of webservice-1 and the service IP is learned from a destination node cent333, which will be the next hop to forward the packet.
- Between compute nodes the forwarding path is programmed with MPLSoUDP tunnel, so the vRouter sends it through MPLS over UDP tunnel with the right MPLS Label.

This process is illustrated next in Figure 7.5:

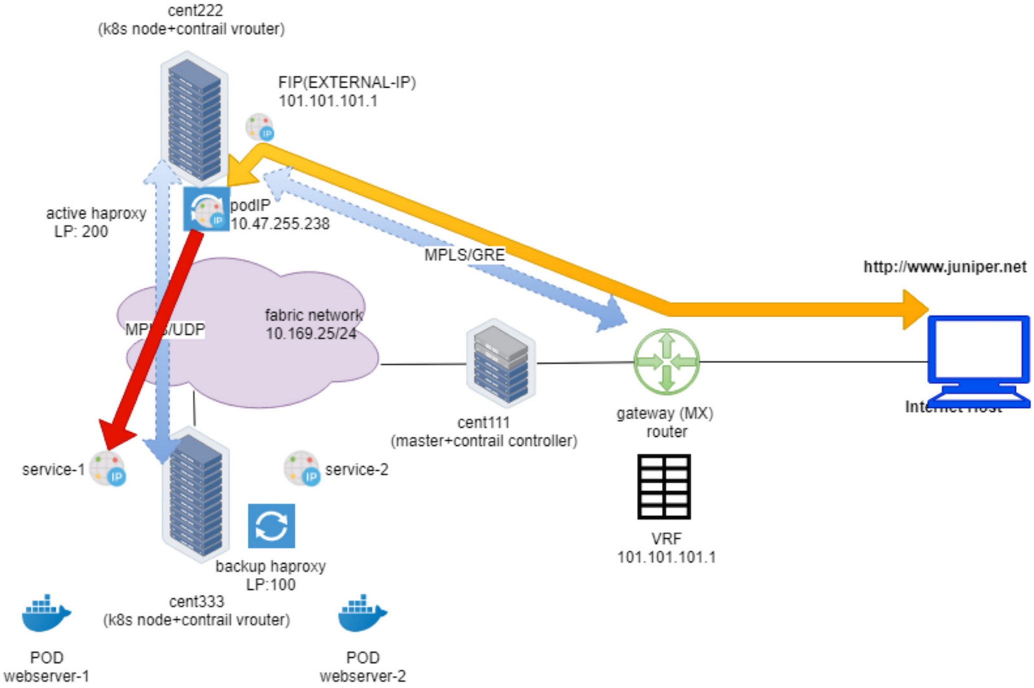
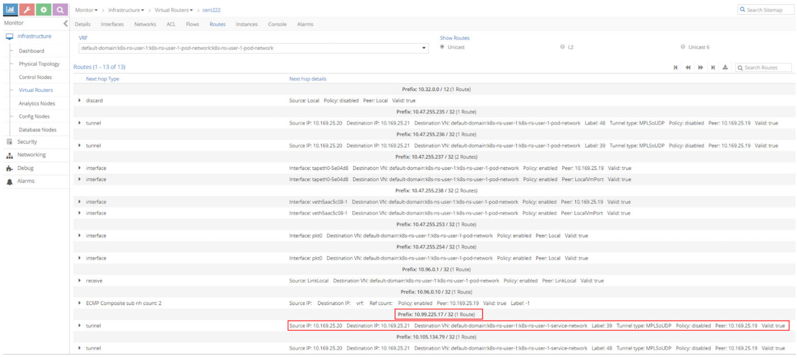


Figure 7.5 Active Haproxy Node: Ingress Pod IP to Service IP

Let's first take a look at the VRF routing table from UI. In UI, we can check the VRF routing table based on the VRF name, from any compute node.



VRF: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network:k8s-ns-user-1-pod-network

Show Routes: ● Unicast ○ L2 ○ Unicast 6

Routes (1 - 13 of 13)

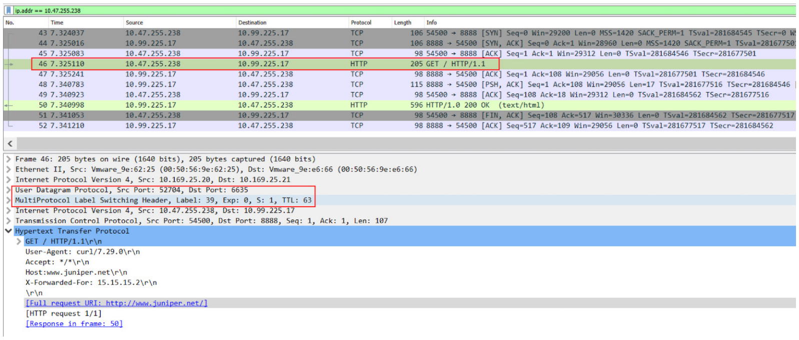
Next hop Type	Next hop details
	Prefix: 10.32.0.0 / 12 (1 Route)
discard	Source: Local Policy: disabled Peer: Local Valid: true
	Prefix: 10.47.255.235 / 32 (1 Route)
tunnel	Source IP: 10.169.25.20 Destination IP: 10.169.25.21 Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network Label: 48 Tunnel type: MPLSoUDP Policy: disabled Peer: 10.169.25.19 Valid: true
	Prefix: 10.47.255.236 / 32 (1 Route)
tunnel	Source IP: 10.169.25.20 Destination IP: 10.169.25.21 Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network Label: 39 Tunnel type: MPLSoUDP Policy: disabled Peer: 10.169.25.19 Valid: true
	Prefix: 10.47.255.237 / 32 (2 Routes)
interface	Interface: tapeth0-5e04d8 Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network Policy: enabled Peer: 10.169.25.19 Valid: true
interface	Interface: tapeth0-5e04d8 Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network Policy: enabled Peer: LocalVmPort Valid: true
	Prefix: 10.47.255.238 / 32 (2 Routes)
interface	Interface: veth5aac5c08-1 Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network Policy: enabled Peer: 10.169.25.19 Valid: true
interface	Interface: veth5aac5c08-1 Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network Policy: enabled Peer: LocalVmPort Valid: true
	Prefix: 10.47.255.253 / 32 (1 Route)
interface	Interface: pkt0 Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network Policy: enabled Peer: Local Valid: true
	Prefix: 10.47.255.254 / 32 (1 Route)
interface	Interface: pkt0 Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network Policy: enabled Peer: Local Valid: true
	Prefix: 10.96.0.1 / 32 (1 Route)
receive	Source: LinkLocal Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network Policy: enabled Peer: LinkLocal Valid: true
	Prefix: 10.96.0.10 / 32 (1 Route)
ECMP Composite sub nh count: 2	Source IP: Destination IP: vrf: Ref count: Policy: enabled Peer: 10.169.25.19 Valid: true Label: -1
	Prefix: 10.99.225.17 / 32 (1 Route)
tunnel	Source IP: 10.169.25.20 Destination IP: 10.169.25.21 Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-service-network Label: 39 Tunnel type: MPLSoUDP Policy: disabled Peer: 10.169.25.19 Valid: true
	Prefix: 10.105.134.79 / 32 (1 Route)
tunnel	Source IP: 10.169.25.20 Destination IP: 10.169.25.21 Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-service-network Label: 48 Tunnel type: MPLSoUDP Policy: disabled Peer: 10.169.25.19 Valid: true

Figure 7.6 Ingress Loadbalancer's VRF Table

From the Ingress podIP's VRF, which is the same VRF for the default pod network of current namespace, we can see that the next hop toward service IP prefix 10.99.225.17/32 is the other compute node cent333 with IP 10.169.25.21 through MPLSoUDP tunnel. The same result can also be found via vRouter rt/nh utilities:

```
$ docker exec -it vrouter_vrouter-agent_1 rt --get 10.99.225.17/32 --vrf 2
Match 10.99.225.17/32 in vRouter inet4 table 0/2/unicast

Flags: L=Label Valid, P=Proxy ARP, T=Trap ARP, F=Flood ARP
vRouter inet4 routing table 0/2/unicast
Destination           PPL        Flags        Label         Nexthop    Stitched MAC(Index)
10.99.225.17/32         0           LP         38             50        -

$ docker exec -it vrouter_vrouter-agent_1 nh --get 50
Id:50          Type:Tunnel         Fmly: AF_INET  Rid:0  Ref_cnt:33         Vrf:0
              Flags:Valid, MPLSoUDP, Etree Root,
              Oif:0 Len:14 Data:00 50 56 9e e6 66 00 50 56 9e 62 25 08 00
              Sip:10.169.25.20 Dip:10.169.25.21
```

Please note that all the traffic from ingress to service happens in the overlay between Contrail compute nodes, which means that all overlay packets should be encapsulated in MPLS over UDP tunnel. To verify the haproxy process packet processing details, let's capture packets on the physical interface of node cent222, where the active haproxy process is running. The next screen capture, Figure 7.7, shows the results:



ip.addr == 10.47.255.238

No.	Time	Source	Destination	Protocol	Length	Info
43	7.324037	10.47.255.238	10.99.225.17	TCP	106	54500 → 8888 [SYN] Seq=0 Win=29200 Len=0 MSS=1420 SACK_PERM=1 TSval=281684545 TSecr=0 W
44	7.325016	10.99.225.17	10.47.255.238	TCP	106	8888 → 54500 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1420 SACK_PERM=1 TSval=28167750
45	7.325083	10.47.255.238	10.99.225.17	TCP	98	54500 → 8888 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=281684546 TSecr=281677501
46	7.325110	10.47.255.238	10.99.225.17	HTTP	205	GET / HTTP/1.1
47	7.325241	10.99.225.17	10.47.255.238	TCP	98	8888 → 54500 [ACK] Seq=1 Ack=108 Win=29056 Len=0 TSval=281677501 TSecr=281684546
48	7.340783	10.99.225.17	10.47.255.238	TCP	115	8888 → 54500 [PSH, ACK] Seq=1 Ack=108 Win=29056 Len=17 TSval=281677516 TSecr=281684546
49	7.340923	10.47.255.238	10.99.225.17	TCP	98	54500 → 8888 [ACK] Seq=108 Ack=18 Win=29312 Len=0 TSval=281684562 TSecr=281677516
50	7.340998	10.99.225.17	10.47.255.238	HTTP	596	HTTP/1.0 200 OK (text/html)
51	7.341053	10.47.255.238	10.99.225.17	TCP	98	54500 → 8888 [FIN, ACK] Seq=108 Ack=517 Win=30336 Len=0 TSval=281684562 TSecr=281677517
52	7.341210	10.99.225.17	10.47.255.238	TCP	98	8888 → 54500 [ACK] Seq=517 Ack=109 Win=29056 Len=0 TSval=281677517 TSecr=281684562

```
> Frame 46: 205 bytes on wire (1640 bits), 205 bytes captured (1640 bits)
> Ethernet II, Src: Vmware_9e:62:25 (00:50:56:9e:62:25), Dst: Vmware_9e:e6:66 (00:50:56:9e:e6:66)
> Internet Protocol Version 4, Src: 10.169.25.20, Dst: 10.169.25.21
> User Datagram Protocol, Src Port: 52704, Dst Port: 6635
> MultiProtocol Label Switching Header, Label: 39, Exp: 0, S: 1, TTL: 63
> Internet Protocol Version 4, Src: 10.47.255.238, Dst: 10.99.225.17
> Transmission Control Protocol, Src Port: 54500, Dst Port: 8888, Seq: 1, Ack: 1, Len: 107
v Hypertext Transfer Protocol
  > GET / HTTP/1.1\r\n
    User-Agent: curl/7.29.0\r\n
    Accept: */*\r\n
    Host:www.juniper.net\r\n
    X-Forwarded-For: 15.15.15.2\r\n
    \r\n
    [Full request URI: http://www.juniper.net/]
    [HTTP request 1/1]
    [Response in frame: 50]
```

Figure 7.7 Packet Capture on Fabric Interface of Active Haproxy Node cent222

From the Wireshark screenshot in Figure 7.7, you can see clearly that:

- Frames 43-45, Ingress private podIP established a new TCP connection toward service IP and port, this happens in overlay.
- Frame 46, on the new TCP connection, haproxy starts a HTTP request to the service IP.
- Frame 50, the HTTP response returns back.

Frame 46 is also the one to use as an example to show the packet encapsulation.

You'll see this IP packet containing the HTTP request is MPLS-labeled, and it is embedded inside of a UDP datagram. The outer source and destination IP of the packet are 10.169.25.20 (compute node cent222) and 10.169.25.21 (compute node cent333), respectively.

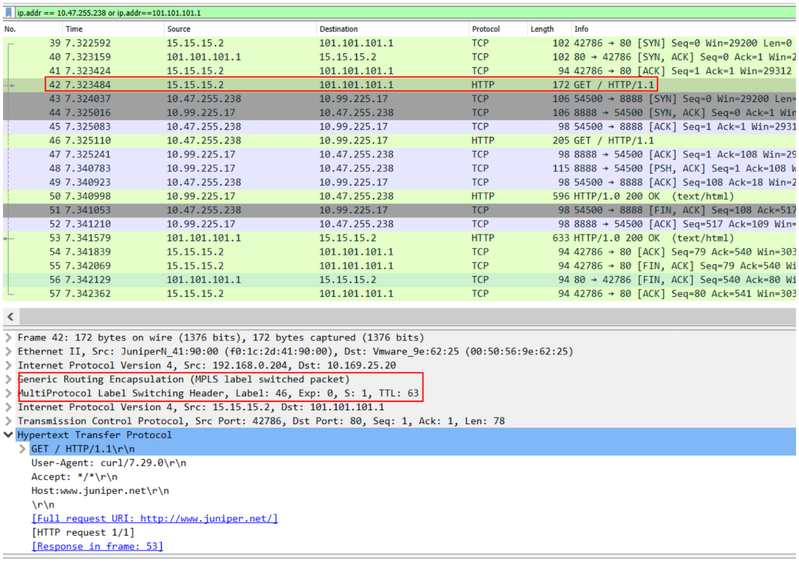
Forward Versus Proxy

If you are observant enough, you should have noticed something weird in this capture. For example:

- Shouldn't the source IP address be the Internet host's IP 15.15.15.2, instead of load balancer's frontend IP?
- Is the original HTTP request forwarded at all?
- Is the transaction within the same TCP session sourcing from Internet host, crossing gateway router and load balancer node `cent222`, all the way down to the backend pod sitting in node `cent333`?

The answer to all of these questions is *No*. The haproxy in this test is doing Layer 7 (Application Layer) load balancing. What it does is:

- Establishes TCP connection with the Internet host and keeps monitoring the HTTP request;
- Whenever it sees a HTTP request coming in, it checks its rules and initiates a brand new TCP connection to the corresponding backend;
- It copies the original HTTP request it receives from the Internet host and pastes into the new TCP connection with its backend. Precisely speaking, the HTTP request is proxied, not forwarded.
- Extending the wireshark display filter to include both 15.15.15.2 and 101.101.101.1.



```
ip.addr == 10.47.255.238 or ip.addr==101.101.101.1

No.       Time        Source            Destination       Protocol  Length  Info
       39 7.322592    15.15.15.2        101.101.101.1     TCP          102 42786 → 80 [SYN] Seq=0 Win=29200 Len=0
       40 7.323159    101.101.101.1     15.15.15.2        TCP          102 80 → 42786 [SYN, ACK] Seq=0 Ack=1 Win=2
       41 7.323424    15.15.15.2        101.101.101.1     TCP           94 42786 → 80 [ACK] Seq=1 Ack=1 Win=29312
       42 7.323484    15.15.15.2        101.101.101.1     HTTP         172 GET / HTTP/1.1
       43 7.324037    10.47.255.238     10.99.225.17      TCP          106 54500 → 8888 [SYN] Seq=0 Win=29200 Len=
       44 7.325016    10.99.225.17      10.47.255.238     TCP          106 8888 → 54500 [SYN, ACK] Seq=0 Ack=1 Win
       45 7.325083    10.47.255.238     10.99.225.17      TCP           98 54500 → 8888 [ACK] Seq=1 Ack=1 Win=2931
       46 7.325110    10.47.255.238     10.99.225.17      HTTP         205 GET / HTTP/1.1
       47 7.325241    10.99.225.17      10.47.255.238     TCP           98 8888 → 54500 [ACK] Seq=1 Ack=108 Win=29
       48 7.340783    10.99.225.17      10.47.255.238     TCP          115 8888 → 54500 [PSH, ACK] Seq=1 Ack=108 W
       49 7.340923    10.47.255.238     10.99.225.17      TCP           98 54500 → 8888 [ACK] Seq=108 Ack=18 Win=2
       50 7.340998    10.99.225.17      10.47.255.238     HTTP         596 HTTP/1.0 200 OK  (text/html)
       51 7.341053    10.47.255.238     10.99.225.17      TCP           98 54500 → 8888 [FIN, ACK] Seq=108 Ack=517
       52 7.341210    10.99.225.17      10.47.255.238     TCP           98 8888 → 54500 [ACK] Seq=517 Ack=109 Win=
       53 7.341579    101.101.101.1     15.15.15.2        HTTP         633 HTTP/1.0 200 OK  (text/html)
       54 7.341839    15.15.15.2        101.101.101.1     TCP           94 42786 → 80 [ACK] Seq=79 Ack=540 Win=303
       55 7.342069    15.15.15.2        101.101.101.1     TCP           94 42786 → 80 [FIN, ACK] Seq=79 Ack=540 Wi
       56 7.342129    101.101.101.1     15.15.15.2        TCP           94 80 → 42786 [FIN, ACK] Seq=540 Ack=80 Wi
       57 7.342362    15.15.15.2        101.101.101.1     TCP           94 42786 → 80 [ACK] Seq=80 Ack=541 Win=303

> Frame 42: 172 bytes on wire (1376 bits), 172 bytes captured (1376 bits)
> Ethernet II, Src: JuniperN_41:90:00 (f0:1c:2d:41:90:00), Dst: Vmware_9e:62:25 (00:50:56:9e:62:25)
> Internet Protocol Version 4, Src: 192.168.0.204, Dst: 10.169.25.20
> Generic Routing Encapsulation (MPLS label switched packet)
> MultiProtocol Label Switching Header, Label: 46, Exp: 0, S: 1, TTL: 63
> Internet Protocol Version 4, Src: 15.15.15.2, Dst: 101.101.101.1
> Transmission Control Protocol, Src Port: 42786, Dst Port: 80, Seq: 1, Ack: 1, Len: 78
v Hypertext Transfer Protocol
  > GET / HTTP/1.1\r\n
    User-Agent: curl/7.29.0\r\n
    Accept: */*\r\n
    Host:www.juniper.net\r\n
    \r\n
    [Full request URI: http://www.juniper.net/]
    [HTTP request 1/1]
    [Response in frame: 53]
```

Figure 7.8 Packet Capture On Active Haproxy Node Cent222 Fabric Interface: The "Whole Story"

- Frame 39-41: Internet host established a TCP connection toward Ingress external public FIP.
- Frame 42: Internet host sent HTTP request.
- Frame 43-52: active haproxy established a new TCP connection toward service, sent the HTTP request, retrieved the HTTP response, and closed the connection.
- Frame 53-54: active haproxy sent the HTTP response back to Internet host.
- Frame 55-57: Internet host closed the HTTP connection.

Here we use frame 42 to display the MPLS over GRE encapsulation between active haproxy node cent222 and the gateway router. When comparing it with frame 46 in the previous screenshot, you will notice this is a different label. The MPLS label carried in the GRE tunnel will be stripped before the vRouter delivers the packet to the active haproxy. A new label will be assigned when active haproxy starts a new TCP session to the remote node.

At the moment we know the HTTP request is proxied to haproxy's backend. According to the ingress configuration, that backend is a Kubernetes service. Now, in order to reach the service, the request is sent to a destination node cent333 where all backend pods are sitting. Next we'll look at what will happen in destination node.

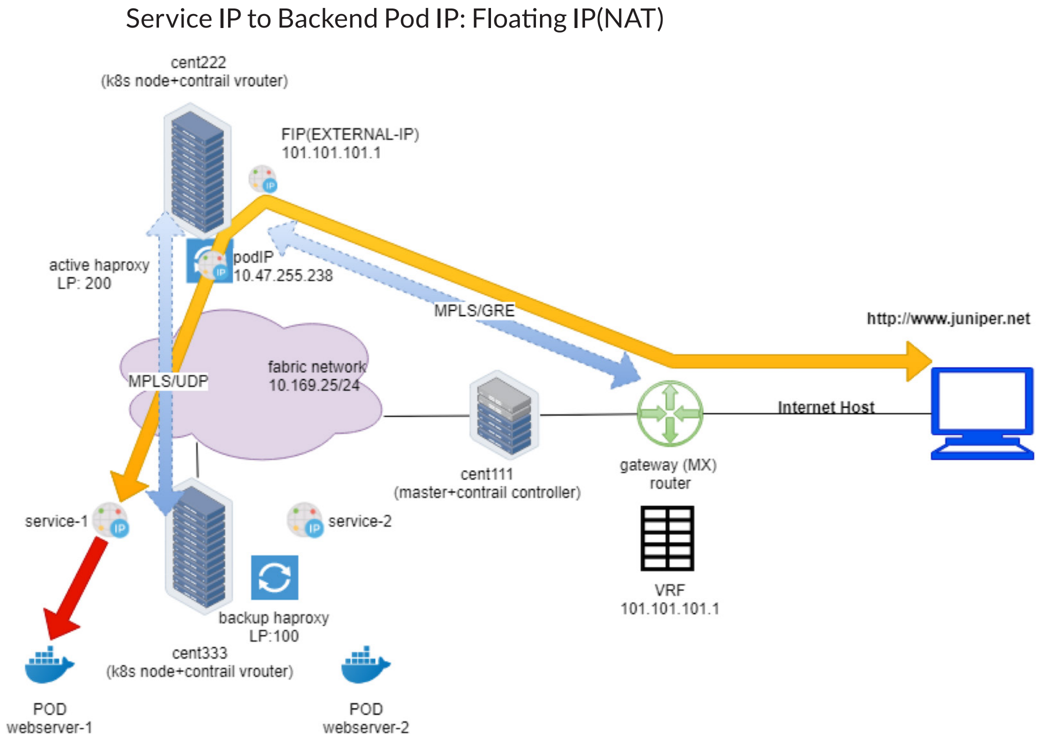


Figure 7.9 Service IP to Backend Pod IP

On destination node cent333, when the packet comes in from Ingress internal IP 10.47.255.238 toward the service IP 10.99.225.17 of webservice-1, the vRouter again does the NAT translation operations. It translates the service IP to the backend podIP 10.47.255.236, pretty much the same way as what you've seen in node cent222, where the vRouter translates between the ingress public floating IP with the ingress internal podIP.

