Chapter 4

Kubernetes and Contrail integration

2019-06-30

This chapter demonstrates the how kubernetes objects works in contrail setup.

# Contrail-Kubernetes architecture

## 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 a 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 ) and 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 organization 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 as 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 lunch its POD in VMs orchestrated by OpenStack and this 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

Then what we mean by contrail enriching standard Kubernetes deployment? 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- service chaining

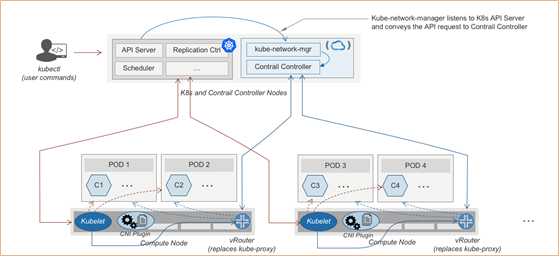
3-distrubted LB and firewall with extensive centralized flow and logs insight

4- rich security policy using tags that can extend to other environment (OpenStack, VMWare, BMS, AWS ,..,etc)

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

## contrail-kube-manager

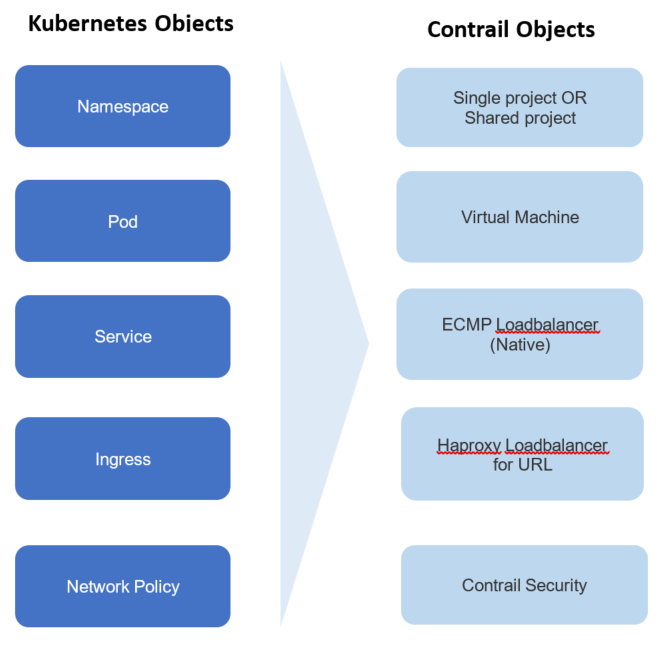
A new components of contrail has been add called contrail-Kube-manager which would listen to API request sent from the Kubernetes API server and translate that to Contrail controller and on the compute node the contrail Vrouter would replace Kube-proxy as shown in the diagram



## 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

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



# multiple interface pod in 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. in contrail solution service chain model, a service instance typically needs a "left", "right" and optionally a "management" interface to manipulate the service traffic. 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.

## contrail as a CNI

in container technology, A virtual network device pair abstraction (the veth) 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. it can also be used to create a bridge to a physical network device in another namespace.

A "CNI plugin" is the one who is responsible for inserting the network interface, as one end of the veth pair, into the container network namespace. 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.

Kubernetes supports a custom extension to represent networks in its object model, through its CustomResourceDefinition(CRD) feature. This extension adds support for a new kind of object called NetworkAttachmentDefinition, which represents a network in Kubernetes data model.

contrail is one of such "CNI plugin" implementations. there are many publicly available CNI plugin implementation today. for a comprehensive list you can check <https://github.com/containernetworking/cni>, where contrail is mentioned. for example, multus-cni, is another CNI plugin that "enables attaching multiple network interfaces to pods". however, its multipe-network support is accomplished by Multus calling multiple other CNI plugins that, each plugin will create its own network so overall a pod can have multiple networks. one of the main advantages that contrail provides, comparing with Mutus and all other implementations, is that contrail by itself provides the ability to attach multiple network interfaces to a kubernetes pod, making it a truly "multi-homed" pod.

contrail CNI follows the Kubernetes Network CRD (Custom Resource Definition) Standard to provide a standardized method to specify the configurations for additional network interfaces. there is no change to the standard kubernetes APIs, making the implementation the most compatible.

## CRD and contrail-kube-manager

a CRD object defines the template for a network object NetworkAttachmentDefinition, which contains all information about each network’s specification and tells Kubernetes API how to understand and expose it. in contrail setup the CRD is created by a component named contrail-Kube-Manager, abbreviated as KM, running as a docker container. KM interfaces with Kubernetes API server and converts objects from kube-apiserver to Contrail config API server. when bootup, KM will validate if network CRD network-attachment-definitions.k8s.cni.cncf.io is found in the Kubernetes API server and creates one if not yet.

here is how a CRD object template looks like:

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 displayed:

$ kubectl get crd  
NAME CREATED AT  
network-attachment-definitions.k8s.cni.cncf.io 2019-06-07T03:43:52Z

with CRD object present, we have the ability to create a NetworkAttachmentDefinition object as our virtual-network.

## NetworkAttachmentDefinition definition

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"  
}'

through NetworkAttachmentDefinition object which is created by CRD, we can define new VNs. like many other standard kubernetes object, basically you specify the VN name, namespace under metadata, and here you see contrail uses kubernetes annotations to specify the extended attributes of a network.

* opencontrail.org/cidr gives CIDR, which gives the subnet for a VN
* opencontrail.org/ip\_fabric\_forwarding is a flag to enable/disable 'ip fabric forwarding' feature
* opencontrail.org/ip\_fabric\_snat is a flag to enable/disable ip fabric snat feature

with the contrail ip-fabric-forwarding feature, A VN can be marked for IP fabric based forwarding without tunneling. When two virtual networks with this type of configuration communicate, 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 and is applicable only to pod networks.

both ip fabric forwarding and ip fabric snap features are out of scope of 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"  
}'

throught this book we’ll use the first template to define our VNs in all examples.

## multiple-interface pod definition

with multiple VNs created, we can now "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" },  
 { "name": "other-ns/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 have noticed, pods in a namespace can not only refer to the networks defined in local NS, but also can refer networks created on other namespaces using their fully scoped name. this is very useful - the same network does not has to be duplicated in every NS that needs it, it can be defined only once and then referred anywhere else.

## lab demo: multi-interface pod

after exploring many theories and templates, you understand how things works basically. now it’s the 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

$ cat 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"  
 }'

$ cat 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:

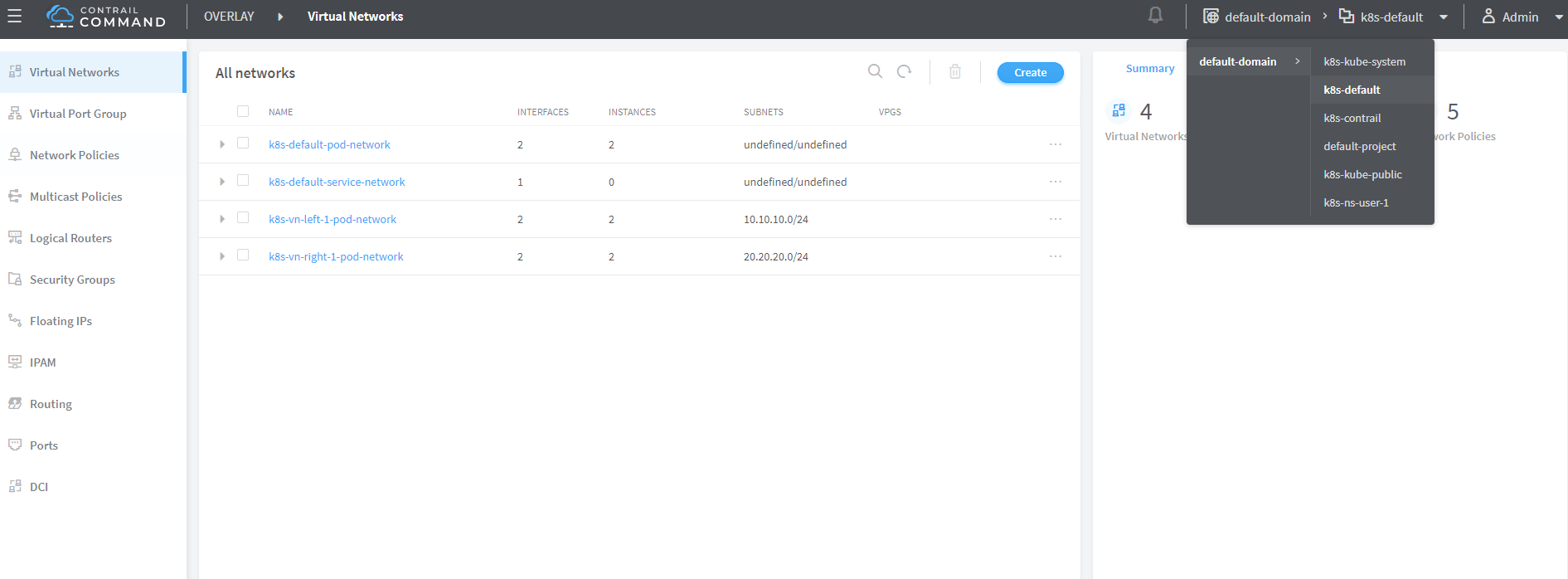
$ kubectl apply -f vn-left-1.yaml  
networkattachmentdefinition.k8s.cni.cncf.io/vn-left-1 created  
  
$ kubectl apply -f vn-right-1.yaml  
networkattachmentdefinition.k8s.cni.cncf.io/vn-right-1 created

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":{"annotations":{"opencontrail.org/cidr":"10.10.10.0/24","opencontrail.org/ip\_fabric\_forwarding":"false"},"name":"vn-left-1","namespace":"ns-user-1"},"spec":{"config":"{ \"cniVersion\": \"0.3.0\", \"type\": \"contrail-k8s-cni\" }"}}  
 opencontrail.org/cidr: 10.10.10.0/24  
 opencontrail.org/ip\_fabric\_forwarding: "false"  
 creationTimestamp: 2019-06-13T14:17:42Z  
 generation: 1  
 name: vn-left-1  
 namespace: ns-user-1  
 resourceVersion: "777874"  
 selfLink: /apis/k8s.cni.cncf.io/v1/namespaces/ns-user-1/network-attachment-definitions/vn-left-1  
 uid: 01f167ad-8de6-11e9-bbbf-0050569e6cfc  
spec:  
 config: '{ "cniVersion": "0.3.0", "type": "contrail-k8s-cni" }'

the VNs are created, as expected. it seems nothing much exciting here. However, if you login to the contrail GUI, you will see something "unexpected".



contrail command: "mega-menu" → "virtual networks"

make sure you select a correct "project", in this case it is k8s-default.

you won’t see any VN with the exact name vn-left-1 or vn-right-1 existing. 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 will automatically add a prefix k8s- to the VN name that you give in the network yaml file, and a suffix -pod-network in the end. This makes sense because now you know a VN can be created by different methods. with a these extra keyword embeded it is easier to tell how the VN was created (from kubernetes or from the GUI manually), what will it be used for, and also potential VN name confliction is avoided.

here is yaml file of a cirros pod.

apiVersion: v1  
kind: Pod  
metadata:  
 name: cirros  
 labels:  
 app: cirros  
 annotations:  
 k8s.v1.cni.cncf.io/networks: '[  
 { "name": "vn-left-1" },  
 { "name": "vn-right-1" }  
 ]'  
spec:  
 containers:  
 - name: cirros  
 image: cirros  
 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  
cirros 1/1 Running 0 20s 10.47.255.238 cent222 <none>  
  
$ kubectl describe pod cirros  
Name: cirros  
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=cirros  
Annotations: k8s.v1.cni.cncf.io/network-status:  
 [  
 {  
 "ips": "10.10.10.250",  
 "mac": "02:87:cf:6c:9a:98",  
 "name": "vn-left-1"  
 },  
 {  
 "ips": "10.47.255.238",  
 "mac": "02:87:98:cc:4e:98",  
 "name": "cluster-wide-default"  
 },  
 {  
 "ips": "20.20.20.1",  
 "mac": "02:87:f9:f9:88:98",  
 "name": "vn-right-1"  
 }  
 ]  
 k8s.v1.cni.cncf.io/networks: [ { "name": "vn-left-1" }, { "name": "vn-right-1" } ]  
 kubectl.kubernetes.io/last-applied-configuration:  
 {"apiVersion":"v1","kind":"Pod","metadata":{"annotations":{"k8s.v1.cni.cncf.io/networks":"[  
 { \"name\": \"vn-left-1\" }, { \"name\": \"vn-...  
Status: Running  
IP: 10.47.255.238  
...<snipped>...

in Annotations, under k8s.v1.cni.cncf.io/network-status we see a list […​], which has 3 items each represented by a block {} of key-value mappings that displays the interface 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. notice the 2nd item which gives IP address 10.47.255.238, that is the interface attached to the default pod network which is created by contrail. you can treat the default pod network as a "managment" network because it is always up and present in every pod’s network namespace, but funtionally 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 cirros 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 1 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 output.

you will see multiple-interface pod again in sevice-chaining example later on. in that example the pod will be based on Juniper CSRX instead of a general docker image. but the basic idea remains the same.

