POLITECNICO DI TORINO

Department of Electronics and Telecommunication (DET)
Master Degree Program in Engineering
Communication and Computer Networks
Master Degree Thesis

Network Automation, Orchestration, Cloud: Tools and Technologies

Supervisor

Prof. RISSO FULVIO GIOVANNI OTTAVIO

Candidate

Alireza Kolahdouzan

December 2019

Dedication

this study is wholeheartedly dedicated to my beloved father and mother…

Acknowledgments

I would like to express my sincere gratitude to my supervisor Prof. RISSO FULVIO GIOVANNI OTTAVIO for giving me the opportunity to do my thesis under his supervision and providing invaluable guidance throughout this thesis. It has a great privilege and honor to work and study under his guidance. I am extremely grateful for what he has offered me. I would also like to thank him for his friendship, empathy, and enthusiastic encouragement.

I am particularly grateful for the assistance given by dott. Alex Palesandro for his valuable and patient guidance.

Contents

List of figures

Chapter 1

Introduction

Nowadays, with the increasing growth of applications and data volume, the importance of lightweight cloud infrastructure for microservices and optimized usage of physical resources through virtualization and cloud operating systems are inevitable. Basically, a cloud system uses virtual machine technology which is used to virtualized physical resources, provide independent computing resources and guest OS, to enable flexible operations of virtualized network function servers and data processing [1]. However, the guest OS contains all the kernel modules and user libraries of hardware needed to run the operating system, so the host's CPU, a memory I/O and resource requirements are high. On the other hand, container-based technologies only share the host's kernel module and it only requires user's library and utility programs, and the resources to run the program on the operating system used in user space such as Guest OS. [2]. Based on the above advantages that virtualization and containerization technologies provisioned, cloud providers offer many useful tools for providing the infrastructure for deploying applications. Nonetheless, there are many different cloud solutions such as (e.g. OpenStack, AWS, or GCP) which would be difficult to prevent vendor-specified for avoiding devotes specified resources.

For solving this problem, Kubernetes is providing a mature set of well-standardized principles and practice for running software in a distributed virtualized environment. Moreover, it supports container-based deployment within Platform-as-a-Service, concentrating on the cluster-based system. There is always a challenge to control an increase in the demand for scaling and auto-healing of the network and virtual instances. Kubernetes put containers into Pods and deploying pods across different servers which based on workload of application can dynamically scale them out. It also supports multiple Docker containers, which are able to make use of services related to a Pod [3]. Furthermore, many development teams want to use the Kubernetes, but some application are not capable to take advantage of containerized. So, KubeVirt technology provides a unified development platform where developers can build, modify, and deploy applications which coexist in both application containers and virtual machine in a common, shared environment.

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter. OpenStack provides a virtualized environment on physical machines with goals of eliminating vendor-lock in a production environment.

1.1. Problem statement

With the advent of cloud computing to the world of computers, demanding cloud computing services which offer is growing very fast, its performance must be good enough to convince the users' needs whether it is private or public cloud. One of the most vital concerns of cloud computing is to attain a good network performance because without it, it is impossible to provide high-performance cloud computing services.

This thesis concentrates on OpenStack cloud computing that provides an infrastructure as a service, also KubeVirt project that presents virtual machines on Kubernetes orchestration for leveraging fully isolated virtual machine on the containerized environment which managed by Kubernetes.

The problem statements of this investigation are:

1. To find out the Network traffic flow in the OpenStack and KubeVirt platforms.
2. To measure network performance in cloud computing based on OpenStack and KubeVirt.
3. Finding solutions for improving the network performance of the existing OpenStack and KubeVirt.

It is believed that investigating these subjects will lead to predicting the behavior of network traffic on OpenStack and KubeVirt which users can recognize what network performance they will be obtained, and cloud service providers can better tune their infrastructure to offer services with higher performance.

1.2. Objectives and layout of the thesis

This study is categorized as follows:

In chapter two, The Kubernetes general architecture and its network architecture also are explained. Chapter three presents the KubeVirt project that contains network architecture and its capabilities are discussed. In chapter four, the OpenStack architecture and more concentrating on analyzing network neutron architecture with goals of understanding what is happening under the hood. Chapter five is dedicated to traffic flows investigation both for OpenStack and KubeVirt. In Chapter six, it is analyzed the obtained results of network performance tests. In the end, the conclusion is represented in chapter seven.

Chapter 2

Kubernetes

Kubernetes is an open-source orchestration tool developed by Google for conducting microservices or containerized applications [3]. It obscures the hardware infrastructure and indicating the whole datacenter as a single huge computational resource. Moreover, it enables developers to deploy and execute any type of applications without being aware of underlay servers. It is turning into the typical model for running distributed applications both in the cloud and on-premises infrastructures, so It simplifies development, preparation, and management for both developers and operations teams.

2.1. Kubernetes architecture

Kubernetes cluster consists of many nodes, but It divides to two main nodes, the master node, and worker node.

The control plane is run by a Master node which controls and manages the worker nodes. The Master node orchestrating the worker nodes in which the application should run on them. Moreover, the master node contains several components that store the state of the cluster. Kubernetes cluster divide into two main sections:

- The Kubernetes control plane
- The worker nodes

The main components that consist of the control plane are:

- **The API server** is a gateway in the Kubernetes cluster which all the components in Kubernetes just communicate with API server, and they interact with each other indirectly. The API server is the only element can talk with etcd directly. Furthermore, provisioning a CRUD (Create, Read, Update, Delete) interface for querying and changing the cluster state by API server is performed through RESTful API over HTTP, then the cluster state and all API objects would be stored in persistent storage backend(etcd).
  The clients use the kubectl command for sending requests to API server, it replies the requests by push and pulls the object information from etcd. Also, it gathers the logs from pods.

- **etcd storage** is a backend, persistent, distributed key-value pair database. All the RESTful API object stored in etcd. It stores configuration and replicating information, which is used

by both master and worker nodes. It is possible to run more than one etcd instance for supplying high availability and improving performance which should usually deployed with an odd number of instances.

- **Scheduler** employs for scheduling the pods to worker node clusters based on resource utilization. It assigns a pod to nodes according to service requirements. The scheduler should be informed about the total available resources and the devoted resources to the current workload on every node.
- **Controller-manager** is responsible for running, monitoring, and managing different controllers that determine behavior in the cluster. Controllers watch the API server resource changes and apply suitable operations for each change. In addition, controllers also perform a re-list occasionally to be sure have not missed an event. Also, controllers are not able to communicate with each other directly and they do not know about the existence of other controllers.

The main elements that exist in each worker node are:

- **Kubelet** monitors the pod specifications through the master and makes sure that allocated pods are running on the node. It gets the pod definition and uses it for creating containers through a container engine which is the most vital part of the worker node for running the containers. Moreover, the only component that runs as a regular system is Kubelet and it runs all the other components as a pod. Also, Kublet deployed on the master node. It informs the API server about node activities such as pod health, node health, and liveness probe.
- **Kube-proxy** is the most critical component which providing Pod-to-Service and External-to-Service networking which it runs on both master and worker nodes. It can forward TCP and UDP packets. Also, it gains the cluster IP through environment variables or DNS. There are three proxy modes, Userspace, Iptables, and IPVS. Userspace mode makes a considerable overhead due to switching between user space and kernel space. On the other hand, iptables mode utilizing iptabale NAT in Linux for routing TCP and UDP packet throughout the containers. Moreover, the IPVS (IP Virtual Server) mode is placed on top of the Netfilter and implement transport load balancing as part of the Linux Kernel [4].

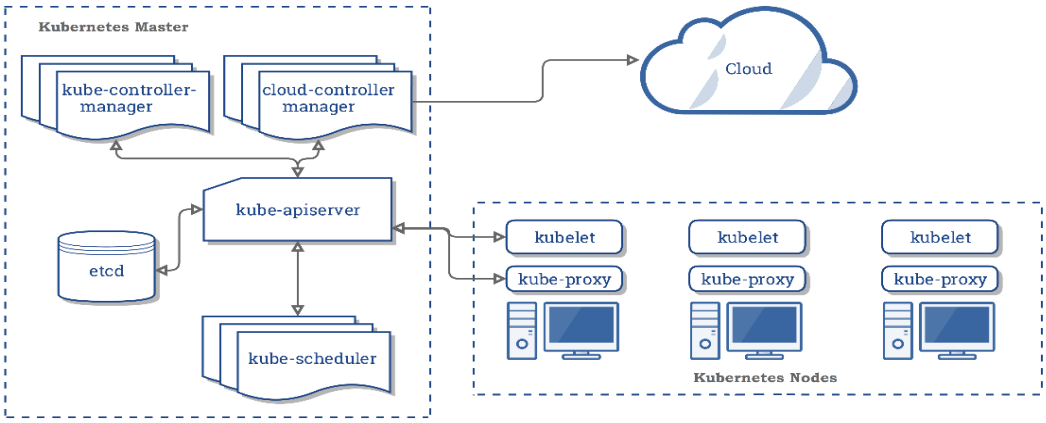

Figure 2.1: Kubernetes Architecture [5]

2.2. Kubernetes network architecture

Every pod has the unique IP address and it can communicate with other pods on all nodes through flat, NAT-less network, not by Kubernetes itself. The Pod consists of one or more containers that are collocated on the same host and are configured to share a network stack and other resources such as volumes. Share network stack means that all the container in pod can reach each other in the localhost [6]. The network is made by the system administrator or by a Container Network Interface (CNI) plugin. Before creating a container, the pause container would be created. The pause container is a container that keeps all the containers of a pod together. Furthermore, it is an infrastructure container whose main purpose is to hold all Linux namespaces which are shared by all containers of a pod.

2.2.1. Pod communication within the same node

Before the infrastructure container is started, a virtual Ethernet interface pair (a veth pair) is created for the container. One interface of the veth pair stays in the host's namespace (it tagged with vethxxx) while the other interface is moved into the container's network namespace and renamed to eth0. These two virtual interfaces are like two ends of a pipe that everything goes in one side, comes out on the other. The interface in the host's network namespace is attached to a network bridge that container runtime is configured to use. The eth0 interface in the container is assigned an IP address from the bridge's address range. Anything that application running inside the container sends to the eth0 network interface and comes out at the other veth Interface in host's namespace and is sent to bridge. So, any network connected to the bridge can receive it. All Pods on a node are connected to the same bridge, which they can communicate with each other.

