

# DAY ONE

## ***BUILDING CONTAINERS WITH KUBERNETES AND CONTRAIL***

Ping Song, Aymon Aborabh, Yuvaraja Mariappan

# Table of Contents

1. Chapter 1: Foundation Principles .....	2
1.1. Containers Overview .....	2
1.1.1. Understanding Containers .....	2
1.1.2. Juniper VSRX vs CSRX .....	3
1.1.3. Understanding Docker .....	4
1.2. Contrail overview .....	5
1.2.1. Contrail Architecture Fundamentals .....	6
1.2.2. Contrail VRouter .....	7
2. Chapter 2: Kubernetes Basics .....	9
2.1. What is kubernetes .....	9
2.2. Kubernetes Architecture and Components .....	10
2.2.1. Kubernetes Master .....	11
2.2.2. Kubernetes Node .....	12
2.2.3. Kubernetes Work Flow .....	12
2.2.4. Kubernetes Objects .....	14
2.3. Kubernetes Pod .....	15
2.3.1. YAML file .....	16
2.3.2. Pause Container .....	20
2.3.3. Intra Pod Communication .....	21
2.4. Kubectl Tool .....	24
3. Chapter 3: Kubernetes in Practice .....	26
3.1. Labels .....	26
3.2. Namespace .....	27
3.2.1. what is Namespace .....	27
3.2.2. Create NS .....	28
3.2.3. Quota .....	28
3.3. Replication Controller .....	30
3.3.1. Create RC .....	30
3.3.2. Evaluate RC .....	32
3.4. ReplicaSet .....	34
3.5. Deployment .....	35
3.5.1. Create Deployment .....	36
3.5.2. Deployment Work Flow .....	37
3.5.3. Rolling Update .....	38
3.6. Secret .....	45
3.6.1. Opaque Secret .....	46
3.6.2. DockerConfigJson secret .....	49
3.6.3. Secret Benefits .....	57

3.7. Service .....	57
3.7.1. ClusterIP Service .....	58
3.7.2. NodePort Service .....	63
3.7.3. Loadbalancer Service .....	66
3.7.4. Kube-Proxy .....	67
3.8. Endpoints .....	67
3.9. Ingress .....	69
3.9.1. Ingress vs Service .....	69
3.9.2. Ingress Object .....	70
3.9.3. Ingress Controller .....	72
3.9.4. Ingress Examples .....	73
3.9.5. Multiple Ingress Controllers .....	75
3.10. contrail Network Policy (ch3) .....	75
3.10.1. network policy introduction .....	75
3.10.2. network policy definition .....	77
3.10.3. create network policy .....	85
3.11. Liveness Probe and Readiness Probe .....	86
3.11.1. Liveness Probe .....	86
3.11.2. Readiness Probe .....	91
3.11.3. Probe Parameters .....	92
3.12. Annotation .....	93
4. Chapter 4: Kubernetes and Contrail Integration .....	94
4.1. Contrail-Kubernetes Architecture .....	94
4.1.1. Why Contrail with Kubernetes ? .....	94
4.1.2. Contrail-Kube-Manager .....	95
4.1.3. Kubernetes to Contrail Object Mapping .....	96
4.2. Contrail Lab environment .....	97
4.2.1. Contrail Setup .....	97
4.2.2. Contrail Command .....	98
4.3. Contrail Namespaces and Isolation .....	99
4.3.1. Non-Isolated NS .....	100
4.3.2. Isolated NS .....	101
4.3.3. Pods Communication across NS .....	103
4.4. Contrail Floating IP .....	107
4.4.1. Overlay Internet Access .....	107
4.4.2. Floating IP and FIP Pool .....	107
4.4.3. FIP for Pods .....	114
4.4.4. Advertising FIP .....	115
4.5. summarization .....	119
5. chapter 5: Contrail Services .....	121
5.1. Kubernetes Service .....	121

5.2. Contrail Service . . . . .	121
5.2.1. Contrail Openstack Loadbalancer . . . . .	122
5.2.2. Contrail Sevice Loadbalancer . . . . .	123
5.2.3. Contrail Loadbalancer Objects . . . . .	125
5.3. Contrail ClusterIP Service . . . . .	131
5.3.1. ClusterIP as FIP . . . . .	131
5.3.2. ECMP Routing Table . . . . .	137
5.3.3. ClusterIP Service Workflow . . . . .	140
5.3.4. Multiple Port Service . . . . .	140
5.3.5. Contrail Flow Table . . . . .	143
5.4. Contrail Loadbalancer Service . . . . .	147
5.4.1. External IP as FIP . . . . .	147
5.4.2. Gateway Router VRF Table . . . . .	149
5.4.3. Loadbalancer Service Workflow . . . . .	151
6. chapter 6: Contrail Ingress . . . . .	159
6.1. Contrail Ingress Loadbalancer . . . . .	159
6.2. Contrail Ingress Workflow . . . . .	159
6.3. Contrail Ingress Traffic Flow . . . . .	160
6.4. Single Service Ingress . . . . .	162
6.4.1. Ingress Objects Definition . . . . .	163
6.4.2. Ingress Post Examination . . . . .	166
6.5. Simple Fanout Ingress . . . . .	180
6.5.1. Ingress Objects Definition . . . . .	181
6.5.2. Ingress post examination . . . . .	186
6.5.3. Ingress verification: from internal . . . . .	193
6.5.4. Ingress verification: from external (Internet host) . . . . .	194
6.6. Virtual Hosting Ingress . . . . .	194
6.6.1. Ingress objects definition . . . . .	195
6.6.2. Ingress post examination . . . . .	198
6.6.3. Ingress verification: from internal . . . . .	203
6.6.4. Ingress verification: from external (Internet host) . . . . .	204
6.7. Service vs Ingress Traffic Flow . . . . .	205
7. chapter 7: Packet Flow in Contrail: End to End View . . . . .	206
7.1. Setup and Utils/Tools . . . . .	206
7.2. Packet Flow Analysis . . . . .	209
7.2.1. Internet Host: Analyze HTTP Request . . . . .	209
7.2.2. Internet Host to Gateway Router . . . . .	211
7.2.3. Gateway router to Ingress Public FIP: MPLS over GRE . . . . .	212
7.2.4. Ingress Public FIP to Ingress Pod IP: FIP(NAT) . . . . .	214
7.2.5. Ingress Pod IP to Service IP: MPLS over UDP . . . . .	215
7.2.6. Service IP to Backend Pod IP: FIP(NAT) . . . . .	219

7.2.7. Backend Pod: Analyze HTTP Request .....	222
7.2.8. Return Traffic .....	224
8. chapter 8: Contrail Network Policy .....	225
8.1. introducing Contrail Firewall .....	226
8.2. contrail kubernetes Network Policy usage case .....	227
8.2.1. network design .....	227
8.2.2. lab preparation .....	228
8.2.3. traffic mode before kubernetes network policy creation .....	232
8.2.4. create kubernetes network policy .....	235
8.2.5. post kubernetes network policy creation .....	236
8.3. contrail implementation details .....	245
8.3.1. construct mappings .....	247
8.3.2. Application Policy Set (APS) .....	248
8.3.3. policies .....	248
8.3.4. firewall policy rules .....	250
8.3.5. sequence number .....	255
8.3.6. tag .....	260
8.3.7. UI visualization .....	260
9. chapter 9: Contrail Multiple Interface Pod .....	262
9.1. Contrail as a CNI .....	262
9.2. NetworkAttachmentDefinition CRD .....	263
9.3. Multiple Interface Pod .....	265
10. chapter 10: Contrail Service Chaining with CSRX .....	271
10.1. Contrail Service Chaining Introduction .....	271
10.2. Bringing Up Client and CSRX Pods .....	271
10.2.1. Create VNs .....	271
10.2.2. Create Client Pods .....	274
10.2.3. Create CSRX Pod .....	277
10.2.4. Verify podIP .....	278
10.2.5. Ping Test .....	280
10.2.6. Troubleshooting Ping Issue .....	281
10.3. Service Chaining .....	286
10.3.1. Create Service Chaining .....	286
10.3.2. Verify Service Chaining .....	291
10.3.3. Security Policy .....	292
11. appendix .....	296
11.1. contrail kubernetes setup installation .....	296
11.1.1. HW/SW prerequisites .....	296
11.1.2. 3 nodes cluster only setup .....	296
11.1.3. deploy setup based on yaml file .....	299
11.1.4. verification .....	299

## 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](http://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](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."

## Content of 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.

# Chapter 1. Chapter 1: Foundation Principles

## 1.1. Containers Overview

### 1.1.1. Understanding Containers

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. Two decades ago Virtualization was the most fashionable keyword in IT as there was a revolution in the way we build servers. It was mainly about the adoption of Virtual machine instead of dedicated physical server in building applications Back in that time Virtual Machine was a clear winner (also valid today) when it comes to scaling, portability, capacity management, cost, ..etc and you can find tones of comparison between the two approaches

If Virtualization was the keyword that sums it up, the keywords of today are Cloud , SDN and Containers

Today, the heavily discussed comparison would be between VM and Containers promising a new way to build and scale applications. While many small organizations are thinking of containers as something too early to adopt, the simple fact that Google stated that “From Gmail to YouTube to Search, everything at Google runs in containers, we run two billion container a week” might give you a clue where the industry is heading

But what is Container and how is it comparable to VM?

From technical prospect container is rooted in the Namespaces and Cgroups concept in Linux but the name might be inspired by the Actual metal cargo shipping containers that you see on ships As both share the ideas of isolating contents , carrier independence , easy movement , ...etc

Container is a logical packaging mechanism, you can think of it as Lightweight virtualization that runs an application and its dependencies in the same operating system but in different context which remove the need to replicate an entire operating system as shown in the figure 1.1 By doing so Application would be confined in a lightweight package that could be developed/tested individually then implemented and scaled much faster than the tradition VM as Developer just need to build/configure this light piece of software. Currently most of the application has been containerized and publicly available. And finally there is no need to manage/support the application per OS

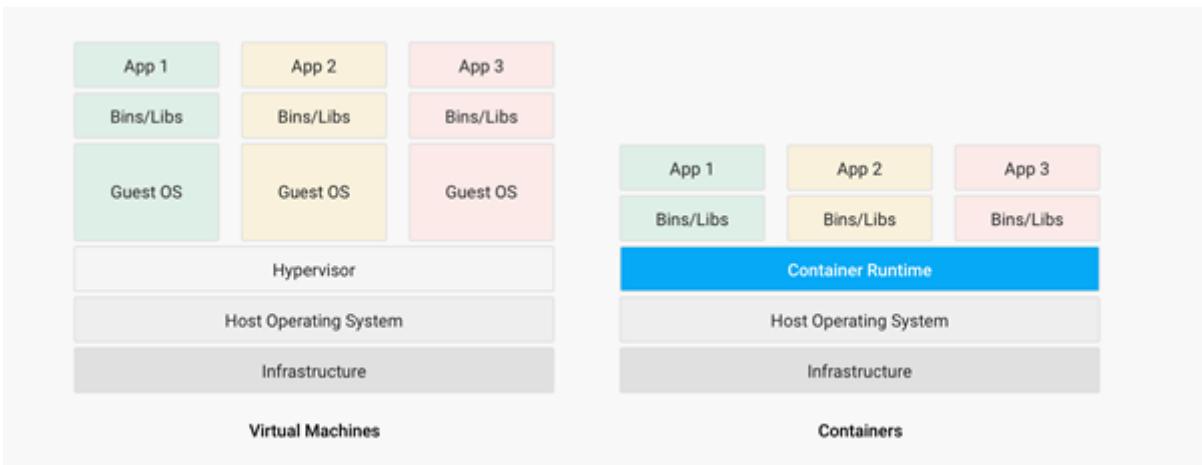


Figure 1. Container vs. VM

Many developers would call the Container Runtime in figure 1.1 “the Hypervisor of Containers” Although this is not a technically correct term, but it may be useful only to visualize the hierarchy.

As in the VM technologies, the most common Hypervisors are KVM and VMware ESX/ESXi. In the Container technologies Docker and Rkt are the most known with Docker being the most widely deployed one. Before we give an overview of Docker let’s see some useful numbers in comparing VM vs Container

### 1.1.2. Juniper VSRX vs CSRX

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

There are a lot of tests that compare performance and scaling for a giving application when it runs on Container vs VM. All of this comparison stress on the benefits of running your application in container.

But what about network functions NFV such as firewall, NAT, Routing, ...,etc? When it comes to VM based NFV, most of network vendors already implemented a Virtualized flavor of the hardware equipment that could be run on the hypervisor of standard x86 hardware. VSRX is a Juniper SRX Series Services Gateway in a virtualized form factor built on Junos, and delivers networking and security features similar to those available for SRX as for containerized based NFV, it’s the new trend. Juniper CSRX is the industry first containerized firewall offering a Compact footprint, high-density firewall for virtualized and cloud environments Table 1.1 showing a comparison between CSRX vs. VSRX which will you can see the idea of CSRX being lightweight NFV

Table 1. CSRX vs. VSRX

	vSRX	cSRX
Use Cases	Integrated routing, security, NAT, VPN, High Performance	L4-L7 Security, Low Footprint
Memory Requirement	4GB Minimum	In MB's
NAT	Yes	Yes
IPSec VPN	Yes	No

	vSRX	cSRX
Boot-up Time	~minutes	<1second
Image size	In GB's	In MB's

**NOTE**

In micro services technique the application would be split into smaller services with each part(container in that case) is doing specific job.

### 1.1.3. Understanding Docker

As we have 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 it all out as one package Docker is the software to facilitate creating, deploying, and running containers

The starting point would be the source code of the Image “Docker file” from them you can build the “Docker Image” this image can be stored and distributed to any registry -most common Docker hub- and finally you use this image to run the containers.

Docker uses a client-server architecture. 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.

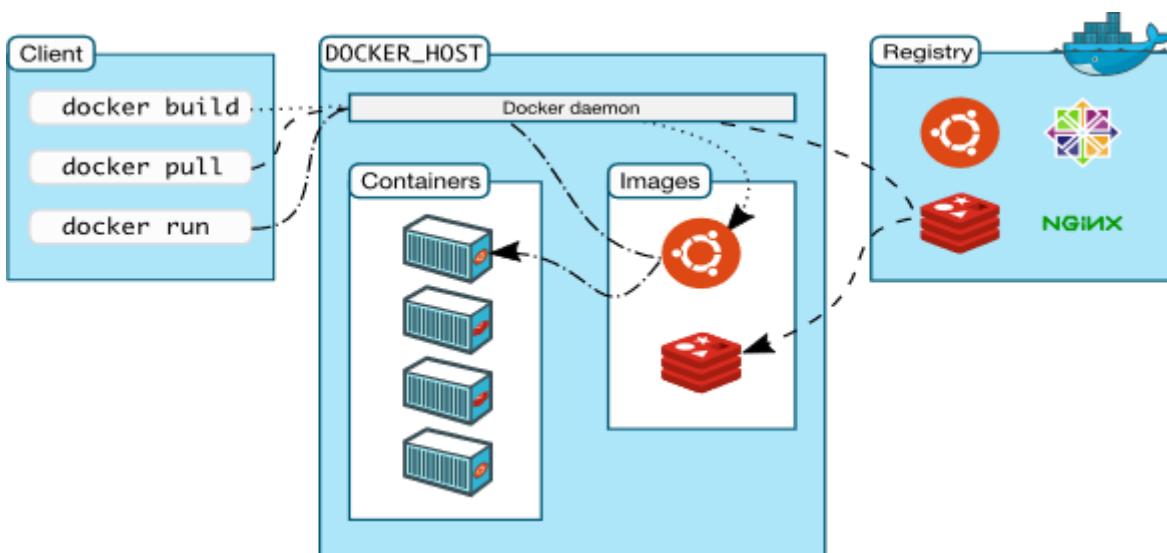


Figure 2. Docker architecture

Containers doesn't exists in a vacuum and in production you won't have just one host with multiple container but multiples of hosts running 100s if not 1000s of containers, which raise two important sets of questions

1. How these containers communicate with each other on the same host or in different host as well with outside world? Basically, the networking parts of containers
2. Who determine which containers get launch over which host? Based on what? Upgrade ? Number of containers per application? Basically, who orchestrate that ?

we will try to answer these two questions in detail for the rest of the book but if you want a quick

answer just think of Contrail and Kubernetes!!

Let's start first lay the basics foundation of contrail

## 1.2. Contrail overview

Contrail provides dynamic end-to-end networking, networking policy and control for any cloud, any workload, and any deployment, from a single user interface.

Although our focus in this book would be building a secure container network orchestrated by Kubernetes but contrail can build Virtual networks that integrate container, VM and BMS

Virtual Networks (VNs) are a key concept in the Contrail System. Virtual networks are logical constructs implemented on top of the physical networks. Virtual networks 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 the figure 1.3, the Network operate only deal with the logical abstraction of the network then contrail do the heavy lifting which include but not limited to building polices, exchanging routes, building tunnels on the physical topology.

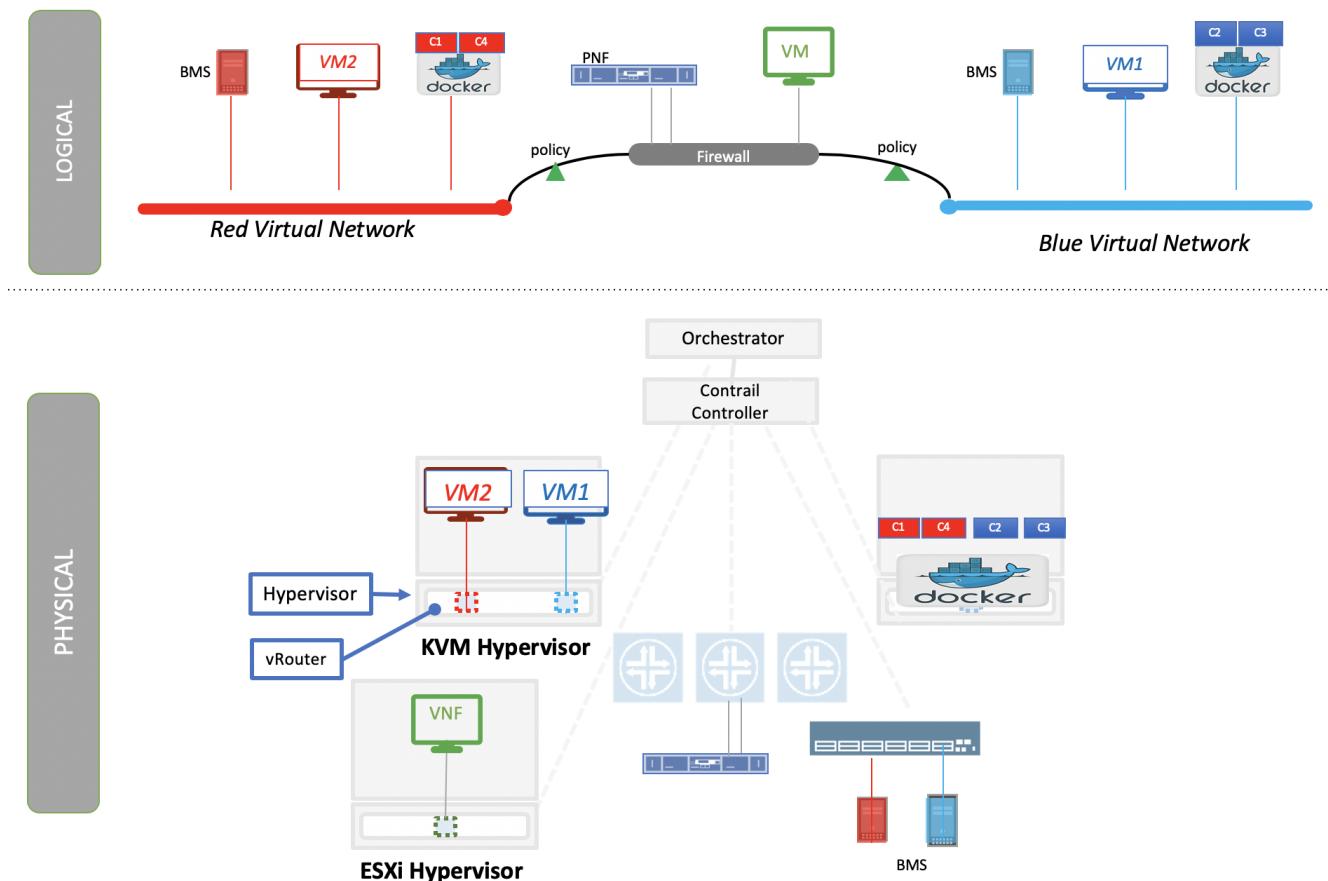


Figure 3. Virtual networks

### **1.2.1. Contrail Architecture Fundamentals**

Contrail run in Logically centralized, physically distributed model as it has two main components, Contrail controller and Contrail vrouter the Controller is the Control and management plane that Manages/configures the vrouter and Collects/presents analytics Contrail vrouter is the Forwarding plane that Provides Layer 2/3 services , Distributed firewall capabilities and Implements policies between virtual networks

Contrail integrates with many orchestrator such as OpenStack , VMware , Kubernetes , OpenShift and Mesos and use multiple protocols to provide SDN to these orchestrators as shown in figure 1.4 where

XMPP : Extensible Messaging and Presence Protocol (XMPP) is an open XML technology for real-time communication defined in RFC 6120, in Contrail it offers two main functionality, distributing routing information and pushing configuration, which are similar to what IBGP do in MPLS VPNs model plus NETCONF in device management.

BGP: is used to exchange route with physical router and in same case Contrail device manager can use Netconf to configure this Gateway

EVPN: Ethernet VPN is a standards-based technology RFC 7432 that provides virtual multipoint bridged connectivity between different Layer 2 domains over an IP network. Contrail controller exchange EVPN routes with TOR switches (acting as L2 VXLAN GW) to offer faster recovery with active-active VXLAN forwarding

MPLSoGRE/UDP or VXLAN: are three different kind of overlay tunnels to carry traffic over IP network. They are all IP-UDP packet but in VXLAN we use the VNI values in VXLAN header for segmentation where in MPLSoGRE and MPLSoUDP we use the MPLS label value for segmentation

To simplify the relation between contrail vrouter, contrail controller and the IP Fabric from the prospective of the Architecture prospective, let's compare it to MPLS VPN model in any services provider vrouter is like PE router and the VM/container is like CE but vrouter is just a slave of contrail controller. and when it comes to BMS the TOR would be the PE

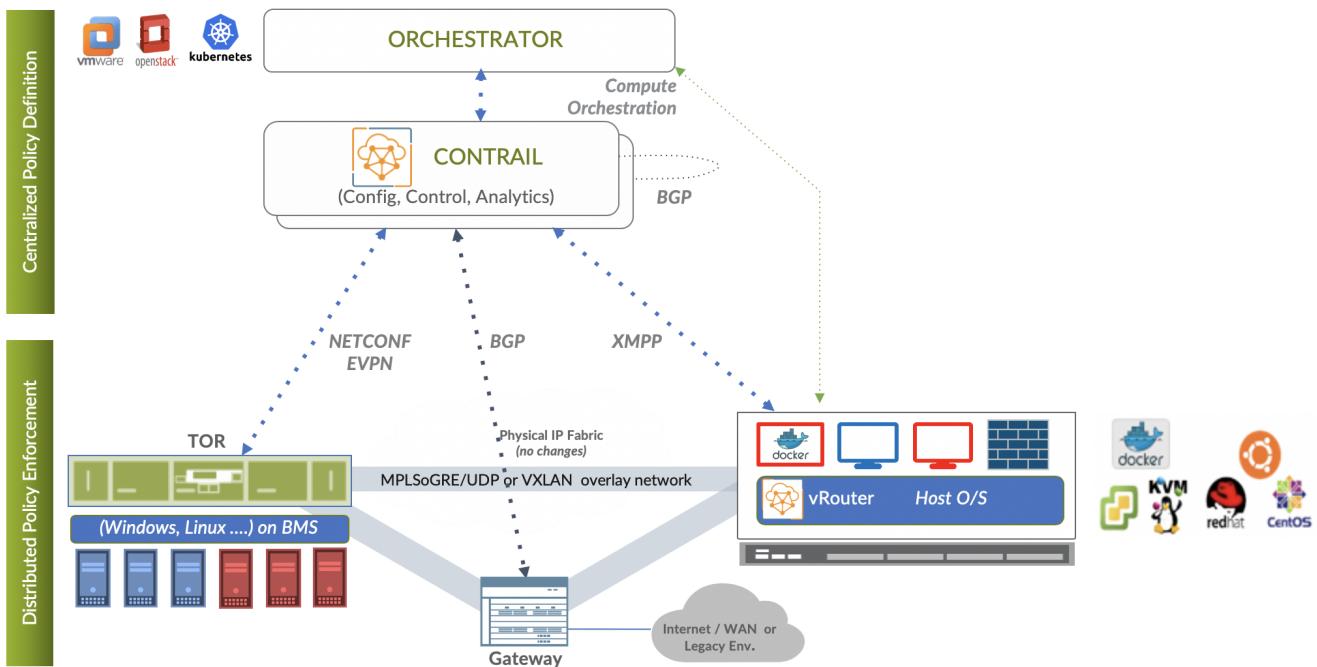


Figure 4. Contrail architecture

**NOTE**

we will be using the words “compute node” and “host” interchangeably in this book. Both would mean the entity will host the containers any container needs a compute node to host it. This host could be a physical server in your DC, or a VM (VM either in your DC or in public cloud).

### 1.2.2. Contrail vRouter

Contrail vRouter is contrail components on compute node/host as shown in figure 1.3

for a compute node in the default docker setup, containers on the same host communicate with each other as well with other containers/services hosted on other host with Docker bridge but with contrail networking, on each compute the vrouting creates VRF per virtual network offering long list of feature as will discuss

From the prospective of control plane the vrouting would

- Receive low-level configuration (routing instances and forwarding policy)
- Exchange routes Install forwarding state into the forwarding plane. Report
- analytics (logs, statistics, and events)

From the prospective of data plan the vrouting would

- Assign received packet from the overlay network to a routing instance based
- on the MPLS label or Virtual Network Identifier (VNI). Proxy DHCP, ARP, and
- DNS. Apply forwarding policy for the first packet of each new flow then
- program the action to the flow entry in the flow table of the forwarding
- plan. Forwarding the packet 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.

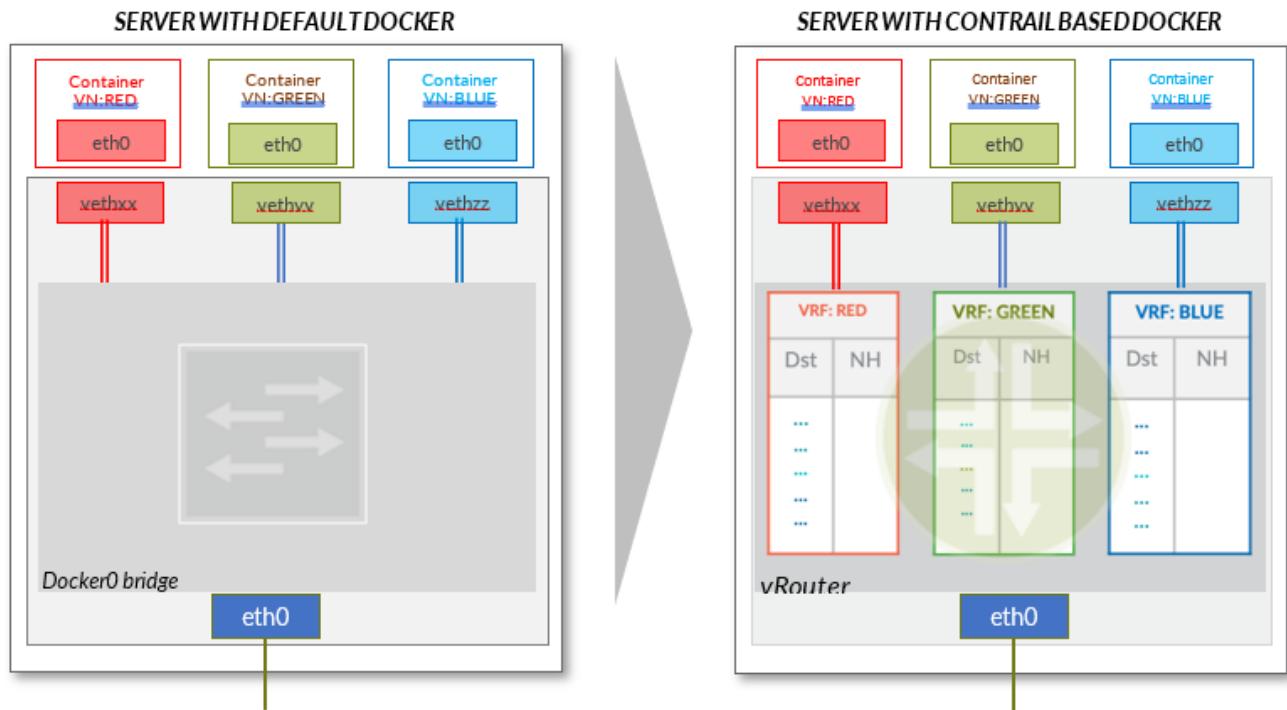


Figure 5. Docker and contrail vRouter

# Chapter 2. Chapter 2: Kubernetes Basics

This chapter introduces kubernetes, what it is, basic terminologies and the key concepts. You will learn most of the frequently referred components in kubernetes architecture. This chapter also provides some examples in kubernetes cluster environment to demonstrate the basic ideas about basic kubernetes objects.

## 2.1. What is kubernetes

starting with offical description (<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.

this tells a few important facts about kubernetes:

- an open-source project initiated by google
- a mature and stable product
- an orchestration tool
- a platform dealing with containers in a higher level

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

In simple words, container technologies like docker provides you the capability of packaging and distributing containerized applications, while an orchestration system like kubernetes allows you to deploy and manage the dockers in a relatively higher level and a much easier way.

- NOTE**
- "Borg" is still being used in google internally today
  - in many document kubernetes is frequently abbreviated as "k8s" (K - eight characters - S),
  - the "current" (as of the writing of this book) major release is v1.14.

In chapter one you've learned docker has been a prevailing and pretty mature container technology. So naturally you may wonder why you need kubernetes.

Technically speaking, kubernetes works in a relatively higher level than dockers. what does that

mean exactly? when you compare kubernetes with docker, One analogy is to compare python with C language. C is powerful enough to build almost everything including the whole bunch of fundamental OS components and APIs. but you in practice you probably would prefer to write scripts to automate tasks in your work using python much more than using C. with python most often you only need to think of which existing module already provides the magic you need, import it in your application and then quickly focus on how to use the feature to get your things done. you rarely need to worry about the low-level system API calls and hardware details.

another analogy is TCP/IP Internet protocols. when you develop a file transfer tool like `ftp`, naturally you prefer to start your work based on TCP socket instead of raw socket. with TCP socket you are seating on top of the TCP protocol, which provides a much more solid fundation 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 from the other end. with raw socket you are working on IP protocol and 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.

back to our topic of kubernetes, Assuming you want to run multiple containers across multiple machines, you will have a lot of work to do if you interact with docker directly. at least the following tasks should be in your "worry list":

- login different machines and Spawning containers across the network
- Scaling up or down by adding or removing containers when demand changes
- Keeping storage consistent with multiple instances of an application
- Distributing load between the containers running in different node
- Launching new containers on different machines if something fails

you will quickly find that doing all of this manually with docker will be overwhelming. with the high-level abstractions and the objects representing them in kubernetes API, all of these tasks become much easier.

**NOTE**

kubernetes is not the only tool in its kind, docker has its own orchestration tool named "swarm". this book will focus on kubernetes.

## 2.2. Kubernetes Architecture and Components

in a Kubernetes cluster there are two type of nodes, each running a very well-defined set of processes:

- head node: called "master", or "master node", the head and brain that does all thinking and decisions, all of intelligence are located here.
- worker node: called "node", or "minion", the arms and feet that conduct the workforce.

The "nodes" are controlled by the "master" and in most of the time you will only need to talk to master .

One of the most common interface between you and the cluster is a 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 you will read example of using kubectl to create kubernetes objects. For now just remember: Whenever you are working with "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 - something we call a "server", which can be either physical server or virtual machine. in context of kubernetes clusters, a "node" often specifically refers to a "worker node".

**NOTE**

you rarely need to "bypass" the master and work with nodes directly, but you can login to node and run all docker command to check running status of containers. there is an example showing this later in this chapter.

### 2.2.1. Kubernetes Master

A kubernetes "master", or "master node", is like one's head and brain. in 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 container. this procedure is called "scheduling".

master is also responsible for maintaining the desired state for the cluster. when you give an order "for this web server make sure there are always 2 containers backing up each other!", the master will keep monitor the running status and spawning new container anytime when the number of the web server containers in "running" status becomes less than 2 due to any failures.

The master is also responsible for other many jobs.

Typically you only need a single master node in the cluster, however, the master can also be replicated for higher availability (HA) and redundancy.

the master's functions is implemented by a collection of processes running in node. The processes in a master node providing the primary features are:

- **kube-apiserver:** front-end of the control plane, providing REST APIs
- **kube-scheduler:** do the "scheduling": decide where to place the containers depending on system requirement (CPU, memory, harddisk, etc) and other custom parameters or constraints (e.g. affinity specification)
- **kube-controller-manager:** the single process implementing most of the different type of "controllers", which makes sure that the state of the system is what it should be. some controller examples:
  - Replication Controller
  - ReplicaSet

- Deployment
  - Service Controller
- **etcd**: database to store the state of the system.

**NOTE**

for the sake of simplicity a few other components are not listed (e.g. **cloud-controller-manager**, **DNS server**, **kubelet**). they are not trivial negligible components, but skipping them for now does not stop you from understanding the kubernetes basics.

### 2.2.2. Kubernetes Node

nodes in a cluster are the machines that run the user end applications. in production there can be dozens or hundreds of nodes in one cluster depending on the designed scales. nodes are the real workforce under the hood provided by a cluster. usually all of the containers and workloads are running in nodes. A "node" runs following processes:

- **Kubelet**: the Kubernetes agent process that runs on master and all the nodes. it interacts with master (through kube-apiserver process) and manage the containers in local host.
- **kube-proxy**: process that implements "kubernetes service" (will introduce in chapter three) using linux iptable in the node
- **container-runtime**: local container - mostly 'docker' in today's market, holding all of the running "dockerized" applications.

**NOTE**

the name "proxy" may sound confusing for kubernetes beginners. it's not really a "proxy" in current kubernetes architecture. kube-proxy is a system that manipulates linux IP tables in that node so that the traffic between the pods and the nodes will flow correctly.

### 2.2.3. Kubernetes Work Flow

after you get some basic idea about the master and node and the main processes running in each, it is time to look at how things works together in figure 2.1

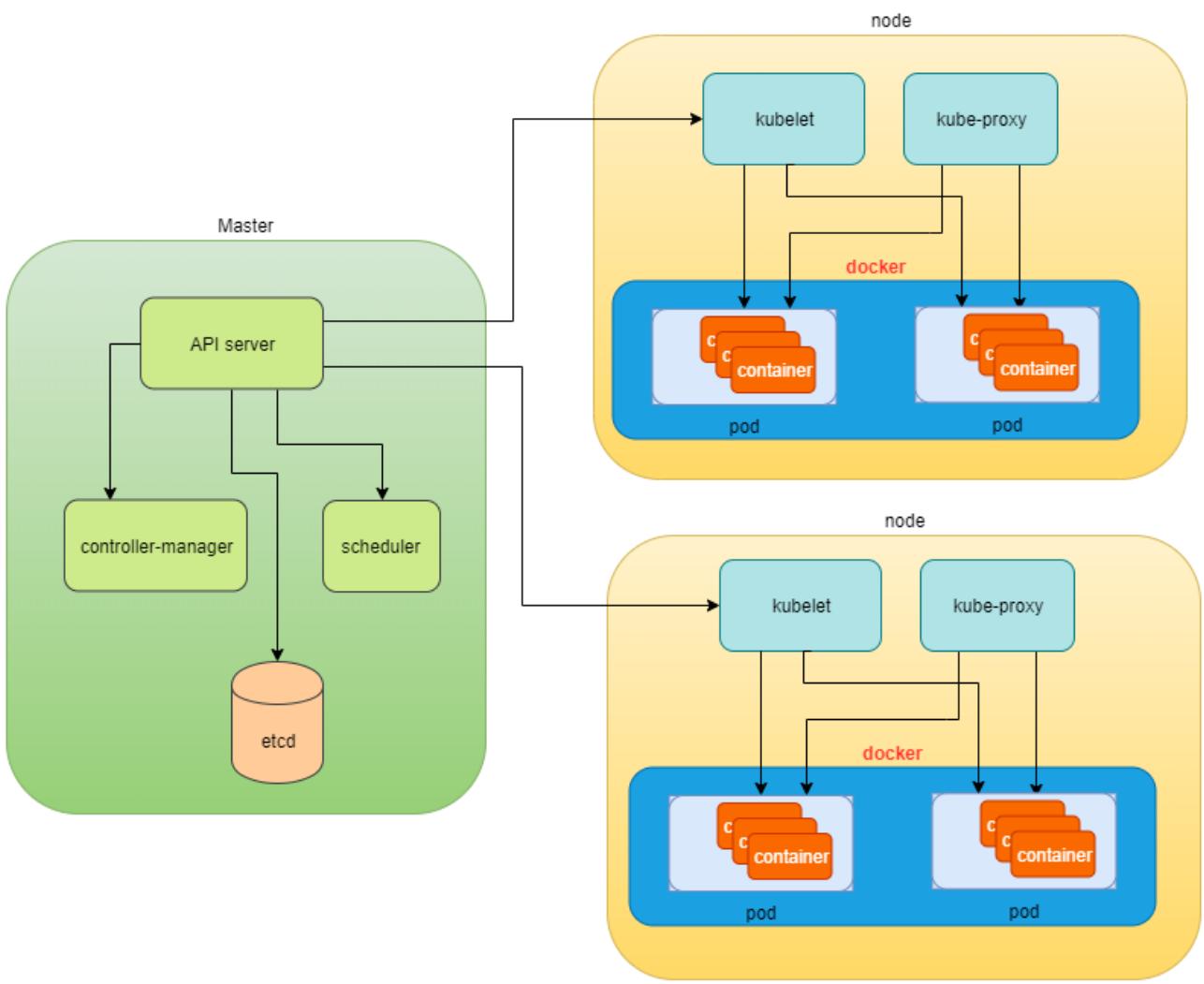


Figure 6. kubernetes architecture

At the top behind `kubectl` is where you are. via `kubectl` commands you talk to kubernetes "master", which manages the 2 "node" boxes on the right. it interacts with the master process "kube-apiserver" via its REST-API exposed to the user and other processes in the system.

Now let's say you send some `kubectl` commands - something like `kubectl create x`, to spawn a new container. You can give details about how exactly you want your container to be spawned along with the running behaviors. the container specifications can be provided either as `kubectl` command line parameters, or options and values defined in a config file. You will read an example on this shortly.

1. The `kubectl` client will first translate your CLI command to one more REST-API call(s) and send 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 all 3 available ones to execute the job. this is the scheduling procedure.
3. Once "kube-scheduler" returns the "target node", "kube-apiserver" will dispatch the task with all of the details describing the task.
4. "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 of the cluster.

**NOTE**

actually a `master` can be also a fully-featured `node` and carry pods workforce just like a node does. therefore, `kubelet` and `kube-proxy` components existing in node also exists in master. in the figure above we didn't include these components in master, just to give a simplified conceptual separation of master and node. in your setup you can use command `kubectl get pod --all-namespaces -o wide` to list all pods with their location. pods spawned in master is usually running as part of the kubernetes system itself - typically within `kube-system` namespace. kubernetes `namespace` will be discussed in chapter 3.

Of course this is just a very simplified work flow, but you get the basic idea. In fact with the power of kubernetes you rarely need to work with containers directly. you will work with some higher level objects which, hide most of the low level operation and details and present the task in a higher level and much simpler form.

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 ('replica controller') with replica two".

what will happen now is that once the 2 docker containers are up and running, kube-apiserver 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. for example you will observe that if any one of two docker containers goes down, a third container will be spawned and the broken one will be removed automatically.

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

The small "cost" of working in a level higher than docker engine is to understand a few extra "kubernetes objects".

you will read more about kubernetes objects in the next section.

## 2.2.4. Kubernetes Objects

Now you understand the role of 'master' and 'node' in a kubernetes cluster, and in figure 2.1 you see how a basic workflow looks. now let's start to look at more kubernetes "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 oftenly used objects are:

- basic Kubernetes objects
  - Pod
  - Service
  - Volume
  - Namespace
- higher-level objects (Controllers):
  - ReplicationController
  - ReplicaSet
  - Deployment
  - StatefulSet
  - DaemonSet
  - Job

**NOTE**

"high-level" objects are build upon the basic objects. They provide additional functionality and convenience features.

in the frontend, kubernetes get all things done via a group of "object". with kubernetes you only needs to think of how to describe your task in the config file of the objects, no need to worry about how it will be implemented in container level. "under the hood", kubernetes interact 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 (e.g. by 'docker build').

you will read more about each object and their magic power with examples in chapter 3. later in this chapter we'll look at the the most fundamental object: POD.

## 2.3. Kubernetes Pod

"POD" is the first kubernetes object you will learn. the kubernetes website describe 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

this brings 2 facts:

- basically pod is nothing but a group of containers
- all containers in a pod shares storage and network resources.

what is the benefit of using "pod" comparing with the old way of dealing with each individual containers? considering a simple usage case that you are deploying a web service with docker. you will need not only the frontend service, e.g. an apache server, but also some "supporting services" like a database server, a logging server, a monitoring server, etc. each of these supporting services needs to be running in its own docker. so essentially you will find yourself always working with a group of docks whenever "a web service" docker is needed. In production the same scenario applies to most of the other docker service as well. eventually you will ask: is there a way to group a bunch of docker containers in a higher-level "unit", so you only need to worry about the low-level inter-docker interaction details once?

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

besides that, containers in the same pod will share the same network space. 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'll see more about these advantages later.

now, let's get your feet wet. we'll look at how to use a config file to launch a "pod" in kubernetes cluster.

### 2.3.1. YAML file

First thing to look at is YAML. Along with many other many ways of configuring kubernetes, YAML is the "standard" format being used in kubernetes config file. YAML is widely used in a lot of software fields so mostly likely you are already familiar with it. In case you are not, its not a big deal because YAML is a pretty easy language to learn. We'll explain each line of the YAML config of a pod and you will understand the YAML format as a "by-product" of your POD learning process.

### Example 1. POD configuration file in YAML format

```
#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 3 basic data types:

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

in this example you are also seeing "nested" YAML data structure:

- "mapping of a mapping": **spec** is the key of a map, where you define a pod's specification. in this example we only define behavior of the containers to be launched in the pod. the value is another map with the key being **containers**.
- "mapping of a list". values of the key "containers" is a list of two items: frontend and redis container, each of which again, are a mapping describing the individual container with a few attributes like name, image and ports to be exposed.

*a few important rules of YAML:*

- case sensitive
- elements in same level share same left indentation, the amount of indentation does not matter
- tab characters are not allowed to be used as indentation
- blank lines does not matter
- comment a line with "#"
- use quote ' to escape special meaning of any character

before we dive into more details of the yaml file, let's finish the pod creation:

## Example 2. create pods

```
$ 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>
```

we created our first kubernetes "object" - a pod named **pod-1**. but where are the containers? the above output tells the clues. it reads:

a pod **pod-1** (**NAME**), containing 2 containers(**READY /2**), has been launched in kubernetes worker node **cent333** with an IP address **10.47.255.237** assigned. both containers in the pod is up (**READY 2/2**) and has been in running **STATUS** for 27s without any **RESTARTS**.

here is a brief line-by-line comments about what the yaml config says:

- line 1: this is a comment line. with a **#** ahead we can put any comment in the yaml file.

**NOTE**

throughout this book we use this first line to give filename of a yaml file. the filename will be used in later command when creating the object from the yaml file.

- line 2,3,4,8: the 4 yaml mappings are the main components of a pod definition.
  - **apiVersion**: there are different versions, for example, v2. here specifically it is version 1.
  - **kind**: remember there are different type of kubernetes object, here we want kubernetes to create a 'pod' object. later you will see kind being **ReplicationController** or **Service** in example of other objects.
  - **metadata**: to identify the created objects. besides the name of the object to be created, another important meta data is "labels". you will read more about it in chapter3.
  - **spec**: gives the specification about the pod behavior.
- line 9-15: the pod specification here is just about the 2 containers. the system downloads the images, launches each container with a name and expose the specified ports respectively.

to get more details of what is running inside of the pod:

*Example 3. describe a 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, our pod **pod-1** is composed of 2 containers declared in the YAML file, **apache** and **db** respectively, with an IP address assigned by kubernetes cluster and shared between all containers as shown in following figure:

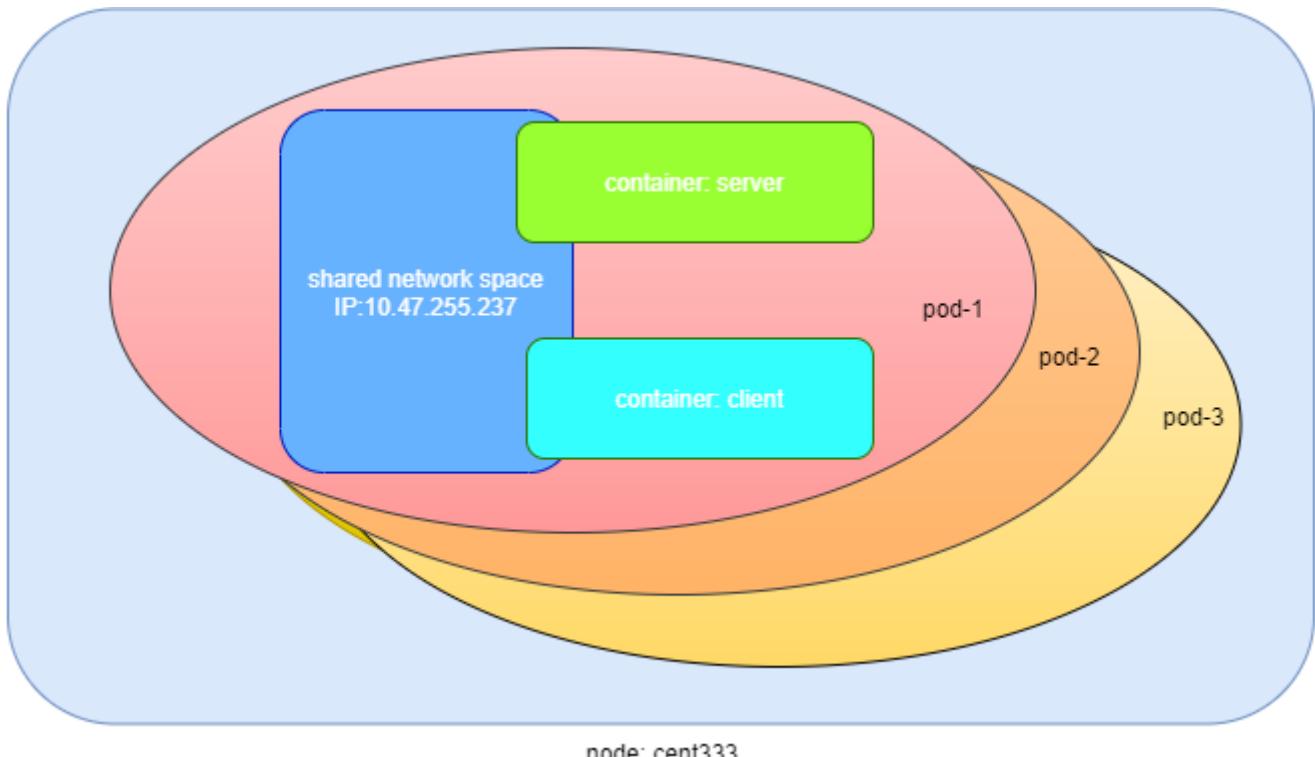


Figure 7. node, pod and containers

### 2.3.2. Pause Container

if you login to node `cent333`, you will see the docker containers running inside of the pod:

*Example 4. pause 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 by the kubernetes system for each pod. The `pause` container is created to manage the network resources for the pod which would be shared by all the containers of that pod.

below figure shows a pod including a few user containers and a `pause` container.

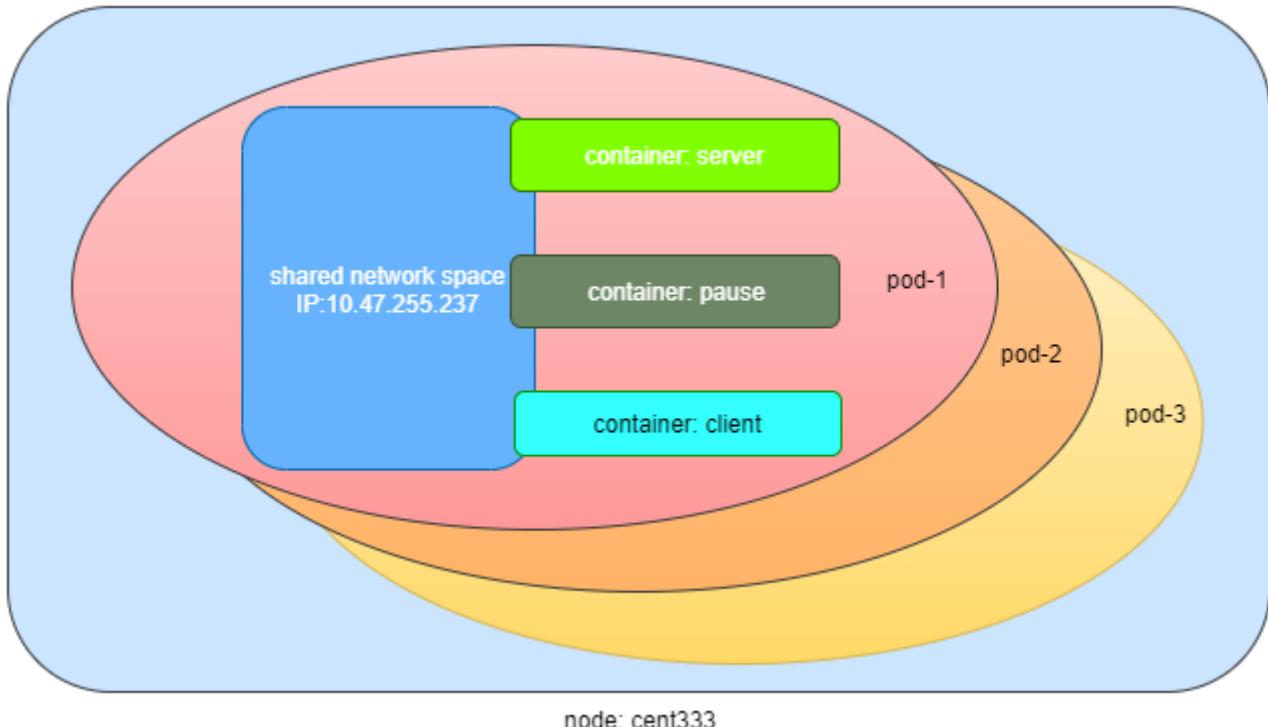


Figure 8. pod, user containers and the special `pause` container

### 2.3.3. Intra Pod Communication

in kubernetes master, to login to a container:

*Example 5. login to a container directly from 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 realized that this is pretty neat. remember the containers were launched at one of the "node", with docker you will have to first login to the correct remote node, and then use a similiar `docker exec` command to login to each container. kubernetes hides these details and allow you to do everything from one node - the master.

now check processes running in the container:

*Example 6. 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 ?          S1  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
```

#### Example 7. 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
```

the `ps` command output shows that each container is running its own process. however, the `ss` and `ip` command output indicate that both container 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`. we can test this out by starting a tcp connection using `curl` command.

suppose from `client` container, we want to get a web page from the `server` container. we can simply start `curl` using `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>
  
</body>
</div>
</html>

```

now monitor the TCP connection state: the connection is established successfully.