**Note**

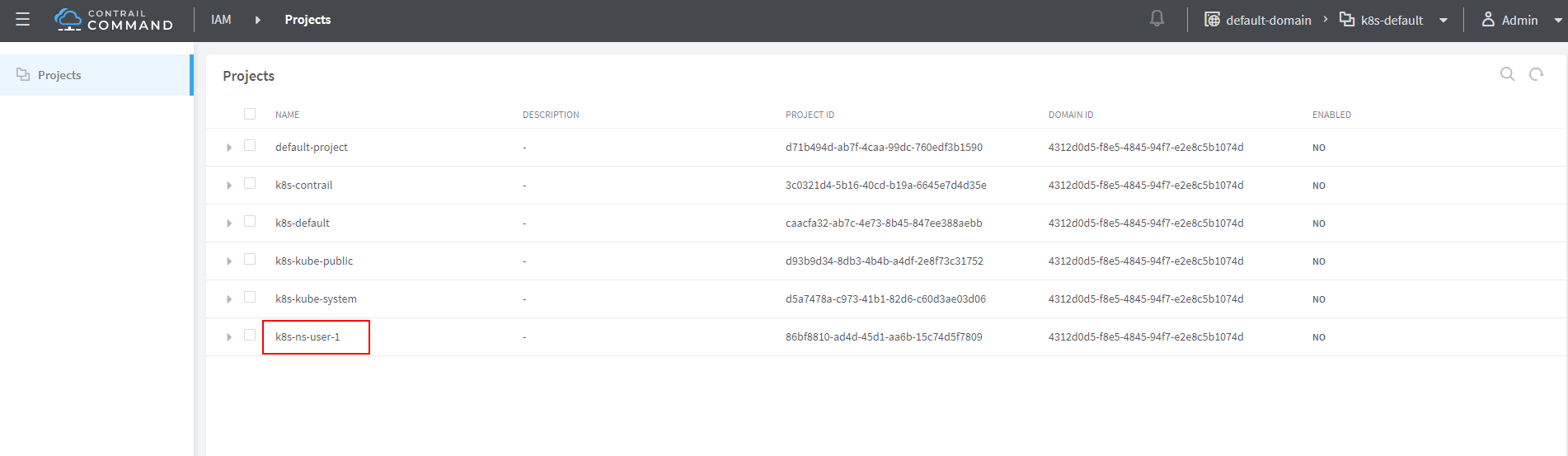
the MAC address will be important under certain cases - if you login a pod and for some reason you lose the track of interface to VN mapping (e.g., you manually changed/removed the IPs, or the pod’s application reset the IP, etc) you can count on the MAC address! later In "service chaining" section you will run into a scenario when you need to use the MAC address to locate the interface IP allocated from a VN.

# namespaces and isolation in contrail

## NS vs project

In chapter3 you`ve read about namespace', or NS in kubernetes, and how to use a  
`quota to apply some constraints to the resource utilization by a NS. 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 will get noticed about the object creation and it will create the corresponding data in contrail api database. to differiciate a kubernetes NS project vs a "normal" contrail/openstack project, a prefix k8s- will be added to the k8s ns project name. so if you create a kubernetes NS ns-user-1, what you will see in contrail GUI will be: k8s-ns-user-1:



**Note**

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

## NS isolation

Just like in openstack you can have shared tenant and private tenant, similiar concept exists for k8s namespace in contrail. By default a k8s generated NS will be a "non-isolated namespace", which means all namespaces share the same routing instance or VRF (virtual routing and forwarding table), so that the bidirectional communication will happen by default between all pods in all shared namespaces, including the default namespace. in contrast, an "isolated namespace", will have its dedicated VRF, so by default only pods launched in this namespace can talk to each other directly. Additional configuration, e.g. policy, is required to make the pod being able to reach the network outside of current namespace.

here is the yaml file to create an isolated namespace:

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

to create the NS:

kubectl create -f ns-isolated.yaml  
  
$ kubectl get ns  
NAME STATUS AGE  
contrail Active 8d  
default Active 8d  
development Active 2d  
ns-isolated Active 1d #<---  
kube-public Active 8d  
kube-system Active 8d

the annotations under metadata are additional comparing to standard (non-isolated) k8s namespace, indicating this is a isolated NS:

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

this part of the definition is Juniper’s extension. contrail-kube-manager 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, the correponding routing instance, and other necessary data in contrail config API database for the isolated namespace. one of the major difference between a non-isolated namespace vs an isolated namespace is that, contrail always create a seperate routing instance for the tenant triggered by the isolated namespace. fundamentally that is how the "isolation" is implemented.

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

## communication between pod in different namespaces

**create a non-isolated namespace and an isolated namespace.**

$ cat ns-non-isolated.yaml  
apiVersion: v1  
kind: Namespace  
metadata:  
 name: ns-non-isolated  
  
$ cat ns-isolated.yaml  
apiVersion: v1  
kind: Namespace  
metadata:  
 annotations:  
 "opencontrail.org/isolation": "true"  
 name: ns-isolated  
  
$ kubectl apply -f ns-non-isolated.yaml  
namespace/ns-non-isolated created  
  
$ kubectl apply -f ns-isolated.yaml  
namespace/ns-isolated created  
  
$ kubectl get ns | grep isolate  
ns-isolated Active 79s  
ns-non-isolated Active 73s

**in both NS and the default NS, create a deployment to launch a pod.**

$ kubectl apply -f deployment-cirros.yaml -n default  
deployment.extensions/cirros created  
  
$ kubectl apply -f deployment-cirros.yaml -n ns-non-isolated  
deployment.extensions/cirros created  
  
$ kubectl apply -f deployment-cirros.yaml -n ns-isolated  
deployment.extensions/cirros created  
  
$ kubectl get pod -o wide -n default  
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE  
cirros-85fc7dd848-tjfn6 1/1 Running 0 13s 10.47.255.242 cent333 <none>  
  
$ kubectl get pod -o wide -n ns-non-isolated  
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE  
cirros-85fc7dd848-nrxq6 1/1 Running 0 23s 10.47.255.248 cent222 <none>  
  
$ kubectl get pod -o wide -n ns-isolated  
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE  
cirros-85fc7dd848-6l7j2 1/1 Running 0 8s 10.47.255.239 cent222 <none>

**ping between all pods in 3 namespaces.**

$ kubectl -n default exec -it cirros1-85fc7dd848-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  
  
$ kubectl -n default exec -it cirros1-85fc7dd848-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 -n ns-isolated exec -it cirros-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 packet passes through. the isolation is between isolated NS and all other tenant in the cluster, but not between pods in same NS!

**Note**

pod-level isolation can be archived via kubernetes network policy, or security groups in contrail. this is not covered in this book.

# floating IP

## introduction

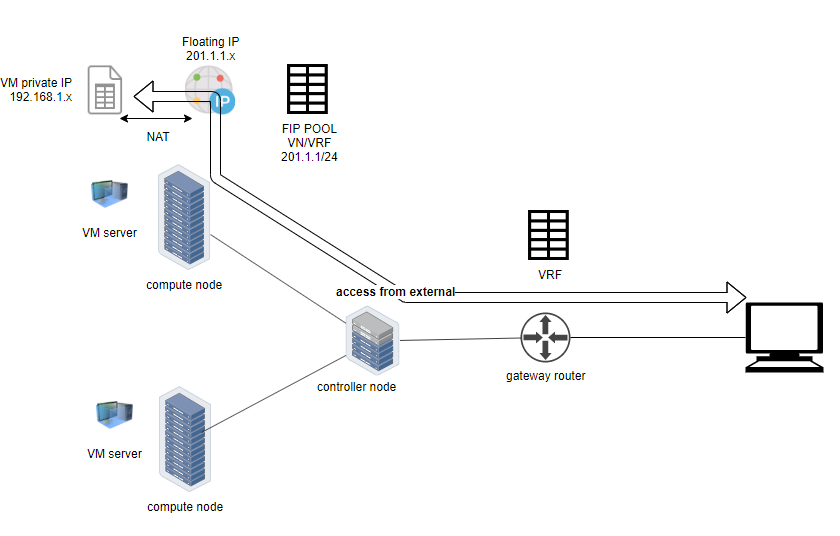
floating IP, or FIP for short, is a "traditional" concept that contrail supports since very early releases. Essentially it is an openstack utility to "map" a VM IP, which is typically a private IP address, to a public IP (the "floating IP" in this context) that is reachable from the outside of the cluster. 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.

Regarding this concept and its role, there is nothing new in contrail kubernetes environment. the usage of floating IP has been extended in service and ingress implementation, and it plays an important role for the end to end access toward kubernetes service, with or without ingress. you will learn more details about how kubernetes service and ingress works in contrail setup in later section.

there are a few basic facts about FIP and FIP pool configuration:

* a FIP is allocated from a FIP pool
* a FIP is associated with a VM’s port, or a VMI (Virtual Machine Interface).
* a FIP pool is created based on a virtual network(FIP-VN)
* the FIP or FIP-VN prefix can be advertised to the outside of the cluster, typically through a gateway router

the diagram below illustrated the basic work flow of FIP:



## creating FIP pool

creating a FIP pool is a 2 step process:

* create a public 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 VN

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 VN can now be created in a "kubernetes style":

**create a public VN named vn-ns-default.**

apiVersion: k8s.cni.cncf.io/v1  
kind: NetworkAttachmentDefinition  
metadata:  
 annotations:  
 "opencontrail.org/cidr": "101.101.101.0/24"  
 "opencontrail.org/ip\_fabric\_forwarding": "false"  
 "opencontrail.org/ip\_fabric\_snat": "false"  
 name: vn-ns-default  
spec:  
 config: '{  
 "cniVersion": "0.3.0",  
 "type": "contrail-k8s-cni"  
 }'

**set the RT.**

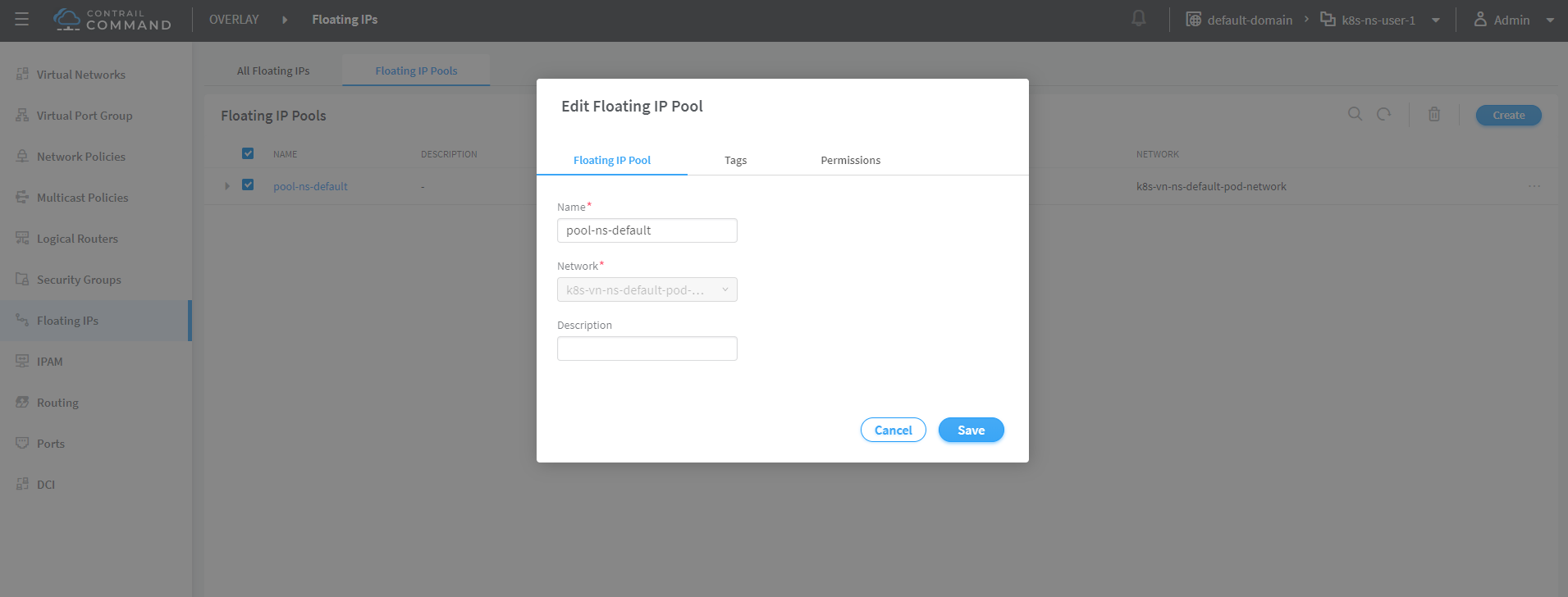
if you need the FIP to be reachable from Internet through gateway router, you’ll need to set a route-target to make the VN prefix imported in the gateway router’s VRF table. this step is required whenever Internet access is required.

**Tip**

that is why in the later lab demo of service or ingress you will also need to set the RT.

**create a FIP pool based on the public VN.**

from contrail command UI, select mega-menu > Overlay > Floating IP, then Create:



**Tip**

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

## FIP pool scope

there are different ways you can define an IP pool, and correspondingly the scope of the pool will also be different. here are 3 possible scopes an IP pool can has:

* global default
* NS default
* object specific

**object specific.**

this is the most specific level of scope. object sepecific FIP pool binds itself only to the object that you are trying to create, and it does not affact any other objects in the same NS or the whole cluster. E.g. you can specify a service web to get FIP from FIP pool pool1, a service dns to get FIP from another FIP pool pool2, etc. This give the most granular control of how you allocate FIP for an object, the cost is you need to explicitly specify it in configuration file of every object (typically a yaml file for kubernetes objects).

**NS default.**

in a NS, a "lazy" way to give FIP is to define a "NS level" FIP pool, so that all objects created in that NS will "by default" get FIP assignment from that pool. with NS default pool defined (e.g. pool-ns-default), there is no need to specify the same 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 poo-ns-default, because it is more specific.

**global default.**

a "even lazier" method is to define a "global" level default pool, which means the scope will be the whole cluster, including all namespaces. 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 FIP from this pool
* for NS dev, define a FIP pool pool-dev, so all objects in dev will by default get FIP from poo-dev
* for NS sales, define a FIP pool pool-sales, so all objects in sales will by default get FIP from poo-dev
* for NS test-only, do NOT define any NS level pool, so by default all objects in it will get FIP from the pool-global-default
* when a service dev-websevice in dev needs a FIP from pool-sales, specify pool-sales in dev-webservice object will archive this goal.

**Tip**

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

## global level default FIP pool

to define the global level default FIP pool, you need to specify the full qualified pool name (domain > project > network) in configuration file of contrail-kube-manager docker container.