Here is the flow table captured with the shell script. This flow shows the state of the second TCP connection between active haproxy and the backend pod:

```
evrouter-agent)[root@cent333 /]$ flow --match 10.47.255.238
Flow table(size 80609280, entries 629760)
Entries: Created 482 Added 482 Deleted 10 Changed 10Processed 482 Used Overflow entries 0
(Created Flows/CPU: 163 146 18 155)(oflows 0)

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

```
Listing flows matching ([10.47.255.238]:*)

    Index                Source:Port/Destination:Port                      Proto(V)
 ----------------------------------------------------------------------------------
   403188<=>462132       10.47.255.236:80                                    6 (2->4)
                         10.47.255.238:54500
(Gen: 1, K(nh):23, Action:N(SPs), Flags:, TCP:SSrEEr, QOS:-1, S(nh):23,
 Stats:2/140,  SPort 52190, TTL 0, Sinfo 4.0.0.0)

   462132<=>403188       10.47.255.238:54500                                 6 (2->2)
                         10.99.225.17:8888
(Gen: 1, K(nh):23, Action:N(DPd), Flags:, TCP:SSrEEr, QOS:-1, S(nh):26,
 Stats:3/271,  SPort 65421, TTL 0, Sinfo 10.169.25.20)
```

You've seen something similar in the service section, so you shouldn't have issues understanding it. Obviously the second entry is triggered by the incoming request from active haproxy IP (the Ingress podIP) towards the service IP. The vRouter knows the service IP is a floating IP that maps to the backend podIP 10.47.255.236, and service port maps to the container targetPort in the backend pod. It does DNAT+DPAT (DPd) in the incoming direction and SNAT+SPAT (SPs) in the outgoing direction.

The other easy way to trace this forwarding path is to look at the MPLS label. In previous step we've seen label 38 is used when the active haproxy computes cent222 sent packets into the MPLSoUDP tunnel to compute cent333. You can use the `vrouter mpls` utility to check the nexthop of this In-label:

```
$ docker exec -it vrouter_vrouter-agent_1 mpls --get 38
MPLS Input Label Map

   Label    NextHop
------------------
     38       26

$ docker exec -it vrouter_vrouter-agent_1 nh --get 26
Id:26       Type:Encap        Fmly: AF_INET  Rid:0  Ref_cnt:9        Vrf:2
           Flags:Valid, Policy, Etree Root,
           EncapFmly:0806 Oif:4 Len:14
           Encap Data: 02 bd e8 bc 46 9a 00 00 5e 00 01 00 08 00

$ vif --get 4
Vrouter Interface Table

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

vif0/4     OS: tapeth0-baa392
          Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:10.47.255.236
```

```
Vrf:2 Mcast Vrf:2 Flags:PL3DEr QOS:-1 Ref:6
RX packets:29389  bytes:1234338 errors:0
TX packets:42264  bytes:1775136 errors:0
Drops:29389
```

Once the next hop is determined, you can find the outgoing interface (Oif) number, then with `vif` utility you can locate the pod interface. The corresponding podIP 10.47.255.236 is the backend pod for the HTTP request, which looks consistent with what the flow table shows above.

Finally the pod sees the HTTP request and responds back with a web page. This returning traffic is reflected by the first flow entry in the capture, which shows:

- The original source IP is a backend podIP of 10.47.255.236
- The original source port is webserver port 80
- The destination IP is Ingress internal podIP 10.47.255.238

Backend Pod: Analyze HTTP Request

Another tcpdump packet capture on the backend pod interface helps to reveal the packet interaction between the ingress internal IP and the backend podIP:

```
$ tcpdump -ni tapeth0-baa392 -v
12:01:07.701956 IP (tos 0x0, ttl 63, id 32663, offset 0, flags [DF], proto TCP (6), length 60)
   10.47.255.238.54500 > 10.47.255.236.http: Flags [S], cksum 0xd88d (correct), seq 2129282145, win 
29200, options [mss 1420,sackOK,TS val 515783670 ecr 0,nop,wscale 7], length 0
12:01:07.702012 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 60)
   10.47.255.236.http > 10.47.255.238.54500: Flags [S.], cksum 0x1468 (incorrect -> 0x8050), seq 
3925744891, ack 2129282146, win 28960, options [mss 1460,sackOK,TS val 515781436 ecr 
515783670,nop,wscale 7], length 0
12:01:07.702300 IP (tos 0x0, ttl 63, id 32664, offset 0, flags [DF], proto TCP (6), length 52)
   10.47.255.238.54500 > 10.47.255.236.http: Flags [.], cksum 0x1f57 (correct), ack 1, win 229, 
options [nop,nop,TS val 515783671 ecr 515781436], length 0
12:01:07.702304 IP (tos 0x0, ttl 63, id 32665, offset 0, flags [DF], proto TCP (6), length 159)
   10.47.255.238.54500 > 10.47.255.236.http: Flags [P.], cksum 0x6fac (correct), seq 1:108, ack 1, win 
229, options [nop,nop,TS val 515783671 ecr 515781436], length 107: HTTP, length: 107
      GET / HTTP/1.1
      User-Agent: curl/7.29.0
      Accept: */*
      Host:www.juniper.net
      X-Forwarded-For: 15.15.15.2

12:01:07.702336 IP (tos 0x0, ttl 64, id 12224, offset 0, flags [DF], proto TCP (6), length 52)
   10.47.255.236.http > 10.47.255.238.54500: Flags [.], cksum 0x1460 (incorrect -> 0x1eee), ack 108, 
win 227, options [nop,nop,TS val 515781436 ecr 515783671], length 0
12:01:07.711882 IP (tos 0x0, ttl 64, id 12225, offset 0, flags [DF], proto TCP (6), length 69)
   10.47.255.236.http > 10.47.255.238.54500: Flags [P.], cksum 0x1471 (incorrect -> 0x5f06), seq 1:18, 
ack 108, win 227, options [nop,nop,TS val 515781446 ecr 515783671], length 17: HTTP, length: 17
      HTTP/1.0 200 OK
12:01:07.712032 IP (tos 0x0, ttl 64, id 12226, offset 0, flags [DF], proto TCP (6), length 550)
   10.47.255.236.http > 10.47.255.238.54500: Flags [FP.], cksum 0x1652 (incorrect -> 0x1964), seq 
18:516, ack 108, win 227, options [nop,nop,TS val 515781446 ecr 515783671], length 498: HTTP
12:01:07.712152 IP (tos 0x0, ttl 63, id 32666, offset 0, flags [DF], proto TCP (6), length 52)
```

```
   10.47.255.238.54500 > 10.47.255.236.http: Flags [.], cksum 0x1ec7 (correct), ack 18, win 229,
options [nop,nop,TS val 515783681 ecr 515781446], length 0
12:01:07.712192 IP (tos 0x0, ttl 63, id 32667, offset 0, flags [DF], proto TCP (6), length 52)
   10.47.255.238.54500 > 10.47.255.236.http: Flags [F.], cksum 0x1ccb (correct), seq 108, ack 517, win
237, options [nop,nop,TS val 515783681 ecr 515781446], length 0
12:01:07.712202 IP (tos 0x0, ttl 64, id 12227, offset 0, flags [DF], proto TCP (6), length 52)
   10.47.255.236.http > 10.47.255.238.54500: Flags [.], cksum 0x1460 (incorrect -> 0x1cd5), ack 109,
win 227, options [nop,nop,TS val 515781446 ecr 515783681], length 0
```

Return Traffic

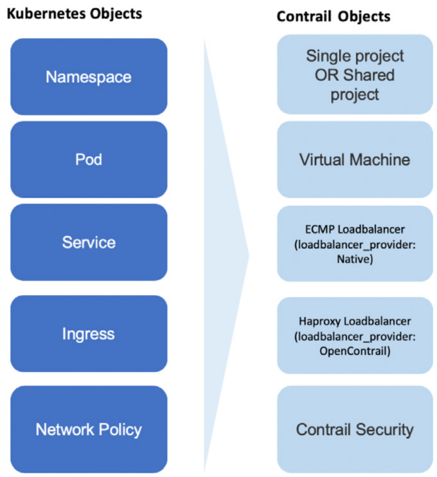
On the reverse direction, podIP runs webserver and responds with its web page. The response follows the reverse path of the request:

- The pod responds to load balancer frontend IP, across MPLSoUDP tunnel.
- The vRouter on node cent333 performs SNAT+SPAT, translating podIP:podPort into serviceIP:servicePort.
- The respond reaches to the active haproxy running on node cent222.
- The haproxy copies the HTTP response from the backend pod, and pastes into its connection with the remote Internet host.
- The vRouter on node cent222 performs SNAT, translating load balancer frontend IP to floating IP.
- The response is sent to the gateway router, which forwards it to the Internet host.
- The Internet host gets the response.

Chapter 8

Contrail Firewall Policy

In Chapter 4, you were given the Kubernetes to Contrail Object Mapping Figure, which is repeated here as Figure 8.1.



Kubernetes Objects
Contrail Objects
Namespace
Single project OR Shared project
Pod
Virtual Machine
Service
ECMP Loadbalancer (loadbalancer_provider: Native)
Ingress
Haproxy Loadbalancer (loadbalancer_provider: OpenContrail)
Network Policy
Contrail Security

Figure 8.1 Contrail Kubernetes Object Mapping

This mapping highlights Contrail's implementation of Kubernetes core objects: Namespace, pod, Service, Ingress, and Network Policy. From Chapters 4 through 7 we've pretty much explored everything in Figure 8.1 except Network Policy.

In this chapter we'll focus on the Network Policy implementation in Contrail. First we'll introduce the Contrail firewall, which is the feature used to implement Kubernetes network policy; then we'll set up a test case to verify how Kubernetes network policy works in Contrail; then, based on the test results, we'll explore the Contrail firewall policies and their rules in detail in order to understand the Contrail implementation, as well as the mapping between the two objects in the object mapping diagram of Figure 8.1.

Introducing Contrail Firewall

In Chapter 3 we introduced the Kubernetes network policy concept. We went through the YAML file definition in detail and created a network policy based on it. We've also mentioned that simply creating network policy objects won't have any effect, unless the Kubernetes networking implementation supports it. Contrail, as a Kubernetes CNI, implements the Kubernetes networking and supports the Kubernetes network policy through Contrail firewall. That is the focus of this chapter - we'll demonstrate how network policy works in the Contrail environment via Contrail firewall.

First let's review some important concepts in Contrail.

Inter-VN Routing.

In Contrail, virtual networks are isolated by default. That means workloads in `VN1` cannot communicate with workloads in another `VN2`. To allow inter-virtual network communications between `VN1` to `VN2`, Contrail network policy is required. Contrail network policy can also provide security between two virtual networks by allowing or denying specified traffic.

Contrail Network Policy.

A Contrail network policy is used to permit inter-virtual network communication and to modify intra-virtual network traffic. It describes which traffic is permitted or not between virtual networks. By default, without a Contrail network policy, intra-virtual network communication is allowed, but inter-virtual network traffic is denied. When you create a network policy you must associate it with a virtual network for it to have any effect.

NOTE Don't confuse Contrail network policy with Kubernetes network policy. They are two different security features and they work separately.

Security Group(SG).

A security group, often abbreviated as a `SG`, is a group of rules that allow a user to specify the type of traffic that is allowed or not allowed through a port. When a VM or pod is created in a virtual network, a SG can be associated with the VM when it is launched. Unlike Contrail network policy, which is configured *globally* and associated to the virtual networks, the SG is configured on the per-port basis and it will take effect on the specific vRouter flows that is associated with the VM port.

The Limitation of Contrail Network Policy and SG

In modern Contrail cloud environments, sometimes it is hard to use only the existing network policy and security group to achieve desired security goals. For example, in cloud environments, workloads may move from one server to another and so most likely the IP is often changing. Just relying on IP addresses to identify the endpoints to be protected is painful. Instead, users must leverage application level attributes to manipulate policies, so that the policies don't need to be updated every time a workload moves and the associated network environment changes. Also, in production, a user might need to group workloads based on combinations of tags, which is hard to translate into the existing language of a network policy or SG.

Contrail Firewall Security Policy.

This chapter introduces another important feature: Contrail firewall security policy.

Contrail firewall security policy allows decoupling of routing from security policies,vand provides multi-dimension segmentation and policy portability, while significantly enhancing user visibility and analytics functions.

In order to implement the multi-dimension traffic segmentation, Contrail firewall introduces the concept of *tags*. Tags are key-value pairs associated with different entities in the deployment. Tags can be pre-defined or custom/user defined. Contrail `tags` are pretty much the same thing as Kubernetes `labels`. Both are used to identify objects and workloads. As you can see, this is similar to Kubernetes network policy design, and it is natural for Contrail to use its firewall security policy to implement Kubernetes network policy. In theory, Contrail network policy or SG can be extended to do the job, but the support of tags by Contrail firewall makes it so much simpler.

NOTE Sometimes Contrail firewall security policy is referred to as Contrail Security, Contrail firewall, Contrail firewall security, or simply Contrail FW.

Contrail Kubernetes Network Policy Use Case

In this section, we'll create a use case to verify how Kubernetes network policy works in Contrail environments. We'll start by creating a few Kubernetes namespaces and pods that are required in the test. We'll confirm that every pod can talk to the DUT (Device Under Test) because of the default *allow-any-any* networking model, then create network policies and observe any changes with same traffic pattern.

Network Design

The use case design is shown in Figure 8.2.

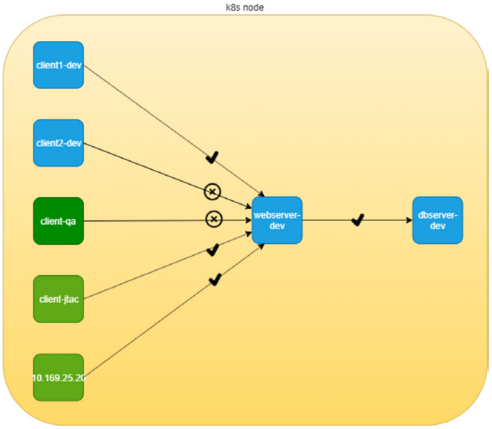


Figure 8.2 Network Policy Test Case Design

In Figure 8.2 six nodes are distributed in three departments: `dev`, `qa`, and `jtac`. The `dev` department is running a database server (`dbserver-dev`) holding all valuable data collected from the customer. The design requires that no one have direct access to this db server, instead, the db server access is only allowed through another Apache frontend server in `dev` department, named `webserver-dev`. Furthermore, for security reasons, the access of customer information should only be granted to authorized clients. For example, only nodes in the `jtac` department, one node in `dev` department named `client1-dev`, and the source IP `10.169.25.20` can access the db via webserver. And finally, the database server `dbserver-dev` should not initiate any connection toward other nodes.

Lab Preparation

Here is a very ordinary, simplified network design that you can see anywhere. If we model all these network elements in the Kubernetes world, it looks like Figure 8.3.

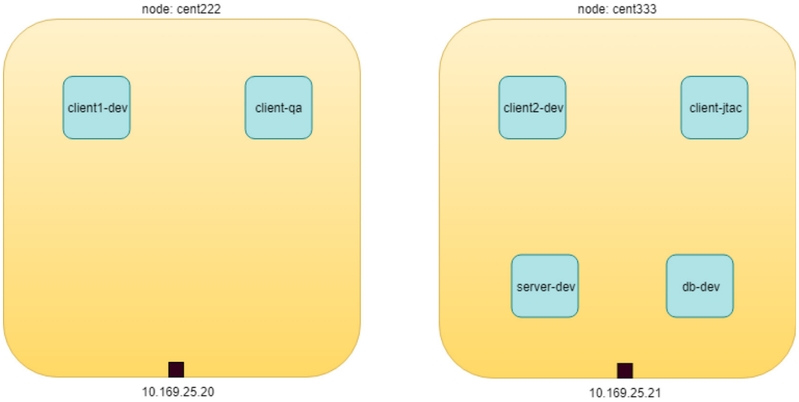


Figure 8.3 Network Policy: NS and Pods

We need to create the following resources:

- Three namespaces: `dev`, `qa`, `jtac`
- Six pods:
  - Two server pods: `webserver-dev`, `dbserver-dev`
  - Two client pods in the same namespace as server pods: `client1-dev`, `client2-dev`
  - Two client pods from two different namespaces: `client-qa`, `client-jtac`
- Two CIDRs:
  - cidr: 10.169.25.20/32, this is fabric IP of node `cent222`
  - cidr: 10.169.25.21/32, this is fabric IP of node `cent333`

Table 8.1 Kubernetes Network Policy Test Environment

NS	pod	role
dev	client1-dev	web client
dev	client2-dev	web client
qa	client-qa	web client
jtac	client-jtac	web client
dev	webserver-dev	webserver serving clients
dev	dbserver-dev	dbserver serving webserver

Okay, let's prepare the required k8s namespace and pods resources with an all-in-one YAML file defining `dev`, `qa`, and `jtac` namespaces:

```
#policy-ns-pod.yaml
##################
# all namespaces #
##################
#policy-ns.yaml
kind: Namespace
apiVersion: v1
metadata:
  name: dev
  labels:
    project: dev
---
kind: Namespace
apiVersion: v1
metadata:
  name: qa
  labels:
    project: qa
---
kind: Namespace
apiVersion: v1
metadata:
  name: jtac
  labels:
    project: jtac
---
##################
#   all pods     #
##################
# policy-pod-do.yaml
apiVersion: v1
kind: Pod
metadata:
  name: webserver-dev
  labels:
    app: webserver-dev
    do: policy
  namespace: dev
spec:
  containers:
    - name: ubuntu
      image: contrailk8sdayone/contrail-webserver
---
apiVersion: v1
kind: Pod
metadata:
  name: dbserver-dev
  labels:
    app: dbserver-dev
    do: policy
  namespace: dev
spec:
  containers:
    - name: ubuntu
      image: contrailk8sdayone/contrail-webserver
```

```
---
apiVersion: v1
kind: Pod
metadata:
  name: client1-dev
  labels:
    app: client1-dev
    do: policy
  namespace: dev
spec:
  containers:
    - name: ubuntu
      image: contrailk8sdayone/contrail-webserver
---
apiVersion: v1
kind: Pod
metadata:
  name: client2-dev
  labels:
    app: client2-dev
    do: policy
  namespace: dev
spec:
  containers:
    - name: ubuntu
      image: contrailk8sdayone/contrail-webserver
---
apiVersion: v1
kind: Pod
metadata:
  name: client-qa
  labels:
    app: client-qa
    do: policy
  namespace: qa
spec:
  containers:
    - name: ubuntu
      image: contrailk8sdayone/contrail-webserver
---
apiVersion: v1
kind: Pod
metadata:
  name: client-jtac
  labels:
    app: client-jtac
    do: policy
  namespace: jtac
spec:
  containers:
    - name: ubuntu
      image: contrailk8sdayone/contrail-webserver
```

TIP Ideally, each pod should run with different images. And, TCP ports usually are different between a webserver and a database server. In our case, to make the test easier, we used the exact same `contrail-webserver` image that we've been using throughout the book for all the pods, so clients to webserver and webserver to database server communication all use the same port number 80 served by the same HTTP server. Also, we added a label `do: policy` in all pods, so that displaying all pods used in this test is also easier.

Okay and now create all the resources:

```
$kubectl create -f policy-ns-pod-do.yaml
namespace/dev created
namespace/qa created
namespace/jtac created
pod/webserver-dev created
pod/dbserver-dev created
pod/client1-dev created
pod/client2-dev created
pod/client-qa created
pod/client-jtac created

$ kubectl get pod --all-namespaces -l "do=policy" -o wide
NAMESPACE   NAME            READY   STATUS    RESTARTS   AGE   IP              NODE
dev         client1-dev     1/1     Running   0          33s   10.47.255.232   cent222
dev         client2-dev     1/1     Running   0          33s   10.47.255.231   cent333
dev         dbserver-dev    1/1     Running   0          33s   10.47.255.233   cent333
dev         webserver-dev   1/1     Running   0          33s   10.47.255.234   cent333
jtac        client-jtac     1/1     Running   0          33s   10.47.255.229   cent222
qa          client-qa       1/1     Running   0          33s   10.47.255.230   cent333
```

Traffic Mode Before Kubernetes Network Policy Creation

Since we have all of the namespace and pods, before we define any network policy, let's go ahead and send traffic between clients and servers.

Of course, Kubernetes networking, by default, follows the allow-any-any model, so we should expect that access works between any pod, which is going to be a fully meshed access relationship. But keep in mind that the `DUT` in this test is `webserver-dev` and `dbserver-dev` is the one we are more interested in observing. To simplify the verification, according to our diagram, we'll focus on accessing the server pods from the client pods, as illustrated in Figure 8.4.

The highlights of Figure 8.4 are that all clients can access the servers, following the permit-any-any model:

- there are no restrictions between clients and `webserver-dev` pod
- there are no restrictions between clients and `dbserver-dev` pod

And the communication between client and servers are bi-directional and symmetrical – each end can initiate a session or accept a session. These map to the egress policy and ingress policy, respectively, in Kubernetes.

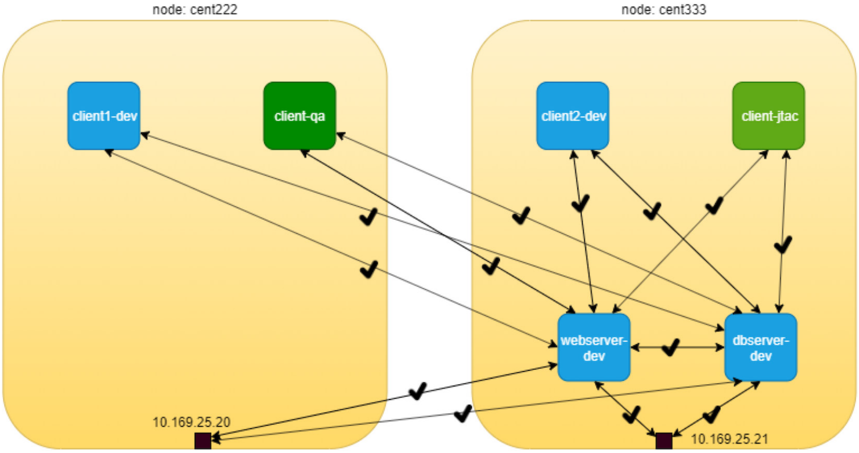


Figure 8.4 Network Policy: Pods Communication Before Network Policy Creation

Obviously, these do not meet our design goal, which is exactly why we need the Kubernetes network policy, and we'll come to that part soon. For now, let's quickly verify the allow-any-any networking model.

First let's verify the HTTP server running at port 80 in `webserver-dev` and `dbserver-dev` pods:

```
$kubectl exec -it webserver-dev -n dev -- netstat -antp| grep 80
tcp        0      0 0.0.0.0:80              0.0.0.0:*     LISTEN      1/python
$kubectl exec -it dbserver-dev -n dev -- netstat -antp| grep 80
tcp        0      0 0.0.0.0:80              0.0.0.0:*     LISTEN      1/python
```

NOTE As mentioned earlier, in this test all pods are with the same container image, so all pods are running the same webserver application in their containers. We simply name each pod to reflect their different roles in the diagram.

Now we can verify accessing this HTTP server from other pods with the following commands.

To test ingress traffic:

```
#from master
dbserverIP=10.47.255.233
webserverIP=10.47.255.234
kubectl exec -it client1-dev -n dev -- curl http://$webserverIP -m5
kubectl exec -it client2-dev -n dev -- curl http://$webserverIP -m5
kubectl exec -it client-qa -n qa -- curl http://$webserverIP -m5
kubectl exec -it client-jtac -n jtac -- curl http://$webserverIP -m5
kubectl exec -it dbserver-dev -n dev -- curl http://$webserverIP -m5

#from node cent222 (fabric interface IP: 10.169.25.20)
curl http://$webserverIP -m5
#from node cent333 (fabric interface IP: 10.169.25.21)
curl http://$webserverIP -m5
```

These commands trigger the HTTP requests to the `webserver-dev` pod from all clients and hosts of the two nodes. The `-m5` `curl` command option makes `curl` wait a maximum of five seconds for the response before it claims time out. As expected, all accesses pass through and return the same output shown next.

From `client1-dev`:

```
$ kubectl exec -it client1-dev -n dev -- \
    curl http://$webserverIP | w3m -T text/html | grep -v "^$"
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.234
                             Hostname = webserver-dev
```

Here, `w3m` gets the output from `curl`, which returns a webpage HTML code and renders it into readable text, then send it to `grep` to remove the empty lines. To make the command shorter you can define an alias:

```
alias webpr='w3m -T text/html | grep -v "^$"'
```

Now the command looks shorter:

```
$ kubectl exec -it client1-dev -n dev -- curl http://$webserverIP | webpr
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.234
                           Hostname = webserver-dev
```

Similarly, you'll get the same test results for access to `dbserver-dev` from any of the other pods.

Create Kubernetes Network Policy

Now let's create the k8s network policy to implement our design. From our initial design goal, these are what we wanted to achieve via network policy:

- `client1-dev` and pods under `jtac` namespace (that is `jtac-dev` pod) can access `webserver-dev` pod
- `webserver-dev` pod (and only it) is allowed to access `dbserver-dev` pod
- all other client pods are not allowed to access the two server pods
- all other client pods can still communicate with each other

Translating these requirements into the language of Kubernetes network policy, we'll work with this network policy YAML file:

```
#policy1-do.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: policy1
  namespace: dev
spec:
  podSelector:
    matchLabels:
      app: webserver-dev
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 10.169.25.20/32
    - namespaceSelector:
        matchLabels:
          project: jtac
    - podSelector:
        matchLabels:
          app: client1-dev
    ports:
    - protocol: TCP
      port: 80
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: dbserver-dev
    ports:
    - protocol: TCP
      port: 80
```

From the network-policy definition, based on what you've learned in Chapter 3, you should easily be able to tell what the policy is trying to enforce in our current setup:

- According to the ingress policy, the following clients can reach the `webserver-dev` server pod located in `dev` namespace:
  - `client1-dev` from `dev` namespace
  - all pods from `jtac` namespace, that is `client-jtac` pod in our setup
  - clients with source IP `10.169.25.20` (`cent222` in our setup)
- According to the egress policy, the `webserver-dev` server pod in `dev` namespace can initiate a TCP session toward `dbserver-dev` pod with destination port 80 to access the data.
- For target pod `server-dev`, all other accesses are denied.
- Communication between all other pods are not affected by this network policy.

TIP Actually, this is the exact network policy YAML file that we've demonstrated in Chapter 3.

Let's create the policy and verify its effect:

```
$ kubectl apply -f policy1-do.yaml
networkpolicy.networking.k8s.io/policy1 created