*Example 8. monitor connections*

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	:::*

same exact connection can be seen from server container:

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	127.0.0.1:34182 #<---
LISTEN	0	128	:::22	:::*

## 2.4. Kubectl Tool

so far you've seen we created the object by `kubectl` command. this command, just like the `docker` command in docker world, is the interface in kubernetes world to talk to the cluster, or more precisely, the kubernetes master, via kubernetes API. it is a very versatile tool that provides many 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 your current environment supported options by logging into the master and typing `kubectl`, followed by two `tab` keystrokes.

*Example 9. kubectl tab completion*

```
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 setup auto-completion for kubectl command, follow the instruction from help of `completion` option: `kubectl completion -h`

you will see and learn some of these options in the rest part of this book.

# Chapter 3. Chapter 3: Kubernetes in Practice

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

## 3.1. Labels

Imagine you have a POD that's need to be host on a machine with certain specifications ( SSD HD, physical location , processing power , ..,etc ) OR imagine you want to search or group your PODs for easier administration what would you do ? then label is your way to go, in Kubernetes Label is a Key/value pairs attached to an object let's see how can we use label to make a POD is lunched on a certain machine

- In kubernetes, any objects can be identified using a label.
- You can assign multiple labels per object but avoid using too much label or too little, too much would get you confused and too little won't give the real benefits of grouping, selecting and searching.
- Best practice is to assign labels to indicate
  - application/program ID use this POD
  - owner (who manage this POD/application)
  - stage (the POD/application in development/testing/ production as well version)
  - resource requirements (SSD, CPU, storage)
  - location (preferred location/zone/ Datacenter to run this POD/application)

**NOTE**

Let's assign label (stage: testing) & (zone: production) to two nodes respectively then try to lunch a POD in a node which 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 lunch 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/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 web-server.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 POD to certain node without label by adding the argument `nodeName: nodeName` under `spec` in the YAML file where `nodeName` is the name of the node

## 3.2. Namespace

### 3.2.1. what is Namespace

As in many other platforms, normally there is more than one users (or teams) working on a kubernetes cluster. suppose a pod named 'webserver1' has been built by 'dev' department, 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 appear more than once in the same scope.

'Namespaces' (or 'NS' for short) provides the scopes for the kubernetes resources like project/tenant in openstack. Names of resources need to be unique within a namespace, but not across namespaces. it is a nature 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 NS is to make some resources readable by all users without authentication. it exists mostly in kubernetes clusters bootstrapped with `kubeadm` tool only.

### 3.2.2. Create NS

creating a NS is pretty simple. just `kubectl` command does the magic. you dont need to have a yaml file.

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

new namespace dev is now created

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

now a `webserver1` pod in `dev` NS won't conflict with a `webserver1` pod in `sales` NS.

```
$ 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>
```

### 3.2.3. Quota

similar to openstack 'tenant', you can now apply constraints that limits resource consumption per namespace. for example, you can limit the quantity of objects that can be created in a namespace, total amount of compute resources that may be consumed by resources, etc. the constraint in k8s is called 'quota'. here is an example:

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

we just created a quota 'quota-onepod', and the constraint we gave is 'pods=1' - only one pod is allowed to be created in this NS.

```
$ 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: foobar
    namespace: quota-onepod
    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: ""
```

now create a pod in it:

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

it works fine. now 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 an error saying "exceeded quota".

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
```

## 3.3. Replication Controller

you have learned how to launch a pod that representing your containers from its yaml file in chapter 2. one question will rise in your mind: what if we need 3 exactly the same pods (each runs a apache container) to make sure the web service appears more robust? shall we change the name in yaml file then repeat the same commands to create required pods? or maybe with a shell script? kubernetes already has the objects to address this exact demand and the right answer are [RC - replicationController](#) or [RS - ReplicaSet](#)

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

### 3.3.1. Create RC

let's look at how it works with an example. first create a yaml file for a RC object named [myweb](#).

```

#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

```

again, `kind` indicates the object type that this yaml file is to define, here it is a RC instead of a pod. in `metadata` it is showing the RC's `name` as `rc-webserver`. in `spec` is the detail specification of this RC object. `replicas 3` indicates a same pod will be cloned to make sure the total number of pods created by the RC is always 3. `template` gives information about the containers that will run in the pod, 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:

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 3 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 pod directly. the workflows are shown in this diagram:

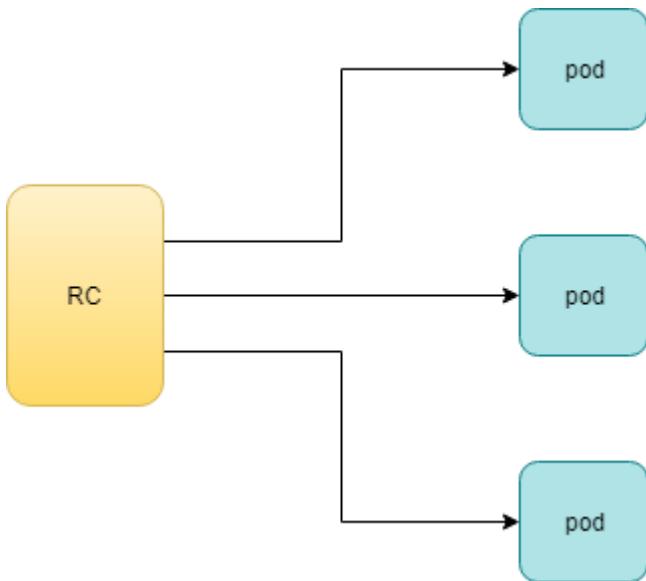


Figure 9. ReplicationController

with `replicas` parameter specified in 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 runs into failures. the key to learn is, individual pod 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`.

### 3.3.2. Evaluate RC

you can evalaute rc's impact by deleting one of the pod. to delete a resource with `kubectl`, use `kubectl delete` sub-command:

```
$ kubectl delete pod rc-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 "Terminating", immediately a new pod is spawned. eventually the old pod will go away and new pod will be up and running. total number of running pod remains unchanged.

you can also scale up/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 in practice that, two very similar objects [RS - ReplicaSet](#) and [Deploy - Deployment](#) have been developped with more powerful features introduced. Roughly, you can call them "next generation of RC". let's stop exploring more RC features for now and move our focus to these 2 new objects.

## 3.4. ReplicaSet

ReplicaSet, or RS object, is pretty much the same thing as a RC object, with just one major exception - the looks of selector.

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

RC uses "Equality-based" selector only while RS support extra selector format - "set-based". function-wise the two forms of selector do the same job - to "select" the pod with a matching "label".

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

#RC selector
app: webserver
```

```
$ kubectl create -f rs-webserver.yaml
replicaset.extensions/webserver created

$ kubectl get pod
NAME                  READY   STATUS    RESTARTS   AGE
webserver-kt9zx      1/1     Running   0          8s
```

a RS is created and it launches a pod, just same as what a RC would do. if you compare the [kubectl describe](#) on the 2 objects:

```
$ kubectl describe rs webserver
.....
Selector: app=webserver,app in (webserver) #<---
.....
Type Reason Age From Message
---- ---- -- -- -----
Normal SuccessfulCreate 15s replicaset-controller Created pod: webserver-kt9zx

$ kubectl describe rc webserver
.....
Selector: app=webserver #<---
.....
Type Reason Age From Message
---- ---- -- -- -----
Normal SuccessfulCreate 19s replication-controller Created pod: webserver-tbbhc
```

as you see, most part of the output are the same, with only exception of selector format. you can also scale the RS same way as you 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
```

## 3.5. Deployment

now you may start to wonder why kubernetes has different objects to do the almost same job. as mentioned earlier the features of RC has 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.

### 3.5.1. Create Deployment

simply changing `kind` attribute from `ReplicaSet` to `deployment` we 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 Deploy:

```
$ 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
```

**NOTE** make sure deleting the RS that you created in earlier section to avoid confusions.

Actually the deployment is a relatively higher level of abstraction than RC and RS. deployment does not create a pod directly, 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>

```

### 3.5.2. Deployment Work Flow

what happens is when you create a Deployment, a replica set is created automatically. The pods defined in a Deployment object are created and supervised by the Deployment's replicaset.

the workflow is shown in the below diagram:

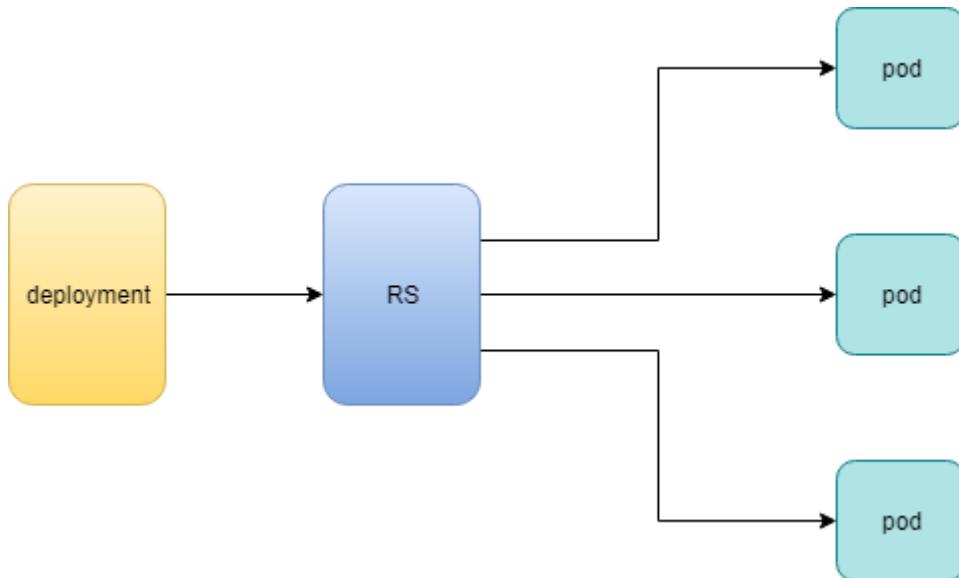


Figure 10. Deployment and RS

You might still be wondering why you need `RS` as one more layer sitting between `deployment` and `pod`. You will find the answer in the next section.

### 3.5.3. Rolling Update

"rolling update" feature is one of the "more powerful feature" coming with deployment object. In this section we'll demonstrate the feature with a test case, then we'll 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 comparing with the new version supported by `Deployment`. In this book we'll focus on the new implementation.

#### evalaute rolling update

Suppose we have a `nginx-deployment`, with `replica=3` and pod image `1.7.9`. Later we want to upgrade the image from version `1.7.9` to new image version `1.9.1`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

with `kubectl` we can use `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  #<---

```

two changes we can observe here:

- 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 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 old image

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

wait and keep checking the pods status, eventually all old pods are terminated and 3 new pods are running - the new pod name confirms 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 new version of the image.

## how it works

after you see our update process, you may argue that: hold on... this is now "update", this should be called "replacement" - kubernetes use 3 new pods running with new image to replace the old pods! precisely speaking, yes that is true. but that is how it works kubernetes's "philosophy" - pod is cheap and replacement is easier. imaging how much work it will be when you have to "login" each pod, uninstall old images, cleaning up the environment and 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 is NOT to "login" to the existing pod and do the image update in there, instead, the new RC 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, making the total number of running pod remaining unchanged. new RS will continue to scale up by one and original RS scales down by same number. this process repeats until number of pods created by new RS reaches the original replica number defined in the deployment, and that is the time when all of the original RS's pods are terminated. this process is depicted in this diagram:

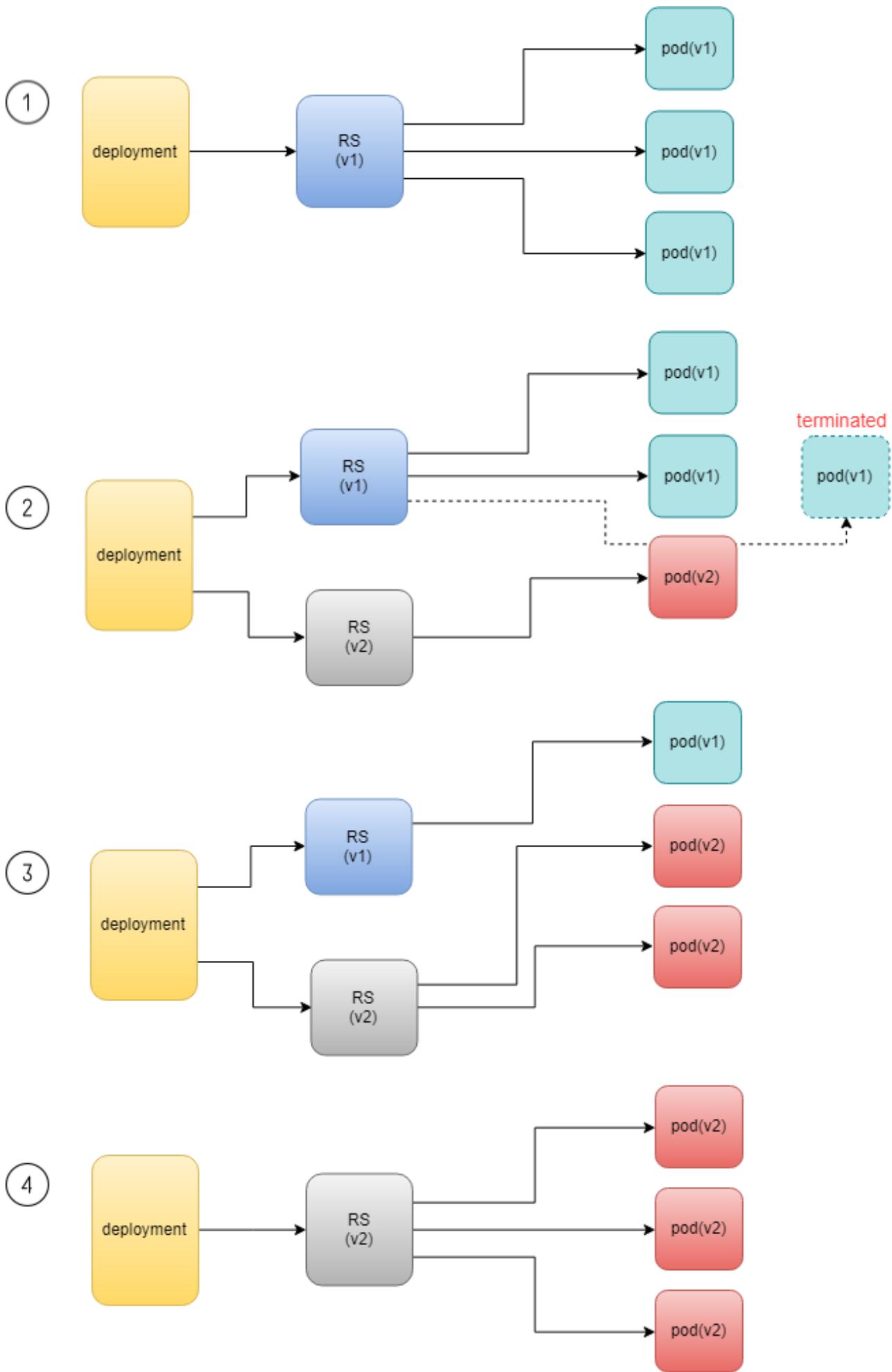


Figure 11. Deployment rolling update

as you can see in this figure, this 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` who is `deploying` and driving `ReplicaSet` object, which, in this sense working as merely a backend of it.

this is why `deployment` is considered a higher layer object in kubernetes, also the reason why it is officially recommended to never use `ReplicaSet` alone without `deployment`.

## record

deployment also has the ability to "record" the whole process, so in case 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. comparing with how much work it takes to prepare for the software upgrade during maintenance window in the old days, this is going to be a killing feature to have!

**TIP** This is pretty much similar as the junos's `rollback` magic command that you probably use everyday when you need to quickly revert the changes you make to your router.

## 3.6. Secret

in any modern network system, user or administrator need to deal with sensitive information, such as `username/passwords/ssh` keys/etc, in the platform. same thing applies to the pods in kubernetes environment. However, exposing these information in your pod specs as cleartext may introduce security concerns and you need a tool/method to resolve the issue - at least to avoid the cleartext credentials as much as possible.

Kubernetes `Secrets` object is designed specifically for this purpose. it encodes all sensitive data and expose it into pods in a "controlled way".

this is the official definition of kubernetes secrets:

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 some secrets. To use a secret, a pod needs to reference the secret.

there are many different types of secrets each serving a specific usage case. there are also many methods to create a secret and a lot of different ways to refer it in a pod. a complete discussion of secrets is out of the scope of this book. refer to the offical document to get all details and track all up-to-date changes.

In this section, we'll look at some commonly used secret types. you will also learn several methods to create a secret and how to refer it in your pods. in the end, we will summarize the main benefits of kubernetes secrets object to understand how it will help to improve the sytem security.

*common secret types:*

**opaque**

this type of Secret can contain arbitrary key-value pairs, so it is treated as "unstructured" data from kubernetes's perspective. all other types of secret has constaint content.

**kubernetes.io/dockerconfigjson**

this type of secret is used to authenticate with a private container registry (e.g. a juniper server) to pull your own private image.

**tls**

tls secret contains a TLS private key and certificate. it is used to secure an Ingress. you will see an example of Ingress with tls secret in chapter 4.

**kubernetes.io/service-account-token**

when processes running in containers of a pod access the apiserver, they has to be authenticated as a particular Account (e.g., account **default** by default). this account that is 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 this type of secret and **service-account** in this book.

### 3.6.1. Opaque Secret

secret of type **opaque**, represents "arbitrary" user-owned data - usually you want to put in **secret** some kind of "sensitive" data, 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 them with **base64** tool:

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

then, we 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: dXNlcjIyMjIyMjIy  
  password: cGFzc3dvcmQx
```

alternatively, you can define the same secret from kubectl CLI directly, with `--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-11e9-90a3-0050569e6cf
type: Opaque

```

## refer opaque secret

once created, next you will need to use the **secret** in a pod, and the user information contained in **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 **secret** can be in one of these forms in a container:

- files
- environmental variables

here we'll demonstrate using secret to generate environmental variables in container:

```

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
```

login 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 we encoded with **base64** is now present in the container!

### 3.6.2. DockerConfigJson secret

**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 kubernetes pod may point to a private container registry. in that case, kubernetes need to authenticate with that registry in order to pull the image. **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 to `kubectl` command and let it to generate the secret:

```
$ kubectl create secret docker-registry secret-jnpr1 \
--docker-server=hub.juniper.net \
--docker-username=JNPR-FieldUser213 \
--docker-password=CLJd2jpMsVc9zrAuTFPn
secret/secret-jnpr created
```

*Example 10. 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
```

please note that only the first line in the output is the secret we just created. the second one is a `kubernetes.io/service-account-token` type of secret that was created by kubernetes system 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: eyJhdXRocYI6eyJodWIuanVuaXBlcj5uZXQvc2...<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-be58-11e9-9367-0050569e6fcf
type: kubernetes.io/dockerconfigjson
```

not surprisingly, we don't see any sensitive information in the form of cleartext, there is a `data` portion of the output where we see a very long string as the value of key `.dockerconfigjson`. it seems to has a transformed look from the original data, but at least it is not that "sensitive" anymore now - afterall one purpose of using 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 back: just pipe the value of key `.dockerconfigjson` into `base64` tool, the original `username` and `password` information is printed again:

### Example 11. decode the secret data

```
$ echo "eyJhdXRoCYI6eyJodWIuanVua..." | base64 -d | python -mjson.tool
{
    "auths": {
        "hub.juniper.net": {
            "auth": "Sj5QUi1GaWVsZFVzZXIyMTM6Q0xKZDJqcE1zVmM5enJBdVRGUG4=",
            "password": "CLJd2kpMsVc9zrAuTFPn",
            "username": "JNPR-FieldUser213"
        }
    }
}
```

some highlights in the above output:

- `python -mjson.tool` is used to format the decoded json data before printing 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 an docker image

here is 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"
        }
    }
}
```

TIP

the `--output=xxxx` option filters the `kubectl get` output so only value of `.dockerconfigjson` under `data` is printed. 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 we can create a secret from docker credential file.

## **docker credential file (~/.docker/config.json)**

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

*generate docker credential info*

first let's check the docker config file:

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

nothing really. depending on the usage of the setups you may see different output here. but the point is that this docker config file will be updated automatically each time when you `docker login` a new registry.

let's test it out.

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

in file `mydockerpass.txt` is login password for my 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.

if you want you can give the password directly, and you will get a friendly warn that this is "insecure".

**TIP**

```
$ docker login hub.juniper.net --username JNPR-FieldUser213 --password
MYPASS
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
Login Succeeded
```

now the docker credential info is generated in the updated config.json file:

```
$ cat .docker/config.json
{
    .....
    "auths": {    #<---
        "hub.juniper.net": {
            "auth": "Sj5QUi1GaWVsZFxZXIyMTM6Q0xKZDJqcE1zVmM5enJBdVRGUG4="
        }
    },
    .....
}
```

The login process creates or updates a config.json file that holds an authorization token.

*Example 12. create secret from .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:
ewoJImF1dGhzIjogewoJCSJodWIuanVuaXBlcis5uZXQi0iB7CgkJCSJhdXRoIjogIlNrNVFVaTFHYVdWc1pGVn
paWE15TVRNNlEweEtaREpxY0Uxe1ZtTTVlbkpCZFZSR1VHND0iCgkJfQoJfSwKCSJIdHRwSGVhZGVycyI6IHsK
CQkiVXNlcis1BZ2VudCI6ICJEb2NrZXItQ2xpZW50LzE4LjAzLjEtY2UgKGxpbnV4KSIKCX0sCgkizGV0YWNoS2
V5cyI6ICJjdHJsLUAiCn0=
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-0050569e6cfcc
  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, just the same way you create other objects like service or Ingress.

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

```

$ cat .docker/config.json | base64
ewoJImF1dGhzIjogewoJCSJodWIuanVuaXBlcis5uZXQi0iB7CgkJCSJhdXRoIjogIlNrNVFVaTFH
YVdWc1pGVnpaWE15TVRNNlEweEtaREpxY0Uxe1ZtTTVlbkpCZFZSR1VHND0iCgkJfQoJfSwKCSJIdHRwSGVhZGVycyI6IHsK
dHRwSGVhZGVycyI6IHsKCQkiVXNlcis1BZ2VudCI6ICJEb2NrZXItQ2xpZW50LzE4LjAzLjEtY2Ug
KGxpbnV4KSIKCX0sCgkizGV0YWNoS2V5cyI6ICJjdHJsLUAiCn0=

```

then put the base64 encoded value of .docker/config.json file as **data** in below 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 compromised secret.

#### refer `dockerconfigjson` secret in pod: `imagePullSecrets`

after a secret is created, it can be referred by a pod/RC or deployment in order to pull an image from the private registry. there are many ways to refer the secrets. in this section we'll look at using `imagePullSecrets` under pod `spec` to refer the secret.

#### `imagePullSecrets`

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 Juniper CSRX container from 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

```

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
```

behind the scene, 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
```

### 3.6.3. Secret Benefits

as you can see from our test, the Secret objects is created independently of the pods, and inspecting the object `spec` does not print the sensitive information directly on the screen.

Secrets are not written to the disk, but instead it is stored in a `tmpfs` FS 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 apiserver 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.

## 3.7. Service

POD gets instantiated, terminated and moved from one Node to another, in doing so POD changes IP address so how would we keep track of that to get uninterrupted functionalities from pod? Even if the POD isn't moving, how traffic reaches group of PODs via single entity?

the answer for both questions is Kubernetes 'SVC - services'.

Services is an abstraction that defines a logical set of Pods and a policy by which you can access them, you may think of Services as your waiter in a big restaurant, this waiter isn't cooking nor preparing the food but he just abstracts everything happening at the kitchen for you as you deal only with this waiter.

Simply Service is a layer 4 loadbalancer exposes pods functionalities via specific ip and port. The service and pods are linked via labels like RS.

so let's understand different type of services:

- ClusterIP
- NodePort
- LoadBalancer

### 3.7.1. ClusterIP Service

the `ClusterIP` type of service is the simplest one. it is the default mode if the `ServiceType` is not specified. this diagram below illustrates how clusterIP service works:

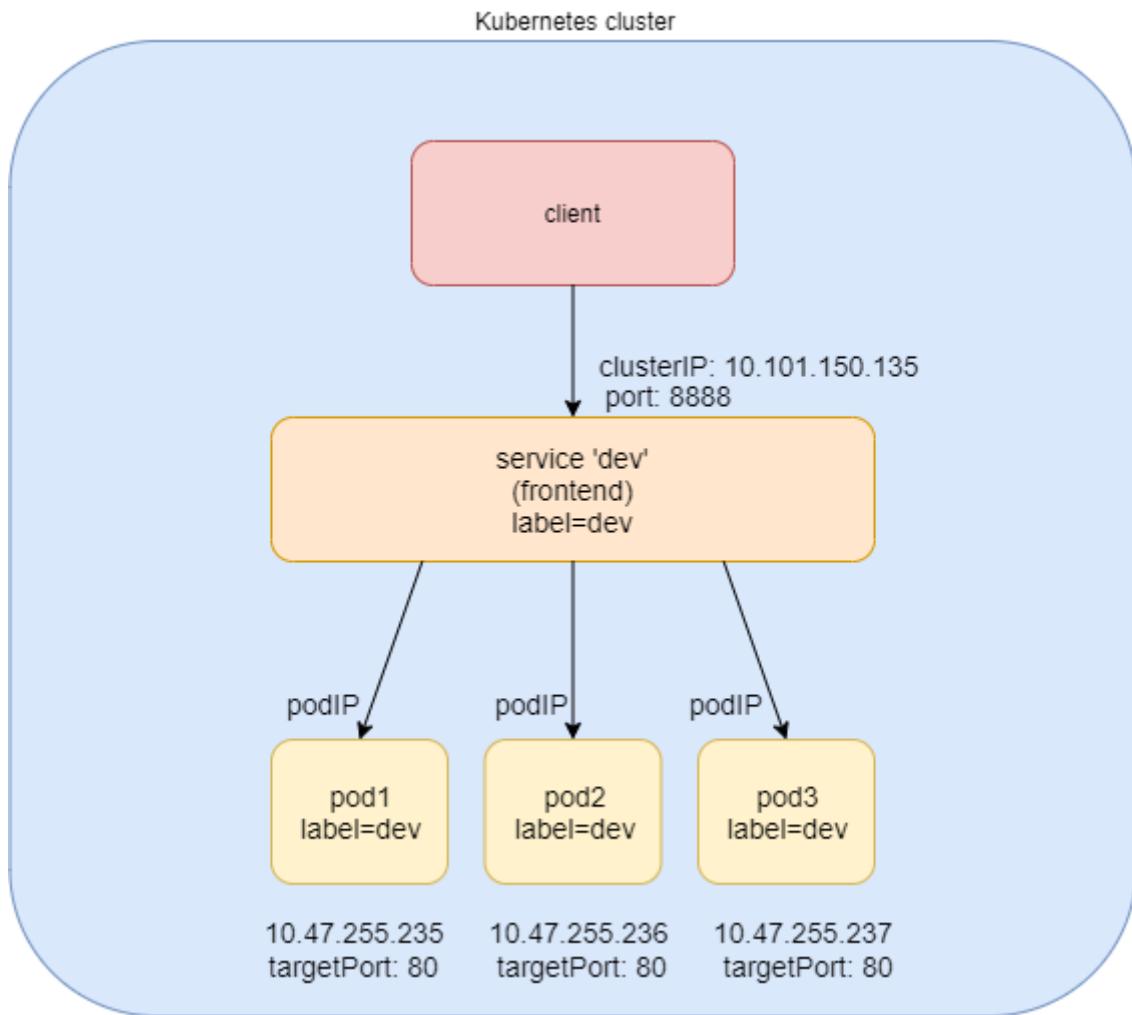


Figure 12. `clusterIP` service

ClusterIP service is exposed on a `clusterIP` and a service port. when client pods need to access the service it sends request toward 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 this service to be only within the cluster. overall by default the ClusterIP is not reachable from external.

#### 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 defined 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 `app: webserver` will be chosen to be the backend pod responding service request.

now generate the service object by `apply` the yaml file:

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

following kubectl commands are commonly used 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.
```

here are some brief summaries about the output:

- the service got a "ClusterIP" or "service IP" of `10.106.176.17` allocated from the service IP pool.
- service port is `8888` as what is defined in yaml.
- by default the protocol type is `TCP` if not declared in yaml file. you can use `protocol: UDP` to declare a UDP service.
- the backend pod can be located with the label selector

the service is created successfully, there is no doubt about it. but there is no pods for the service. the reason is there is no pod with the label matching to the `selector` in the service. now we just need to create the pod with a proper label.

we can define a pod directly, but given the benefits of RC and deployment over pod as we've introduced earlier, use RC or deployment is more practical. later on you will understand this is the right choice. in our example we 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 in defined in our service. `replicas: 1` instruct RC controller to launch only 1 pod at the moment.

```
$ kubectl apply -f deploy-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>
```

immediately the pod is chosen to be the backend.

**TIP**

the example shown use a "equality-based" selector (`-l`) to locate the backend pod, 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 cluserIP service

Now to verify if 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 login to a "client" pod and use `curl` command to send a http request toward the service. you'll see the same "client" pod being used to send request throughout of this book.

the test pod yaml file:

```
#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 cirros pod:

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

**TIP**

the "client" pod, is nothing but the another pod spawned based on the same exact image as what our "webserver" Deployment and its pods do. same as with physical servers and virtual machines, nothing stops a "server" do the client's job.

now test the service with `curl` in `client` pod:

```
$ 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>

</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, we are running a customized pod image that runs a simple web server. the web server is configured in such a way that whenever receiving a request, it will return a simple HTML page with local pod IP and hostname embeded. This way the curl returns something more meaningful in our test.

the returned HTML looks relatively "OK" to read, but there is a way to make it more "eye-friendly":

```
$ 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** we can render a html webpage into text, which is more readable than the HTML page.

now we are convinced our service works. requests to service has been redirected to the correct backend pod, with a pod IP **10.47.255.238**, pod name **webserver-7c7c458cc5-vl6zs**.

## specify a clusterIP

if you want to have a specific 'clusterIP', you can mention it in the spec. Ip address should be in service ip pool.

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
```

### 3.7.2. NodePort Service

NodePort service exposes a service on each node's ip at a static port number. It maps a static port number on each node to a port of the container in the POD, as shown in the figure:

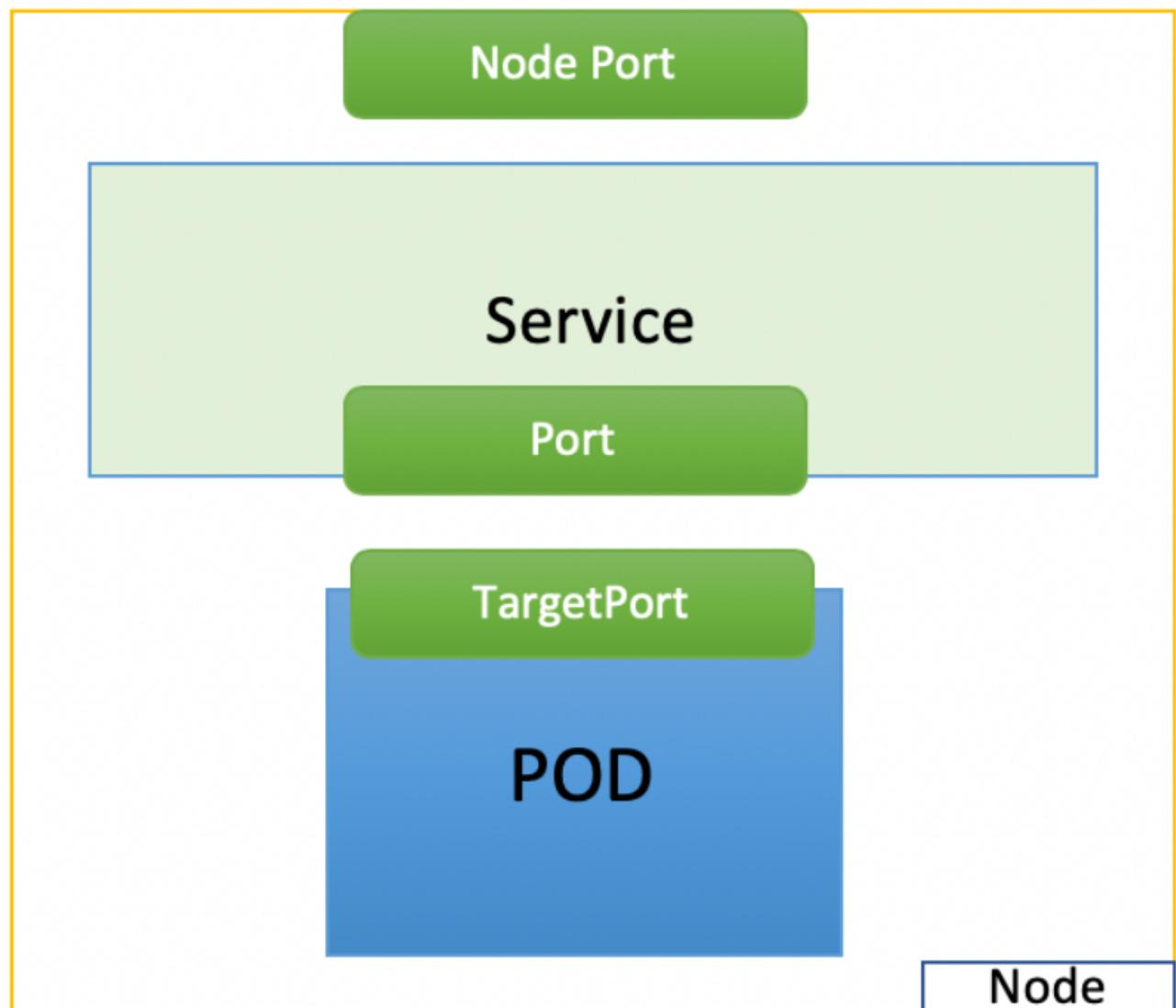


Figure 13. NodePort service

this is a NodePort service yaml file:

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

some highlights in this services YAML file:

- **selector**: the label selector which determine which set of pods targeted by this services, in here any POD with 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 container. in here its port 80, as we are planning to run a web server
- **nodePort**: port on the host of each node in the cluster.

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 to type `NodePort`

- **NodePort**: Kubernetes by default allocate node port from (30000-32767) range if it is not mentioned in the spec. it could be changed using the flag --service-node-port-range. nodePort value also can be set. but it should be in the configurad range.
- **Endpoints**: the podIP and exposed container port. this is where the request toward service IP and service port will be directed to. **10.47.255.252:80** indicates we have created a pod that has a matching label with the service, so it's IP is selected as one of the backend.

**NOTE** for this test make sure there is at least one pod with the label "app: webserver" running. pods created in previous sections are all with this label. recreating the **client** pod suffices if you've removed them already.

Now we can test that by using **curl** command to trigger a http request toward any node IP address.

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

with the power of **NodePort** service, we can access the web server running in the pod from any node via the nodePort 32001:

```
$ 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>
  
</body>
</div>
</html>
```

### 3.7.3. Loadbalancer Service

essentially, a loadBalancer service goes one more step beyond what the NodePort service does. it exposes the Service externally using a cloud provider's loadbalancer. loadbalancer by its nature automatically includes all 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 3 type of services are the `type` value. to reuse the same NodePort service yaml file and create a loadbalancer 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 `loadbalancerIP` is allocated serving 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 an transport layer operation. `loadbalancerIP` and port will be translated to private backend cluster IP and it's `targetPort`. it does not involve any application layer activities. there is no such things like parsing URL, proxy HTTP request, etc like what will happen in HTTP proxying process. because the `loadbalancerIP` is publicly reachable, any Internet host whoever has access to the it (and the service port) can access the service provided by kubernetes cluster.

from Internet host's perspective, when it requests service, it refers this public external `loadbalancerIP` plus service port and the request will reaches the backend pod. the `loadbalancerIP` is acting as a "gateway" between service inside of the cluster and 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 `loadBalancer` 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 is a loadbalancer implemented in loadbalancer service is "vendor-specific". a GCE loadbalancer may work in a totally different way with a AWS loadbalancer. we'll have a detail demonstration about how loadbalancer service works in contrail kubernetes environment in chapter 4.

#### externalIPs

Exposing service outside of the cluster can also be achieved via `externalIPs` option. here is 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 ServiceTypes. `externalIPs` are not managed by Kubernetes and are the responsibility of the cluster administrator.

**NOTE** `externalIPs` are different from `loadbalancerIP`. `loadbalancingIP` is the IP assigned by cluster administrator, while `externalIPs` comes with the loadbalancer created by the cluster that supports it.

### 3.7.4. Kube-Proxy

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

*kube-proxy deployment mode*

`kube-proxy` can be deployed in one of the 3 modes:

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

when the traffic hits the node, it would be forwarded to one of the back end pod via a deployed `kube-proxy` forwarding plane. the detail explanations and comparison of these 3 modes will not be covered by this book, but you can check kubernetes official website for more informations. in chapter4 we'll illustrate how contrail as CNI provider implements the service.

## 3.8. Endpoints

in our 'service' introduction, there is one object that is involved but we haven't explored is 'EP - endpoint'. we've learned it is through label selector that a particular pod or group of pods with matching labels are chosen to be the backend, so that 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 anytime, the "mortal" nature of the pod will most possibly

make the new pods be respawned with new IP address. 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 has their own service implementation update the backends of the service based on the endpoints objects)

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

list the endpoint:

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

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

```
$ kubectl get pod -o wide -l 'app=webserver'  
NAME          READY   STATUS    RESTARTS   AGE     IP           NODE  
... LABELS  
webserver-7c7c458cc5-rjlg  1/1     Running   4        5d17h  10.47.255.252  cent333  
... app=webserver
```

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  
webserver-7c7c458cc5-rjlg  1/1     Running   4        5d17h  10.47.255.252  cent333  
... app=webserver  
webserver-7c7c458cc5-45skv  1/1     Running   0        5s     10.47.255.251  cent222  
... app=webserver  
webserver-7c7c458cc5-m2cp5  1/1     Running   0        5s     10.47.255.250  cent111  
... app=webserver
```

```
$ 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 generated automatically by the kubernetes system whenever a service is created, and at least one pod with matching label exists. Another use case of endpoint, is for a service that has no label selector defined. in that case you can manually map the

service to the network address and port where it's running, by adding an endpoint object manually and you can connect the endpoint with the service. this can be very useful in some scenarios. for example, in your setup you have a backend web server running in a physical server, you still want to integrate it into a kubernetes [Service](#). 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 redirect the service request traffic exactly the same way as if all backend is pod.

## 3.9. Ingress

You've now seen ways of exposing a service to clients outside the cluster. another method is [Ingress](#) in service section, we understand that [service](#) works in transport layer. in reality, you access all services via URLs.

'Ingress' or 'ing' for short is another core concept of kubernetes 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" ie ingress exposes services via HTTP/HTTPS routes. we've learned a lot about kubernetes [service](#) so far, so you understand what will happen after that - the requests will be forwarded to each service's corresponding backend pods.

### 3.9.1. Ingress vs Service

there are similarities between loadbalancer service and ingress. both can expose service to outside of the cluster. but there are some main differences.

*operation layer/level*

[Ingress](#) operates at the application layer of the OSI network model, while [service](#) operates at transport layer only. [Ingress](#) understand the HTTP/HTTPS protocol, service only does forwarding based on IP and port, which means it does not care about the application layer protocol (HTTP/HTTPS) details. Ingress can operate at transport layer. Operating ingress at transoprt layer does not make sense since service does the same unless there is a special reason to do.

*forwarding mode*

Ingress does the application layer "proxy", in pretty much the same way a traditional web loadbalancer does. a typical web loadbalancer 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 loadbalancer. basically It creates two connections each with source (A) and destination (B) machine. Machine A does not even know about the existence of machine B at all. For machine A, 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 an public ip if it is exposed directly to outside of the cluster. when ingress is a front-end to all these services, one public ip would be sufficient which makes life easy for cloud-admin.

### 3.9.2. Ingress Object

before we talk about Ingress object, the best way to get a feel of 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
```

it looks pretty simple. the `spec` defines only one item that is the `rules`. the rules says a `host`, which is "juniper" URL here, may have 2 possible `path` 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 arrives, it proxies the traffic to each URL path's associated backend service. each service, as we've learned this in `service` section, will deliver the request to their corresponding backend path. that's it. actually this is one of the 3 types of Ingress that kubernetes supports today - "simple fan-out Ingress". later we'll introduce the other two types of Ingress.

*about URL, host, path*

term **host** and **path** are used frequently in kubernetes Ingress documentations. **host::** is "fully qualified domain name" of a server. **path**, or **url-path::** is the rest part of the string after the **host** in a URL. in the case of having a **port** in the URL, then it is the strings after the port.

let's 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

**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 **path**. for more details you can read rfc1738, but for the context of this book understanding what we introduce here would suffice.

if you now think kubernetes Ingress is nothing but to define 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 in the high level. the figure below shows the dependency between the 3 kubernetes object: **Ingress**, **service** and **pod**:

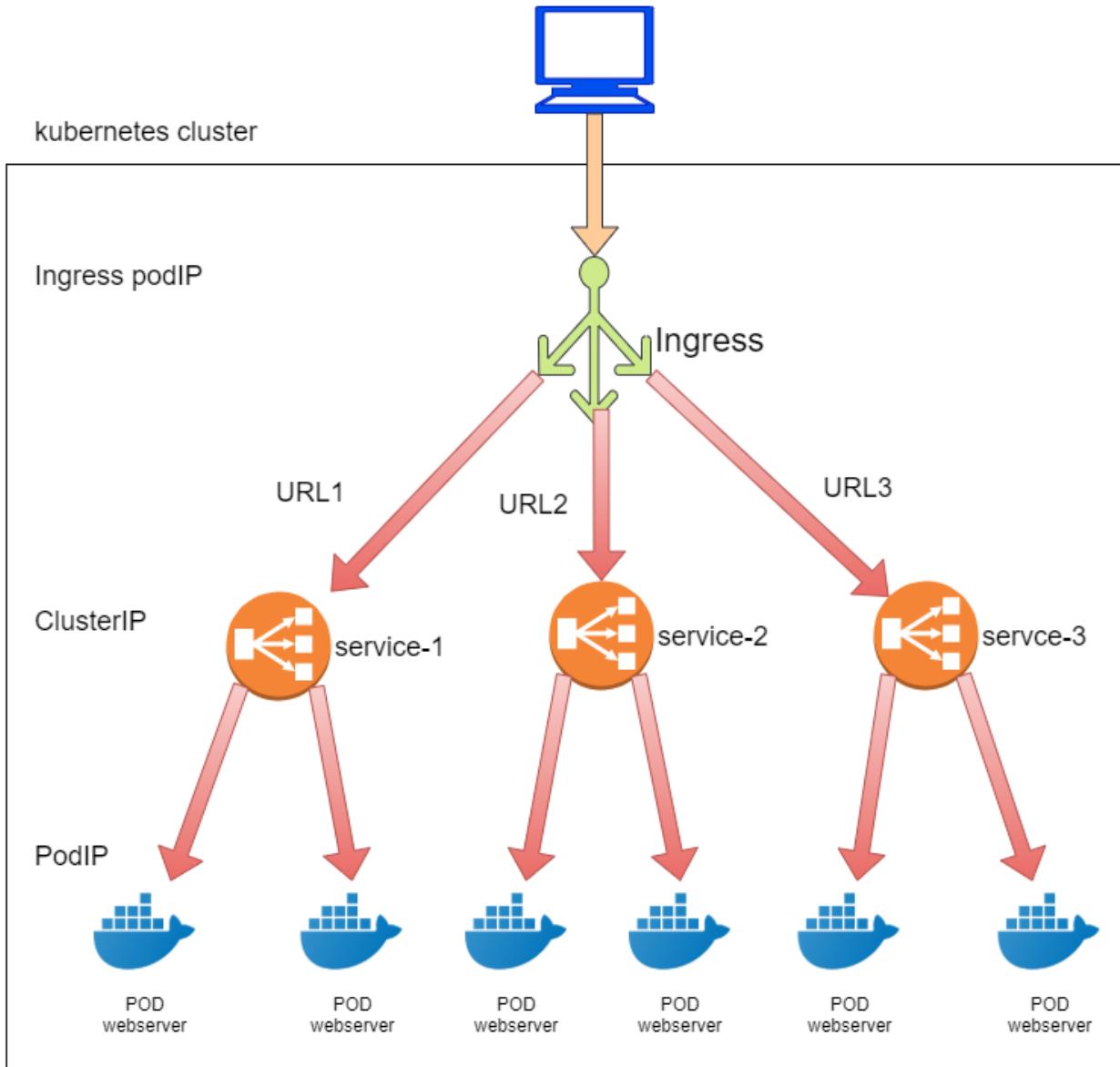


Figure 14. Ingress

in practice there are other things you need to understand. in reality to handle the ingress rules, you need at least another component called [ingress controller](#)

### 3.9.3. Ingress Controller

An ingress controller is responsible for reading the Ingress rules and program the rules into the proxy which does the real work - dispatching traffic based on [host / URL](#).

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

some ingress controller providers:

- 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. `annotation` used in ingress objects will be explained in the "annotation" section.

### 3.9.4. Ingress Examples

there are basically 3 types of ingresses:

- Single Service Ingress
- Simple fanout Ingress
- Name based virtual hosting Ingress

we've looked at the "simple fanout Ingress". now let's also look at yaml file example for the other two type 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 will be exposed to the public, however, it has no `rules` defined, so it does not parse `host` or `path` in the URLs. all requests will goes to one 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've checked this out in the beginning of this section. comparing with **single service** ingress, **simple fanout** ingress is more practical. it is not only able to expose service via a public IP, but 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 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

```

**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 above may not be practical (unless one day the two

domains merge!) but it is good enough to showcase the idea. in the yaml file 2 "hosts" are defined, the "juniperhr" and "junipersales" URL respectively. even though ingress will be allocated with one public IP only, based on the `host` in URL, request toward that same public IP will still be routed to different backend services - that is why it is called a "virtual hosting Ingress". we'll have a very detail case study in chapter 4 about this example.

**NOTE**

it is also possible to merge a "simple fanout" Ingress and a "virtual host" Ingress in one Ingress, in this book we won't cover this topic though.

### 3.9.5. Multiple Ingress Controllers

you can have multiple ingress controllers in one cluster. in that case the cluster needs to know which one to choose. for example, later on 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 "nginx" Ingress controller. we will end up having 2 Ingress controllers in the same cluster with the names are:

- `opencontrail` (default)
- `nginx`

contrail's implementation is the `default` one so you don't have to select it explicitly. to select nginx as Ingress controller, use this annotations `kubernetes.io/ingress.class`:

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

this will make contrail's Ingress controller `opencontrail` to ignore the Ingress configuration.

## 3.10. contrail Network Policy (ch3)

### 3.10.1. network policy introduction

kubernetes networking model requires all pod can access the the other pods by default. we call this a "flat network" sometimes because it follows a "allow-any-any" model - basically a kubernetes pod can reach any other pods **by default**. this makes the design and implementation of kubernetes networking significantly simplified and much more scalable.

**NOTE**

In chapter 4 we'll read more about the requirements that kubernetes enforces on the networking implementation.

on the other hand, with companies large and small rapidly adopting the platform, security has emerged as an important concern. In reality, in many cases certain level of network segmentation methods are required to ensure that only certain pods can talk to each other. that is when kubernetes network policy comes into the picture. a Kubernetes "network policy" defines the access permissions for groups of pods in a way pretty much like 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 that we've learned earlier in this chapter. the role of **NetworkPolicy** object is to define how groups of pods are allowed to communicate with each other. now let's explain the way kubernetes network policy works:

1. initially, in a kubernetes cluster, all pods are non-isolated by default. they works in "allow-any-any" model so anyone can talk to any others.
2. now you apply a network policy named **policy1** to pod A. in policy **policy1**, you define a rule to explicitly allow A to talk to pod B. in this case we will 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, we can call the rules 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 policy **policy1**, it will be enforced by the nature of kubernetes network policy. we can call this an implicit policy
    - the "**deny all**" policy.
  - as for other non-target pods, for example, pod B or C, which are NOT applied with this network policy **policy1**, and not any other network policies either, 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. we can call this another implicit policy - A "**allow all**" policy.
4. assuming you want pod A to also be able to communicate to pod C, you need to update the network policy **policy1** and its rules to **explicit** allow it. in another word, you need to keep updating the "whitelist" to allow more traffic types.

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

- explicit **policy1**: the network policy you defined, with the whitelist rules allowing certain type of traffic for the selected (target) pod.
- an implicit "**deny all**" network policy: deny all other traffic that is not in the whitelist for the target pod.
- an implicit "**allow all**" network policy: allow all other traffic for other non-targeted pods that are not selected by the **policy1** network policy. we'll see "deny all" and "allow all" policies again later in chapter 8.

here are some highlights of 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 compose a "whitelist", each rule describe a certain type of traffic to be allowed. all other traffic that is 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 any network policy, and it will be "affected" by the selecting network policy only. absence of network policy applied on a pod indicates an implicit "allow all" policy to this pod. in another word, if a non-targeted pod continues its "allow-any-any" networking model.

#### *seperation 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 same flow will also be allowed. 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 3 way TCP connection establishment and all data and acknowledgment in both directions.

#### **NOTE**

Network polices are implemented by the network component, so you must be using a network solution which supports network policy. Simply creating the [NetworkPolicy](#) resource without a controller to implement it will have no effect. in our book contrail is such a network components with network policy implemented. in chapter 8, you will see how these network policies works in contrail.

### **3.10.2. network policy definition**

like all other objects in kubernetes, network policy can be defined in a yaml file. let's go ahead to look at an example of it.

#### **NOTE**

this is the same example that we're going to read again 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

```

before explaining it in detail, let's look at the **spec** part of this yaml file since other sections are self-explanatory after you've read yaml file of other objects. the **spec** has the following structure:

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

here we see that a network policy definition yaml file can logically be divided into four sections:

- **podSelector**: this defines the pods selection. it identifies pods where the current Network policy would be applied to.
- **policyTypes**: specify type of policy rules, **Ingress**, **Egress** or both.
- **ingress**: define the ingress policy rules for the target pods
- **egress**: define the egress policy rules for the target pods

next we'll look at each sections 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 the service select its backend pods, the network policy select which pods it will be applied to based on **labels**:

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

here all pods which has the label **app: webserver-dev** is selected to be the "target" pods by the network policy. all of the following contents in **spec** will apply to the target pods only.

### policy types

The second section defines the **policyTypes** for the target pods.

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

policyTypes: - Ingress

it can be either `ingress`, `egress`, or both. both types define specific traffic types in the form of one or more rules, which we'll discuss next.

## policy rules

`ingress` and `egress` section defines the direction of traffic, from the selected target pods's perspective. for example considering the following simplified example:

```
ingress:  
- from:  
  - podSelector:  
    matchLabels:  
      app: client1-dev  
  ports:  
    - protocol: TCP  
      port: 80  
egress:  
- to:  
  - podSelector:  
    matchLabels:  
      app: client2-dev  
  ports:  
    - protocol: TCP  
      port: 8080
```

assuming 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:

- ingress direction: pod `webserver-dev` can accept a TCP session with a destination pod 80, initiated from pod `client1-dev`. this explains why we said kubernetes network policy is "flow" based instead of "packet" based. the TCP connection would not be able to establish if the policy would have been designed on "packet" based, because on receiving the incoming TCP "sync", the returning outgoing TCP "sync-ack" would have been rejected without a matching egress policy.
- egress direction: pod `webserver-dev` can initiate a TCP session with a destination pod 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 has `ports` statement, which will be discussed later.

so you can define multiple rules to allow complex traffic mode 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 endpoint can be identified by different methods:

- **ipBlock**: select pods based on ip address block
- **namespaceSelector**: select pods based on label of namespace (NS), all pods in the matching namespaces will be
- **podSelector**: select pods based on label of pod

**NOTE**

**podSelector** select different things when it is used in different places of a yaml file. previously (under **spec** directly) it selects pods that the network policy applies, which we've called "target" pods. here in a rule (under **from** or **to**), it selects which pods the target pods is communicating with. sometime we can 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 in our 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 which has the label `project: jtac`, or
  - pods which has the label `app: client1-dev` in current namespace (namespace of target pod)
- The egress network point is pod `dbserver-dev`

we'll come to the `ports` part soon.

#### AND vs OR

It is also possible to specify only a few pods from namespaces instead of all pods to communicate with. in our example `podSelector` is used 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 belongs to is those with matching labels with `namespaceSelector`, instead of same as the target pod's namespace.

for example, assuming 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 label **app: client1-dev in the namespace qa**

Please be careful it is totally different than the below definition, which allow 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. **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
```

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

## line by line explanation

after explaining everything, you should find that the policy rules in our example start to makes more sense, so let's look at it again in more detail:

```
podSelector:          ①
  matchLabels:
    app: webserver-dev ②
policyTypes:          ④
  - Ingress           ⑤
  - Egress            ⑥
ingress:              ⑦
  - from:
    - ipBlock:
      cidr: 10.169.25.20/32 ⑩
    - namespaceSelector:
      matchLabels:
        project: jtac ⑬
    - podSelector:
      matchLabels:
        app: client1-dev ⑯
ports:                ⑰
  - protocol: TCP    ⑱
    port: 80          ⑲
egress:
  - to:
    - podSelector:
      matchLabels:
        app: dbserver-dev
ports:
  - protocol: TCP
    port: 80
```

from this definition, we now can understand exactly what the network policy is trying to enforce:

- line 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 only.
- line 4-6: the policy will define rules for both **Ingress** and **Egress** traffic
- line 7-19: **ingress:** section defines the **ingress policy**
  - line 8: **from:** and line 17: **ports**, these two sections defines one **policy rule** in ingress policy.
    - line 9-16: these 8 lines under **from:** section compose an ingress "whitelist":
      1. line 9-10: any incoming data with source IP being 10.169.25.20/32 can access the target pod **webserver-dev**
      2. line 11-13: any pods under namespace **jtac** can access target pod **webserver-dev**
      3. line 14-16: any pod **client1-dev** can access target pod **webserver-dev**
    - line 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 pods will be denied.