$ docker ps -a | grep kubemanager  
2260c7845964 9bd730ca1d5f "/entrypoint.sh /usr…" 3 days ago Up 2 minutes kubemanager\_kubemanager\_1  
869bf2080530 ci-repo.englab.juniper.net:5000/contrail-node-init:master-latest "/entrypoint.sh" ...(snipped)...

the configuration file is /etc/contrail/common\_kubemanager.env:

$ cat /etc/contrail/common\_kubemanager.env  
VROUTER\_GATEWAY=10.169.25.1  
CONTROLLER\_NODES=10.85.188.19  
KUBERNETES\_API\_NODES=10.85.188.19  
RABBITMQ\_NODE\_PORT=5673  
CLOUD\_ORCHESTRATOR=kubernetes  
KUBEMANAGER\_NODES=10.85.188.19  
CONTRAIL\_VERSION=master-latest  
KUBERNETES\_API\_SERVER=10.85.188.19  
TTY=True  
ANALYTICS\_SNMP\_ENABLE=True  
STDIN\_OPEN=True  
ANALYTICS\_ALARM\_ENABLE=True  
ANALYTICSDB\_ENABLE=True  
CONTROL\_NODES=10.169.25.19

as you can see, this .env file contains important environmental parameters about the setup. to specify a global FIP pool, add 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 default FIP pool is called pool-global-default, and it is defined based on a VN vn-global-default under project k8s-ns-user-1. the project name indicates the corresponding kubernetes namespace - ns-user-1.

now with that piece of configuration placed added, 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 default FIP pool is defined.

## NS level default FIP pool

the next FIP pool scope is in NS level. each NS can define its own default 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 will look like:

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

in this example, NS ns-user-1 is given a NS level default 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 NS default 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.

## object level FIP pool

the last scope we introduce is object-specific pool. an example will look:

apiVersion: v1  
kind: Service  
metadata:  
 name: service-web-lb-pool-public-1  
 annotations:  
 "opencontrail.org/fip-pool": "{'domain': 'default-domain','name': 'pool-public-1','network': 'vn-public-1','project': 'k8s-ns-user-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.

# ingress in contrail

reviewing

# Services in contrail

## k8s service

service is the core object in kubernetes. in chapter3 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 load balancer that is seating between any client "requesting" a service and the pods "providing" the service. 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 is a kind of virtual IP (VIP) that is also called a cluster 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.

## contrail implementation

in contrail kubernetes integration environment, typically you create 2 types of services:

* clusterIP
* loadbalancer

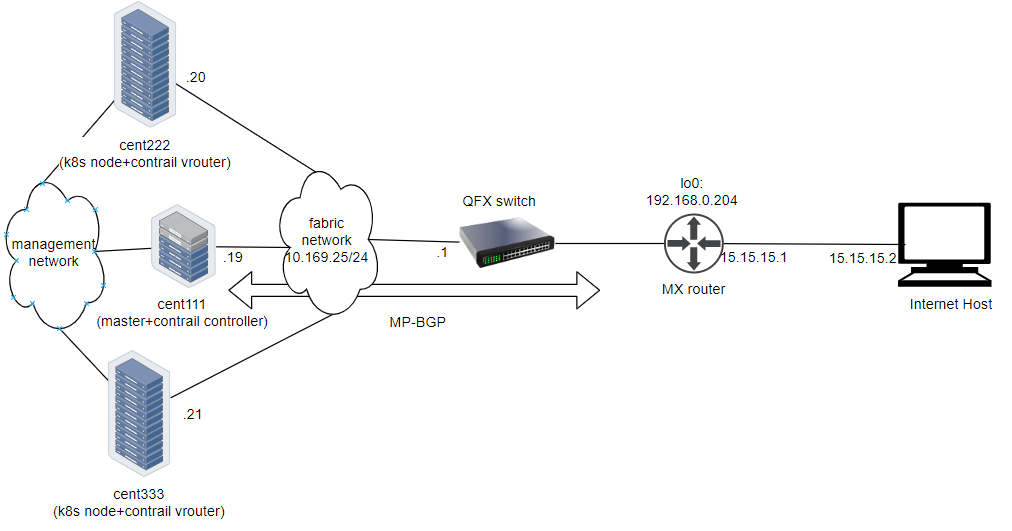
the clusterIP type of service is the default mode if the ServiceType is not given. you can also specify ServiceType as LoadBalancer if you want to expose the service to the external world. what happens whenever a service of type: LoadBalancer get created is that, not only a clusterIP' will be  
allocated and exposed to other pods within the cluster, but also a "Load  
Balancer" instance will be created internally, with a `floating ip assigned 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 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 acoss the gateway router.

we’ll demonstrate how both type of service works in contrail setup, especially, how does the LoadBalancer type of service works in an end to end lab setup including the k8s cluster, fabric switch, gateway router, and Internet host.

## service testbed

our testbed composes of:

* one server runs as k8s master and contrail controllers
* two servers, each is running as a k8s node and contrail vrouter
* one QFX runs as the underlay switch
* one MX as a gateway router
* one server runs as an Internet host machine



**Tip**

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

## service: clusterIP

let’s create our first service in contrail environment, with service type clusterIP.

$ cat 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 "container port" 80 in some pod. the selector indicates whichever pod with a label app: webserver will be choosen to be the backend pod responding service request. in our example it will be pod spawned by a RC object named rc-webserver.

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, the associated endpoint, and backup 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 ep -o wide  
NAME ENDPOINTS AGE  
service-web-lb <none> 10m  
  
$ kubectl get pod -o wide -l 'app=webserver'  
No resources found.

the service is created successfully, there is no doubt about it. but there is endpoint. and the reason is there is pod with the label matching to the SELECTOR in the service. this makes good sense - in chapter 3 you’ve learned what kubernetes endpoint provides is nothing but a list of backend pod with label matching the selector. 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 in chapter 3, use a RC is more pratical. later on you will realize this is the right choice.

$ cat rc-webserver.yaml  
apiVersion: v1  
kind: ReplicationController  
metadata:  
 name: rc-webserver  
 labels:  
 app: webserver  
spec:  
 replicas: 1 #<---  
 selector:  
 app: webserver  
 template:  
 metadata:  
 name: webserver  
 labels:  
 app: webserver #<---  
 spec:  
 containers:  
 - name: webserver  
 image: savvythru/contrail-frontend-app  
 securityContext:  
 privileged: true  
 ports:  
 - containerPort: 80

the RC rc-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 rc-webserver.yaml  
replicationcontroller/rc-webserver created  
  
$ kubectl get ep  
NAME ENDPOINTS AGE  
service-web-clusterip 10.47.255.252:80 2m58s  
  
$ kubectl get pod -o wide -l 'app=webserver'  
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE  
rc-webserver-vl6zs 1/1 Running 0 24s 10.47.255.238 cent333 <none>

immediately the pod is choosen to be the backend, and endpoint reflects the update. 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 service has been associated with an "endpoint" of "10.47.255.252:80", which indicates there **is** a backend pod running with that IP, and in that pod there is a container inside of the pod running a service application (web server in our case) listening on port 80.
* the backend pod can be located with the label selector

**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: kubectl get pod -o wide -l 'app in (webserver)'

there are 3 default subnets existing in contrail kubernetes environment, each serving a different feature:

* pod subnet
* service subnet
* ip fabric subnet

throught this book you will see pod by default gets the interface IP from the pod subnet 10.32/12, each service gets its cluster IP from service subnet 10.96/12. To examine these subnet you can login to the contrail-kube-manager docker container kubemanager\_kubemanager\_1 and find these subnets in the config file:

(kubernetes-kube-manager)[root@cent111 /]$ grep subnet /etc/contrail/contrail-kubernetes.conf  
pod\_subnets=10.32.0.0/12  
ip\_fabric\_subnets=10.64.0.0/12  
service\_subnets=10.96.0.0/12

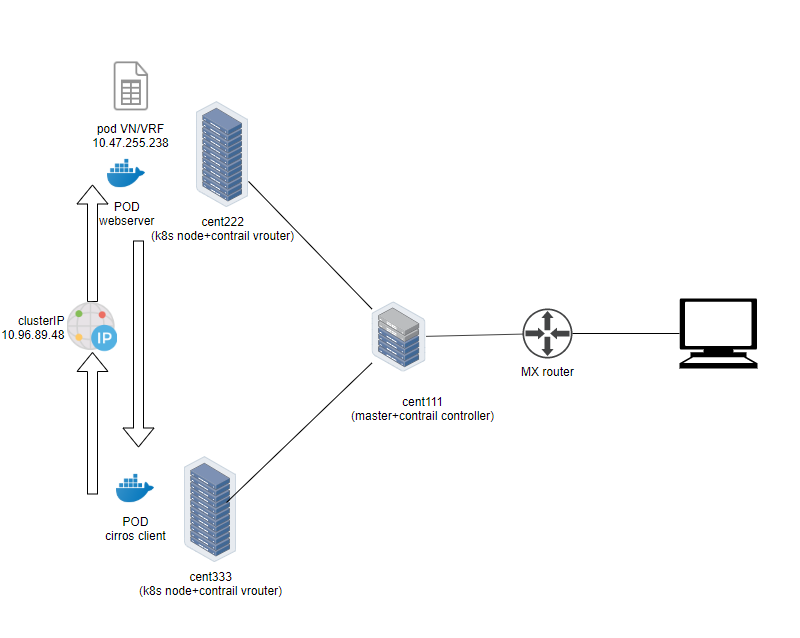
## verify the cluserIP service

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 login to the same cirros pod as you’ve seen in "multiple interface pod" section, and use curl command to send a http request toward the service:

$ kubectl exec -it cirros -- 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 =  
 rc-webserver-vl6zs</h3>  
 <img src="/static/giphy.gif">  
 </body>  
 </div>  
</html>

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

the traffic flow is illustrated here:



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 pod IP and hostname embeded in it. This way the curl returns something more meaningful to our test.

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

$ kubectl exec -it cirros -- 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 = rc-webserver-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 rc-webserver-vl6zs.

## service: loadbalancer

Next let’s look at LoadBalancer type of service. the yaml file looks very similar except just one more line declaring the service type:

$ cat 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 will understand this "EXTERNAL-IP" is actually a FIP allocated from the NS level FIP pool - we did not give any specific FIP pool information in the yaml file so NS default FIP pool will be used automatically.

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 inside of the cluster. Examining the gateway router’s VRF table reveals this:

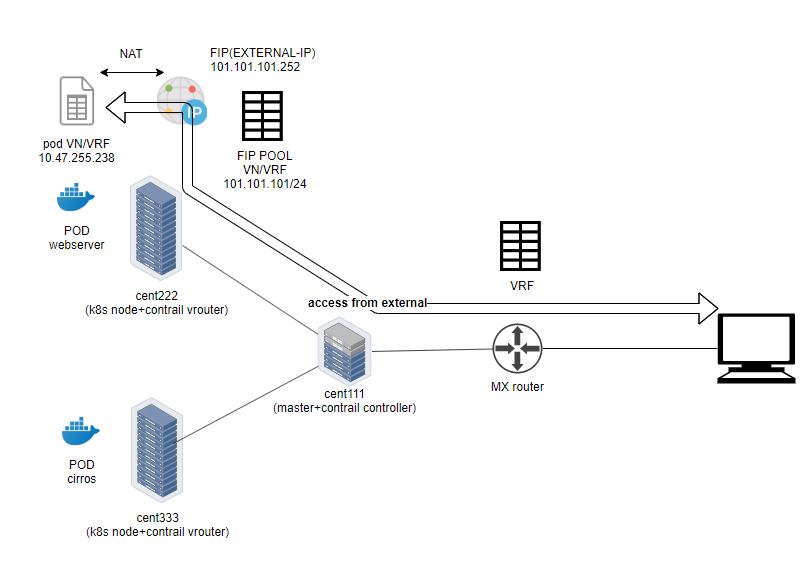
labroot@camaro> show route table k8s-test.inet.0 101.101.101/24  
Jun 19 03:56:11  
  
k8s-test.inet.0: 23 destinations, 40 routes (23 active, 0 holddown, 0 hidden)  
+ = Active Route, - = Last Active, \* = Both  
  
101.101.101.252/32 \*[BGP/170] 00:01:11, MED 100, localpref 200, from 10.169.25.19  
 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 reflects 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: 10.169.25.20 #<---  
 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 encapsulation:unknown(0x2) encapsulation:mpls-in-udp(0xd) unknown type 8004 value eac4:7a1207 unknown type 8071 value eac4:b unknown type 8084 value eac4:10000 unknown type 8084 value eac4:ff0004 unknown type 8084 value eac4:1020006 unknown type 8084 value eac4:1030001  
 Import Accepted  
 VPN Label: 44  
 Localpref: 200  
 Router ID: 10.169.25.19  
 Primary Routing Table bgp.l3vpn.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 case
* 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 represents the (MPLS over) GRE tunnel between node cent222 and the gateway router.

this diagram below illustrates the service workflow:

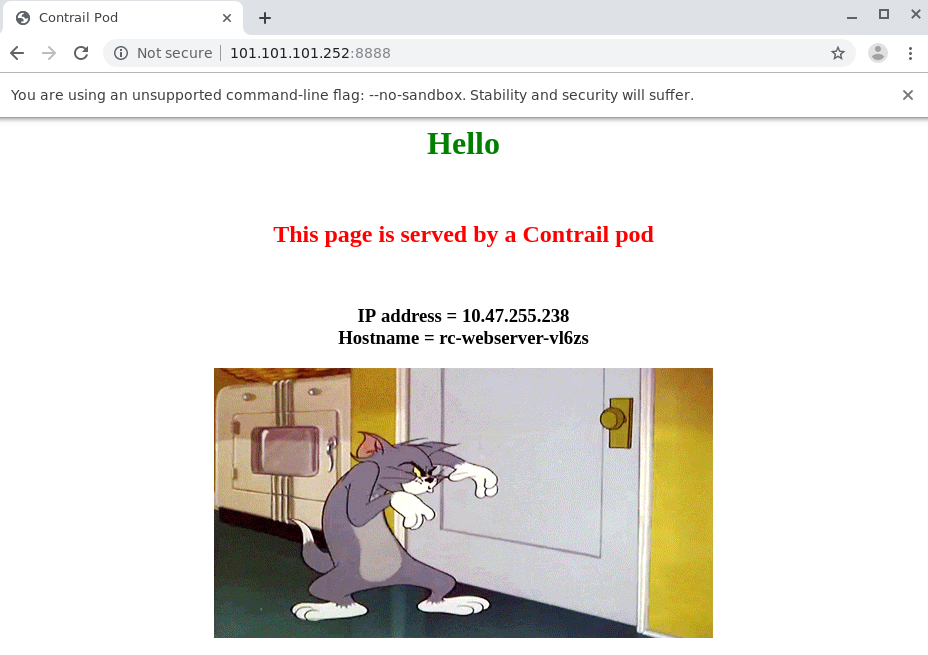


* you create a FIP pool from a public VN, with route-target the VN is advertised to the remote gateway router via MP-BGP
* you create a pod with a label app: webserver, kubernetes decides the pod will be created in node cent222. via XMPP the node publish the pod IP
* you create a loadbalancer type of 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 cent222 advertises the podIP and FIP to contrail controller cent111, which then advertises only the FIP to the gateway router. at this moment the gateway learns the nexthop of the FIP is cent222, so it generate a soft GRE tunnel toward cent222.
* 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 cent222
* 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 the 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. Remember the request has to be sent to the **FIP**, not the **service IP**(**clusterIP**) or backend **podIP**.

this is the returned web page:



in our testbed we installed a centos server as an Internet host. as with any linux distribution, if you need to login the "GUI", you need to install Xwindow or linux desktop applications and set it up properly. also you need a web browser if it does not come with the server.