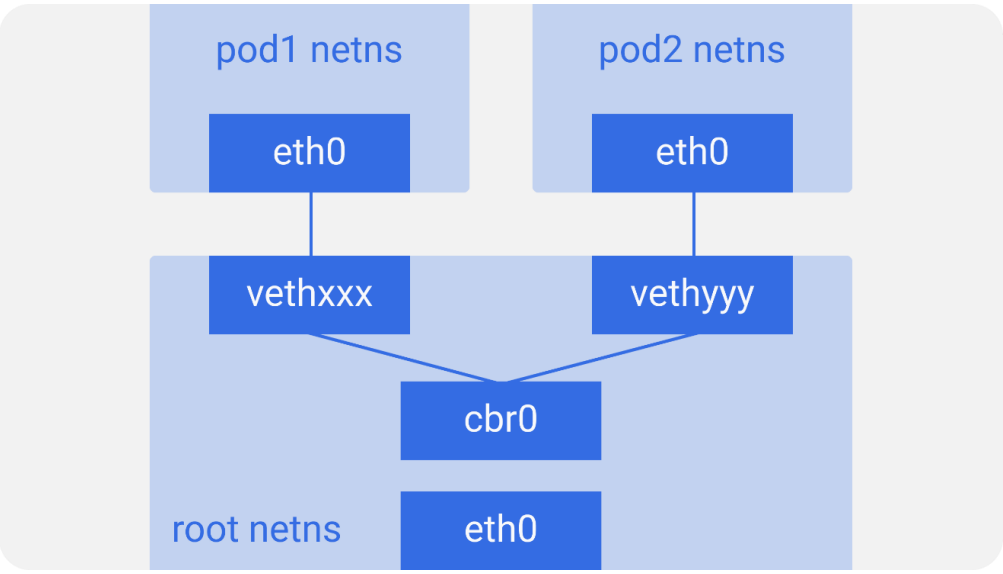

Figure 2.2: Pod communication within the same node [7]

2.2.2. Pod communication on different nodes

Pod IP addresses must be unique across the whole cluster, so the bridges across the nodes must use non-overlapping address ranges to prevent pods from different nodes from getting the same IP address. There are many methods for connecting the bridges on different nodes. This can be done with overlay or underlay networks or by regular layer 3 routing [8].

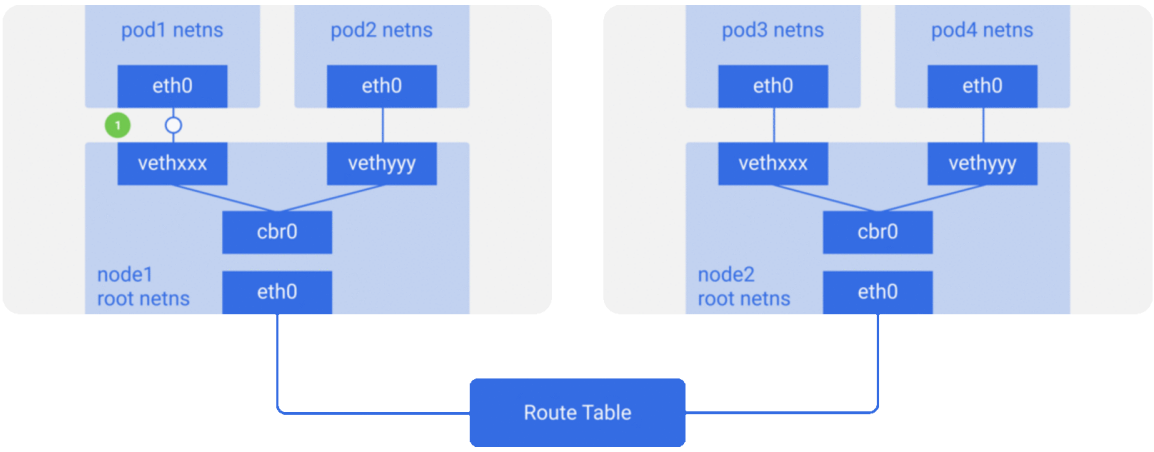

Figure 2.2: Pod communication on different nodes [7]

2.2.3. CNI (Container Network Interface)

It is a standard designed to make it easy to configure container networking when containers are created or destroyed. Kubernetes uses the CNI specifications and plug-ins to orchestrate networking. Also, it can address other container's IP addresses without using the Network Address Translation (NAT). Every time a Pod is initialized or removed, the default CNI plug-in is called with the default configuration, which this CNI plug-in creates a pseudo interface, attaches it to the underlay network, sets IP Address, routes, and maps it to the Pod namespace. It should be passed --network-plugin = cni to the Kubelete when launching it for using the CNI plugin. If the environment is not using the default configuration directory (/etc/cni.net.d), the CNI plugin passes the correct configuration directory as a value to --cni-conf-dir. Moreover, the Kubeletet looks for the CNI plugin binary at /opt/cni/bin, but it can be specified an alternative location with --cni-bin-dir.

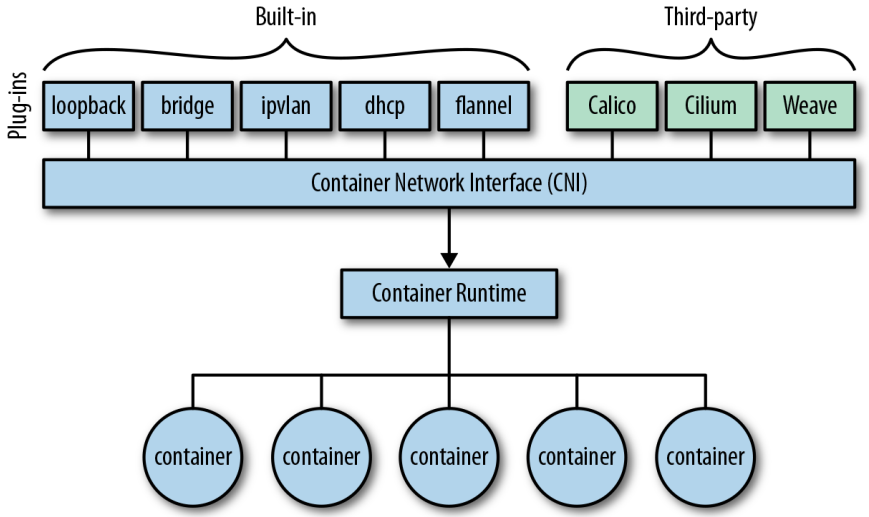

Figure 2.3: CNI [9]

2.2.4. Pod-to-Service networking

Pod IP addresses are not durable and will appear and disappear in response to scaling up or down, application crashes, or node reboots. Each of these events can make the Pod IP address change without warning. Services were built into Kubernetes to address this problem.

The Kubernetes service manages the state of Pods, allowing us to track a set of the pod IP address that dynamically changes over time. Services act as an abstraction over Pods and assign a single virtual IP address to a group of Pod IP addresses. Any traffic addressed to the virtual IP of the service will be routed to the set of Pods that are associated with the virtual IP. This allows the set of Pods associated with a service to change at any time clients only need to know the service's virtual IP, which does not change [10]

2.3. Exposing services to external clients

There are few ways to make a service accessible externally:

2.3.1. NodePort

Each cluster node opens a port on the node itself and redirects traffic received on that port to underlying service. The service is not accessible only at the internal IP cluster and port, but also through a dedicated port on all nodes. Exposing a set of pods to external clients is accomplished by creating a service and settings its type to NodePort. By creating a NodePort service, Kubernetes reserves a port on all its nodes and forwarding incoming connections to the Pods that are part of service.

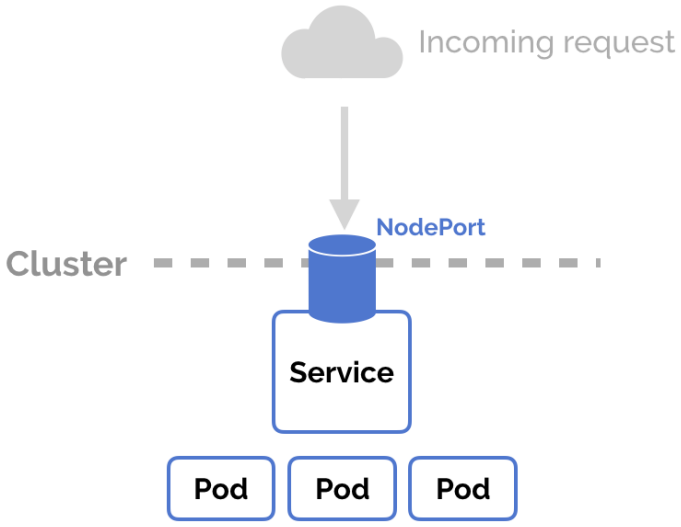

Figure 2.4:NodePort [11]

2.3.2. LoadBalancer

The load balancer will have its own unique, publicly accessible IP address and will redirect all connections to your service. The service is accessible through load balancer's IP address. Kubernetes clusters running on cloud providers usually support the automatic provision of a load balancer form the cloud infrastructure. It should be set the service type to LoadBalancer. One of the important advantages of putting a load balancer in front of the nodes is that making sure to spread the requests across all healthy nodes and never sending them to a node that is offline now.