- line 20-26: `egress`: section defines the **egress policy**
  - line 21: `to`: and line 24: `ports`, these two sections define one **policy rule** in egress policy.
    1. line 21-24: these 4 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 destination port of 8080 to other pods.

and that is not all.

if you remember in the beginning of this chapter we've 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 (the policy `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.
- a "`allow all`" network policy: allow 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 deeper look at these implicit network policies and their rules in contrail implementation.

### 3.10.3. create network policy

you can create and verify the network policy same way as 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 setup a test environment to verify the effect of this network policy in more detail.

## 3.11. Liveness Probe and Readiness Probe

### 3.11.1. Liveness Probe

What happen if the application in the POD is running but it can't serve its main purpose for whatever reason? also applications that runs for long time might transition to broken states. In all cases the last thing you want have is a call reporting a problem in an application that could be easily fixed with restarting the POD. liveness probes is a Kubernetes features made specially for that. liveness probes sent a pre-defined request to the POD on a regular basis then restart the POD if this request failed. The most commonly used liveness probe is HTTP GET request, but it could also be opening TCP socket or issuing a command

this is a 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 second as specified in "periodSeconds" If that failed the POD would be restarted automatically. you have the

option to specify the path which in here just the main website. also you can send the probe with customized header

```
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
```

let's launch this POD then login to it to terminate the process that handle the httpGet

```
[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 the POD got restarted automatically and in the event it stated the reason for that restart :

```
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:fa2930cb8f4b766e5b335dfa42de510ecd30af6433ceada14cdcaa8de9065d2a
      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. TCP socket probe is similar to the HTTP GET request probes, but it will open 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

```

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

```

### 3.11.2. Readiness Probe

Liveness probe make sure that your POD is in good health, but for some application Liveness alone isn't enough. some application need to load large files before it start. you might think if we set a higher “initialDelaySeconds” value then problem solve. but this is not an efficient way. Readiness probe is solution in here specially with Kubernetes services, as the POD will not receive traffic until it is ready. Whenever the readiness probe fails, the endpoint for the pod would be removed from the service and it would be added back when the readiness probe succeeds. Readiness Probe is configured the same way as 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** its recommended to use both Readiness Probe and Liveness Probe where Liveness probe restart the POD if it failed and Readiness Probe make sure the POD is ready before it gets the traffic

### 3.11.3. Probe Parameters

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

1. **initialDelaySeconds**: Number of seconds after the container has started before liveness or readiness probes are initiated.
2. **periodSeconds**: How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.
3. **timeoutSeconds**: Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.

4. **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.
5. **failureThreshold**: When a Pod starts and the probe fails, Kubernetes will try **failureThreshold** times before giving up. Giving up in case of liveness probe means restarting the Pod. In case of readiness probe the Pod will be marked Unready. Defaults to 3. Minimum value is 1.

HTTP probes have additional parameters that can be set on httpGet.

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

## 3.12. Annotation

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

Another way is Annotations which is a key/value maps that attach non-identifying metadata to objects, Annotation has a lot of use cases such as attaching

- pointers for logging and analytics
- phone number, directory entries and web site
- timestamps, image hashes and registry address
- network, namespaces
- type of ingress controller

here is an example for annotations:

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-demo
  annotations: #<---
    imageregistry: "https://hub.docker.com/"
spec:
  containers:
    - name: nginx
      image: contrailk8sdayone/contrail-webserver
      ports:
        - containerPort: 80
```

# Chapter 4. Chapter 4: Kubernetes and Contrail Integration

This chapter explains contrail's role in the Kubernetes world. We'll start with a section about contrail's Kubernetes integration architecture, where you will learn how Kubernetes objects are handled in contrail. Those objects are NS, pod, service, ingress, network policy and etc. Then we'll look into the implementation of each of them in detail. We also introduce some contrail objects whenever needed. Multiple interface pod is a highlight of contrail's advantages as one of Kubernetes' Network CNI over other implementations, so we cover that also. In the end, we will demonstrate service chain using Juniper CSRX container.

## 4.1. Contrail-Kubernetes Architecture

### 4.1.1. Why Contrail with Kubernetes ?

Now after we have seen the main concepts of Kubernetes in chapter 2 and 3, what could be the gain in adding Contrail to standard Kubernetes deployment?

In brief, Contrail offers common deployment for multiple environments (OpenStack, Kubernetes, etc) as well it enriches Kubernetes networking and security capabilities.

When it comes to deployment for multiple environments, Yes containers is the current trend to build applications, but don't expect everyone to migrate everything from VM to containers that fast (This is not to mention the nested approach where containers are hosted in VM). If we add to the picture workload fully or partially run in the public cloud, we end up feeling the misery for network and security administrators where Kubernetes becomes just one thing to manage Network and security.

Administrator in many organizations manage individual orchestrator/manager for each environment. OpenStack or VMware NSX for VM, Kubernetes or Mesos for Containers, AWS console. And here what contrail could put the network and security administrators out of their misery is it provides dynamic end-to-end networking policy and control for any cloud, any workload, and any deployment.

From a single user interface contrail translates abstract workflows into specific policies, simplifying the orchestration of virtual overlay connectivity across all environments by building and securing virtual networks that connect BMS, VM and Containers located in private or public cloud.

A very common way to deploy Kubernetes is to launch its POD in VMs orchestrated by OpenStack. This is one of the many use cases of contrail doing its magic.

In this book we won't cover contrail integration with other environments as we focus only in Kubernetes. But any feature that we explain in here could be extended for other environments.

What we mean by contrail enriching standard Kubernetes deployment?

Even though Kubernetes does not provide the networking, it imposes fundamental requirements on the networking implementation and it is taken care by all CNI ("Container Network Interface")

providers. contrail is one of the cni providers. you can refer to <https://kubernetes.io/docs/concepts/cluster-administration/networking/> for more available CNI providers.

kubernets has defined some fundamental requirements in networking implementation:

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

with these requirements to all CNI plugins implementations, Kubernetes offers flat network connectivity with some security feature confined in a cluster, but Contrail could offer on top of that:

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

In this chapter we will cover some of these aspects, but first let's talk about Kubernetes/contrail architecture and the object mapping

#### 4.1.2. Contrail-Kube-Manager

A new components of contrail has been added called **contrail-kube-manager**, abbreviated as **KM**. what it does basically is to watch kubernetes apiserver for interested kubernetes resources, and translates into Contrail controller object. the following figure illustrates the basic work flow:

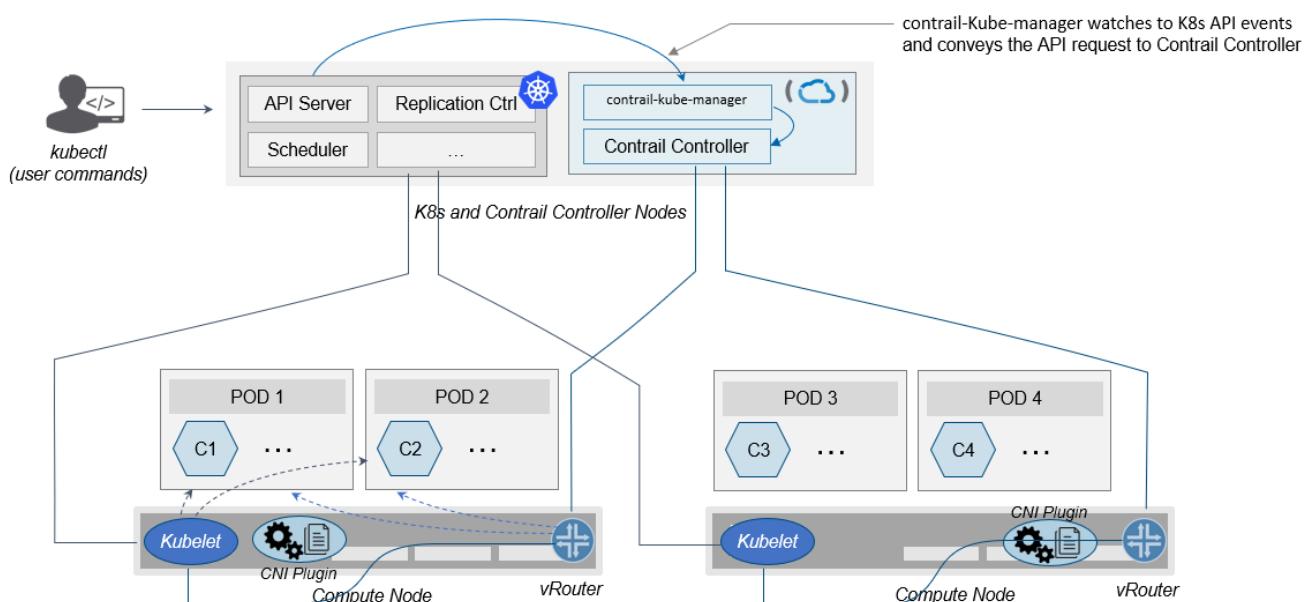


Figure 15. contrail kubernetes architecture

### 4.1.3. Kubernetes to Contrail Object Mapping

So not much of change of the regular contrail that we have seen before and all of that is happening behind the scene. what we have to be aware of it before dealing with Kubernetes/contrail is the object mapping. because contrail is single interface managing multiple environments - as explained before – each environment has its own acronym and terms hence the need for this mapping, which will be done by a plugin. in kubernetes `contrail-kube-manager` does this.

**NOTE** | contrail has specific plugins for each environments/orchestrator.

For example, Namespace in Kubernetes are intended for segmentation between multiple teams, or projects as if we are creating virtual cluster. In contrail the similar concept would be named as project so when you create a namespace in Kubernetes it will automatically create an equivalent project in contrail. more on that will come later on for now kindly make yourself familiar with this list of object mapping

## Kubernetes Objects



## Contrail Objects

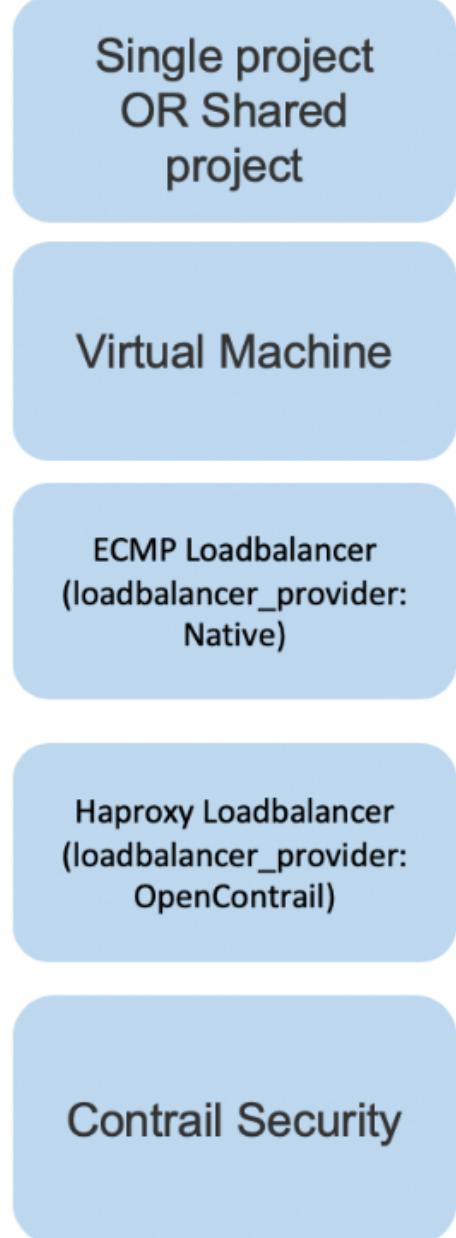


Figure 16. contrail kubernetes object mapping

## 4.2. Contrail Lab environment

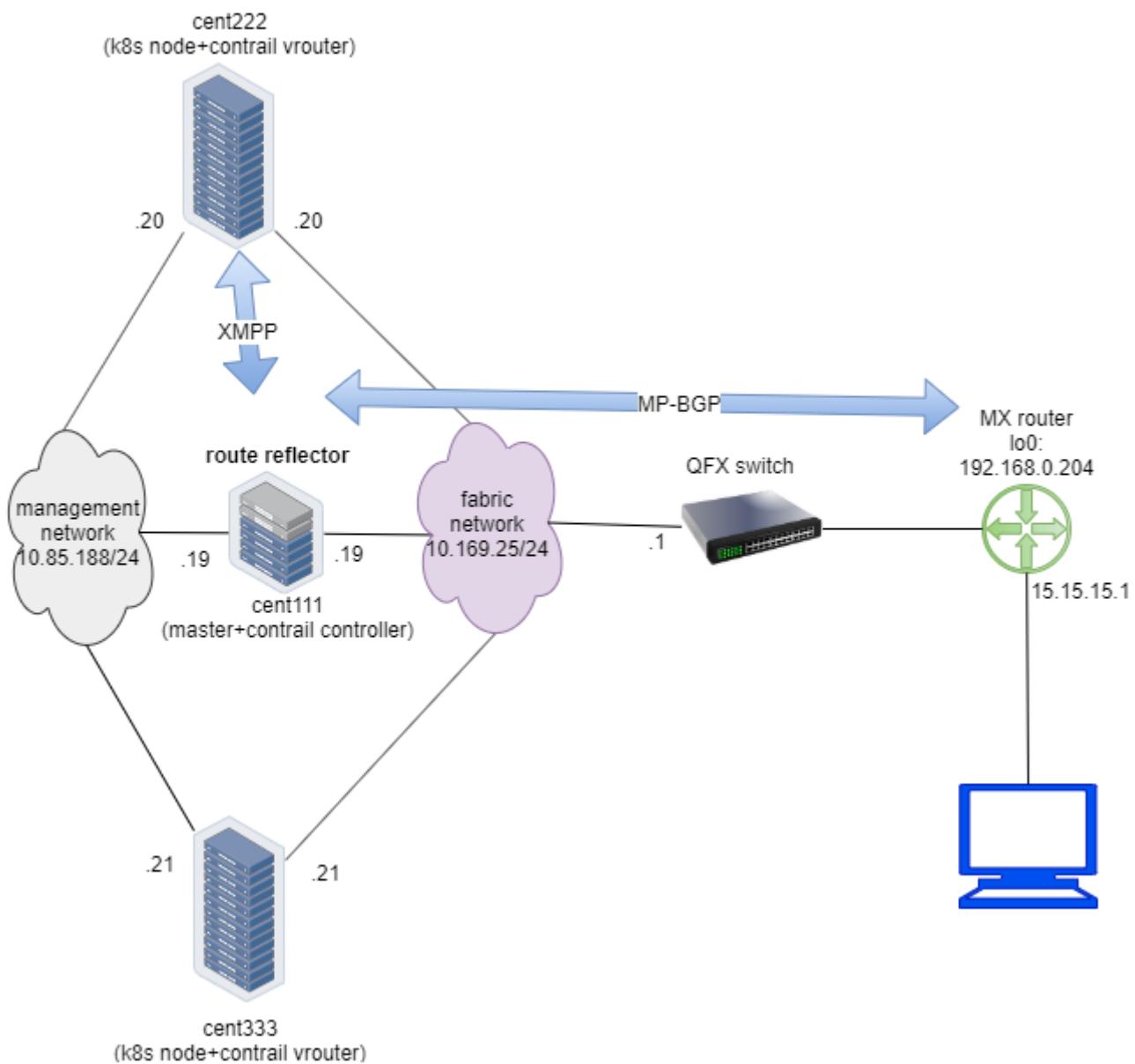
### 4.2.1. Contrail Setup

before starting our investigation, let's look at our setup. in this book we build 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 runs as an Internet host machine

the digaram is here:



#### TIP

To minimize the resource utilization, all "servers" are actually centos virtual machines created by vmware ESXI hypervisor running in one physical HP server.

in appendix you will find all details about the setup. the prerequisites, software/hardware specifications, sample configuration files, and installation steps. following the steps you will be able to build a same setup in your lab.

#### 4.2.2. Contrail Command

Before getting into deeper, we just want to introduce briefly contrail-command(CC) which is the new user interface (UI) available from contrail 5.0.1. throughout this book we use both CC and old UI to demonstrate most of lab studies. just keep in mind that in the future CC will be the only UI and the "legacy" one will be deprecated.

detail information about CC is available in Juniper documentation website so we won't elaborate it

here. to access CC use this URL in web browser: <https://Contrail-Command-Server-IP-Address:9091> the "Contrail-Command-Server" can be the same or different server as kubernetes master or contrail controller node. in our setup we've installed them in same server.

in CC, the functions and settings are groups in a main menu". it is also the entry point from where you can navigate through different functions.

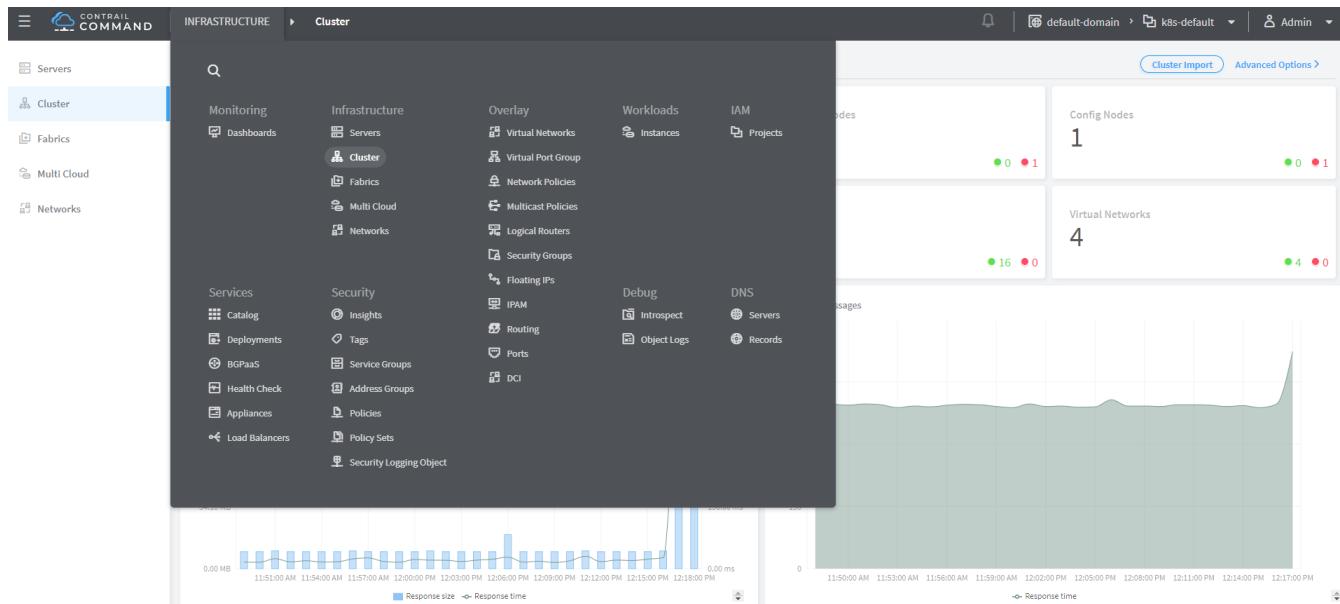


Figure 17. contrail command main menu

in order to get this menu, click on group name right next to the "contrail command" logo on the upper left of the UI. in the above screen capture that group is "Infrastructure", but regardless it can be any group, just click it and you will get the main menu, then from there you can select and jump into all other settings.

Again our focus is not CC. we are trying to give base insight about CC which would be helpful for our primary goal of the book.

## 4.3. Contrail Namespaces and Isolation

In chapter3 you've read about **namespace** or **NS** in kubernetes. in the beginning of this chapter we've mentioned object mappings between kubernetes and contrail. in this section we'll see how NS works in contrail environments and how contrail extends the feature.

one analogy we've given when introducing **namespace** concept is openstack **project**, or **tenant**. that is exactly how contrail is looking at it. whenever a new **namespace** object is created, **contrail-kube-manager** (KM) gets noticed 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 NS or project name. the default kubernetes cluster name is **k8s**. so if you create a kubernetes NS **ns-user-1**, **k8s-ns-user-1** project will be created in contrail and you can see the same in the contrail GUI.

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	-	caacfaf32-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
<b>k8s-ns-user-1</b>	-	86bf8810-ad4d-45d1-aa6b-15c74d5f7809	4312d0d5-f8e5-4845-94f7-e2e8c5b1074d	NO

Figure 18. contrail command: projects

the kubernetes `cluster name` is configurable, during deployment process only. if you don't configure it `k8s` will be the default. once the cluster is created, the name can not be changed anymore. to view the `cluster name`, go to `contrail-kube-manager` (KM) docker and check its the configuration file.

Example 13. to locate the KM docker container

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

Example 14. to login to the KM container

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

Example 15. find the `cluster_name` option

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

#### NOTE

in the rest part of this book we will refer all these terms `namespace`, `NS`, `tenant`, `project` interchangeably.

### 4.3.1. Non-Isolated NS

you are aware that kubernetes basic networking requirement is a "flat"/"NATless" network - any pod can talk to any pod in any namespace, any cnf providers should 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 namespace, there should be a **common** VRF (virtual routing and forwarding table) or RI (routing instance). in contrail kubernetes environment, two "default" VNs are pre-configured in k8s default NS, for pod and service respectively. correspondingly there are 2 VRFs each with same name as their correspondingly VN.

the name of the two VNs/VRFs are in this format:

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

so for **default** NS with a default cluster name **k8s**, the two VN/VRF names will become:

- **k8s-default-pod-network**: pod VN/VRF, with the default subnet **10.32.0.0/12**
- **k8s-default-service-network**: service VN/VRF, 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 2 default VNs are **shared** between all of the "non-isolated" namespaces. what that means is, they will be available for any new non-isolated NS that you create, implicitly. that is why pods from all non-isolated NS including default NS can talk to each other.

on the other hand, any VNs that you create will be isolated with other VNs, regardless of same or different NS. communication between pods in two different VNs requires contrail network policy.

**NOTE** later when you read about kubernetes **service**, you may wonder why packets destined service VN/VRF can reach the backend pod in pod VN/VRF? the answer is also contrail network policy. by default contrail network policy is enabled between service and pod network which allows packets arriving service VN/VRF to reach the pod, and vice versa.

for the isolated NS, however, it will be a different story.

### 4.3.2. Isolated NS

in contrast, "isolated" namespace, will have its own default pod-network and service-network, accordingly two new VRFs 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, by default it is isolated with other NS. pods launched in isolated NS can only talk to service and pods on the same namespace. Additional configurations, e.g. policy, is required to make the pod being able to reach the network outside of current namespace.

to illustrate this concept let's take an example. suppose you have 3 namespaces, the **default** NS and two user NS: **ns-non-isolated** and **ns-isolated**. in each NS you create one user VN: **vn-left-1**. you will end up to have following VN/VRFs in contrail:

*NS default*

- 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

*NS ns-non-isolated*

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

*NS ns-isolated*

- 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 name is mentioned in the FQDN format. In contrail domain is the top-level object, followed by project/tenant and followed by virtual-networks.

this can be illustrated in below diagram:

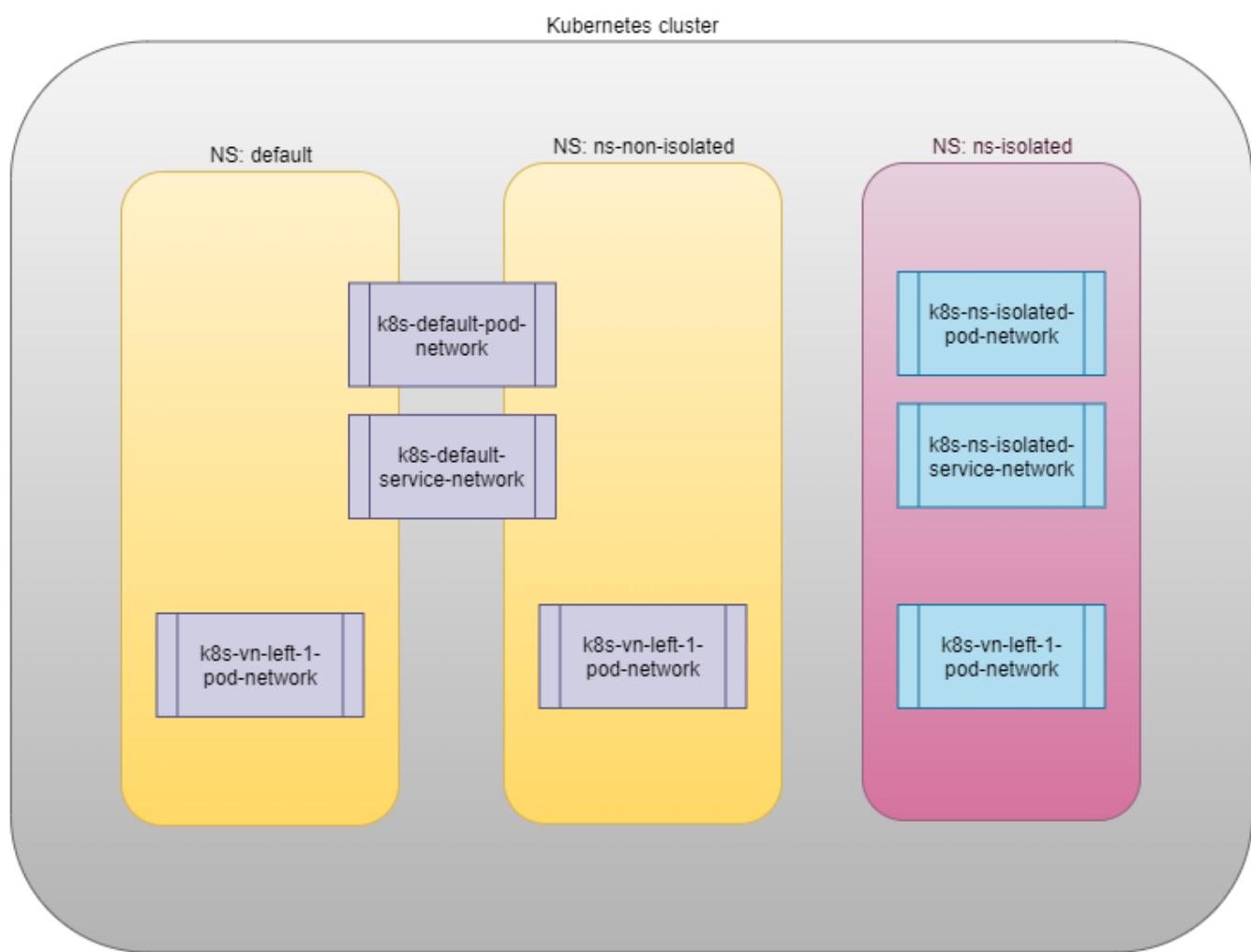


Figure 19. NS and VN

here is the yaml file to create an isolated namespace:

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

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 something additional comparing to standard (non-isolated) k8s namespace, the value of **true** indicates this is an isolated NS:

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

this part of the definition is Juniper's extension. **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 correponding routing instances(one for pod and one for service) instead of using the default ns routing instances for the isolated namespace. fundamentally that is how the "isolation" is implemented.

in the following sections we'll verify how the routing isolation works.

#### 4.3.3. Pods Communication across NS

create a non-isolated namespace and an isolated namespace:

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

#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 NS and the default NS, create a webserver deployment to launch a webserver pod:

```

#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

```

```

$ 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 3 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 non-isolated NS (`default` NS) toward isolated NS does not pass through. what about traffic within the same isolated NS?

with the power of the `deployment` we can quickly test it out: in isolated NS `ns-isolated`, clone one more pod by `scale` the deployment with `replicas=2` and ping between the 2 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 NS
- traffic is isolated between an isolated NS and all other tenant in the cluster
- traffic is not isolated in same NS

**NOTE**

pod-level isolation can be achieved via kubernetes network policy, or security groups in contrail. this will be covered later in this chapter.

## 4.4. Contrail Floating IP

### 4.4.1. Overlay Internet Access

we've discussed and tested the communication between pods in the same or different NS. so far we've only tested it **inside** of the same cluster. what about communication with devices **outside** of the cluster? you may already know that in traditional (openstack) contrail environment, there are many ways for the overlay entities (typically a VM) to access the Internet, the 3 frequently used methods among them are:

- floating IP
- fabric SNAT
- logical router

the preferred kubernetes solution to expose any service is via **service** and **Ingress** objects which you've read about and got the idea in chapter 3. in contrail kubernetes environment, floating IP is used in the service and Ingress implementation to expose them to outside of the cluster. later in this chapter we'll have a very detail discussion for each of these two objects. before that, in this section, we'll review the "floating IP" basis and look at how it works with kubernetes.

**NOTE**

**fabric SNAT** and **logical router** are used by overlay workloads(VM and POD) to reach the internet and the reverse direction is not possible. **floating IP** however, supports both direction - you can configure it to support ingress traffic, egress traffic, or both and default is bi-direction. in this book we focus on **floating IP** only. you can refer contrail documents for detail information about fabric SNAT and logical router.

### 4.4.2. Floating IP and FIP Pool

**floating IP**, or **FIP** for short, is a "traditional" concept that contrail supports since very early releases. Essentially it is 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 reverse direction. Eventually both VM and Internet host can talk to each other, and both can initiate the communication.

**NOTE**

vrouter is a contrail forwarding plane resides in each compute node handles workloads traffic

the figure below illustrates the basic work flow of FIP:

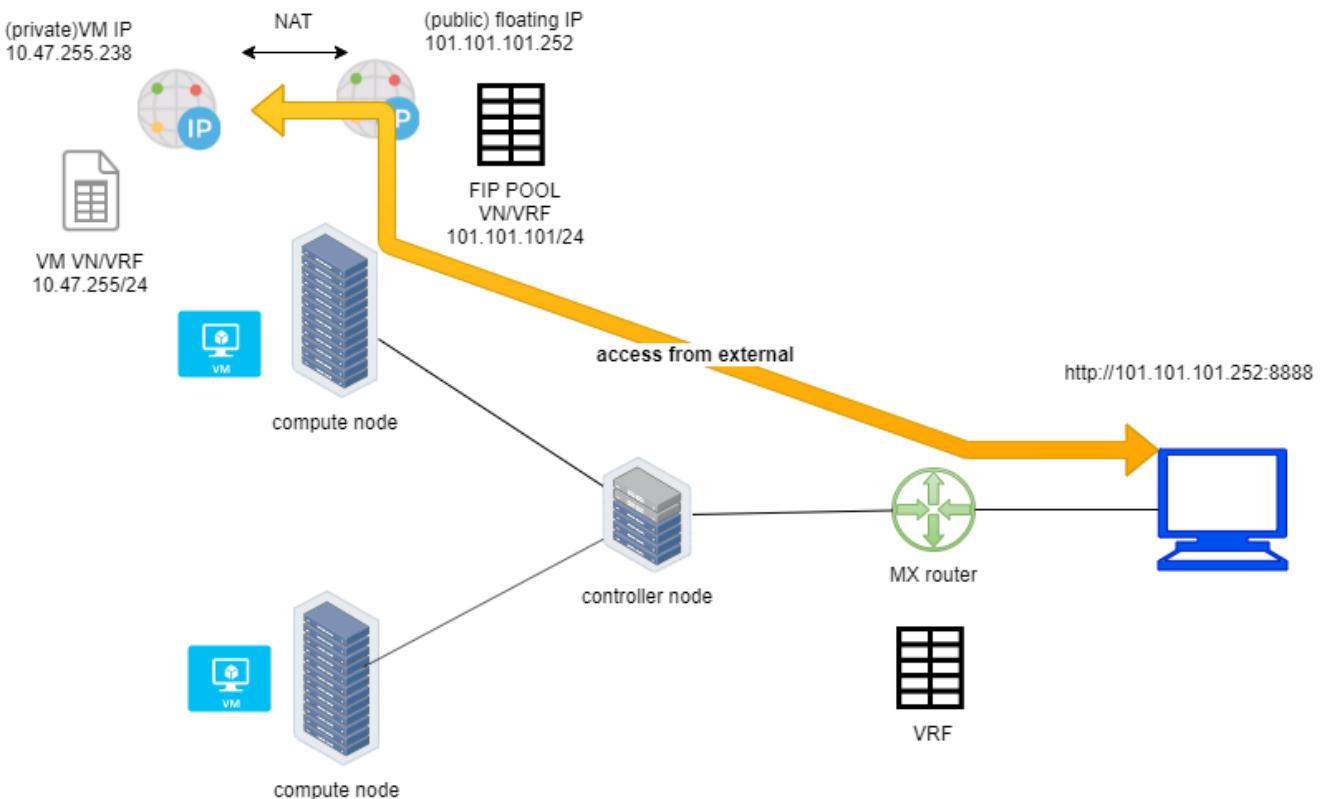


Figure 20. Floating IP

here are some highlights regarding FIP:

- a FIP is associated with a VM's **port**, or a **VMI** (Virtual Machine Interface).
- a FIP is allocated from a **FIP pool**
- a FIP 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 gateway routers 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 will be able to communicate with the FIP.

Regarding the FIP concept and role, there is nothing new in contrail kubernetes environment. But the usage of floating IP has been extended in kubernetes **service** and **ingress** object implementation, and it plays an important role for accessing toward kubernetes **service** and **ingress** from external. you can check later sections in this chapter for more details on this.

## Create FIP Pool

creating a FIP pool is a 3 steps process:

- create a public FIP-VN,
- set **RT** (route-target) for the VN so it can be advertised and imported into the gateway router's VRF.
- create a FIP pool based on the public FIP-VN

again this is nothing new but the same steps as with other contrail environment without

kubernetes. however, as you've learned in previous section, with kubernetes integration a FIP-VN can now be created in a "kubernetes style":

*Example 16. create a public FIP-VN 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 unchanged

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

set the RT

if you need the FIP to be reachable from Internet through gateway router, you'll need to set a route-target to make the VN prefix getting imported in the gateway router's VRF table. this step is necessary whenever Internet access is required.

The screenshot shows the Contrail Command interface for editing a virtual network. The left sidebar lists various network-related options like Virtual Networks, Virtual Port Group, Network Policies, Multicast Policies, Logical Routers, Security Groups, Floating IPs, IPAM, Routing, Ports, and DCI. The main area has tabs for Network, Tags, and Permissions. Under the Network tab, there's a section for 'Routing, Bridging and Policies'. A red box highlights the 'Route Target(s)' section, which contains fields for ASN (set to 500) and Target (set to 500). There are also '+ Add' buttons for Export Route Target(s) and Import Route Target(s).

Figure 21. contrail command: setting RT

**NOTE** the UI navigation path to set RT is: contrail command(CC): main-menu > Overlay > "Virtual Networks" > k8s-vn-ns-default-pod-network > Edit > "Routing, Bridging and Policies"

*create a FIP pool based on the public VN*

this is the final step. from contrail command UI, Create a floating IP pool based on the public VN:

The screenshot shows the Contrail Command interface for creating a floating IP pool. The left sidebar has 'Floating IPs' selected. The main area shows a list of 'Floating IP Pools' with one entry named 'pool-ns-default'. A modal dialog titled 'Edit Floating IP Pool' is open, showing fields for 'Name' (set to 'pool-ns-default') and 'Network' (set to 'k8s-vn-ns-default-pod...'). There is also a 'Description' field and 'Cancel' and 'Save' buttons at the bottom.

Figure 22. contrail command: create a FIP pool

**NOTE** the UI navigation path for this setting is: contrail-command: main-menu > Overlay > Floating IP > Create

**TIP** in contrail UI, you can also set the "external" flag in VN "Advanced" options so that a FIP pool named "public" will automatically be created.

## FIP Pool Scope

there are different ways you can refer an floating IP pool in contrail kubernetes environment, and correspondingly the scope of the pools will also be different. here are 3 possible levels with descending priority:

- object specific
- Namespace level
- global level

### *object specific*

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

### *NS level*

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

### *global level*

the scope of the "global" level pool will be the whole cluster. objects in any namespaces can use the "global" FIP pool.

you can combine all 3 methods to take advantages of the flexibility. here is a practical example:

- define a global pool `pool-global-default`, so any objects in a NS that has no NS-level or object-level pool defined, will get a FIP from this pool
- for NS `dev`, define a FIP pool `pool-dev`, so all objects created in NS `dev` will by default get FIP from `pool-dev`
- for NS `sales`, define a FIP pool `pool-sales`, so all objects created in NS `sales` will by default get FIP from `pool-sales`
- for NS `test-only`, do NOT define any NS level pool, so by default objects created in it will get FIP from the `pool-global-default`
- when a service `dev-websevice` in NS `dev` needs a FIP from `pool-sales` instead of `pool-dev`, specify `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 FIP Pool

let's first take a look at the object-specific FIP pool. here is an example:

```
#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 FIP from pool `pool-public-1`, which is created based on VN `vn-public-1` under current project `k8s-ns-user-1`. the corresponding kubernetes NS is `ns-user-1`. since object level FIP pool is assigned for this specific object only, with this method each new object needs to be assigned a FIP pool explicitly.

### NS FIP Pool

the next FIP pool scope is in NS level. each NS can define its own FIP pool. same way as kubernetes annotations object is used to give a subnet to a VN, it is also used to specify a FIP pool. the yaml file looks:

```
#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
```

in this example, NS `ns-user-1` is given a NS level FIP pool named `pool-ns-default`, and the corresponding VN is `vn-ns-default`. once the NS `ns-user-1` is created with this yaml file, any new service which requires an FIP, if not created with the object-specific pool name in its yaml file, will get a FIP allocated from this pool. In practice, most NS (especially those isolated NS) will need its own NS default pool so you will see this type of configuration very often in field.

## Global FIP pool

to specify a global level FIP pool, you need to give the full 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 '/etc/contrail/common\_kubemanager.env` file in master node:

```
#/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 following line in it:

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

it reads: the global FIP pool is called `pool-global-default`, and it is defined based on a VN `vn-global-default` under project `k8s-ns-user-1`. which 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 FIP pool is specified for the cluster.

**NOTE**

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

#### 4.4.3. FIP for Pods

once FIP pool is created and available, an FIP can be allocated from the FIP pool for the pods that requires one. this can be done by associating an FIP to a VMI (VM or pod interface),

you can manually create a FIP out of a FIP pool in contrail UI, and then associate it with a pod VMI.

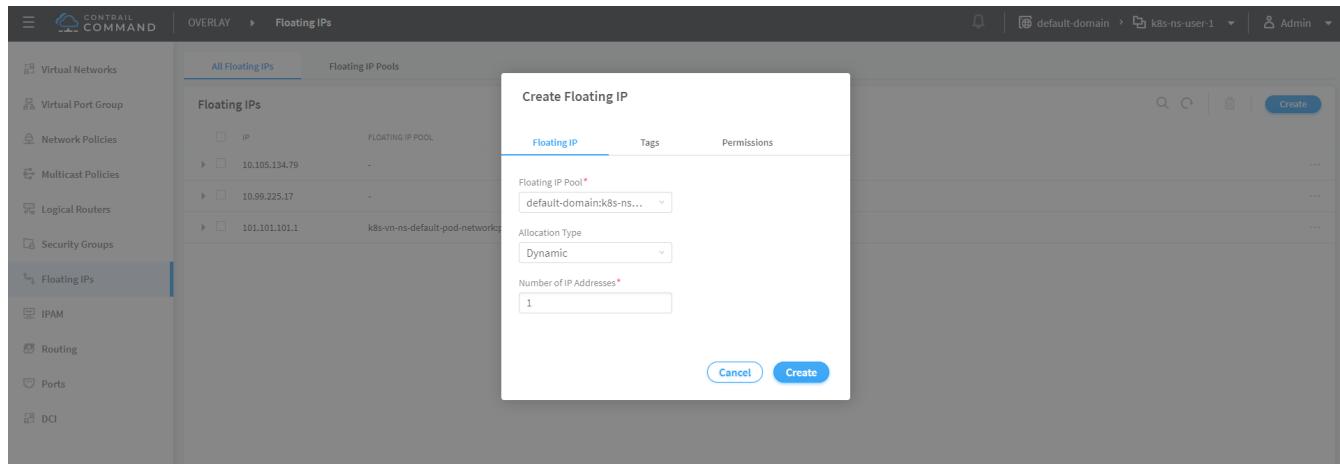


Figure 23. create FIP

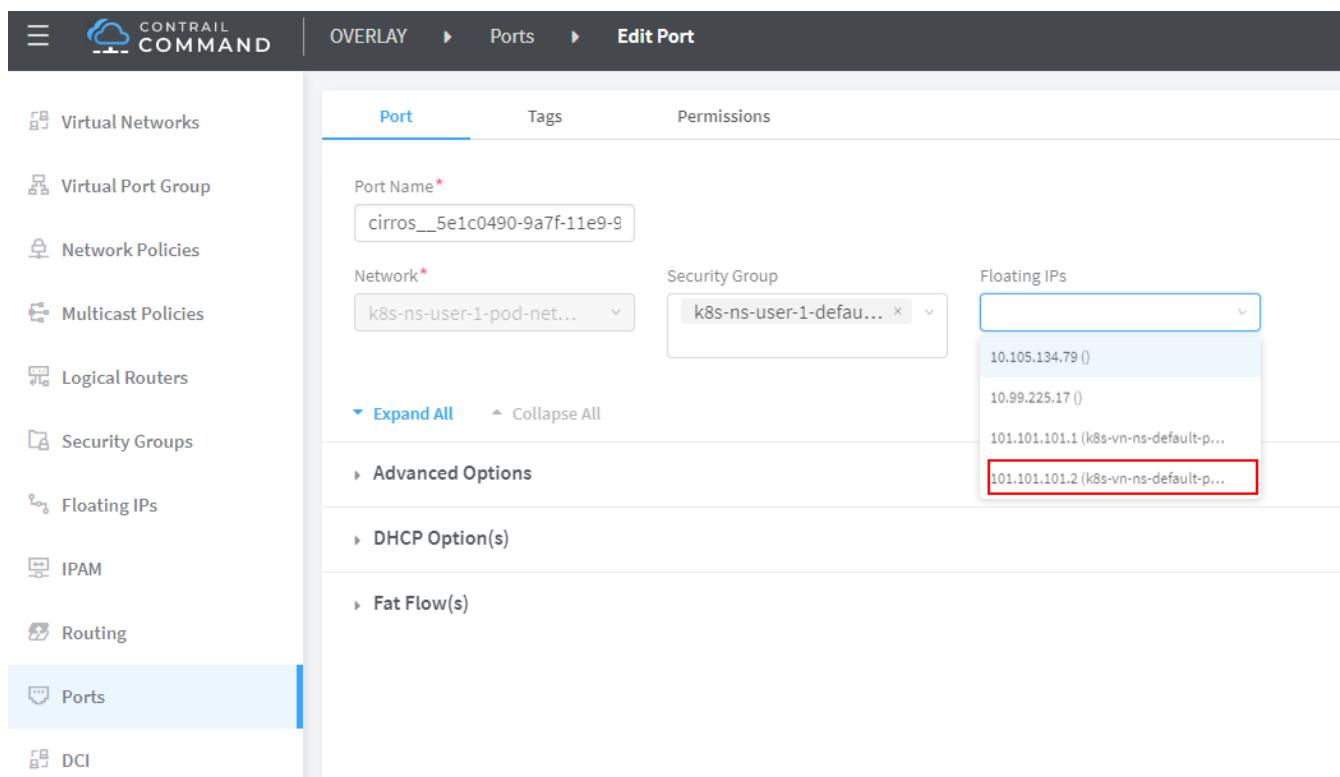


Figure 24. associate a FIP in a pod interface

**NOTE**

make sure the FIP pool is shared to the project where FIP is going to be created.

#### 4.4.4. Advertising FIP

once a FIP is associated to a pod interface, it will be advertised to the MP-BGP peers, which are typically gateway routers.

following screenshot shows how to add/edit a BGP peer.

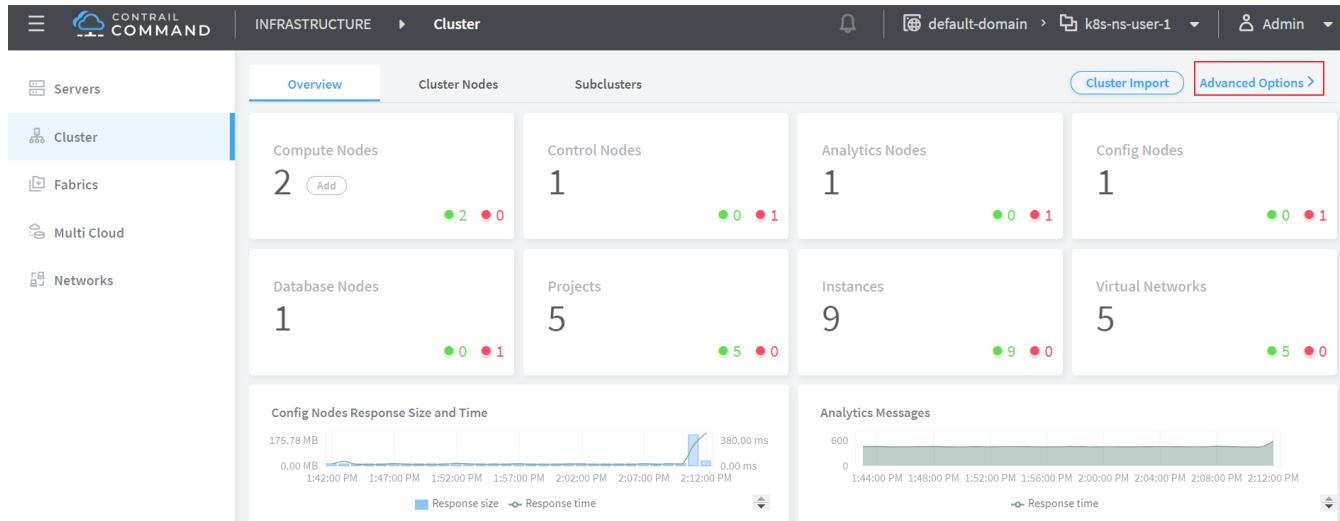


Figure 25. contrail command: select "main-menu" > INFRASTRUCTURE: "Cluster" > "Advanced Options"

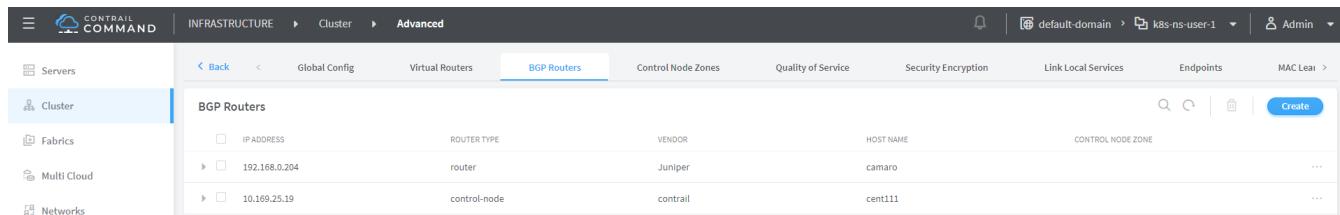


Figure 26. contrail command: select "BGP router" > "create"

Figure 27. edit BGP peer parameters

input all the BGP peer information, don't forget to associate the controller(s), which is shown next:

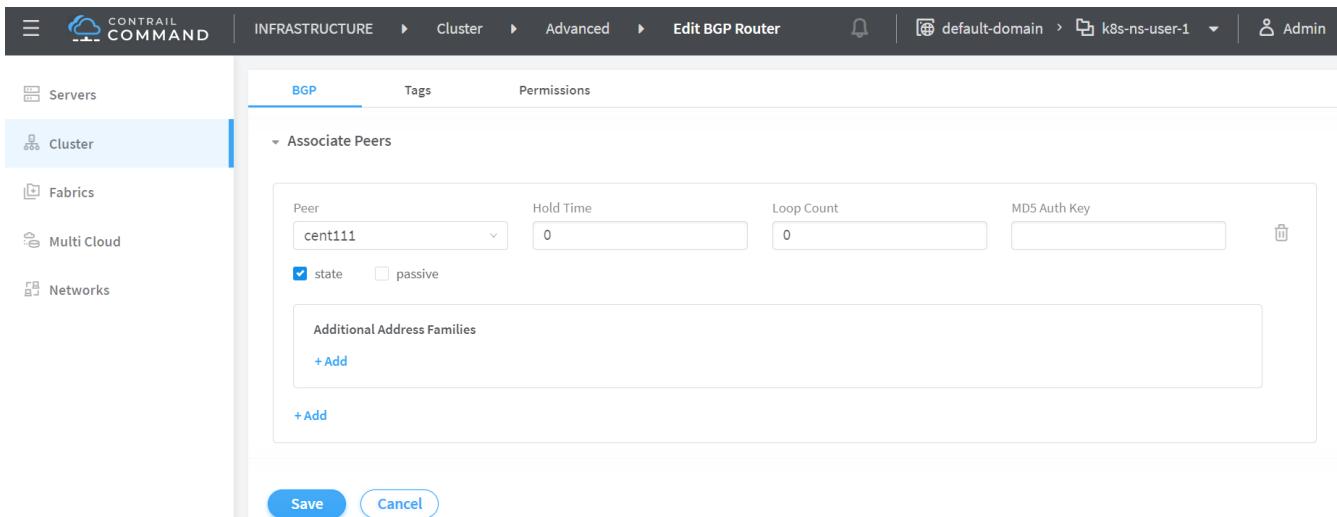


Figure 28. associate the peer to a controller

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.

The screenshot shows the Contrail Command interface with the following details:

- Top Bar:** CONTRAIL COMMAND, INFRASTRUCTURE > Cluster > Advanced, default-domain > ns-user-1.
- Current View:** BGP Routers tab selected.
- BGP Routers Table:**

	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	
- Actions:** Create (button), Edit (button over the second row), ... (dropdown menu over the second row).

Figure 29. a new BGP router in the BGP router list

now we've added a peer BGP router as type "router". for local BGP speaker which is with type "control-node", we just need to double check the parameters by clicking **edit** button. in our test we want to build MP-IBGP neighborship between contrail controller and gateway router, so we make sure the ASN and "Address Families" matches on both end.