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 = rc-webserver-vl6zs  
 [giphy.gif]

the kubernetes service is available from Internet!

## service: loadbalancer ECMP

so far you’ve seen how loadbalancer type of service is exposed to the Internet and how the FIP did the "trick". what you haven’t seen yet is how does the "loadbalancer" perform "ECMP" processing. To demonstrate this we need more than one backend pod behind the service. 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 RC or deployment, as what you`ve seen in chapter 3 and previous sections. in our service example we`ve been using RC object to spawn our webserver pod on purpose:

$ kubectl scale rc rc-webserver --replicas=2  
replicationcontroller/rc-webserver scaled  
  
$ kubectl get pod -l app=webserver -o wide  
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE  
rc-webserver-r9zdt 1/1 Running 0 25m 10.47.255.238 cent333 <none>  
rc-webserver-xkjpw 1/1 Running 0 23s 10.47.255.236 cent222 <none>

immediately after you create a new webserver pod by scaling the RC with replicas 2, a new pod is launched from the other node cent333, 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.

here is the question: with 2 pods on different node as backend now, from the gatway 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 has been learned twice! an additional MPLSoGRE tunnel is created. besides giving detail option to the show route command, another method to find the tunnel endpoints is to examine the soft GRE gr- interface:

labroot@camaro> show interfaces gr-2/2/0.32771  
Jun 30 00:56:01  
 Logical interface gr-2/2/0.32771 (Index 392) (SNMP ifIndex 1801)  
 Flags: Up Point-To-Point SNMP-Traps 0x4000  
 IP-Header 10.169.25.21:192.168.0.204:47:df:64:0000000800000000 #<---  
 Encapsulation: GRE-NULL  
 Copy-tos-to-outer-ip-header: Off, Copy-tos-to-outer-ip-header-transit: Off  
 Gre keepalives configured: Off, Gre keepalives adjacency state: down  
 Input packets : 0  
 Output packets: 0  
 Protocol inet, MTU: 9142  
 Max nh cache: 0, New hold nh limit: 0, Curr nh cnt: 0, Curr new hold cnt: 0, NH drop cnt: 0  
 Flags: None  
 Protocol mpls, MTU: 9130, Maximum labels: 3  
 Flags: None  
  
labroot@camaro> show interfaces gr-2/3/0.32771  
 Logical interface gr-2/3/0.32771 (Index 393) (SNMP ifIndex 1703)  
 Flags: Up Point-To-Point SNMP-Traps 0x4000  
 IP-Header 10.169.25.20:192.168.0.204:47:df:64:0000000800000000 #<---  
 Encapsulation: GRE-NULL  
 Copy-tos-to-outer-ip-header: Off, Copy-tos-to-outer-ip-header-transit: Off  
 Gre keepalives configured: Off, Gre keepalives adjacency state: down  
 Input packets : 11  
 Output packets: 11  
 Protocol inet, MTU: 9142  
 Max nh cache: 0, New hold nh limit: 0, Curr nh cnt: 0, Curr new hold cnt: 0, NH drop cnt: 0  
 Flags: None  
 Protocol mpls, MTU: 9130, Maximum labels: 3  
 Flags: None

the IP-Header of 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 node where backend pod is running. now you understand the router will perform ECMP load balancing between the two GRE tunnel, whenever it got service request toward the same FIP.

## verify the service ECMP

till now we believe we’ve got the whole picture of loadbalancer service, to verify it we’ll just pull the webpage a few more time and we should see both podIP displayed.

turns out this never happens.

[root@cent-client ~]# curl http://101.101.101.252:8888 | lynx -stdin --dump  
 Hello  
This page is served by a Contrail pod  
 IP address = 10.47.255.236  
 Hostname = rc-webserver-xkjpw

the only webpage we got is from 10.47.255.236, by backend pod rc-webserver-xkjpw, running in node cent222. the other one never respond, indicating the expected ECMP does not happen yet. with Junos the detail or extensive keyword tells the reason:

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

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 nature of the default Junos BGP path selection and detail discussion of it is out of the scope of this book. the solution is to enable the multipath  
vpn-unequal-cost knob under the VRF:

labroot@camaro# set routing-instances ping-tes routing-options multipath vpn-unequal-cost

now a Multipath with both GRE interface will be added under the FIP prefix, and the forwarding table reflects the same:

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-2/1/0.32771, 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-2/1/0.32771, Push 72  
 > via gr-2/2/0.32771, Push 52  
  
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

**Tip**

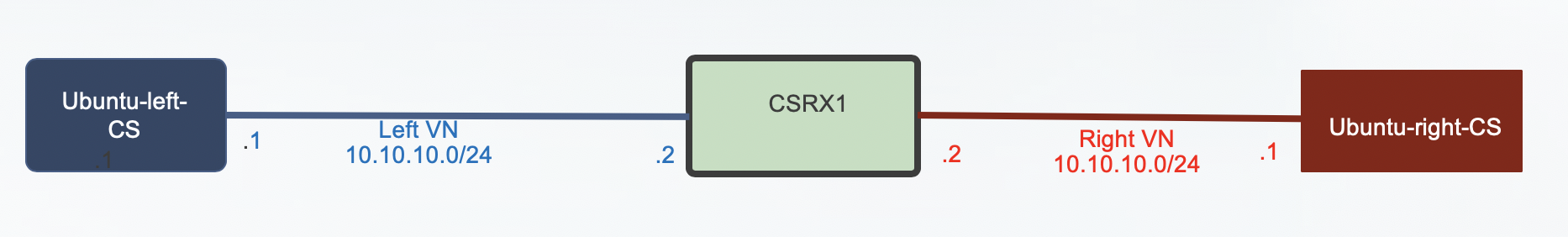
for Junos BGP path selection algorithm, check this link: <https://www.juniper.net/documentation/en_US/junos/topics/topic-map/bgp-path-selection.html>

now try to pull the webpage from Internet host multiple times with curl or web browser, you will see the random result.

[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 = rc-webserver-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 = rc-webserver-r9zdt

# service chaining with CSRX

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 diagram



**Tip**

* 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

so let’s start create two networks using this YAML files

[root@cent11]# cat vn-left.yaml  
apiVersion: k8s.cni.cncf.io/v1  
kind: NetworkAttachmentDefinition  
metadata:  
 annotations:  
 opencontrail.org/cidr: “10.10.10.0/24"  
 opencontrail.org/ip\_fabric\_forwarding: "false"  
 opencontrail.org/ip\_fabric\_snat: "false"  
 name: vn-left  
 namespace: default  
spec:  
 config: '{ "cniVersion": "0.3.0", "type": "contrail-k8s-cni" }'  
  
[root@cent11]# cat vn-left.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" }'

[root@cent11]# kubectl create -f vn-left.yaml  
[root@cent11]# kubectl create -f vn-right.yaml

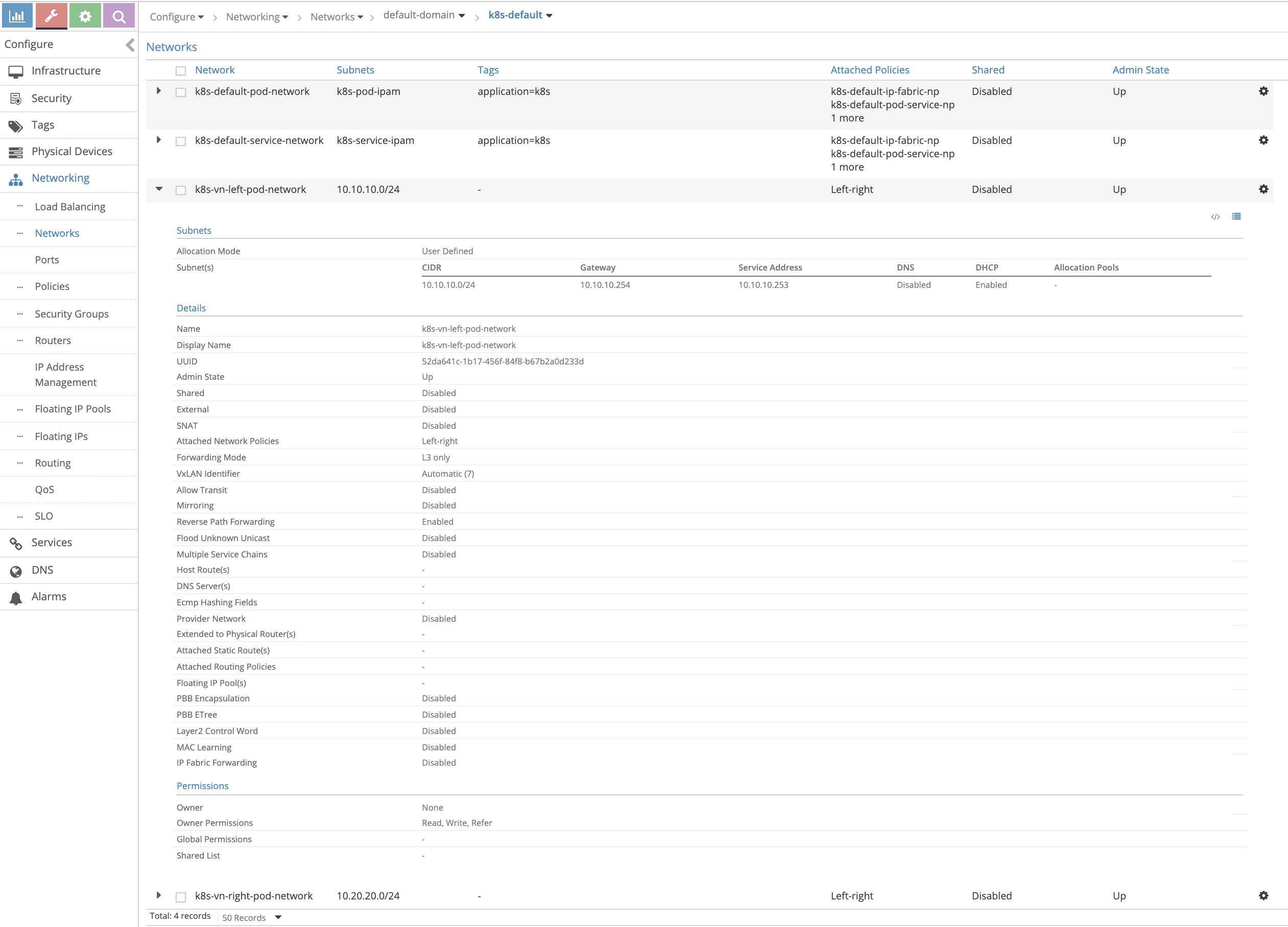
Verify using Kubectl

[root@cent11 ~]# kubectl get network-attachment-definition  
NAME AGE  
vn-left 19d  
vn-right 17d  
  
[root@cent11 ~]# 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 Controller module control node (<http://10.85.188.16:8143> in our setup), select Configure > Networking > Networks > default-domain > k8s-default As shown in the diagram which focus on left network

**Tip**

using namespace: default object in the YAML file for a network will create it n in domain “default-domain” and project “K8s-default”



Create two ubuntu Pods, one in each network using the annotation object

[root@cent11 ~]# cat 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: virtualhops/ato-ubuntu:latest  
 securityContext:  
 privileged: true  
 capabilities:  
 add:  
 - NET\_ADMIN  
  
  
[root@cent11 ~]# cat 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: virtualhops/ato-ubuntu:latest  
 securityContext:  
 privileged: true  
 capabilities:  
 add:  
 - NET\_ADMIN  
  
[root@cent11 ~]# kubectl create -f right-ubuntu-sc.yaml  
[root@cent11 ~]# kubectl create -f left-ubuntu-sc.yaml  
  
  
[root@cent11 ~]# kubectl get pod  
NAME READY STATUS RESTARTS AGE  
left-ubuntu-sc 1/1 Running 0 25h  
right-ubuntu-sc 1/1 Running 0 25h

create Juniper CSRX container that have one interface on the left network and one interface on the right network using this YAML file

[root@cent11 ~]# cat csrx1-sc.yaml  
apiVersion: v1  
kind: Pod  
metadata:  
 name: csrx1-sc  
 labels:  
 app: webapp-sc  
 annotations:  
 k8s.v1.cni.cncf.io/networks: '[  
 { "name": "vn-left" },  
 { "name": "vn-right" }  
 ]'  
spec:  
 containers:  
 - name: csrx1-sc  
 image: csrx  
 ports:  
 - containerPort: 22  
 imagePullPolicy: Never  
 stdin: true  
 tty: true  
 securityContext:  
 privileged: true  
  
[root@cent11 ~]# kubectl create -f csrx1-sc.yaml  
  
  
Confirm the interface placement in the correct network  
  
[root@cent11 ~]# kubectl describe pod  
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: csrx  
 Image ID: docker://sha256:329e805012bdf081f4a15322f994e5e3116b31c90f108a19123cf52710c7617e  
 Port: 22/TCP  
 Host Port: 0/TCP  
 State: Running  
 Started: Thu, 13 Jun 2019 03:40:46 -0400  
 Ready: True  
 Restart Count: 0  
 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: <none>  
  
  
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: virtualhops/ato-ubuntu:latest  
 Image ID: docker-pullable://virtualhops/ato-ubuntu@sha256:fa2930cb8f4b766e5b335dfa42de510ecd30af6433ceada14cdaae8de9065d2a  
 Port: <none>  
 Host Port: <none>  
 State: Running  
 Started: Thu, 13 Jun 2019 03:40:27 -0400  
 Ready: True  
 Restart Count: 0  
 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: <none>  
  
  
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: virtualhops/ato-ubuntu:latest  
 Image ID: docker-pullable://virtualhops/ato-ubuntu@sha256:fa2930cb8f4b766e5b335dfa42de510ecd30af6433ceada14cdaae8de9065d2a  
 Port: <none>  
 Host Port: <none>  
 State: Running  
 Started: Thu, 13 Jun 2019 04:09:25 -0400  
 Ready: True  
 Restart Count: 0  
 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: <none>

**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

Login to the left, right Pods and the CSRX to confirm the IP/MAC address

[root@cent11 ~]# 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  
  
  
  
[root@cent11 ~]# 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  
  
  
[root@cent11 ~]# 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 start with factory default configuration hence it need to be configured.