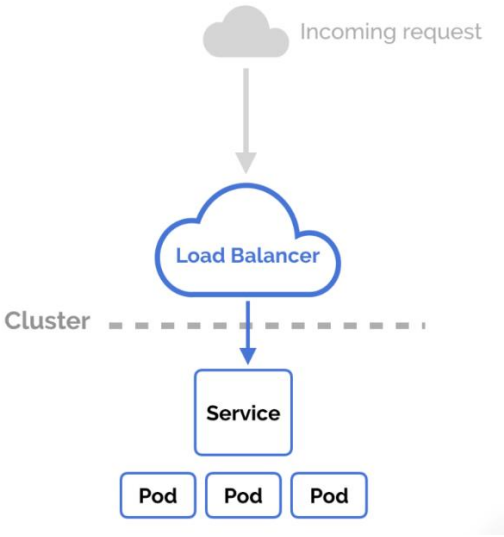

Figure 2.5: LoadBalancer [11]

2.3.3. MetalLB Load Balancer for On-premise /BareMetal

Kubernetes does not offer an implementation of network load balancers for bare metal clusters. If the Kubernetes runs on the bare metal, LoadBalancer will remain in the "pending" state indefinitely when created. MetalLB [12] goals are to compensate for the lack of load balancer that the external services can work well on bare metal clusters, Based on standard routing. The metalLB load balancer handles assigning IP addresses from the pools of IP addresses that it can use. It will take care of allocating or deallocating individual addresses as service come and go. But It only considers IPs that devoted to its configured pools. MetalLB uses standard routing protocols to aware of the IP lives in the cluster. It supports to routing modes:

- **Layer 2 mode:**
  In this mode, one node assumes the responsibility of advertising a service to the local network. So, all the traffic for a service IP goes to one node. From there, Kube-proxy distributes the traffic to all the service's pods. Accordingly, layer 2 mode does not implement a load balancer, on the contrary, it applies a failover mechanism so that a different node can dominate as the node leader. If the leader node fails for some reason, failover is done automatically. Furthermore, it has two main limitations which are single node bottlenecking and potentially slow failover.

- **BGP mode:**
  Each node in the cluster sets up a BGP peering session with network routers and uses that peering to advertise the IPs of external cluster services. Assuming the routers are configured to support multipath, this enables true load-balancing: the routes published by MetalLB are equivalent to each other except for their next hop. This means that the routers

will use all next hopes together and load balance between them. Once the packets arrive at the node, Kube-proxy is responsible for the final hop of traffic routing to get the packets to one specific pod in the service.

The MetalLB deploys to the cluster under the metallb-system namespace. It has two manifest components:

- **Controller:**
  This is the cluster-wide controller that handles IP address assignments.
- **Speaker:**
  This component speaks the protocol of our choice to make service reachable.

2.4. Exposing services externally through Ingress resources

Ingresses run at the application layer of the network stack (HTTP) and can supply features such as cookie-based session affinity which service cannot. One major difference of Ingress with load balancer is that each LoadBalancer service requires its own load balancer with its own public IP address, whereas an Ingress requires one public IP address, even when providing access to dozens of services. It is necessary to be running the Ingress controller for making Ingress resources work. The client connects to Pods through the Ingress controller, first performed a DNS lookup of given domain name and DNS server returned the IP of the Ingress controller. The client then sent HTTP request to the Ingress controller and specified given domain name in the Host header. From that header, the controller determined which service the client is trying to access, looked up the Pod IPs through the Endpoints object associated with the service, and sent the client's request to one of the pods.

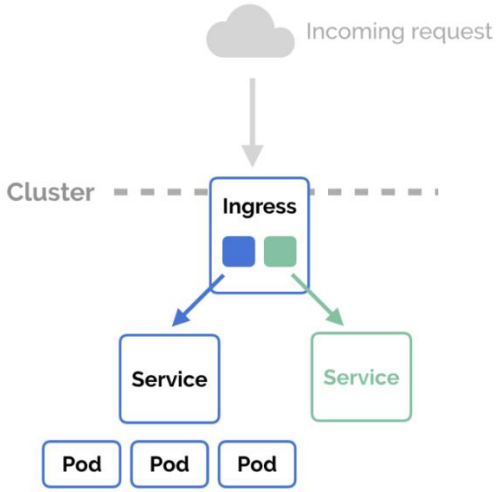

Figure 2.6: Ingress [11]

Chapter 3
KubeVirt

There are many old applications that are in developing are slowly migrating or not moved to Kubernetes yet, and they are still running in old infrastructures like Virtual Machines or Bare metals. KubeVirt [13] technology provides a unified development platform where developers can build, modify, and deploy applications residing in both Application containers as well as virtual machines in a common, shared environment. It empowers teams with a reliance on existing virtual machine-base workload to rapidly containerize applications. KubeVirt extends Kubernetes by adding resource type for VMs and sets of VMs through Kubernetes's Custom Resource Definitions API(CRD). KubeVirt VMs run within regular Kubernetes Pods, where they have access to standard Pod networking and storage and can be managed using standard Kubernetes tools such as kubectl. It requires an additional binary is provided to get quick access to the serial and graphical ports of a VM, and handle start/stop operations. The tool is called *virtctl*. CRD introduces two resources VirtalMachine and VirtualMachineInstance to Kubernetes, which define properties of VM such as Machine, CPU type, RAM size, CPU count etc.

3.1. KubeVirt Components

- **Virt-api**
  Serves as an entry point for all virtualization related flows and take care to update the virtualization related Custom Resource Definition (CRD). It also acts as the main entry point to kubeVirt and it is responsible for validating all VM and VMI resources type for each request coming user through api-server.
- **Virt- controller**
  It is a Kubernetes Operator that is responsible for cluster-wide virtualization functionality. When new VM objects are posted to the Kubernetes API server, the virt-controller takes notice and creates the pod in which the VM will run. When the Pod is scheduled on a particular node, the virt-controller updates the VM object with the node name and hands-off further responsibilities to a node-specific KubeVirt component, the virt-handler, an instance of which runs on every node in the cluster
- **Virt-handler**
  The virt-handler is watching for changes to the VM object and performing all necessary operations to change a VM to meet the required state. The virt-handler references the VM specification and signals the creation of a corresponding domain using a libvirtd instance in the VM's Pod.

- **Virt-launcher**
  For every VM object, one Pod is created. This Pod's container runs the virt-launcher KubeVirt component. The main purpose of the virt-launcher Pod is to provide the cgroups and namespaces which will be used to host the VM process. Virt-handler signals virt-launcher to start a VM by passing the VM's CRD object to virt-launcher. Virt-launcher then uses a local libvirtd instance within its container to start the VM. From there virt-launcher monitors the VM process and terminates once the VM has exited.
- **Libvirtd**
  An instance of libvirtd is presented in every VM Pod. Virt-launcher uses libvirtd to manage the life-cycle of the VM process.

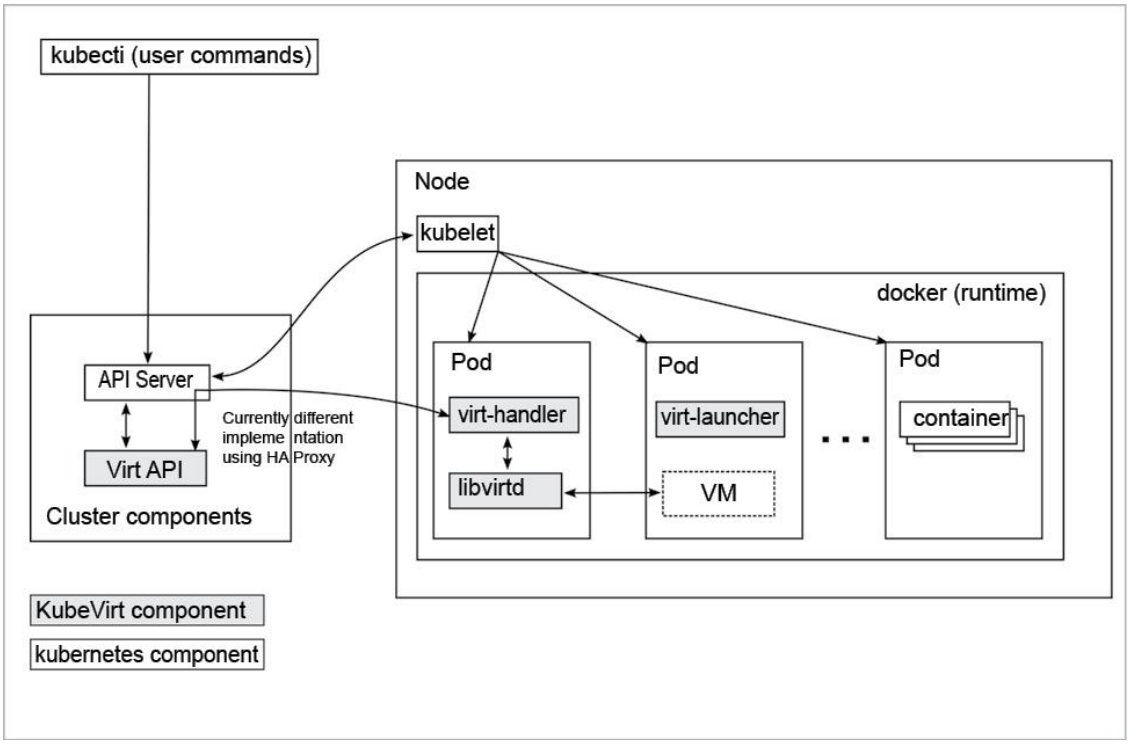

Figure 3. 1: KubeVirt Components [13]

3.2. KubeVirt Networking

3.2.1. Virt-launcher - virtwrap

virt-launcher [14] is the pod that runs the necessary components instantiate and run a virtual machine.

- **Virtwrap manager**
  Before the virtual machine is started the preStartHook will run SetupPodNetwork
- **SetupPodNetwork**
  This function calls three functions that are detailed below:
  - **discoverPodNetworkInterface**
    this function gathers the following information about Pod interface:
    - IP address
    - Routes
    - Gateway
    - MAC address

    This is stored for later use in configuring DHCP

  - **preparePodNetworkInterfaces**
    Once the current details of the Pod interface have been stored, following operations are performed:
    - Delete the IP address from the Pod interface
    - Set the Pod interface down
    - Change the Pod interface MAC address
    - Set the Pod interface up
    - Create the bridge
    - Add the Pod interface to the bridge

    This will provide libvirt a bridge to use for the virtual machine that will be created.

  - **SingleClinetDHCPServer**
    This DHCP server only provides a single address to a client in this case the virtual machine that will be started. The network details such as the IP address, gateway, routes, DNS servers, and suffixes are taken form the Pod which will be served to the virtual machine.

3.3. Networking in detail

Now that we have a clearer picture of KubeVirt networking [14], in this part, it would be explained with details regarding Kubernetes objects, host, Pod and virtual machine networking components.