Figure 30. contrail controller BGP parameters: ASN

now you can check BGP neighborship status in 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, that is the time we'll see that the FIP assigned to the kubernetes object is advertised by 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 same command tells more: the FIP route is reflected from the contrail controller, but "Protocol next hop" being the compute node ([10.169.25.20](#)) indicates that the FIP is assigned to a compute node. one entity currently running in that compute node own the FIP.

```
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 make 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:64:0000008000000000 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 GRE outer IP header, so the "tunnel" is built from current gateway router whose BGP local address is **192.168.0.204**, to remote node **10.169.25.20**, in this case it's one of the contrail compute nodes.

the FIP advertisement process is illustrated in this figure below:

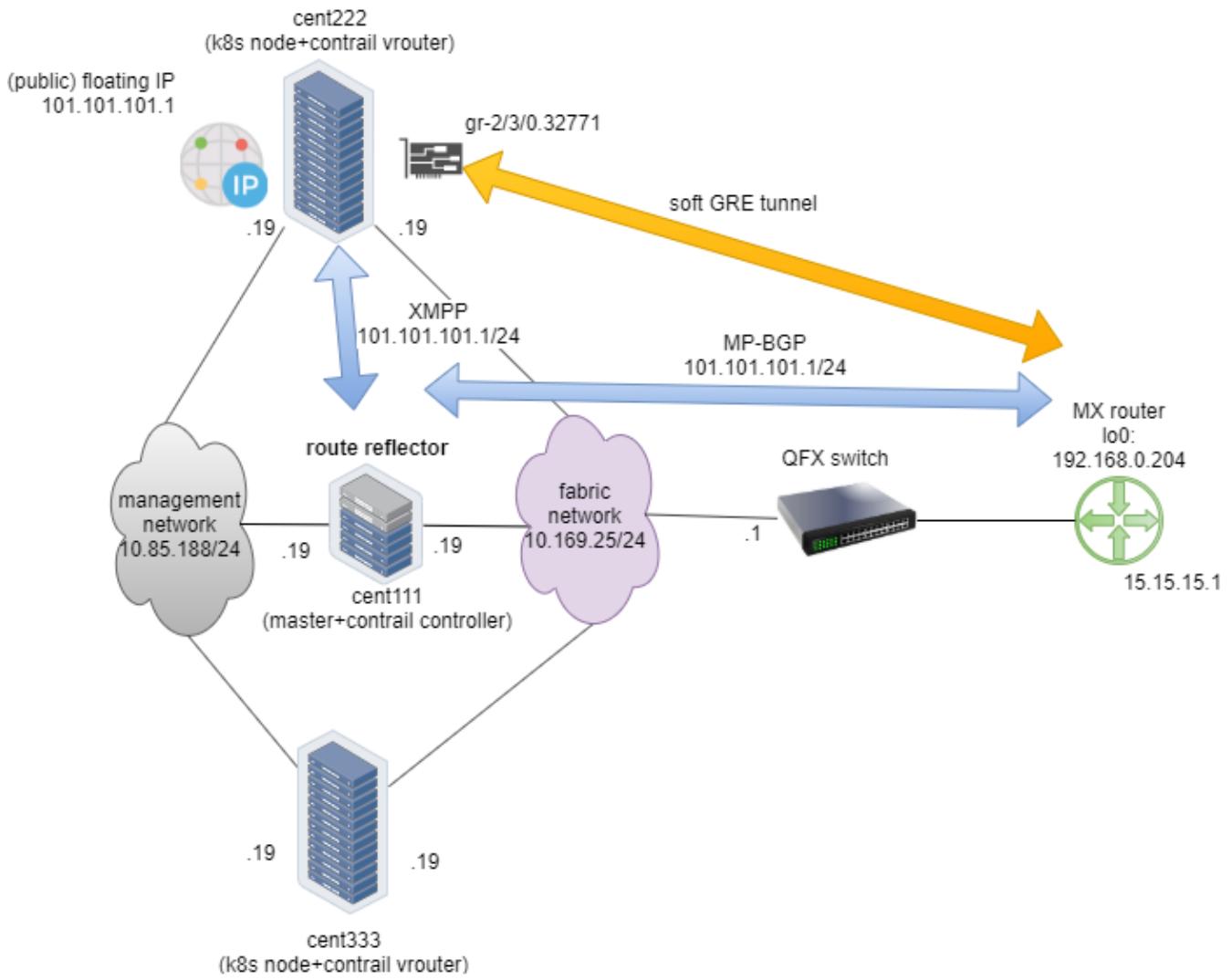


Figure 31. FIP advertisement

## 4.5. summarization

in this section we created the following objects:

1. NS: `ns-user-1`
2. FIP-VN: `vn-ns-default`
3. FIP-pool: `pool-ns-default`

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

if you have yaml files ready for the NS and FIP-VN, to create these object:

```
$ 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
```

**NOTE**

**TIP**

the FIP-pool needs to be created seperately in contrail UI. refer "contrail floating IP" section for the details.

with these objects we now have a NS associated with a FIP pool. from inside of this NS we can proceed to create and study more other kubernetes objects. Next we'll look at [Service](#).

**NOTE**

In this book, all tests that we are going to demonstrate in [service](#) and [Ingress](#) section will be created under this [ns-user-1](#) NS.

# Chapter 5. chapter 5: Contrail Services

in this chapter, we look at kubernetes `service` in contrail environment. specifically, we'll focus on `clusterIP` and `loadbalancer` type of services that is commonly used in practice. contrail uses its `loadbalancer` object to implement these two type of services. we'll first review the concept of legacy contrail neutron loadbalancer, then we'll look into the extended ECMP loadbalancer object which is the object that these two type of `service` are based on in contrail, for the rest part of this section we will explore how `clusterIP` and `loadbalancer` service works in detail, each with a test we build in our testbed.

## 5.1. Kubernetes Service

service is the core object in kubernetes. in chapter 3 you've learned what is kubernetes service and how to create a `service` object with yaml file. functional-wise, a service is running as a layer 4 (transport layer) load balancer that is sitting between clients and servers. client can be anything "requesting" a service. server in our context is the backend pods "responding" the request. the client only sees the "frontend" - a service IP and service port exposed by a service, it does not (and no need to) care about which backend pods (and with what "pod IP") actually responds the service request. inside of the cluster, that `service IP`, also called `cluster IP`, is a kind of virtual IP (`VIP`).

**NOTE** in contrail environment it is implemented through floating IP.

This design model is very powerful and efficient in one 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 contrail kubernetes integration environment, all 3 types of services are supported:

- `clusterIP`
- `nodePort`
- `loadbalancer`

next we'll introduce how service is implemented in contrail environment.

## 5.2. Contrail Service

in chapter 3 we've introduced kubernetes default implementation of service through `kube-proxy`. in there we mentioned CNI providers can have its own implementations. in contrail, `nodePort` service is implemented by `kube-proxy`. however, `clusterIP` and `loadbalancer` services are implemented by contrail's `loadbalancer (LB)`.

before we dive into the details of kubernetes service in contrail, it will be good to review the legacy openstack based loadbalancer concept in contrail.

**TIP** for brevity we'll sometimes also refer `loadbalancer` as `LB`.

### 5.2.1. Contrail Openstack Loadbalancer

contrail loadbalancer is an relatively "old" feature that is supported since version 1.x. it enables the creation of a pool of VMs serving applications, sharing one virtual-ip (**VIP**) as the frontend IP towards clients. this diagram below illustrates contrail loadbalancer and its components.

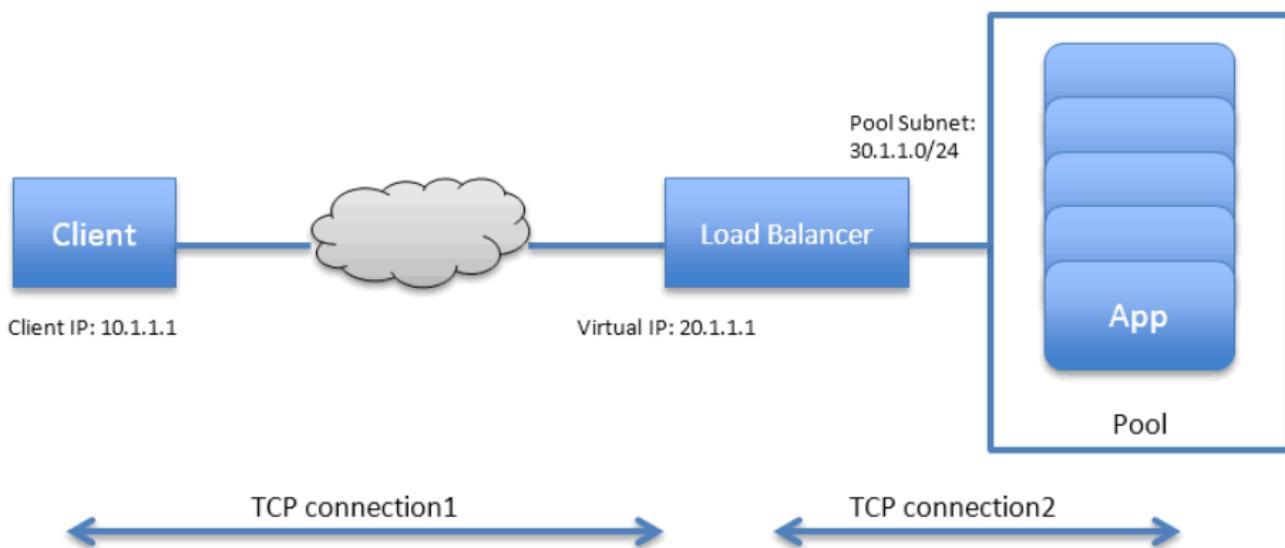


Figure 32. contrail openstack loadbalancer

some highlights of this figure:

- the LB is created with a internal VIP **30.1.1.1**. a **LB listener** is also created for each listening ports.
- all backend VMs together compose a **pool** which is with subnet **30.1.1.0/24**, same as LB's internal VIP.
- each backend 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 external world, it is 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

how it works:

- when LB sees a request coming from the client, it does TCP connection proxying. what that means is it establishes the TCP connection with the client, extracts the clients' HTTP/HTTPS requests, creates a new TCP connection towards one of the backend VMs from the pool, and send the request in the new TCP connection.
- when LB gets its response from the VM, it forwards the response to the client.
- when client closes the connection to the LB, the LB may also close its connection with the backend VM.

**TIP** when client closes its connection to LB, LB may or may not close its connection to backend VM. depending on the performance or other consideration it may use a timeout before it tears down the session.

you see that this loadbalancer model is very similar to 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 kubernetes service implementation. to support service loadbalancing, contrail extends the loadbalancer with a new driver, with it service will be implemented as "equal cost multiple path"(ECMP) loadbalancer working in layer 4(transport layer). this is the primary difference comparing with the "proxy" mode that the openstack loadbalancer type does.

*some more implementation details:*

- Actullay any loadbalancer can be integrated with contrail via contrail component `contrail-svc-monitor`.
- Each loadbalancer has a loadbalancer driver that is registerd to contrail with a `loadbalancer_provider` type.
- `contrail-svc-monitor` listens to contrail `loadbalancer`, `listener`, `pool` and `member` objects, it also calls the registered loadbalancer driver to do other necessary jobs based on the `loadbalancer_provider` type.
- contrail by default provides "ecmp loadbalancer" (`loadbalancer_provider` is `native`) and "haproxy loadbalancer" (`loadbalancer_provider` is `opencontrail`).
- The openstack loadbalancer is using "haproxy loadbalancer".
- ingress, on the other hand, is conceptually even closer with the openstack loadbalancer in the sense that both are layer 7 (application layer) "proxy" based. more about ingress will be discussed in later section.

### 5.2.2. Contrail Sevice Loadbalancer

let's take a look at service loadbalancer and the related objects.

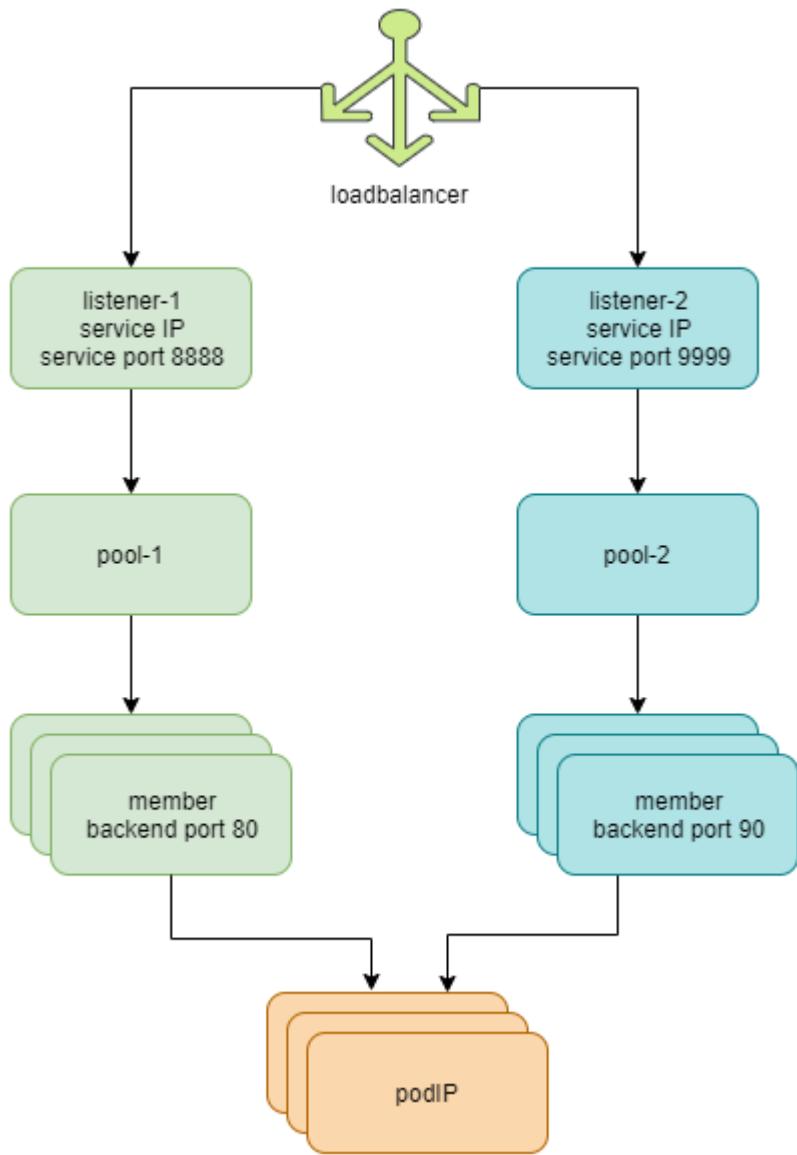


Figure 33. service loadbalancer

highlights in this figure:

- Each service is represented by a `loadbalancer` object.
- the `loadbalancer` 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 number of backend pod one pool may have multiple `members`
- each member object in the pool will map to one of pod backend

this is how service works in contrail:

- `contrail-kube-manager` listens `kube-apiserver` for k8s service and when a `clusterIP` 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 same as the `service IP`

- The `service-ip/VIP` will be linked to each backend pod's interface. This is done by a ecmp loadbalancer driver.
- the linkage from service-ip to multiple backend pods interface creates an ECMP next-hop in contrail, traffic will be loadbalanced from the source pod towards one of the backend pod directly. later we'll show the ECMP prefix in the pod's VRF table
- `contrail-kube-manager` continues to listen to `kube-apiserver` for any changes, based on pod list in `Endpoints` it will know the most current backend pods, and update members in the pool .

the most important thing to understand in this diagram, as we've mentioned, is that in contrast to the legacy neutron loadbalancer (and the ingress loadbalancer which we'll discussed later), there is no application layer "proxy" in this process. contrail service implementation is based on layer 4 (transport layer) ECMP based loadbalancing.

### 5.2.3. Contrail Loadbalancer Objects

we've talked a lot about the contrail "loadbalancer object" and you may wonder what exactly it looks like. now we'll dig a little bit deeper to look at the loadbalancers and the supporting objects: listener, pool, members.

in contrail setup you can pull the object data either from contrail UI, CLI (`curl`) or third party UI tools based on restapi. in production depending on which one is available and handy you can select your favorite.

*explore loadbalancer object with `curl`*

with `curl` tool you just need a FQDN of the URL pointing to the object.

e.g.: to find the loadbalancer object URL for the service `service-web-clusterip` from loadbalancers 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 loadbalancer 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 a whole bunch of details that may not be of our interests at this moment. but it does tell something interesting:

- in "loadbalancer\_properties", the LB use service IP as its VIP
- the LB is connected to a listener by a reference
- **loadbalancer\_provider** attribute is **native**, this is a new extension to implement layer 4 (transport layer) ECMP for kubernetes service

## explore LB from UI

in the rest part of the exploration to LB and its related objects, we'll use the legacy contrail UI.

**TIP** you can also use the new contrail command UI to do the same.

for each service there is a LB object, in the below capture it shows 2 LB objects:

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

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

Figure 34. loadbalancer object list

this indicates 2 services were created. the service loadbalancer object's name is composed by connecting NS name with service name, hence we can tell the 2 service's name:

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

## Loadbalancer

click on the small triangle icon in left of the first loadbalancer object ns-user-1-service-web-clusterip to expand it, then click on advanced json view icon on the right, you will see the similar detail information as what you've seen in curl capture. for example the VIP, loadbalancer\_provider, loadbalancer\_listener object that refers it, etc.

from here you can keep expanding the loadbalancer\_listener object by clicking the + character to see the detail information of it. you then see a loadbalancer\_pool, expand it again you will see member. you can repeat this process to explore through the object data. by the reference all of these objects are connected to each other and work together.

```

{
  "href": "http://10.85.188.19:9082/loadbalancer/99fe8ce7-9e75-11e9-b485-0050569e6cfc",
  "fq_name": [
    "default-domain",
    "k8s-ns-user-1",
    "service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc"
  ],
  "uid": "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-87d4-22e1abdc7e67"
        },
        "parent": {
          "parent_uid": "86bf8810-add4-45d1-aed0-15c74d5f7809",
          "parent_type": "project"
        },
        "perm2": [
          ...
        ],
        "id_perms": [
          ...
        ],
        "loadbalancer_refs": [
          ...
        ],
        "fq_name": [
          ...
        ],
        "loadbalancer_pool": [
          {
            "name": "service-web-clusterip__99fe8ce7-9e75-11e9-b485-0050569e6cfc-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1abdc7e67"
          }
        ]
      }
    ],
    "display_name": "ns-user-1__service-web-clusterip",
    "uid": "99fe8ce7-9e75-11e9-b485-0050569e6cfc",
    "service_appliance_set_refs": [
      ...
    ],
    "parent": {
      "parent_uid": "86bf8810-add4-45d1-aed0-15c74d5f7809",
      "parent_type": "project"
    },
    "virtual_machine_interface_refs": [
      ...
    ],
    "loadbalancer_properties": {
      "status": null,
      "provisioning_status": "ACTIVE",
      "admin_state": "true",
      "ip_address": "10.105.139.153",
      "vif_subnet_id": null,
      "operating_status": "ONLINE"
    }
  },
  "perm2": [
    ...
  ],
  "id_perms": [
    ...
  ],
  "fq_name": [
    ...
  ],
  "loadbalancer_provider": "native"
}

```

Figure 35. *loadbalancer*

## Listener

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

**TIP**

in order to see the detail parameters of an object in JSON format, click the triangle in the left of the loadbalancer name to expand it, then click on the "Advanced JSON view" icon on the up right corner in the expanded view. We'll use the JSON view a lot in this book to explore different contrail objects.

```

Configure > Networking > Load Balancing > ns-user-1_service-web-clusterip >
Load Balancer Info Listener

Listener
Name Description Protocol Port Pool
service-web-clusterip_99fe8ce7-9e75-11e9-b485-0050569e6fcf-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-b485-0050569e6fcf-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
    uuid: 3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
    parent_uuid: 86bf8810-add4-45d1-aa6b-15c74d5f7809
    parent_type: project
    perms2: [
        ...
    ]
    id_perms: [
        ...
    ]
    loadbalancer_refs: [
        ...
    ]
    fq_name: [
        ...
    ]
    loadbalancer_pool: [
        {
            display_name: service-web-clusterip_99fe8ce7-9e75-11e9-b485-0050569e6fcf-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
            uuid: 9999b0da-1424-4c98-adb7-e352b1368366
            parent_uuid: 86bf8810-add4-45d1-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-b485-0050569e6fcf-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
        }
    ]
}

```

Figure 36. listener

## Pool and Member

just repeat the exploring process we will get down to the pool and two **members** in it. the member is with a port of **80**, which maps to the container targetPort in pod.

```

Configure > Networking > Load Balancing > ns-user-1_service-web-clusterip > service-web-clusterip_99fe8ce7-9e75-11e9-b485-0050569e6fcf-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67 >
Listener Info Pool

Pool
Name Description Protocol Loadbalancer Method Pool Members Health Monit
service-web-clusterip_99fe8ce7-9e75-11e9-b485-0050569e6fcf-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67 TCP - 2 0

{
    display_name: service-web-clusterip_99fe8ce7-9e75-11e9-b485-0050569e6fcf-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67
    uuid: 9999b0da-1424-4c98-adb7-e352b1368366
    parent_uuid: 86bf8810-add4-45d1-aa6b-15c74d5f7809
    parent_type: project
    loadbalancer_members: [
        {
            fq_name: [
                ...
            ]
            uuid: f11a170d-a824-4df4-b662-8eb106afa395
            parent_uuid: 9999b0da-1424-4c98-adb7-e352b1368366
            parent_type: loadbalancer-pool
            perms2: [
                ...
            ]
            id_perms: [
                ...
            ]
            display_name: f11a170d-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-0050569e6fcf
                    },
                    {
                        key: vmi
                        value: 3051c0fc-9e78-11e9-ae5a-0050569e6fcf
                    }
                ]
            }
        }
    ]
}

```

Figure 37. pool

Name	Description	Port	Address	Weight
dd4db0b1b-0503-4ee0-b1c2-b026be59825c	-	80	-	1

```

{
  "fq_name": [ ... ],
  "uuid": "dd4db0b1b-0503-4ee0-b1c2-b026be59825c",
  "parent_uuid": "99fe8ce7-9e75-11e9-b485-0050569e6fc-TCP-8888-3702fa49-f1ca-4bbb-87d4-22e1a0dc7e67",
  "parent_type": "loadbalancer-pool",
  "permssn": [ ... ],
  "id_perms": [ ... ],
  "display_name": "dd4db0b1b-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": "a2fd0de26-9e75-11e9-b485-0050569e6fc"
      },
      {
        "key": "vni",
        "value": "a3114884-9e75-11e9-a170-0050569e6fc"
      }
    ]
  ],
  "name": "dd4db0b1b-0503-4ee0-b1c2-b026be59825c"
}

```

Total: 2 records | 50 Records ▾

Figure 38. members

next we'll examine the vrouter VRF table for the pod to show contrail service loadbalancer ECMP operation details. in order to better understand the "1 to N" mapping between loadbalancer and listener shown in the loadbalancer 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.

## 5.3. Contrail ClusterIP Service

in chapter 3 we've demonstrated how to create and verify a clusterIP service. in this section we'll revisit the lab and look at some important details about contrail specific implementations. we'll continue and add a few more tests to illustrate the contrail service loadbalancer implementation details.

### 5.3.1. ClusterIP as FIP

this is the yaml file we 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
```

let's review what we got from service lab in chapter3:

```
$ 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
```

here we see one service is created, with one pod running as its backend. the label in the pod matches to the SELECTOR in service. the pod name also indicates this is a deploy-generated pod. later we can scale the deploy for ECMP case study, for now we'll stick to one pod and examine the ClusterIP implementation details.

in contrail, a **ClusterIP** is essentially implemented in the form of a FIP. once a service is created, a FIP will be allocated from the service subnet and associated to all the backend pod VMI to form the ECMP loadbalancing. Now all backend pods can be reached via cluserIP(along with the POD IP). This clusterIP(FIP) is acting as a "VIP" to the client pods inside of the cluster.

**TIP** Why contrail chose FIP to implement clusterIP? In the previous section, we have learned that contrail does NAT for FIP and service also needs NAT. So it is natural to use the FIP for clusterIP.

For loadbalancer type of service, contrail will allocate a second FIP - the "EXTERNAL-IP" as the VIP, and the external VIP is advertised outside of the cluster through gateway router. you will get more details about these later.

from UI we'll see the automatically allocated FIP as ClusterIP.

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-0050569e6fc)	

Figure 39. ClusterIP as FIP

the FIP is also associated with the pod VMI and podIP, in this case the VMI is representing the pod interface.

The screenshot shows the tungstenfabric interface. The navigation path is: Monitor > Infrastructure > Virtual Routers > cent333 > Interfaces. The left sidebar shows categories like Dashboard, Physical Topology, Control Nodes, etc. The main content area displays a table of interfaces:

Name	Label	Status	Type	Network	IP Address	Floating IP	Instance
enc192	-1	Up	eth			None	
vhhost0	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 IPv4: 10.105.139.153	03f8fdb1-9a0c-11e9-9dff-0050569e6cfc	03f8fdb1-9a0c-11e9-9dff-0050569e6cfc

Figure 40. pod interface

the interface can be expanded to display more details:

```
- {
    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 >
    ipv6_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 41. pod interface detail

expand the **fip\_list**, we'll see more information below:

```

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` tells 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 FIP representing ClusterIP, NAT will happen in service. later we'll examine NAT again in the flow table.

### *Scaling Backend Pods*

in chapter 3 clusterIP service example, we have created a sevice and a backend pod. to verify the ECMP, let's increase the replica to 2 to generate a second backend pod. this is a more realistic and rebost model: each pod will now be backing up each other to avoid a single point failure.

instead of using yaml file to manually create a new webserver pod, with the "kubernetes spirit" in mind you should think of to `scale` a Deployment, as what you've seen earlier in this book. in our service example we've been using `Deployment` object to spawn our webserver pod on purpose:

```
$ 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. we end up having 2 backend pods now, one is running in 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 `-o wide` option, only first endpoint will be displayed properly.

we go ahead and check the FIP again.

The screenshot shows the tungstenfabric UI interface. On the left, there is a navigation sidebar with various options like Configure, Infrastructure, Security, Tags, Physical Devices, Networking, Load Balancing, Networks, Ports, Policies, Security Groups, Routers, IP Address Management, Floating IP Pools, and Routing, QoS, SLO. The 'Floating IP Pools' section is expanded, and the 'Floating IPs' sub-section is highlighted with a red box. In the main content area, the 'Networking > Floating IPs > default-domain > k8s-ns-user-1' path is shown. A floating IP address, 10.105.139.153, is selected and highlighted with a red box. Below it, the 'Mapped Fixed IP Address' section shows two entries: 10.47.255.238 (rc-webserver-vl0zs\_0404e594-9a0c-11e9-a5ed-0050569e6cf0) and 10.47.255.236 (rc-webserver-v92dt\_e2e308e6-9a97-11e9-9c44-0050569e6cf0). There are also sections for 'Details', 'Permissions', and a table at the bottom showing 'Total: 1 records | 50 Records'.

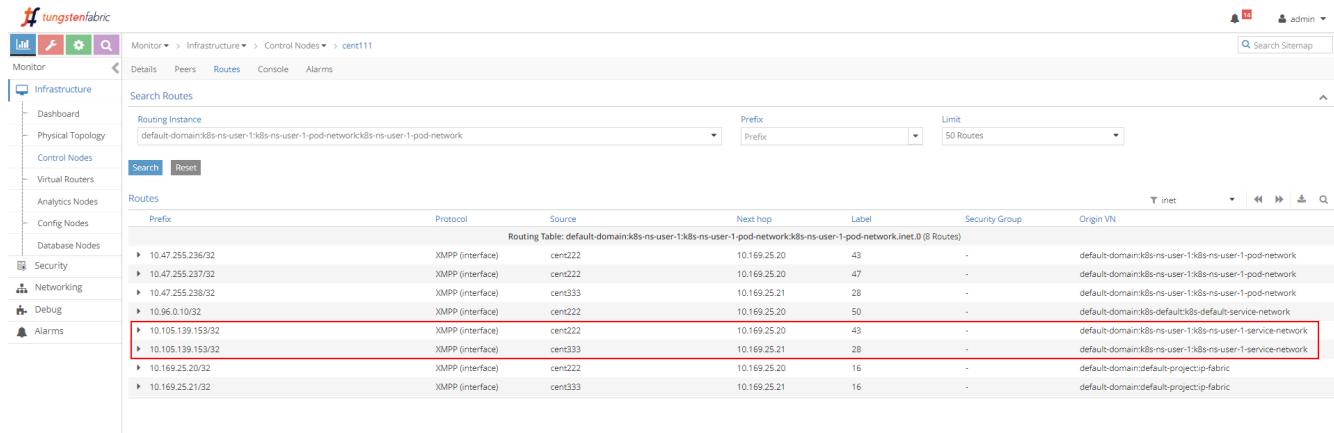
Figure 42. ClusterIP as FIP (ECMP)

we see the same FIP, but now it is associated with two podIP, each representing a separate pod.

### 5.3.2. ECMP Routing Table

#### Control Node Perspective

first, to examine the ECMP, let's take a look at the routing table in the controller's routing instance.



The screenshot shows the tungstenfabric control node interface. The left sidebar has sections for Infrastructure, Physical Topology, Control Nodes, Virtual Routers, Analytics Nodes, Config Nodes, Database Nodes, Security, Networking, Debug, and Alarms. The main area shows a search bar for 'Search Routes' and a table titled 'Routes'. The table has columns: Prefix, Protocol, Source, Next hop, Label, Security Group, and Origin VN. A dropdown menu above the table says 'Routing Table: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network:k8s-ns-user-1-pod-network.inet.0 (8 Routes)'. The table lists several routes, with the last two entries highlighted by a red box:

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 43. control node routing instance table

the routing instance (RI) has a full name with the following format:

```
<DOMAIN>:<PROJECT>:<VN>:<RI>
```

in most cases RI inheritate the same name from it's VN, so in our 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** indicate the routing table type is unicast IPv4. there are many other tables which is not of our interests 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 a hint about the route propagation process: both nodes(compute) has advertised the same clusterIP toward the master(contrail controller), to indicate the presence of the running backend pods in itself. this route propagation is via XMPP. master(contrail controller) then reflect the routes to all other compute nodes.

#### Compute Node Perspective

next, starting from the client pod node **cent222**, we'll look at the the pod's VRF table to understand how the packets are forwarded towards the backend pods

VRF		Show Routes	L2	Unicast 6
Routes (1 - 11 of 11)				
Next hop Type				
► discard	Source: Local Policy: disabled Peer: Local Valid: true	Prefix: 10.32.0.0 / 12 (1 Route)		
► interface	Interface: tapeth0-304431 Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network Policy: enabled Peer: 10.169.25.19 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: LocalVmPort Valid: true	Prefix: 10.47.255.237 / 32 (2 Routes)		
► interface	Interface: tapeth0-5e04db Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network Policy: enabled Peer: 10.169.25.19 Valid: true	Prefix: 10.47.255.238 / 32 (1 Route)		
► interface	Interface: tapeth0-5e04db Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network Policy: enabled Peer: LocalVmPort Valid: true	Prefix: 10.47.255.239 / 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: 25 Tunnel type: MPLSoUDP Policy: disabled Peer: 10.169.25.19 Valid: true	Prefix: 10.47.255.253 / 32 (1 Route)		
► interface	Interface: ptk0 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: ptk0 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: 10.96.0.0 / 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)		
► tunnel	Source IP: 10.169.25.20 Destination IP: 10.169.25.21 Destination VN: default-domain:default-project:ip-fabric Label: 16 Tunnel type: MPLSoUDP Policy: disabled Peer: 10.169.25.19 Valid: true	Prefix: 10.169.25.22 / 32 (1 Route)		

Figure 44. vrouter vrftable

the most important part of the screenshot is the routing entry **Prefix: 10.105.139.153 / 32 (1 Route)**, it is our ClusterIP address. underneath the prefix there is a statement **ECMP Composite sub nh count: 2**. this indicates the prefix has multiple possible next hop to reach. now expand it by clicking the small triangle icon in the left, you will be given a lot more details about this prefix.

```
Prefix: 10.105.139.153 / 32 (1 Route)
Source IP: Destination IP: vrf: Ref count: Policy: enabled Peer: 10.169.25.19 Valid: true Label: -1

▼ ECMP Composite sub nh count: 2

- {
  nh: - {
    NsSandeshData: - {
      type: ECMP Composite sub nh count: 2
      ref_count: 1
      valid: true
      policy: enabled
      mc_list: + { ... }
      nh_index: 87
      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
  etree_leaf: false
  layer2_control_word: false
  tag_list: + { ... }
  inactive: false
}
}
```

Figure 45. vrouter ECMP nexthop

among all of the details in this outputs, the most important thing that is of our focus is **nh\_index: 87**, which is the next hop ID (**NHID**) for the clusterIP prefix. from vrouter agent docker, we can further resolve the "Composite" NHID to the **sub-NHs**, which is the "member" nexthops under the

"Composite" next hop:

**TIP** don't forget to execute the vrouter commands from the vrouter docker container. doing it from the host directly 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 capture:

- NHID 87 is an "ECMP composite nexthop"
- the ECMP nexthop contains 2 "sub" nexthops: nexthop 43 and nexthop 28, each representing a separate path towards the backend pods
- nexthop 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 2 nodes.
- nexthop 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 `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.

### 5.3.3. ClusterIP Service Workflow

the clusterIP service's loadbalancer ECMP workflow is illustrated in this figure:

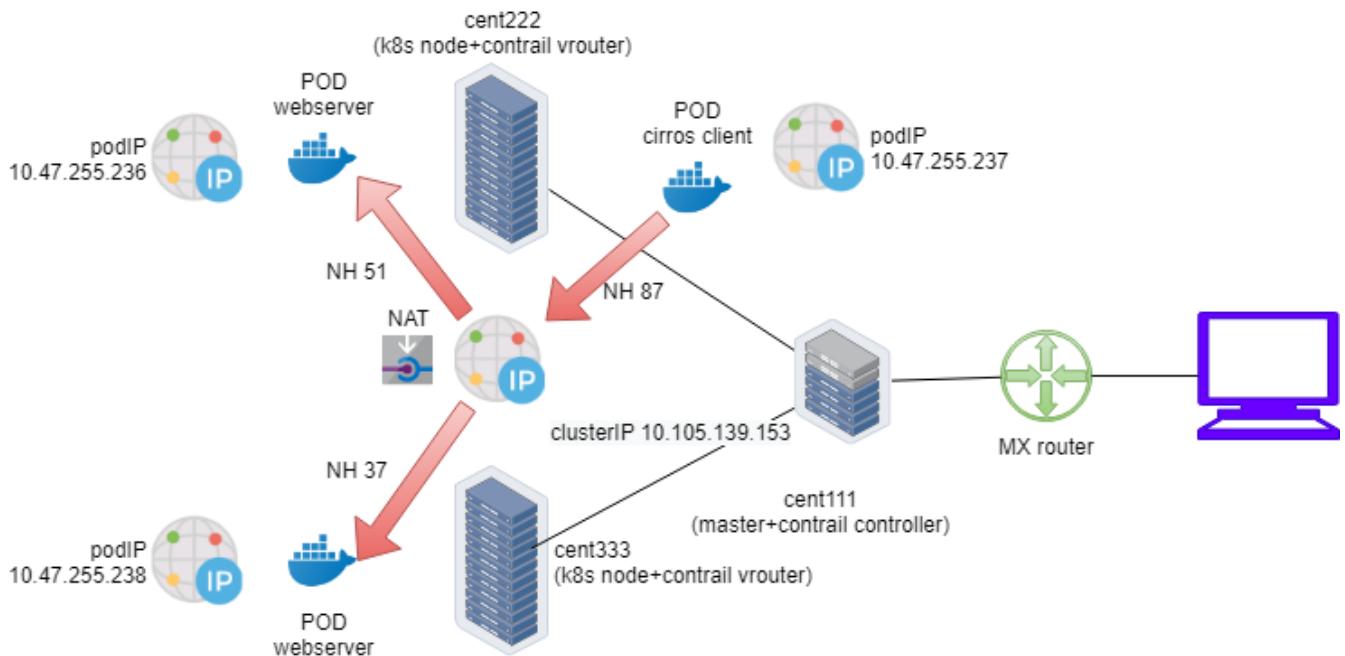


Figure 46. contrail service loadbalancer ECMP forwarding

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
- **client** sends the packet to **cent222** vrouter based on the default route.
- vrouter on **cent222** got the packet, it checks its corresponding VRF table, get a "Composite" nexthop ID **87**, which resolves to two sub-nexthops **51** and **37**, representing a remote and local backend pod respectively. this indicates ECMP.
- vrouter on **cent222** starts to forward the packet to one of the pod based on its ECMP algorithm. Suppose the remote backend pod is selected, the packet will be sent through MPLSoUDP tunnel to the remote pod on node **cent333**, after establishing the flow in the flow table. all subsequent packets belongs to the same flow will follow this same path. same applies to the local path towards local backend pod.

### 5.3.4. Multiple Port Service

we've understood how the service layer 4 ECMP works and we've explored the LB objects in lab. remember in the figure showing the LB and relevant objects, we saw that one LB may have 2 or more LB listeners. each listener has an individual backend pool that has one or multiple member(s).

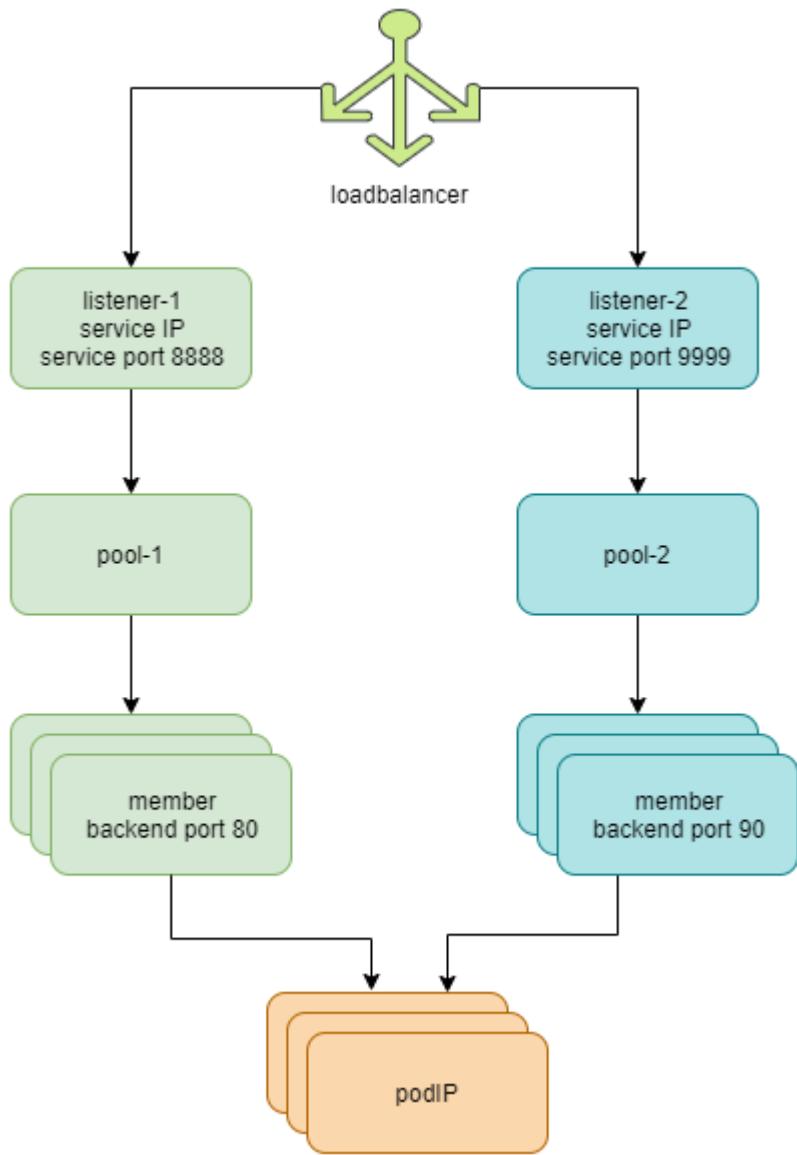


Figure 47. service loadbalancer

in kubernetes, this 1:N mapping between loadbalancer and listeners indicates a **multiple port service** - one service with multiple ports. let's look at the yaml file of it:

#### Example 17. 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 we've added is another item in the `ports` list: a new service port `9999` that maps to container's `targetPort 90`. now with two port mappings we have to give each port a name, `port1` and `port2` respectively.

**NOTE** without a port `name` the multiple ports yaml file won't work.

now we apply the yaml file and a new service `service-web-clusterip-mp` with 2 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 simply the case study we've scaled down the backend deployment's replicas number to one.

it looks everything is ok, isn't it? the new service comes up with 2 service ports exposed, `8888` is the old one we've tested in previous examples, and the new `9999` port should work equally well.

turns out that is not the case.

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. reason is the `targetPort` is not running in pod container,

so there is no way you will 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
```

**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 pod to listen on the new port **90** also. one of the easiest way today to start a HTTP server is to use the **SimpleHTTPServer** module coming with **python** package. in our 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 ...
```

now the **targetPort** is on, we can start the request towards service port **9999** again from the **client** pod. this time it succeed and get the returned webpage from python **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/
- 

for each incoming request the **SimpleHTTPServer** logs one line output, with an IP address showing where the request came from. in our case the request coming from client pod is with the IP **10.47.255.237**.

```
10.47.255.237 - - [04/Jul/2019 23:49:44] "GET / HTTP/1.1" 200 -
```

### 5.3.5. Contrail Flow Table

so far we've tested clusterIP service, and we see client request is sent towards the service IP. in contrail environment **vrouter** is the module that does all of the packet forwarding job. when the

`vrouter` in client pod gets the packet, it looks up the corresponding VRF table in vrouter module for the client pod(`client`), gets the nexthop and resolves the correct egress interface and proper encapsulation. in our test so far, the client and backend pods are in 2 different nodes, the source `vrouter` decides the packets need to be sent in MPLSoUDP tunnel, towards the node where backend pod is running. what interests us the most is:

- how the service IP and backend podIP is 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 purpose?

the most "straightforward" method you would think of is to capture the packets, then decode and see. doing that however, may not be as easy as what you've expected.

1. first you need to capture the packet at different places:

- at the pod interface, this is after the address is translated, that part is easy
- 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.

2. 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.

the easier way is to check the vrouter flow table which records IP and port details about a traffic flow. in this test we will prepare a big file `file.txt` in backend webserver pod and try to download it from the client pod.

you may wonder to trigger a flow why we don't simply use same curl test to pull the webpage, as what we've done in 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 we can take time to investigate the flow. later on in `ingress` section we will demonstrate a different method with a one-liner shell script.

now in the client pod curl URL, instead of just give root path `/` to list the files in folder, we try to pull the file: `file.txt`

```
$ kubectl exec -it client -- curl 10.101.102.27:9999/file.txt
```

in 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 ongoing, we have enough time to collect the flow table from both client

and server node, in the vrouter docker container.

*Example 18. 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)(offlows 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)
                           10.101.102.27:9999
(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      10.101.102.27:9999          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:

- client starts 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 bidirectionally.
- Action **F** means "forwarding". note that there is no special processing like **NAT** happening here.

**NOTE** with a filter **--match 15.15.15.2**. only flow entries with Internet Host IP is printed.

we can conclude, from client's perspective, it only see the service IP. it is not aware of any backend pod IP at all.

*server node flow table*

now look at flow table in 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 422 Processed 1116 Used Overflow
entries 0
(Created Flows/CPU: 377 319 76 344)(offlows 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(SP), 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)
                           10.101.102.27:9999
(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)
```

let's look at the second flow entry first - the IPs looks same as the one we just saw in client side capture. traffic lands vrouter fabric interface from remote client node, across MPLSoUDP tunnel. destination IP and port are service IP and service port respectively. it seems nothing special here.

however, the flow **Action** now is 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 backend pod IP and port.

now look at the first flow entry. source IP **10.47.255.238** is the backend pod IP and source port is python server port **90** opened in backend container . obviously this is the returning traffic indicating the file downloading is still ongoing. the **Action** is also **NAT(N)**, but this time it is the reverse operation - source NAT (**SNAT**) and source PAT(**SPAT**). vrouter will translate backend's source IP source port to the service IP and port, before putting it into the MPLSoUDP tunnel and returning back to client pod in remote node.

the complete end to end traffic flow is illustrated here:

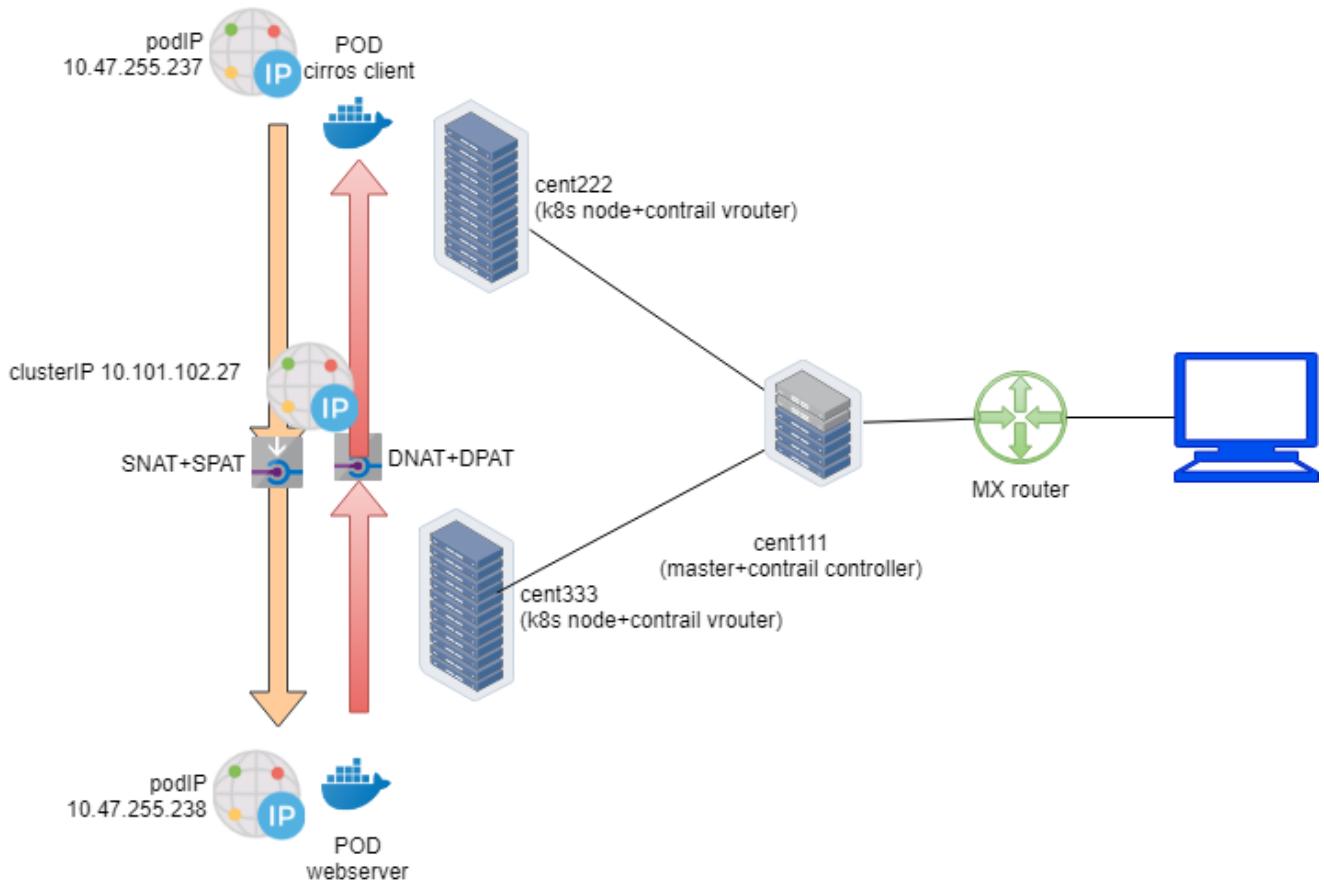


Figure 48. clusterIP service traffic flow (NAT)

## 5.4. Contrail Loadbalancer Service

in chapter 3 we've briefly talked about `LoadBalancer` service. in there we mentioned if the goal is to expose the service to the external world outside of the cluster, we just specify `ServiceType` as `LoadBalancer` in the service yaml file.

whenever a service of `type: LoadBalancer` get created, in contrail environment what will happen is , not only a `clusterIP` will be allocated and exposed to other pods within the cluster, but also a `floating ip` from public fip pool will be assigned to the loadbalancer 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 client sitting **outside** of the cluster, for example, a remote Internet host which sends request to the service across the gateway router.

in this section we'll demonstrate how does 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.