**Tip**

By default, CSRX eth0 is visible only from shell and used for management. And when attaching networks, the first attach 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 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

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

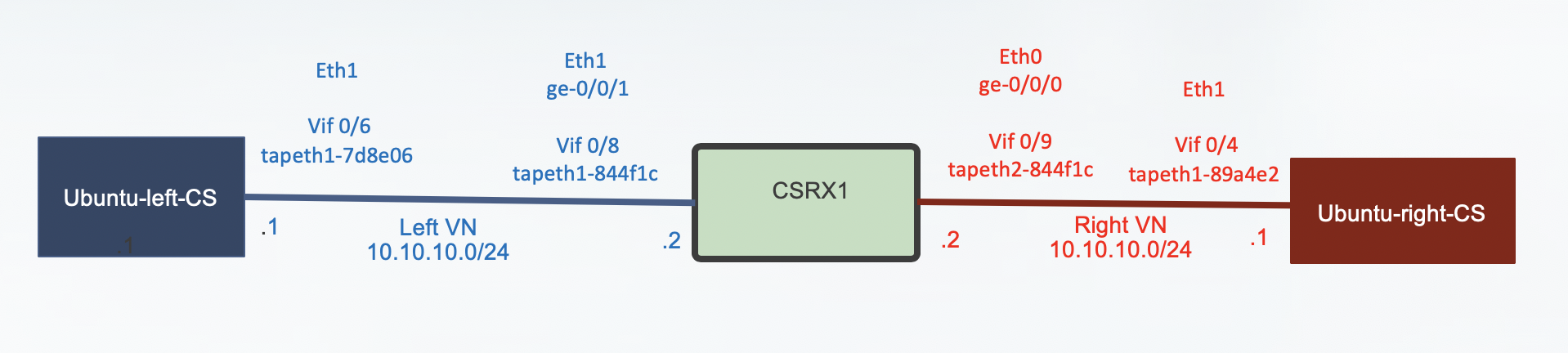
Adding static route to the left and right PODs and try to ping again

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.  
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  
  
vif0/0 OS: ens160 (Speed 10000, Duplex 1)  
 Type:Physical HWaddr:00:50:56:9e:bb:98 IPaddr:0.0.0.0  
 Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpEr QOS:-1 Ref:7  
 RX packets:896021 bytes:291885987 errors:0  
 TX packets:885150 bytes:851902554 errors:0  
 Drops:20  
  
vif0/1 OS: vhost0  
 Type:Host HWaddr:00:50:56:9e:bb:98 IPaddr:10.85.188.17  
 Vrf:0 Mcast Vrf:65535 Flags:PL3DEr QOS:-1 Ref:8  
 RX packets:883626 bytes:851837334 errors:0  
 TX packets:912984 bytes:292597411 errors:0  
 Drops:17  
  
vif0/2 OS: pkt0  
 Type:Agent HWaddr:00:00:5e:00:01:00 IPaddr:0.0.0.0  
 Vrf:65535 Mcast Vrf:65535 Flags:L3Er QOS:-1 Ref:3  
 RX packets:151865 bytes:13060426 errors:0  
 TX packets:465234 bytes:52562395 errors:0  
 Drops:0  
  
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  
  
vif0/4350 OS: pkt3  
 Type:Stats HWaddr:00:00:00:00:00:00 IPaddr:0.0.0.0  
 Vrf:65535 Mcast Vrf:65535 Flags:L3L2 QOS:0 Ref:1  
 RX packets:0 bytes:0 errors:0  
 TX packets:0 bytes:0 errors:0  
 Drops:0  
  
vif0/4351 OS: pkt1  
 Type:Stats HWaddr:00:00:00:00:00:00 IPaddr:0.0.0.0  
 Vrf:65535 Mcast Vrf:65535 Flags:L3L2 QOS:0 Ref:1  
 RX packets:8 bytes:552 errors:0  
 TX packets:8 bytes:552 errors:0  
 Drops:0

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 diagram 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 CSRX 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 in the compute node

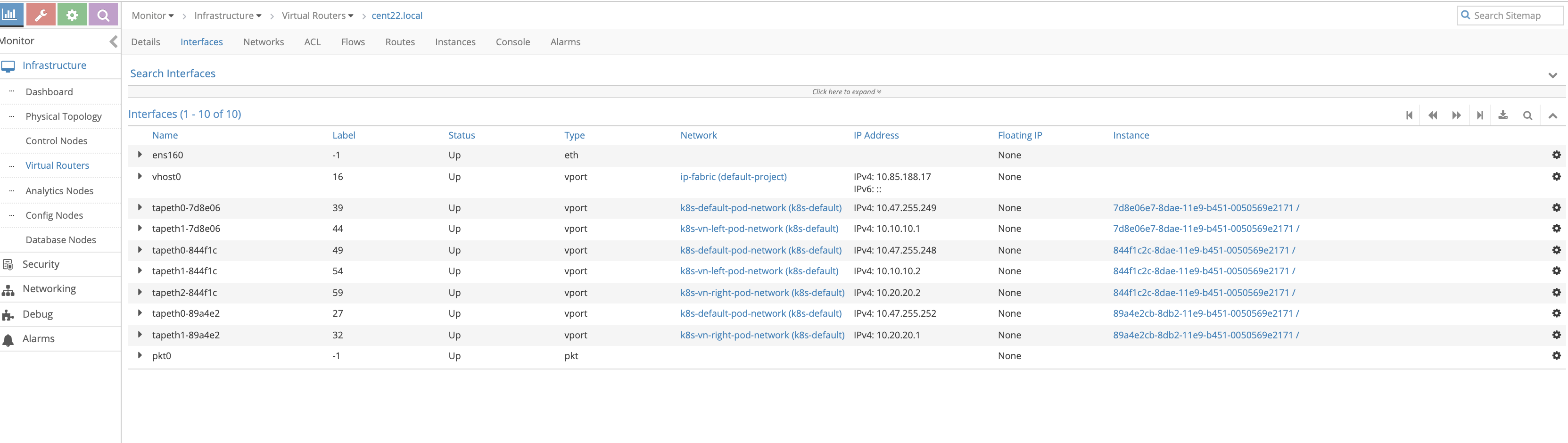
[root@cent22 ~]# docker ps | grep vrouter  
9a737df53abe ci-repo.englab.juniper.net:5000/contrail-vrouter-agent:master-latest "/entrypoint.sh /usr…" 2 weeks ago Up 47 hours vrouter\_vrouter-agent\_1  
e25f1467403d ci-repo.englab.juniper.net:5000/contrail-nodemgr:master-latest "/entrypoint.sh /bin…" 2 weeks ago Up 47 hours vrouter\_nodemgr\_1  
  
[root@cent22 ~]# docker exec -it vrouter\_vrouter-agent\_1 bash  
(vrouter-agent)[root@cent22 /]$  
(vrouter-agent)[root@cent22 /]$ rt --dump 6 | grep 10.20.20.  
(vrouter-agent)[root@cent22 /]$

Note that 6 is the routing table VRF of the left network, 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 /]$

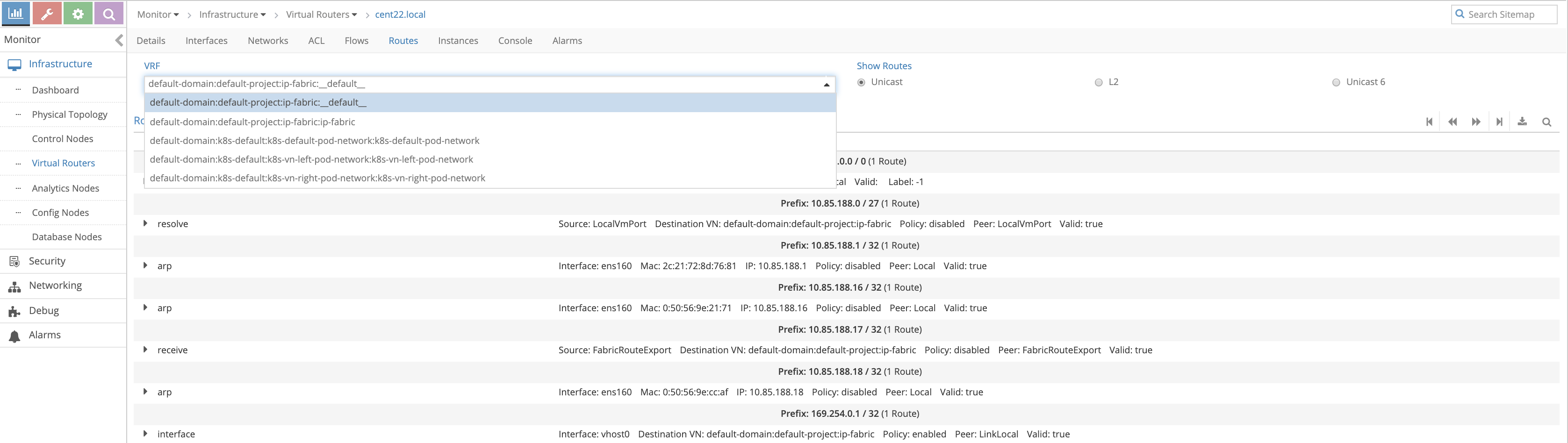
So even if all the PODs are hosted on the same compute nodes, they can’t reach each other. And if these PODs are hosted on different compute nodes then you have a bigger problem to solve Service chaining isn’t about adjusting the routes on the containers but mainly about exchange routes between the vrouter-agent 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 module control node (<http://10.85.188.16:8143> in oursetup), select monitor > infrastructure > virtual router then select the node the that host the POD , in our case “Cent22.local”

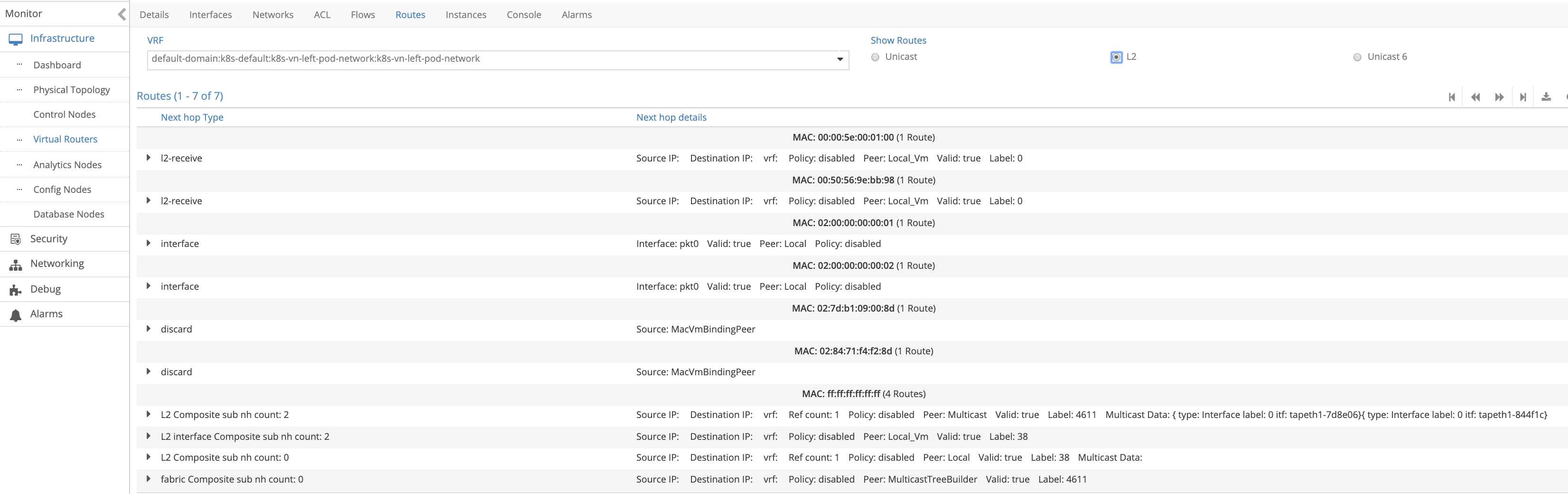


as shown in the diagram from the interface tab which is equivalent to running “ vif -l” command on the vrouter\_vrouter-agent-1 container and even showing more information notice the mapping between the instance ID and tap interface naming where the first 6 character of the instance ID are always reflected 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



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 equvilant to “rt --dump 6 --family bridge” command