$ kubectl get networkpolicies --all-namespaces
NAMESPACE   NAME       POD-SELECTOR        AGE
dev         policy1    app=webserver-dev   17s
```

Post Kubernetes Network Policy Creation

After the network policy `policy1` is created, let's test the accessing of the HTTP server in `webserver-dev` pod from pod `client1-dev`, `client-jtac` , and node `cent222` host:

```
$ kubectl exec -it client1-dev -n dev -- curl http://$webserverIP | webpr
                                     Hello
                    This page is served by a Contrail pod
                         IP address = 10.47.255.234
                          Hostname = webserver-dev
```

The access from these two pods to `webserver-dev` is okay and that is what we want. Now, if we repeat the same test from the other pod `client2-dev`, `client-qa` and another node `cent333` now get timed out:

```
$ kubectl exec -it client2-dev -n dev -- curl http://$webserverIP -m 5
curl: (28) Connection timed out after 5000 milliseconds
command terminated with exit code 28

$ kubectl exec -it client-jtac -n jtac -- curl http://$webserverIP -m 5
curl: (28) Connection timed out after 5000 milliseconds
command terminated with exit code 28

$ curl http://$webserverIP -m 5
curl: (28) Connection timed out after 5000 milliseconds
```

The new test results after the network policy is applied are illustrated in Figure 8.5.

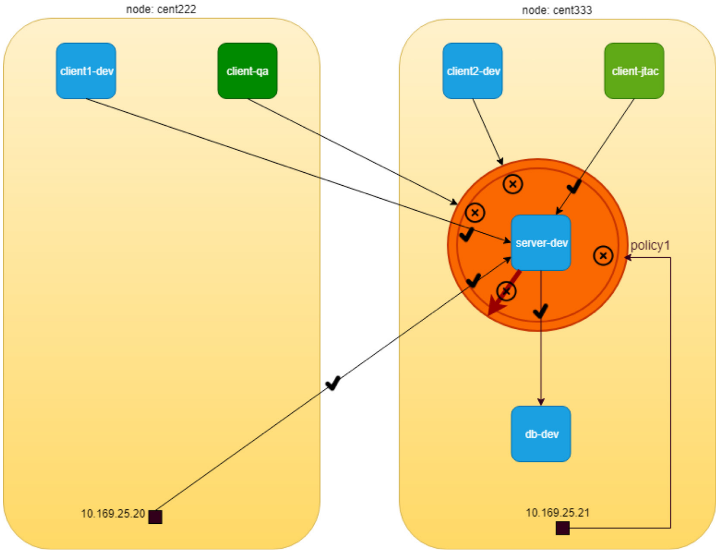


Figure 8.5 *Network Policy: After Applying Policy1*

A detail of the network policy object tells the same things:

```
$ kubectl describe netpol -n dev policy1
Name:         policy1
Namespace:    dev
Created on:   2019-09-29 21:21:14 -0400 EDT
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
            {"apiVersion":"networking.k8s.io/v1","kind":"NetworkPolicy",
            "metadata":{"annotations":{},"name":"policy1","namespace":"dev"},
            "spec":{"egre...
Spec:
  PodSelector:     app=webserver-dev
  Allowing ingress traffic:     #<---
    To Port: 80/TCP
    From:
      IPBlock:
        CIDR: 10.169.25.20/32
        Except:
    From:
      NamespaceSelector: project=jtac
    From:
      PodSelector: app=client1-dev
  Allowing egress traffic:
    To Port: 80/TCP
    To:
      PodSelector: app=dbserver-dev
  Policy Types: Ingress, Egress
```

From the above exercise, we can conclude that k8s network policy works as expected in Contrail.

But our test is not done yet. In the network policy we defined both ingress and egress policy, but so far from `webserver-dev` pod's perspective we've only tested that the ingress policy of `policy1` works successfully. Additionally, we have not applied any policy to the other server pod `dbserver-dev`. According to the default allow any policy, any pods can directly access it without a problem. Obviously, this is not what we wanted according to our original design. Another ingress network policy is needed for `dbserver-dev` pod, and finally, we need to apply an egress policy to `dbserver-dev` to make sure it can't connect to any other pods. So there are at least three more test items we need to confirm, namely:

- Test the egress policy of `policy1` applied to `webserver-dev` pod;
- Define and test ingress policy for `dbserver-dev` pod;
- Define and test egress policy for `dbserver-dev` pod.

Let's look at the egress policy of `policy1` first.

Egress Policy on `webserver-dev` Pod

Here's the test on egress traffic:

```
#test access to all pods
kubectl exec -it webserver-dev -n dev -- curl http://$dbserverIP -m5
kubectl exec -it webserver-dev -n dev -- curl http://<other pod IPs> -m5

#test access to all ipBlock
kubectl exec -it webserver-dev -n dev -- curl http://10.169.25.20 -m5
kubectl exec -it webserver-dev -n dev -- curl http://10.169.25.21 -m5
```

The result shows that only access to `dbserver-dev` succeeds while other egress access times out:

```
$ kubectl exec -it webserver-dev -n dev -- curl $dbserverIP -m5 | webpr
                                Hello
                    This page is served by a Contrail pod
                         IP address = 10.47.255.233
                           Hostname = dbserver-dev
$ kubectl exec -it webserver-dev -n dev -- curl 10.47.255.232 -m5
curl: (28) Connection timed out after 5001 milliseconds
command terminated with exit code 28
```

Network Policy on `dbserver-dev` Pod

So far, so good. Let's look at the second test items, ingress access to `dbserver-dev` pod from other pods other than `webserver-dev` pod. Test the egress traffic:

```
#test access to all pods
kubectl exec -it webserver-dev -n dev -- curl http://$dbserverIP -m5
kubectl exec -it client1-dev -n dev -- curl http://$dbserverIP -m5
kubectl exec -it client2-dev -n dev -- curl http://$dbserverIP -m5
kubectl exec -it client-jtac -n dev -- curl http://$dbserverIP -m5
kubectl exec -it client-qa -n dev -- curl http://$dbserverIP -m5

#test access to all ipBlock
#from node cent222 (fabric interface IP: 10.169.25.20)
curl http://10.47.255.233 -m5
#from node cent333 (fabric interface IP: 10.169.25.21)
curl http://10.47.255.233 -m5
```

All pods can access `dbserver-dev` pod directly:

```
$ kubectl exec -it client1-dev -n dev -- curl http://$dbserverIP -m5 | webpr
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.233
                            Hostname = dbserver-dev
```

Our design is to block access from all pods except the `webserver-dev` pod. For that we need to apply another policy. Here is the YAML file of the second policy:

```
#policy-do2.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: policy2
  namespace: dev
spec:
  podSelector:
    matchLabels:
      app: dbserver-dev
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: webserver-dev
    ports:
    - protocol: TCP
      port: 80
```

This network policy, `policy2`, is pretty much like the previous `policy1`, except that it looks simpler – the `policyTypes` only has `Ingress` in the list so it will only define an ingress policy. And that ingress policy defines a whitelist using only a `podSelector`. In our test case, only one pod `webserver-dev` has the matching label with it so it will be the only one allowed to initiate the TCP connection toward target pod `dbserver-dev` on port 80. Let's create the policy `policy2` now and verify the result again:

```
$ kubectl exec -it webserver-dev -n dev -- curl http://$dbserverIP -m5 | webpr
                                     Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.233
                            Hostname = dbserver-dev

$ kubectl exec -it client1-dev -n dev -- curl http://$dbserverIP -m5 | webpr
command terminated with exit code 28
curl: (28) Connection timed out after 5002 milliseconds
```

Now the access to `dbserver-dev` pod is secured!

Egress Policy on `dbserver-dev`

Okay, just one last requirement from our design goal: server `dbserver-dev` should not be able to initiate any connection toward other nodes.

When you reviewed `policy2`, you may have wondered how we make that happen. In Chapter 3 we emphasized that network policy is whitelist-based only by design. So whatever you put in the whitelist means it is *allowed*. Only a blacklist gives a deny, but even with a blacklist you won't be able to list all the other pods just to get them denied.

Another way of thinking about this is to make use of the deny all implicit policy. So assuming this sequence of policies is in current Kubernetes network policy design:

- policy2 on `dbserver-dev`
- deny all for `dbserver-dev`
- allow all for other pods

It looks like if we give an empty whitelist in egress policy of dbserver-dev, then nothing will be allowed and the deny all policy for the target pod will come into play. The problem is, how do we define an empty whitelist:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: policy2-tryout
  namespace: dev
spec:
  podSelector:
    matchLabels:
      app: dbserver-dev
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: webserver-dev
    ports:
    - protocol: TCP
      port: 80
  engress:      #<---
```

Turns out this doesn't work as expected:

```
$ kubectl exec -it dbserver-dev -n dev -- curl http://10.47.255.232 -m5 | webpr
                                 Hello
                     This page is served by a Contrail pod
                          IP address = 10.47.255.232
                            Hostname = client1-dev
```

Checking the policy object detail does not uncover anything obviously wrong:

```
$ kubectl describe netpol policy2-tryout -n dev
Name:         policy2-tryout
Namespace:    dev
Created on:   2019-10-01 17:02:18 -0400 EDT
Labels:       <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
                {"apiVersion":"networking.k8s.io/v1","kind":"NetworkPolicy",
                "metadata":{"annotations":{},"name":"policy2-tryout",
                "namespace":"dev"},"spec"...
Spec:
  PodSelector:     app=dbserver-dev
  Allowing ingress traffic:
    To Port: 80/TCP
    From:
      PodSelector: app=webserver-dev
  Allowing egress traffic:
    <none> (Selected pods are isolated for egress connectivity)         #<---
  Policy Types: Ingress
```

The problem is on the `policyTypes`. We haven't added the `Egress` in, so whatever is configured in egress policy will be ignored. Simply adding `- Egress` in `policyTypes` will fix it. Furthermore, to express an empty whitelist, the `egress:` keyword is optional and not required. Below is the new policy YAML file:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: policy2-egress-denyall
  namespace: dev
spec:
  podSelector:
    matchLabels:
      app: dbserver-dev
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: webserver-dev
    ports:
    - protocol: TCP
      port: 80
```

Now delete the old `policy2` and apply this new policy. Requests from `dbserver-dev` to any other pods (for example pod `client1-dev`) will be blocked:

```
$ kubectl exec -it dbserver-dev -n dev -- curl http://10.47.255.232 | webpr
command terminated with exit code 28
curl: (7) Failed to connect to 10.47.255.232 port 80: Connection timed out
```

And here is the final diagram illustrating our network policy test result in Figure 8.6.

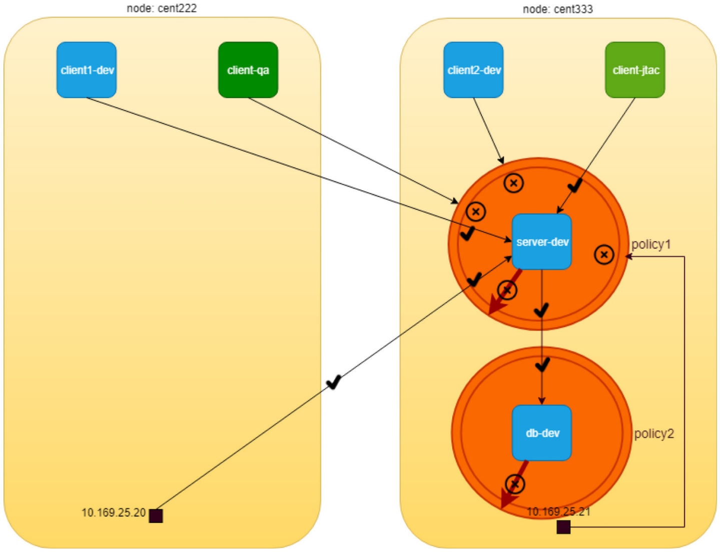


Figure 8.6 Network Policy: After Applying an Empty Egress Policy on `observer-dev` *Pod*

The Drop Action in Flow Table

Before concluding the test, let's take a look at the vRouter flow table when traffic is dropped by the policy. On node `cent333` where pod `dbserver-dev` is located:

```
$ docker exec -it vrouter_vrouter-agent_1 flow --match 10.47.255.232:80
Flow table(size 80609280, entries 629760)

Entries: Created 33 Added 33 Deleted 30 Changed 54Processed 33 Used Overflow entries 0
(Created Flows/CPU: 7 9 11 6)(oflows 0)

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

Listing flows matching ([10.47.255.232]:80)

    Index                Source:Port/Destination:Port                      Proto(V)
 -----------------------------------------------------------------------------------
   158672<=>495824       10.47.255.232:80                                    6 (5)
                         10.47.255.233:42282
(Gen: 1, K(nh):59, Action:D(Unknown), Flags:, TCP:Sr, QOS:-1, S(nh):63,
 Stats:0/0,  SPort 54194, TTL 0, Sinfo 0.0.0.0)

   495824<=>158672       10.47.255.233:42282                                 6 (5)
                         10.47.255.232:80
(Gen: 1, K(nh):59, Action:D(FwPolicy), Flags:, TCP:S, QOS:-1, S(nh):59,
 Stats:3/222,  SPort 52162, TTL 0, Sinfo 8.0.0.0)
```

The Action:D is set to D(FwPolicy), which means Drop due to the firewall policy. Meanwhile, in the other node cent222, where the pod client1-dev is located, we don't see any flow generated, indicating the packet does not arrive:

```
$ docker exec -it vrouter_vrouter-agent_1 flow --match 10.47.255.233
Flow table(size 80609280, entries 629760)

......

Listing flows matching ([10.47.255.233]:*)

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

Contrail Implementation Details

We've reiterated that Contrail implements Kubernetes network policy with a Contrail firewall security policy. You also know that Kubernetes labels are exposed as tags in Contrail. These tags are used by Contrail security policy to implement specified Kubernetes policies. Tags will be created automatically from Kubernetes objects labels or created manually in the UI.

In this section we'll take a closer look at the Contrail firewall policies, policy rules, and the tags. In particular, we'll examine the mapping relationships between the Kubernetes objects that we created and tested in the last section, and the corresponding Contrail objects in Contrail firewall system.

Contrail firewall is designed with a hierarchical structure:

- The top level object is named Application Policy Set, abbreviated as APS.
- APS has firewall policies.
- Firewall policy has firewall rules.
- Firewall rules have the endpoints.
- Endpoints can be identified via tags or address groups (CIDRs).

The structure is illustrated in Figure 8.7.

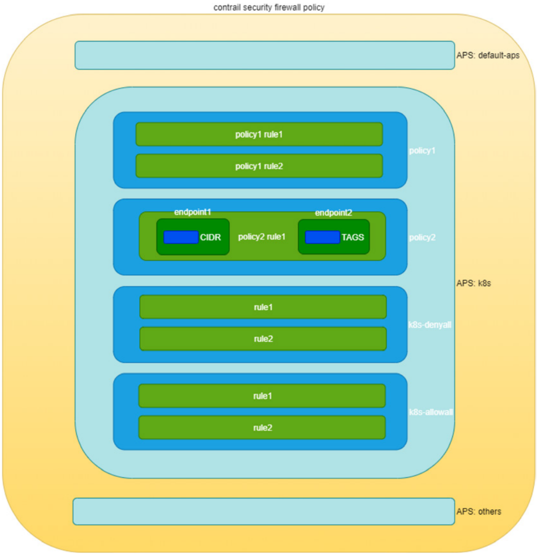


Figure 8.7 Contrail Firewall

Construct Mappings

Kubernetes network policy and Contrail firewall policy are two different entities in terms of the semantics of the network policy in which each is specified. In order for Contrail firewall to implement Kubernetes network policy, Contrail needs to implement the one-to-one mapping for a lot of data construct from Kubernetes to Contrail firewall. These data constructs are the basic building blocks of Kubernetes network policy and the corresponding Contrail firewall policy.

Table 8.1 lists Kubernetes network policy constructs and the corresponding constructs in Contrail:

Table 8.1 K8s Network Policy And Contrail Firewall Construct Mapping

K8s Network Policy Construct	Contrail Firewall Construct
Cluster Name	APS (one per k8s cluster)
Network Policy	Firewall Policy (one per k8s network policy)
Ingress and Egress policy rule	Firewall Rule (one per k8s ingress/egress policy rule)
CIDR	Address Group(one per k8s network policy CIDR)
Label	Tag (one for each k8s label)
Namespace	Custom Tag (one for each namespace)

The `contrail-kube-manager`, the `KM`, as we've read many times earlier in this book, does all of the translations between the two worlds. Basically the following will happen in the context of Kubernetes network policy:

1. The `KM` will create an APS with a Kubernetes cluster name during its initialing process. Typically the default Kubernetes cluster name is `k8s`, so you will see an APS with the same name in your cluster.
2. The `KM` registers to `kube-apiserver` to watch the network policies events.
3. Whenever a Kubernetes network policy is created, a corresponding Contrail firewall policy will be created with all matching firewall rules and network endpoints.
4. For each `label` created in a Kubernetes object there will be a corresponding Contrail `tag` created.
5. Based on the `tag`, the corresponding Contrail objects (VN, pods, VMI, projects, etc.) can be located.
6. Contrail will then apply the Contrail firewall policies and rules in the APS on the Contrail objects, this is how the specific traffic is permitted or denied.

The APS can be associated to different Contrail objects, for example:

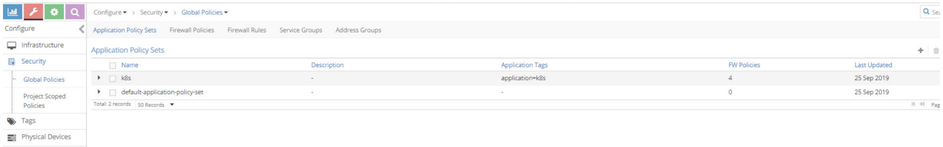
- VMI (virtual machine interface)
- VM (virtual machine) or pods
- VN (virtual network)
- project

In Contrail Kubernetes cluster, the APS is associated to virtual network. Whenever traffic goes on those networks, firewall policies associated on the APS would be evaluated and respective action would be taken for the traffic.

In the previous section, we created two Kubernetes network policies in our use case. Now let's explore the Contrail objects that are created for these Kubernetes network policies.

Application Policy Set (APS)

As mentioned, `contrail-kube-manager` will create an APS using the Kubernetes cluster name during the initialization stage. In Chapter 3, when we introduced *Contrail Namespaces and Isolation*, we learned the cluster name is `k8s` by default in Contrail. Therefore the APS name will also be `k8s` in the Contrail UI shown in Figure 8.8.



Name	Description	Application Tags	FW Policies	Last Updated
k8s	-	application=k8s	4	25 Sep 2019
default-application-policy-set	-	-	0	25 Sep 2019

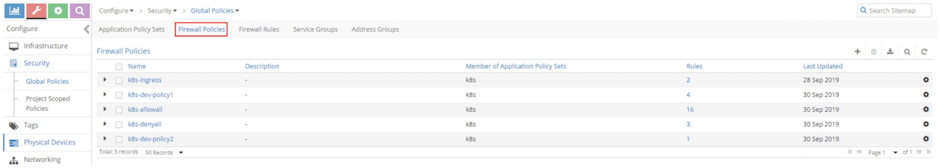
Figure 8.8 Contrail UI: APS Configure> Security > Global Policies > Application Policy Sets

There is one more APS `default-application-policy-set` that is created by default.

Policies

Now click on Firewall Policies to display all firewall polices in the cluster. In our test environment, you will find the following policies available:

- k8s-dev-policy1
- k8s-dev-policy2
- k8s-denyall
- k8s-allowall
- k8s-Ingress



Name	Description	Member of Application Policy Sets	Rules	Last Updated
k8s-Ingress	-	k8s	2	28 Sep 2019
k8s-dev-policy1	-	k8s	4	30 Sep 2019
k8s-allowall	-	k8s	16	30 Sep 2019
k8s-denyall	-	k8s	3	30 Sep 2019
k8s-dev-policy2	-	k8s	1	30 Sep 2019

Figure 8.9 Contrail UI: Firewall Policies

Contrail Firewall Policy Naming Convention

The `k8s-dev-policy1` and `k8s-dev-policy2` policies are what we've created. Although they look different from the object names we gave in our YAML file, it is easy to tell which is which. When `KM` creates the Contrail firewall policies based on the Kubernetes network policies, it prefixes the firewall policy name with the cluster name, plus the namespace, in front of our network policy name:

```
<cluster name>-<namespace-name>-<kubernetes network policy name>
```

This should sound familiar. Earlier we showed how `KM` names the virtual network in Contrail UI after the Kubernetes virtual network objects name we created in the YAML file.

The `K8s-ingress` firewall policy is created for the ingress loadbalancer to ensure that ingress to works properly in Contrail. A detailed explanation is beyond the scope of this book.

But the bigger question is, why do we still see two more firewall policies here, since we have never created any network policies like `allowall`, or `denyall`?

Well, remember when we introduced Kubernetes network policy back in Chapter 3, and mentioned that Kubernetes network policy uses a whitelist method and the implicit deny all and allow all policies? The nature of the whitelist method indicates deny all action for all traffic other than what is added in the whitelist, while the implicit allow all behavior makes sure a pod that is not involved in any network policies can continue its allow-any-any traffic model. The problem with Contrail firewall regarding this implicitness is that by default it follows a deny all model - anything that is not explicitly defined will be blocked. That is why in Contrail implementation, these two corresponding implicit network policies are honored by two explicit policies generated by the `KM` module.

One question may be raised at this point. With multiple firewall policies, which one should be applied and evaluated first and which ones later? In other words, in what sequence will Contrail apply and evaluate each policy – a firewall policy evaluation with a different sequence will lead to completely different result. Just imagine these two sequences denyall - allowall versus allowall- denyall. The former gives a deny to all other pods, while the latter gives a pass The answer is the *sequence number*.

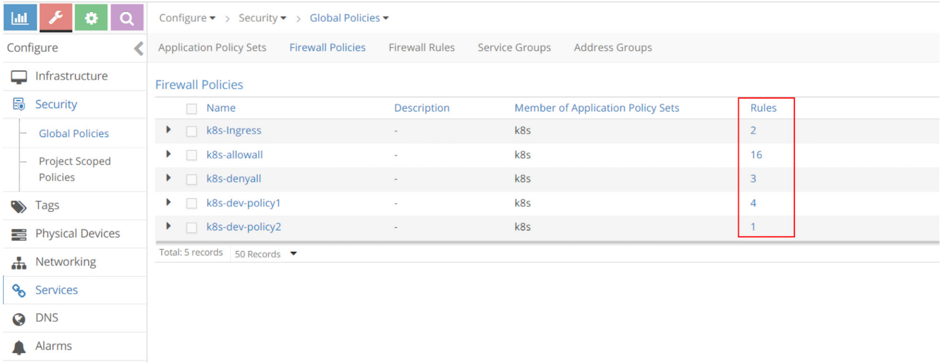
Sequence Number

When firewall polices in an APS are evaluated, they have to be evaluated in a certain sequence. All firewall polices and all firewall rules (we will come to this soon) in each of the policies has a `sequence number`. When there is a matching policy, it will be executed, and the evaluation will stop. It is again `contrail-Kube-manager` that allocates the right sequence number for all firewall policies and firewall rules, so that everything works in the correct order. The process is automatically done without manual intervention. You don’t have to worry about these things when you create the Kubernetes network policies.

We’ll visit sequence numbers again later, but for now let’s look at the rules defined in the firewall policy.

Firewall Policy Rules

In the following capture of the Firewall Policies list, on the right side you can see the number of Rules for each policy.

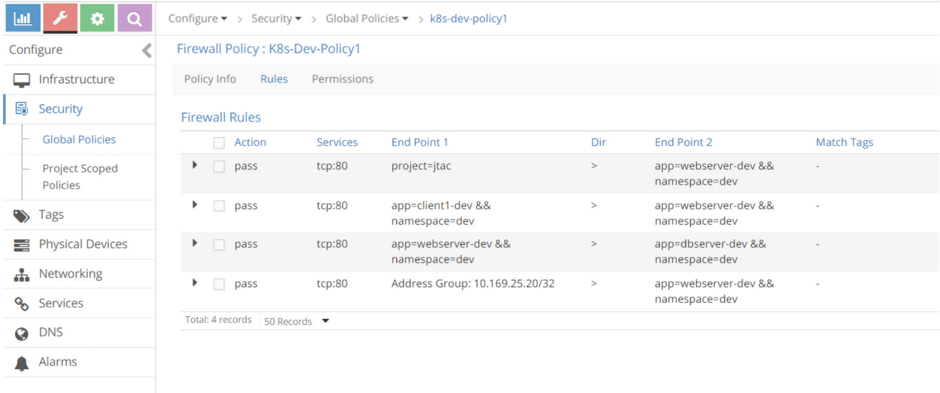


□	Name	Description	Member of Application Policy Sets	Rules
□	k8s-Ingress	-	k8s	2
□	k8s-allowall	-	k8s	16
□	k8s-denyall	-	k8s	3
□	k8s-dev-policy1	-	k8s	4
□	k8s-dev-policy2	-	k8s	1

Total: 5 records 50 Records ▾

Figure 8.10 Contrail UI:Firewall Policy Rules

There are four rules for the `k8s-dev-policy1` policy. Clicking on Rules will show the rules in detail as in Figure 8.11.