### 5.4.1. External IP as FIP

let's look at the yaml file of a `LoadBalancer` service. it is same as 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
```

comparing with the **clusterIP** service type, this time in the "EXTERNAL-IP" column there is an IP allocated. 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 FIP pool information in the service object yaml file, so based on the algorithm right FIP pool will be used automatically.

from UI we'll see that for **loadbalancer** service we now have 2 FIPs: one as clusterIP (internal VIP), the other one as "EXTERNAL-IP" (external VIP):

Configure	Floating IPs	Mapped Fixed IP Address	Floating IP Pool
Infrastructure	Floating IP	101.101.101.252 (rc-webserver-vl6zs_0404e594-9a0c-11e9-a5ed-0050569e6cf0)	k8s-vn-ns-default-pod-network:pool-ns-default
Security		10.96.89.48 (k8s_Service_service-web-lb_acb06596-8d95-11e9-bbbf-0050569e6cf0)	-

Figure 49. 2 FIPs for a **loadbalancer** service

both FIPs are associated with the pod interface:

Name	Label	Status	Type	Network	IP Address	Floating IP	Instance
eno192	-1	Up	eth			None	
vhost0	16	Up	vport	ip-fabric (default-project)	IPv4: 10.169.25.21	None	
tapeth0-03f8fd	36	Up	vport	k8s-ns-user-1-pod-network (k8s-ns-user-1)	IPv4: 10.47.255.238 IPv6: ::	10.96.89.48, 101.101.101.252	03f8fdb1-9a0c-11e9-9dff-0050569e6fcf /
+ { ... }							
pkt0	-1	Up	pkt			None	

Figure 50. pod interface

expand the tap interface, you will see the two FIPs are listed in `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 51. pod interface detail

now you should understand, the only difference here between the two type of services, is that for loadbalancer 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 expose itself to the external world.

## 5.4.2. Gateway Router VRF Table

in "contrail floating IP" section you've learned how to advertise FIP. here we'll review the main concepts to understand how it works in contrail `service` implementation.

the **route-target** community setting in the FIP 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 40
```

the FIP host route is learned by gateway router, from 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 detail version of the same route displays more information about this 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.13vpn.0

```

- the `source` indicates from which BGP peer the route is learned, `10.169.25.19` is the contrail controller (and kubernetes master) in our lab
- `protocol next hop` tells who generates the route. `10.169.25.20` is node `cent222` where the backend webserver pod is running
- `gr-2/2/0.32771` is an interface representing the (MPLS over) GRE tunnel between the gateway router and node `cent333`.

#### 5.4.3. Loadbalancer Service Workflow

to summarize, the FIP given to the service as its external ip is advertised to gateway router, and get loaded in the router's VRF table. when Internet host sends a request to the FIP, through MPLSoGRE tunnel the gateway router will forward it to the compute node where backend pod is locating.

the packet flow is illustrated in this figure:

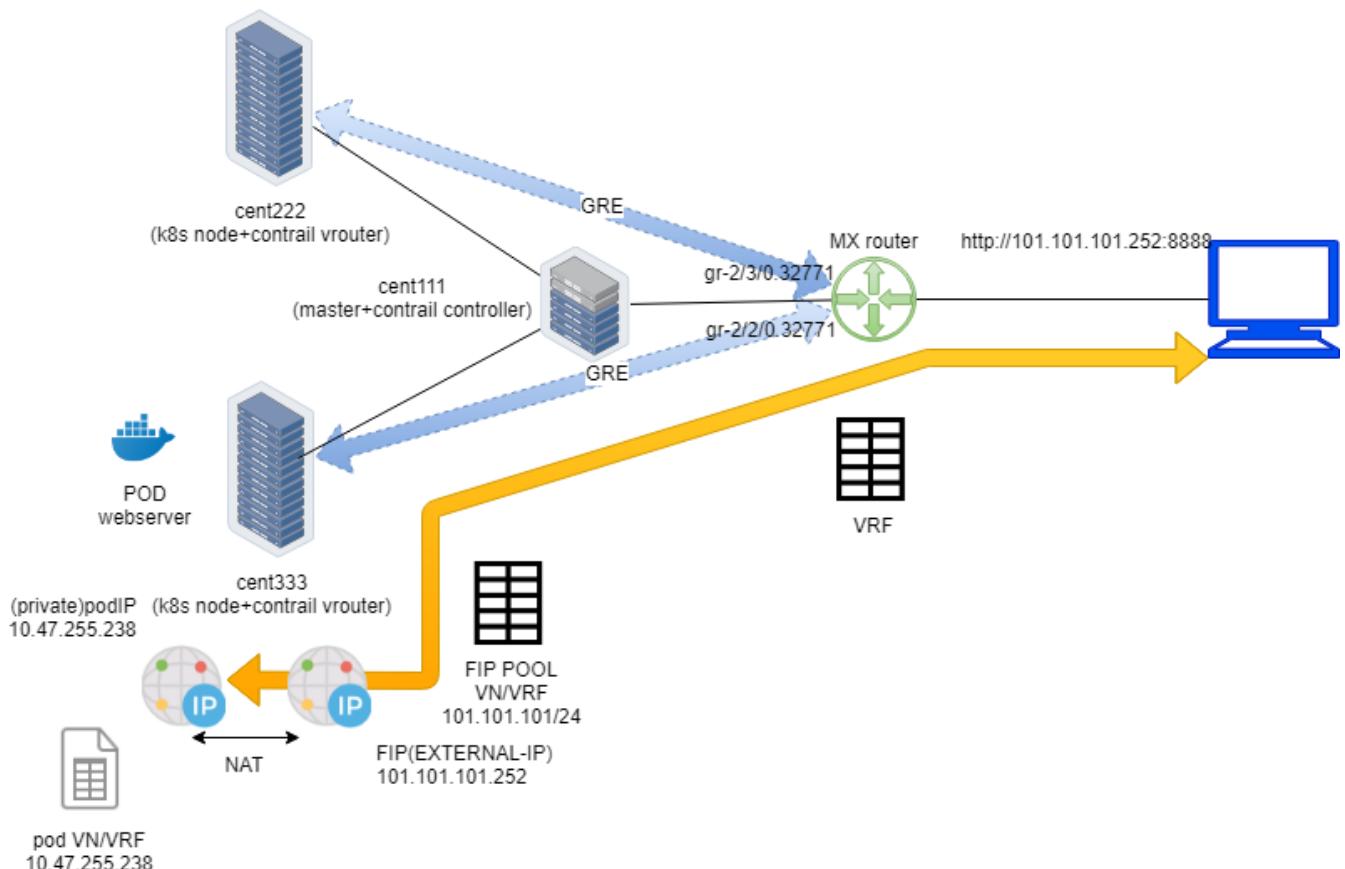


Figure 52. `loadbalancer` service workflow

here is the full story:

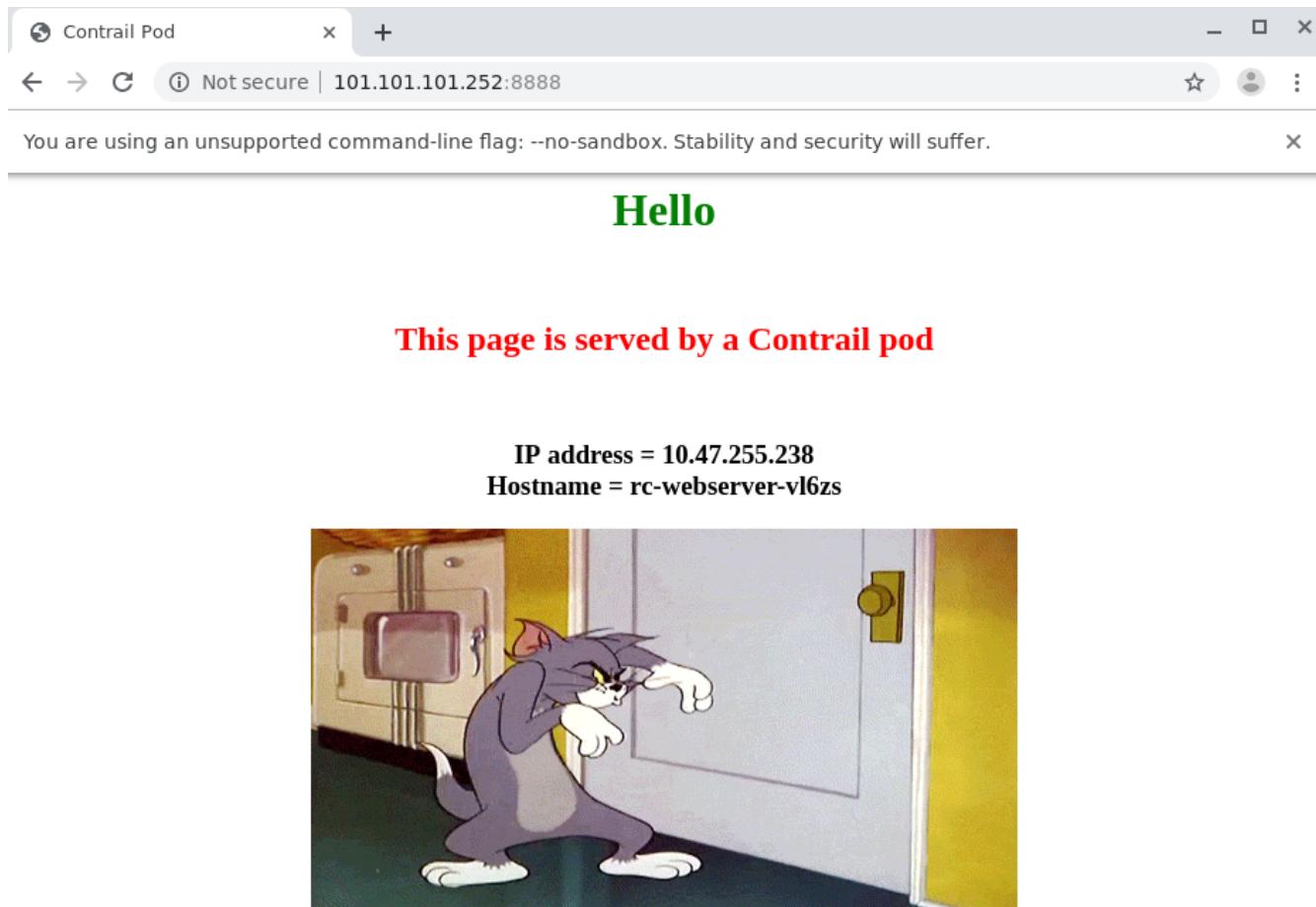
- you create a **FIP pool** from a public VN, with route-target the VN is advertised to the remote gateway router via MP-BGP
- you create a pod with a label `app: webserver`, kubernetes decides the pod will be created in node `cent333`. via XMPP the node publish the pod IP
- you create a loadbalancer 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 update the `endpoint` with the pod IP and port information.
- contrail create a loadbalancer instance and assign a FIP to it. contrail also associate that FIP with the pod interface, so there will be one to one NAT operation between the FIP and podIP.
- via XMPP, node `cent333` advertises this FIP to contrail controller `cent111`, which then advertises it to the gateway router.
- on receiving the FIP prefix, gateway router checks and see a the RT of the prefix matches to what it is expecting, it will import the prefix in local VRF. at this moment the gateway learns the nexthop of the FIP is `cent333`, so it generate a soft GRE tunnel toward `cent333`.
- when gateway router see a request coming from Internet toward the FIP, through the MPLS over GRE tunnel it will send the request to the node `cent333`
- vrouter in the node sees the packets destined to the FIP, it will perform NAT so the packets will be sent to the right backend pod.

## Verify Loadbalancer Service

To verify the end to end service access from Internet host to the backend pod, we will login to the Internet host desktop and launch a browser, with URL pointing to <http://101.101.101.252:8888>.

**TIP** just to keep in mind that the internet host request has to be sent to the public **FIP**, not to the **service IP(clusterIP)** or backend **podIP** which are only reachable from inside of the cluster!

this is the returned web page:



TIP: in our testbed we installed a centos desktop as an Internet host.

To simplify the test, you can also ssh into the Internet host and test it with **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
```

the kubernetes service is available from Internet!

## Loadbalancer Service ECMP

so far you've seen how loadbalancer type of service is exposed to the Internet and how the FIP did the "trick". in ClusterIP service section, you've also seen how the service loadbalancer ECMP works. what you haven't seen yet is how does the "ECMP" processing works under loadbalancer type of service. To demonstrate this again we 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 is the question: with 2 pods on different node as backend now, from the gateway router's perspective when it get the service request, which node it will 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 FIP prefix is imported as we've seen in previous example, except that now the same route is learned twice and an additional MPLSoGRE tunnel is created. previously in ClusterIP service example we use `detail` option in `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:64:0000008000000000      #<---
  Encapsulation: GRE-NUL
  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:64:0000008000000000      #<---
  Encapsulation: GRE-NUL
  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 **gr-** interface indicates the two end points of a GRE tunnel:

- **10.169.25.20:192.168.0.204**: tunnel between node **cent222** and gateway router
- **10.169.25.21:192.168.0.204**: tunnel between node **cent333** and gateway router

We end up to have 2 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 tunnel, whenever it got service request toward the same FIP. let's check it out.

### Verify Loadbalancer Service ECMP

to verify the ECMP we'll just pull the webpage a few more time and we expect to see both podIP displayed eventually.

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 pretty much similar with the *w3m* that we used earlier.

the only webpage we got is from the first backend pod *10.47.255.236*, *webserver-846c9ccb8b-xkjpw*, running in node *cent222*. the other one never show up. so the expected ECMP does not happen yet. when we examine the route again with *detail* or *extensive* keyword we 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
```

from that we realize that, even if the router learned the same prefix from both node, only one is *Active* and the other one 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 and detail discussion of it is out of the scope of this book.

**NOTE** for Junos BGP path selection algorithm, check this link: [https://www.juniper.net/documentation/en\\_US/junos/topics/topic-map/bgp-path-selection.html](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:

```
labroot@camaro# set routing-instances k8s-test routing-options multipath vpn-unequal-
cost
```

now 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 interface will be added under the FIP prefix, 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 Internet host multiple times with **curl** or web browser, we see the random result - both backend pod 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 here:

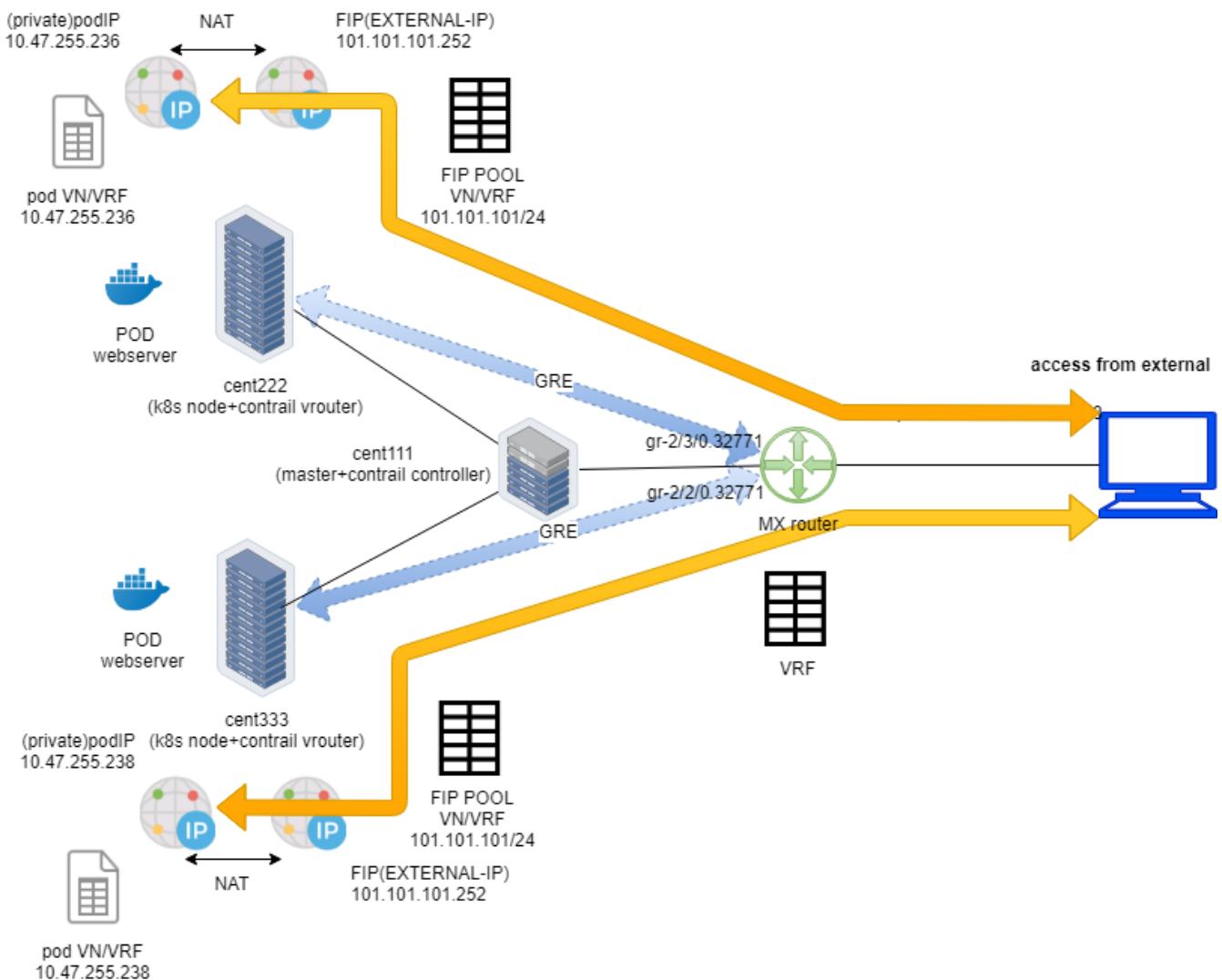


Figure 53. loadbalancer service ECMP

# Chapter 6. chapter 6: Contrail Ingress

in chapter 3 we've learned Ingress basis, the relation to service, Ingress types and the yaml file of each type.

in this chapter we'll introduce details of ingress workflow in contrail implementation, then we'll use a few test cases to demonstrate and verify how ingress works exactly in contrail environment.

## 6.1. Contrail Ingress Loadbalancer

like contrail's `service` implementation, contrail `Ingress` is also implemented through loadbalancer, but with a different `loadbalancer_provider` attribute. accordingly `contrail-svc-monitor` component takes different actions to implement `Ingress` in contrail environment.

Remember in "Contrail-Kubernetes architecture" section we gave the "object mapping" between kubernetes and contrail. in that section you've learned kubernetes `service` maps to `ECMP loadbalancer (native)` and `Ingress` maps to `Haproxy loadbalancer`.

in `service` section when we were exploring the loadbalancer and the relevant objects (`listener`, `pool`, and `member`), we noticed the loadbalancer's `loadbalancer_provider` type is `native`.

```
"loadbalancer_provider": "native",
```

in this section we'll see `loadbalancer_provider` type is `opencontrail` for Ingress's `loadbalancer`. we'll also look into the similarities and differences between `service` loadbalancer and `Ingress` loadbalancer.

## 6.2. Contrail Ingress Workflow

When an `Ingress` is configured in 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 out of the scope of this book, but in a high level here is the workflow:

1. `contrail-kube-manager` keeps listening to the events of `kube-apiserver`
2. user creates an `ingress` object (rules)
3. `contrail-kube-manager` gets the event from `kube-apiserver`
4. `contrail-kube-manager` creates a `loadbalancer` object in contrail DB, and set `loadbalancer_provider` type as `opencontrail` for ingress (where as it is `native` for `service`).
5. As mentioned earlier `contrail-service-monitor` component sees the `loadbalancer` creation event, based on `loadbalancer_provider` type, it invokes registered loadbalancer driver for the specified `loadbalancer_provider` type:
  - if the `loadbalancer_provider` type is `native`, It will invoke ECMP loadbalancer driver for ECMP loadbalancing which we've learned in previous section.
  - if the `loadbalancer_provider` type is `opencontrail`, It will invoke haproxy loadbalancer driver

which triggers haproxy processes to be launched in kubernetes nodes.

as you can see, contrail implements **Ingress** with haproxy loadbalancer, this is what you've read in the section of "contrail kubernetes object mapping". in chapter 3, we've talked about "Ingress controller", and multiple "Ingress Controllers" can coexist in contrail. in contrail environment the **contrail-kube-manager** plays **Ingress controller** role. it reads the Ingress rules that user input and programs them into the loadbalancer. furthermore:

- for each **Ingress** object, one loadbalancer 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 Ingress object.

## 6.3. Contrail Ingress Traffic Flow

client request, as a type of **overlay** traffic, may come from two sources depending on who initiates the request:

- internal request: requests coming from another pod inside of the cluster
- external request: requests coming from an Internet host outside of the cluster

the only difference between the two, is how the traffic hit the "active" haproxy.

an Ingress will be allocated 2 IPs:

- cluster-internal virtual IP
- external virtual IP

here is the traffic flow for client request:

*client to Ingress podIP*

1. for internal request it hits Ingress's "internal" VIP directly.
2. for external request it first hits Ingress's "external" VIP - the FIP, which is the one exposed to external, and that is the time when NAT starts to play as we've explained in **FIP** section. after NAT processing, traffic is forwarded to internal Ingress VIP.

*Ingress podIP to backend service*

3. from this moment on, both type of requests is processed exactly the same way.
4. the requests will be "proxied" to the corresponding service IP.

*backend service to backend pod*

5. based on the backend pods' availability, it will be sent to the node where one of the backend pods are located and reaches the target pods eventually.

6. In the case that 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 node.

here is the end to end service request flow when accessing from a pod in the cluster:

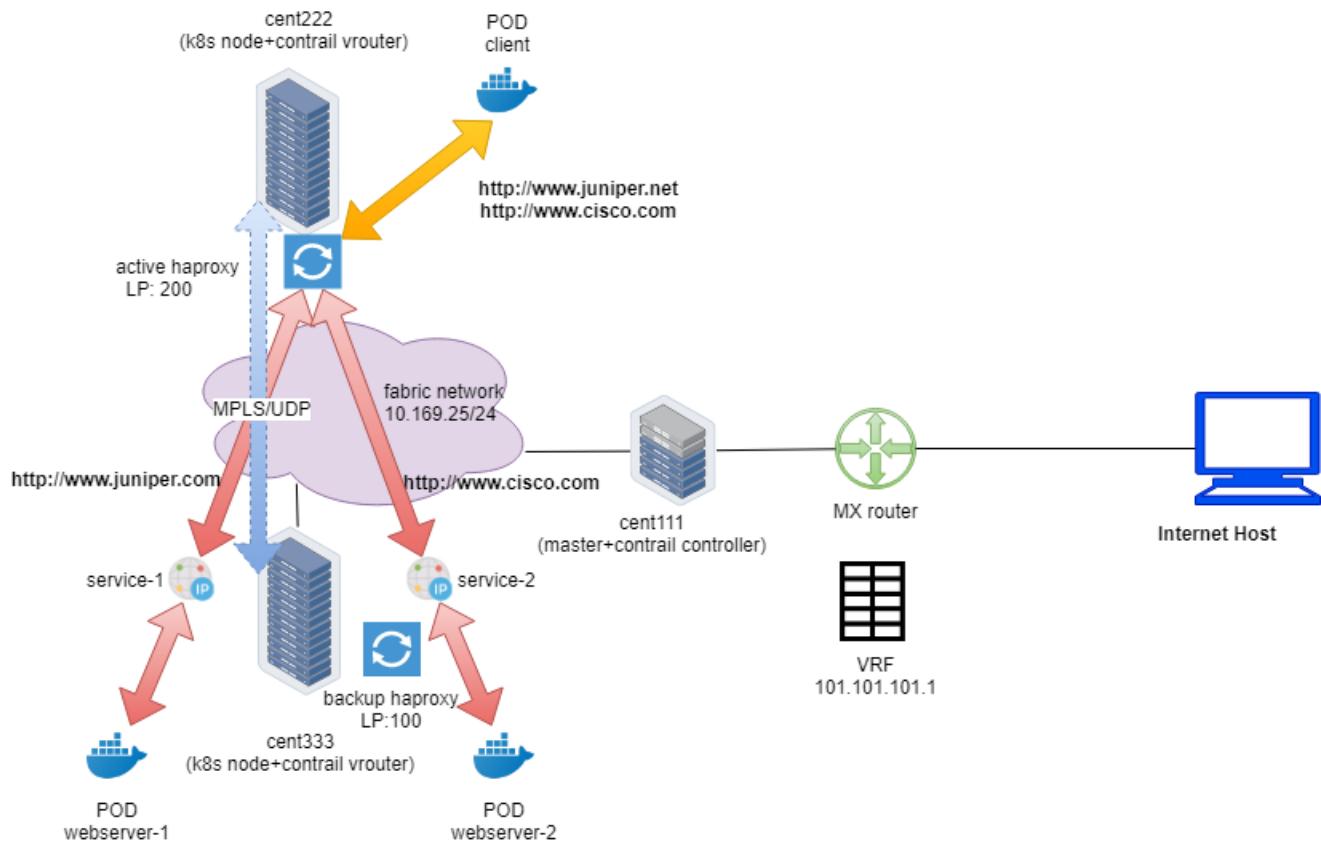


Figure 54. Ingress traffic flow: access from internal

here is the end to end service request flow when accessing from Internet host:

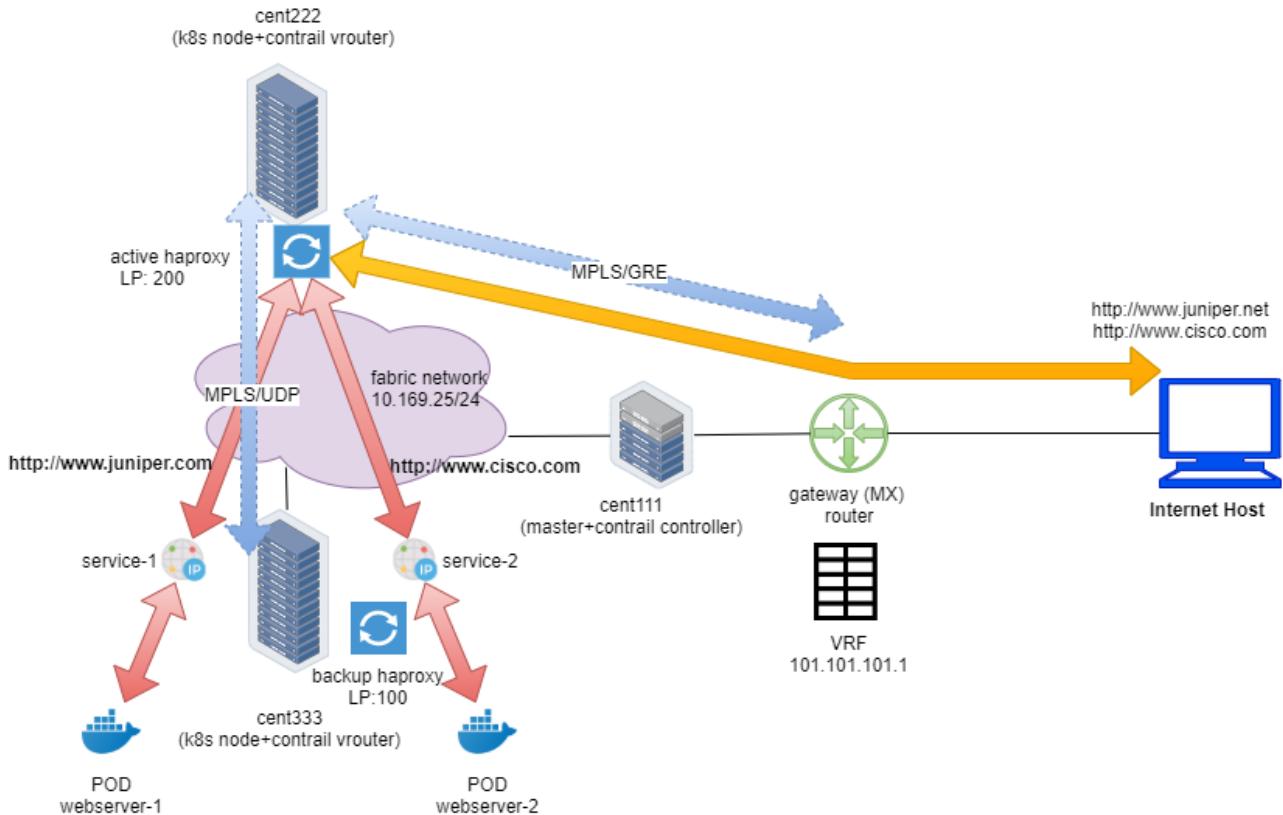


Figure 55. Ingress traffic flow: access from external

contrail supports all 3 types of ingress:

- http-based single-service ingress,
- simple-fanout ingress
- name-based virtual hosting ingress.

we'll look into each type of ingress.

## 6.4. Single Service Ingress

single service Ingress is the most basic form of Ingress. it does not define any rules and its main usage is to expose service to the outside world. it will proxy all incoming service request 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

## 6.4.1. Ingress Objects Definition

### Ingress Definition

in our single service ingress test lab, we want to achieve this goal:

- request toward any URLs will be 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 anything fancy. basically in this `single service Ingress` there is nothing else but a reference to one "single service" `webserver-1` as its "backend". all HTTP request will be dispatched to this service, and from there the request will reach a backend pod. next we'll look at the backend service.

### Backend service Definition

we can use exactly the same service as introduced in `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 to external directly anymore. therefore `LoadBalancer` type of service is not required.

### Backend pod Definition

same as in `service` example, we 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

```

## An "all in one" Yaml File

as usual, we can create an individual yaml file for each of the objects. but considering in [Ingress](#), these objects always need to be created and removed together, it is better to "merge" definitions of all these objects into one yaml file. yaml syntax supports this by using a "document delimiter", 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 this 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 objects independently, by giving the object name

**TIP** During test process you may need to create and delete all objects as a whole very often, grouping multiple objects in one yaml file can be very convenient.

## Deploy the Single Service Ingress

before applying the yaml file to get all objects created, let's take a quick look at our two nodes. we want to see if there is any `haproxy` process running without Ingress, so later after we deploy Ingress we can compare:

```
$ ps aux | grep haproxy
$
```

So the answer is no in both node. haproxy will be created only after we create `Ingress` and the corresponding loadbalancer object is seen by `contrail-service-monitor`. we'll check this again after we create an `Ingress`.

```
$ kubectl apply -f ingress/ingress-single-service.yaml
ingress.extensions/ingress-ss created
service/service-web-clusterip created
replicationcontroller/webserver created
```

the Ingress, one service and one Deployment object are now created.

### 6.4.2. Ingress Post Examination

#### Ingress Object

let's start to 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 applied to the Ingress properly. In this **single service Ingress** there is no explicit rules defined to map a certain URL to a different service - all HTTP requests will be dispatched to the same backend service.

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}}}
```

it actually contains the configuration information that you gave. can format it (with JSON formatting tool like python `json.tool` module) to get a better view:

TIP

```
{  
    "apiVersion": "extensions/v1beta1",  
    "kind": "Ingress",  
    "metadata": {  
        "annotations": {},  
        "name": "ingress-ss",  
        "namespace": "ns-user-1"  
    },  
    "spec": {  
        "backend": {  
            "serviceName": "service-web-clusterip",  
            "servicePort": 80  
        }  
    }  
}
```

you can do same formatting for all other objects to make it more readable.

what may confuse you is the two IP addresses shown here:

```
loadBalancer:  
  ingress:  
    - ip: 101.101.101.1  
    - ip: 10.47.255.248
```

we've seen these two subnets in service examples:

- `10.47.255.x` is an cluster-internal `podIP` allocated from the pod's default subnet
- `101.101.101.x` is the public `FIP` associated with an internal IP.

but the question is why an Ingress even requires a `podIP` and `FIP`?

let's hold the answer for now and continue to check service and pod object created from the all-in-one yaml file. we'll come back to this question shortly.

## Service Objects

```
$ 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 also created and allocated a clusterIP. we've seen this before and it looks nothing special. now look at the pods.

## Backend and Client Pod

```
$ 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. there is a backend pod running for the service. we have learned how selector and label works in service-pod associations so there is nothing new here. next we'll examine the haproxy and try to make some sense out of the 2 IPs allocated to Ingress object.

## Haproxy Processes

earlier before the Ingress is created, we were looking for haproxy process in node but could not see anything. let's check it again and see if any magic happens:

*Example 19. 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-
0050569e6cfbc/haproxy.conf
-p /var/lib/contrail/loadbalancer/haproxy/5be035d8-a918-11e9-8112-
0050569e6cfbc/haproxy.pid
-sf 23447
```

*Example 20. node cent333*

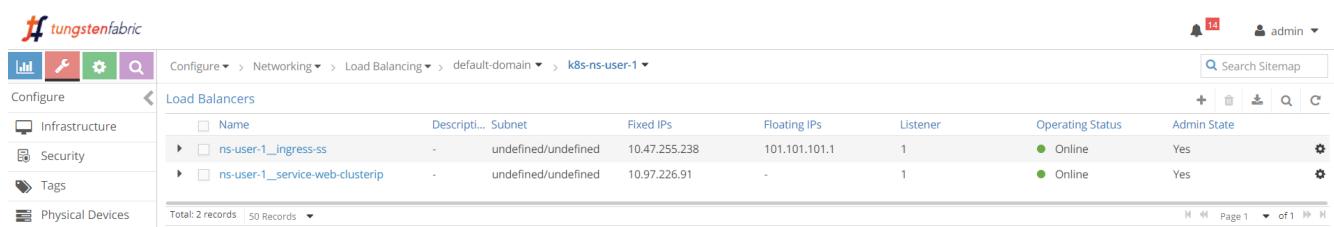
```
$ ps aux | grep haproxy
188 16335 0.0 0.0 55440 2892 ? Ss 00:58 0:00 haproxy
-f /var/lib/contrail/loadbalancer/haproxy/5be035d8-a918-11e9-8112-
0050569e6cfbc/haproxy.conf
-p /var/lib/contrail/loadbalancer/haproxy/5be035d8-a918-11e9-8112-
0050569e6cfbc/haproxy.pid
-sf 16317
```

right after ingress got created, we see a haproxy process created in each of our two nodes!

remember earlier when we talk about ingress contrail implementation, we've said contrail **Ingress** is also implemented through loadbalancer (just like **service**). Since Ingress's **loadbalancer\_provider** type is **opencontrail**, '**contrail-svc-monitor**' invokes haproxy loadbalancer driver. The haproxy driver generates 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 Loadbalancer Objects

we've mentioned "Ingress loadbalancer" for a few time but we haven't looked into it yet. In **service** section, we've looked into service loadbalancer object in UI and we inspected some details about the object data structure. now after we created Ingress object, let's check the list of loadbalancers object again and see what Ingress brings in here.



The screenshot shows the tungstenfabric UI interface. The top navigation bar includes icons for Home, Search, Configuration, Networking, Load Balancing, default-domain, and k8s-ns-user-1. The main content area is titled 'Load Balancers' and lists two entries:

Name	Description	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

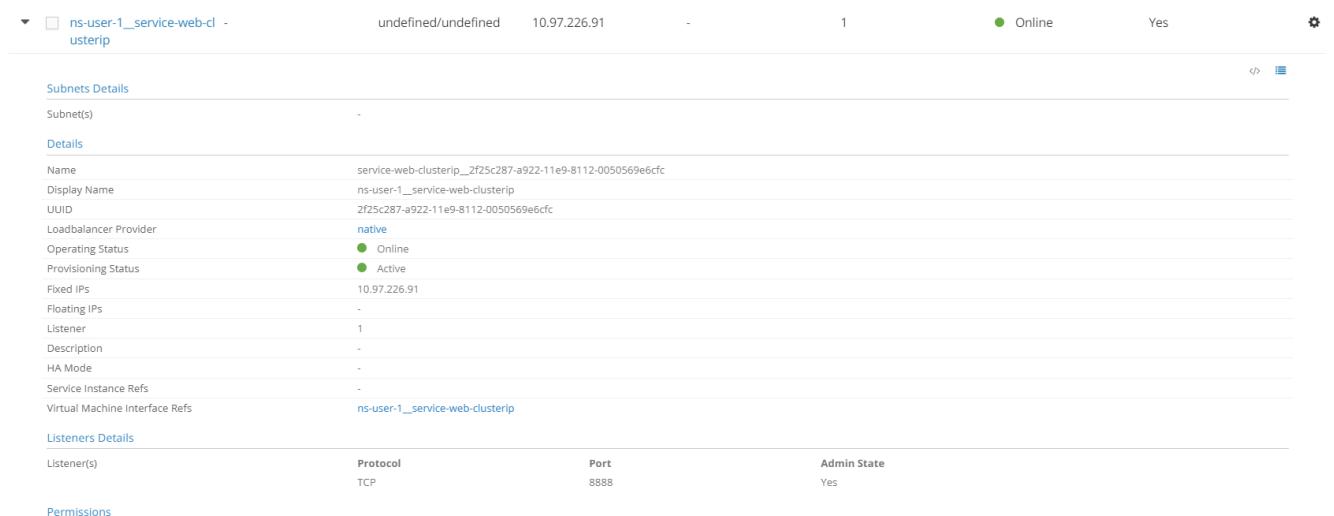
Total: 2 Records | 50 Records | Page 1 of 1

Figure 56. *loadbalancers (configuration > Networking > Floating IPs)*

2 loadbalancers are generated after we applied the all-in-one yaml file.

- loadbalancer **ns-user-1\_\_ingress-ss** for Ingress **ingress-ss**
- loadbalancer **ns-user-1\_\_webserver-clusterip** for service **webserver-clusterip**

we've learned the service loadbalancer object previously, if you expand the service you will see more details, but nothing would surprise us now.



The screenshot shows the expanded details of the service loadbalancer object 'ns-user-1\_\_service-web-clusterip'. The expanded view includes sections for Subnets Details, Details, Listeners Details, and Permissions.

Subnets Details	Subnet(G)	Details	Listeners Details	Permissions
ns-user-1__service-web-cl - usterip	undefined/undefined	<b>ns-user-1__service-web-clusterip</b> Name: service-web-clusterip_2f25c287-a922-11e9-8112-0050569e6fc Display Name: ns-user-1__service-web-clusterip UUID: 2f25c287-a922-11e9-8112-0050569e6fc 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	Protocol: TCP Port: 8888 Admin State: Yes	

Figure 57. *service loadbalancer object (click the triangle in the left of the loadbalancer name)*

as expected, the service loadbalancer has a ClusterIP, and a listener object that is listening on port 8888. one thing we want to highlight here again is the **loadbalancer\_provider**. the type is "native", so the action **contrail-svc-monitor** takes is layer 4 (application layer) ECMP process, which we've explored a lot in service section. now let's expand Ingress loadbalancer and look 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-a922-11e9-8112-0050569e6fcf
  fq_name: - [
    default-domain
    k8s-ns-user-1
    ingress-ss__2f23bddf-a922-11e9-8112-0050569e6fcf
  ]
  uuid: 2f23bddf-a922-11e9-8112-0050569e6fcf
  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-a922-11e9-8112-0050569e6fcf
    loadbalancer_provider: opencontrail
    display_name: ns-user-1_ingress-ss
    uuid: 2f23bddf-a922-11e9-8112-0050569e6fcf
    virtual_machine_interface_refs: - [
      - {
        to: + [ ... ]
        attr: null
        uuid: 5522c57b-3e06-4a78-a3e9-c3f9f377b784
        name: k8s__Ingress__ingress-ss__2f23bddf-a922-11e9-8112-0050569e6fcf
        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: - {
      uid: 2f070a47-bba9-44fc-9e44-ef67338672ab
      display_name: k8s-ns-user-1-pod-network
      name: k8s-ns-user-1-pod-network
      network_ipam_refs: - [
        + { ... }
      ]
    }
  }
}

```

Figure 58. ingress loadbalancer object

some highlights in the figure:

- `loadbalancer_provider` is `opencontrail`
- Ingress loadbalancer has a reference to a `virtual-machine-interface` (VMI) object
- the `VMI` object is referred by an `instance-ip` object with an (fixed) IP `10.47.255.238` and a `floating-ip` object with an (floating) IP `101.101.101.1`

at this moment, we can explain the Ingress IP `10.47.255.248` seen in ingress. basically:

- it is an cluster-internal IP address allocated from the default pod network as loadbalancer vip
- it is the frontend IP that the Ingress loadbalancer will listen for HTTP requests
- it is also what the public FIP `101.101.101.1` maps to with NAT

**TIP**

in this book we'll refer this private IP with different names interchangeably: "Ingress Internal IP", "Ingress internal VIP", "Ingress private IP", "Ingress loadbalancer interface IP", etc. to differentiate it from the Ingress public FIP, we can also name it as "Ingress podIP" since the internal vip is allocated from the pod-network. similarly we'll refer the Ingress public FIP as "Ingress external IP".

Now to compare the different purposes of these two IPs:

- Ingress **podIP** is the VIP facing other pods inside of the cluster. To reach Ingress from inside of the cluster, requests coming from other pods will have their destination IP set to Ingress **podIP**.
- Ingress **FIP** is VIP facing "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 node receives traffic destined to the Ingress FIP from outside of the cluster, vrouter will translate it into the Ingress **podIP**

the detail Ingress loadbalancer object implementation refers to a SI (service instance), and the SI again includes other data structure or reference to other objects (VM, VMI, etc). overall it is more complicated and involves more details than what we've covered and it is hard to put everything in this book. we've tailored the details into a high level overview so that important concepts like haproxy and the two Ingress IPes can be understood.

Once a HTTP/HTTPS request arrives to the Ingress **podIP**, from internal or external, Ingress loadbalancer will do HTTP/HTTPS proxy operation through haproxy process, and dispatch the requests towards the service and eventually to the backend pod.

we've seen the haproxy process is running, to examine more details of this proxy operation, next we can further check its configuration file for the running parameters details.

### haproxy.conf File

in each (compute) node, under `/var/lib/contrail/loadbalancer/haproxy/` folder there will be a subfolder for each loadbalancer uuid. the file structure looks like this:

```
8fd3e8ea-9539-11e9-9e54-0050569e6cfc
├── haproxy.conf
├── haproxy.pid
└── haproxy.sock
```

you can check haproxy.conf file for the haproxy configuration:

```

$ cd /var/lib/contrail/loadbalancer/haproxy/8fd3e8ea-9539-11e9-9e54-0050569e6cfec/
$ 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-
0050569e6cfec/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 10.97.226.91:80 weight 1
                                                ^^^^^^^^^^^^^^

```

the configuration is simple, and here is the illustration of it:

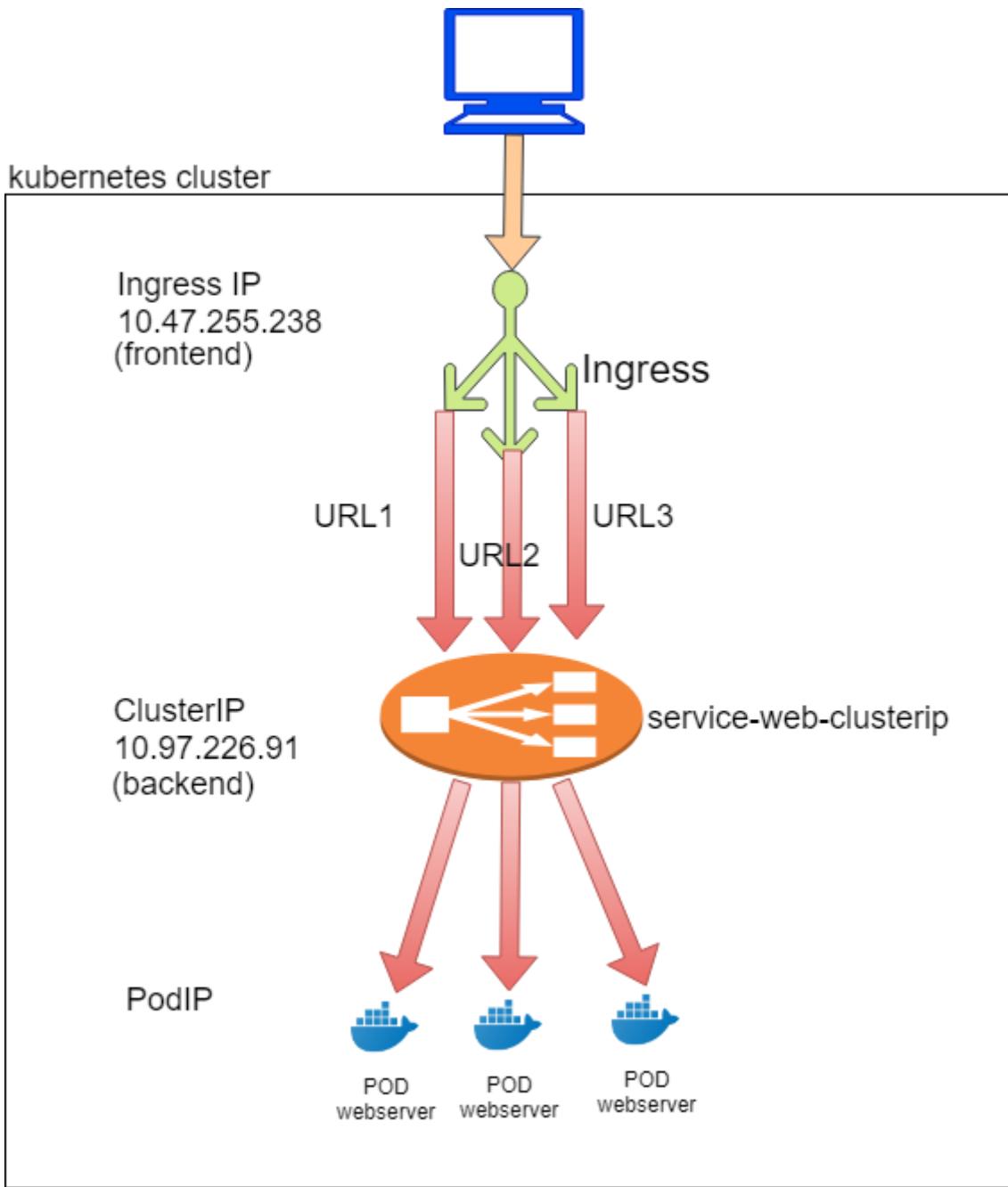


Figure 59. single service Ingress

- 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.
- the haproxy **backend** section defines the **server**, which is 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 **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 the only **backend** service **10.97.226.91:80**. this is due to the fact that there is 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.

**NOTE**

later in `simple fanout Ingress` and `name-based virtual hosting Ingress` examples, we will see another type of configuration statement `use_backend…if…` that can be used to force each URL to go to a different backend.

through this configuration, the haproxy implemented our single service Ingress.

### Gateway Router VRF Table

we've explored a lot inside of the cluster. 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 service example, from outside of the cluster, only FIP is visible. `detail` version of it 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

```

- through XMPP, vrouter advertises the FIP prefix to contrail controller. at least 2 pieces of information from the output indicates who represents the FIP in this example - node **cent222**:
  - **Protocol next hop** being **10.169.25.20**
  - **Route Distinguisher** being **10.169.25.20:5**
- through MP-BGP, contrail controller "reflects" the FIP prefix to the gateway router, **Source: 10.169.25.19** indicates this fact.

so it looks **cent222** is "selected" to be the active haproxy node, and the other node **cent333** is the standby one. therefore you should expect client request coming from Internet host goes to node **cent222** first. of course, the overlay traffic will be carried in MPLS over GRE tunnel, same as what you've seen from service example.

the FIP advertisement towards gateway router is exactly the same in all types of Ingresses.

another fact that we've skipped on purpose is the different "local preference" value used by the active and standby node when advertising FIP prefix. saying that will involve other complex topics like the active node selection algorithm and so on. but it is worth to understand from high level:

both nodes have loadbalancer and haproxy running so both will advertise the FIP prefix **101.101.101.1** to gateway router. however, they are advertised with different local preference value. the "Active" node advertise with a value of **200** and the "standby" node with **100**. contrail controller have both routes from the 2 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 Ingress public FIP route and internal VIP route advertisement.

## Ingress Verification: Internal

we've explored a lot about ingress loadbalancer and the related service, pod objects, etc. now it is 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 cluster, then from an Internet host outside of it.

*Example 21. from 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
```

we 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 services `service-web-clusterip`.

in the fourth request we didn't give a URL via `-H`, `curl` will fill `host` with the request IP address - `10.47.255.238` in this test, again it goes to the same backend pod and get the same returned response.

**NOTE**

The `-H` option is important in Ingress test with `curl`. it carries the full URL in HTTP payload that the Ingress loadbalancer 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 same as with a unknown URL.

### Ingress Verification: External (Internet host)

the more exciting part of the test is to visit the URLs from external. overall we hope `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 FIP address, we need to update `/etc/hosts` file by adding one line in the end - you probably don't want to just end up with a nice webpage from `Juniper`cisco`'s 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 internet host's "desktop", we launch chrome browser, and input one of the 3 URLs: `www.juniper.net`, `www.cisco.com` or `www.google.com`. By keep refreshing the pages we can confirm all HTTP request is returned by the same backend pod.

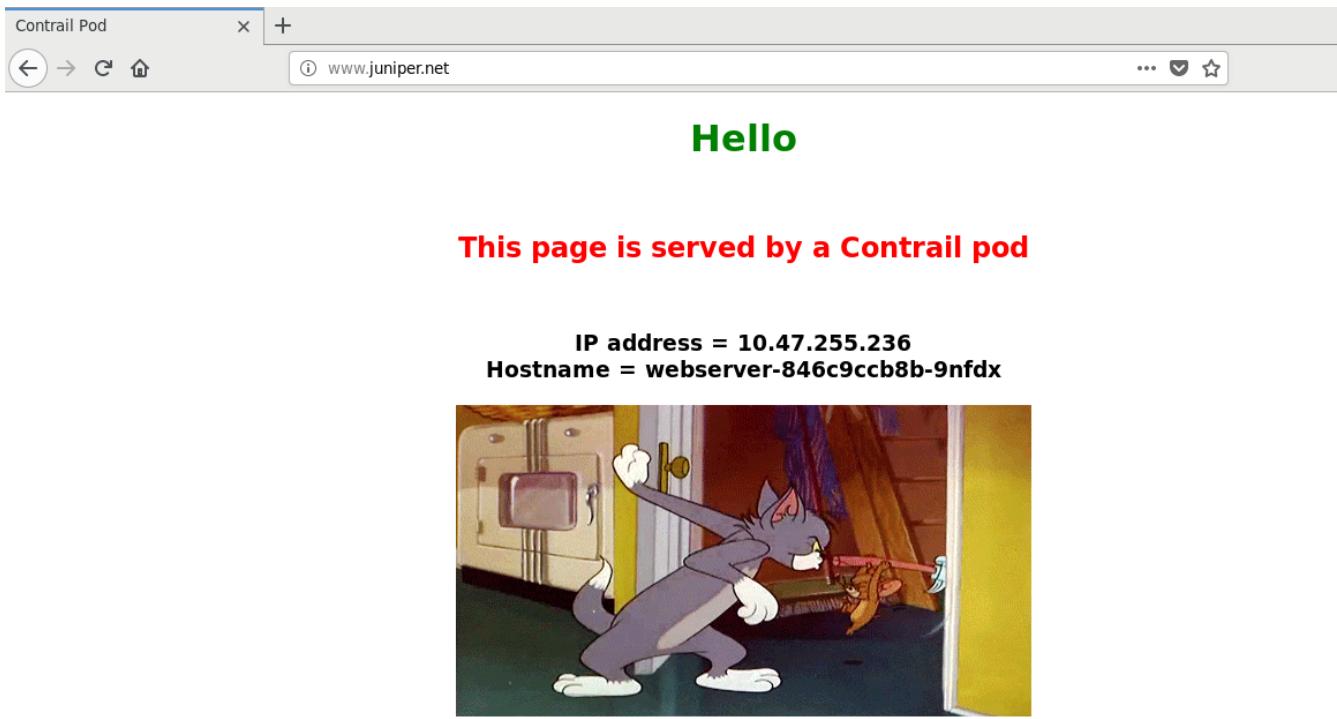


Figure 60. access [www.juniper.net](http://www.juniper.net) from Internet host

same result can be seen from `curl` also. the command is exactly the same as what we've seen when testing from a pod, except this time we send requests to Ingress external FIP, instead of the Ingress internal podIP.

*Example 22. from 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

$ 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

$ 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

$ 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
```

everything works!

next we'll look at the second Ingress type **simple fanout Ingress**. before moving forward, it is better to clean up everything. now we can take advantage of the all-in-one yaml file - 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
```

## 6.5. Simple Fanout Ingress

both **simple fanout Ingress** and **name-based virtual host Ingress** support "URL routing", the only difference is 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 loadbalancer directs traffic to different backend services.



to demonstrate **simple fan-out** type of Ingress, the objects that we need to create are:

- an **Ingress** object: defines the rules, mapping 2 paths to 2 backend services
- 2 backend services objects
- each service requires at least one pod as backend

we use the same **client** pod as cluster-internal client we've used in previous examples.

### 6.5.1. Ingress Objects Definition

#### ingress Definition

in our **simple fanout Ingress** test lab, we want to achieve these goals for host **www.juniper.net**:

- request toward path **/dev** will be directed to a service **webservice-1** with **servicePort 8888**
- request 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") we see "rules" defined - here it is the mappings from multiple "paths" to different backend services.

## backend service definition

since we defined 2 rules each for a [path](#), we need two services accordingly. we can "clone" the previous service in [single service Ingress](#) example and just change the service's name and selector to generate the second service. e.g.: 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 we have 2 backend services now, apparently we also need at least two backend pods each with a label matching to a service. we can clone the previous [Deployment](#) into two and just change the name and label of the second [Deployment](#).

There are the definition of the [Deployment](#):

*Example 23. 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
```

*Example 24. 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**

same as in **single service Ingress**, we 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

```

*Example 25. 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.

### 6.5.2. Ingress post examination

#### ingress objects and ingress loadbalancer

let's look at the kubernetes objects created from the all-in-one yaml file:

*Example 26. 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": "ingress-sf", "namespace": "ns-user-1"}, "spec": {"rules": [{"host": "www.juniper.net", "http": {"paths": [{"backend": {"serviceName": "webservice-1", "servicePort": 8888}, "path": "/dev"}, {"backend": {"serviceName": "webservice-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, within each rule there is a mapping from a **path** to the corresponding **service**. we see same Ingress internal podIP and external FIP as we've seen in the previous **single service Ingress** example:

```
loadBalancer:  
  ingress:  
    - ip: 101.101.101.1  
    - ip: 10.47.255.238
```

That is why from gateway router's perspective, there is no differences between all types of Ingress. in all cases a public FIP 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 backend services and pods:

#### *Example 27. 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    85m      app=webserver-1  
webservice-2  ClusterIP  10.100.156.38  <none>        8888/TCP    85m      app=webserver-2
```

### Example 28. backend and client pod

```
$ 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   ...
Vwebserver-2-846c9ccb8b-t75d8 1/1     Running   ... 156m  10.47.255.235 cent333 ..
```

two services are created, each with a different clusterIP allocated. for each service there is a backend pod. later when we verify Ingress from client we'll see these podIPs in the returned web pages.