Now lets utilize the CSRX to service chaining using contrail command GUI

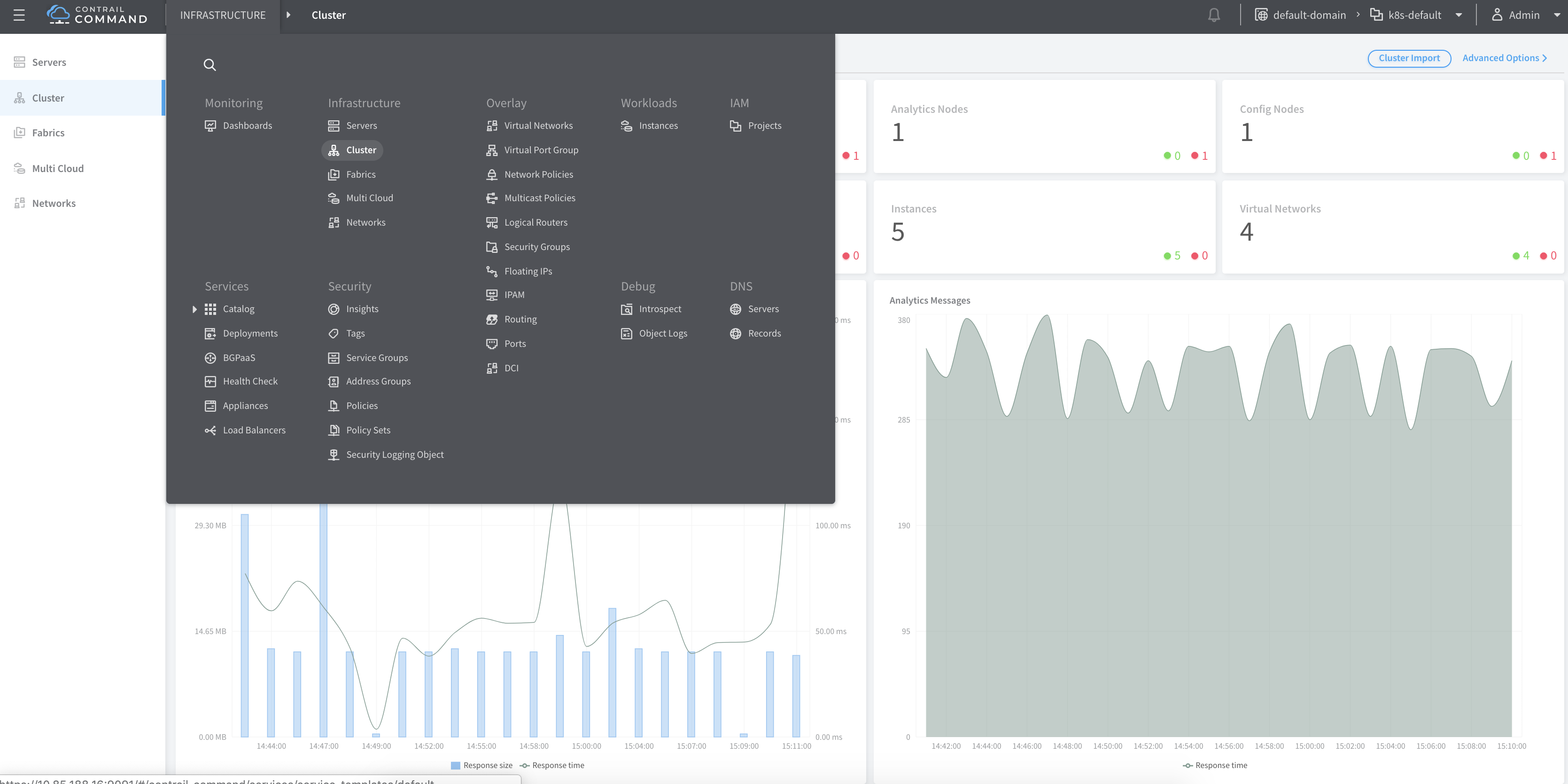
creating Service chaining is 4 steps make sure to do them in this order

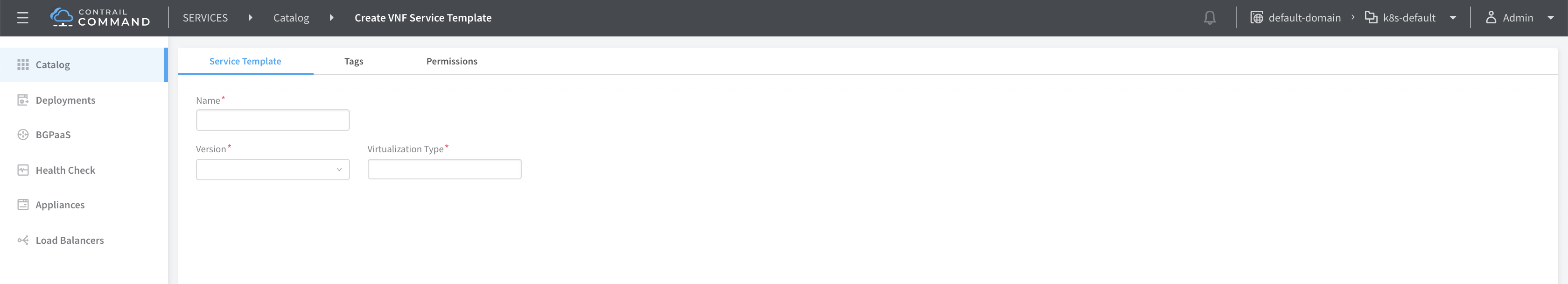
1. create Service template
2. creating service instance based on the service template you created before
3. creating network policy and select the service instance you created before
4. apply this network policy on network

**Note**

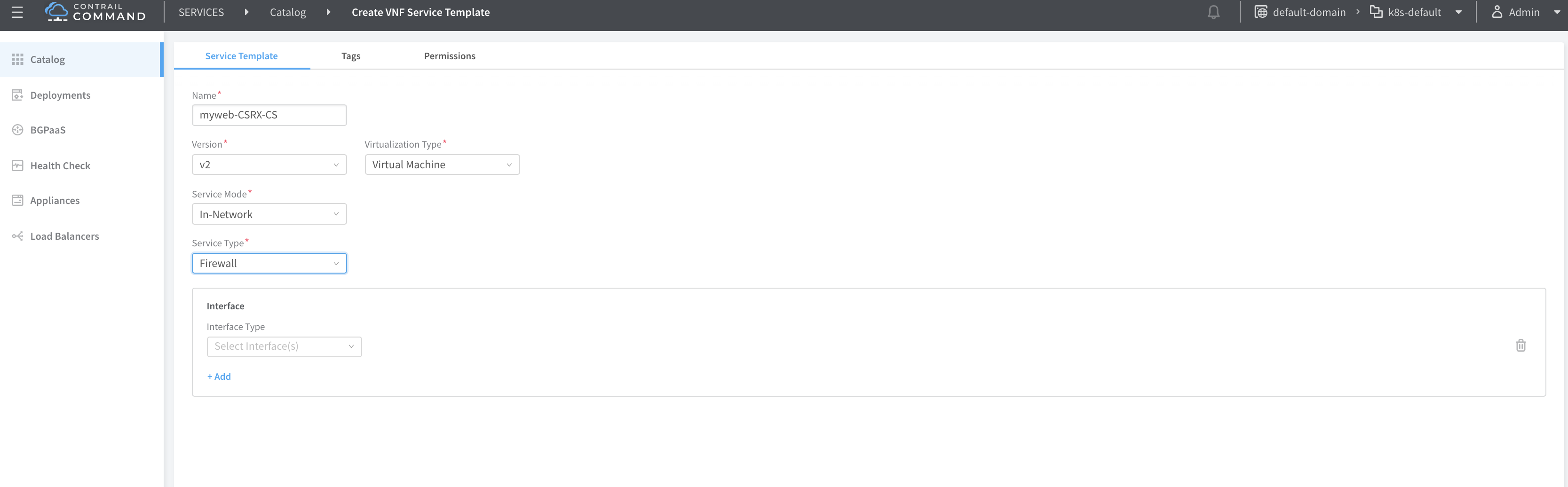
since contrail command GUI is the solution to provide a single point of management for all environments, we will use it to build service changing but you still can use the normal contrail controller GUI to build service changing

Login to contrail command GUI ( in our setup <https://10.85.188.16:9091/>) then select service > catalog > create

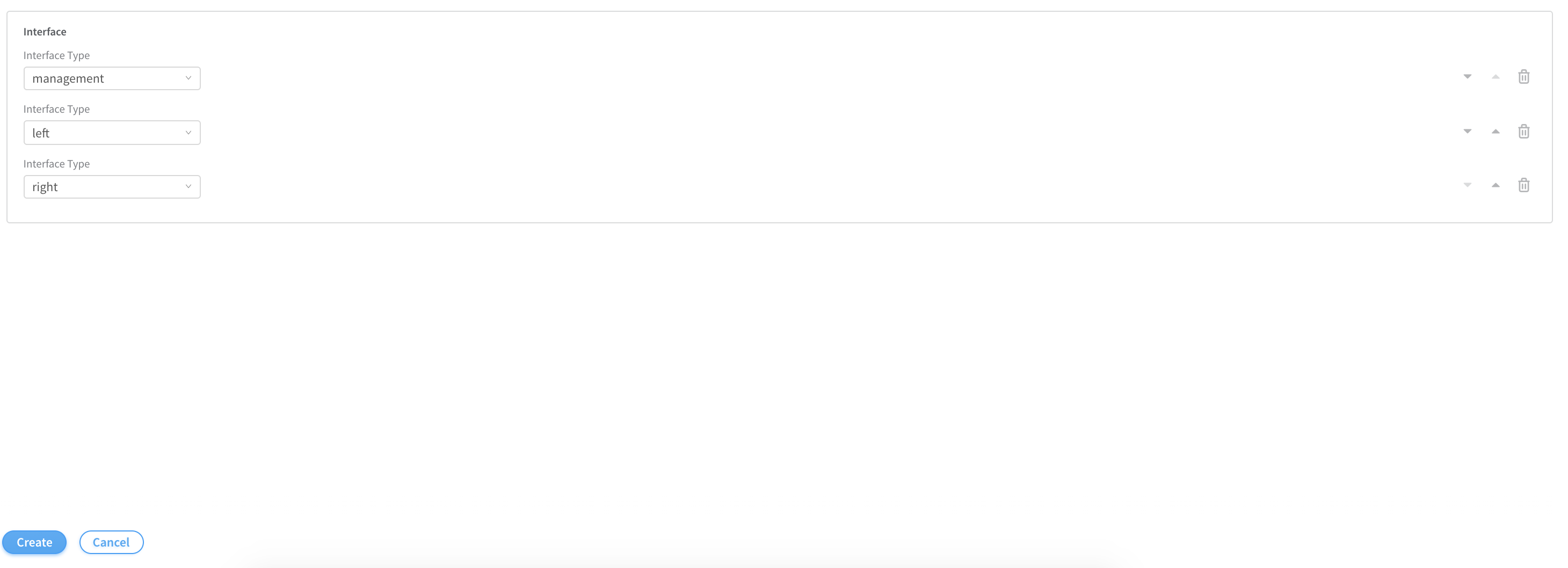




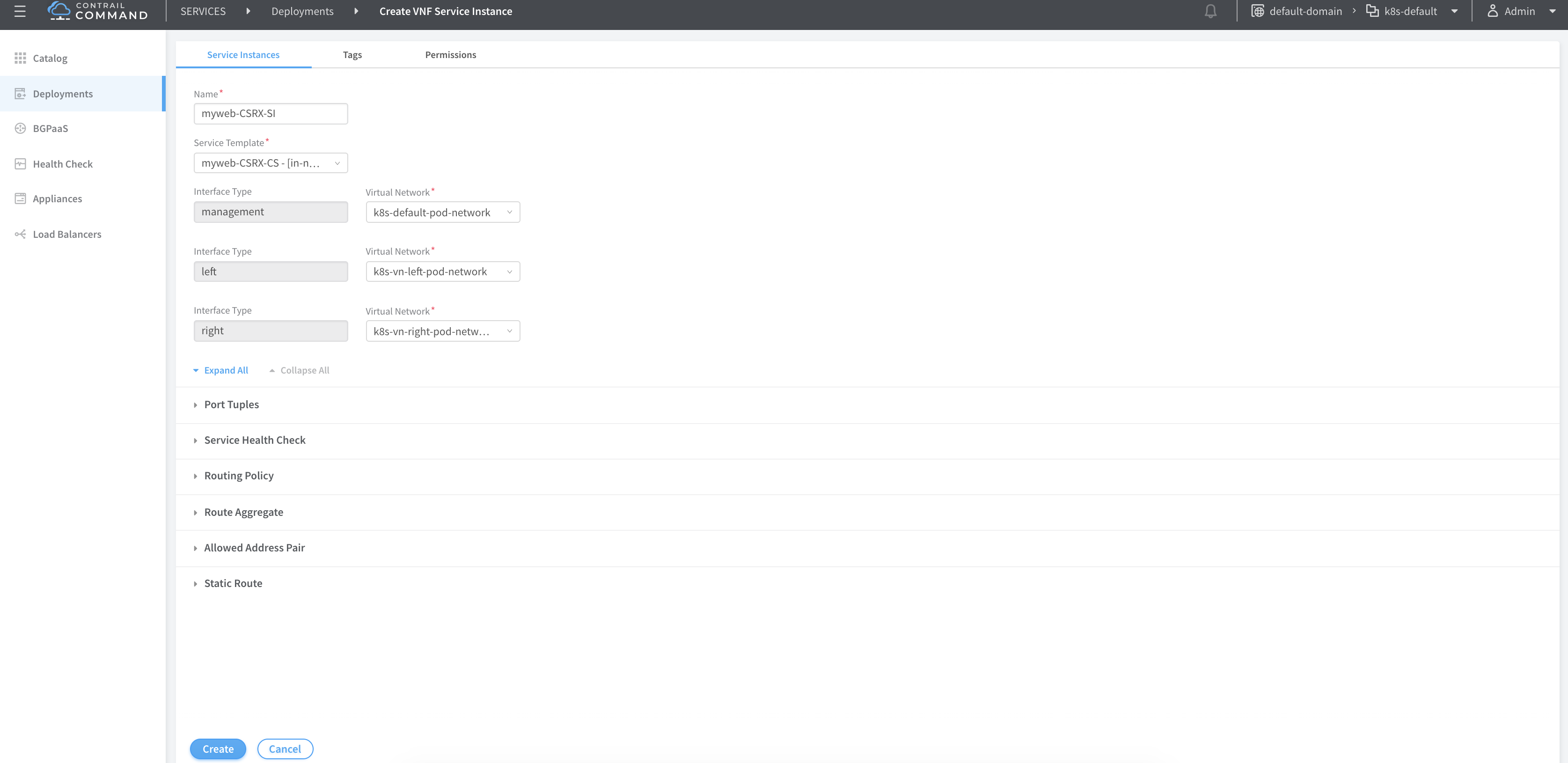
insret a name of services template “myweb-CSRX-CS” in here then chose v2 , virtual machine ( no other option available) for service mode we will work with In-network and firewall as service type