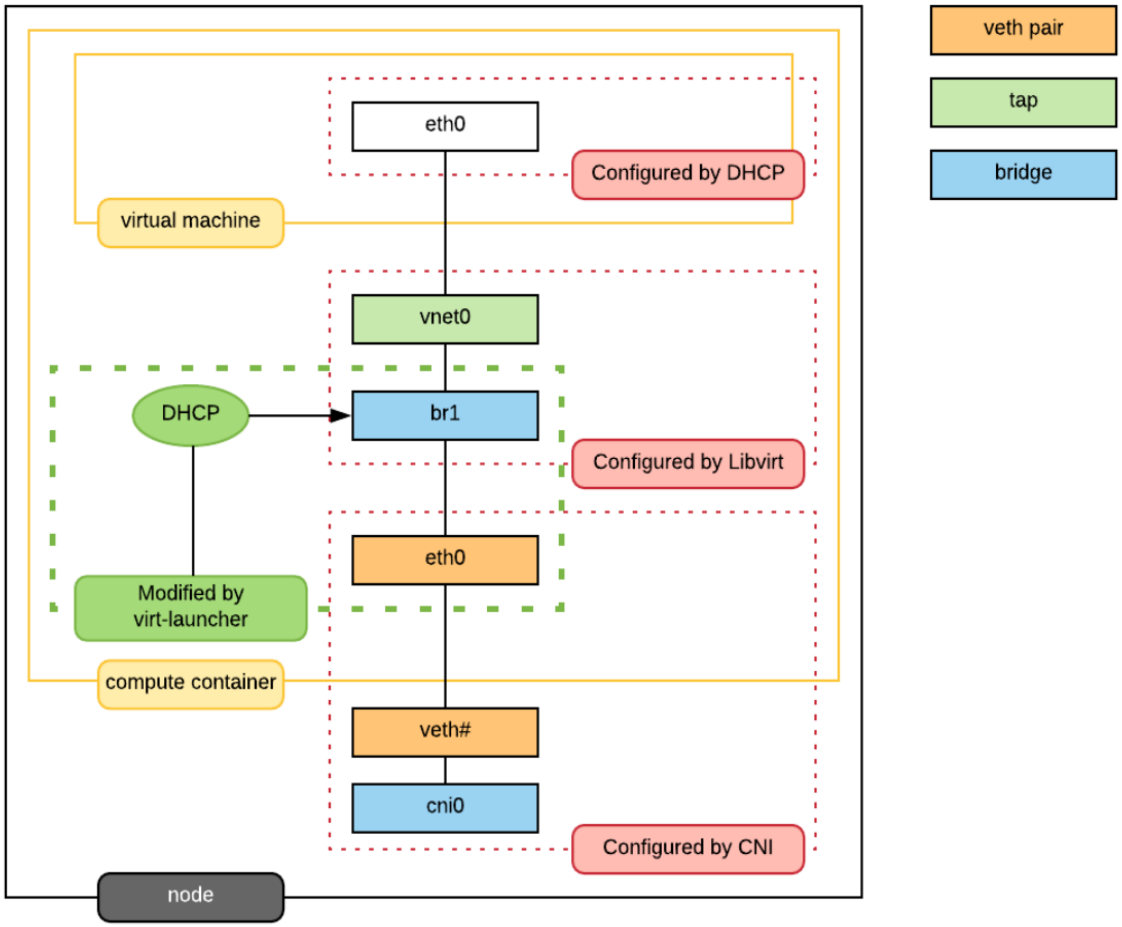

Figure 3. 2: KubeVirt Networking architecture [14]

3.3.1. Kubernetes-level

Services

once the VirtualMachineInstance is started, in order to connect to a VirtualMachineInstance, it can be created a Service object for a VirtualMachineInstance. Three kinds of services are supported: ClusterIP, NodePort, and LoadBalancer. The default type is ClusterIP.

In our production, we use metalLB load balancer and the vmiservice has been created for exposing VirtualMachineInstances as a service with type of LoadBalancer. The vmiservice has been used for connecting the VirtualMachineInstances to the external world. For reaching this goal, Labels

on a VirtualMachineInstance are passed through to the Pod, so it should be added a selector special: key for service creation to VirtualMachineInstance. From there on it works like exposing any other Kubernetes resource, by referencing these labels in a service.

```
  selector:
    special: key
```

```
cube3@master-node:~$ kubectl get svc
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
jenkins      ClusterIP      10.98.42.218    <none>           80/TCP,443/TCP   24d
kubernetes   ClusterIP      10.96.0.1       <none>           443/TCP          38d
nginx2       NodePort       10.107.104.66   <none>           8080:31760/TCP   34d
vmiservice   LoadBalancer   10.104.27.11    130.192.225.71   22:31756/TCP     34d
```

Figure 3. 3: LoadBalancer for virtual machines

Endpoints

The following endpoints were automatically created because in this case, there was a label in VirtualMachineInstances which points to vmiservice.

```
  labels:
    kubevirt.io: virt-launcher
    kubevirt.io/domain: vm1
    special: key
```

```
kubectl get pods -o wide
virt-launcher-vm1-jsgp9   1/1   Running   0   4d22h   10.244.3.223   spring     <none>   <none>
virt-launcher-vm2-tzmvg   1/1   Running   0   3d21h   10.244.3.244   spring     <none>   <none>
virt-launcher-vm3-d2c7d   1/1   Running   0   23d     10.244.2.122   worker02   <none>   <none>
```

```
kubectl get endpoints
vmiservice   10.244.2.122:22,10.244.3.223:22,10.244.3.244:22   34d
```

Figure 3. 4: Virtual machine Endpoints connecting to vmiservice LoadBalancer

3.3.2. Host-level

Interfaces

If the list of interfaces from the host is taken , it could be observed that few important interfaces. The flannel.1 interface is type VXLAN for connectivity between hosts. cni0 is a bridge where one side of the veth interface pair is attached.

```
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:5f:61:fa:d1 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
5: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether 02:07:d2:92:18:b8 brd ff:ff:ff:ff:ff:ff
    inet 10.244.0.0/32 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::7:d2ff:fe92:18b8/64 scope link
       valid_lft forever preferred_lft forever
6: cni0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default qlen 1000
    link/ether b2:2d:87:54:7e:77 brd ff:ff:ff:ff:ff:ff
    inet 10.244.0.1/24 scope global cni0
       valid_lft forever preferred_lft forever
    inet6 fe80::b02d:87ff:fe54:7e77/64 scope link
       valid_lft forever preferred_lft forever
```

```
7: vethf2ead0c6 state UP @enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 master cni0 state forwarding priority 32 cost 2
8: veth20b0b740 state UP @enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 master cni0 state forwarding priority 32 cost 2
```

Figure 3. 5: Host level interfaces

Routes

the Pod network subnet is 10.244.0.0/16 and divided per hosts:

- Master node: 10.244.0.0/24
- Node-1: 10.244.1.0/24
- Node-2: 10.244.2.0/24

So, the IP table below shows the packets to correct interface:

```
vagrant@node-1:~$ ip r
default via 192.168.0.254 dev enp0s8 proto dhcp src 192.168.0.115 metric 100
10.244.0.0/24 via 10.244.0.0 dev flannel.1 onlink
10.244.1.0/24 via 10.244.1.0 dev flannel.1 onlink
10.244.2.0/24 dev cni0 proto kernel scope link src 10.244.2.1
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
192.168.0.0/24 dev enp0s8 proto kernel scope link src 192.168.0.16
192.168.0.0/23 dev enp0s8 proto kernel scope link src 192.168.0.115
192.168.0.254 dev enp0s8 proto dhcp scope link src 192.168.0.115 metric 100
```

Figure 3. 6: Host level routs

Iptables

Kube-proxy writes iptables rules for supporting deployed services. In the output below, it could be observed vmiservice with destination NAT rules defined.

```
root@master-node:~# iptables -n -L -t nat | grep vmiservice
KUBE-MARK-MASQ  all  --  0.0.0.0/0            0.0.0.0/0            /* default/vmiservice: loadbalancer IP */
KUBE-SVC-U2GQAR37FZ3RWKWV  all  --  0.0.0.0/0            0.0.0.0/0            /* default/vmiservice: loadbalancer IP */
KUBE-MARK-DROP  all  --  0.0.0.0/0            0.0.0.0/0            /* default/vmiservice: loadbalancer IP */
KUBE-MARK-MASQ  tcp  --  0.0.0.0/0            0.0.0.0/0            /* default/vmiservice: */ tcp dpt:31756
KUBE-SVC-U2GQAR37FZ3RWKWV  tcp  --  0.0.0.0/0            0.0.0.0/0            /* default/vmiservice: */ tcp dpt:31756
KUBE-MARK-MASQ  tcp  -- !10.244.0.0/16        10.104.27.11         /* default/vmiservice: cluster IP */ tcp dpt:22
KUBE-SVC-U2GQAR37FZ3RWKWV  tcp  --  0.0.0.0/0            10.104.27.11         /* default/vmiservice: cluster IP */ tcp dpt:22
KUBE-FW-U2GQAR37FZ3RWKWV  tcp  --  0.0.0.0/0            130.192.225.71       /* default/vmiservice: loadbalancer IP */ tcp dpt:22
```

Figure 3. 7: Iptables rules

3.3.3. Pod-level

Interfaces

The bridge br1 is the main focus in the pod level. It contains the eth0 and vnet0 ports. eth0 becomes the uplink to the bridge which is the other side of the veth pair which is a port on the host's cni0 bridge. Since eth0 has no IP address and br1 is in the self-assigned range, the Pod has no network access. This can be resolved for troubleshooting by creating a veth pair, adding one of the interfaces to the bridge and assigning an IP address in the pod subnet for the host. Routes are also required to be added.

```
[root@vm1 /]# 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
3: eth0@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br1 state UP group default
    link/ether d6:b4:f9:01:82:cb brd ff:ff:ff:ff:ff:ff link-netnsid 0
4: br1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether d6:b4:f9:01:82:cb brd ff:ff:ff:ff:ff:ff
    inet 169.254.75.10/32 brd 169.254.75.10 scope global br1
       valid_lft forever preferred_lft forever
5: vnet0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc fq_codel master br1 state UNKNOWN group default qlen 1000
    link/ether fe:b4:f9:6b:6d:3b brd ff:ff:ff:ff:ff:ff
```

Figure 3. 8: Pod-level interface

DHCP

The virtual machine network is configured by DHCP. It could be seen virt-launcher has UDP port 67 open on the br1 interface to serve DHCP to the virtual machine.