### contrail Ingress loadbalancer object

comparing with **single service Ingress**, in here the only difference is one more **service** loadbalancer:

Load Balancers							
	Name	D...	Subnet	Fixed IPs	Floating IPs	Listener	Operating S... Admin State
▶	ns-user-1__ingress-sf	-	undefined/und efined	10.47.255.238	101.101.101.1 2	● Online	Yes
▶	ns-user-1__webservice-1	-	undefined/und efined	10.96.51.227	-	1	● Online Yes
▶	ns-user-1__webservice-2	-	undefined/und efined	10.100.156.38	-	1	● Online Yes

Figure 61. simple fanout Ingress loadbalancers (UI: configuration > Networking > Floating IPs)

totally 3 loadbalancers are generated in this test:

- loadbalancer **ns-user-1\_\_ingress-sf** for Ingress **ingress-sf**
- loadbalancer **ns-user-1\_\_webservice-1** for service **webservice-1**
- loadbalancer **ns-user-1\_\_webservice-2** for service **webservice-2**

we won't explore the details of each objects again this time since we've investigated the key parameters of **service** and **Ingress** loadbalancers in **single service Ingress** - there is really nothing new here.

## haproxy process and haproxy.cfg file

in **single service Ingress** example, we've demonstrated the two haproxy processes invoked by **contrail-svc-monitor** when it sees **loadbalancer** appears with **loadbalancer\_provider** set to **opencontrail**. in 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 killed. now with a new Ingress creation, two new haproxy processes are invoked again:

*Example 29. 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
```

*Example 30. 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
```

what interests us is how the **simple fanout Ingress** "rules" are programmed in the **haproxy.conf** file this time. 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 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

  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

backend 020e371c-e222-400f-b71f-5909c93132de
  mode http
  balance roundrobin
  option forwardfor
  server c13b0d0d-6e4a-4830-bb46-2377ba4caf23 10.96.51.227:8888 weight 1

backend 46f7e7da-0769-4672-b916-21fdd15b9fad
  mode http
  balance roundrobin
  option forwardfor
  server d58689c2-9e59-494b-bffd-fb7a62b4e17f 10.111.234.187:8888 weight 1

```

**NOTE** the configuration file is slightly formatted 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 is looks pretty straightforward.

- the haproxy **frontend** section: it now defines URLs. each URL is represented by a pair of **acl** statement, one for **host**, and the other for **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 are is a host **www.juniper.net** with two different paths: **\dev** and **\qa**.
- the haproxy **backend** section: now we see 2 of them. for each **path** there is a dedicated service.
- **use\_backend...if...** command in **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 ACLs pairs, "use" the corresponding "backend" (that is a service), to forward the traffic.

for example, `acl 020e371c-e222-400f-b71f-5909c93132de 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 **backend** section. The backend is a UUID referring to `server c13b0d0d-6e4a-4830-bb46-2377ba4caf23 10.97.77.82:8888 weight 1`, which essentially is a service. you can identify this by looking at the `serviceIP:port: 10.96.51.227:8888`.

this configuration file can be illustrated in this figure:

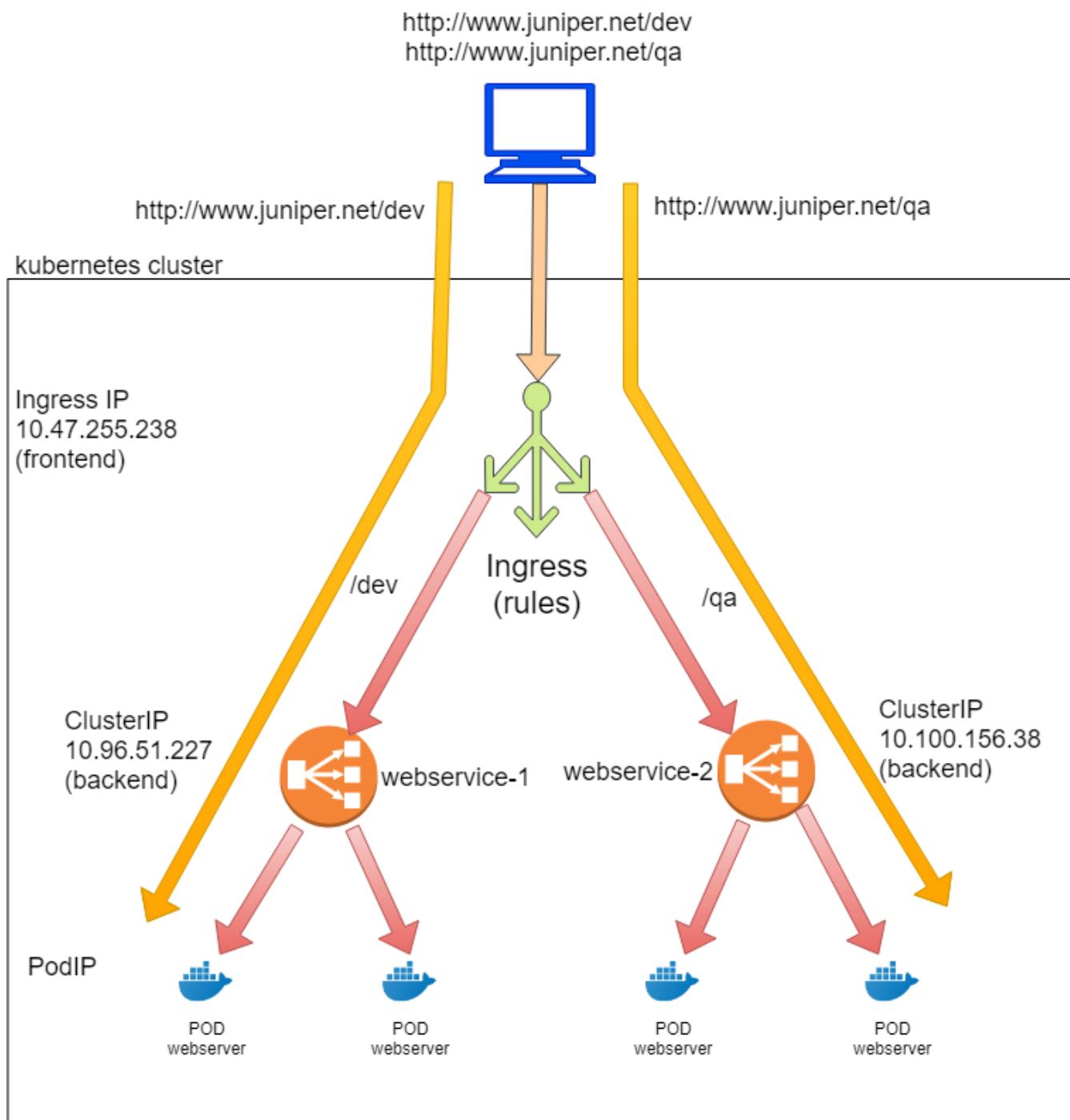


Figure 62. simple fanout Ingress

with this `proxy.conf` file, the haproxy implements our **simple fanout Ingress**:

- if the full URL composes **host** "www.juniper.net" and **path** "/dev" , request will be dispatched to **webservice-1 (10.96.51.227:8888)**
- if the full URL composes **host** "www.juniper.net" and **path** "/qa" , 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 we often need a the **default\_backend** service to process all those HTTP request with no matching URLs in the rules. we've seen it in the previous example of **single service Ingress**. later in **name-based virtual hosting Ingress** section we'll combine the **use\_backend** and **default\_backend** together to provide this type of flexibility.

### 6.5.3. Ingress verification: from internal

*Example 31. from inside of the cluster*

```
$ 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 return proves our **Ingress** works: the 2 requests towards "/qa" and "/dev" paths are proxied to 2 different backend pods, through 2 backend services **webservice-1** and **webservice-2** respectively.

the third request with a path `abc` composes a "unknown" URL which does not have a matching service in `Ingress` configuration, so it won't be served. same for the last 2 requests, without a path, or with a different Host, the URL become unknown to our Ingress so it won't be served.

you may think that we should add more rules to include these scenarios. doing that works fine but not scalable apparently - you can never cover all possible paths and URLs that could come into your server. as we mentioned earlier, one solution is to use `default_backend` service to process all "other" HTTP requests. we'll cover this in the next example.

#### 6.5.4. Ingress verification: from external (Internet host)

to test `simple fanout Ingress` from outside of the cluster, the command is the same as what we've seen when initiating the HTTP request from inside of a pod, except this time we are initiating from an Internet host. we will send the HTTP requests to the Ingress's public FIP, instead of its internal podIP.

from 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

$ 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

$ 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.
```

## 6.6. Virtual Hosting Ingress

`Virtual Hosting Ingress` support routing HTTP traffic to multiple host names at the same IP address. based on the URL and rules, an Ingress loadbalancer directs traffic to different backend services, and each service direct traffic to its backend pods.



to demonstrate **virtual host** type of Ingress, the objects that we need to create are same as previous **simple fanout Ingress**:

- an **Ingress** object: the rules, mapping 2 URLs to 2 backend services
- 2 backend services objects
- each service requires at least one pod as backend

### 6.6.1. Ingress objects definition

#### ingress definition

in our virtual host ingress test lab, we define the following rules:

- request toward URL **www.juniper.net** will be directed to a service **webservice-1** with **servicePort 8888**
- request toward URL **www.cisco.com** will be directed to a service **webservice-2** with **servicePort 8888**
- request toward any URLs other than these 2, will be directed to **webservice-1** with **servicePort 8888**. Effectively we want **webservice-1** to become the default backend service in here.

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*

same exact service and Deployment definition that were used in [simple fanout Ingress](#) can be used here.

#### **an "all in one" yaml file**

```

#ingress/ingress-virtual-host.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

```

*Example 32. apply the all-in-one yaml file to create Ingress and all 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

```

the **Ingress**, two services and two **Deployment** objects are now created.

## 6.6.2. Ingress post examination

### examine ingress objects

let's start to look at the **Ingress** object.

NAME	HOSTS	ADDRESS	PORTS	AGE
ingress-vh	www.juniper.net, www.cisco.com	10.47.255.248, 101.101.101.1	80	8m27s

comparing with **simple fanout Ingress**, this time we 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: 10.47.255.248
  kind: List
  metadata:
    resourceVersion: ""
    selfLink: ""
```

the rules are defined properly, within each rule there is a mapping from a **host** to the corresponding service.

the services, pods and FIP prefix advertisement to gateway router behavior are all exactly the same as those in **simple fanout Ingress**.

## exploring Ingress loadbalancer objects

3 loadbalancers are generated after we applied the all-in-one yaml file.

- 1 for Ingress
- 2 for services

loadbalancers created in this test is almost the same as the ones created in `simple fanout` Ingress test:

Name	Description	Subnet	Fixed IPs	Floating IPs	Listener	Operating Status	Admin State
ns-user-1_nginx-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

Figure 63. loadbalancers

next we can check haproxy configuration file for `name-based virtual host` Ingress.

### examine `haproxy.conf` file

```
$ cd /var/lib/contrail/loadbalancer/haproxy/8fd3e8ea-9539-11e9-9e54-0050569e6cfec/
$ 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+A
ESGCM:RSA+AES:RSA+3DES:!aNULL:!MD5:!DSS
    ulimit-n 200000
    maxconn 65000
    stats socket /var/lib/contrail/loadbalancer/haproxy/8fd3e8ea-9539-11e9-9e54-
0050569e6cfec/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 10.47.255.248:80          #<--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 10.101.158.92:8888 weight 1

backend 1e1e9596-85b5-4b10-8e14-44d1ca50a92f      #<--webservice-2
mode http
balance roundrobin
option forwardfor
server aa0cde60-2526-4437-b943-6f4eaa04bb05 10.104.4.232:8888 weight 1

backend cd7a7a5b-6c49-4c23-b656-e23493cf7f46      #<--default
mode http
balance roundrobin
option forwardfor
server e8384ee4-7270-4272-b765-61488e1d3e9c 10.101.158.92:8888 weight 1

```

here are the highlights:

- the haproxy `frontend` section defines each URL, or `host`, and its path. here the 2 hosts are `www.juniper.net` and `www.cisco.com`. 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 the `service` we've created.
- `use_backend...if...` command in `frontend` section declares the ingress rules: `if` the request includes a specified URL and path, "use" the corresponding "backend" to forward the traffic

- `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.

this configuration file can be illustrated in this figure:

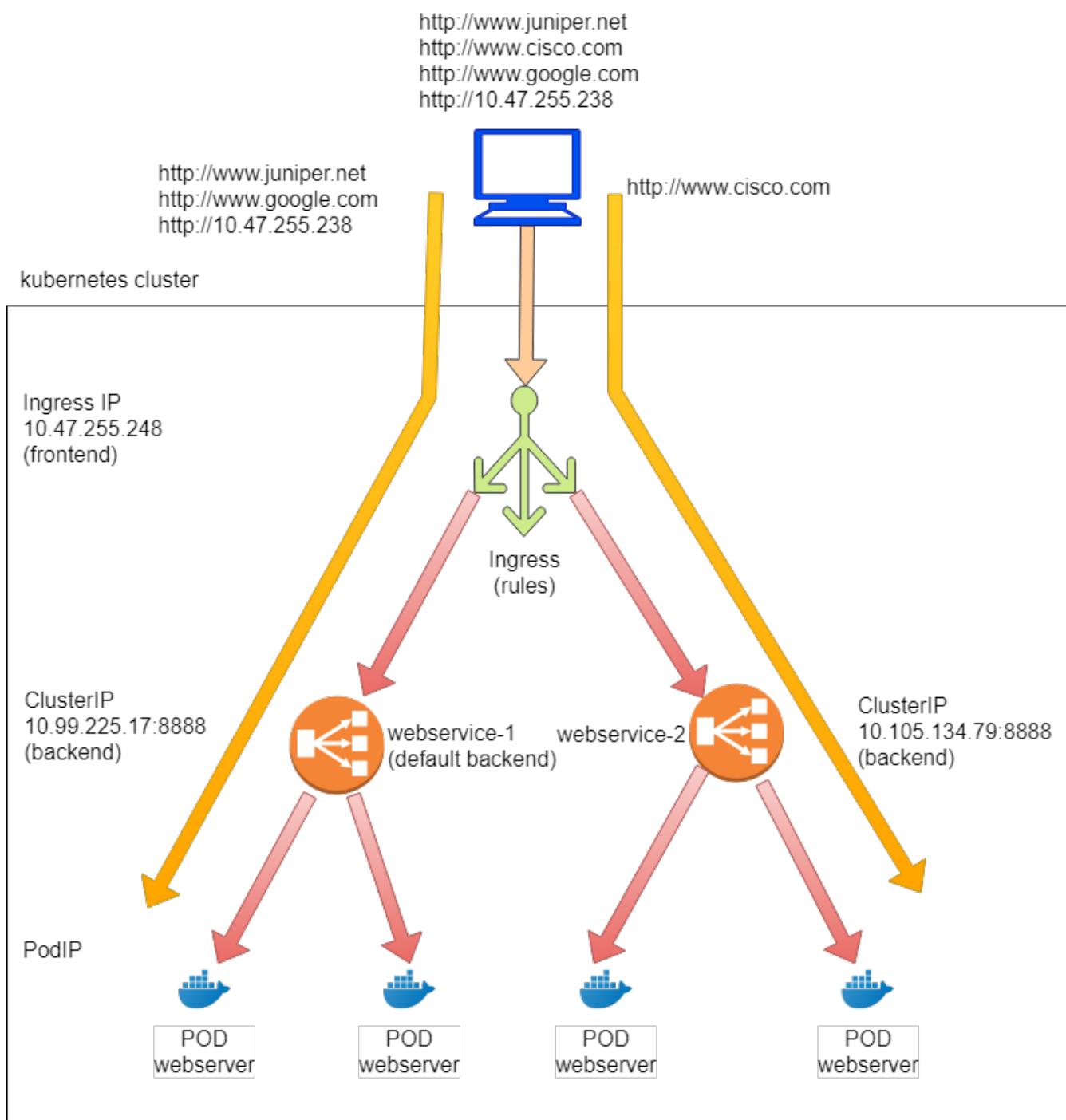


Figure 64. virtual host Ingress

through this configuration, the haproxy implemented our ingress:

- `www.juniper.net` and `/` composes the full URL, request will be dispatched to `webservice-1 (10.101.158.92:8888)`
- `www.cisco.net` and `/` composes the full URL, request will be dispatched to `webservice-2 (10.104.4.232:8888)`
- other URLs goes to default backend which is service `webservice-1`.

Next we'll verify these behaviors.

### 6.6.3. Ingress verification: from internal

*Example 33. 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 return proves our **Ingress** works: the 2 requests towards "juniper" and "cisco" URL is proxied to 2 different backend pods, through 2 backend services **webservice-1** and **webservice-2** respectively. the third request towards "google" is a "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.

same rule applies to the fourth request. without given a URL via **-H**, **curl** will fill **host** with the request IP address, **10.47.255.238** in this test, and since that "URL" does not have a defined backend service so the default backend service will be used. in our test, for each service we use backend pods spawned by same Deployment, so the podIP in returned webpage tells who is who. except in the second test the returned podIP **10.47.255.235** represent **webservice-2**, all other three tests returns podIP for **webservice-1**, as expected.

#### 6.6.4. Ingress verification: from external (Internet host)

From internet host's "desktop", we launch two chrome page side by side, and input URLs [www.juniper.net](http://www.juniper.net) and [www.cisco.com](http://www.cisco.com). keep refreshing the 2 pages we can confirm "juniper" page is always returned by Deployment `webserver-1` pod `10.47.255.236`, "cisco" page is always returned by Deployment `webserver-2` pod `10.47.255.235`. we launch a third chrome page and input [www.google.com](http://www.google.com), we see "google" page is returned by the same pod serving "Juniper" URL.

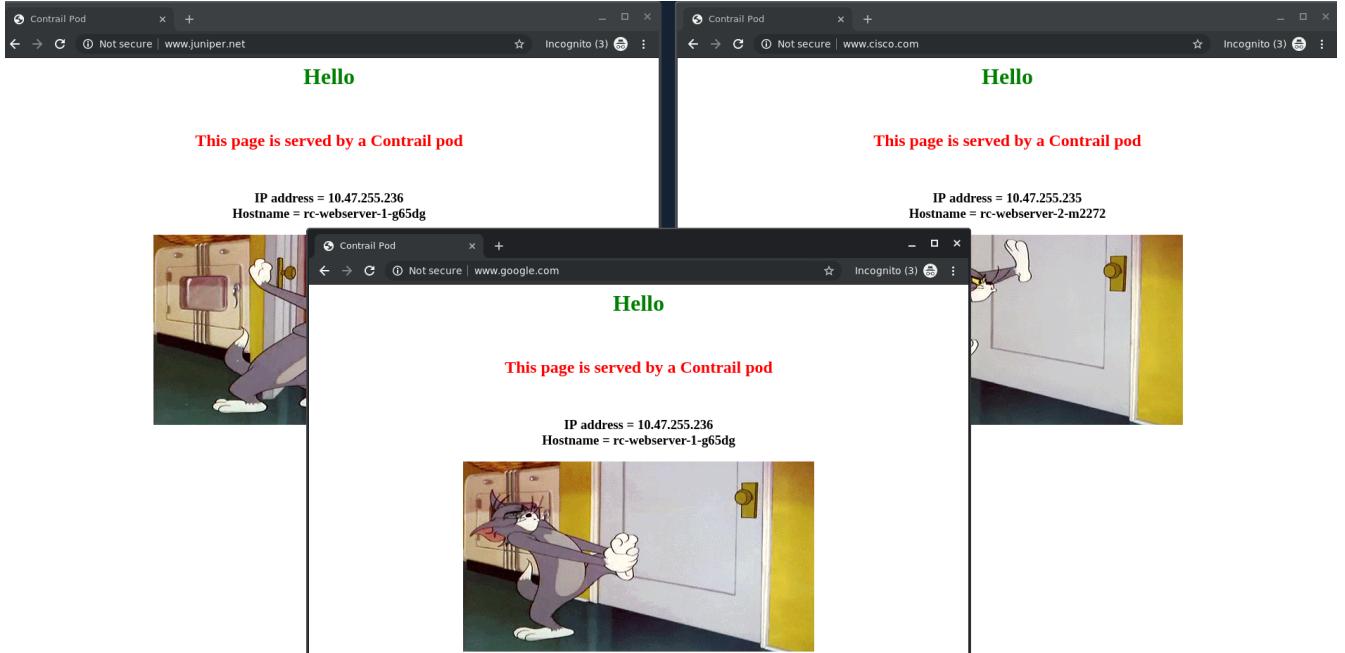


Figure 65. name based virtual hosting Ingress: access [www.juniper.net](http://www.juniper.net) from Internet host

same result can be seen from `curl` also.

*Example 34. from 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
```

## 6.7. Service vs Ingress Traffic Flow

in contrail, eventough both **service** and **Ingress** are implemented based via **loadbalancers** (but with different **loadbalancer\_provider** types). the forwarding mode for **service** and **Ingress** are quite different. with **service** forwarding it's "one hop" process: client sends request to the clusterIP or FIP, through NAT the request reaches the destination backend pod; while with **Ingress** forwarding, the traffic takes a "two hops" process to arrive the destination pod: the request first goes to the active haproxy, which then start a HTTP/HTTPS level proxy procedure and do the **service** forwarding to reach the final pod. NAT processing happens in both forwarding process, since both Ingress and service public FIP implementation relies on it. The next section will give a detailed view of packet flow for traffics in contrail.

# Chapter 7. chapter 7: Packet Flow in Contrail: End to End View

so far we've looked at **floating IP**, **service**, and **Ingress** in details. You probably found that all these objects are related to each other in certain aspects. in contrail, both **service** and **Ingress** are implemented based on **loadbalancers** (but with different **loadbalancer\_provider** types). conceptually, **Ingress** is designed based on **service**. VIP of both type of loadbalancers are implemented based on **floating IP**.

in order to understand the detail packet flow in contrail kubernetes environment, let's examine the end to end HTTP request from external Internet host to the destination pod in our **Ingress** lab setup. We'll examine the forwarding state step by step, starting from Internet host, through gateway router, then active haproxy, backend service and to the final destination pod.

after this chapter, you will get a deep understanding to the packet flow and you will be able to troubleshoot the forwarding plane problems in contrail kubernetes environment.

## 7.1. Setup and Utils/Tools

you've seen this figure in **Ingress** section:

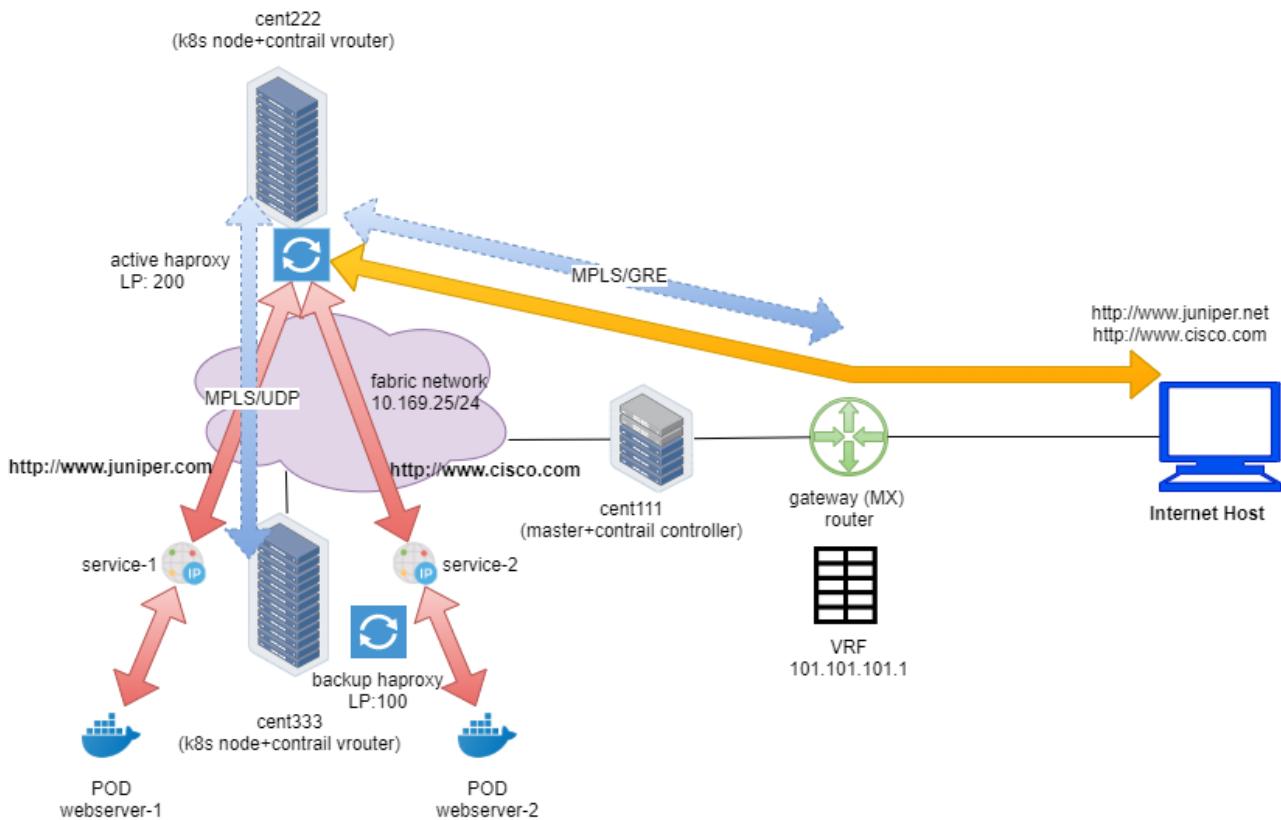


Figure 66. Ingress traffic flow: access from external

earlier we looked at the external gateway router's VRF routing table and use the **protocol next hop** information to find out which node get the packet from the client. in practice, very often you need to find out the same from the cluster and nodes themselves. a contrail cluster typically comes with a group of built-in utils that you can use to inspect the packet flow and forwarding state. in **service**

section you've seen the usage of `flow`, `nh`, `vif`, etc. in this section we'll revisit these utils and introduce some more useful utils/tools to demonstrate additional information about the packet flow.

here are some of the available utils/tools that we can use for this test:

- 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 vrouter: `flow/rt/nh/mpls/vif` and etc.

#### *curl*

one behavior in the `curl` tool implementation is that when it runs in a shell terminal, it will always close the TCP session right after the HTTP response is returned. although this is a safe and clean behavior in practice, it may bring some difficulties to our test. in this lab we actually want to "hold" the TCP connection so we can look into the details. However, a TCP flow entry in contrail `vrouter` is "bound" to the TCP connection - when TCP session closes the flow will also be cleared. the problem is, `curl` get its job done "too fast". it establish the TCP connection, send HTTP request, get the response, close the session right after. this 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 1 or 2 seconds. by the time you type in `flow` command in compute node, everything is done and you end up with no useful information. we actually prefer the connection to remain open for a while so we can take time to capture the flow table.

there are some methods to workaround that. here are some examples:

#### *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 "service" section.

#### *telnet*

we can also make use of `telnet` protocol. with it first we establish the TCP connection toward the URL's corresponding IP and port, then and manually input a few HTTP commands and headers to trigger the HTTP request. doing this allow you some period of time before the haproxy times out and tear down the TCP connection toward the client.

**NOTE** however, haproxy may still tear down its session immediately toward the backend pod. how haproxy behaves varies depending on haproxy's implementation and configurations.

from 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 '^]'.
```

now the TCP connection is established - we'll check what is the other end in a while. next we'll send HTTP **GET** command and **host** header:

```
GET / HTTP/1.1
Host: www.juniper.net
```

basically what it does is to send a HTTP **GET** request to retrieve data and **Host** provides the URL of the request. one more **return** indicates the end of the request, which will trigger a response from the server immediately:

```
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>
</body>
</div>
</html>
```

from this moment, we can collect flow table in active haproxy compute node for later analysis.

### *shell script*

a script can "type" much faster than we do. so another method is to 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 Internet host to keep sending request with **curl**, over the time we will have a good chance to have the flow table captured in

compute node at the right moment.

for instance, Internet host side script can be:

```
while :; do curl -H 'Host:www.juniper.net' 101.101.101.1; sleep 3; done
```

a compute side script may look like:

```
while :; do flow --match 10.47.255.238; sleep 0.2; done
```

first shell one-liner starts a new test every 3 second, then the second one captures a specific flow entry every 0.2 second. 20 tests can be done in a 2 minutes and we will capture some useful information in a short while.

In this section we'll use the "script method" to capture the required information from compute nodes.

## 7.2. Packet Flow Analysis

### 7.2.1. Internet Host: Analyze HTTP Request

*curl -v*

earlier we've used `curl` tool a lot to trigger HTTP requests for our test. `curl` supports extensive options for various features. we've seen `-H` option which specify the `host` field in a HTTP request. this time for debugging purpose we use another useful option `-v` in `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 =
  Vwebserver-1-846c9ccb8b-g65dg</h3>
</body>
</div>
</html>
* Connection #0 to host 101.101.101.1 left intact
```

with this option, it prints more verbose information about the HTTP interaction:

- > lines are the messages content that `curl` sent out
- < lines are message content that it receives from remote.

from the interaction we see:

- `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.

- the rest part of the response is the HTML source code of a returned web page.
- the connection is closed immediately afterward.

now you've seen the verbose interactions curl performed under the hood, and you can understand the GET command and host head we sent in `telnet` test - in that test we were just emulating what `curl` would do but we just did it manually!

## 7.2.2. Internet Host to Gateway Router

first let's start from the client - the Internet host.

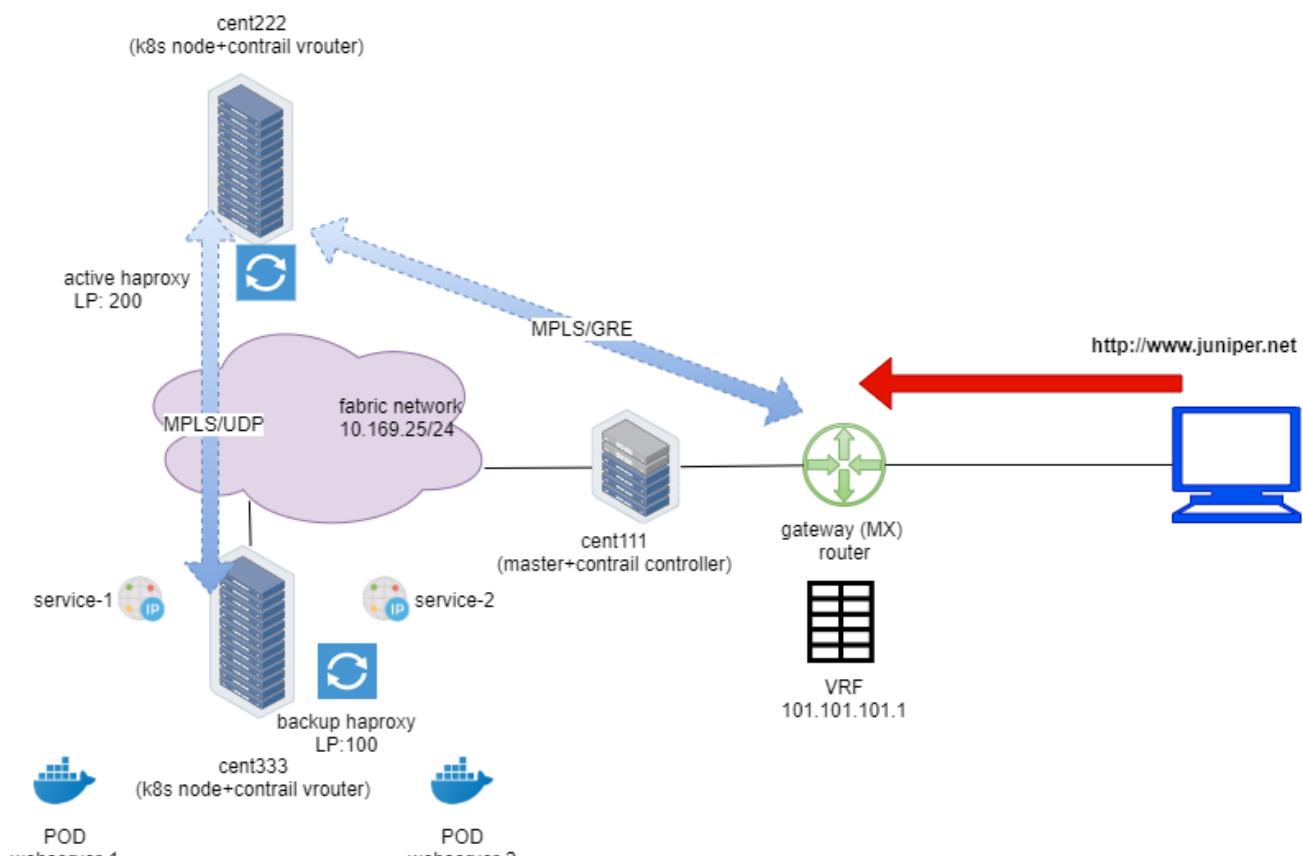


Figure 67. Internet Host: send a HTTP request

### Internet host routing table: default route

as in any host, the routing table is pretty simple. static route, or more typically a default route, pointing to gateway route is all what 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 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 VPN and Internet routing table, so that an Internet host which is not part of the VPN can also access the overlay networks in contrail.

### 7.2.3. Gateway router to Ingress Public FIP: MPLS over GRE

earlier in Ingress section we've seen gateway router's routing table, from the "protocol next hop" we can find out that the packet will be sent to active haproxy node **cent222** via MPLSoGRE tunnel.

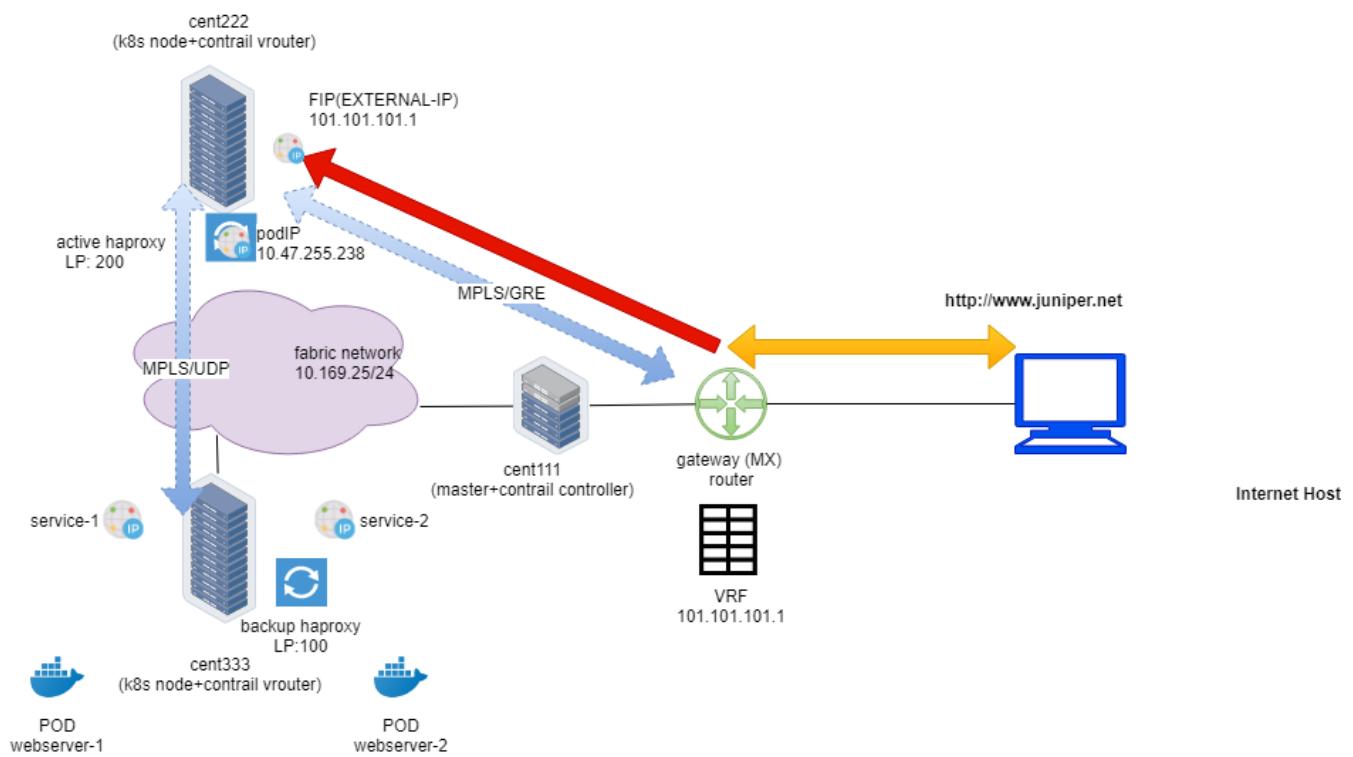


Figure 68. gateway router to Ingress public FIP

now with 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 1367 Processed 586803
Used Overflow entries 0
(Created Flows/CPU: 147731 149458 144549 145065)(offlows 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 101.101.101.1:80	6 (2->2)
	(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 15.15.15.2:42786	6 (2->5)
	(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 89
Id:89      Type:Encap          Fmly: AF_INET  Rid:0  Ref_cnt:7      Vrf:2
            Flags:Valid, Policy, Etree Root,
            EncapFmly:0806 Oif:8 Len:14
            Encap Data: 02 c0 0a c1 e6 6c 00 00 5e 00 01 00 08 00
```

This flow reflect the state of the TCP connection originated from Internet host client to active haproxy. let's first 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** FIP in current node **cent222**
- S(nh):61** is the next hop to the source of the request - the Internet host. this is similiar concept like the reverse path forwarding(RPF). vrouter always maintains the path toward the source of the packet in the flow.
- nh --get** command resolves the nexthop 61 with more details, we see a **MPLSoGRE** flag is set, **Sip** and **Dip** is the two end of the GRE tunnel, they are current node and gateway router's loopback IP respectively.
- TCP:SSrEEr** is TCP flags showing the state of this the TCP connection. vrouter detected the **SYN (S)**, **SYN-ACK (Sr)**, so 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 nexthop for a prefix in the VRF routing table.

overall the first flow entry confirms: the request packet from Internet host traverses gateway router, and via MPLSoGRE tunnel it hit the Ingress external VIP **101.101.101.1**. NAT will happen and we'll look into it next.

#### 7.2.4. Ingress Public FIP to Ingress Pod IP: FIP(NAT)

to verify the NAT operation, we 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 - **DNAT**. destination Ingress external FIP **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 - **SNAT**. indicate source NAT - **SNAT**, source IP **10.47.255.238** which is the internal internal Ingress VIP will be translated to the Ingress external VIP.

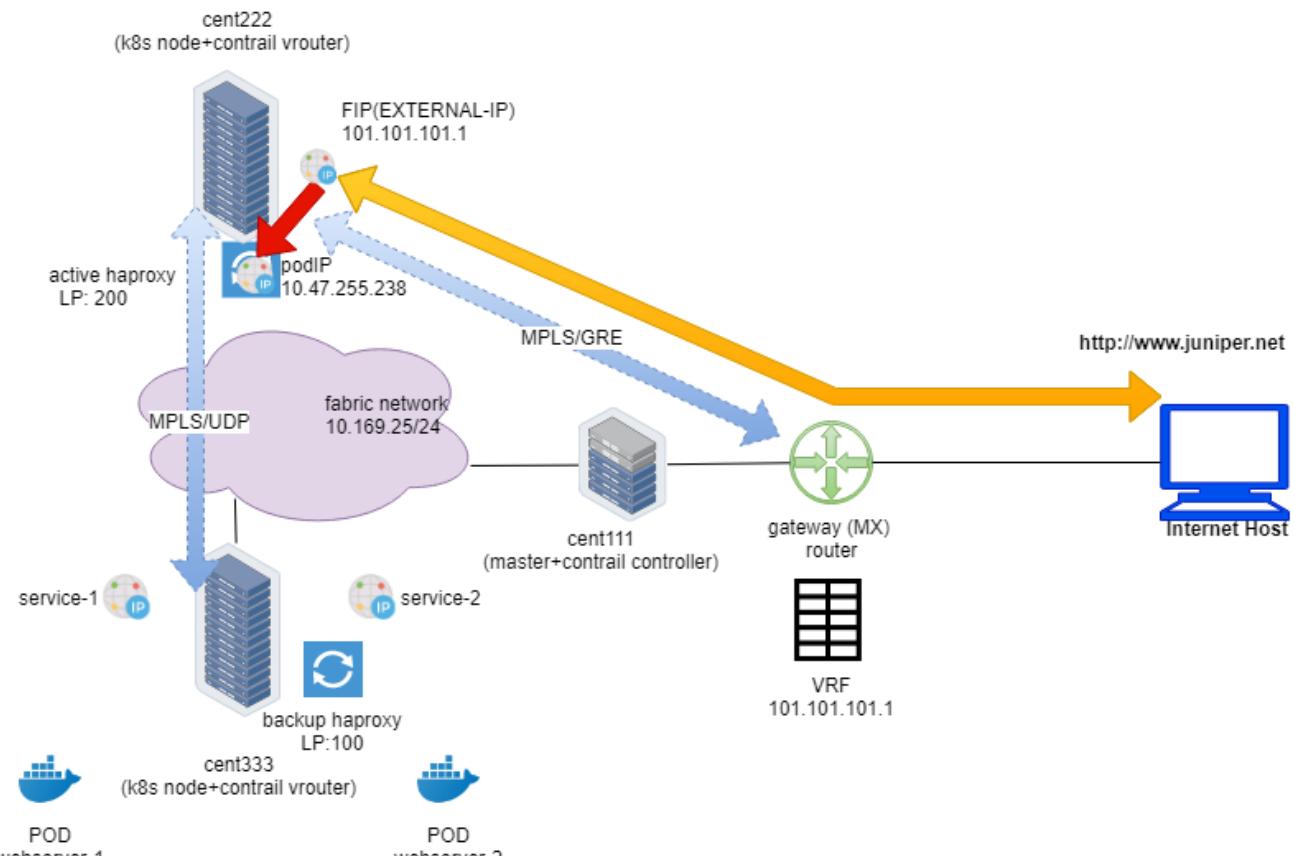


Figure 69. Ingress public FIP to Ingress Pod IP

in summary, what the flow table of active haproxy node **cent222** tells is that on receiving the packet destined to the Ingress FIP, vrouter on node **cent222** performs NAT operation and translates

destination FIP ([101.101.101.1](#)) to the Ingress's internal VIP ([10.47.255.238](#)). after that the packet lands the Ingress loadbalancer's VRF and forwarded to the active haproxy's listening interface. from this moment HTTP proxy operation will 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 uses to send returning packet towards Internet host. from Ingress loadbalancer'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 FIP. same thing for the external Internet host, it only knows how to reach the FIP and has no clues about the private Ingress internal VIP. it is vrouter that is doing the two way NAT translations in between.

### 7.2.5. Ingress Pod IP to Service IP: MPLS over UDP

now the packet lands in Ingress loadbalancer's VRF and it is in the frontend of the haproxy. what the haproxy supposes to do is:

- haproxy listening on the frontend IP (Ingress internal podIP/VIP) and port 80 see the packet.
- haproxy checks the ingress rule programmed in its config file, decides that the requests need to be proxied to service IP of [webservice-1](#).
- vrouter checks the Ingress loadbalancer's VRF table and sees the prefix of [webservice-1](#) service IP is learned from a destination node [cent333](#), which will be the next hop to forward the packet.
- between compute node the forwarding path is programmed with MPLSoUDP tunnel, so vrouter sends it through MPLS over UDP tunnel with right MPLS Label.

this part of the process is illustrated in below figure:

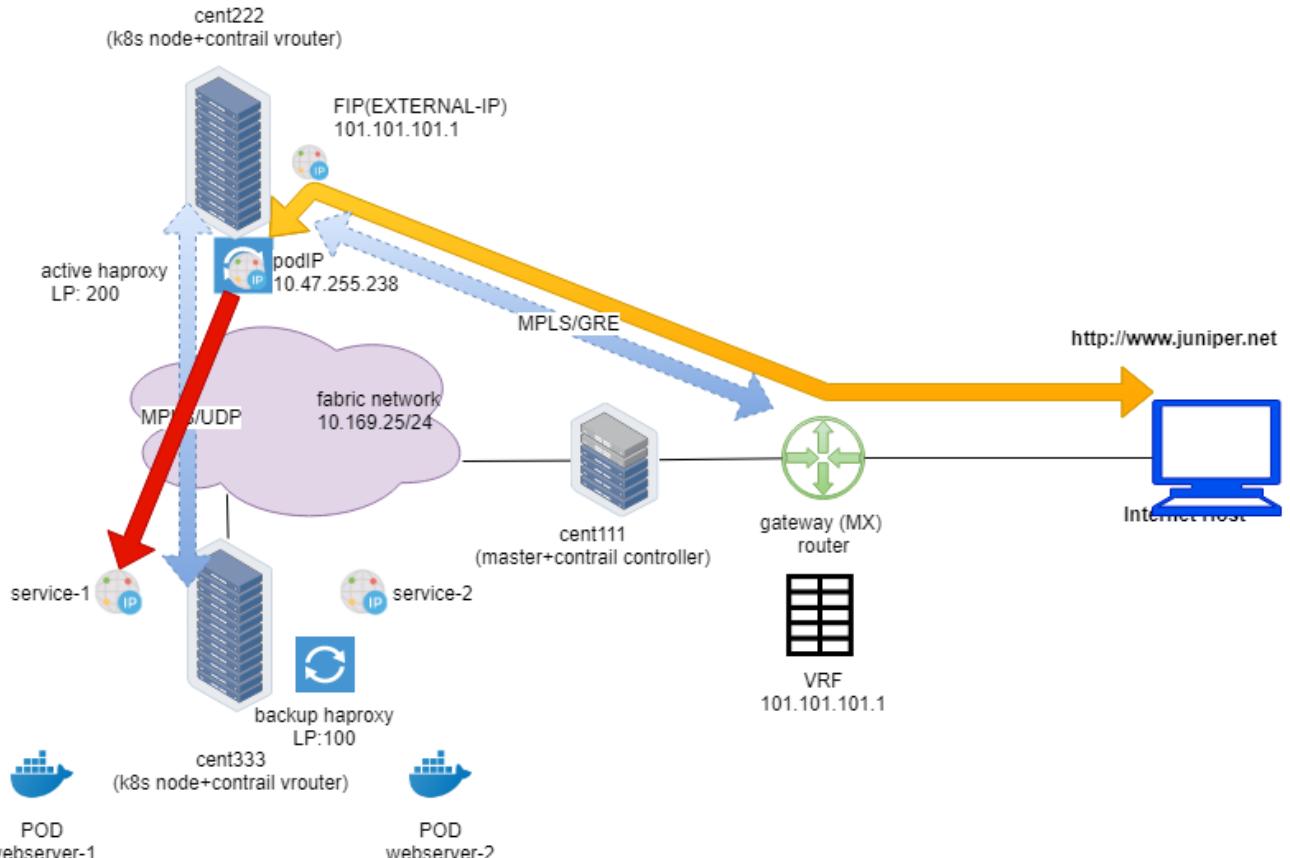


Figure 70. Ingress pod IP to service IP

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.

The screenshot shows the NMS interface for monitoring infrastructure. The left sidebar shows navigation paths: Monitor > Infrastructure > Virtual Routers > cent222. The main panel displays the VRF routing table for the selected VRF: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network:k8s-ns-user-1-pod-network. The table lists 13 routes:

Next hop Type	Next hop details	Prefix	Source	Policy	Peer	Valid
discard		10.32.0.0 / 12 (1 Route)	Local	disabled	Local	true
tunnel	Source IP: 10.169.25.20 Destination IP: 10.169.25.21	10.47.255.235 / 32 (1 Route)	10.169.25.20	MPLSoUDP	disabled	true
tunnel	Source IP: 10.169.25.20 Destination IP: 10.169.25.21	10.47.255.236 / 32 (1 Route)	10.169.25.20	MPLSoUDP	disabled	true
interface	Interface: tapeth0-0-5e04d8	10.47.255.237 / 32 (2 Routes)	default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network	disabled	Peer: 10.169.25.19	true
interface	Interface: tapeth0-0-5e04d8	10.47.255.237 / 32 (2 Routes)	default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network	disabled	Peer: LocalVmPort	true
interface	Interface: veth5aac5c08-1	10.47.255.238 / 32 (2 Routes)	default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network	enabled	Peer: 10.169.25.19	true
interface	Interface: veth5aac5c08-1	10.47.255.238 / 32 (2 Routes)	default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network	enabled	Peer: LocalVmPort	true
interface	Interface: pkt0	10.47.255.254 / 32 (1 Route)	default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network	enabled	Peer: Local	true
interface	Interface: pkt0	10.96.0.1/32 (1 Route)	default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network	enabled	Peer: Local	true
receive	Source: LinkLocal Destination VN: default-domain:k8s-ns-user-1:k8s-ns-user-1-pod-network	10.96.0.10/32 (1 Route)	LinkLocal	enabled	Peer: LinkLocal	true
ECMP Composite sub nh count: 2	Source IP: 10.169.25.20 Destination IP: 10.169.25.21	10.99.225.17/32 (1 Route)	10.169.25.20	disabled	Peer: 10.169.25.19	true
tunnel	Source IP: 10.169.25.20 Destination IP: 10.169.25.21	10.99.225.17/32 (1 Route)	10.169.25.20	MPLSoUDP	disabled	true
tunnel	Source IP: 10.169.25.20 Destination IP: 10.169.25.21	10.99.225.17/32 (1 Route)	10.169.25.20	MPLSoUDP	disabled	true

Figure 71. Ingress loadbalancer's VRF table

from the Ingress podIp's VRF, which is the same VRF for the default pod network of current NS, 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. 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 traffic from Ingress to service happens in overlay. between contrail compute nodes that means all overlay packets should be encapsulated in MPLS over UDP tunnel. to verify haproxy process packet processing details, we capture packets on the physical interface of node **cent222**, where the active haproxy process is running.

with a wireshark display filter `ip.addr == 10.47.255.238`, we can get the underlay TCP flow:

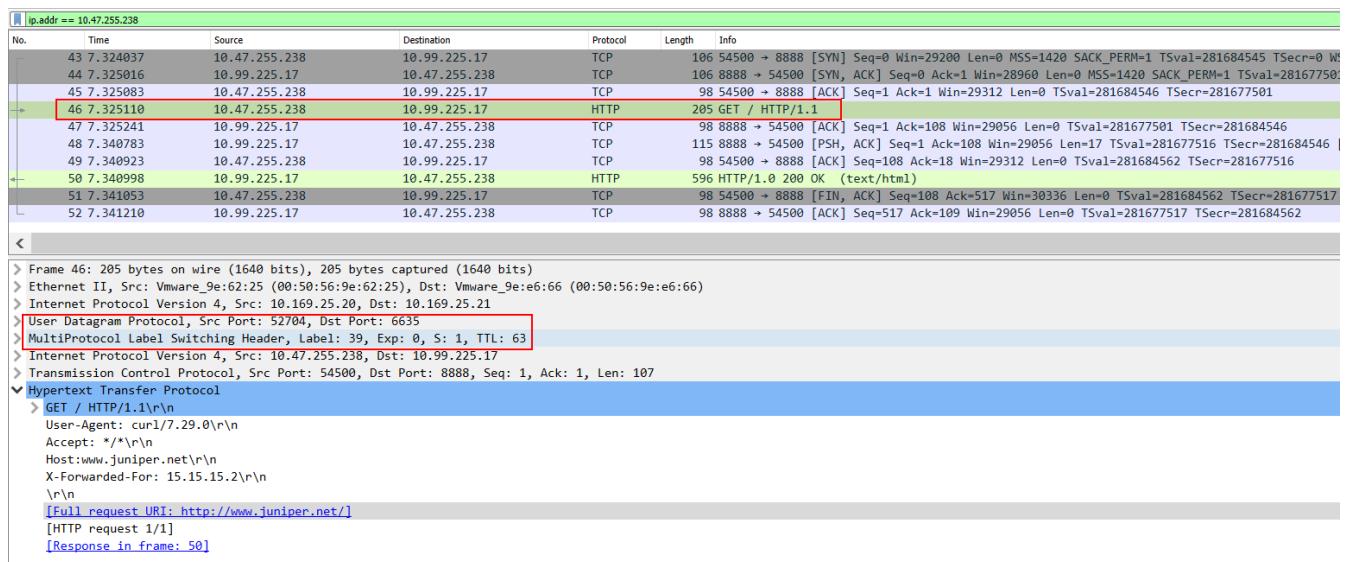


Figure 72. packet capture on active haproxy node **cent222** fabric interface

from the wireshark screenshot, we see clearly that:

- frame 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 start a HTTP request to the service IP.
- frame 50, the HTTP response returns back.

frame 46 is also the one we use as an example to show the packet encapsulation. you will 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" vs "proxy"*

if you are observant enough, you should have noticed something "weird" in this capture. questions are:

- shouldn't the source IP address be the Internet host's IP **15.15.15.2**, instead of loadbalancer's frontend IP?
- is the original HTTP request "forwarded" at all?
- is the transaction within the same TCP session sourcing from Internet host, accrossing gateway router and loadbalancer node **cent222**, all the way down to the backend pod sitting in node 'cent333`?

The answers are NO. the haproxy in this test is doing layer 7 (application layer) loadbalancing. what it does is:

- establish TCP connection with Internet host and keep monitoring the HTTP request.
- whenever it see a HTTP request coming in, it checks its rule and initiates a "brand new" TCP connection to the corresponding backend
- it "copies" the original HTTP request it received from Internet host and "paste" 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** will discover the "whole story":

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=0
44	7.325016	10.99.225.17	10.47.255.238	TCP	106	8888 → 54500 [SYN, ACK] Seq=0 Ack=1 Win=2
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 Win=29
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=303
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 Win=303
56	7.342129	101.101.101.1	15.15.15.2	TCP	94	80 → 42786 [FIN, ACK] Seq=540 Ack=80 Win=303
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
< 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 73. 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 send HTTP request
- frame 43-52: active haproxy establish a new TCP connection toward service, send the HTTP request, retrieve the HTTP response, and close the connection
- frame 53-54: active haproxy send the HTTP response back to Internet host
- frame 55-57: Internet host close the HTTP connection

here we use frame 42 to display the MPLS over GRE encapsulation between active haproxy node cent222 and gateway router. comparing 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 vrouter deliver the packet to the active haproxy. a new label will be assigned when active haproxy start a new TCP session to the remote node.

at the moment we know the http request is "proxied" to haproxy's backend. according to Ingress configuration that backend is a kubernetes **service**. now, to reach the **service** the request is sent to a "destination node" cent333 where all backend pod is sitting. next we'll look at what will happen in destination node.

## 7.2.6. Service IP to Backend Pod IP: FIP(NAT)

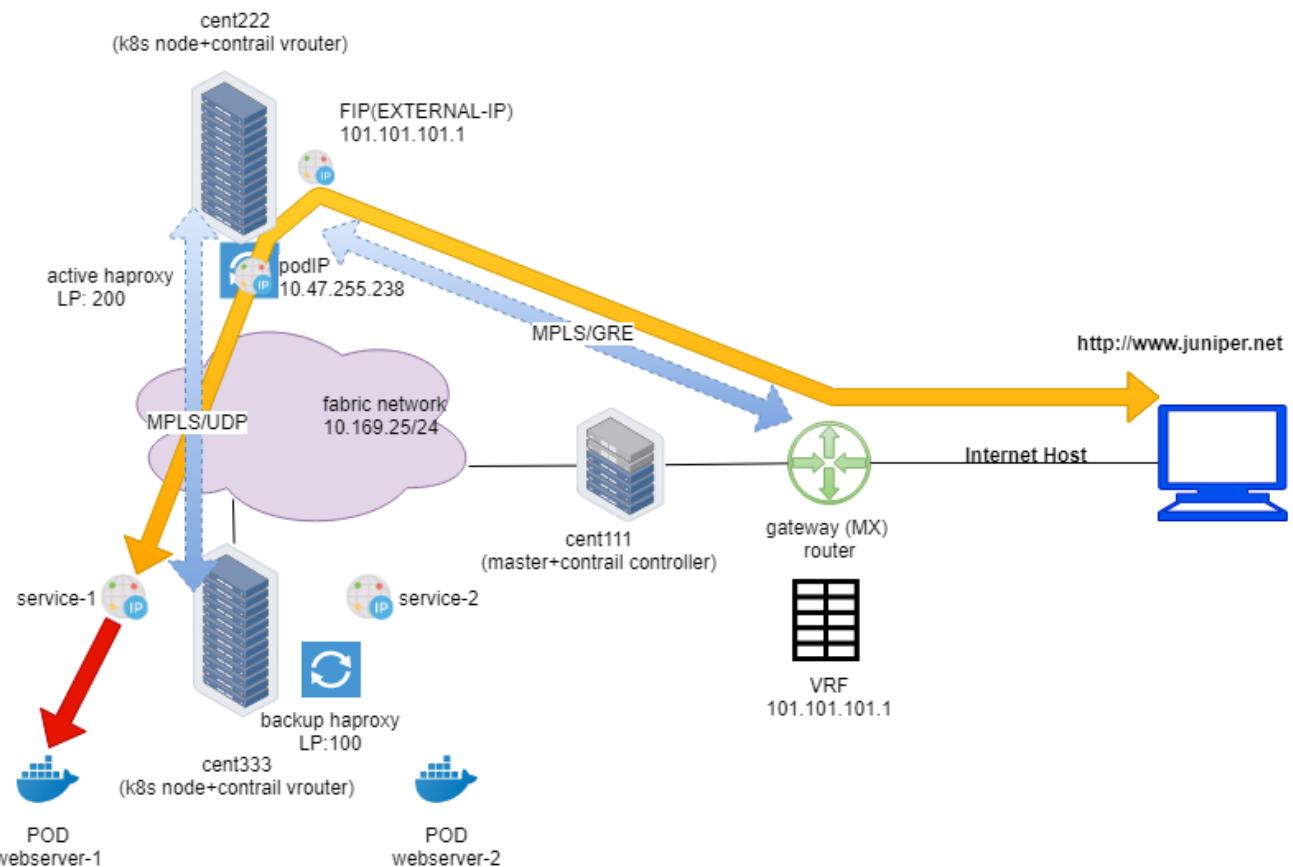


Figure 74. service IP to backend pod IP

on destination node `cent333`, when packet comes in from Ingress internal IP `10.47.255.238` toward the service IP `10.99.225.17` of `webservice-1`, vrouter again do the NAT translation operations. it translates the service IP to the backend podIP `10.47.255.236`, pretty much the same way as what we've seen in node `cent222`, where vrouter translate between Ingress public FIP with Ingress internal podIP.

here is the flow table we captured with 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 10 Processed 482 Used Overflow
entries 0
(Created Flows/CPU: 163 146 18 155)(offlows 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 10.47.255.238:54500	6 (2->4)
	(Gen: 1, K(nh):23, Action:N(SP <sub>s</sub> ), 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 10.99.225.17:8888	6 (2->2)
	(Gen: 1, K(nh):23, Action:N(DP <sub>d</sub> ), 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 [service](#) section so you may not have issues to read it through. obviously the second entry is triggered by the incoming request from active haproxy IP (the Ingress podIP) towards service IP. vrouter knows the service IP is a FIP that maps to the backend podIP [10.47.255.236](#), and service port maps to the container targetPort in backend pod. it does [DNAT+DPAT \(DP<sub>d</sub>\)](#) in incoming direction and [SNAT+SPAT \(SP<sub>s</sub>\)](#) in outgoing direction.

the other easy way to trace this forwarding path is to start from the MPLS label. in previous step we've seen label [38](#) is used when the active haproxy compute [cent222](#) sent packets into MPLSoUDP tunnel to compute [cent333](#), we can use 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 nexthop is determined, we can find the outgoing interface (`Oif`) number, then with `vif` utility we 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, destination backend 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:

- original source IP as backend podIP `10.47.255.236`
- original source port as webserver port `80`
- destination IP as `Ingress` internal podIP `10.47.255.238`

### 7.2.7. 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 backend podID.