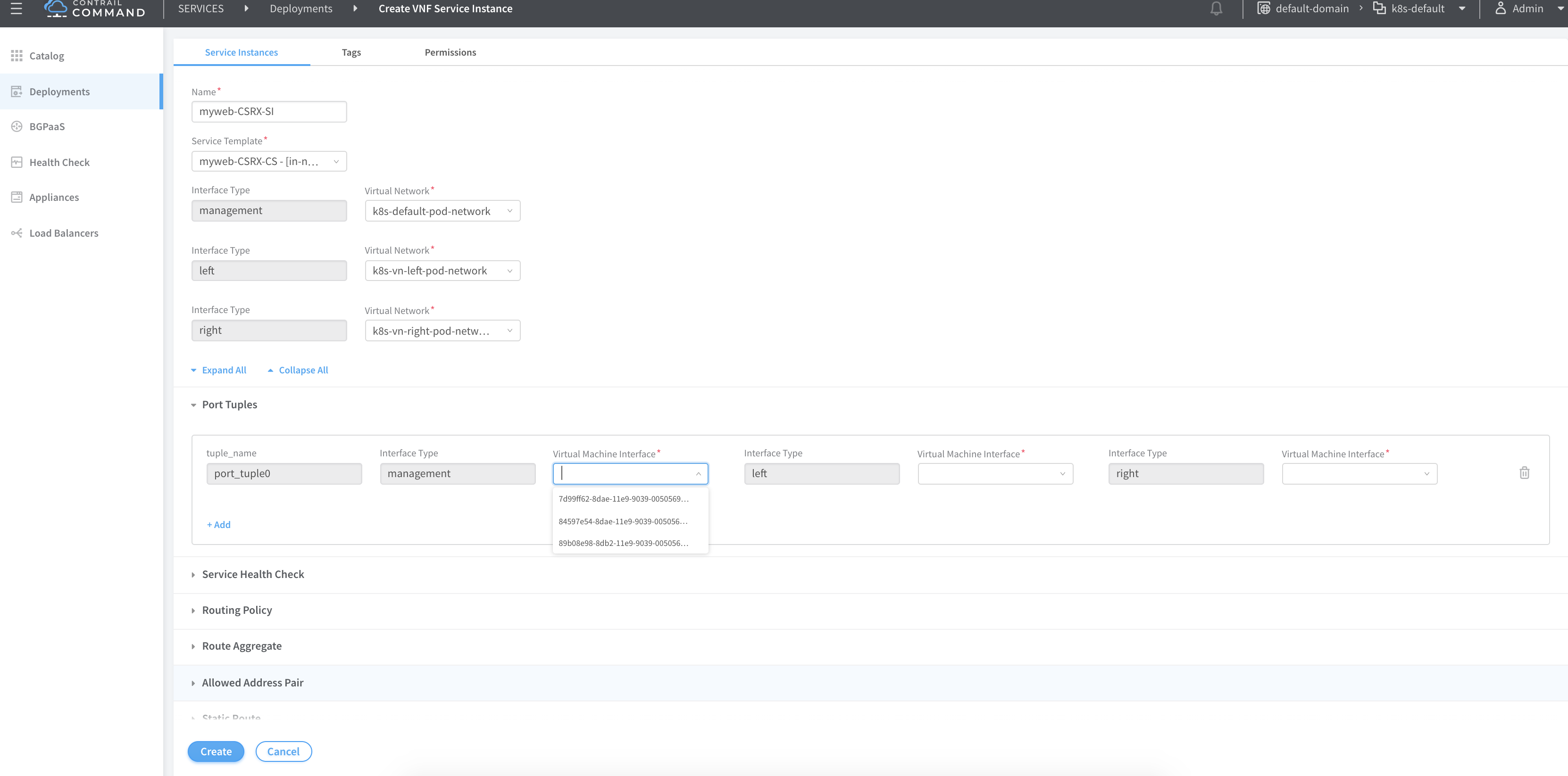
Select interfaces management, left and right then click create



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

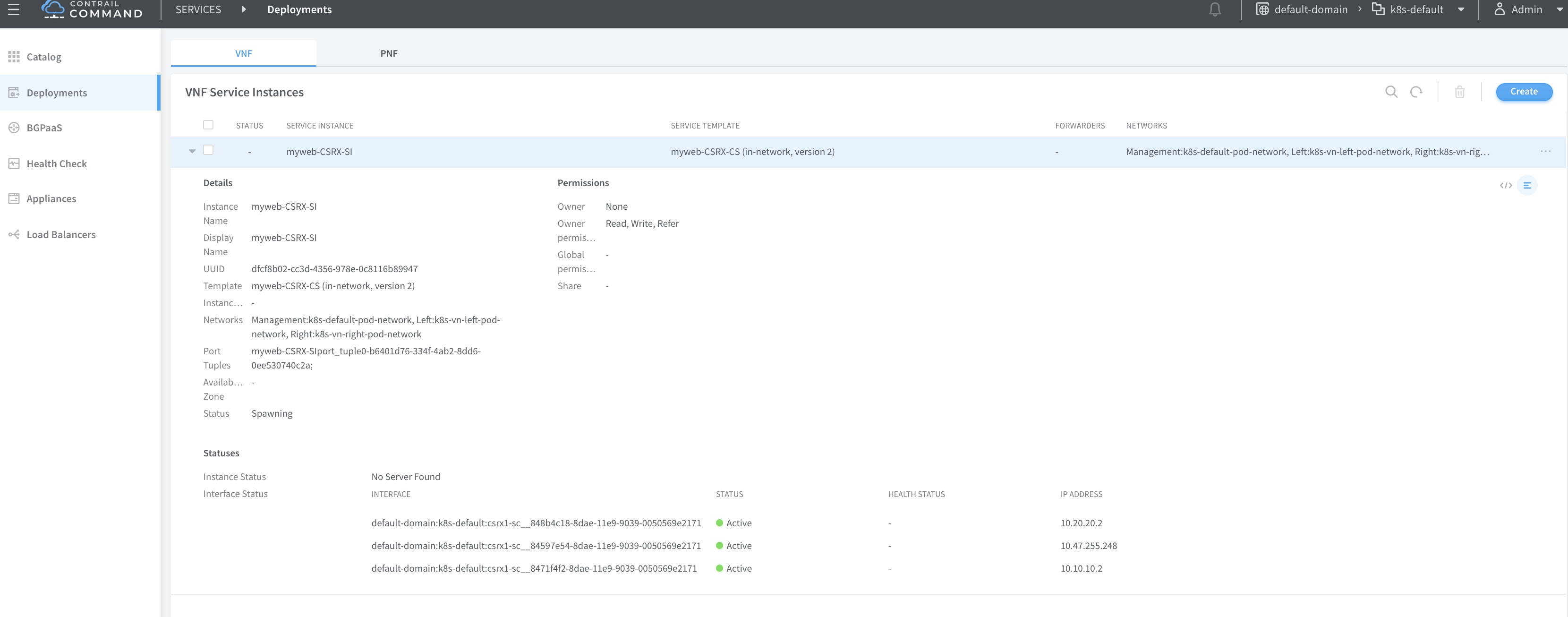


then for each of the three interface bound one interface of the CSRX then click create

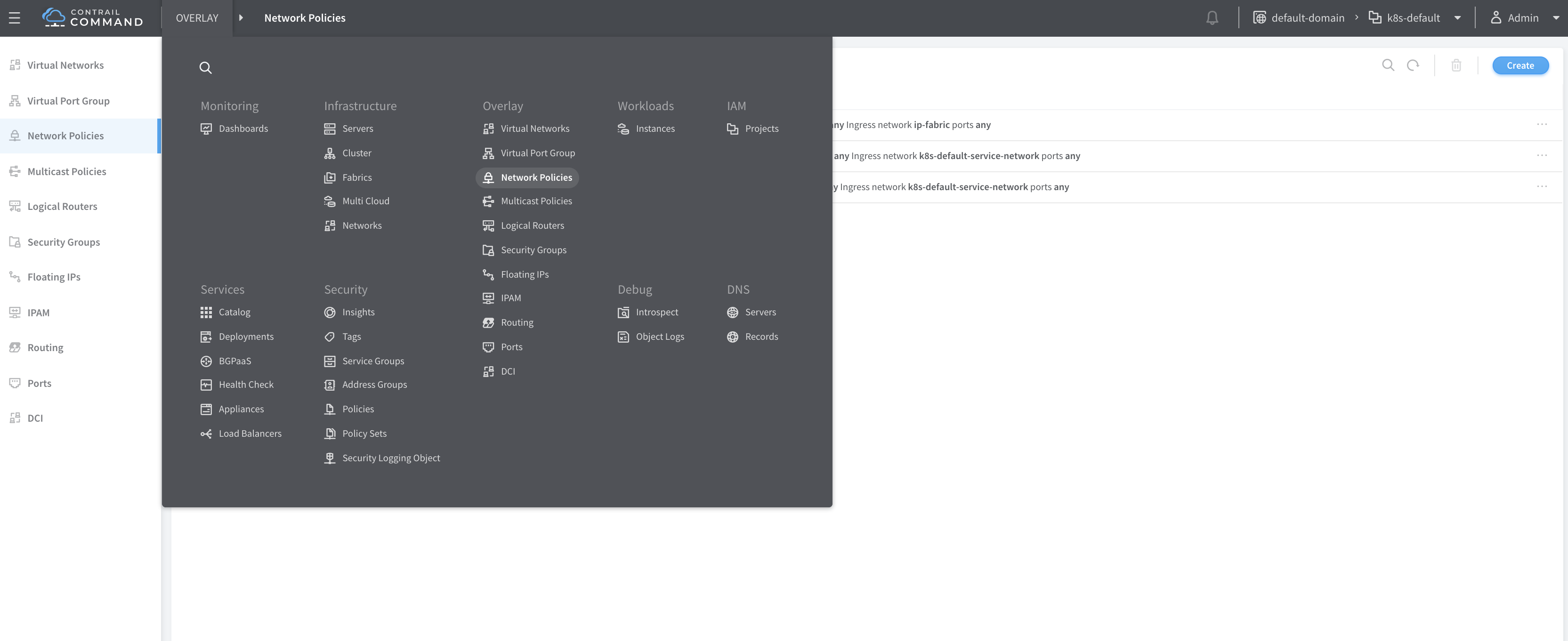
**Tip**

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

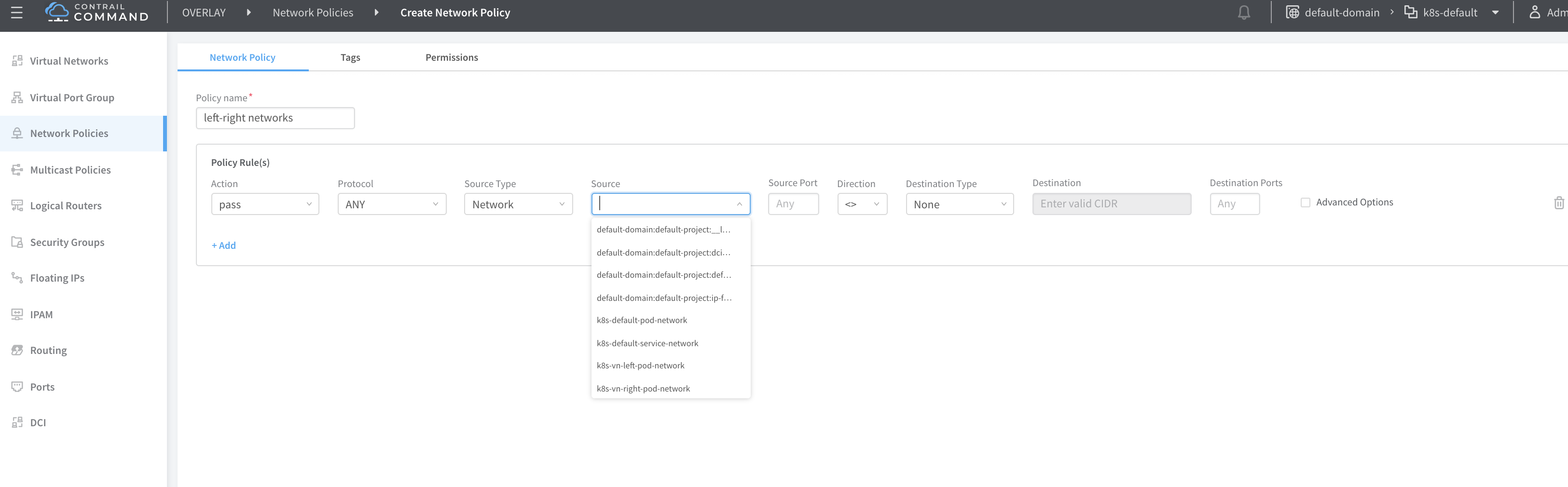
Before you procced make sure the status of the three interfaces are up and they are showing the correct IP address of the CSRX instance