□	Action	Services	End Point 1	Dir	End Point 2	Match Tags
□	pass	tcp:80	project=jtac	>	app=webserver-dev && namespace=dev	-
□	pass	tcp:80	app=client1-dev && namespace=dev	>	app=webserver-dev && namespace=dev	-
□	pass	tcp:80	app=webserver-dev && namespace=dev	>	app=dbserver-dev && namespace=dev	-
□	pass	tcp:80	Address Group: 10.169.25.20/32	>	app=webserver-dev && namespace=dev	-

Total: 4 records 50 Records ▾

Figure 8.11 Contrail UI: k8s-dev-policy1 Rules

It looks similar to the Kubernetes network policy `policy1` that we've tested. Let's put the rules, displayed in the screen captures, into Table 8.2.

Table 8.2 Rules

Rule#	Action	Services	End Point1	Dir	End Point2	Match Tags
1	pass	tcp:80	project=jtac	>	app=webserver-dev && namespace=dev	-
2	pass	tcp:80	app=client1-dev && namespace=dev	>	app=webserver-dev && namespace=dev	-
3	pass	tcp:80	app=webserver-dev && namespace=dev	>	app=dbserver-dev && namespace=dev	-
4	pass	tcp:80	Address Group: 10.169.25.20/32	>	app=webserver-dev && namespace=dev	-

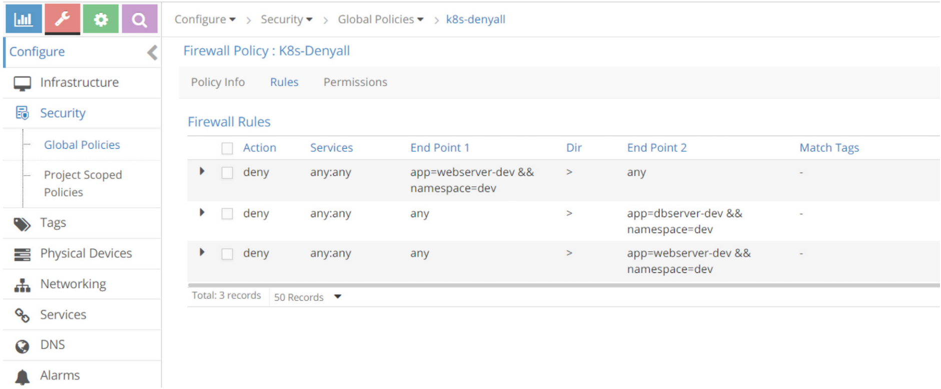
The first column of Table 8.2 is the rule number that we added; all other columns are imported from the UI screenshot. Now let's compare it with the Kubernetes object information:

```
$ kubectl get netpol --all-namespaces -o yaml
apiVersion: v1
items:
- apiVersion: extensions/v1beta1
  kind: NetworkPolicy
  metadata:
    ......
  spec:
    egress:
    - ports:
      - port: 80
        protocol: TCP
      to:
      - podSelector:            #<---rule#3
          matchLabels:
            app: dbserver-dev
    ingress:
    - from:
      - ipBlock:                #<---rule#4
          cidr: 10.169.25.20/32
      - namespaceSelector:      #<---rule#1
          matchLabels:
            project: jtac
      - podSelector:            #<---rule#2
          matchLabels:
            app: client1-dev
      ports:
      - port: 80
        protocol: TCP
    podSelector:
      matchLabels:
        app: webserver-dev
    policyTypes:
    - Ingress
    - Egress
```

The rules we see in firewall policy `k8s-dev-policy1` match with rules in Kubernetes network policy `policy1`.

Rules in `k8s-denyall` Firewall Policy

Now let's go back and examine the rules in the `k8s-denyall` policy that `KM` generated for our Kubernetes network policies.



Firewall Policy : K8s-Denyall

Firewall Rules

	Action	Services	End Point 1	Dir	End Point 2	Match Tags
▸ □	deny	any:any	app=webserver-dev && namespace=dev	>	any	-
▸ □	deny	any:any	any	>	app=dbserver-dev && namespace=dev	-
▸ □	deny	any:any	any	>	app=webserver-dev && namespace=dev	-

Total: 3 records 50 Records ▾

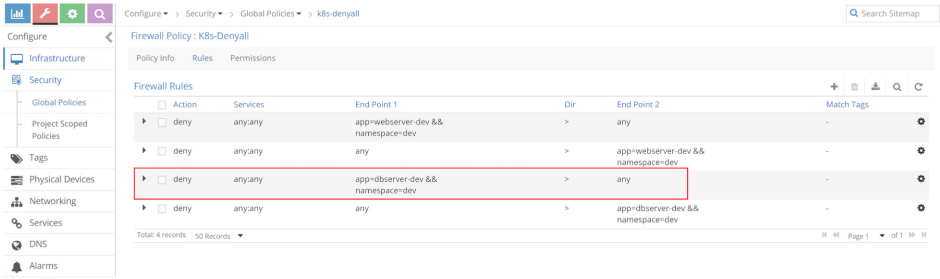
Figure 8.12 Contrail UI: k8s-denyall Policy Rules

Again, if we convert that into a table it appears as shown in Table 8.3.

Table 8.3 Contrail UI: k8s-denyall Policy Rules

rule#	Action	Services	End Point1	Dir	End Point2	Match Tags
1	deny	any:any	app=webserver-dev && namespace=dev	>	any	-
2	deny	any:any	any	>	app=dbserver-dev && namespace=dev	-
3	deny	any:any	any	>	app=webserver-dev && namespace=dev	-

The `k8s-alldeny` rules are simple. They just tell Contrail to deny communication with all other pods that are not in the whitelist. One thing worth mentioning is that there is a rule in the direction from `app=webserver-dev && namespace=dev` to `any`, so that egress traffic is denied for `webserver-dev` pod, while there is no such a rule from `app=dbserver-dev && namespace=dev` to `any`. If you review our test in the last section, in the original policy `policy2`, we did not define an `Egress` option in `policyTypes` to deny egress traffic of `dbserver-dev`, that is why there is no such rule when translated into Contrail firewall, either. If we change `policy2` to the new policy `policy2-egress-deny-all` and examine the same, we'll see the missing rule now.



Firewall Policy : K8s-Denyall

Firewall Rules

Action	Services	End Point 1	Dir	End Point 2	Match Tags
deny	any:any	app=webserver-dev && namespace=dev	>	any	-
deny	any:any	any	>	app=webserver-dev && namespace=dev	-
deny	any:any	app=dbserver-dev && namespace=dev	>	any	-
deny	any:any	any	>	app=dbserver-dev && namespace=dev	-

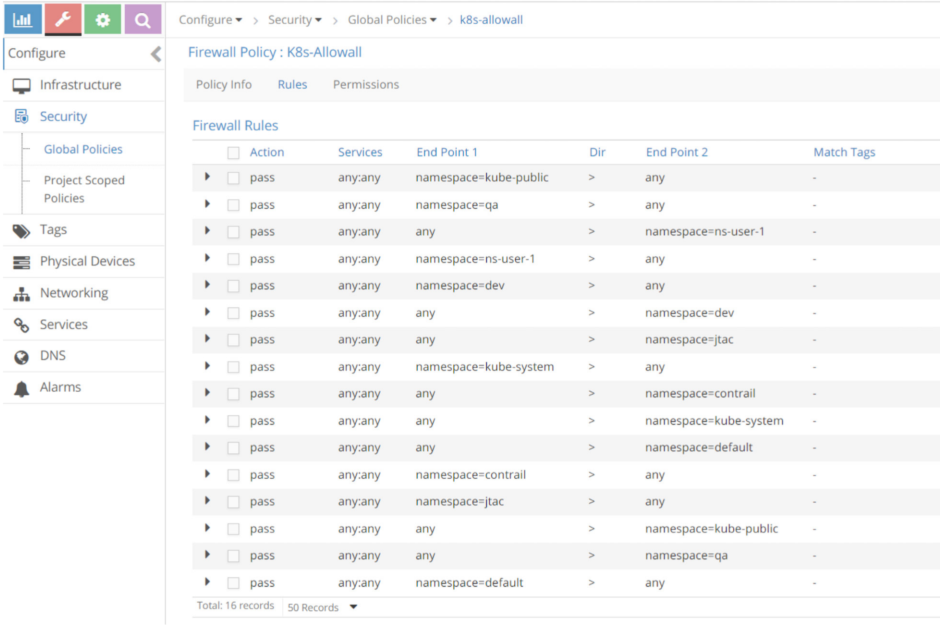
Total: 4 records 50 Records Page 1 of 1

Figure 8.13 Contrail UI: k8s-denyall Rules

Pay attention to the fact that the k8s-denyall policy only applies to those target pods – pods that are selected by the network policies. In this case it only applies to pods `webserver-dev` and `dbserver-dev`. Other pods like `client-jtac` or `client-qa` will not be effected. Instead, those pods will be applied by `k8s-allowany` policy, which we'll examine next.

Rules in k8s-allowall Firewall Policy

In Figure 8.14, the `k8s-allowall` policy seems to have more rules than other policies.



Firewall Policy : K8s-Allowall

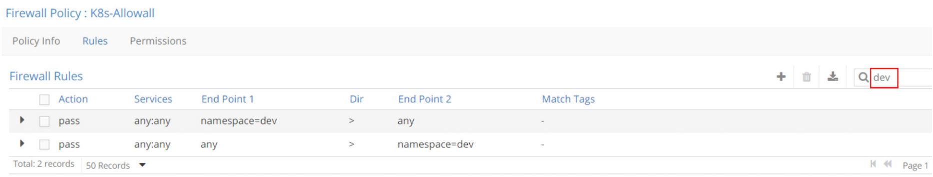
Firewall Rules

Action	Services	End Point 1	Dir	End Point 2	Match Tags
pass	any:any	namespace=kube-public	>	any	-
pass	any:any	namespace=qa	>	any	-
pass	any:any	any	>	namespace=ns-user-1	-
pass	any:any	namespace=ns-user-1	>	any	-
pass	any:any	namespace=dev	>	any	-
pass	any:any	any	>	namespace=dev	-
pass	any:any	any	>	namespace=jtac	-
pass	any:any	namespace=kube-system	>	any	-
pass	any:any	any	>	namespace=contrail	-
pass	any:any	any	>	namespace=kube-system	-
pass	any:any	any	>	namespace=default	-
pass	any:any	namespace=contrail	>	any	-
pass	any:any	namespace=jtac	>	any	-
pass	any:any	any	>	namespace=kube-public	-
pass	any:any	any	>	namespace=qa	-
pass	any:any	namespace=default	>	any	-

Total: 16 records 50 Records

Figure 8.14 Contrail UI: k8s-allowall Rules

Despite the number of rules, in fact `k8s-allowall` is the simplest one. It works at the NS level and simply has two rules for each NS. In the UI, within the search field, key in a namespace as the filter, for example, *dev* or *qa*, and you'll see these results in Figure 8.15 and 8.16.



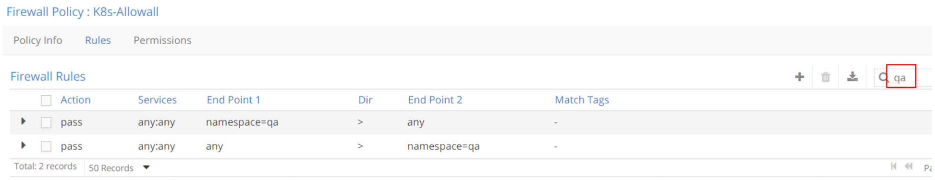
Firewall Policy : K8s-Allowall

Firewall Rules

	Action	Services	End Point 1	Dir	End Point 2	Match Tags
□	pass	any:any	namespace=dev	>	any	-
□	pass	any:any	any	>	namespace=dev	-

Total: 2 records 50 Records Page 1

Figure 8.15 Contrail UI: k8s-allowall Rules Filtered by NS ***dev***



Firewall Policy : K8s-Allowall

Firewall Rules

	Action	Services	End Point 1	Dir	End Point 2	Match Tags
□	pass	any:any	namespace=qa	>	any	-
□	pass	any:any	any	>	namespace=qa	-

Total: 2 records 50 Records Pa

Figure 8.16 Contrail UI: k8s-allowall Rules Filtered by NS ***qa***

What this policy says is: for those pods that do not have any network policy applied yet, let's continue the Kubernetes default allow-any-any networking model and allow everything!

Sequence Number

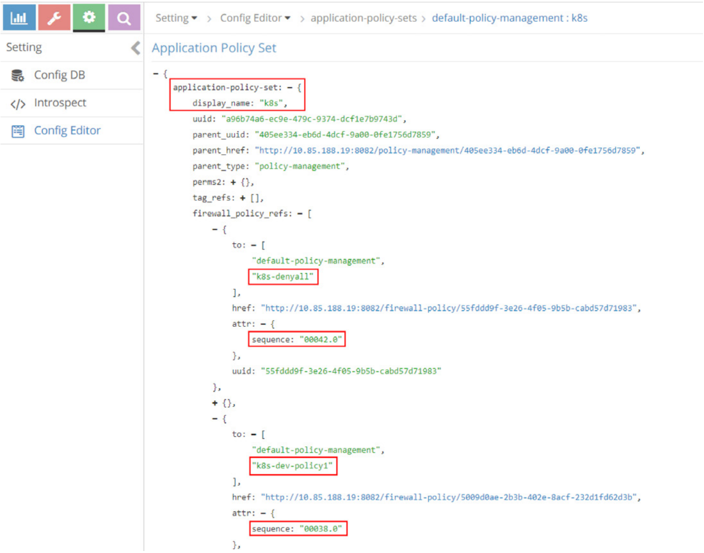
After having explored the Contrail firewall policy rules, let's come back to the `sequence number` and see exactly how it works.

The `sequence number` is a number attached to all firewall policies and their rules that decides the order in which all policies are applied and evaluated, and does the same in one particular policy. The lower the sequence number the higher the priority. To find the `sequence number` you have to look into the firewall policy and policy rule object attributes in Contrail configuration database.

First let's explore the firewall policy object in APS to check their `sequence number`.

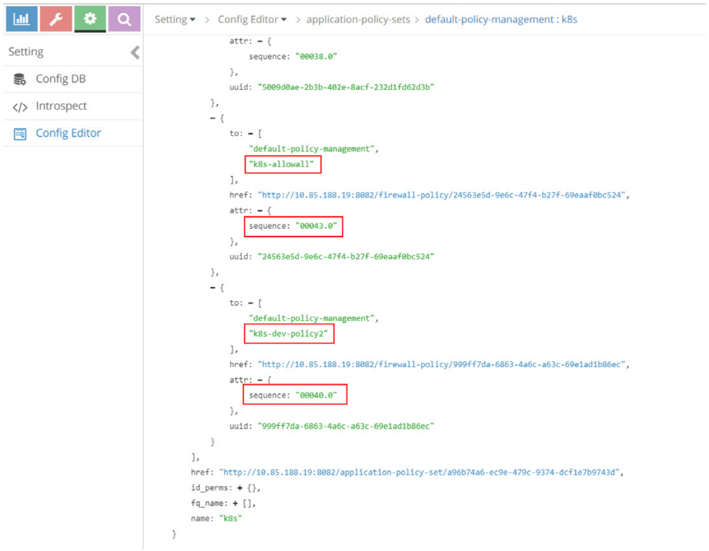
TIP In Chapter 5 we used the `curl` command to pull the loadbalancer object data when we introduced `service`. Here we used Config Editor to do the same.

Figures 8.17 and 8.18 capture the sequence number in firewall policies.



```
- {
    application-policy-set: - {
        display_name: "k8s",
        uuid: "a96b74a6-ec9e-479c-9374-dcf1e7b9743d",
        parent_uuid: "405ee334-eb6d-4dcf-9a00-0fe1756d7859",
        parent_href: "http://10.85.188.19:8082/policy-management/405ee334-eb6d-4dcf-9a00-0fe1756d7859",
        parent_type: "policy-management",
        perms2: + {},
        tag_refs: + [],
        firewall_policy_refs: - [
            - {
                to: - [
                    "default-policy-management",
                    "k8s-denyall"
                ],
                href: "http://10.85.188.19:8082/firewall-policy/55fddd9f-3e26-4f05-9b5b-cabd57d71983",
                attr: - {
                    sequence: "00042.0"
                },
                uuid: "55fddd9f-3e26-4f05-9b5b-cabd57d71983"
            },
            + {},
            - {
                to: - [
                    "default-policy-management",
                    "k8s-dev-policy1"
                ],
                href: "http://10.85.188.19:8082/firewall-policy/5009d0ae-2b3b-402e-8acf-232d1fd62d3b",
                attr: - {
                    sequence: "00038.0"
                },
```

Figure 8.17 *Contrail UI: Sequence Number for Policies: Setting> Config Editor*



```
                attr: - {
                    sequence: "00038.0"
                },
                uuid: "5009d0ae-2b3b-402e-8acf-232d1fd62d3b"
            },
            - {
                to: - [
                    "default-policy-management",
                    "k8s-allowall"
                ],
                href: "http://10.85.188.19:8082/firewall-policy/24563e5d-9e6c-47f4-b27f-69eaaf0bc524",
                attr: - {
                    sequence: "00043.0"
                },
                uuid: "24563e5d-9e6c-47f4-b27f-69eaaf0bc524"
            },
            - {
                to: - [
                    "default-policy-management",
                    "k8s-dev-policy2"
                ],
                href: "http://10.85.188.19:8082/firewall-policy/999ff7da-6863-4a6c-a63c-69e1ad1b86ec",
                attr: - {
                    sequence: "00040.0"
                },
                uuid: "999ff7da-6863-4a6c-a63c-69e1ad1b86ec"
            }
        ],
        href: "http://10.85.188.19:8082/application-policy-set/a96b74a6-ec9e-479c-9374-dcf1e7b9743d",
        id_perms: + {},
        fq_name: + [],
        name: "k8s"
    }
```

Figure 8.18 *Contrail UI: Sequence Number for Policies (continued)*

All five policies that we've seen appear in these screenshots, under APS `k8s`. For example, the policy `k8s-dev-policy1`, which maps to the Kubernetes network policy `policy1` that we explicitly defined, and the policy `k8s-denyall`, which is what the system automatically generated. The figures show `k8s-dev-policy1` and `k8s-denyall` have sequence numbers of 00038.0 and 00042.0, respectively. Therefore `k8s-dev-policy1` has a higher priority and it will be applied and evaluated first. That means the traffic types we defined in the whitelist will be allowed first, then all other traffic to or from the target pod will be denied. This is the exact goal that we wanted to achieve.

All `sequence numbers` for all firewall policies are listed in Table 8.4, from the highest priority to the lowest:

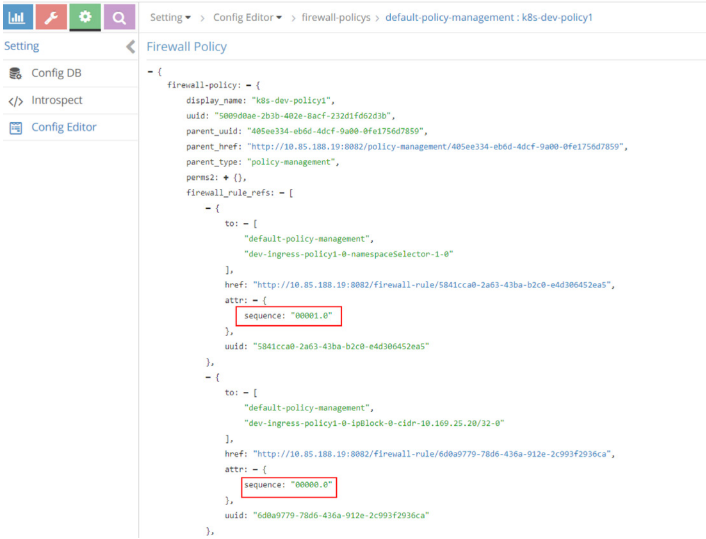
Table 8.4 Sequence Numbers

Sequence Number	Firewall Policy
00002.0	k8s-Ingress
00038.0	k8s-dev-policy1
00040.0	k8s-dev-policy2
00042.0	k8s-denyall
00043.0	k8s-allowall

Based on the sequence number, the application and evaluation order are the first explicit policies, followed by the deny all policy and ending with the allow all policy. The same order as in Kubernetes is honored.

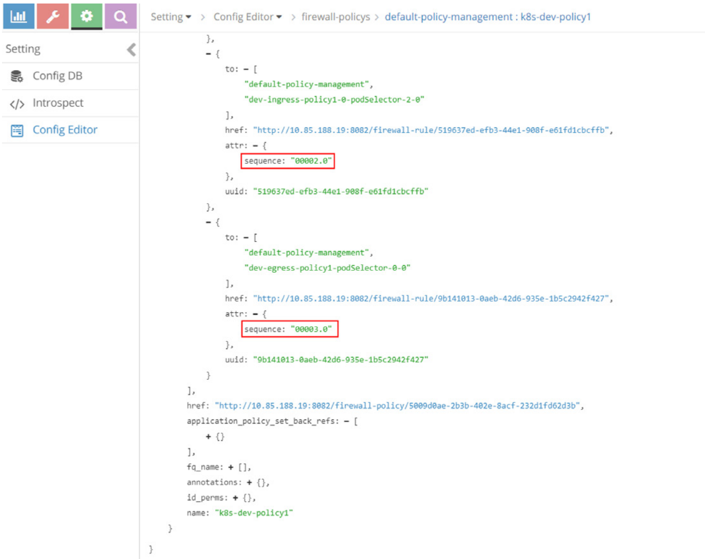
Sequence Number in Firewall Policy Rules

As mentioned previously, in the same firewall policy, policy rules will also have to be applied and evaluated in a certain order. In Contrail firewall that is again ensured by the sequence number. The sequence numbers in the rules of firewall policy `k8s-dev-policy1` are displayed in Figures 8.19 and 8.20.


Setting ▾ > Config Editor ▾ > firewall-policys > default-policy-management : k8s-dev-policy1
Setting
Config DB
Introspect
Config Editor
Firewall Policy

```
- {
    firewall-policy: - {
        display_name: "k8s-dev-policy1",
        uuid: "5009d0ae-2b3b-402e-8acf-232d1fd62d3b",
        parent_uuid: "405ee334-eb6d-4dcf-9a00-0fe1756d7859",
        parent_href: "http://10.85.188.19:8082/policy-management/405ee334-eb6d-4dcf-9a00-0fe1756d7859",
        parent_type: "policy-management",
        perms2: + {},
        firewall_rule_refs: - [
            - {
                to: - [
                    "default-policy-management",
                    "dev-ingress-policy1-0-namespaceSelector-1-0"
                ],
                href: "http://10.85.188.19:8082/firewall-rule/5841cca0-2a63-43ba-b2c0-e4d306452ea5",
                attr: - {
                    sequence: "00001.0"
                },
                uuid: "5841cca0-2a63-43ba-b2c0-e4d306452ea5"
            },
            - {
                to: - [
                    "default-policy-management",
                    "dev-ingress-policy1-0-ipBlock-0-cidr-10.169.25.20/32-0"
                ],
                href: "http://10.85.188.19:8082/firewall-rule/6d0a9779-78d6-436a-912e-2c993f2936ca",
                attr: - {
                    sequence: "00000.0"
                },
                uuid: "6d0a9779-78d6-436a-912e-2c993f2936ca"
            },
```

Figure 8.19 Contrail UI Sequence Number for Rules: Setting> Config Editor


Setting ▾ > Config Editor ▾ > firewall-policys > default-policy-management : k8s-dev-policy1
Setting
Config DB
Introspect
Config Editor

```
            },
            - {
                to: - [
                    "default-policy-management",
                    "dev-ingress-policy1-0-podSelector-2-0"
                ],
                href: "http://10.85.188.19:8082/firewall-rule/519637ed-efb3-44e1-908f-e61fd1cbcffb",
                attr: - {
                    sequence: "00002.0"
                },
                uuid: "519637ed-efb3-44e1-908f-e61fd1cbcffb"
            },
            - {
                to: - [
                    "default-policy-management",
                    "dev-egress-policy1-podSelector-0-0"
                ],
                href: "http://10.85.188.19:8082/firewall-rule/9b141013-0aeb-42d6-935e-1b5c2942f427",
                attr: - {
                    sequence: "00003.0"
                },
                uuid: "9b141013-0aeb-42d6-935e-1b5c2942f427"
            }
        ],
        href: "http://10.85.188.19:8082/firewall-policy/5009d0ae-2b3b-402e-8acf-232d1fd62d3b",
        application_policy_set_back_refs: - [
            + {}
        ],
        fq_name: + [],
        annotations: + {},
        id_perms: + {},
        name: "k8s-dev-policy1"
    }
}
```

Figure 8.20 Contrail UI Sequence Number for Rules (continued)

And Table 8.5 lists the `sequence number` of all rules of the firewall policy `k8s-dev-policy1`, from highest priority to the lowest.