Libvirt

with virsh domiflist we can also see that the vnet0 interface is a port on the br1 (k6-eth0) bridge.

```
vagrant@k8s-master:~$ kubectl exec -ti virt-launcher-vm1-kclsm -- virsh domiflist default_vm1
 Interface   Type      Source     Model    MAC
-------------------------------------------------------------
 vnet0       bridge    k6t-eth0   virtio   d6:b4:f9:6b:6d:3b
```

Figure 3. 9: Networking statistics for pod level virtual machine

3.3.4. VM-level

Interface

The VM interface is typical. Just the single interface has been assigned to the original Pod IP address. Also, it could be detected that the MTU of the virtual machine interface is set to 1500. The network interfaces of virtual launcher pods are set to 1450.

```
ubuntu@vm2:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 36:d8:7a:b2:52:0b brd ff:ff:ff:ff:ff:ff
    inet 10.244.3.244/32 brd 10.244.3.244 scope global enp1s0
       valid_lft forever preferred_lft forever
    inet6 fe80::34d8:7aff:feb2:520b/64 scope link
       valid_lft forever preferred_lft forever
```

Figure 3. 10: VM-level interfaces

DNS

If the path of /etc/resolv.conf be cat, it could be noticed that the DNS is configured, So the Kube-dns will be properly queried.

```
ubuntu@vm2:~$ cat /etc/resolv.conf
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)
#     DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN
nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local ipv6.polito.it
```

Figure 3. 11: VM-level DNS

3.4. CDI (Containerized Data Importer)

It is a persistent storage management add-on for Kubernetes. The purpose of it to provide a declarative way to build Virtual Machine Disks on PVCs (Persistent Volume claim) for Kubevirt VMs. The data can come from various sources: URL, container registry, another PVC (clone), or an upload from a client. CDI [15] works with standard core Kubernetes resources and is a storage device compatible, while its primary focus is to build a disk image for Kubevirt. It is also useful outside of a kubevirt context to use for initializing the Kubernetes Volumes with data.

3.4.1. CDI DataVolumes

CDI includes a CRD that provides an object of type DataVolume. The DataVolume is an abstraction on top of the standard Kubernetes PVC and can be used to automate the creation and population of a PVC with data. Although it can be used PVCs directly with CDI, DataVolumes are the preferred method since they offer full functionality, a stable API, and better integration with Kubevirt.

3.4.2. Import from URL

This method is selected when a DataVolume with an Http source is created. CDI will populate the volume using a Pod that will download from the given URL and handle the content according to the contentType setting. It is also possible to configure basic authentication using a secret and specifying custom TLS certificates in a ConfigMap. The PVC is created, when a cluster user needs to persistent storage in one of their Pods.

First, The PVC manifest is created which specifying the minimum size and the access mode that is required. Then, a user submits the PVC to the Kubernetes API server and Kubernetes finds the appropriate persistent volumes and binds the volume to the claim. Afterward, the PVC can be used as one of the volumes inside a Pod. A custom controller watches for importer specific claims, and when discovered, starts an import process to create a new raw image named disk.img with the desired content into the associated PVC. For instance, the Ubuntu cloud image as a PVC will be imported and launched a Virtual Machine making use of it. This will create PVC with a proper annotation; therefore, CDI controller detects it and launches an importer Pod to gather the image specified in the cdi.kubevirt.io/storage.import.endpoint annotation

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: "ubuntu"
  labels:
    app: containerized-data-importer
  annotations:
    cdi.kubevirt.io/storage.import.endpoint: "https://cloud-images.ubuntu.com/xenial/20191031/xenial-server-cloudimg-amd64-disk1.img"
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 4Gi
  storageClassName: hostpath
```

```
vagrant@k8s-master:~$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
importer-ubuntu2-xkfkl        1/1     Running   0          81s
virt-launcher-vm1-kclsm       1/1     Running   0          20h
```

```
vagrant@k8s-master:~$ kubectl logs -f importer-ubuntu2-xkfkl
I1105 10:16:12.636206       1 importer.go:51] Starting importer
I1105 10:16:12.656407       1 importer.go:107] begin import process
I1105 10:16:13.005184       1 data-processor.go:252] Calculating available size
I1105 10:16:13.136945       1 data-processor.go:260] Checking out file system volume size.
I1105 10:16:13.137209       1 data-processor.go:264] Request image size not empty.
I1105 10:16:13.137343       1 data-processor.go:269] Target size 4Gi.
I1105 10:16:13.137668       1 data-processor.go:182] New phase: Convert
I1105 10:16:13.137795       1 data-processor.go:188] Validating image
I1105 10:16:14.799000       1 qemu.go:212] 0.00
I1105 10:16:18.128243       1 qemu.go:212] 1.00
I1105 10:16:18.293121       1 qemu.go:212] 2.01
I1105 10:16:18.502644       1 qemu.go:212] 3.01
I1105 10:16:18.624973       1 qemu.go:212] 4.02
I1105 10:16:18.723253       1 qemu.go:212] 5.02
I1105 10:16:19.465622       1 qemu.go:212] 6.03
I1105 10:16:22.415236       1 qemu.go:212] 7.03
I1105 10:16:26.420147       1 qemu.go:212] 8.04
I1105 10:16:30.281458       1 qemu.go:212] 9.04
I1105 10:16:31.638822       1 qemu.go:212] 10.05
I1105 10:16:34.345999       1 qemu.go:212] 11.05
I1105 10:16:36.544057       1 qemu.go:212] 12.06
```