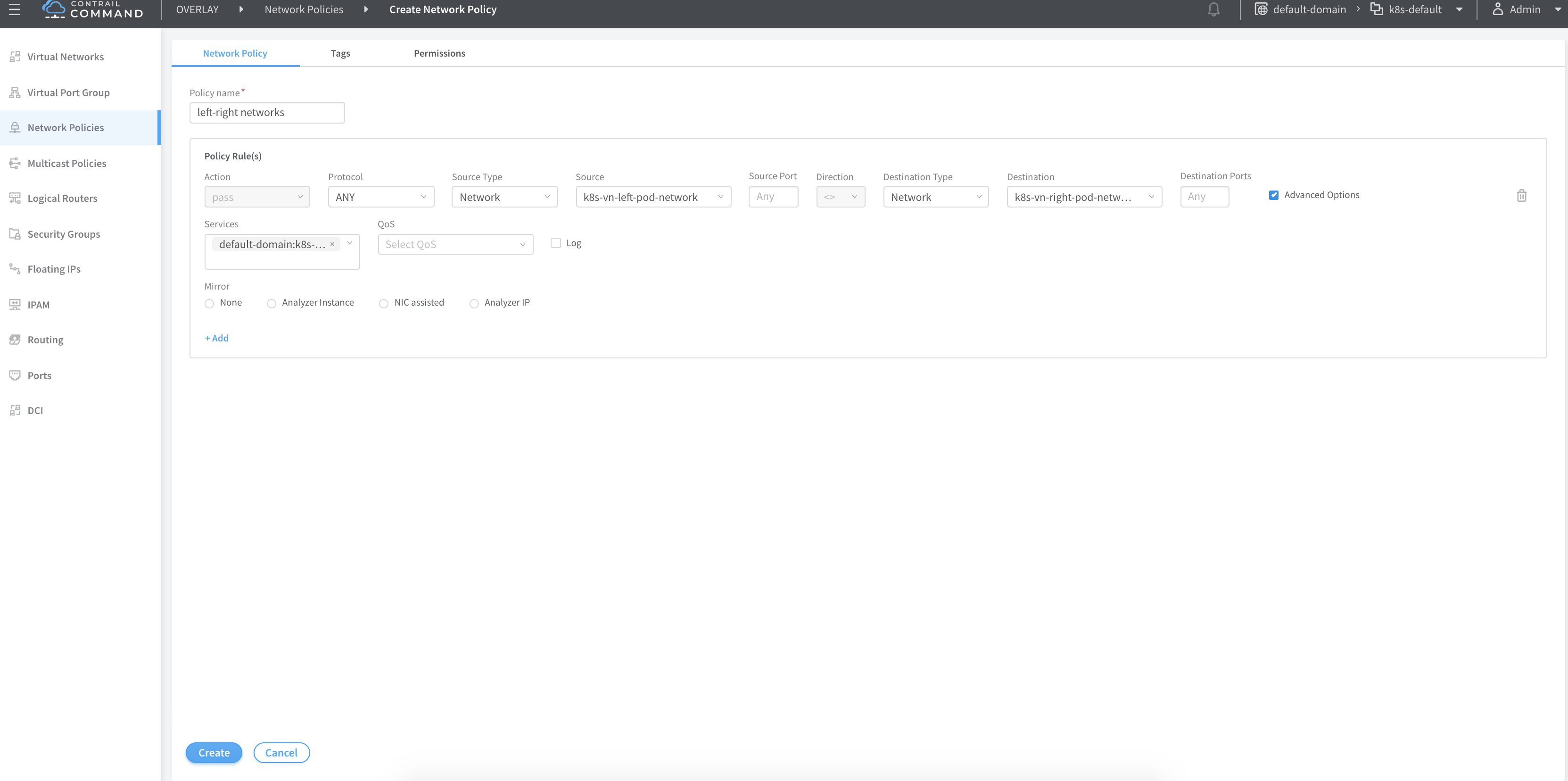
To create network policy go to overlay > network policies > create



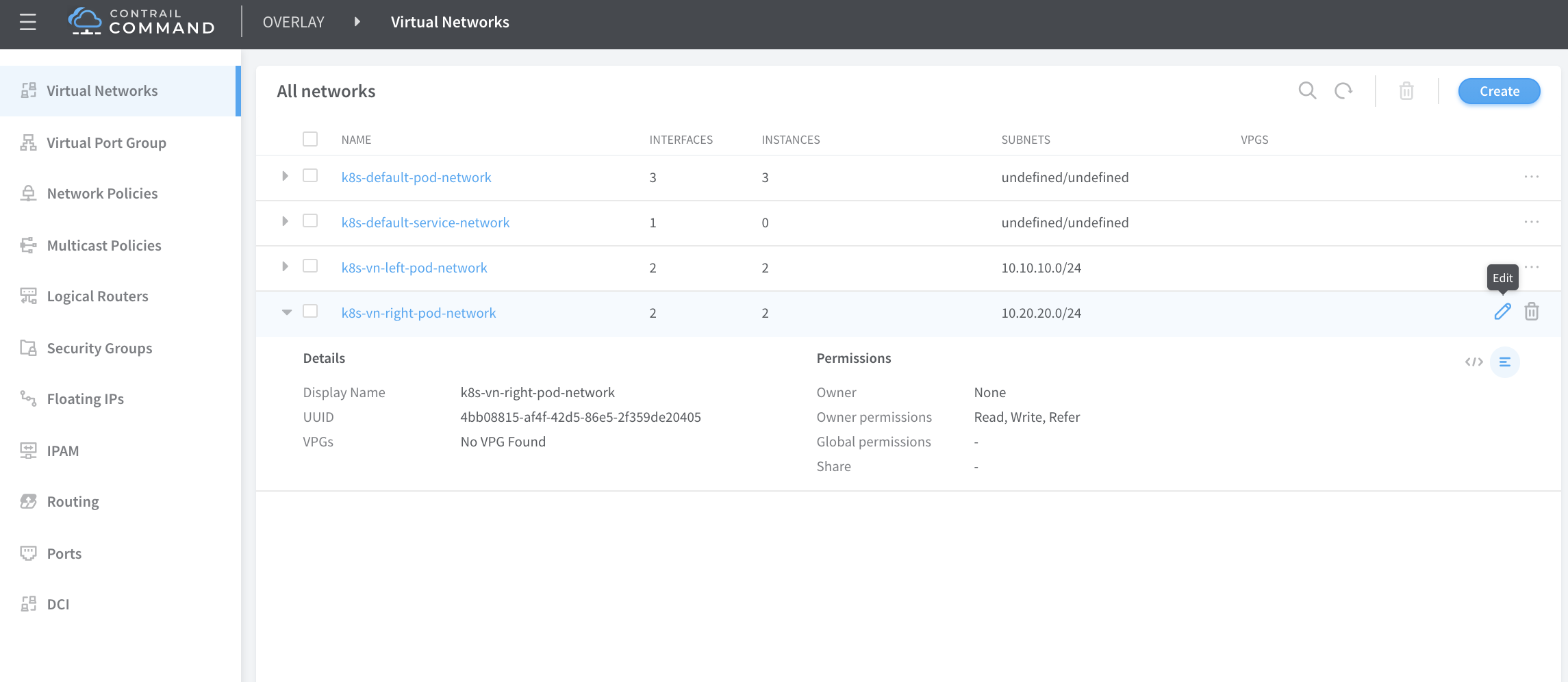
Insert a name for your network policy then in the first rule add left network as source network and right network as destination with action pass



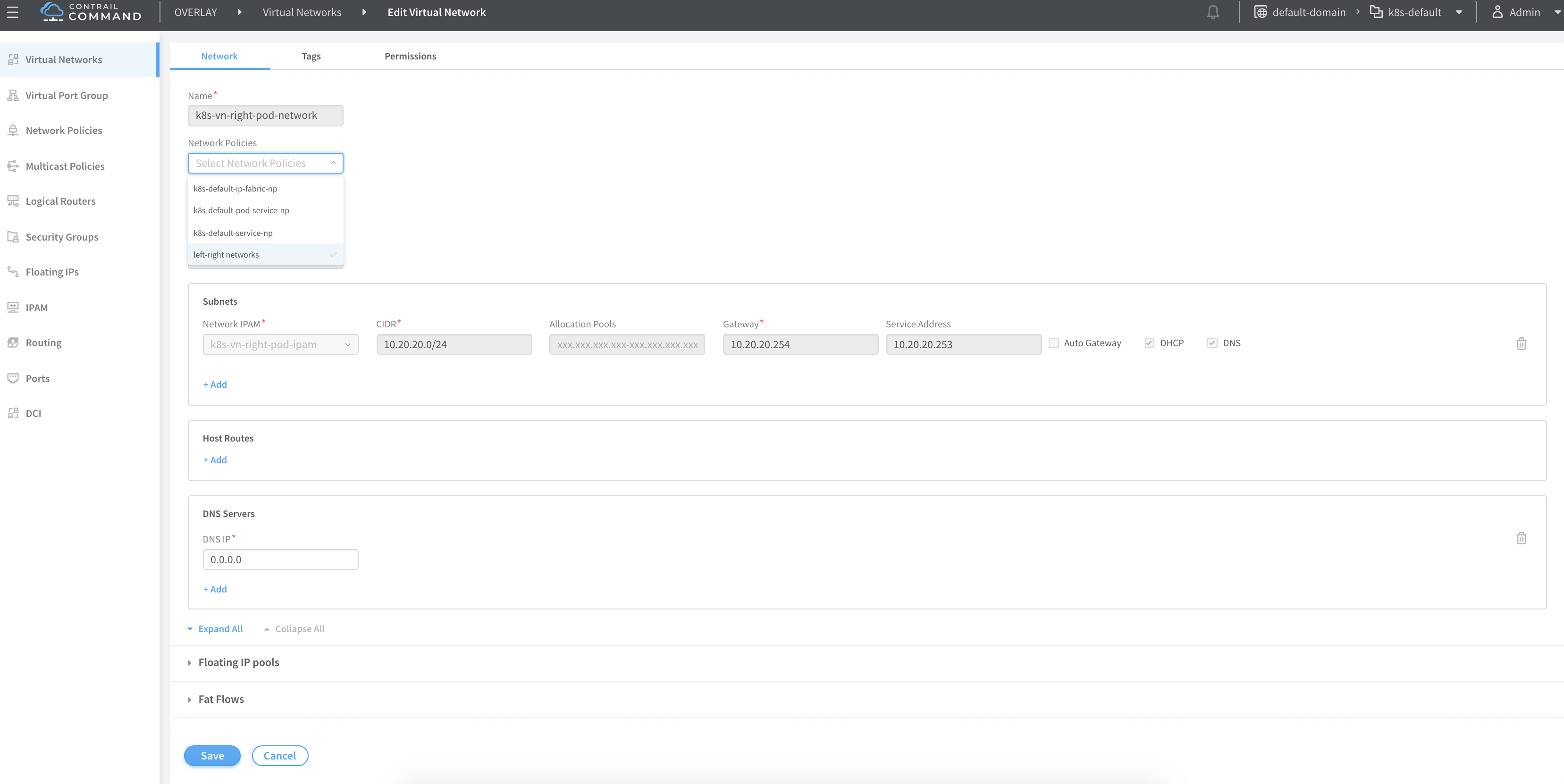
Select advanced option to attached the service instance you create before and click create



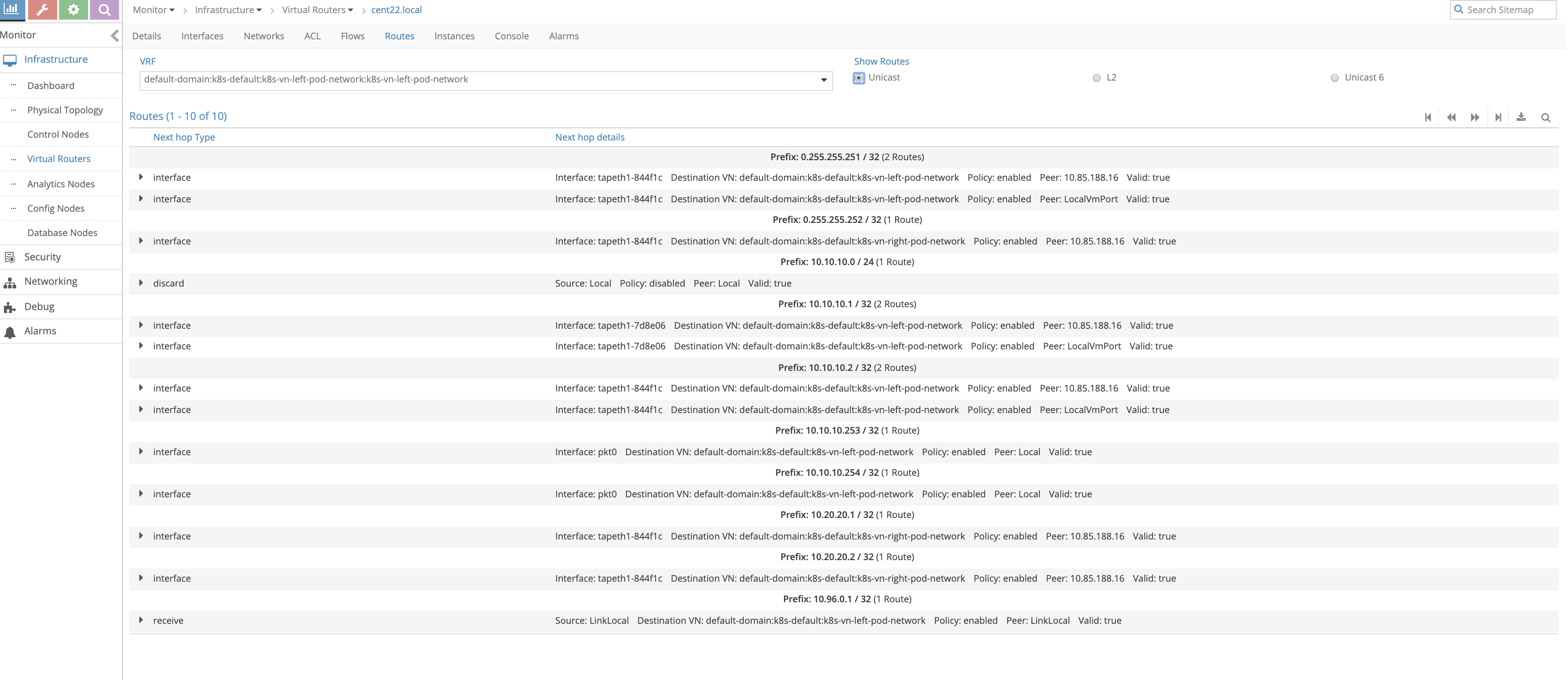
To attach this network policy to network click virtual network and select the left network and edit



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



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 oursetup), 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 the right networks host routes has 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

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-http-s trust untrust 5434 N/A(N/A) ge-0/0/1.0 UNKNOWN UNKNOWN UNKNOWN N/A N/A -1  
Jun 14 23:07:35 csrx1-sc flowd-0x2[374]: RT\_FLOW: RT\_FLOW\_SESSION\_CLOSE: session closed TCP FIN: 10.10.10.1/47190->10.20.20.1/80 0x0 junos-http 10.10.10.1/47190->10.20.20.1/80 0x0 N/A N/A N/A N/A 6 only-http-s trust untrust 5434 14(940) 12(12452) 2 UNKNOWN UNKNOWN N/A(N/A) ge-0/0/1.0 UNKNOWN N/A N/A -1