```

$ 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
```

## 7.2.8. Return Traffic

on the reverse direction, podIP runs webserver and responds with it's web page. the response follows the reverse path of the request:

- pod responds to loadbalancer frontend IP, across MPLSoUDP tunnel
- vrouter on node **cent333** perform SNAT+SPAT, translating podIP:podPort into serviceIP:servicePort
- respond reaches to active haproxy running on node **cent222**
- haproxy "copies" the http response from backend pod, and "paste" into its connection with the remote Internet host
- vrouter on node **cent222** perform SNAT, translating loadbalancer frontend IP to FIP
- response is sent to gateway router, which forwards to Internet host
- Internet host gets the response.

# Chapter 8. chapter 8: Contrail Network Policy

in chapter 4, we've given the "Kubernetes to Contrail Object Mapping" table as shown below:

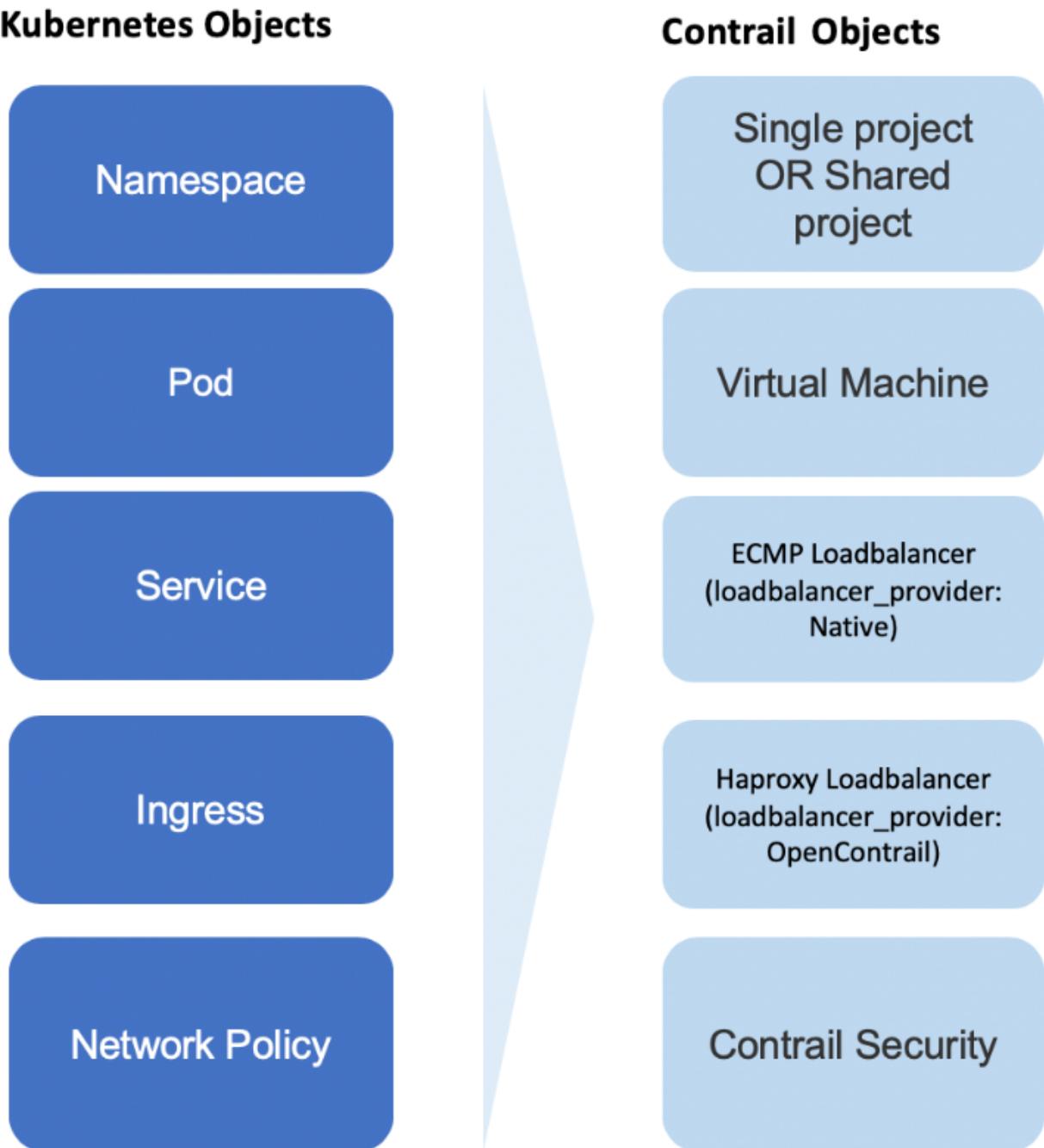


Figure 75. contrail kubernetes object mapping

the mapping highlights contrail's implementation of kubernetes core objects: **Namespace**, **pod**, **Service**, **Ingress** and **Network Policy**. so far from chapter 4 through 7 we've pretty much explored everything except **Network Policy**. in this chapter we'll focus on the **Network Policy** implementation in contrail. we'll first introduce the contrail firewall which is the feature we used to implement kubernetes network policy; we'll then setup a test case to verify how does kubernetes network policy works in contrail; based on the test result, in the end we'll explore the contrail firewall

policies and their rules in details to understand the contrail implementation, as well as the mapping between the two objects in the object mapping diagram.

## 8.1. introducing Contrail Firewall

In chapter 3 we introduced kubernetes network policy concept. we went through the yaml file definition in details and created the network policy based on it. we've also mentioned that simply creating network policy object won't have any effect, unless the kubernetes networking implementation support it. contrail as a kubernetes CNI implements the kubernetes networking, and it supports the kubernetes network policy through contrail firewall. that is the focus of this chapter - we'll demonstrate how network policy work in contrail environment through contrail firewall.

before that, let's review some of the important concept and design in contrail to understand why we implement kubernetes network policy through contrail firewall.

### *inter-VN routing*

in contrail, virtual networks (VN) are isolated by default. that means by default Workloads in **VN1** cannot communicate with workloads in another VN **VN2**. to allow inter-VN communications between **VN1** to **VN2**, additional configuration is required. for example you can use a neutron router, also called "logical router" in contrail, to connect the multiple VNs so inter-VN traffic can be "routed". another commonly used method is to define a "contrail network policy" to connect VNs. contrail network policy also provides security between two virtual networks by allowing or denying specified traffic. actually in this respect, security group is a similar feature. next we'll talk about each feature briefly.

### *contrail network policy*

A contrail network policy is used to permit inter-VN communication and to modify intra-VN traffic. it describes which traffic is permitted or not between VNs. by default, without a contrail network policy, intra-VN communication is allowed, but inter-VN traffic is denied. when you create a network policy you must associate it with a VN to have any effect.

NOTE: don't confuse "contrail network policy" with "kubernetes network policy". these 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 through a **port**. When a VM or pod is created in a VN, a **SG** can be associated with the VM when it is launched. unlike contrail network policy, which is configured "globally" and associated to the VNs, 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 SG and contrail network policy*

in modern contrail cloud environments, sometimes it is hard to only use the existing network policy and security group to achieve desired security goal. for example: in cloud environments, workloads may move from one server to another and so most likely the IP is changing often. 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 everytime 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 existing language of network policy or Security Group.

#### *contrail firewall security policy*

in this chapter we'll introduce another important feature: "contrail firewall security policy".

Contrail Firewall security policy allows decoupling of routing from security policies and 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 the 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 make it much simpler.

**NOTE** later in this chapter, we'll sometimes refer "contrail firewall security policy" as "contrail Security", "contrail firewall", "contrail firewall security" or simply "contrail FW".

## 8.2. contrail kubernetes Network Policy usage case

in this section, lets create a usage case to verify how does the kubernetes network policy works in contrail environments. we'll start from creating a few kubernetes namespaces and pods resources that is required in the test, confirming every pod can talk to the DUT (Device Under Test) because of the default "allow-any-any" networking model, then creating network policies and observing any changes with same traffic pattern.

### 8.2.1. network design

suppose we have this network design:

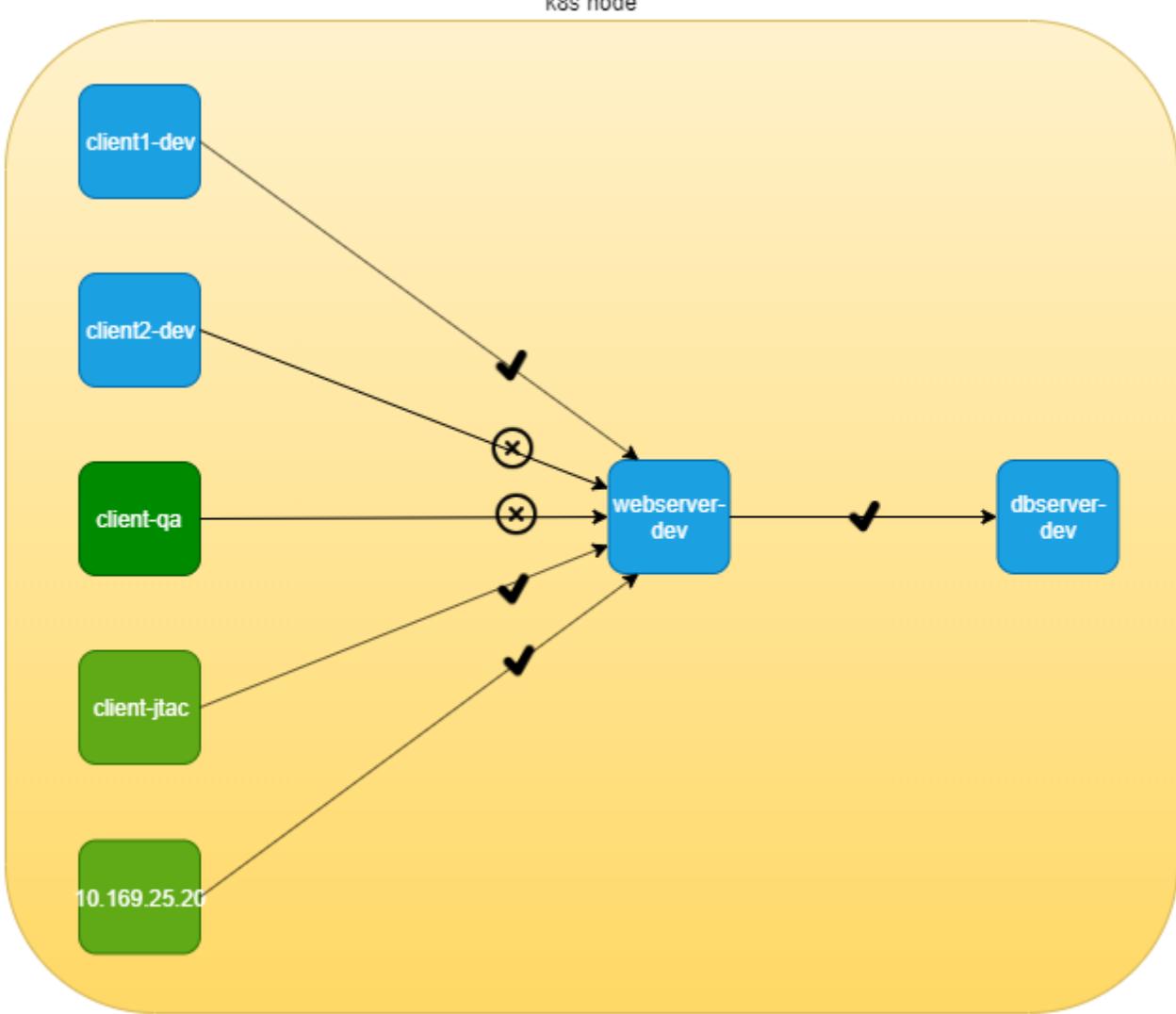


Figure 76. network policy: the test case design

in this diagram, 6 nodes are distributed in 3 departments: **dev**, **qa** and **jtac**. **dev** department is running a database server (**dbserver-dev**) holding all valuable data collected from customer. the design requires that no one should have direct access to this db server, instead, db server access is only allowed through another apache frontend server in **dev** department, named **webserver-dev**. furthermore, for security reason, the access of customer information should be granted only to authorized clients. for example, only nodes in **jtac** department, one node in **dev** department named **client1-dev** and source IP **10.169.25.20** can access the db via webserver. finally, the database server **dbserver-dev** should not initiate any connection toward other nodes.

### 8.2.2. lab preparation

this is a very ordinary, simplified network design that you will see everywhere. if we model all these network elements in kubernetes world, it will look like this:

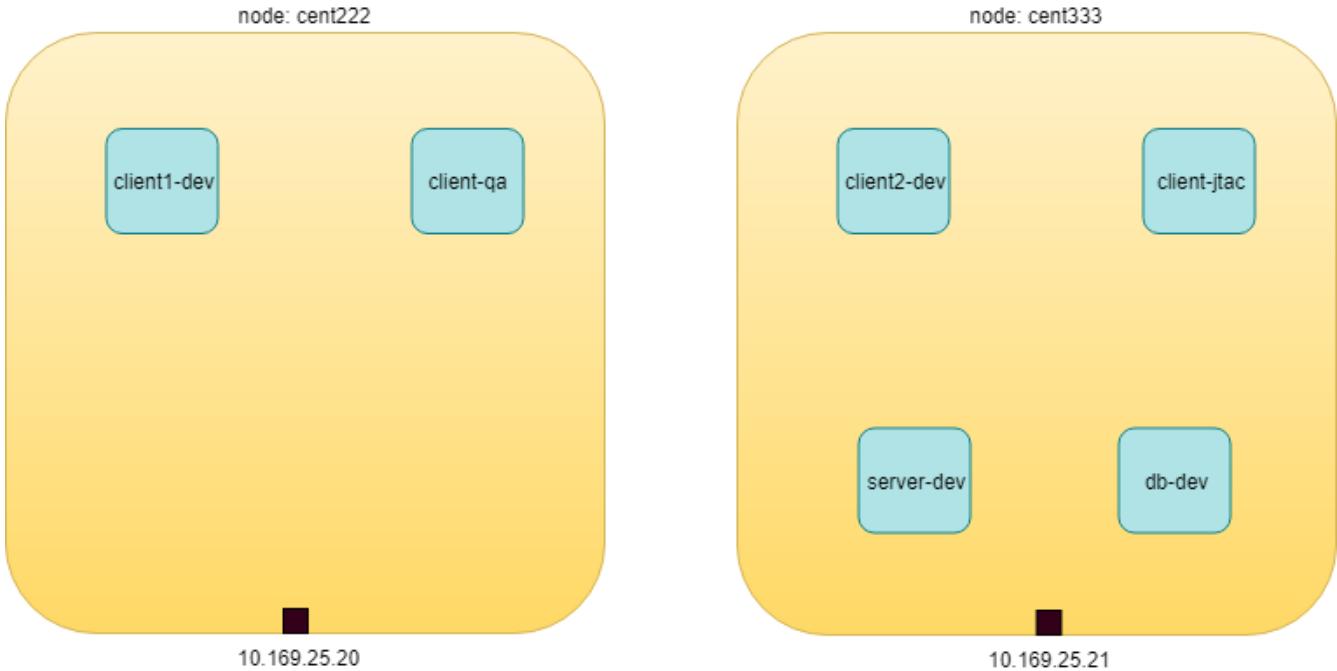


Figure 77. network policy: NS and pods

we need to create following resources:

- 3 namespaces: `dev`, `qa`, `jtac`
- 6 pods:
  - 2 server pods: `webserver-dev`, `dbserver-dev`
  - 2 client pods in the same namespace as of server pods: `client1-dev`, `client2-dev`
  - 2 clients pods from two different namespaces: `client-qa`, `client-jtac`
- 2 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 2. 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

Lets prepare the required k8s NS and pods resources.

here is the all-in-one yaml file defining `dev`, `qa` and `jtac` namespaces:

```
#####
# 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 pods may run based on different images. TCP ports usually are different between a webserver and a database server. in our case to make the test easier, we use the same exact `contrail-webserver` image that we've been using throughout the book for all pods, so clients to webserver and webserver to dbserver all share the same port number 80 served by same HTTP server. also, we add a label `do: policy` in all pods so that displaying all pods used in this test is also easier.

create all 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
```

### 8.2.3. traffic mode before kubernetes network policy creation

having all of the NS and pods, before we define any network policy yet, we can go ahead to send the traffic between clients and servers.

of course, kubernetes networking by default follows "allow-any-any" model, so we should expect access works between any pod, which is going to be a fully meshed access relationships. but keep in mind that the `DUT` in this test is `webserver-dev` and `dbserver-dev` which we are more interested to observe. to simply the verification, according to our diagram, we'll focus on accessing the server pods from the client pods, illustrated in below figure:

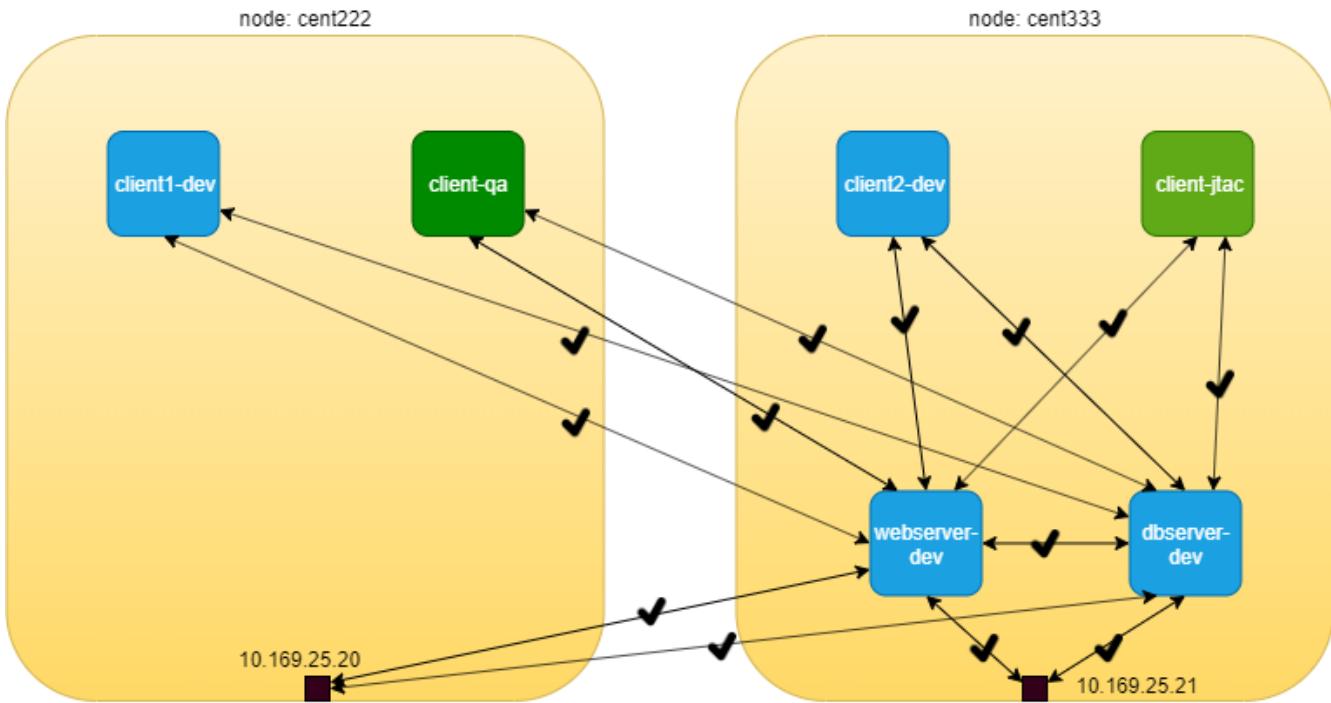


Figure 78. network policy: pods communication before network policy creation

a few highlights here:

- all clients can access the servers, following the "permit-any-any" model
  - there is no restrictions between clients and `webserver-dev` pod
  - there is no restrictions between clients and `dbserver-dev` pod
- the communication between client and servers are bi-directional and "symmetrical" - each end can "initiate a session" or "accept a session". these matches to the "egress policy" and "ingress policy" respectively in kubernetes network policy's term.

obviously, these do not meet our design goal, which is exactly why we need kubernetes network policy, 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.*      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.*      LISTEN      1/python
```

**TIP** as mentioned earlier, in this test all pods is with the same container image, so all pods are running the same webserver application in their containers. in this test we simply use the name to each pod to reflect their different roles in the diagram.

now we can verify accessing this http server from other pods with these commands:

*Example 35. `webserver-dev: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 triggers the HTTP requests to the `webserver-dev` pod from all clients and the hosts of the 2 nodes. `-m5` curl command option make curl to wait maximum 5 seconds for the response before it claims time out. as expected, all accesses pass through and returns the same output as below:

*Example 36. 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
```

in the command above, `w3m` get the output from curl which returns a webpage HTML code and renders into readable text, then send to grep to remove the empty lines. to make the command shorter we 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, we'll get the same test results for access to `dbserver-dev` from any other pods.

## 8.2.4. create kubernetes network policy

now lets 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` NS (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 language of kubernetes network policy, we'll have 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 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 towards `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
```

## 8.2.5. post kubernetes network policy creation

### ingress policy on `webserver-dev`

after network policy `policy1` is created, let's test the accessing of 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 2 pod to `webserver-dev` is OK and that is what we want. now if we repeat the same test from other pods `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 result after network policy applied is illustrated in this figure:

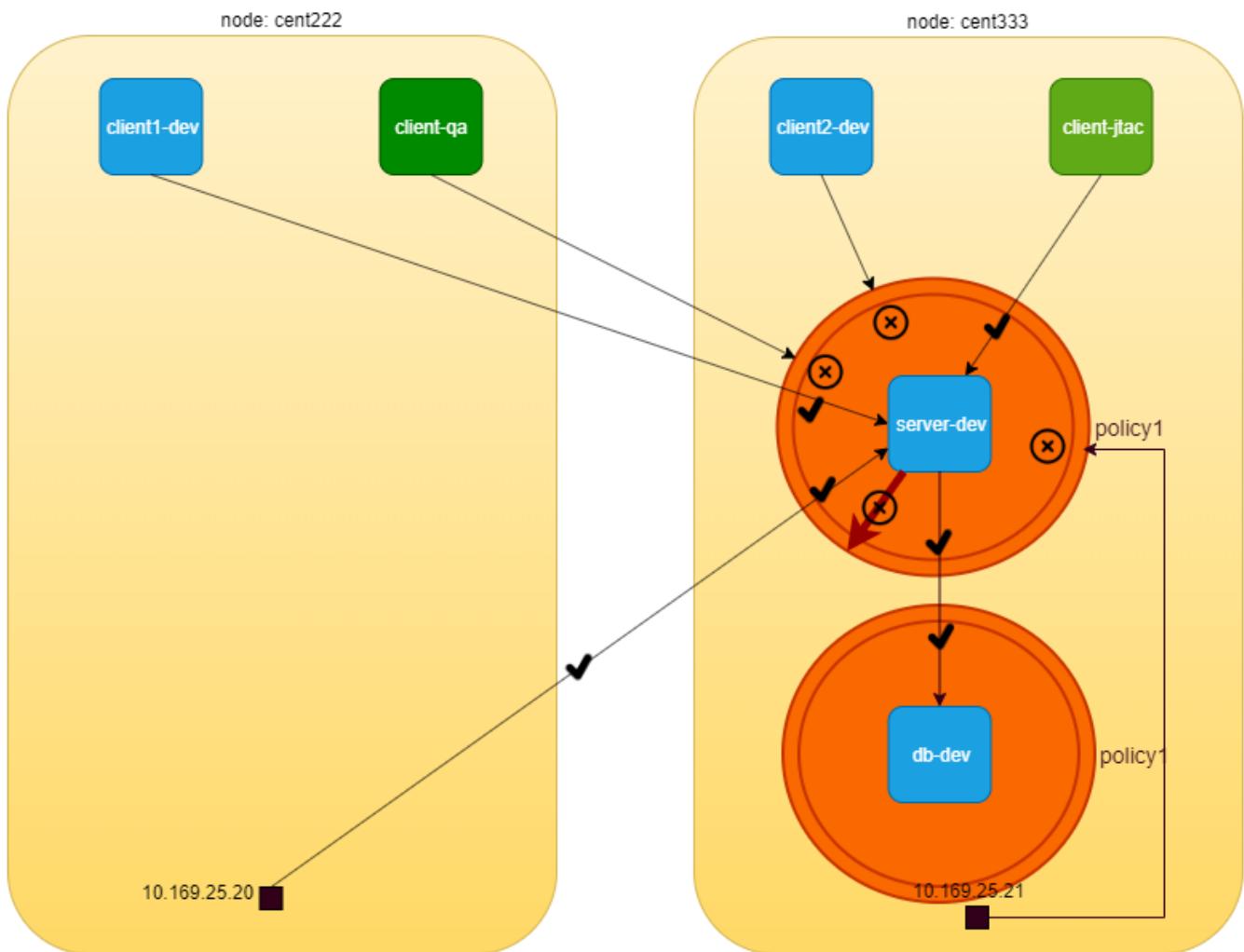


Figure 79. network policy: after applying policy1

detail information 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 perspective we've only tested that the ingress policy of `policy1` works successfully. secondly, 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 at last, 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:

- test 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**

*Example 37. test 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 only access to **dbserver-dev** succeeds. all other egress access is timed 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:

*Example 38. dbserver-dev:test ingress 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://$dbserverIP -m5
#from node cent333 (fabric interface IP: 10.169.25.21)
curl http://$dbserverIP -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 `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 network policy `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`

there is the one last requirement we haven't met in our designed goal. server `dbserver-dev` should not be able to initiate any connection toward other nodes. which is the final item we need to test in our plan.

now when you review our policy `policy2`, you may wonder how do we make that happen. we've highlighted in chapter 3 that network policy is "whitelist" based only by design. so whatever you put in the whitelist means "allowed". only a "blacklist" gives a "deny", but even with a blacklist you won't be able to list all other pods just to get them denied.

another thinking is to make the use of the "deny all" implicit policy. so assuming this is the sequence of policies in current kubernetes network policy design:

1. `policy2` on `dbserver-dev` to allow `webserver-dev`
2. "deny all" for `dbserver-dev`
3. "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 target pod will come into play. 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
  egress:      #<--
```

turn out this doesn't work as expected. the `dbserver-dev` can still initiate egress session.

```
$ 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 into 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
```

it does says "Selected pods are isolated for egress connectivity", which is what we want.

the problem is on the **policyTypes** here. we haven't added the **Egress** in. that is why in the object detail information above the "Policy Types: Ingress" is shown. whatever 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 policy `policy2` and apply this new policy, request 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
```

here is the final diagram illustrating our network policy test result, all egress session from `dbserver-dev` will be blocked.

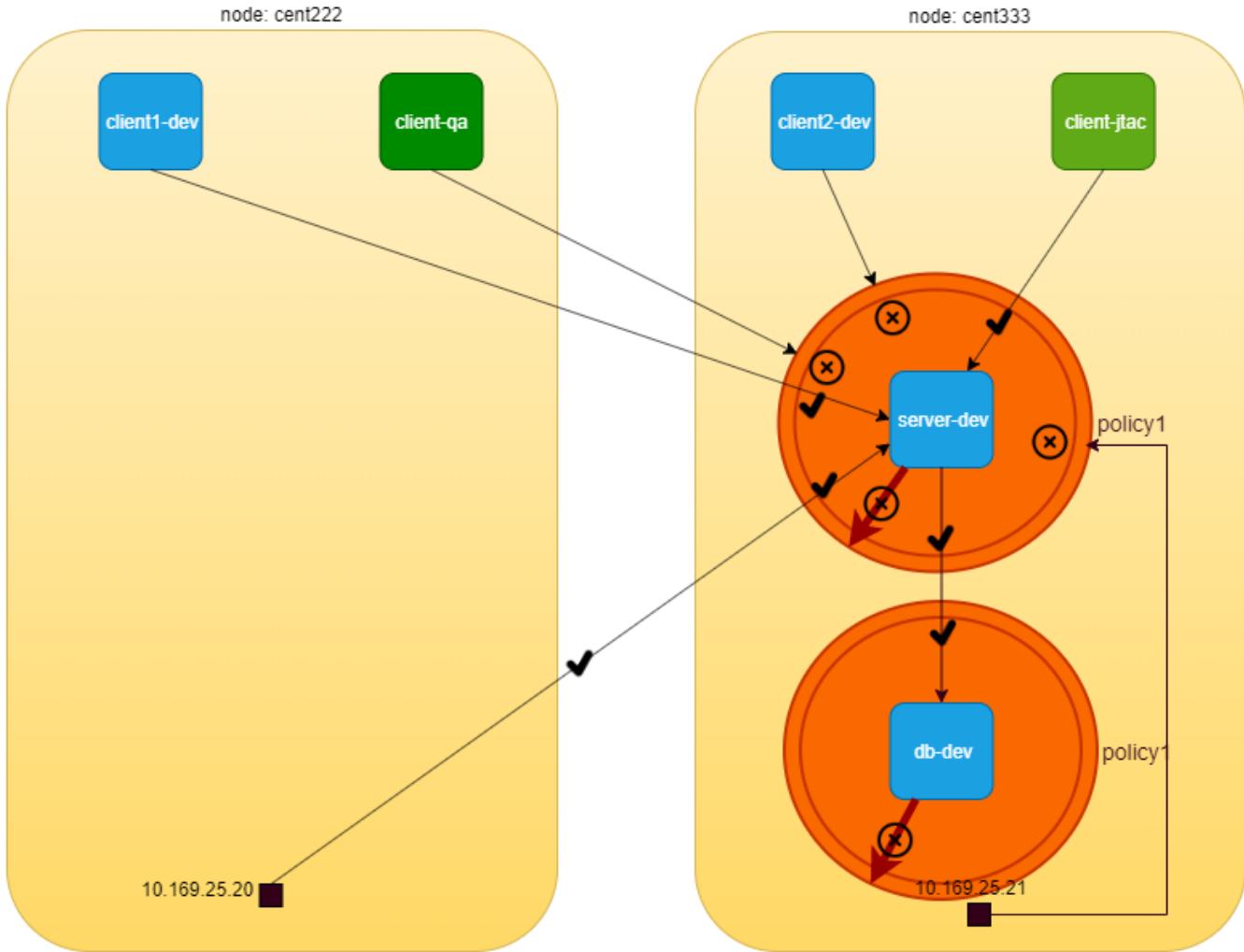


Figure 80. network policy: after applying an empty egress policy on `dbserver-dev` pod

### the drop action in flow table

before we conclude the test, lets take a look at the vrouter flow table when traffic is dropped by the policy.

check the flow table 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)(offlows 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 Firewall Policy. meanwhile, in the other node `cent222` where the destination pod `client1-dev` is located, we don't see any flow generated, indicating the packet does not arrive at all.

```

$ 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)
-----
```

## 8.3. contrail implementation details

we've introduce that contrail implements kubernetes network policy by Contrail firewall security Policy. you also knows 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 in the UI manually.

In this section we'll take a closer look at the contrail firewall policies, policy rules, and the tags. especially, we'll examine the mapping relationships between the kubernetes object 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 has the endpoints;
- Endpoints can be identified via tags or address groups (CIDRs).

the structure is illustrated in this figure:

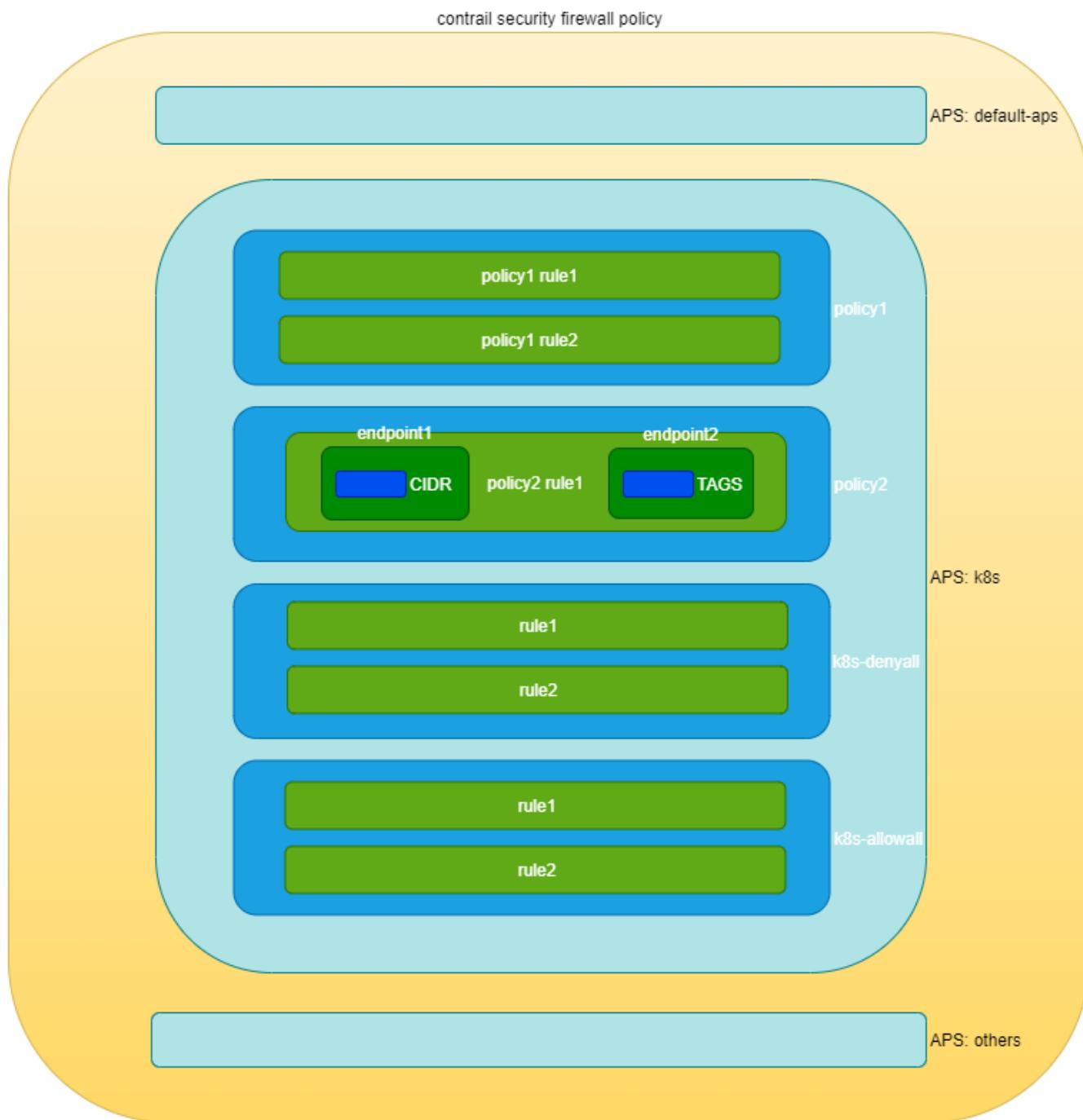


Figure 81. contrail firewall

### 8.3.1. construct mappings

Kubernetes network policy and contrail firewall policy are two different entities in terms of the semantics in which network policy is specified in each. 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.

Below table represents kubernetes network policy constructs and the corresponding constructs in contrail:

*Table 3. 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)

**contrail-kube-manager**, the **KM**, as we've read many time 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. **KM** will create a APS with 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. **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** in 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.

APS can be associated to different contrail objects, e.g.

- VMI(virtual machine interface)
- VM (virtual machine) or pods
- VN (virtual network)
- project

**NOTE**

In contrail kubernetes cluster, it 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 have created two kubernetes network policies in our usage case. now lets explore the contrail objects that are created for these kubernetes network policies.

### 8.3.2. Application Policy Set (APS)

As mentioned above, **contrail-kube-manager** will create an Application Policy Set(APS) using the kubernetes cluster name during the initialization stage. in chapter 3 when we introduce "Contrail Namespaces and Isolation", we've learned the cluster name is **k8s** by default in contrail. therefore the APS name will also be **k8s** in the contrail UI.

Name	Description	Application Tags	FW Policies	Last Updated
k8s	-	application=k8s	4	25 Sep 2019
default-application-policy-set	-	-	0	25 Sep 2019

Figure 82. contrail UI: APS: configure → Security → Global Policies → "Application Policy Sets"

There is one more APS **default-application-policy-set** which is created by default.

### 8.3.3. policies

now clicking on the "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 83. 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 looks different from the object name we gave in our yaml file, it is easy to tell which one 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 sounds familiar. earlier we've showed how KM name the VN in contrail UI after the kubernetes VN objects name we created in yaml file.

the `K8s-ingress` firewall policy is created for the ingress loadbalancer. basically this is to ensure the Ingress to work properly in contrail. the detail explanation is out of the scope of this book, so we can ignore it here.

## the `k8s-allowall` and `k8s-denyall` firewall policy

but the bigger question is that why we still see 2 more firewall policies here, because we had never created any network policies like `allowall`, or `denyall`?

remember when we introduce kubernetes network policy in chapter 3, we've mentioned kubernetes network policy uses "whitelist" method and the implicit "deny all" and "allow all" policies. the nature of "whitelist" method indicates "deny all" action for all traffic other than what is added in the whitelist, while the implicit "allow all" behavior make 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 these 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 explicitly 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 afterward? in another word, in what "sequence" shall contrail apply and evaluate each policy? firewall policies evaluation with a different sequence will lead to completely different result. just imagine these two sequences "denyall - allowall" vs "allowall- denyall", the former give a pass to all other pods, while the latter give a stop.

the answer is the "sequence number".

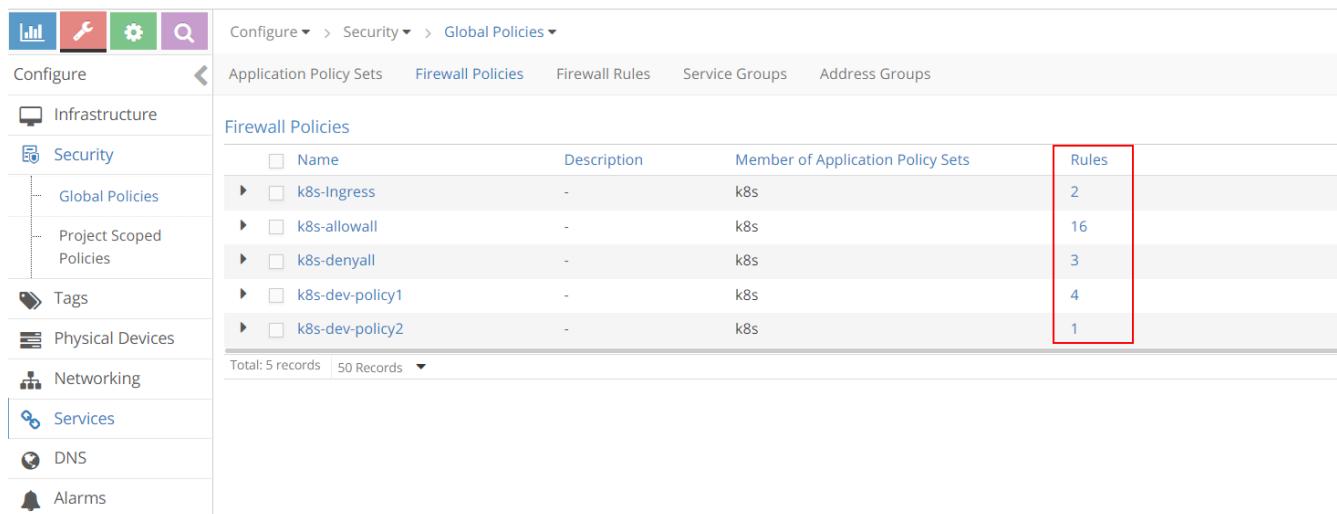
## sequence number

When firewall polices in an APS are evaluated, it has to be evaluated in a certain sequence. all firewall polices and all firewall rules (will come to this soon) in each of the policy 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 everythings works in correct order. the process is automatically done without manual intervention. we don't have to worry about these things when we create the kubernetes network policies.

we'll visit sequence number again later, now let's look at the rules defined in the firewall policy.

### 8.3.4. firewall policy rules

in the same view of "Firewall policies" list, in the right side we see number of "Rules" for each policy:



Name	Description	Member of Application Policy Sets	Rules
k8s-Ingress	-	k8s	2
k8s-allowall	-	k8s	16
k8s-denyal	-	k8s	3
k8s-dev-policy1	-	k8s	4
k8s-dev-policy2	-	k8s	1

Figure 84. contrail UI:"Firewall Policy rules"

#### rules in **k8s-dev-policy1** firewall policy

there are 4 rules for the **k8s-dev-policy1** policy. Clicking at it we will see the rules in detail:

Configure ▾ > Security ▾ > Global Policies ▾ > k8s-dev-policy1

**Firewall Policy : K8s-Dev-Policy1**

Policy Info   Rules   Permissions

**Firewall Rules**

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 85. contrail UI:"k8s-dev-policy1" rules

doesn't it look familiar with our kubernetes network policy **policy1** that we've tested? let's put the rules displayed in the screenshot into a table:

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 is the rule number that we added, all other columns are imported from the UI screenshot. now 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

```

all rules we see in firewall policy **k8s-dev-policy1** matches with rules in kubernetes network policy **policy1**.

### rules in **k8s-denyall** firewall policy

now let's go back and exmine the rules in **k8s-denyall** policy that **KM** generated for our kubernetes network policies.

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	-

Figure 86. contrail UI:"k8s-denyall" rules

again we convert that into a table:

rule #	Action	Services	End Point1	Dir	End Point2	Match Tags
1	deny	any:an y	app=webserver-dev && namespace=dev	>	any	-
2	deny	any:an y	any	>	app=dbserver-dev && namespace=dev	-
3	deny	any:an y	any	>	app=webserver-dev && namespace=dev	-

the **k8s-alldeny** rules are simple. it just tell contrail to deny communication with all other pods that is not in the whitelist. one thing worth to mention 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 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 when translated into contrail firewall there is no such a rule either. if we change **policy2** to the new policy **policy2-egress-denyall** and examine the same, we'll see the "missing" rule now:

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	-

Figure 87. contrail UI:"k8s-denyall" rules

please pay attention that the "k8s-denyall" policy only apply to those "target" pods - pods that are

selected by the network policies. in our case it only applies to pods `webserver-dev` and `dbserver-dev`. other pods like `client-jtac` or `client-qa` will not be affected. instead, those pods will be applied by `k8s-allowany` policy, which we will examine the next.

## rules in `k8s-allowall` firewall policy

the `k8s-allowall` policy seems to have more rules than other policies:

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	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	-

Figure 88. 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 UI within the "search" box, apply a namespace as the filter e.g. "jtac" or "qa", we'll see these results:

Action	Services	End Point 1	Dir	End Point 2	Match Tags
pass	any:any	namespace=dev	>	any	-
pass	any:any	any	>	namespace=dev	-

Figure 89. contrail UI:"`k8s-allowall`" rules filtered by NS "jtac"

Policy Info	Rules	Permissions					
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	-		

Figure 90. contrail UI:"k8s-allowall" rules filtered by NS "qa"

basically what this policy says is: for those pods that does not has any network policy applied yet, let's continue the kubernetes default "allow-any-any" networking model and allow everything!

### 8.3.5. sequence number

after having explored the contrail firewall policy rules, let's come back to the **sequence number** and see how it works exactly.

**sequence number** is a number attached in all firewall-policies and their rules. it decides the order in which all policies are applied and evaluated, similarly, in one particular policy, it decides the order in which all rules are applied and evaluated. the lower the sequence number the higher the priority. to find the **sequence number** we 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've used `curl` command to pull the loadbalancer object data when we introduce **service**. here we use "Config Editor" to do the same.

#### sequence number in firewall policies

The screenshot shows the Contrail UI Config Editor interface. The left sidebar has icons for Setting, Config DB, Introspect, and Config Editor, with 'Config Editor' selected. The main area shows the path: Setting > Config Editor > application-policy-sets > default-policy-management : k8s. The title is 'Application Policy Set'. The JSON configuration is displayed:

```
- {
  application-policy-set: - {
    display_name: "k8s",
    uid: "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-denyal"
        ],
        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"
        }
      }
    ]
  }
}
```

Red boxes highlight several sequence numbers: '00042.0' and '00038.0'.

Figure 91. contrail UI:sequence number for policies: "setting" → "Config Editor"

```

Setting > Config Editor > application-policy-sets > default-policy-management : k8s
Setting
Config DB
Introspect
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 92. contrail UI:sequence number for policies: (continue)

in these screenshots, under APS **k8s** there are all the 5 policies that we've saw. for example, the policy **k8s-dev-policy1** which maps to the kubernetes network policy **policy1** that we explicitly defined, and the policy **k8s-denyal** which is what the system automatically generated. in the figure it shows **k8s-dev-policy1** and **k8s-denyal** has a sequence number as "00038.0" and "00042.0" respectively. therefore, **k8s-dev-policy1** has a higher priority and it will be applied and also evaluated first. that means the traffic types we defined in 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 want to achieve.