```
vagrant@k8s-master:~$ kubectl get pvc
NAME      STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
ubuntu    Bound    pvc-37389376-69b2-4a59-99fe-0fe638f98046   4Gi        RWO            hostpath       22h
ubuntu2   Bound    pvc-45cc2474-8cb7-445f-9824-9fb2fcd60b50   4Gi        RWO            hostpath       3m36s
vagrant@k8s-master:~$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM             STORAGECLASS   REASON   AGE
pvc-37389376-69b2-4a59-99fe-0fe638f98046   4Gi        RWO            Delete           Bound    default/ubuntu    hostpath                22h
pvc-45cc2474-8cb7-445f-9824-9fb2fcd60b50   4Gi        RWO            Delete           Bound    default/ubuntu2   hostpath                3m38s
```

Figure 3. 12: CDI (Containerized Data Importer)

3.5. Bastion Host

Anything that provides perimeter access control security can be considered as the Bastion host [16] or the Bastion server. In fact, s Bastion host also known as a Jump Box is a particular purpose computer on a network that acts as a proxy server and allows the client machines to connect to the remote server. The Bastion hosts are used in the cloud environment as a server to provide access to a private network from an external network such as the internet. Since it is exposed to potential attacks, a Bastion host must be protected against the chances of penetration.

3.5.1. How Bastion host works

Bastion hosts are there to provide a point of entry into a network containing private network instances. The following figure shows that the details of how a Bastion server works.

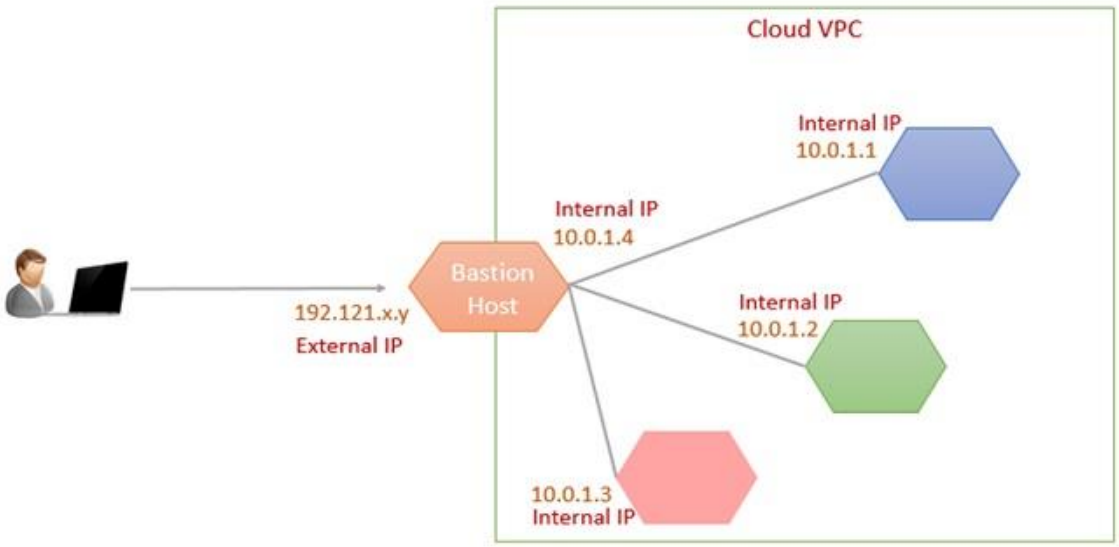

Figure 3. 13: Bastion host [16]

As shown in the figure, all the isolated instances have only internal IP addresses. However, the Bastion host instance has an external IP address as well as an internal IP address. If you need to access instances on the internal network that do not have an external IP address, you can connect to a bastion host then connect to the internal instances from that bastion host. this method uses a two-step SSH connection. Since the Bastion host uses a two-step SSH connection, it allows you to connect to the development environment without external IPs and additional firewall rules. When using a bastion host, the user will log into the bastion host first, and then into the private target instance. Because of this two-step login, the bastion hosts are sometimes called “jump servers”. The bastion host also can be started and stopped to enable or disable inbound SSH communication from the internet.

Chapter 4
OpenStack

This chapter is inspired by OpenStack documentation. OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all manage and provisioned through API with a common authentication mechanism. OpenStack is not a single piece of software but consist of multiple modules that are meant to serve different functionalities. The server where OpenStack software is called OpenStack node. Because it is modular, it does not need to run all these modules on every node. For example, if a server is providing just compute resources to be virtualized, only the computing virtualization management (Nova) module is needed on those nodes, and the nodes are called OpenStack compute nodes. Moreover, OpenStack needs an OpenStack controller node to manage its modules and perform centralized functions. These nodes can be spread across separate locations and may be geographically distributed. All other node types can coexist on the same server or be implemented separately on different servers. To the end-user, OpenStack is a self-service for infrastructure and applications. Users can do everything from simply provisioning virtual machines (VMs), to construct advanced virtual networks and applications, all within an isolated tenant (project) space which a way that OpenStack isolates assignments of resources. A hypervisor or Virtual Machine Monitor (VMM) is software that manages the emulation of physical hardware for virtual machines. OpenStack is not a hypervisor, but it does control hypervisor operations. Many hypervisors are supported under the OpenStack framework, including XenServer/XCP, KVM, QEMU, LXC, ESXi, Hyper-V, and others. Each OpenStack module may be implemented in multiple functional blocks of code, running as separate processes. When these modules need to communicate within their own functional blocks, they use a more efficient communication method through a messaging queue. A host system where OpenStack is running should, therefore, have this messaging queue capability installed. Several applications are available for that purpose, and if they are compliant with Advanced Message Queuing Protocol (AMQP), OpenStack can use it. The default one suggested is Rabbit Messaging Queue (RabbitMQ), which is an open-source implementation of AMQP.

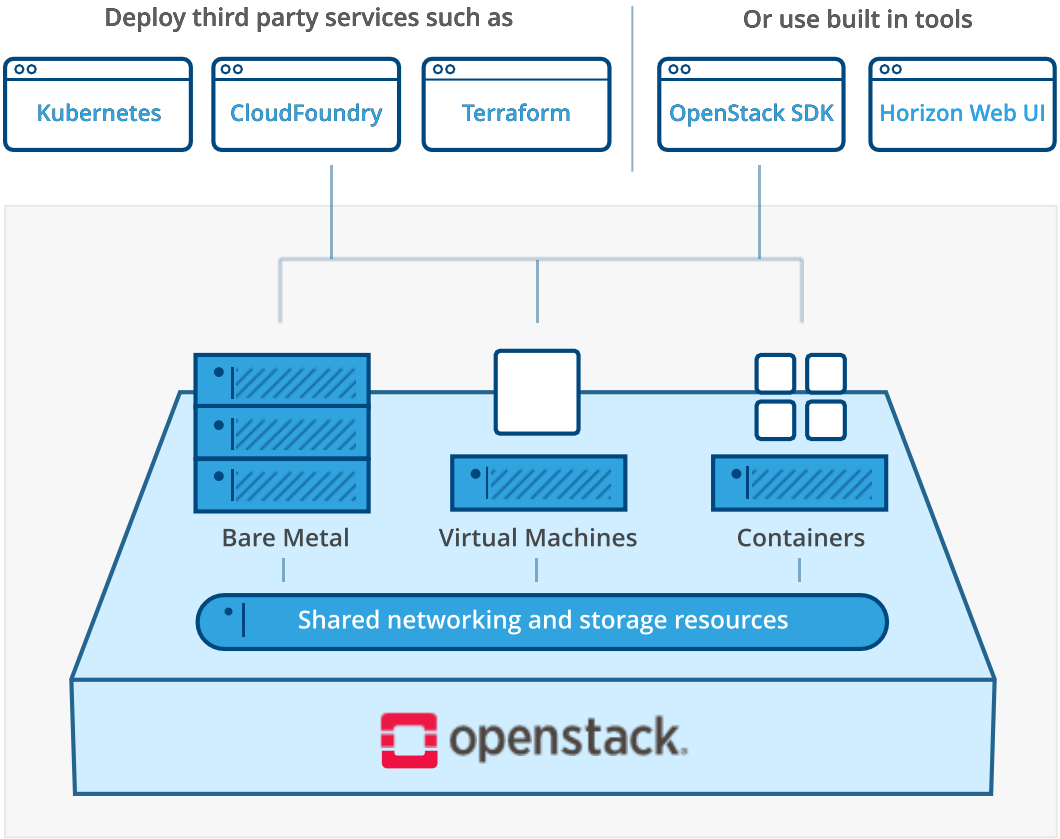

Figure 4. 1: OpenStack [17]

4.1. OpenStack Architecture

OpenStack consists of several independent parts, named the OpenStack services. All services authenticate through common identity service. Individual services interact with each other through public APIs, except where privileged administrator commands are necessary.

Internally, OpenStack services are composed of several processes. All services have at least one API process, which listens for API requests, preprocesses them, and passes them on the other parts of service. Apart from the Identity service, the actual work is done by distinct processes. For communication between the processes of one service, an AMQP message broker is used. The service's state is stored in a database. When deploying and configuring the OpenStack cloud, it is possible to choose among several message broker and database solution, such as RabbitMQ, MySQL, MariaDB, and SQLite.

4.1.1. OpenStack Compute(nova)

Nova is an OpenStack project that provides a way to provision compute instances (virtual servers). Nova supports creating virtual machines, BareMetal servers (through the use of ironic), and has limited support for system containers. Nova runs a set of daemons on top of existing Linux servers

to provide that service. Nova's main function is to interact with the hypervisor and facilitate the creation, deletion, and modification of allocated resources and image management of the virtual machine. It supplies a means to manage the virtual machine's life cycle. It requires the following additional OpenStack services for basic function:

- **Keystone:** This provides identity and authentication for all OpenStack services.
- **Glance:** This provides the compute image repository. All compute instances launch from glance images.
- **Neutron:** This is responsible for provisioning the virtual or physical networks that compute instances connect to on boot.
- **Placement:** this is responsible for tracking inventory of resources available in a cloud and assisting in choosing which provider of those resources will be used when creating a virtual machine.

The following diagram shows how various processes and daemons work together to form the compute service (nova) and interlink between them:

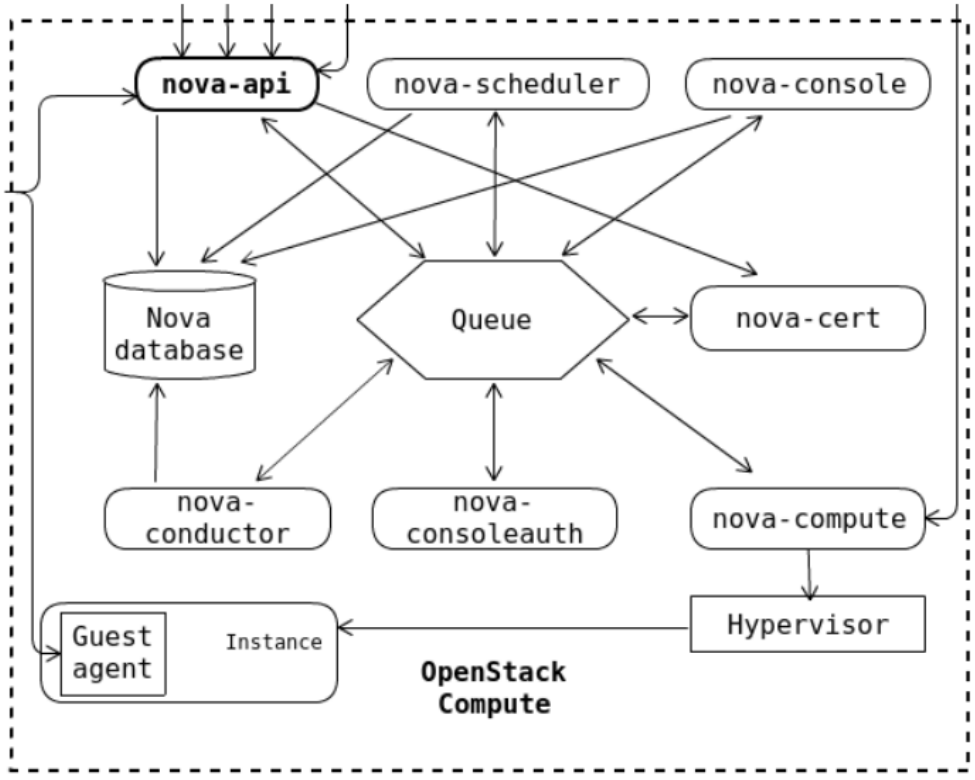

Figure 4. 2: OpenStack Compute(nova) [18]

Nova-api service

It listens and responds to the end-user compute API calls. The nova-api service takes care of some policies and initiating most orchestration activities, such as provisioning new virtual machines.

Nova-api-metadata service

The metadata service delivers the instance-specific data to the virtual machine instances. It is only used when running in multi-host mode. The instance-specific data includes hostname, instance-id, ssh-keys, and so on. The virtual machine accesses the metadata service via the special IP address at http://169.254.169.254:80, and this is translated to metadata_host:metadata_port by an iptables rule established by the nova-network service.

Nova compute service

Underneath, the entire lifecycle of the virtual machine is managed by the hypervisors. Whenever the end-user submits the instance creation API call to the nova-api service, the nova-api service processes it and passes the request to the nova-compute service. the nova-compute service processes the nova-api call for new instance creation and triggers the appropriate API request for virtual machine creation in a way that underlying hypervisor can understand. OpenStack has the flexibility to use multi-hypervisor environments in the same setup, that is, we could configure different hypervisors like KVM and VMware in the same OpenStack setup. The nova-compute service will take care of triggering the suitable APIs for the hypervisors to manage the virtual machine lifecycle.