Table 8.5 Sequence Numbers for Policy1

seq#	firewall rule
00000.0	dev-ingress-policy1-0-ipBlock-0-cidr-10.169.25.20/32-0
00001.0	dev-ingress-policy1-0-namespaceSelector-1-0
00002.0	dev-ingress-policy1-0-podSelector-2-0
00003.0	dev-egress-policy1-podSelector-0-0

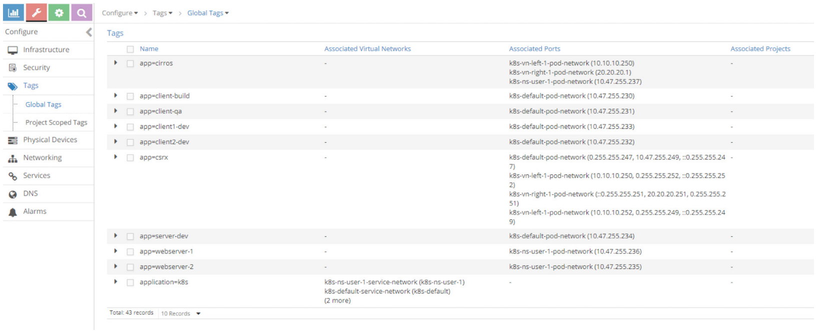
Let's compare with our network policy YAML file configuration:

```
ingress:
- from:
  - ipBlock:
      cidr: 10.169.25.20/32           #<---seq# 00000.0
  - namespaceSelector:                #<---seq# 00001.0
      matchLabels:
        project: jtac
  - podSelector:                      #<---seq# 00002.0
      matchLabels:
        app: client1-dev
  ports:
  - protocol: TCP
    port: 80
egress:
- to:
  - podSelector:                      #<---seq# 00003.0
      matchLabels:
        app: dbserver-dev
  ports:
  - protocol: TCP
    port: 80
```

We find that the rules `sequence number` is consistent with the sequence that appears in the YAML file. In other words, rules will be applied and evaluated in the same order as they are defined.

Tag

We've been talking about the Contrail `tags` and we already know that `contrail-kube-manager` will translate each Kubernetes label into a Contrail tag, which is attached to the respective port of a pod as shown in Figure 8.21.



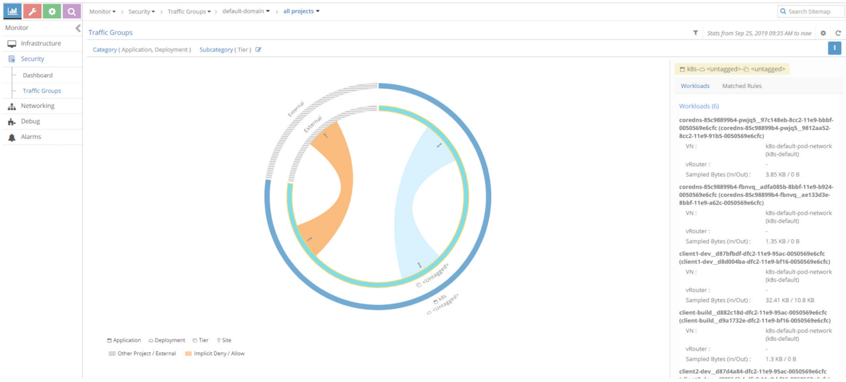
	Name	Associated Virtual Networks	Associated Ports	Associated Projects
▸ □	app=cirros	-	k8s-vn-left-1-pod-network (10.10.10.250) k8s-vn-right-1-pod-network (20.20.20.1) k8s-ns-user-1-pod-network (10.47.255.237)	-
▸ □	app=client-build	-	k8s-default-pod-network (10.47.255.230)	-
▸ □	app=client-qa	-	k8s-default-pod-network (10.47.255.231)	-
▸ □	app=client1-dev	-	k8s-default-pod-network (10.47.255.233)	-
▸ □	app=client2-dev	-	k8s-default-pod-network (10.47.255.232)	-
▸ □	app=csrx	-	k8s-default-pod-network (0.255.255.247, 10.47.255.249, ::0.255.255.247) k8s-vn-left-1-pod-network (10.10.10.250, 0.255.255.252, ::0.255.255.252) k8s-vn-right-1-pod-network (::0.255.255.251, 20.20.20.251, 0.255.255.251) k8s-vn-left-1-pod-network (10.10.10.252, 0.255.255.249, ::0.255.255.249)	-
▸ □	app=server-dev	-	k8s-default-pod-network (10.47.255.234)	-
▸ □	app=webserver-1	-	k8s-ns-user-1-pod-network (10.47.255.236)	-
▸ □	app=webserver-2	-	k8s-ns-user-1-pod-network (10.47.255.235)	-
▸ □	application=k8s	k8s-ns-user-1-service-network (k8s-ns-user-1) k8s-default-service-network (k8s-default) (2 more)	-	-

Total: 43 records 10 Records ▾

Figure 8.21 Tags UI

UI Visualization

Contrail UI provides a nice visualization for security as shown in Figure 8.22. It's self- explanatory if you know how Contrail security works.



Workloads (6)

coredns-85c98899b4-pwjq5__97c148eb-8cc2-11e9-bbbf-0050569e6cfc (coredns-85c98899b4-pwjq5__9812aa52-8cc2-11e9-91b5-0050569e6cfc)

VN : k8s-default-pod-network (k8s-default)
vRouter : -
Sampled Bytes (In/Out) : 3.85 KB / 0 B

coredns-85c98899b4-fbnvq__adfa085b-8bbf-11e9-b924-0050569e6cfc (coredns-85c98899b4-fbnvq__ae133d3e-8bbf-11e9-a62c-0050569e6cfc)

VN : k8s-default-pod-network (k8s-default)
vRouter : -
Sampled Bytes (In/Out) : 1.35 KB / 0 B

client1-dev__d87bfbdf-dfc2-11e9-95ac-0050569e6cfc (client1-dev__d8d004ba-dfc2-11e9-bf16-0050569e6cfc)

VN : k8s-default-pod-network (k8s-default)
vRouter : -
Sampled Bytes (In/Out) : 32.41 KB / 10.8 KB

client-build__d882c18d-dfc2-11e9-95ac-0050569e6cfc (client-build__d9a1732e-dfc2-11e9-bf16-0050569e6cfc)

VN : k8s-default-pod-network (k8s-default)
vRouter : -
Sampled Bytes (In/Out) : 1.3 KB / 0 B

client2-dev__d87d4a84-dfc2-11e9-95ac-0050569e6cfc

Figure 8.22 Sample Traffic Visualization for the Above Policy with Workload

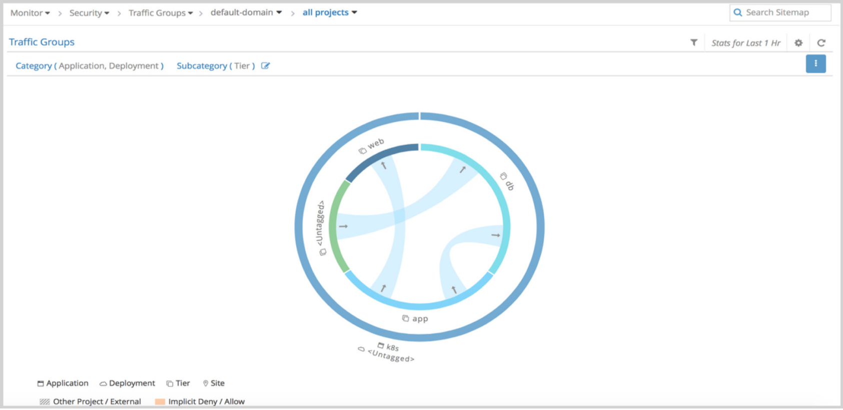


Figure 8.23 Sample Traffic Visualization with More Network Policies

Chapter 9

Contrail Multiple Interface Pod

Typically each pod in the Kubernetes cluster has only one network interface (except the `loopback` interface). In reality, there are scenarios where multiple interfaces are required. For example, a VNF (virtualized network function) typically needs a left, right, and optionally, a management interface to complete network functions. A pod may require a data interface to carry the data traffic, and a management interface for the reachability detection. Service providers also tend to keep the management and tenant networks independent for isolation and management purposes. Multiple interfaces provide a way for containers to be simultaneously connected to multiple devices in multiple networks.

Contrail as a CNI

In container technology, a *veth* (virtual Ethernet) pair functions pretty much like a virtual cable that can be used to create tunnels between network namespaces. One end of it is plugged in the container and the other end is in the host or docker bridge namespace.

A Contrail CNI plugin is responsible for inserting the network interface (that is one end of the veth pair) into the container network namespace and it will also make all the necessary changes on the host, like attaching the other end of the veth into a bridge, assigning IP, configuring routes, and so on.

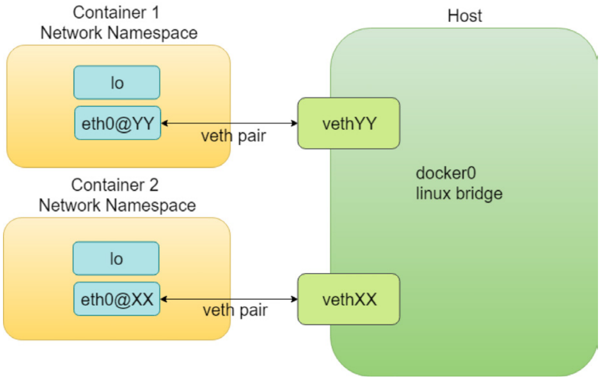


Figure 9.1 The Container and veth Pair

There are many such CNI plugin implementations that are publicly available today. Contrail is one of them and it is our favorite. For a comprehensive list check https://github.com/containernetworking/cni.

Another CNI plugin, `multus-cni`, enables you to attach multiple network interfaces to pods. Multiple-network support of `multus-cni` is accomplished by Multus calling multiple other CNI plugins. Because each plugin will create its own network, multiple plugins allow the pod to have multiple networks. One of the main advantages that Contrail provides, compared to `mutus-cni`, and all other current implementations in the industry, is that Contrail provides the ability to attach multiple network interfaces to a Kubernetes pod by itself, without having to call in any other plugins. This brings support to a truly multi-homed pod.

Network Attachment Definition CRD

Contrail CNI follows the Kubernetes CRD (Custom Resource Definition) *Network Attachment Definition* to provide a standardized method to specify the configurations for additional network interfaces. There is no change to the standard Kubernetes upstream APIs, making the implementation with the most compatibilities.

In Contrail, the Network Attachment Definition CRD is created by `contrail-kube-manager(KM)`. When booted up, `KM` validates if a network CRD `network-attachment-definitions.k8s.cni.cncf.io` is found in the Kubernetes API server, and creates one if it isn't.

Here is a `CRD` object YAML file:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: network-attachment-definitions.k8s.cni.cncf.io
spec:
  group: k8s.cni.cncf.io
  version: v1
  scope: Namespaced
  names:
    plural: network-attachment-definitions
    singular: network-attachment-definition
    kind: NetworkAttachmentDefinition
    shortNames:
    - net-attach-def
  validation:
    openAPIV3Schema:
      properties:
        spec:
          properties:
            config:
             type: string
```

In the Contrail Kubernetes setup, the CRD has been created and can be verified:

```
$ kubectl get crd
NAME                                              CREATED AT
network-attachment-definitions.k8s.cni.cncf.io   2019-06-07T03:43:52Z
```

Using this new kind of `Network-Attachment-Definition` created from the above CRD, we now have the ability to create a virtual network in Contrail Kubernetes environments.

To create a virtual network from Kubernetes, use a YAML template like this:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: <network-name>
  namespace: <namespace-name>
  annotations:
    "opencontrail.org/cidr" : [<ip-subnet>]
    "opencontrail.org/ip_fabric_snat" : <True/False>
    "opencontrail.org/ip_fabric_forwarding" : <True/False>
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "contrail-k8s-cni"
}'
```

Like many other standard Kubernetes objects, you specify the virtual network name, the namespace under `metadata`, and the `annotations` that are used to carry additional information about a network. In Contrail, the following annotations are used in the `NetworkAttachmentDefinition` CRD to enable certain attributes for the virtual network:

- `opencontrail.org/cidr`: This CIDR defines the subnet for a virtual network.
- `opencontrail.org/ip_fabric_forwarding`: This flag is to enable/disable the `ip fabric forwarding` feature.
- `opencontrail.org/ip_fabric_snat`: This is a flag to enable/disable the `ip fabric snat` feature.

In Contrail, the `ip-fabric-forwarding` feature enables IP fabric-based forwarding without tunneling for the virtual network. When two `ip_fabric_forwrding` enabled virtual networks communicate with each other, the overlay traffic will be forwarded directly using the underlay.

With the Contrail `ip-fabric-snat` feature, pods that are in the overlay can reach the Internet without floating IPs or a logical router. The `ip-fabric-snat` feature uses compute node IP for creating a source NAT to reach the required services.

Note that `ip fabric forwarding` and `ip fabric snat` features are not covered in this book.

Alternatively, you can define a new virtual network by referring an existing virtual network:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: extns-network
  annotations:
    "opencontrail.org/network" : '{"domain":"default-domain", "project": "k8s-extns", "name":"k8s-
extns-pod-network"}'
spec:
  config: '{
    "cniVersion": "0.3.1",
    "type": "contrail-k8s-cni"
}'
```

In this book we use the first template to define our virtual networks in all examples.

Multiple Interface Pod

With multiple virtual networks created, you can attach (you could also say *plug*, or *insert*) any of them into a pod, with the pod's YAML file like this:

```
kind: Pod
metadata:
  name: my-pod
  namespace: my-namespace
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
      { "name": "VN-a" },
      { "name": "VN-b" },
      { "namespace": "other-ns", "name": "VN-c" }
    ]'
spec:
  containers:
```

Or, another valid format:

```
kind: Pod
metadata:
  name: my-pod
  namespace: my-namespace
  annotations:
    k8s.v1.cni.cncf.io/networks: 'VN-a,VN-b,other-ns/VN-c'
spec:
  containers:
```

You've probably noticed that pods in a namespace can not only refer to the networks defined in the local namespace, but also the networks created on other namespaces using their fully scoped name. This is very useful. The same network does not have to be duplicated again and again in every namespace that needs it. It can be defined once and then referred to everywhere else.

We've gone through the basic theories and explored the various templates, so let's get a working example in the real world. Let's start by creating two virtual networks, examining the virtual network objects, then create a pod and attach the two virtual networks into it. We'll conclude by examining the pod interfaces and the connectivity with other pods sharing the same virtual networks.

Here is the YAML file of the two virtual networks (`vn-left-1` and `vn-right-1`):

```
#vn-left-1.yaml
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  annotations:
    "opencontrail.org/cidr": "10.10.10.0/24"
    "opencontrail.org/ip_fabric_forwarding": "false"
    "opencontrail.org/ip_fabric_snat": "false"
  name: vn-left-1
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "contrail-k8s-cni"
  }'
#vn-right-1.yaml
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  annotations:
    "opencontrail.org/cidr": "20.20.20.0/24"
    "opencontrail.org/ip_fabric_forwarding": "false"
    "opencontrail.org/ip_fabric_snat": "false"
  name: vn-right-1
  #namespace: default
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "contrail-k8s-cni"
  }'
```

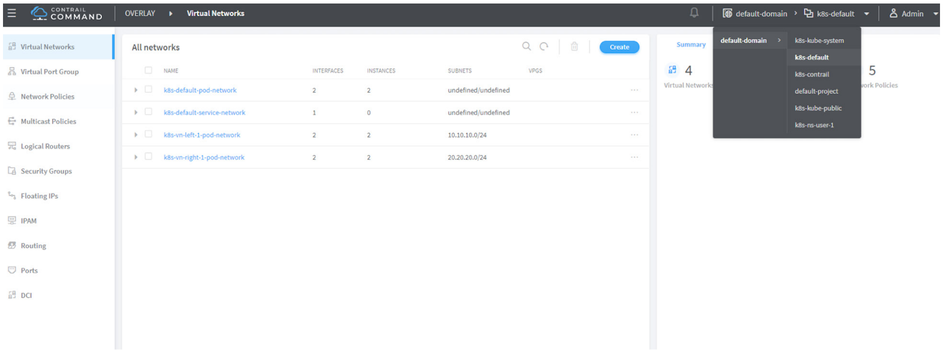
Now create both virtual networks:

```
$ kubectl apply -f vn-left-1.yaml
networkattachmentdefinition.k8s.cni.cncf.io/vn-left-1 created

$ kubectl apply -f vn-right-1.yaml
networkattachmentdefinition.k8s.cni.cncf.io/vn-right-1 created
```

Examine the virtual networks:

```
$ kubectl get network-attachment-definitions.k8s.cni.cncf.io
NAME            AGE
vn-left-1       3s
vn-right-1      10s
$ kubectl get network-attachment-definitions.k8s.cni.cncf.io vn-left-1 -o yaml
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"k8s.cni.cncf.io/v1","kind":"NetworkAttachmentDefinition","metadata":{"annotatio
ns":{"opencontrail.org/cidr":"10.10.10.0/24","opencontrail.org/ip_fabric_
forwarding":"false"},"name":"vn-left-1","namespace":"ns-user-
1"},"spec":{"config":"{ \"cniVersion\": \"0.3.0\", \"type\": \"contrail-k8s-cni\" }"}}
    opencontrail.org/cidr: 10.10.10.0/24
    opencontrail.org/ip_fabric_forwarding: "false"
  creationTimestamp: 2019-06-13T14:17:42Z
  generation: 1
  name: vn-left-1
  namespace: ns-user-1
  resourceVersion: "777874"
  selfLink: /apis/k8s.cni.cncf.io/v1/namespaces/ns-user-1/network-attachment-definitions/vn-left-1
  uid: 01f167ad-8de6-11e9-bbbf-0050569e6cfc
spec:
  config: '{ "cniVersion": "0.3.0", "type": "contrail-k8s-cni" }'
```

The virtual networks are created, as expected. Nothing much exciting here but, if you log in to the Contrail UI, you will see something unexpected in the next screen capture, Figure 9.2.



All networks

NAME	INTERFACES	INSTANCES	SUBNETS	VPGS
k8s-default-pod-network	2	2	undefined/undefined	
k8s-default-service-network	1	0	undefined/undefined	
k8s-vn-left-1-pod-network	2	2	10.10.10.0/24	
k8s-vn-right-1-pod-network	2	2	20.20.20.0/24	

Figure 9.2 Contrail Command: Main Menu > Virtual Networks

NOTE Make sure you select the correct project, in this case it is `k8s-default`.

And what you'll find is the lack of any virtual network with the exact name of `vn-left-1` or `vn-right-1` in the UI. Instead, there two virtual networks, named `k8s-vn-left-1-pod-network` and `k8s-vn-right-1-pod-network`.

There is nothing wrong here. What happened is whenever a virtual network gets created from Kubernetes, Contrail automatically adds the Kubernetes cluster name (by default `k8s`) as a prefix to the virtual network name that you give in your network YAML file, and a suffix `-pod-network` in the end. This makes sense because we know a virtual network can be created by different methods and with these extra keywords embedded in the name, it's easier to tell how the virtual network was created (from Kubernetes or manually from the UI) and what will it be used for. Also, potential virtual network name conflicts can be avoided when working across multiple Kubernetes clusters.

Here is YAML file of a pod with multiple virtual networks:

```
#pod-webserver-multivn-do.yaml
apiVersion: v1
kind: Pod
metadata:
  name: webserver-mv
  labels:
    app: webserver-mv
  annotations:
   k8s.v1.cni.cncf.io/networks: '[
       { "name": "vn-left-1" },
       { "name": "vn-right-1" }
   ]'
spec:
  containers:
  - name: webserver-mv
    image: contrailk8sdayone/contrail-webserver
    imagePullPolicy: Always
  restartPolicy: Always
```

In pod annotations under metadata, insert two virtual networks: `vn-left-1` and `vn-right-1`. Now, guess how many interfaces the pod will have on bootup? You might think two because that is what we specified in the file. Let's create the pod and verify:

```
$ kubectl get pod -o wide
NAME         READY  STATUS   RESTARTS  AGE  IP             NODE     NOMINATED  NODE
webserver-mv  1/1     Running  0         20s  10.47.255.238  cent222  <none>

$ kubectl describe pod webserver-mv
Name:               webserver-mv
Namespace:          ns-user-1
Priority:           0
PriorityClassName:  <none>
Node:               cent222/10.85.188.20
Start Time:         Wed, 26 Jun 2019 12:51:30 -0400
```

```
Labels:             app=webserver-mv
Annotations:        k8s.v1.cni.cncf.io/network-status:
                      [
                          {
                              "ips": "10.10.10.250",
                              "mac": "02:87:cf:6c:9a:98",
                              "name": "vn-left-1"
                          },
                          {
                              "ips": "10.47.255.238",
                              "mac": "02:87:98:cc:4e:98",
                              "name": "cluster-wide-default"
                          },
                          {
                              "ips": "20.20.20.1",
                              "mac": "02:87:f9:f9:88:98",
                              "name": "vn-right-1"
                          }
                      ]
                    k8s.v1.cni.cncf.io/networks: [
                        { "name": "vn-left-1" }, { "name": "vn-right-1" } ]
                    kubectl.kubernetes.io/last-applied-configuration:
                      {"apiVersion":"v1","kind":"Pod","metadata":
                      {"annotations":{"k8s.v1.cni.cncf.io/networks":"[
                      { \"name\": \"vn-left-1\" }, { \"name\": \"vn-...
Status:             Running
IP:                 10.47.255.238
...<snipped>...
```

In Annotations, under k8s.v1.cni.cncf.io/network-status, you can see a list […], which has three items, each represented by a curly brace {} of key-value mappings. Each curly brace includes information about one interface: the allocated IP, the MAC, and the virtual network it belongs to. So you will end up having three interfaces created in the pod instead of two.

Notice the second item stating the IP address 10.47.255.238. It is the interface attached to the default pod network named cluster-wide-default, which is created by the system. You can look at the default pod network as a management network because it is always up and running in every pod's network namespace. Functionally, it's not much different from the virtual network you created, except that you can't delete it.

Let's log in to the pod, list the interfaces, and verify the IP and MAC addresses:

```
$ kubectl exec -it webserver-mv sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
37: eth0@if38: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
```

```
    link/ether 02:87:98:cc:4e:98 brd ff:ff:ff:ff:ff:ff
    inet 10.47.255.238/12 scope global eth0
       valid_lft forever preferred_lft forever
39: eth1@if40: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:87:cf:6c:9a:98 brd ff:ff:ff:ff:ff:ff
    inet 10.10.10.250/24 scope global eth1
       valid_lft forever preferred_lft forever
41: eth2@if42: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:87:f9:f9:88:98 brd ff:ff:ff:ff:ff:ff
    inet 20.20.20.1/24 scope global eth2
       valid_lft forever preferred_lft forever
```

You can see one `lo` interface and three interfaces plugged by Contrail CNI, each with the IP allocated from the corresponding virtual network. Also notice the MAC addresses match what we've seen in the `kubectl describe` command output.

NOTE Having the MAC address in the annotations can be useful under certain cases. For example, in the service chaining section, you will run into a scenario where you have to use the MAC address to locate the proper interface, so that you can assign the right podIP that Kubernetes allocated from a virtual network. Read on for more details.

You'll see the multiple-interfaces pod again in the example where the pod will be based on a Juniper cSRX image instead of a general Docker image. The basic idea remains the same.

Chapter 10

Contrail Service Chaining with CSRX

Service chaining is the concept of forwarding traffic through multiple network entities in a certain order, and each network entity performs specific functions, such as firewall, IPS, NAT, LB, etc. The legacy way of doing service chaining would be to use standalone HW appliances, but this makes service chaining inflexible, expensive, and lengthens set up times. In *dynamic* service chaining network functions are deployed as a VM or a container and can be chained automatically in a logical way. For example, Figure 10.1 uses Contrail for service chaining between two pods in two different networks using a cSRX container Level 4 – Level 7 firewall to secure the traffic between them.

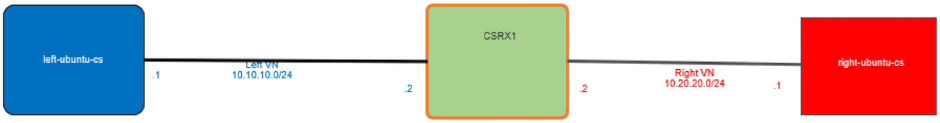


Figure 10.1 *Service Chaining*

NOTE *Left* and *right* networks are used here just for simplicity's sake, to follow the flow from left to right, but you can use your own names of course. Make sure to configure the network before you attach a pod to it or else the pod will not be created.

Bringing Up Client and CSRX Pods

Let's create two virtual networks using this YAML file:

```
# cat vn-left.yaml
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  annotations:
    opencontrail.org/cidr: "10.10.10.0/24"
    opencontrail.org/ip_fabric_forwarding: "false"
    opencontrail.org/ip_fabric_snat: "false"
  name: vn-left
  namespace: default
spec:
 config: '{ "cniVersion": "0.3.0", "type": "contrail-k8s-cni" }'

# cat vn-right.yaml
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  annotations:
    opencontrail.org/cidr: "10.20.20.0/24"
    opencontrail.org/ip_fabric_forwarding: "false"
    opencontrail.org/ip_fabric_snat: "false"
  name: vn-right
  namespace: default
spec:
 config: '{ "cniVersion": "0.3.0", "type": "contrail-k8s-cni" }'
# kubectl create -f vn-left.yaml
# kubectl create -f vn-right.yaml
```

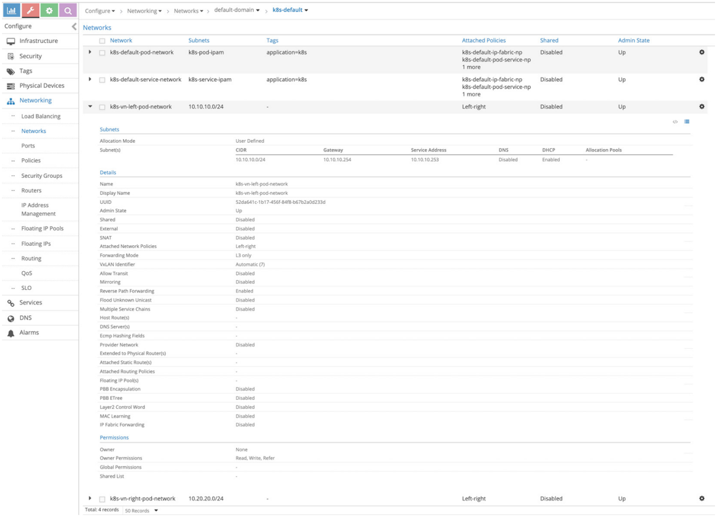
Verify using `Kubectl`:

```
# kubectl get network-attachment-definition
NAME      AGE
vn-left   19d
vn-right  17d

# kubectl describe network-attachment-definition
Name:        vn-left
Namespace:   default
Labels:      <none>
Annotations:  opencontrail.org/cidr: 10.10.10.0/24
           opencontrail.org/ip_fabric_forwarding: false
           opencontrail.org/ip_fabric_snat: false
API Version:  k8s.cni.cncf.io/v1
Kind:        NetworkAttachmentDefinition
Metadata:
  Creation Timestamp:  2019-05-25T20:28:22Z
  Generation:        1
  Resource Version:   83111
  Self Link:        /apis/k8s.cni.cncf.io/v1/namespaces/default/network-attachment-definitions/
vn-left
  UID:             a44fe276-7f2b-11e9-9ff0-0050569e2171
Spec:
  Config:  { "cniVersion": "0.3.0", "type": "contrail-k8s-cni" }
```

```
Events:    <none>


Name:         vn-right
Namespace:    default
Labels:       <none>
Annotations:  opencontrail.org/cidr: 10.20.20.0/24
              opencontrail.org/ip_fabric_forwarding: false
              opencontrail.org/ip_fabric_snat: false
API Version:  k8s.cni.cncf.io/v1
Kind:         NetworkAttachmentDefinition
Metadata:
  Creation Timestamp:  2019-05-28T07:14:02Z
  Generation:          1
  Resource Version:    380427
  Self Link:           /apis/k8s.cni.cncf.io/v1/namespaces/default/network-attachment-definitions/
vn-right
  UID:                 2b8d394f-8118-11e9-b36d-0050569e2171
Spec:
  Config:  { "cniVersion": "0.3.0", "type": "contrail-k8s-cni" }
Events:    <none>
```

It's good practice to confirm that these two networks are now in Contrail before proceeding. From the Contrail UI, select Configure > Networking > Networks > default-domain > k8s-default is shown in Figure 10.2, which focuses on the *left* network.

NOTE If you use the `default` namespace in the YAML file for a network, it will create it in the domain default-domain and project k8s-default.



	Network	Subnets	Tags	Attached Policies	Shared	Admin State
□	k8s-default-pod-network	k8s-pod-ipam	application=k8s	k8s-default-ip-fabric-np k8s-default-pod-service-np 1 more	Disabled	Up
□	k8s-default-service-network	k8s-service-ipam	application=k8s	k8s-default-ip-fabric-np k8s-default-pod-service-np 1 more	Disabled	Up
□	k8s-vn-left-pod-network	10.10.10.0/24	-	Left-right	Disabled	Up

Subnets

Allocation Mode User Defined

Subnet(s)

CIDR	Gateway	Service Address	DNS	DHCP	Allocation Pools
10.10.10.0/24	10.10.10.254	10.10.10.253	Disabled	Enabled	-

Details

Name	k8s-vn-left-pod-network
Display Name	k8s-vn-left-pod-network
UUID	52da641c-1b17-456f-84f8-b67b2a0d233d
Admin State	Up
Shared	Disabled
External	Disabled
SNAT	Disabled
Attached Network Policies	Left-right
Forwarding Mode	L3 only
VxLAN Identifier	Automatic (7)
Allow Transit	Disabled
Mirroring	Disabled
Reverse Path Forwarding	Enabled
Flood Unknown Unicast	Disabled
Multiple Service Chains	Disabled
Host Route(s)	-
DNS Server(s)	-
Ecmp Hashing Fields	-
Provider Network	Disabled
Extended to Physical Router(s)	-
Attached Static Route(s)	-
Attached Routing Policies	-
Floating IP Pool(s)	-
PBB Encapsulation	Disabled
PBB ETree	Disabled
Layer2 Control Word	Disabled
MAC Learning	Disabled
IP Fabric Forwarding	Disabled

Permissions

Owner	None
Owner Permissions	Read, Write, Refer
Global Permissions	-
Shared List	-

	Network	Subnets	Tags	Attached Policies	Shared	Admin State
□	k8s-vn-right-pod-network	10.20.20.0/24	-	Left-right	Disabled	Up

Total: 4 records 50 Records

Figure 10.2 Confirming the Creation of Two Networks

Create Client Pods

Now let's create two Ubuntu Pods, one in each network using the following annotation object:

```
#left-ubuntu-sc.yaml
apiVersion: v1
kind: Pod
metadata:
  name: left-ubuntu-sc
  labels:
    app: webapp-sc
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
        { "name": "vn-left" }]'
spec:
  containers:
    - name: ubuntu-left-pod-sc
      image: contrailk8sdayone/ubuntu
      securityContext:
          privileged: true
          capabilities:
           add:
             - NET_ADMIN


#right-ubuntu-sc.yaml
apiVersion: v1
kind: Pod
metadata:
  name: right-ubuntu-sc
  labels:
    app: webapp-sc
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
        { "name": "vn-right" }]'
spec:
  containers:
    - name: ubuntu-right-pod-sc
      image: contrailk8sdayone/ubuntu
      securityContext:
          privileged: true
          capabilities:
           add:
             - NET_ADMIN

# kubectl create -f right-ubuntu-sc.yaml
# kubectl create -f left-ubuntu-sc.yaml
# kubectl get pod
NAME              READY   STATUS    RESTARTS   AGE
left-ubuntu-sc    1/1     Running   0          25h
right-ubuntu-sc   1/1     Running   0          25h

# kubectl describe pod
Name:               left-ubuntu-sc
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               cent22/10.85.188.17
Start Time:         Thu, 13 Jun 2019 03:40:20 -0400
Labels:             app=webapp-sc
Annotations:        k8s.v1.cni.cncf.io/network-status:
                      [
                          {
                              "ips": "10.10.10.1",
                              "mac": "02:7d:b1:09:00:8d",
                              "name": "vn-left"
                          },
                          {
                              "ips": "10.47.255.249",
                              "mac": "02:7d:99:ff:62:8d",
                              "name": "cluster-wide-default"
                          }
                      ]
                    k8s.v1.cni.cncf.io/networks: [ { "name": "vn-left" }]
Status:             Running
IP:                 10.47.255.249
Containers:
  ubuntu-left-pod-sc:
    Container ID:   docker://2f9a22568d844c68a1c4a45de4a81478958233052e08d4473742827482b244cd
    Image:          contrailk8sdayone/ubuntu
    Image ID:       docker-pullable://contrailk8sdayone/ubuntu@sha256:fa2930cb8f4b766e5b335dfa42de510
ecd30af6433ceada14cdaae8de9065d2a

...<snipped>...

Name:               right-ubuntu-sc
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               cent22/10.85.188.17
Start Time:         Thu, 13 Jun 2019 04:09:18 -0400
Labels:             app=webapp-sc
Annotations:        k8s.v1.cni.cncf.io/network-status:
                      [
                          {
                              "ips": "10.20.20.1",
                              "mac": "02:89:cc:86:48:8d",
                              "name": "vn-right"
                          },
                          {
                              "ips": "10.47.255.252",
                              "mac": "02:89:b0:8e:98:8d",
                              "name": "cluster-wide-default"
                          }
                      ]
                    k8s.v1.cni.cncf.io/networks: [ { "name": "vn-right" }]
Status:             Running
IP:                 10.47.255.252
Containers:
  ubuntu-right-pod-sc:
    Container ID:   docker://4e0b6fa085905be984517a11c3774517d01f481fa43aadd76a633ef15c58cbfe
    Image:          contrailk8sdayone/ubuntu
    Image ID:       docker-pullable://contrailk8sdayone/ubuntu@sha256:fa2930cb8f4b766e5b335dfa42de51
0ecd30af6433ceada14cdaae8de9065d2a

 ...<snipped>...
```

Create cSRX Pod

Now create a Juniper cSRX container that has one interface on the left network and one interface on the right network, using this YAML file:

```
# cat csrx1-sc.yaml
apiVersion: v1
kind: Pod
metadata:
  name: csrx1-sc
  labels:
    app: webapp-sc
  annotations:
   k8s.v1.cni.cncf.io/networks: '[
       { "name": "vn-left" },
       { "name": "vn-right" }
   ]'
spec:
  containers:
  - name: csrx1-sc
    image: contrailk8sdayone/csrx
    ports:
    - containerPort: 22
    imagePullPolicy: Never
    stdin: true
    tty: true
    securityContext:
      privileged: true

# kubectl create -f csrx1-sc.yaml
```

Confirm that the interface placement is in the correct network:

```
# kubectl describe pod csrx1-sc
Name:               csrx1-sc
Namespace:          default
Priority:           0
```

```
PriorityClassName:  <none>
Node:               cent22/10.85.188.17
Start Time:         Thu, 13 Jun 2019 03:40:31 -0400
Labels:             app=webapp-sc
Annotations:        k8s.v1.cni.cncf.io/network-status:
                      [
                          {
                              "ips": "10.10.10.2",
                              "mac": "02:84:71:f4:f2:8d",
                              "name": "vn-left"
                          },
                          {
                              "ips": "10.20.20.2",
                              "mac": "02:84:8b:4c:18:8d",
                              "name": "vn-right"
                          },
                          {
                              "ips": "10.47.255.248",
                              "mac": "02:84:59:7e:54:8d",
                              "name": "cluster-wide-default"
                          }
                      ]
                    k8s.v1.cni.cncf.io/networks: [ { "name": "vn-left" }, { "name": "vn-right" } ]
Status:             Running
IP:                 10.47.255.248
Containers:
  csrx1-sc:
    Container ID:   docker://82b7605172d937895269d76850d083b6dc6e278e41cb45b4cb8cee21283e4f17
    Image:          contrailk8sdayone/csrx
    Image ID:       docker://sha256:329e805012bdf081f4a15322f994e5e3116b31c90f108a19123cf52710c7617e

...<snipped>...
```

NOTE Each container has one interface belonging to `cluster-wide-default` network regardless of the use of the annotations object because the annotations object above creates, and puts one extra interface in, a specific network.

Verify PodIP

To verify the podIP, log in to the left pord, right Pod and the cSRX to confirm the IP/MAC addresses:

```
# kubectl exec -it left-ubuntu-sc bash
root@left-ubuntu-sc:/# 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
13: eth0@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:7d:99:ff:62:8d brd ff:ff:ff:ff:ff:ff
    inet 10.47.255.249/12 scope global eth0
       valid_lft forever preferred_lft forever
15: eth1@if16: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:7d:b1:09:00:8d brd ff:ff:ff:ff:ff:ff
    inet 10.10.10.1/24 scope global eth1
       valid_lft forever preferred_lft forever
```

```
# kubectl exec -it right-ubuntu-sc bash
root@right-ubuntu-sc:/# 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
23: eth0@if24: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:89:b0:8e:98:8d brd ff:ff:ff:ff:ff:ff
    inet 10.47.255.252/12 scope global eth0
       valid_lft forever preferred_lft forever
25: eth1@if26: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:89:cc:86:48:8d brd ff:ff:ff:ff:ff:ff
    inet 10.20.20.1/24 scope global eth1
       valid_lft forever preferred_lft forever


# kubectl exec -it csrx1-sc cli
root@csrx1-sc>
root@csrx1-sc> show interfaces
Physical interface: ge-0/0/1, Enabled, Physical link is Up
  Interface index: 100
  Link-level type: Ethernet, MTU: 1514
  Current address: 02:84:71:f4:f2:8d, Hardware address: 02:84:71:f4:f2:8d

Physical interface: ge-0/0/0, Enabled, Physical link is Up
  Interface index: 200
  Link-level type: Ethernet, MTU: 1514
  Current address: 02:84:8b:4c:18:8d, Hardware address: 02:84:8b:4c:18:8d
```

NOTE Unlike other pods the cSRX didn't acquire IP with DHCP, and it starts with the factory default configuration hence it needs to be configured.

NOTE By default, cSRX eth0 is visible only from the shell and used for management. When attaching networks, the first attached network is mapped to eth1, which is GE-0/0/1, and the second attached is mapped to eth2, which is GE-0/0/0.

Configure cSRX IP

Configure this basic setup on the cSRX. To assign the correct IP address, use the MAC/IP address mapping from the `kubectl describe pod` command output as well as to configure the default security policy to allow everything for now:

```
set interfaces ge-0/0/1 unit 0 family inet address 10.10.10.2/24
set interfaces ge-0/0/0 unit 0 family inet address 10.20.20.2/24

set security zones security-zone trust interfaces ge-0/0/0
set security zones security-zone untrust interfaces ge-0/0/1
set security policies default-policy permit-all
commit
```

Verify the IP address assigned on the cSRX:

```
root@csrx1-sc> show interfaces
Physical interface: ge-0/0/1, Enabled, Physical link is Up
  Interface index: 100
  Link-level type: Ethernet, MTU: 1514
  Current address: 02:84:71:f4:f2:8d, Hardware address: 02:84:71:f4:f2:8d

  Logical interface ge-0/0/1.0 (Index 100)
    Flags: Encapsulation: ENET2
    Protocol inet
        Destination: 10.10.10.0/24, Local: 10.10.10.2

Physical interface: ge-0/0/0, Enabled, Physical link is Up
  Interface index: 200
  Link-level type: Ethernet, MTU: 1514
  Current address: 02:84:8b:4c:18:8d, Hardware address: 02:84:8b:4c:18:8d

  Logical interface ge-0/0/0.0 (Index 200)
    Flags: Encapsulation: ENET2
    Protocol inet
        Destination: 10.20.20.0/24, Local: 10.20.20.2
```

A ping test on the left pod would fail as there is no route:

```
root@left-ubuntu-sc:/# ping 10.20.20.1
PING 10.20.20.1 (10.20.20.1) 56(84) bytes of data.
^C
--- 10.20.20.1 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 1999ms

root@left-ubuntu-sc:/# ip r
default via 10.47.255.254 dev eth0
10.10.10.0/24 dev eth1  proto kernel  scope link  src 10.10.10.1
10.32.0.0/12 dev eth0  proto kernel  scope link  src 10.47.255.249
```

Add a static route to the left and right pods and then try to ping again:

```
root@left-ubuntu-sc:/# ip r add 10.20.20.0/24 via 10.10.10.2

root@right-ubuntu-sc:/# ip r add 10.10.10.0/24 via 10.20.20.2

root@left-ubuntu-sc:/# ping 10.20.20.1
PING 10.20.20.1 (10.20.20.1) 56(84) bytes of data.
^C
--- 10.20.20.1 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 2999ms
```

The ping still failed, as we didn't create the service chaining, which will also take care of the routing. Let's see what happened to our packets:

```
root@csrx1-sc# run show security flow session
Total sessions: 0
```

There's no session on the cSRX. To troubleshoot the ping issue, log in to the compute node `cent22` that hosts this container to dump the traffic using TShark and check the routing. To get the interface linking the containers:

```
[root@cent22 ~]# vif -l
Vrouter Interface Table

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

...<snipped>...

vif0/3      OS: tapeth0-89a4e2
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:10.47.255.252
            Vrf:3 Mcast Vrf:3 Flags:PL3DEr QOS:-1 Ref:6
            RX packets:10760  bytes:452800 errors:0
            TX packets:14239  bytes:598366 errors:0
            Drops:10744

vif0/4      OS: tapeth1-89a4e2
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:10.20.20.1
            Vrf:5 Mcast Vrf:5 Flags:PL3DEr QOS:-1 Ref:6
            RX packets:13002  bytes:867603 errors:0
            TX packets:16435  bytes:1046981 errors:0
            Drops:10805

vif0/5      OS: tapeth0-7d8e06
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:10.47.255.249
            Vrf:3 Mcast Vrf:3 Flags:PL3DEr QOS:-1 Ref:6
            RX packets:10933  bytes:459186 errors:0
            TX packets:14536  bytes:610512 errors:0
            Drops:10933

vif0/6      OS: tapeth1-7d8e06
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:10.10.10.1
            Vrf:6 Mcast Vrf:6 Flags:PL3DEr QOS:-1 Ref:6
            RX packets:12625  bytes:1102433 errors:0
            TX packets:15651  bytes:810689 errors:0
            Drops:10957

vif0/7      OS: tapeth0-844f1c
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:10.47.255.248
            Vrf:3 Mcast Vrf:3 Flags:PL3DEr QOS:-1 Ref:6
            RX packets:20996  bytes:1230688 errors:0
            TX packets:27205  bytes:1142610 errors:0
            Drops:21226

vif0/8      OS: tapeth1-844f1c
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:10.10.10.2
            Vrf:6 Mcast Vrf:6 Flags:PL3DEr QOS:-1 Ref:6
            RX packets:13908  bytes:742243 errors:0
            TX packets:29023  bytes:1790589 errors:0
            Drops:10514

vif0/9      OS: tapeth2-844f1c
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:10.20.20.2
            Vrf:5 Mcast Vrf:5 Flags:PL3DEr QOS:-1 Ref:6
            RX packets:16590  bytes:1053659 errors:0
            TX packets:31321  bytes:1635153 errors:0
            Drops:10421

...<snipped>...
```

Note that `vif0/3` and `vif0/4` are bound with the right pod and both linked to `tapeth0-89a4e2` and `tapeth1-89a4e2` respectively. The same goes for the left pod for `Vif0/5` and `vif0/6` while `vif0/7`, `vif 0/8`, and `vif0/9` are bound with the cSRX1. From this you can also see the number of the packet/bytes that hit the interface, as well as the VRF. VRF 3 is for the default-cluster-network, while VRF 6 is for the left network and VRF 5 is for the right network. In Figure 10.3 you can see the interface mapping from all the perspectives (container, Linux , vr-agent).

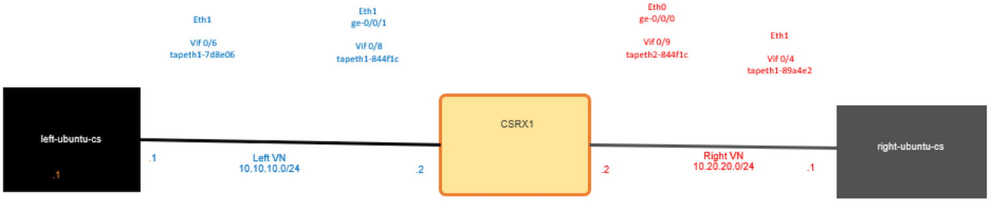


Figure 10.3 Interface Mapping

Let's try to ping from the left pod to the right pod again, and use TShark on the tap interface for the right pod for further inspection:

```
[root@cent22 ~]# tshark -i tapeth1-89a4e2
Running as user "root" and group "root". This could be dangerous.
Capturing on 'tapeth1-89a4e2'
  1 0.000000000 IETF-VRRP-VRID_00 -> 02:89:cc:86:48:8d ARP 42 Gratuitous ARP for 10.20.20.254 (Request)
  2 0.000037656 IETF-VRRP-VRID_00 -> 02:89:cc:86:48:8d ARP 42 Gratuitous ARP for 10.20.20.253 (Request)
  3 1.379993896 IETF-VRRP-VRID_00 -> 02:89:cc:86:48:8d ARP 42 Who has 10.20.20.1?  Tell 10.20.20.253
```

Looks like the ping isn't reaching the right pod at all, let's check the cSRX's left network tap interface:

```
[root@cent22 ~]# tshark -i tapeth1-844f1c
Running as user "root" and group "root". This could be dangerous.
Capturing on 'tapeth1-844f1c'
  1 0.000000000 IETF-VRRP-VRID_00 -> 02:84:71:f4:f2:8d ARP 42 Who has 0.255.255.252?  Tell 0.0.0.0
  2 0.201392098   10.10.10.1 -> 10.20.20.1   ICMP 98 Echo (ping) request  id=0x020a, seq=410/39425,
ttl=63
```

```
  3 0.201549430   10.10.10.2 -> 10.10.10.1   ICMP 70 Destination unreachable (Port unreachable)
  4 1.201444156   10.10.10.1 -> 10.20.20.1   ICMP 98 Echo (ping) request  id=0x020a, seq=411/39681, 
ttl=63
  5 1.201600074   10.10.10.2 -> 10.10.10.1   ICMP 70 Destination unreachable (Port unreachable)
  6 1.394074095 IETF-VRRP-VRID_00 -> 02:84:71:f4:f2:8d ARP 42 Gratuitous ARP for 10.10.10.254 (Request)
  7 1.394108344 IETF-VRRP-VRID_00 -> 02:84:71:f4:f2:8d ARP 42 Gratuitous ARP for 10.10.10.253 (Request)
  8 2.201462515   10.10.10.1 -> 10.20.20.1   ICMP 98 Echo (ping) request  id=0x020a, seq=412/39937, 
ttl=63
```

We can see the packet but there is nothing in the cSRX security prospective to drop this packet

Check the routing table of the left network VRF by logging to the `vrouter_vrouter-agent_1` container in the compute node:

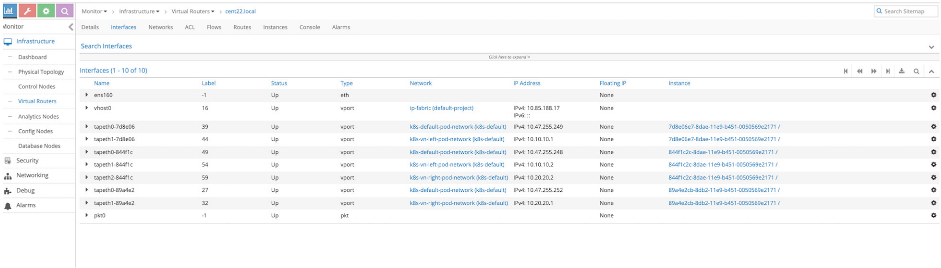
```
[root@cent22 ~]# docker ps | grep vrouter
9a737df53abe        ci-repo.englab.juniper.net:5000/contrail-vrouter-agent:master-latest   "/
entrypoint.sh /usr…"   2 weeks ago         Up 47 hours                             vrouter_vrouter-
agent_1
e25f1467403d        ci-repo.englab.juniper.net:5000/contrail-nodemgr:master-latest         "/
entrypoint.sh /bin…"   2 weeks ago         Up 47 hours                             vrouter_nodemgr_1