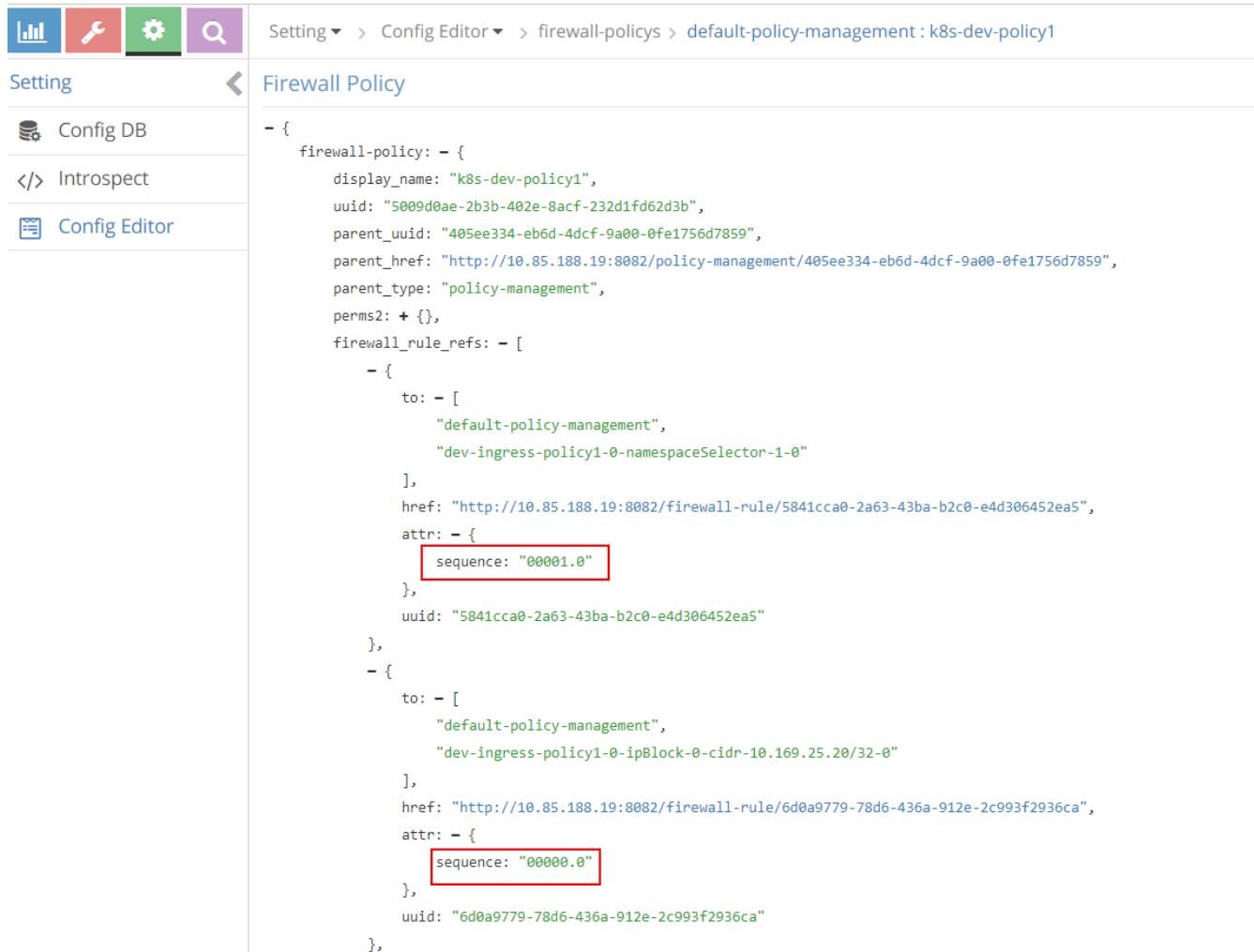
all **sequence number** for all firewall policies are listed in below table, from highest priority to the lowest:

seq#	firewall policy
0002.0	k8s-Ingress
00038.0	k8s-dev-policy1
00040.0	k8s-dev-policy2
00042.0	k8s-denyal
00043.0	k8s-allowall

based on the sequence number, the application and evaluation order is the explicit policies first, followed by the "deny all" policy and then the "allow all" policy at the last. the same order as in kubernetes is honored. next let's check the sequence number in policy rules.

## sequence number in firewall policy rules

as mentioned, 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 number in rules of firewall policy **k8s-dev-policy1** is displayed in below figures:



The screenshot shows the Contrail UI Config Editor interface. The left sidebar has tabs for Setting, Config DB, Introspect, and Config Editor, with Config Editor selected. The main area is titled "Firewall Policy" and displays the JSON configuration for a firewall policy named "k8s-dev-policy1". The "firewall\_rule\_refs" section contains two entries, each with a "sequence" attribute highlighted with a red box. The first entry has "sequence: \"00001.0\"", and the second entry has "sequence: \"00000.0\"".

```
Setting > Config Editor > firewall-policies > 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"
                },
                uid: "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"
                },
                uid: "6d0a9779-78d6-436a-912e-2c993f2936ca"
            }
        ]
    }
}
```

Figure 93. contrail UI:sequence number for rules: "setting" → "Config Editor"

```

Setting > Config Editor > firewall-policies > 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-e61fd1cbcfffb",
  attr: - {
    sequence: "00002.0"
  },
  uid: "519637ed-efb3-44e1-908f-e61fd1cbcfffb"
},
- {
  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"
  },
  uid: "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 94. contrail UI:sequence number for rules: (continue)

below table shows **sequence number** of all rules of firewall policy **k8s-dev-policy1**, from highest priority to the lowest:

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

comparing with our network policy yaml file configuration:

```

ingress:
- from:
  - ipBlock:
    cidr: 10.169.25.20/32          #<---seq# 00000.0
  - namespaceSelector:
    matchLabels:
      project: jtac
  - podSelector:                  #<---seq# 00001.0
    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 can find that the rules **sequence number** is consistent with the sequence they appear in the yaml file. in another word, rules will be applied and evaluated in the same order as they are defined.

### 8.3.6. 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.

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=client-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)	-	-

### 8.3.7. UI visualization

Contrail UI provides nice visualization for security. It is self explanatory if you know how contrail security works.

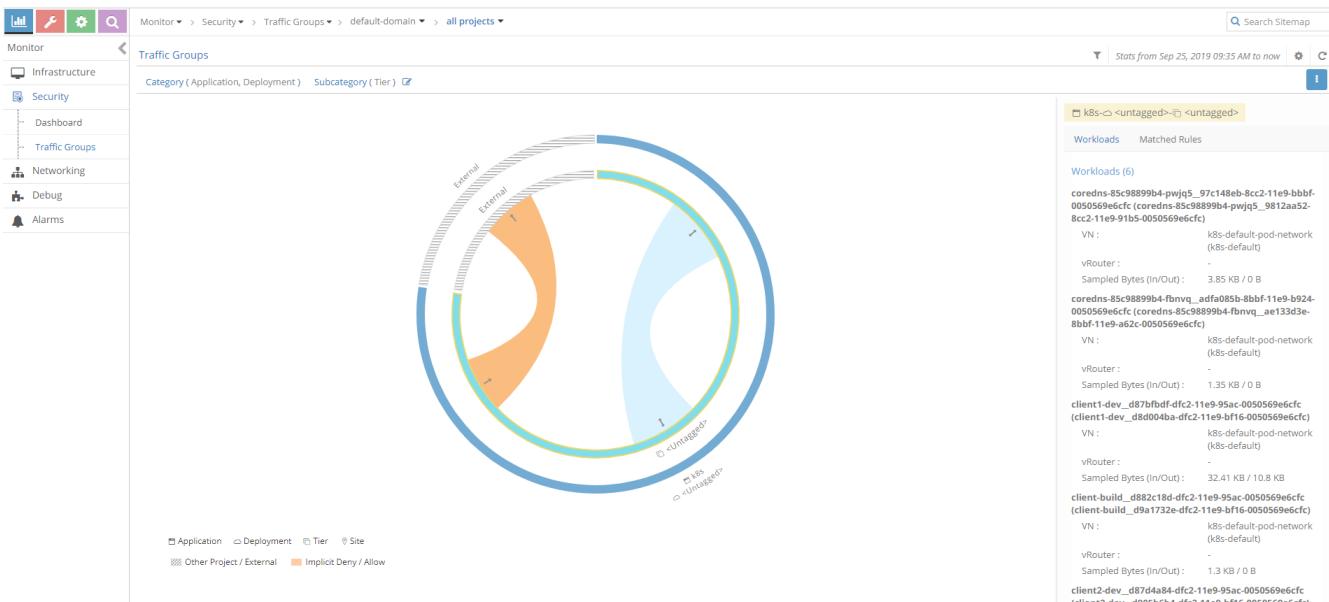


Figure 95. Sample traffic visualization for the above policy with workload

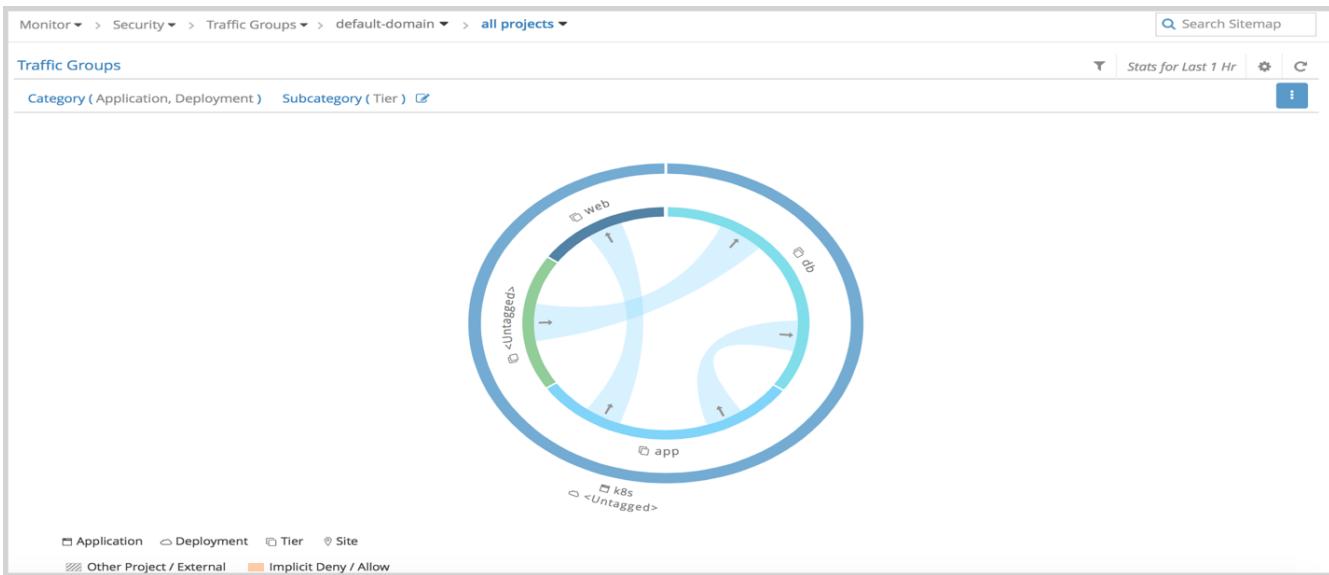


Figure 96. Sample traffic visualization with more network policies

# Chapter 9. chapter 9: Contrail Multiple Interface Pod

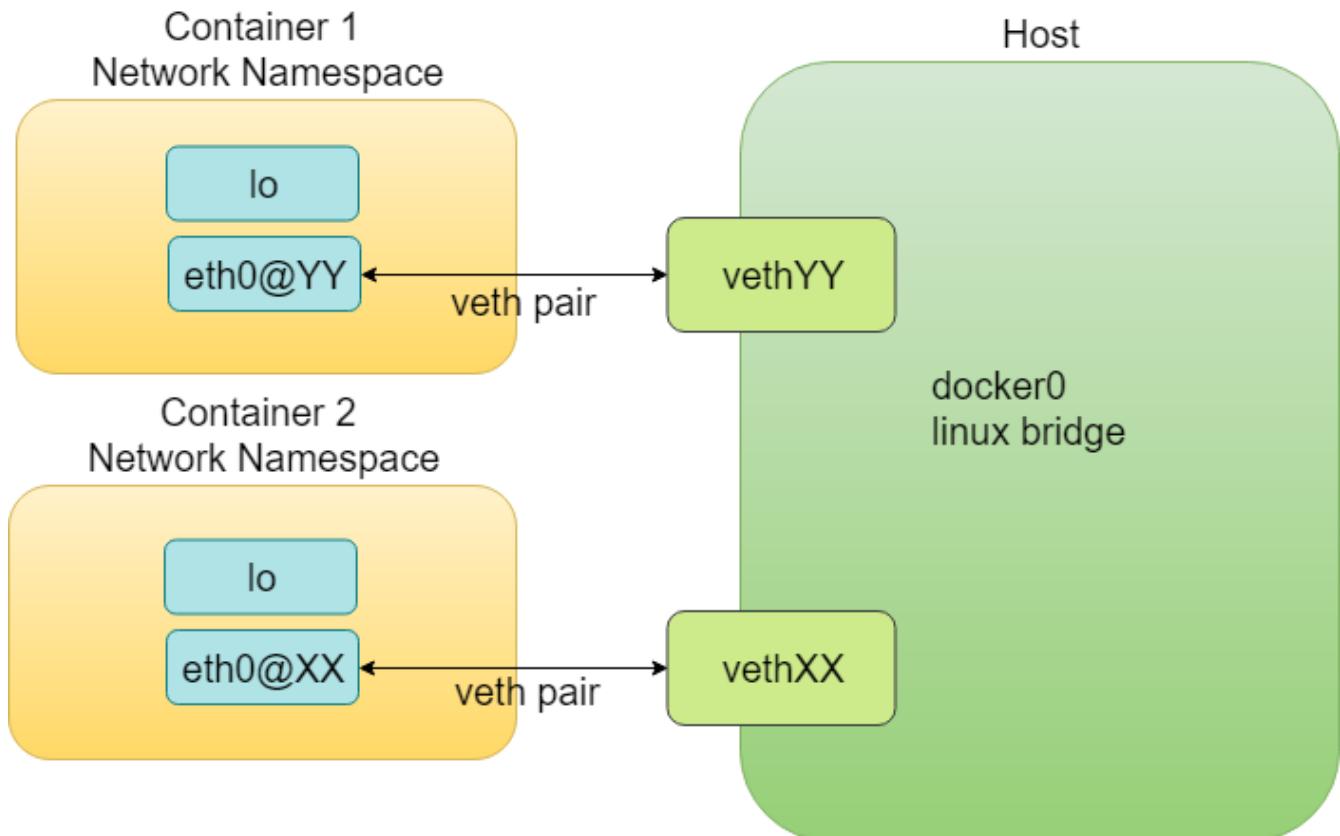
in Kubernetes cluster, typically each pod only has one network interface (except the [loopback](#) interface). In reality, there are scenarios where multiple interfaces are required. e.g. a vnf(virtual network function) typically needs a "left", "right" and optionally a "management" interface to do network fucntions. a pod may requires 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 purpose. Multiple interfaces provide a way for containers to be connected to multiple devices in multiple networks simultaneously.

## 9.1. Contrail as a CNI

in container technology, A veth(Virtual ETHernet) pair is functioning pretty much like a virtual "cabel", 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 "CNI plugin" is the one who is responsible for inserting the network interface (that is one end of the veth pair) into the container network namespace and it will also makes all necessary changes on the host. e.g. attaching the other end of the veth into a bridge, assigning IP, configuring routes, and so on.

*container and veth pair*



there are many such CNI plugin implementations that are publicly available today. contrail is one of

them. for a comprehensive list you can check <https://github.com/containernetworking/cni> where contrail is also listed.

**multus-cni**, is another CNI plugin that "enables attaching multiple network interfaces to pods". multipe-network support of **multus-cni** is accomplished by Multus calling multiple other CNI plugins. because each plugin will create its own network, multiple plugins make the pod be able to have multiple networks. one of the main advantages that contrail provides, comparing with **mutus-cni** and all other current implementations in the industry, is that contrail by itself provides the ability to attach multiple network interfaces to a kubernetes pod, without the need to call any other plugins. this brings support to a truly "multi-homed" pod.

## 9.2. NetworkAttachmentDefinition CRD

contrail CNI follows the Kubernetes Network **CRD**(Custom Resource Definition) - NetworkAttachmentDefinition 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 coming with the most compatibilities.

in contrail setup the NetworkAttachmentDefinition CRD is created by **contrail-kube-manager(KM)**. when bootup, **KM** validates if a network CRD **network-attachment-definitions.k8s.cni.cncf.io** is found in the Kubernetes API server, and creates one if not yet.

here is a **CRD** object yaml:

```
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 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 `NetworkAttachmentDefinition` created from the above CRD, now we have the ability to create `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 object, basically you specify the VN name, namespace under `metadata`, and `annotations` which is used to carry additional information about a network. in contrail the following annotations are used in `NetworkAttachmentDefinition` CRD to enable certain attributes for the virtual-network:

- `opencontrail.org/cidr`: CIDR, which defines the subnet for a VN
- `opencontrail.org/ip_fabric_forwarding`: a flag to enable/disable `ip fabric forwarding` feature
- `opencontrail.org/ip_fabric_snat`: a flag to enable/disable `ip fabric snat` feature

In contrail, `ip-fabric-forwarding` feature enables ip fabric based forwarding without tunneling for the VN. When two `ip_fabric_forwrding` enabled virtual networks communicate with each other, 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.

`ip fabric forwarding` and `ip fabric snat` features are not covered in this book.

alternatively, you can define a new VN by referring an existing VN:

```

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"
}'

```

throughout this book we'll use the first template to define our VNs in all examples.

## 9.3. Multiple Interface Pod

with multiple VNs created, we can "attach" (you may also say "plug", or "insert") any of them into a pod, with a pod 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:

```

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 probably notice, pods in a namespace not only can refer to the networks defined in local NS, but also 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 NS that needs it, it can be defined just one time and then referred anywhere else.

We've understood the basic theories and explored the various templates. Now it's time to look at a "working example" in the real world. We'll start from creating two VNs, examining the VN objects, then create a pod and attach the 2 VNs into it. We'll conclude the test and this section by examining the pod interfaces and connectivity with other pods sharing the same VNs.

Here is a YAML file of two VNs: `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"
}'
```

Create both VNs:

```
$ 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 VNs:

```
$ 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": {"a  
    nnnotations": {"opencontrail.org/cidr": "10.10.10.0/24", "opencontrail.org/ip_fabric_forwa  
    rding": "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 VNs are created, as expected. it seems nothing much exciting here. However, if you login to the contrail UI, you will see something "unexpected".

Figure 97. contrail command: "main-menu" → "virtual networks"

**NOTE** make sure you select a correct "project", in this case it is **k8s-default**.

you won't be able to find any VN with the exact name `vn-left-1` or `vn-right-1` in the UI. instead, what you will find are two VNs named `k8s-vn-left-1-pod-network` and `k8s-vn-right-1-pod-network` got created.

there is nothing wrong here. What happened is whenever a VN get created from kubernetes, contrail automatically adds kubernetes cluster name(by default `k8s`) as a prefix to the VN name that you give in the network yaml file, and a suffix `-pod-network` in the end. This makes sense because we know a VN can be created by different methods. with these extra keywords embeded in the name, it is easier to tell how the VN was created(from kubernetes or from the UI manually), what will it be used for, etc. also potential VN name conflicts can be avoided across multiple kubernetes clusters.

here is yaml file of a multiple interfaces pod:

```
#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, we insert 2 VNs: `vn-left-1` and `vn-right-1`. Now guess how many interfaces will the pod has on bootup? you may think it will be two because that is what we gave 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-right-1\" } ] "
Status:        Running
IP:           10.47.255.238
...<snipped>...
```

in `Annotations`, under `k8s.v1.cni.cncf.io/network-status` we see a list `[ ... ]`, which has 3 items each represented by a curly brace block `{ }` of key-value mappings. each curly brace block includes information about one interface: the allocated IP, MAC and the VN it belongs to. so you will end up to have 3 interfaces created in the pod instead of 2. please notice the second item which gives IP address `10.47.255.238`, that 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 is no much different with the VN you create - except that you can't delete it.

we can "login to" the pod, list the interfaces and verify the IP and MAC.

```
$ 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 02:53:47:06:d8:98 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 02:53:6b:a0:e2:98 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 02:53:8e:8a:80:98 brd ff:ff:ff:ff:ff:ff
    inet 20.20.20.1/24 scope global eth2
        valid_lft forever preferred_lft forever
```

we see one lo interface and 3 interfaces plugged by contrail CNI, each with the IP allocated from the corresponding VN. also you will notice the MAC addresses match what we've seen in [kubectl describe](#) command output.

**NOTE**

having the MAC address in the annotations could be useful under certain cases. for example, in "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 which kubernetes allocated from a VN. check "service chaining" section for more details.

you will see multiple-interfaces pod again in service-chaining example later on. in that example the pod will be based on Juniper CSRX image instead of a general docker image. but the basic idea remains the same.

# Chapter 10. chapter 10: Contrail Service Chaining with CSRX

## 10.1. Contrail Service Chaining Introduction

service chaining is the idea of forwarding traffic through multiple network entity in a certain order, each network entity do specific function such as firewall, IPS , NAT , LB , ...etc. the legacy way of doing service chaining would use standalone HW appliances which made service chaining inflexible, expensive and takes a long time to setup. Dynamic service chaining is where network functions deployed as VM or Container and could be chained automatically in a logical way. in the next example we use contrail for services chaining between two PODs in two different networking using CSRX container L4-L7 firewall to secure the traffic between these two networks as shown in the figure:

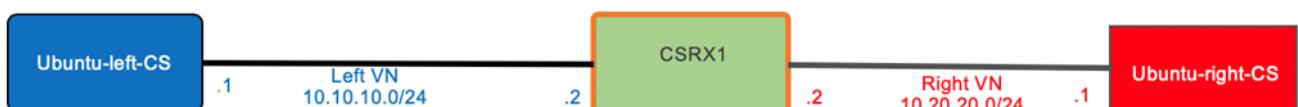


Figure 98. service chaining

**NOTE**

- left and right networks are just a common name used for simplicity and expected the traffic to follow from left to right but you can use your own names
- make sure to configure the network before you attached a POD to it otherwise POD would fail to be created

## 10.2. Bringing Up Client and CSRX Pods

### 10.2.1. Create VNs

so let's start create two networks using this YAML files

```

#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" }'

#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 a good practice to confirm these two networks are seen now in contrail before proceeding. From the Contrail UI, select Configure > Networking > Networks > default-domain > k8s-default, As shown in the figure which focus on left network

## NOTE

using **default** namespace in the YAML file for a network will create it in 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	10.10.0.0/24	-	Left-right	Disabled	Up

Allocation Mode	User Defined
Subnet(s)	CIDR      Gateway      Service Address      DNS      DHCP      Allocation Pools
	10.10.0.0/24      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 (?)
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

## 10.2.2. Create Client Pods

Create two ubuntu Pods, one in each network using the 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:fa2930cb8f4b766e5b335dfa42de510ecd30af6433c
eeda14cdcaaee8de9065d2a

...<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:fa2930cb8f4b766e5b335dfa42de510ecd30af6433c
eeda14cdcaaee8de9065d2a

...<snipped>...

```

### 10.2.3. Create CSRX Pod

create Juniper CSRX container that have one interface on the left network and one interface on the right network using this YAML file

```
#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: hub.juniper.net/security/csrx:18.1R1.9
    #image: csr
    image: contrailk8sdayone/csrx
    ports:
    - containerPort: 22
      imagePullPolicy: Never
      stdin: true
      tty: true
      securityContext:
        privileged: true
    imagePullSecrets:
    - name: secret-jnpr

# kubectl create -f csrx1-sc.yaml
```

Confirm the interface placement 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/csr
    Image ID:       docker://sha256:329e805012bdf081f4a15322f994e5e3116b31c90f108a19123cf52710c7617e
...
...<snipped>...

```

**NOTE** each container has one interface belong to “cluster-wide-default” network regardless the use of the annotations object because annotations object above creates and put one extra interface in a specific network

#### 10.2.4. Verify podIP

*verify podIP*

Login to the left, right Pods and the CSRX to confirm the IP/MAC address

```
# 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 factory default configuration hence it needs to be configured.

**NOTE**

By default, CSRX eth0 is visible only from shell and used for management. And when attaching networks, the first attached network is mapped to eth1 which is GE-0/0/1 And the second attach 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 show output as well configure 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
```

## 10.2.5. Ping Test

From the Left POD try to ping the left POD, ping 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

```

Adding static route to the left and right PODs and 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

```

Still ping failed, as we didn't create the service chaining which will also take care of the routing.  
let's see what happen to our packets

```

root@csrx1-sc# run show security flow session
Total sessions: 0

```

No session on the CSRX.

### 10.2.6. Troubleshooting Ping Issue

Login to the compute node “cent22” that host 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 bounded with the right POD and both linked to tapeth0-89a4e2 and tapeth1-89a4e2 respectively same goes for the left POD for Vif0/5 and vif0/6 while vif0/7, vif 0/8 and vif0/9 are bound with CSRX1. | from that you can also see the number of packets/bytes hits that interface as well the VRF which is this interface belong in here VRF 3 is for the default-cluster-network while VRF 6 for the left network and VRF 5 for the right network in this figure you can see the interface mapping from the all prospective (container, Linux , vr-agent)



try to ping again from the left POD to the right POD 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 , lets see on the CSR1X 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  
 checking the routing table of the left network VRF by logging to the `vrouter_vrouter-agent_1` docker 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 /]$
```

Note that 6 is the routing table VRF of the left network, same would goes for the right network VRF routing table there is 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 different compute nodes then you have a bigger problem to solve. Service chaining isn't about adjusting the routes on the containers but mainly about exchange routes

between the vrouter-agent between the compute nodes regardless of the location of the POD, as well adjust that automatically if the POD moved to another compute node. Before we build service chaining lets address an important concerns for network administrator who are not fan of this kind of CLI troubleshooting, can we do the same troubleshooting using contrail controller GUI?

the answer is yes and lets do it.

From the Contrail Controller UI, select monitor > infrastructure > virtual router then select the node the that host the POD , in our case “Cent22.local”

Name	Label	Status	Type	Network	IP Address	Floating IP	Instance
ens160	-1	Up	eth		IPv4: 10.85.188.17	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-8d82-11e9-b451-0050569e2171 /
tapeth1-89a4e2	32	Up	vport	k8s-vn-right-pod-network (k8s-default)	IPv4: 10.20.20.1	None	89a4e2cb-8d82-11e9-b451-0050569e2171 /
pkt0	-1	Up	pkt			None	

as shown in the figure from the interface tab which is equivalent to running “vif -l” command on the vrouter\_vrouter-agent-1 container and even showing more information notice the mapping between the instance ID and tap interface naming where the first 6 character of the instance ID are always reflected in the tap interface naming

to check the routing tables of each VRF move to the “routes” tab and select the VRF you want to see

VRF	Route Details
default-domain:default-project:ip-fabric..._default	Prefix: 10.85.188.0/27 (1 Route) Source: LocalVmPort Destination VN: default-domain:default-project:ip-fabric... Policy: disabled Peer: LocalVmPort Valid: true Prefix: 10.85.188.1/32 (1 Route)
default-domain:default-project:ip-fabric..._fabric	
default-domain:k8s-default:k8s-default-pod-network	
default-domain:k8s-default:k8s-vn-left-pod-network	
default-domain:k8s-default:k8s-vn-right-pod-network	

If we select the left network ( the name is longer as it include the domain , project ) we can confirm there is not 10.20.20.0/24 prefix from the right network We can also check the mac address learned in the left network by selecting L2 ( which is equivilant to “rt --dump 6 --family bridge” command

Monitor	Details	Interfaces	Networks	ACL	Flows	Routes	Instances	Console	Alarms
Infrastructure									
- Dashboard									
- Physical Topology									
Control Nodes									
- Virtual Routers									
- Analytics Nodes									
- Config Nodes									
Database Nodes									
Security									
Networking									
Debug									
Alarms									

VRF: default-domain:k8s-default:k8s-vn-left-pod-network:k8s-vn-left-pod-network

Show Routes L2 Unicast 6

Routes (1 - 7 of 7)

Next hop Type Next hop details

- i2-receive MAC: 00:00:5e:00:01:00 (1 Route)
  - Source IP: Destination IP: vrf: Policy: disabled Peer: Local\_Vm Valid: true Label: 0
  - MAC: 00:50:56:9e:bb:98 (1 Route)
- i2-receive MAC: 02:00:00:00:00:01 (1 Route)
  - Source IP: Destination IP: vrf: Policy: disabled Peer: Local\_Vm Valid: true Label: 0
  - MAC: 02:00:00:00:00:01 (1 Route)
- interface MAC: 02:00:00:00:00:02 (1 Route)
  - Interface: pkt0 Valid: true Peer: Local\_Vm Policy: disabled
  - MAC: 02:7d:b1:09:00:8d (1 Route)
- interface MAC: 02:84:71:f4:f2:8d (1 Route)
  - Interface: pkt0 Valid: true Peer: Local\_Vm Policy: disabled
  - MAC: 02:84:71:f4:f2:8d (1 Route)
- discard Source: MacVmBindingPeer MAC: ffff:ffff:ffff:ff (4 Routes)
  - Source IP: Destination IP: vrf: Ref count: 1 Policy: disabled Peer: Multicast Valid: true Label: 4611 Multicast Data / type: interface label: 0 if: tapeth1-7d8e06 / type: interface label: 0 if: tapeth1-844f1c
  - Source IP: Destination IP: vrf: Ref count: 1 Policy: disabled Peer: Local\_Vm Valid: true Label: 38
  - Source IP: Destination IP: vrf: Ref count: 1 Policy: disabled Peer: Local\_Vm Valid: true Label: 38 Multicast Data:
  - Source IP: Destination IP: vrf: Policy: disabled Peer: MulticastTreeBuilder Valid: true Label: 4611
- L2 Composite sub nh count: 2
- L2 Interface Composite sub nh count: 2
- L2 Composite sub nh count: 0
- fabric Composite sub nh count: 0

## 10.3. Service Chaining

### 10.3.1. Create Service Chaining

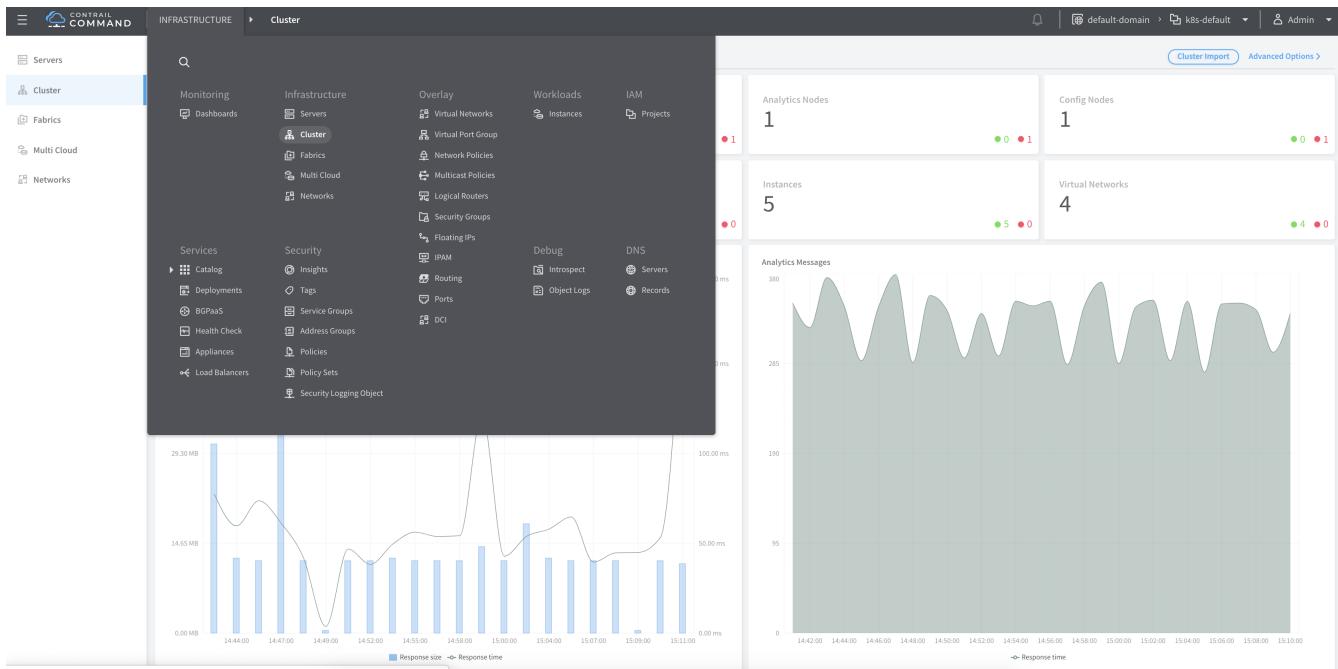
Now lets utilize the CSRX to service chaining using contrail command GUI

creating Service chaining is 4 steps make sure to do them in this order

1. create Service template
2. creating service instance based on the service template you created before
3. creating network policy and select the service instance you created before
4. apply this network policy on network

**NOTE** since contrail command GUI is the solution to provide a single point of management for all environments, we will use it to build service changing but you still can use the normal contrail controller GUI to build service changing

Login to contrail command GUI ( in our setup <https://10.85.188.16:9091/>) then select service > catalog > create



The screenshot shows the Contrail Command interface with the following details:

- Top navigation bar: CONTRAIL COMMAND, SERVICES, Catalog, Create VNF Service Template.
- Left sidebar: Catalog, Deployments, BGPaaS, Health Check, Appliances, Load Balancers.
- Main content area:
  - Service Template tab selected.
  - Name: myweb-CSRX-CS
  - Version: v2
  - Virtualization Type: Virtual Machine
  - Service Mode: In-Network
  - Service Type: Firewall
  - Interface section: Interface Type dropdown set to "Select Interface(s)".
- Top right: default domain, k8s default, Admin.

Insert a name of services template “myweb-CSRX-CS” in here then chose v2 , virtual machine ( no other option available) for service mode we will work with In-network and firewall as service type

The screenshot shows the same interface as above, but with the following changes:

- Name: myweb-CSRX-CS
- Version: v2
- Virtualization Type: Virtual Machine
- Service Mode: In-Network
- Service Type: Firewall
- Interface section: Interface Type dropdown now includes "management", "left", and "right".

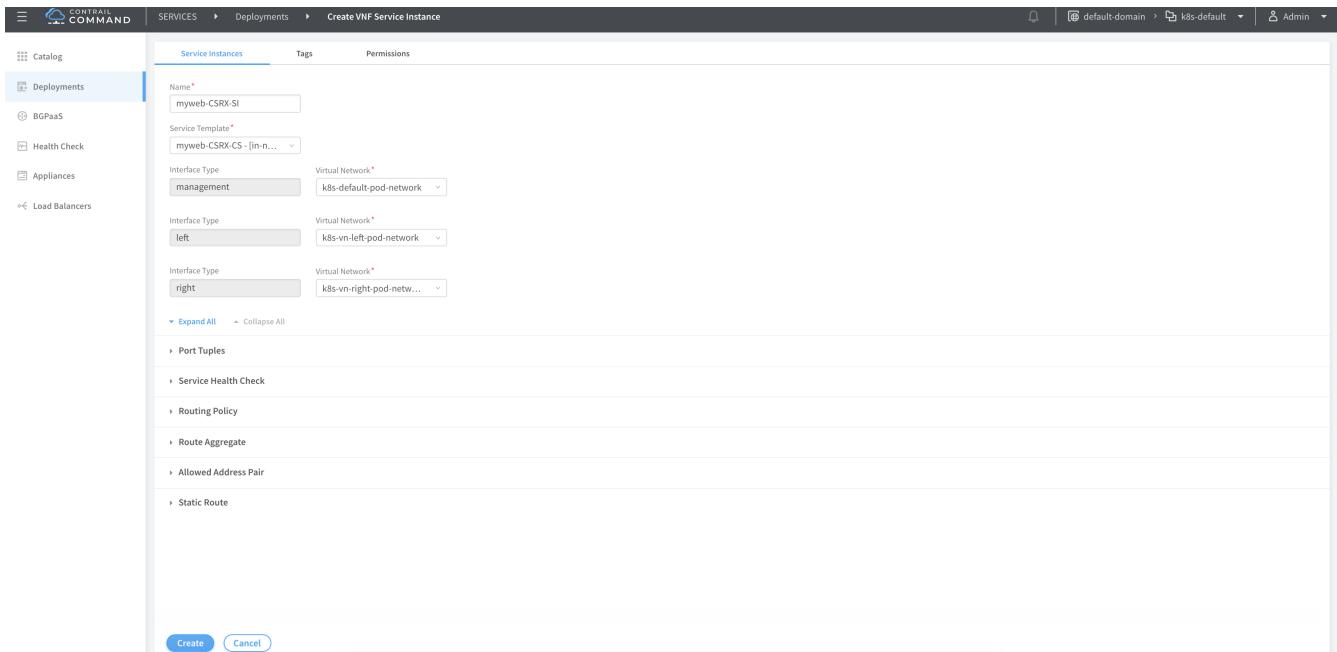
Select interfaces management, left and right then click create

The screenshot shows the 'Interfaces' configuration dialog with the following fields:

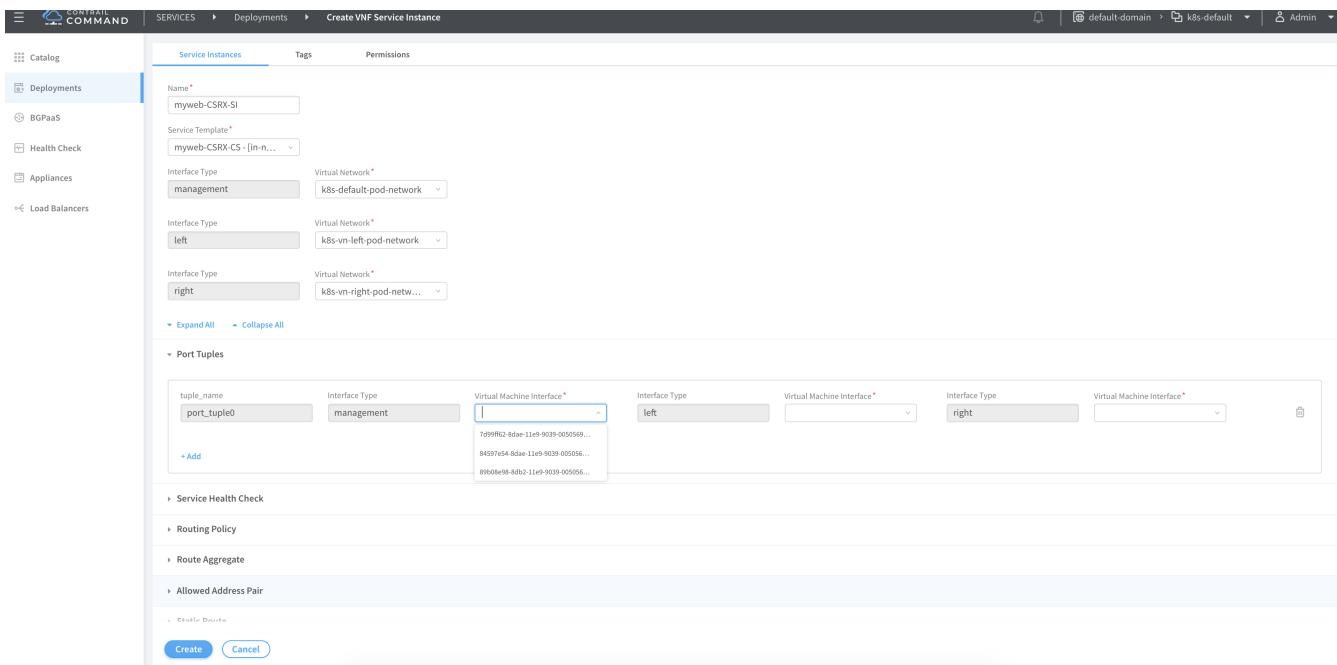
Interface	Interface Type
management	management
left	left
right	right

At the bottom are 'Create' and 'Cancel' buttons.

Now select deployment and click create to create the service instances



Insert a name for this service instance then select from the drop down menu the name of the template you created before then chose the proper network from the prospective of the CSRX being the instance (container in that case) that will do the service chaining and click on port tuples to expand it



then for each of the three interface bound one interface of the CSRX then click create

#### NOTE

the name of the virtual machine interface isn't shown in the drop down menu instead the instance ID, you can identify that from the tap interface name as we showed before. In other word all you have to know is most 6 left character for any interface belong to that container as all the interface in a given instance ( VM or container) share the same first characters from the left

Before you procced make sure the status of the three interfaces are up and they are showing the correct IP address of the CSRX instance

VNF Service Instances

STATUS	SERVICE INSTANCE	SERVICE TEMPLATE	FORWARDERS	NETWORKS
<input checked="" type="checkbox"/>	myweb-CSRX-SI	myweb-CSRX-CS (in-network, version 2)		Management:k8s-default:pod-network, Left:k8s-vn-left:pod-network, Right:k8s-vn-right:...

**Details**

Instance	myweb-CSRX-SI	Owner	None
Name	myweb-CSRX-SI	Owner	Read, Write, Refer
Display Name	myweb-CSRX-SI	permits...	
UUID	dfc8fb02-cc3d-4356-978e-0d8116b89947	Global	-
Template	myweb-CSRX-CS (in-network, version 2)	permits...	
Ports	myweb-CSRX-SIport, tuple0:b6401d76-334f-4ab2-8dd6-0ee530749c2a;	Share	-
Tuples			
Availability	-		
Zone	-		
Status	Spawning		

**Statuses**

INSTANCE STATUS	INTERFACE	STATUS	HEALTH STATUS	IP ADDRESS
No Server Found				
default-domain:k8s-default:csrx1-sc_848bd4c18-8dae-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_847114f2-8dae-11e9-9039-0050569e2171		Active	-	10.10.10.2

To create network policy go to overlay > network policies > create

OVERLAY > Network Policies

Monitoring	Infrastructure	Overlay	Workloads	IAM
<input checked="" type="checkbox"/> Dashboards	<input type="checkbox"/> Servers	<input type="checkbox"/> Virtual Networks	<input type="checkbox"/> Instances	<input type="checkbox"/> Projects
<input type="checkbox"/> Cluster	<input type="checkbox"/> Virtual Port Group	<input type="checkbox"/> Fabric	<input type="checkbox"/> Network Policies	
<input type="checkbox"/> Multi Cloud	<input type="checkbox"/> Multicast Policies	<input type="checkbox"/> Logical Routers	<input type="checkbox"/> Security Groups	
<input type="checkbox"/> Networks	<input type="checkbox"/> Floating IPs	<input type="checkbox"/> IPAM	<input type="checkbox"/> Debug	<input type="checkbox"/> DNS
<input type="checkbox"/> Services	<input type="checkbox"/> Security	<input type="checkbox"/> Routing	<input type="checkbox"/> Introspect	<input type="checkbox"/> Servers
<input type="checkbox"/> Catalog	<input type="checkbox"/> Insights	<input type="checkbox"/> Tags	<input type="checkbox"/> Ports	<input type="checkbox"/> Object Logs
<input type="checkbox"/> Deployments	<input type="checkbox"/> Service Groups	<input type="checkbox"/> Address Groups	<input type="checkbox"/> DCI	<input type="checkbox"/> Records
<input type="checkbox"/> BGPaaS	<input type="checkbox"/> Policies	<input type="checkbox"/> Policy Sets		
<input type="checkbox"/> Health Check	<input type="checkbox"/> Policy Logging Object			
<input type="checkbox"/> Appliances				
<input type="checkbox"/> Load Balancers				

Insert a name for your network policy then in the first rule add left network as source network and right network as destination with action pass

OVERLAY > Network Policies > Create Network Policy

Network Policy

Action	Protocol	Source Type	Source	Source Port	Direction	Destination Type	Destination	Destination Ports
pass	ANY	Network	k8s-vn-right:pod-network	Any	<>	None	Enter valid CIDR	Any

Policy Rule(s)

Action	Protocol	Source Type	Source	Source Port	Direction	Destination Type	Destination	Destination Ports
pass	ANY	Network	k8s-vn-right:pod-network	Any	<>	None	Enter valid CIDR	Any

**Tags**

**Permissions**

Select advanced option to attached the service instance you create before and click create

Network Policy

Policy name\*: left-right networks

Policy Rule(s)

Action: pass	Protocol: ANY	Source Type: Network	Source: k8s-vn-left-pod-network	Source Port: Any	Direction: <>	Destination Type: Network	Destination: k8s-vn-right-pod-netw...	Destination Ports: Any	<input checked="" type="checkbox"/> Advanced Options
Services: default-domain:k8s... x	QoS: Select QoS	Log							
Mirror:	<input type="radio"/> None	<input type="radio"/> Analyzer Instance	<input type="radio"/> NIC assisted	<input type="radio"/> Analyzer IP					

+ Add

Create Cancel

To attach this network policy to network click virtual network and select the left network and edit

OVERLAY > Virtual Networks

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	<span style="color: #0070C0;">Edit</span> ...

Details

Display Name: k8s-vn-right-pod-network	Permissions
UUID: 4bb08815-af4f-42d5-86e5-2f359de20405	Owner: None
VPGs: No VPG Found	Owner permissions: Read, Write, Refer
	Global permissions: -
	Share: -

In network policies select the network policy you just created from the drop down menu then click save do the same for the right network

## 10.3.2. Verify Service Chaining

Now lets check 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 the that host the POD , in our case “Cent22.local” then select the “routes” tab and select the left VRF

Now we can see the right network's 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 try to 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

```

### 10.3.3. Security Policy

Now let try to create 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 failed as the policy on the CSRX drop 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-https 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-https 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

```

create some files with name being same as the paths: **dev**, **qa**, and **abc**.

```
kubectl exec -it Vwebserver-1-846c9ccb8b-s2zn9 -- bash -c "echo Vwebserver-1-846c9ccb8b-s2zn9:10.47.255.249 > dev"
kubectl exec -it Vwebserver-1-846c9ccb8b-s2zn9 -- bash -c "echo Vwebserver-1-846c9ccb8b-s2zn9:10.47.255.249 > qa"
kubectl exec -it Vwebserver-1-846c9ccb8b-s2zn9 -- bash -c "echo Vwebserver-1-846c9ccb8b-s2zn9:10.47.255.249 > abc"
kubectl exec -it Vwebserver-2-846c9ccb8b-k9x26 -- bash -c "echo Vwebserver-2-846c9ccb8b-k9x26:10.47.255.248 > dev"
kubectl exec -it Vwebserver-2-846c9ccb8b-k9x26 -- bash -c "echo Vwebserver-2-846c9ccb8b-k9x26:10.47.255.248 > qa"
kubectl exec -it Vwebserver-2-846c9ccb8b-k9x26 -- bash -c "echo Vwebserver-2-846c9ccb8b-k9x26:10.47.255.248 > abc"
```

# Chapter 11. appendix

## 11.1. contrail kubernetes setup installation

**NOTE** the HW/SW requirements that is listed here apply to the testbed we use to test the theories and examples in our book only. please refer to Juniper website for official requirements if you want to build a more scalable setup.

### 11.1.1. HW/SW prerequisites

- centos 7.6
- 32G mem
- 50G HD

```
$ 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%  /
```

### 11.1.2. 3 nodes cluster only setup

#### topology

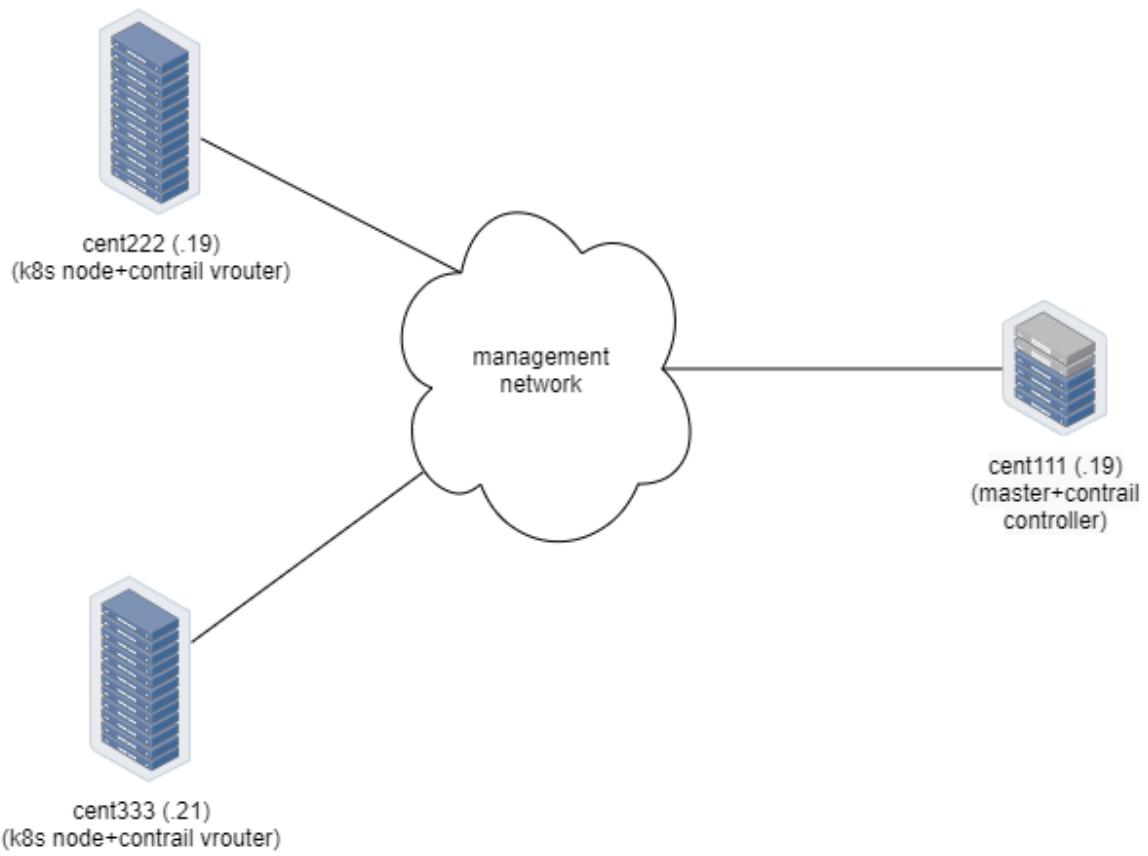


Figure 99. 3 nodes cluster-only setup (no external connections)

#### 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

```

only the following parameters needs to be changed:

```

ssh_pwd: Juniper
ssh_user: root
ntpserver: ntp.juniper.net
ip: 10.85.111.26 ~ 28

```

### 11.1.3. 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
```

### 11.1.4. verification

Pod Status	Service	Original Name	State	Id
	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
```