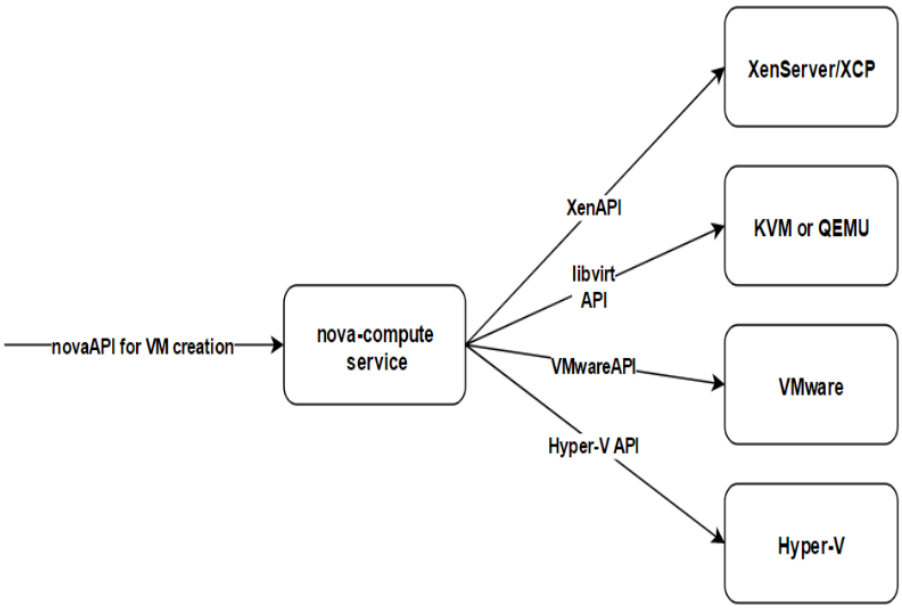

Figure 4. 3: Nova compute service [18]

Nova-scheduler service

When we have more than one compute node in our OpenStack environment, the nova-scheduler service will take care of determining where the new virtual machine will provision. Based on the various resource filters, such as RAM, CPU, Disk, availability zone, the nova-scheduler will filter the suitable compute host for the new instance.

Nova-conductor module

The nova-compute service running on the computer host has not direct access to the database because if one of the computer nodes is compromised, then the attacker has full access to the database. The nova-conductor module scales horizontally. However, do not deploy it on nodes where the nova-compute runs. With the nova-conductor daemon, the compromised node cannot access the database directly, and all the communication can only go through the nova-conductor daemon. So, the compromised node is now limited to the extent that the conductor APIs allow it.

Nova-consoleauth daemon

The nova-consoleauth daemon takes care of authorizing the tokens for the end-users, to access a remote console of the guest virtual machines provided by the following control proxies:

- The nova-novncproxy daemon provides a proxy for accessing running instances through a VNC onneciton. it Supports browser-based novnc clients.
- The nova-spicehtml5proxy daemon provides a proxy for accessing running instances through SPCIE connection. It supports browser-based HTML5 client.

The Queue (AMQP message broker)

A central hub for passing messages between daemons. Usually, AMQP message queue is implemented with RabbitMQ or ZeroMQ. In OpenStack, the AMQP message broker is used for all communication between the processes and daemons of one service. However, the communication between two different services in OpenStack uses service endpoints

Database

Most of the OpenStack services use an SQL database to store the build-time states for cloud infrastructure, such as for instance status, available networks, projects, and the list goes on.

4.1.2. Image service (Glance)

The image service enables users to discover, register and retrieve virtual machine images. It offers a REST API that enables you to query virtual machine image metadata and retrieve an actual

image. It supports the storage of disk or server images on various repository types, including OpenStack Object Storage.

The OpenStack Image service includes the following components:

Glance-api

The glance-api service processes the image API calls for image discovery, retrieval, and storage.

Glance registry service

the registry service takes care of storing, processing, and retrieving metadata about the images. Notably, the registry service only deals with the image metadata, not the image itself. Metadata includes image information such as size and type.

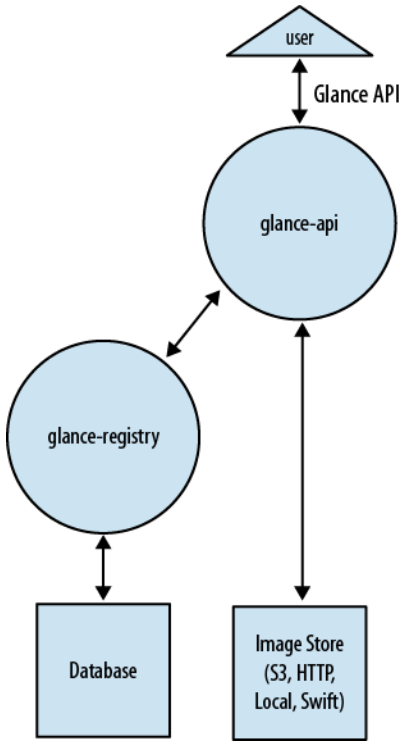

Figure 4. 4: Image service (Glance)

4.1.3. Identity service (KeyStone)

the exception to all other OpenStack services, the identity service has no dedicated API service to process and listen to the API calls. However, the actual work is done by distinct processes.

Server (KeyStone-all)

A centralized server processes the authentication and authorization request using a RESTful interface.

Drivers (optional)

With the help of the backend drivers, the centralized Keystone server can be integrated with the selected service backend, such as LDAP servers, Active Directory, and SQL databases. They can be used for accessing the identity information from the common repositories external to OpenStack.

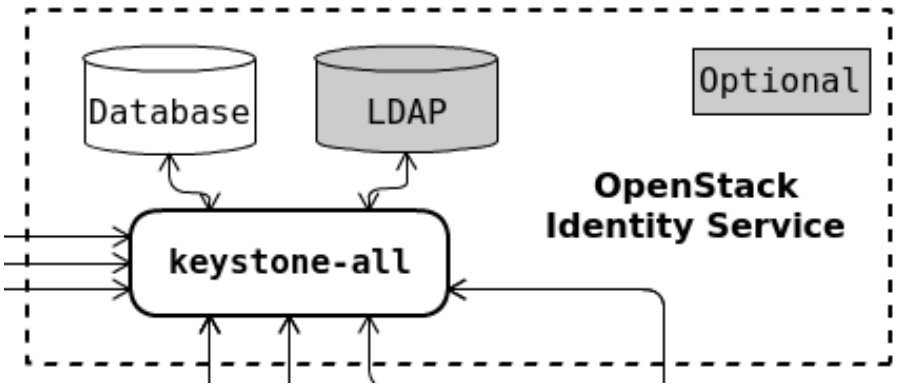

Figure 4. 5: Identity service (KeyStone)

4.1.4. Block storage service (Cinder) [18]

the Block Storage service provides persistent block storage management for virtual hard drives. Block storage allows the user to create and delete block devices, and to manage the attachment of block devices to servers. The actual attachment and detachment of devices are handled through integration with the Compute service. Both regions and zones can be used to handle distributed block storage hosts. The logical architecture diagram shows how various processes and daemons work together to perform the block storage (Cinder) in OpenStack, and the interconnection between them [19].

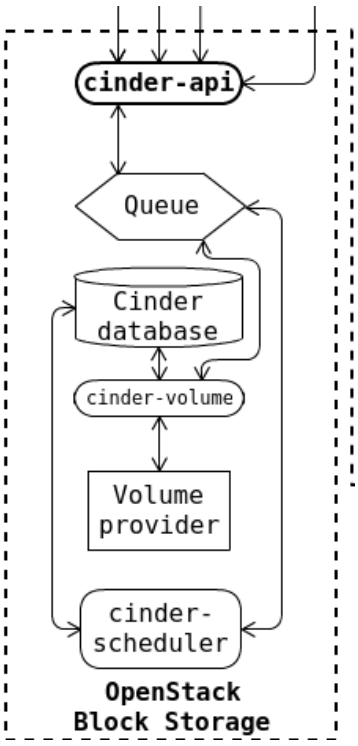

Figure 4. 6: Block storage service (Cinder) [18]

Cinder-api service

The cinder-api service accepts API requests and routes them to the cinder-volume for action.

Cinder-scheduler daemon

The cinder-scheduler daemon schedules and routes the requests to the appropriate volume service based upon your filter in the configuration settings, such as storage capacity, availability zones, volume types, and capabilities.

Cinder-volume

After filtering and choosing the suitable volume service, the cinder-scheduler will route the storage request to the cinder-volume services to process them. Then, the cinder-volume service interacts directly with the Block Storage service to provide required storage space. Notably, the cinder-volume service could interact with different storage providers at the same time through various device drivers available

Cinder-backup daemon

Whenever a request is received from volume backup/restore operations, the cinder-backup daemon will interact with the backup targets such as Swift object storage or NFS file store for backing up or restoring the volumes of any type.

Messaging queue

Routes information between the Block Storage processes.

4.1.5. Object storage (Swift)

OpenStack Object Storage is used for redundant, scalable data storage using clusters of standardized servers to store petabytes of accessible data. It is a long-term storage system for a large amount of static data that can be retrieved and updated. Object Storage uses a distributed architecture with no central point of control, providing greater scalability, redundancy, and permanence. Objects are written to multiple hardware devices, with the OpenStack software responsible for ensuring data replication and integrity across the cluster. Storage cluster scale horizontally by adding new nodes. When a node fails, OpenStack works to replicate its content from other active nodes. Because OpenStack uses software logic to ensure data replication and distribution across different devices, inexpensive commodity hard drives and servers can be used instead of more expensive equipment.

The Swift service is quite different from other OpenStack services because we can configure Swift services as standalone services to provide only the Object storage service to the end-users, without settings IAAS features. So, it is not the core service.

Proxy servers (swift-proxy-server)

the proxy server service accepts OpenStack object storage APIs and raw HTTP requests. The API and HTTP requests include file upload, create folder (container), and modify the metadata. For each request, it will look up the location of the account, container or object in the ring and route the request accordingly. It is also responsible for encoding and decoding the object data. Objects are streamed through the proxy server to the user.

Account servers (Swift-account-server)

It handles the request to process the metadata information for the individual accounts and the list of the containers mapped for each account.

Container servers (Swift-container-server)

The container servers handle the requests about container metadata and the list of objects within each container. The objects stored in the container have no information about the actual storage location but have information about the particular container where the objects get stored.

Object server (Swift-object-server)

It is a very simple block storage server that can store, retrieve, and delete objects stored on local devices. Objects are stored as binary files on the filesystem with metadata stored in the file's extended attributes(xattrs). This requires that the underlying filesystem choice for object servers support xattrs on files.