[root@cent22 ~]# docker exec -it vrouter_vrouter-agent_1 bash
(vrouter-agent)[root@cent22 /]$
(vrouter-agent)[root@cent22 /]$ rt --dump 6 | grep 10.20.20.
(vrouter-agent)[root@cent22 /]$
```

Note that 6 is the routing table VRF of the left network; the same would go for the right network VRF routing table but there is a missing route:

```
(vrouter-agent)[root@cent22 /]$ rt --dump 5 | grep 10.10.10.
(vrouter-agent)[root@cent22 /]$
```

So, even if all the pods are hosted on the same compute nodes, they can't reach each other. And if these pods are hosted on some different compute nodes then you have a bigger problem to solve. Service chaining isn't just about adjusting the routes on the containers but also about exchanging routes between the vRouter-agent between the compute nodes regardless of the location of the pod (as well as adjusting that automatically if the pod moved to another compute node). Before labbing service-chaining let's address an important concern for network administrators who are not fans of this kind of CLI troubleshooting… you can do the same troubleshooting using the Contrail Controller GUI.

From the Contrail Controller UI, select Monitor > Infrastructure > Virtual Routers and then select the node that hosts the pod, in our case cent22.local, as shown in the next screen capture, Figure 10.4.



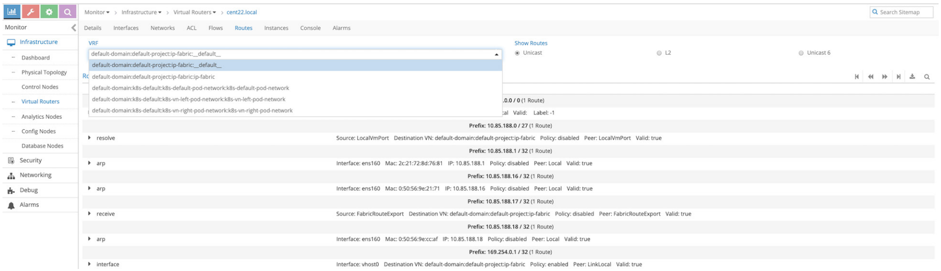
Interfaces (1 - 10 of 10)

Name	Label	Status	Type	Network	IP Address	Floating IP	Instance
ens160	-1	Up	eth			None	
vhost0	16	Up	vport	ip-fabric (default-project)	IPv4: 10.85.188.17 IPv6: ::	None	
tapeth0-7d8e06	39	Up	vport	k8s-default-pod-network (k8s-default)	IPv4: 10.47.255.249	None	7d8e06e7-8dae-11e9-b451-0050569e2171 /
tapeth1-7d8e06	44	Up	vport	k8s-vn-left-pod-network (k8s-default)	IPv4: 10.10.10.1	None	7d8e06e7-8dae-11e9-b451-0050569e2171 /
tapeth0-844f1c	49	Up	vport	k8s-default-pod-network (k8s-default)	IPv4: 10.47.255.248	None	844f1c2c-8dae-11e9-b451-0050569e2171 /
tapeth1-844f1c	54	Up	vport	k8s-vn-left-pod-network (k8s-default)	IPv4: 10.10.10.2	None	844f1c2c-8dae-11e9-b451-0050569e2171 /
tapeth2-844f1c	59	Up	vport	k8s-vn-right-pod-network (k8s-default)	IPv4: 10.20.20.2	None	844f1c2c-8dae-11e9-b451-0050569e2171 /
tapeth0-89a4e2	27	Up	vport	k8s-default-pod-network (k8s-default)	IPv4: 10.47.255.252	None	89a4e2cb-8db2-11e9-b451-0050569e2171 /
tapeth1-89a4e2	32	Up	vport	k8s-vn-right-pod-network (k8s-default)	IPv4: 10.20.20.1	None	89a4e2cb-8db2-11e9-b451-0050569e2171 /
pkt0	-1	Up	pkt			None	

Figure 10.4 Contrail Controller GUI in Action

Figure 10.4 shows the Interface tab, which is equivalent to running the `vif -l` command on the `vrouter_vrouter-agent-1` container, but it shows even more information. Notice the mapping between the instance ID and the tap interface naming, where the first six characters of the instance ID are always reflected in the tap interface naming.

We are GUI cowboys. Let's check the routing tables of each VRF by moving to the Routes tab and selecting the VRF you want to see, as in Figure 10.5.



VRF: default-domain:default-project:ip-fabric:__default__

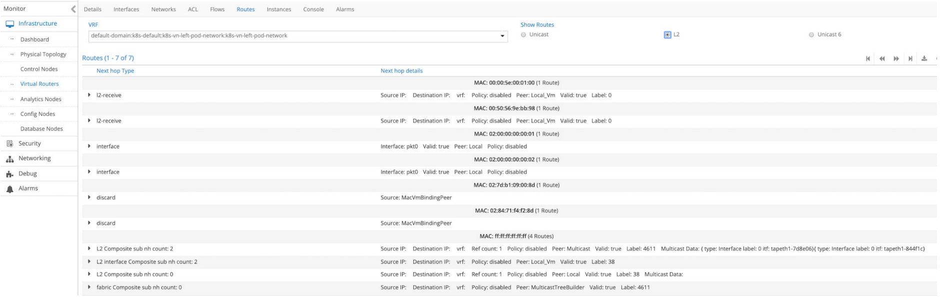
- default-domain:default-project:ip-fabric:__default__
- default-domain:default-project:ip-fabric:ip-fabric
- default-domain:k8s-default:k8s-default-pod-network:k8s-default-pod-network
- default-domain:k8s-default:k8s-vn-left-pod-network:k8s-vn-left-pod-network
- default-domain:k8s-default:k8s-vn-right-pod-network:k8s-vn-right-pod-network

Show Routes: ● Unicast ○ L2 ○ Unicast 6

```
.0.0 / 0 (1 Route)
cal   Valid:   Label: -1
Prefix: 10.85.188.0 / 27 (1 Route)
resolve     Source: LocalVmPort   Destination VN: default-domain:default-project:ip-fabric   Policy: disabled   Peer: LocalVmPort   Valid: true
Prefix: 10.85.188.1 / 32 (1 Route)
arp         Interface: ens160   Mac: 2c:21:72:8d:76:81   IP: 10.85.188.1   Policy: disabled   Peer: Local   Valid: true
Prefix: 10.85.188.16 / 32 (1 Route)
arp         Interface: ens160   Mac: 0:50:56:9e:21:71   IP: 10.85.188.16   Policy: disabled   Peer: Local   Valid: true
Prefix: 10.85.188.17 / 32 (1 Route)
receive     Source: FabricRouteExport   Destination VN: default-domain:default-project:ip-fabric   Policy: disabled   Peer: FabricRouteExport   Valid: true
Prefix: 10.85.188.18 / 32 (1 Route)
arp         Interface: ens160   Mac: 0:50:56:9e:cc:af   IP: 10.85.188.18   Policy: disabled   Peer: Local   Valid: true
Prefix: 169.254.0.1 / 32 (1 Route)
interface   Interface: vhost0   Destination VN: default-domain:default-project:ip-fabric   Policy: enabled   Peer: LinkLocal   Valid: true
```

Figure 10.5 Checking the Routing Tables of each VRF

Select the left network. The name is longer because it includes the domain (and project). You can confirm there is no 10.20.20.0/24 prefix from the right network. You can also check the MAC address learned in the left network by selecting L2, the GUI equivalent to the `rt --dump 6 --family bridge` command.



VRF: default-domain:k8s-default:k8s-vn-left-pod-network:k8s-vn-left-pod-network

Show Routes: ○ Unicast ● L2 ○ Unicast 6

Routes (1 - 7 of 7)

Next hop Type	Next hop details
	MAC: 00:00:5e:00:01:00 (1 Route)
l2-receive	Source IP: Destination IP: vrf: Policy: disabled Peer: Local_Vm Valid: true Label: 0
	MAC: 00:50:56:9e:bb:98 (1 Route)
l2-receive	Source IP: Destination IP: vrf: Policy: disabled Peer: Local_Vm Valid: true Label: 0
	MAC: 02:00:00:00:00:01 (1 Route)
interface	Interface: pkt0 Valid: true Peer: Local Policy: disabled
	MAC: 02:00:00:00:00:02 (1 Route)
interface	Interface: pkt0 Valid: true Peer: Local Policy: disabled
	MAC: 02:7d:b1:09:00:8d (1 Route)
discard	Source: MacVmBindingPeer
	MAC: 02:84:71:f4:f2:8d (1 Route)
discard	Source: MacVmBindingPeer
	MAC: ff:ff:ff:ff:ff:ff (4 Routes)
L2 Composite sub nh count: 2	Source IP: Destination IP: vrf: Ref count: 1 Policy: disabled Peer: Multicast Valid: true Label: 4611 Multicast Data: { type: Interface label: 0 itf: tapeth1-7d8e06}{ type: Interface label: 0 itf: tapeth1-844f1c}
L2 interface Composite sub nh count: 2	Source IP: Destination IP: vrf: Policy: disabled Peer: Local_Vm Valid: true Label: 38
L2 Composite sub nh count: 0	Source IP: Destination IP: vrf: Ref count: 1 Policy: disabled Peer: Local Valid: true Label: 38 Multicast Data:
fabric Composite sub nh count: 0	Source IP: Destination IP: vrf: Policy: disabled Peer: MulticastTreeBuilder Valid: true Label: 4611

Figure 10.6 Getting MAC Address

Service Chaining

Now let's utilize the cSRX to service chaining using the Contrail Command GUI.

Service chaining consists of four steps that need to be completed in order:

1. Create a service template;
2. Create a service instance based on the service template just completed;
3. Create a network policy and select the service instance you created before;
4. Apply this network policy onto the network.

NOTE Since Contrail Command GUI is the best solution to provide a single point of management for all environments, we will use it to build service changing. You can still use the normal Contrail controller GUI to build service chaining, too.

First let's log in to Contrail Command GUI (in our setup https://10.85.188.16:9091/) as shown next in Figure 10.7, and then select Service > Catalog > Create as shown in Figure 10.8.

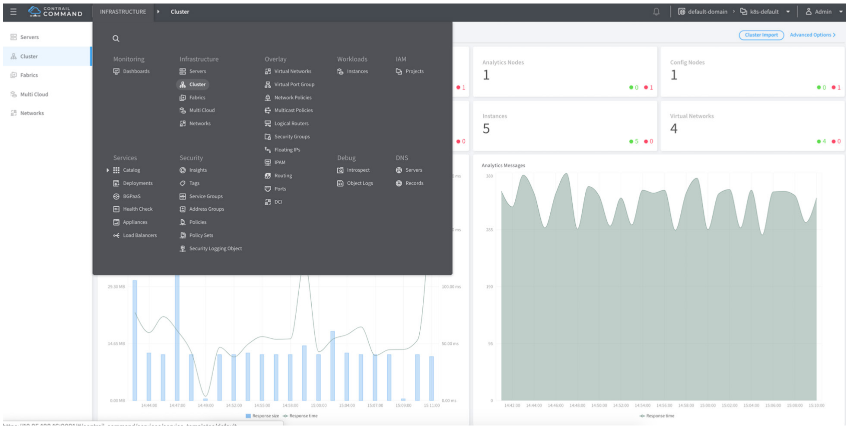


Figure 10.7 Log in to Contrail Command

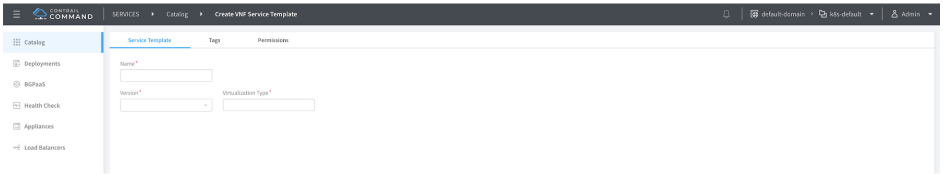
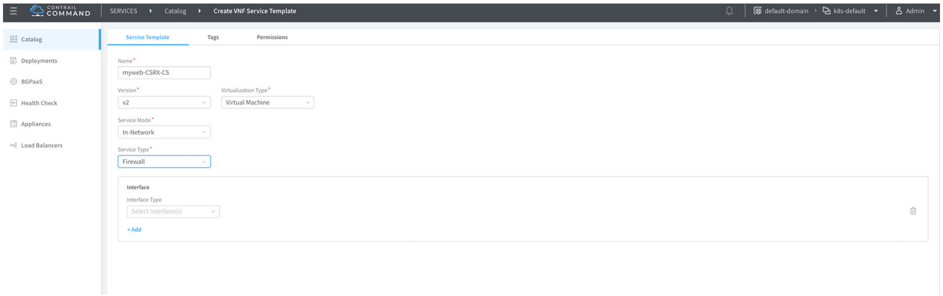


Figure 10.8 Create a New Service

Insert a name of a services template, here *myweb-cSRX-CS*, then chose *v2* and *virtual machine* for service modes. Choose *In-network* and *Firewall* as service types as shown in Figure 10.9.



Name*: myweb-CSRX-CS

Version*: v2

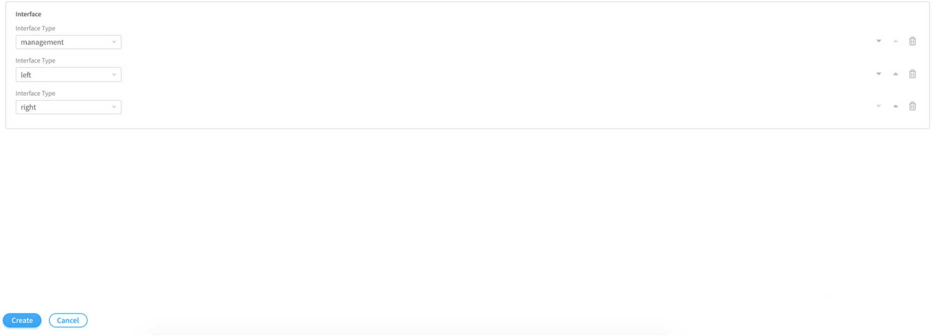
Virtualization Type*: Virtual Machine

Service Mode*: In-Network

Service Type*: Firewall

Figure 10.9 Choosing Service Types

Next select *Management*, *Left* and *Right*, and then click *Create*.



Interface

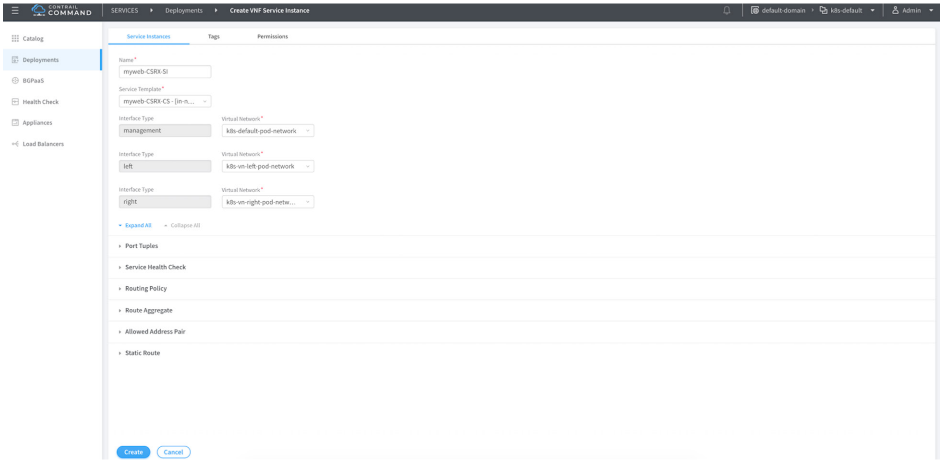
Interface Type: management

Interface Type: left

Interface Type: right

Figure 10.10 Create Service

Now, select Deployment and click on the Create button to create the service instances as shown next in Figure 10.11.



Name*: myweb-CSRX-SI

Service Template*: myweb-CSRX-CS - [in-n...

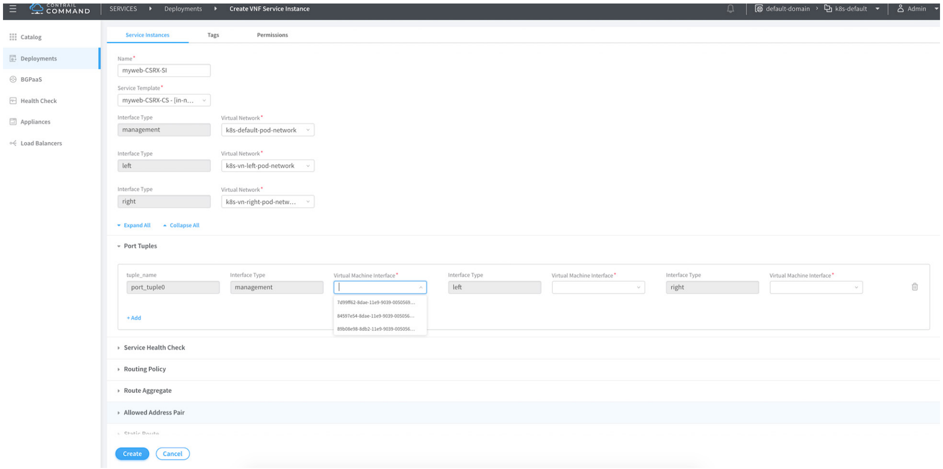
Interface Type: management — Virtual Network*: k8s-default-pod-network

Interface Type: left — Virtual Network*: k8s-vn-left-pod-network

Interface Type: right — Virtual Network*: k8s-vn-right-pod-netw...

Figure 10.11 Deploy Service Instance

Name this service instance, then select from the drop-down menu the name of the template you created before you chose the proper network from the prospective of the cSRX being the instance (container in that case) that will do the service chaining. Click on the port tuples to expand it as shown in Figure 10.12.



Name* myweb-CSRX-SI

Service Template* myweb-CSRX-CS - [in-n...

Interface Type	Virtual Network*
management	k8s-default-pod-network
left	k8s-vn-left-pod-network
right	k8s-vn-right-pod-netw...

Port Tuples

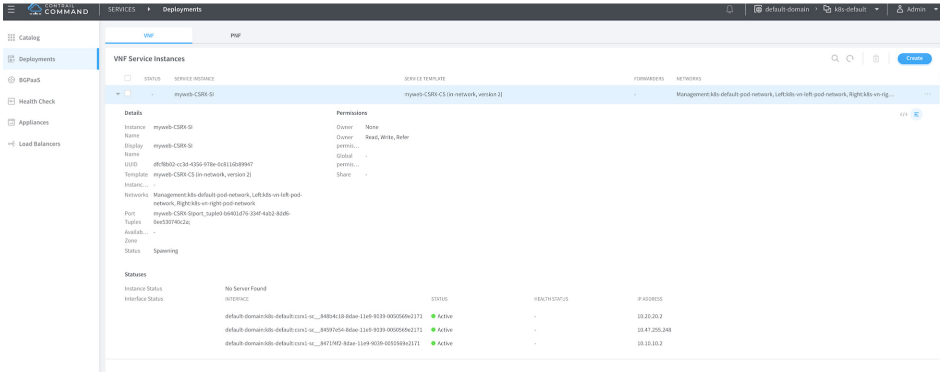
tuple_name	Interface Type	Virtual Machine Interface*	Interface Type	Virtual Machine Interface*	Interface Type	Virtual Machine Interface*
port_tuple0	management	7d99ff62-8dae-11e9-9039-0050569... 84597e54-8dae-11e9-9039-005056... 89b08e98-8db2-11e9-9039-005056...	left		right	

Figure 10.12 Expanding the Port Tuples

Then, for each of the three interfaces bind one interface of the cSRX, then click Create.

NOTE The name of the VM interface isn't shown in the drop-down menu, instead it's the instance ID. You can identify that from the tap interface name as we mentioned before. In other words, all you have to know is the first six characters for any interface belonging to that container. All the interfaces in a given instance (VM or container) share the same first characters.

Before proceeding, make sure the statuses of the three interfaces are up and they are showing the correct IP address of the cSRX instance as shown in Figure 10.13.



VNF Service Instances

STATUS	SERVICE INSTANCE	SERVICE TEMPLATE	FORWARDERS	NETWORKS
-	myweb-CSRX-SI	myweb-CSRX-CS (in-network, version 2)	-	Management:k8s-default-pod-network, Left:k8s-vn-left-pod-network, Right:k8s-vn-rig...

Details

Instance Name	myweb-CSRX-SI
Display Name	myweb-CSRX-SI
UUID	dfcf8b02-cc3d-4356-978e-0c8116b89947
Template	myweb-CSRX-CS (in-network, version 2)
Instanc...	-
Networks	Management:k8s-default-pod-network, Left:k8s-vn-left-pod-network, Right:k8s-vn-right-pod-network
Port Tuples	myweb-CSRX-SIport_tuple0-b6401d76-334f-4ab2-8dd6-0ee530740c2a;
Availab... Zone	-
Status	Spawning

Permissions

Owner	None
Owner permis...	Read, Write, Refer
Global permis...	-
Share	-

Statuses

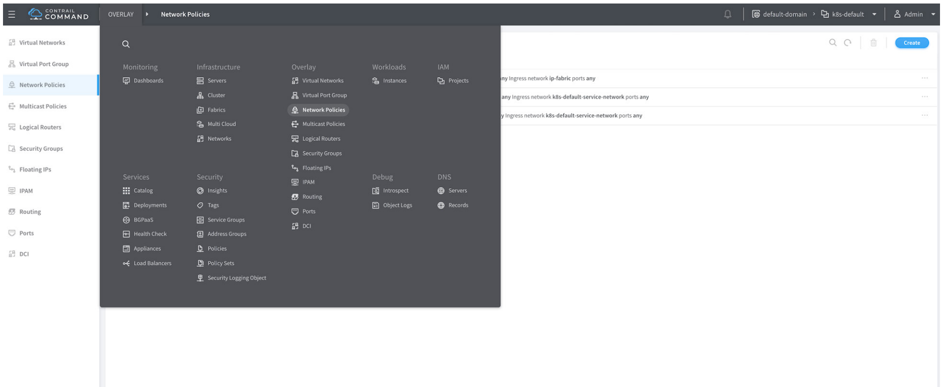
Instance Status No Server Found

Interface Status

INTERFACE	STATUS	HEALTH STATUS	IP ADDRESS
default-domain:k8s-default:csrx1-sc__848b4c18-8dae-11e9-9039-0050569e2171	● Active	-	10.20.20.2
default-domain:k8s-default:csrx1-sc__84597e54-8dae-11e9-9039-0050569e2171	● Active	-	10.47.255.248
default-domain:k8s-default:csrx1-sc__8471f4f2-8dae-11e9-9039-0050569e2171	● Active	-	10.10.10.2

Figure 10.13 All Interfaces Up and Running

To create the network policy go to Overlay > Network Policies > Create as in Figure 10.14.



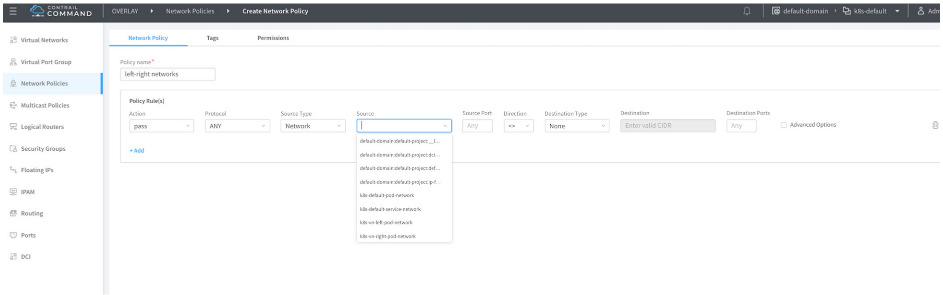
ny Ingress network **ip-fabric** ports **any**

any Ingress network **k8s-default-service-network** ports **any**

y Ingress network **k8s-default-service-network** ports **any**

Figure 10.14 *Create Network Policy*

Name your network policy, then in the first rule add *left network* as the source network and *right network* as the destination with the action of pass.



Policy name* left-right networks

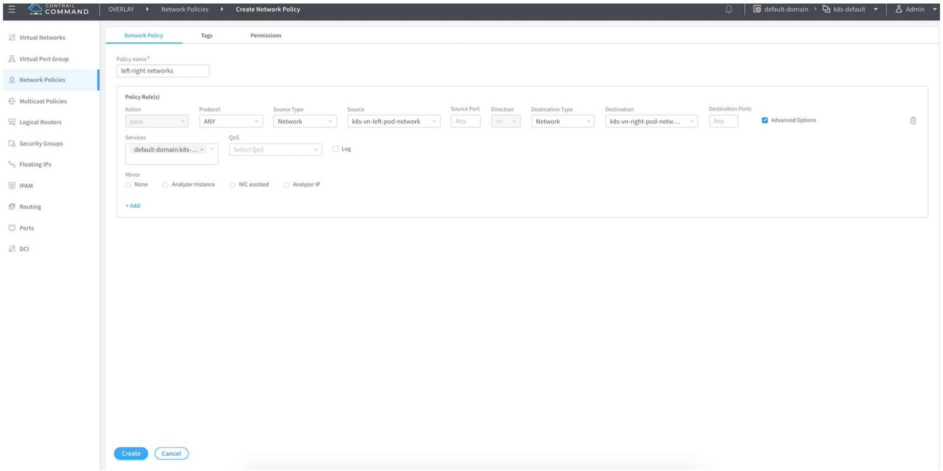
Policy Rule(s)

Action	Protocol	Source Type	Source	Source Port	Direction	Destination Type	Destination	Destination Ports	
pass	ANY	Network		Any	<>	None	Enter valid CIDR	Any	□ Advanced Options

default-domain:default-project:__l...
default-domain:default-project:dci...
default-domain:default-project:def...
default-domain:default-project:ip-f...
k8s-default-pod-network
k8s-default-service-network
k8s-vn-left-pod-network
k8s-vn-right-pod-network

Figure 10.15 *Source and Destination*

Select the advanced option and attach the service instance to the one you created before, then click the Create button.



Policy name* left-right networks

Policy Rule(s)

Action	Protocol	Source Type	Source	Source Port	Direction	Destination Type	Destination	Destination Ports	
pass	ANY	Network	k8s-vn-left-pod-network	Any	<>	Network	k8s-vn-right-pod-netw...	Any	☑ Advanced Options

Services: default-domain:k8s-... ×

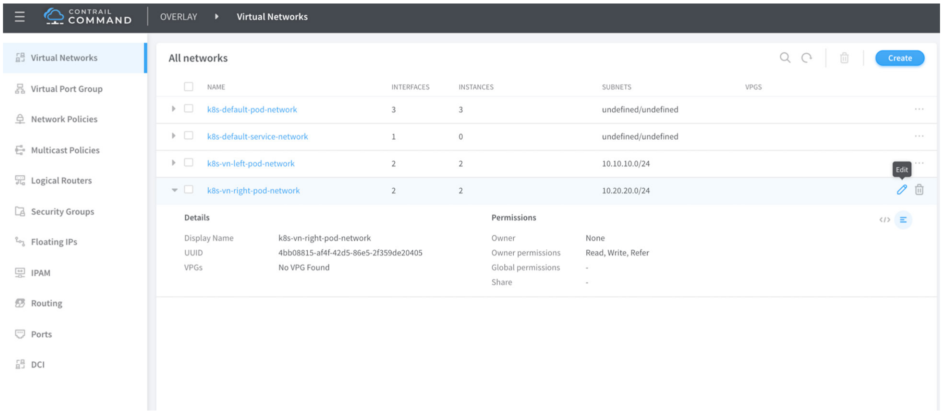
QoS: Select QoS

□ Log

Mirror: ○ None ○ Analyzer Instance ○ NIC assisted ○ Analyzer IP

Figure 10.16 *Attaching the Service Instance*

To attach this network policy to the network click on Virtual Network in the left-most column and select the *left network* and edit.



All networks

NAME	INTERFACES	INSTANCES	SUBNETS	VPGS
k8s-default-pod-network	3	3	undefined/undefined	
k8s-default-service-network	1	0	undefined/undefined	
k8s-vn-left-pod-network	2	2	10.10.10.0/24	
k8s-vn-right-pod-network	2	2	10.20.20.0/24	

Details

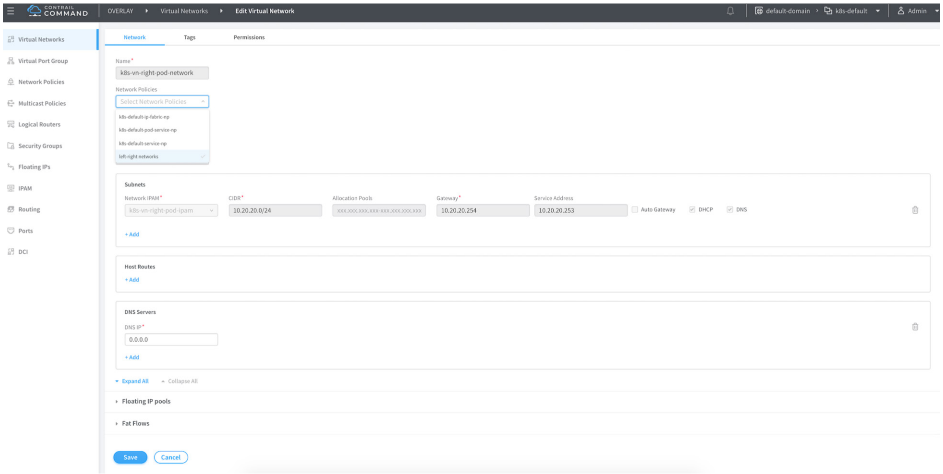
Display Name k8s-vn-right-pod-network
UUID 4bb08815-af4f-42d5-86e5-2f359de20405
VPGs No VPG Found

Permissions

Owner None
Owner permissions Read, Write, Refer
Global permissions -
Share -

Figure 10.17 *Attach the Policy to the Network*

In Network Policies select the network policy you just created from the drop-down menu list, and then click Save. Do the same for the *right network*.



CONTRAIL COMMAND | OVERLAY > Virtual Networks > **Edit Virtual Network** | default-domain > k8s-default | Admin

Virtual Networks
Virtual Port Group
Network Policies
Multicast Policies
Logical Routers
Security Groups
Floating IPs
IPAM
Routing
Ports
DCI

Network | Tags | Permissions

Name*: k8s-vn-right-pod-network

Network Policies: Select Network Policies
- k8s-default-ip-fabric-np
- k8s-default-pod-service-np
- k8s-default-service-np
- left-right networks ✓

Subnets

Network IPAM*	CIDR*	Allocation Pools	Gateway*	Service Address			
k8s-vn-right-pod-ipam	10.20.20.0/24	xxx.xxx.xxx.xxx-xxx.xxx.xxx.xxx	10.20.20.254	10.20.20.253	□ Auto Gateway	☑ DHCP	☑ DNS

+ Add

Host Routes

+ Add

DNS Servers

DNS IP*: 0.0.0.0

+ Add

Expand All   Collapse All

Floating IP pools

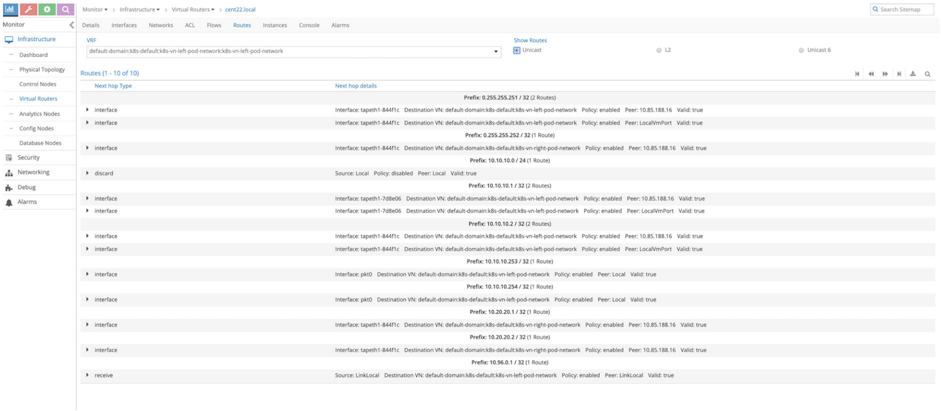
Fat Flows

Save   Cancel

Figure 10.18 Save the Network Policy

Verify Service Chaining

Now let's verify the effect of this service changing on routing. From the Contrail Controller module control node (http://10.85.188.16:8143 in our setup), select Monitor > Infrastructure > Virtual Router then select the node that hosts the pod , in our case `Cent22.local`, then select the Routes tab and select the left VRF.



Monitor > Infrastructure > Virtual Routers > cent22.local

Monitor: Infrastructure (Dashboard, Physical Topology, Control Nodes, Virtual Routers, Analytics Nodes, Config Nodes, Database Nodes), Security, Networking, Debug, Alarms

Details | Interfaces | Networks | ACL | Flows | Routes | Instances | Console | Alarms

VRF: default-domain:k8s-default:k8s-vn-left-pod-network:k8s-vn-left-pod-network

Show Routes: ● Unicast ○ L2 ○ Unicast 6

Routes (1 - 10 of 10)

Next hop Type	Next hop details
	Prefix: 0.255.255.251 / 32 (2 Routes)
interface	Interface: tapeth1-844f1c Destination VN: default-domain:k8s-default:k8s-vn-left-pod-network Policy: enabled Peer: 10.85.188.16 Valid: true
interface	Interface: tapeth1-844f1c Destination VN: default-domain:k8s-default:k8s-vn-left-pod-network Policy: enabled Peer: LocalVmPort Valid: true
	Prefix: 0.255.255.252 / 32 (1 Route)
interface	Interface: tapeth1-844f1c Destination VN: default-domain:k8s-default:k8s-vn-right-pod-network Policy: enabled Peer: 10.85.188.16 Valid: true
	Prefix: 10.10.10.0 / 24 (1 Route)
discard	Source: Local Policy: disabled Peer: Local Valid: true
	Prefix: 10.10.10.1 / 32 (2 Routes)
interface	Interface: tapeth1-7d8e06 Destination VN: default-domain:k8s-default:k8s-vn-left-pod-network Policy: enabled Peer: 10.85.188.16 Valid: true
interface	Interface: tapeth1-7d8e06 Destination VN: default-domain:k8s-default:k8s-vn-left-pod-network Policy: enabled Peer: LocalVmPort Valid: true
	Prefix: 10.10.10.2 / 32 (2 Routes)
interface	Interface: tapeth1-844f1c Destination VN: default-domain:k8s-default:k8s-vn-left-pod-network Policy: enabled Peer: 10.85.188.16 Valid: true
interface	Interface: tapeth1-844f1c Destination VN: default-domain:k8s-default:k8s-vn-left-pod-network Policy: enabled Peer: LocalVmPort Valid: true
	Prefix: 10.10.10.253 / 32 (1 Route)
interface	Interface: pkt0 Destination VN: default-domain:k8s-default:k8s-vn-left-pod-network Policy: enabled Peer: Local Valid: true
	Prefix: 10.10.10.254 / 32 (1 Route)
interface	Interface: pkt0 Destination VN: default-domain:k8s-default:k8s-vn-left-pod-network Policy: enabled Peer: Local Valid: true
	Prefix: 10.20.20.1 / 32 (1 Route)
interface	Interface: tapeth1-844f1c Destination VN: default-domain:k8s-default:k8s-vn-right-pod-network Policy: enabled Peer: 10.85.188.16 Valid: true
	Prefix: 10.20.20.2 / 32 (1 Route)
interface	Interface: tapeth1-844f1c Destination VN: default-domain:k8s-default:k8s-vn-right-pod-network Policy: enabled Peer: 10.85.188.16 Valid: true
	Prefix: 10.96.0.1 / 32 (1 Route)
receive	Source: LinkLocal Destination VN: default-domain:k8s-default:k8s-vn-left-pod-network Policy: enabled Peer: LinkLocal Valid: true

Figure 10.19 Verify Service Chaining

You can see that the right network host routes have been leaked to the left network (10.20.20.1/32, 10.20.20.2/32 in this case).

Now let's ping the right pod from the left pod to see the session created on the cSRX:

```
root@left-ubuntu-sc:/# ping 10.20.20.1
PING 10.20.20.1 (10.20.20.1) 56(84) bytes of data.
64 bytes from 10.20.20.1: icmp_seq=1 ttl=61 time=0.863 ms
64 bytes from 10.20.20.1: icmp_seq=2 ttl=61 time=0.290 ms
^C
--- 10.20.20.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.290/0.576/0.863/0.287 ms

root@csrx1-sc# run show security flow session
Session ID: 5378, Policy name: default-policy-logical-system-00/2, Timeout: 2, Valid
  In: 10.10.10.1/2 --> 10.20.20.1/534;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,
  Out: 10.20.20.1/534 --> 10.10.10.1/2;icmp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 1, Bytes: 84,

Session ID: 5379, Policy name: default-policy-logical-system-00/2, Timeout: 2, Valid
  In: 10.10.10.1/3 --> 10.20.20.1/534;icmp, Conn Tag: 0x0, If: ge-0/0/1.0, Pkts: 1, Bytes: 84,
  Out: 10.20.20.1/534 --> 10.10.10.1/3;icmp, Conn Tag: 0x0, If: ge-0/0/0.0, Pkts: 1, Bytes: 84,
Total sessions: 2
```

Security Policy

Create a security policy on the cSRX to allow only HTTP and HTTPS:

```
root@csrx1-sc# show security
policies {
    traceoptions {
        file ayma;
        flag all;
    }
    from-zone trust to-zone untrust {
        policy only-http-s {
            match {
                source-address any;
                destination-address any;
                application [ junos-http junos-https ];
            }
            then {
                permit;
                log {
                    session-init;
                    session-close;
                }
            }
        }
        policy deny-ping {
            match {
                source-address any;
                destination-address any;
                application any;
            }
            then {
                reject;
                log {
                    session-init;
                    session-close;
                }
            }
        }
    }
    default-policy {
        deny-all;
    }
}
zones {
    security-zone trust {
        interfaces {
            ge-0/0/0.0;
        }
    }
    security-zone untrust {
        interfaces {
            ge-0/0/1.0;
        }
    }
}
root@left-ubuntu-sc:/# ping 10.20.20.1
PING 10.20.20.1 (10.20.20.1) 56(84) bytes of data.
^C
--- 10.20.20.1 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2000ms
```

The ping fails because the policy on the cSRX drops it:

```
root@csrx1-sc> show log syslog | last 20
Jun 14 23:04:01 csrx1-sc flowd-0x2[374]: RT_FLOW: RT_FLOW_SESSION_DENY: session denied 10.10.10.1/8-
>10.20.20.1/575 0x0 icmp 1(8) deny-ping trust untrust UNKNOWN UNKNOWN N/A(N/A) ge-
0/0/1.0 No policy reject 5394 N/A N/A -1
Jun 14 23:04:02 csrx1-sc flowd-0x2[374]: RT_FLOW: RT_FLOW_SESSION_DENY: session denied 10.10.10.1/9-
>10.20.20.1/575 0x0 icmp 1(8) deny-ping trust untrust UNKNOWN UNKNOWN N/A(N/A) ge-
0/0/1.0 No policy reject 5395 N/A N/A -1
Try to send http traffic from the left to the right POD and verify the session status on the CSRX
root@left-ubuntu-sc:/# wget 10.20.20.1
--2019-06-14 23:07:34--  http://10.20.20.1/
Connecting to 10.20.20.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 11510 (11K) [text/html]
Saving to: 'index.html.4'

100%[=====================================>] 11,510      --.-K/s   in 0s

2019-06-14 23:07:34 (278 MB/s) - 'index.html.4' saved [11510/11510]
```

And in the cSRX we can see the session creation:

```
root@csrx1-sc> show log syslog | last 20
Jun 14 23:07:31 csrx1-sc flowd-0x2[374]: csrx_l3_add_new_resolved_unicast_
nexthop: Adding resolved unicast NH. dest: 10.20.20.1, proto v4 (peer initiated)
Jun 14 23:07:31 csrx1-sc flowd-0x2[374]: csrx_l3_add_new_resolved_unicast_
nexthop: Sending resolve request for stale ARP entry (b). NH: 5507 dest: 10.20.20.1
Jun 14 23:07:34 csrx1-sc flowd-0x2[374]: RT_FLOW: RT_FLOW_SESSION_
```

```
CREATE: session created 10.10.10.1/47190->10.20.20.1/80 0x0 junos-http 10.10.10.1/47190-
>10.20.20.1/80 0x0 N/A N/A N/A N/A 6 only-http-s trust untrust 5434 N/A(N/A) ge-
0/0/1.0 UNKNOWN UNKNOWN UNKNOWN N/A N/A -1
Jun 14 23:07:35 csrx1-sc flowd-0x2[374]: RT_FLOW: RT_FLOW_SESSION_
CLOSE: session closed TCP FIN: 10.10.10.1/47190->10.20.20.1/80 0x0 junos-http 10.10.10.1/47190-
>10.20.20.1/80 0x0 N/A N/A N/A N/A 6 only-
http-s trust untrust 5434 14(940) 12(12452) 2 UNKNOWN UNKNOWN N/A(N/A) ge-0/0/1.0 UNKNOWN N/A N/A -1
```

Appendix

Contrail Kubernetes Setup Installation

Note that the HW/SW requirements and installation steps that are listed here apply to the testbed used to test the theories and examples in this book. Please refer to the Juniper TechLibrary for the official HW/SW requirements and installation steps, especially if you want to build a scalable setup or more practical work.

The hardware and software required for the setup of this book are:

- Centos 7.6
- 32G memory
- 50G disk space

```
$ cat /etc/centos-release
CentOS Linux release 7.6.1810 (Core)

$ free -h
              total        used        free      shared  buff/cache   available
Mem:            31G         20G        7.0G         72M        3.8G         10G
Swap:            0B          0B          0B

$ df -h | head
Filesystem               Size  Used Avail Use% Mounted on
/dev/mapper/centos-root   47G   40G  7.3G  85% /
```

Three Node Cluster Only Setup

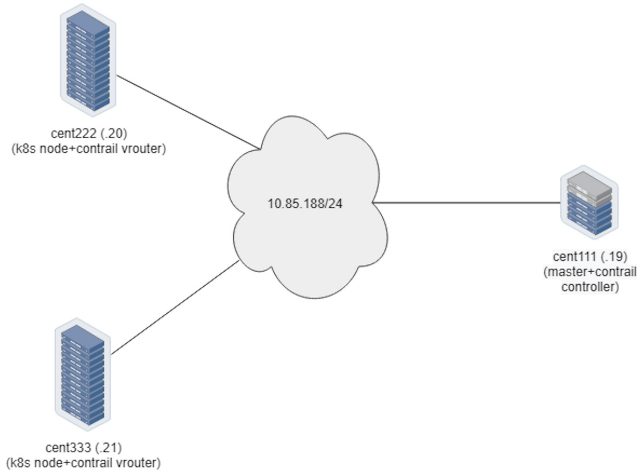


Figure A.1: Three Node Cluster Only Setup (No External Connections)

Here's the YAML template:

```
global_configuration:
  CONTAINER_REGISTRY: ci-repo.englab.juniper.net:5000
  REGISTRY_PRIVATE_INSECURE: true
provider_config:
  bms:
    ssh_pwd: Juniper
    ssh_user: root
    ntpserver: ntp.juniper.net
    domainsuffix: local
instances:
  bms1:
    provider: bms
    ip: 10.85.111.26
    roles:
        analytics:
        analytics_alarm:
        analytics_database:
        analytics_snmp:
        config:
        config_database:
        control:
        k8s_master:
        kubemanager:
        webui:
  bms2:
    provider: bms
    ip: 10.85.111.27
    roles:
        k8s_node:
```

```
        vrouter:
  bms3:
    provider: bms
    ip: 10.85.111.28
    roles:
        k8s_node:
        vrouter:
contrail_configuration:
  CLOUD_ORCHESTRATOR: kubernetes
  CONTRAIL_VERSION: master-latest
  RABBITMQ_NODE_PORT: 5673
```

Note, only the following parameters need to be changed:

```
ssh_pwd: Juniper
ssh_user: root
ntpserver: ntp.juniper.net
ip: 10.85.111.26 ~ 28
```

In this book the 3 nodes IPes are changed to 10.85.188.19~21 respectively.

Deploy Setup Based on YAML File

```
yum -y install epel-release git ansible net-tools
yum -y install python-pip python-urllib3 python-requests

git clone https://github.com/Juniper/contrail-ansible-deployer.git
cp instances.yaml contrail-ansible-deployer/config/instances.yaml
cd contrail-ansible-deployer

ansible-playbook -i inventory/ playbooks/configure_instances.yml

#if it is openstack
ansible-playbook -i inventory/ playbooks/install_openstack.yml
#if it is k8s
ansible-playbook -i inventory/ playbooks/install_k8s.yml

ansible-playbook -i inventory/ playbooks/install_contrail.yml

verification
[root@cent1 ~]# contrail-status
Pod           Service       Original Name                     State    Id           Status
             redis         contrail-external-redis             running  ac7ccf200841  Up 12 hours
analytics      api          contrail-analytics-api              running  4d5df940f2c9  Up 12 hours
analytics      collector     contrail-analytics-collector         running  eede6985b56b  Up 12 hours
analytics      nodemgr       contrail-nodemgr                    running  9a695d3ad116  Up 12 hours
analytics-alarm  alarm-gen      contrail-analytics-alarm-gen         running  a9a2b63a13e7  Up 12
hours
analytics-alarm  kafka         contrail-external-kafka             running  f2b8b87e7891  Up 12 hours
analytics-alarm  nodemgr       contrail-nodemgr                   running  539d41216ec0  Up 12 hours
analytics-snmp   nodemgr       contrail-nodemgr                   running  3a15390a119f  Up 12 hours
analytics-snmp   snmp-collector  contrail-analytics-snmp-collector     running  894c8695c8a5  Up 12
hours
analytics-snmp   topology       contrail-analytics-snmp-topology      running  1325d917c62b  Up 12
hours
config         api          contrail-controller-config-api       running  6bdf6530afd5  Up 12 hours
config         device-manager  contrail-controller-config-devicemgr   running  2eb24b537089  Up 12 hours
config         nodemgr        contrail-nodemgr                  running  13c3a8a63597  Up 12 hours
config         schema         contrail-controller-config-schema     running  2b571e48b2c1  Up 12 hours
config         svc-monitor    contrail-controller-config-svcmonitor  running  79ccd1a6975a  Up 12 hours
config-database  cassandra      contrail-external-cassandra         running  f0fc9e49fab2  Up 12 hours
config-database  nodemgr        contrail-nodemgr                  running  73fc28f9325e  Up 12 hours
config-database  rabbitmq       contrail-external-rabbitmq          running  d5bb7e78331a  Up 12 hours
config-database  zookeeper      contrail-external-zookeeper         running  cedb14f696d3  Up 12 hours
control        control        contrail-controller-control-control    running  7a25b10adb13  Up 12 hours
control        dns           contrail-controller-control-dns       running  3b660a355a44  Up 12 hours
control        named          contrail-controller-control-named     running  eb2eb603cb2d  Up 12 hours
control        nodemgr        contrail-nodemgr                  running  7bb60c059042  Up 12 hours
database       cassandra      contrail-external-cassandra         running  fcb268d42098  Up 12 hours
database       nodemgr        contrail-nodemgr                  running  7d44a2334ef3  Up 12 hours
database       query-engine    contrail-analytics-query-engine       running  3f4c5a64e7db  Up 12 hours
device-manager   dnsmasq        contrail-external-dnsmasq           running  3be66d74f44e  Up 12 hours
kubernetes      kube-manager    contrail-kubernetes-kube-manager      running  804a9badb60a  Up 12 hours
webui          job           contrail-controller-webui-job        running  786aad4792be  Up 12 hours
webui          web           contrail-controller-webui-web        running  715ebaa06bb9  Up 12 hours

== Contrail control ==
control: active
nodemgr: active
named: active
dns: active

== Contrail analytics-alarm ==
nodemgr: active
kafka: active
alarm-gen: active

== Contrail Kubernetes ==
kube-manager: active

== Contrail database ==
nodemgr: initializing (Disk for DB is too low. )
query-engine: active
cassandra: active

== Contrail analytics ==
nodemgr: active
api: active
collector: active

== Contrail config-database ==
nodemgr: initializing (Disk for DB is too low. )
zookeeper: active
rabbitmq: active
cassandra: active

== Contrail webui ==
web: active
job: active

== Contrail analytics-snmp ==
snmp-collector: active
nodemgr: active
```

```
topology: active

== Contrail device-manager ==

== Contrail config ==
svc-monitor: active
nodemgr: active
device-manager: active
api: active
schema: active loadbalancer
```

The Contrail config node, control node, analytics node, and database node are all active and running. You will see this line if your node's disk size is lower than 150G:

```
nodemgr: initializing (Disk for DB is too low. )
```

Fortunately, this is a negligible issue in the context of this book, so you can either ignore it, or, allocate a bigger hard disk to your nodes and reinstall everything.