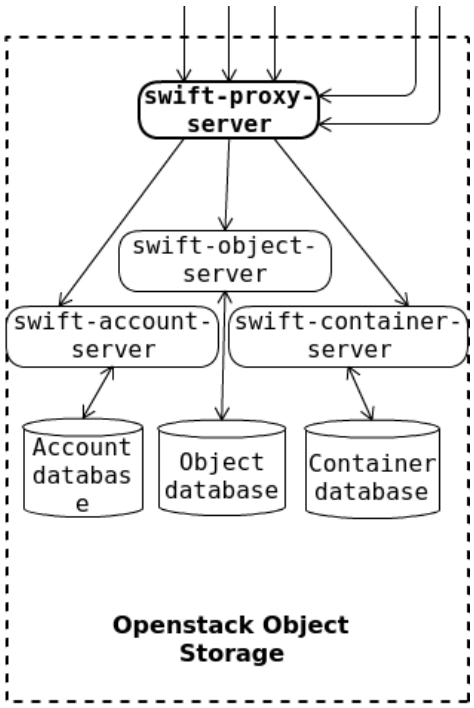

Figure 4. 7: Object storage (Swift) [18]

4.1.6. Telemetry service (ceilometer)

It provides user-level data for OpenStack based clouds, which can be used for customer billing, system monitoring, or alerts. Several modules combine their responsibilities to collect, normalized, and redirect data to be used for use cases such as metering, monitoring, and alerting. With help of agents, the ceilometer service will collect load and loads of metering data about all the OpenStack service usage. The collected metrics can be used for billing the resources and can also use for triggering the alarms when the obtained metrics or the event data break the defined threshold. For

better performance, the ceilometer service usually configured with a dedicated NoSQL database such as MongoDB.

Ceilometer-agent-compute

it runs on the compute node to collect the host and the virtual machines resources utilization statistics at regular polling intervals and send the collected statistics to the ceilometer collector to process them.

Ceilometer-agent-central

Apart from the metrics collected from the compute node, the ceilometer-agent-central takes care of collecting the resource utilization statistics at regular polling intervals from the other OpenStack services such as glance, cinder, and neutron.

Ceilometer-agent-notification

Unlike the other two ceilometer agents, the notification agent does not work with the polling method. However, when a new action is carried out by any OpenStack service, the incident will be communicated through the AMQP bus. The ceilometer-notification agent monitors the message queues for notifications and consumes the messages generated on the notification bus, then transforms them into Ceilometer metering data or events.

Ceilometer-collector

As the name states, the ceilometer collector runs on the central management server and collects the metering data from all the ceilometer agents mentioned previously. The collected metering data is stored in a database or can be configured to send to the external monitoring service, such as the Nagios monitoring service.

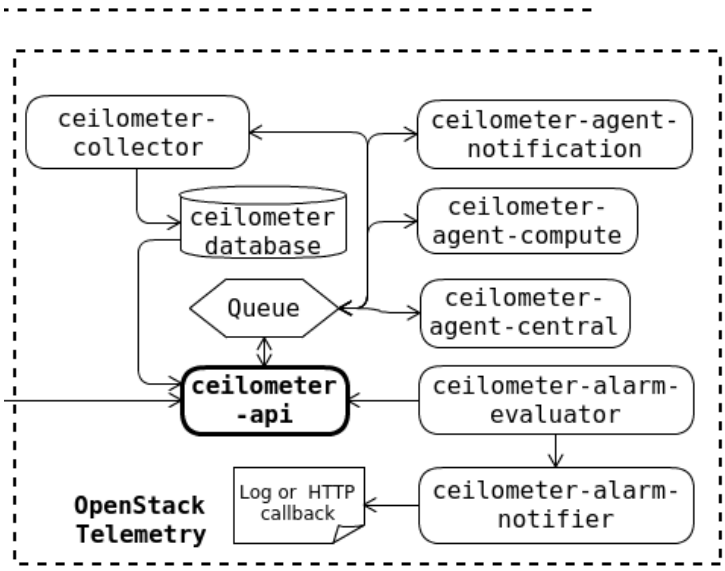

Figure 4. 8: Telemetry service (ceilometer) [18]

4.1.7. Orchestration service (Heat)

the Orchestration service provides a template-based way to create and manage cloud resources such as storage, networking, instances, or applications. Templates are used to create stacks, which are collection of resources (for example instances, floating IPs, volumes, security groups, or users). The service offers access to all OpenStack core services using a single modular template, with additional orchestration capabilities such as auto-scaling and basic high availability.

Heat-api component

The heat-api component provides an OpenStack-native REST API requests by sending them to the heat-engine over RPC (Remote Procedure Call).

Heat-api-cfn

the heat-api-cfn component provides an AWS Query API that compatible with AWS CloudFormation and processes API requests by sending them to the heat-engine over RPC.

Heat-engine

The heat engine handles the templates orchestration and reports back to the API consumer.

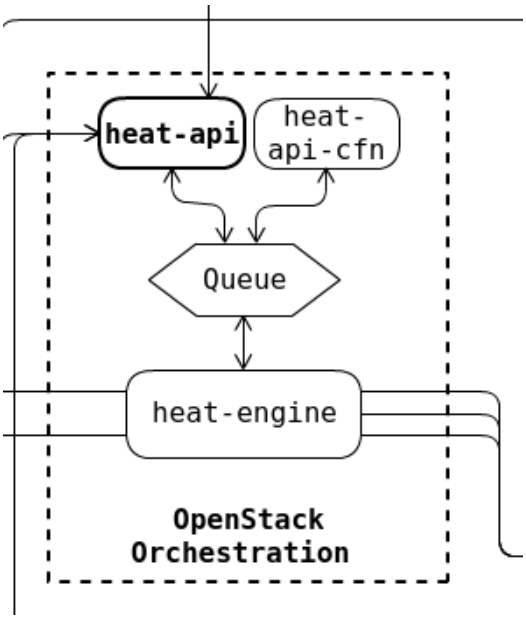

Figure 4. 9: Orchestration Service (Heat) [18]

4.2. OpenStack Networking (neutron)

the OpenStack networking service provides an API that allows users to setup and defines network connectivity and addressing in the cloud. It handles the creation and management of a virtual networking infrastructure, including networks, switches, subnets, and routers for devices managed by OpenStack compute service. Advanced services such as firewall or virtual private network (VPN) can also be used. OpenStack networking consists of the neutron-server, a database for persistent storage, and any number of plug-in agents, which provide other services such as interfacing with native Linux networking mechanisms, external devices, or SDN controllers. OpenStack networking is entirely standalone service that often deploys several processes across several nodes. These processes interact with each other and other OpenStack services.

The OpenStack networking components are:

Neutron server (neutron-server)

The neutron server, a Python daemon that exposes the OpenStack networking API and passes tenant to a suitable plug-in for additional processing. It also enforces the network model and IP addressing of each port. The neutron-server requires indirect access to a persistent database. This is accomplished through plugins, which communicate with the database using AMQP.

Plugin agent (neutron-*-agent)

it runs on the compute node to manage the local virtual switch (vswitch) configuration. Moreover, the agents receive messages and instructions form the Neutron server via plugins or directly on the AMQP message bus and the actual networking related commands and configuration are executed by the neutron-*-agent on the compute and the network node. The plug-in that you determine which agents run. This service requires message queue access and depends on the plugin used. Some of the most commonly used plug-in used in neutron agents listed here:

- neutron-lbass-agent: LBaaS agent
- neutron-linuxbridge-agent: Linuxbridge agent
- neutron-macvtap-agent: Macvtap L2 agent

DHCP agent (neutron-dhcp-agent)

Neutron utilizes dnsmasq, a free and lightweight DNS forwarder, and DHCP server, to provide DHCP services to networks. The neutron-dhcp-agent service is responsible for spawning and configuration dnsmasq and metadata processes for each network that leverage DHCP.

L3 agent (neutron-L3-agent)

L3 agent Provides L3/NAT forwarding for external network access of VMs on tenant networks. It requires message queue access.

Network provider services (SDN server/services)

It Provides additional networking services to tenant networks. These SDN services may interact with neutron-server, neutron-server, neutron-plugin, and plugin-agents through communication channels such as REST APIs. The Neutron SDN layer indicates a logical model for communicating with the underlaying infrastructure to route network traffic. The SDN layer is divided into two categories which are core plugin or service plugins. Core plugins define the core definitions of network, subnet, and port, and deals with L2 networking and IP address management (IPAM). Service plugins deal L3 routers, Load balancer, VPNs, Firewalls and L4-L7 services. When a neutron is using a third-party SDN layer, it is the plugin code that is executed as part of neutron server on the controller node. These plugins implement the core neutron APIs which interact with the neutron server, database, and agents [17].

The following figure shows an architectural and networking flow diagram of the OpenStack networking components:

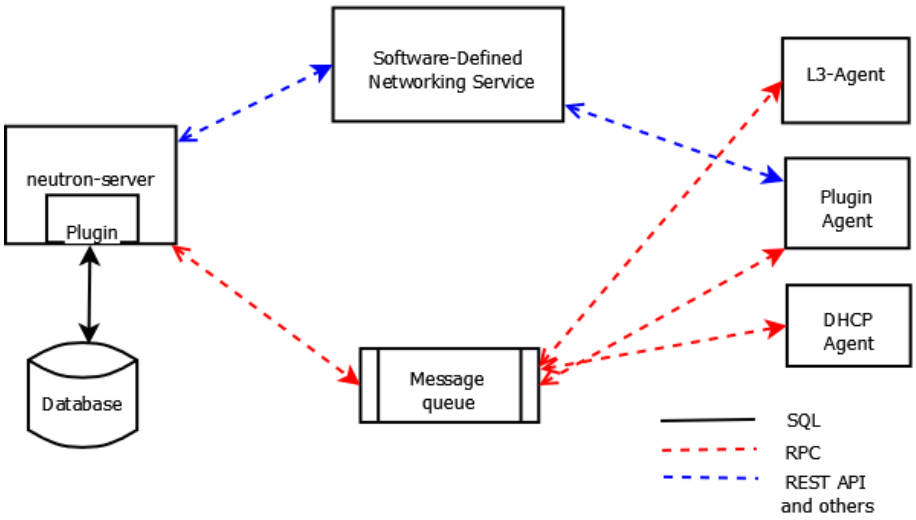

Figure 4. 10: Flow diagram of OpenStack networking components [20]

OpenStack Networking service placement on physical servers

A standard OpenStack networking setup has up to four physical data center networks:

Management network

It used for internal communication between OpenStack components. The IP addresses on this network should be reachable only within the data center and is considered the Management Security Domain.

Guest network

It used for VM data communication within the cloud deployment. The IP addressing requirements of this network depend on the OpenStack networking plug-in are using and the network configuration choices of the virtual networks made by the tenant. This network is considered the Guest Security Domain.

External network

Used to provide VMs with Internet access un some deployment scenarios. The IP addresses on this network should be reachable by anyone on the Internet. This network is considered to be in the Public Security Domain

API network

Exposes all OpenStack APIs, including the OpenStack networking API, to the tenant. The IP addresses on this network should be reachable by anyone on the Internet. This may be the same network as the external networks, as it is possible to create a subnet for the external network that uses IP allocation ranges to use only less than the full range of IP addresses in an IP blocks. This network is considered the Public Security Domain.

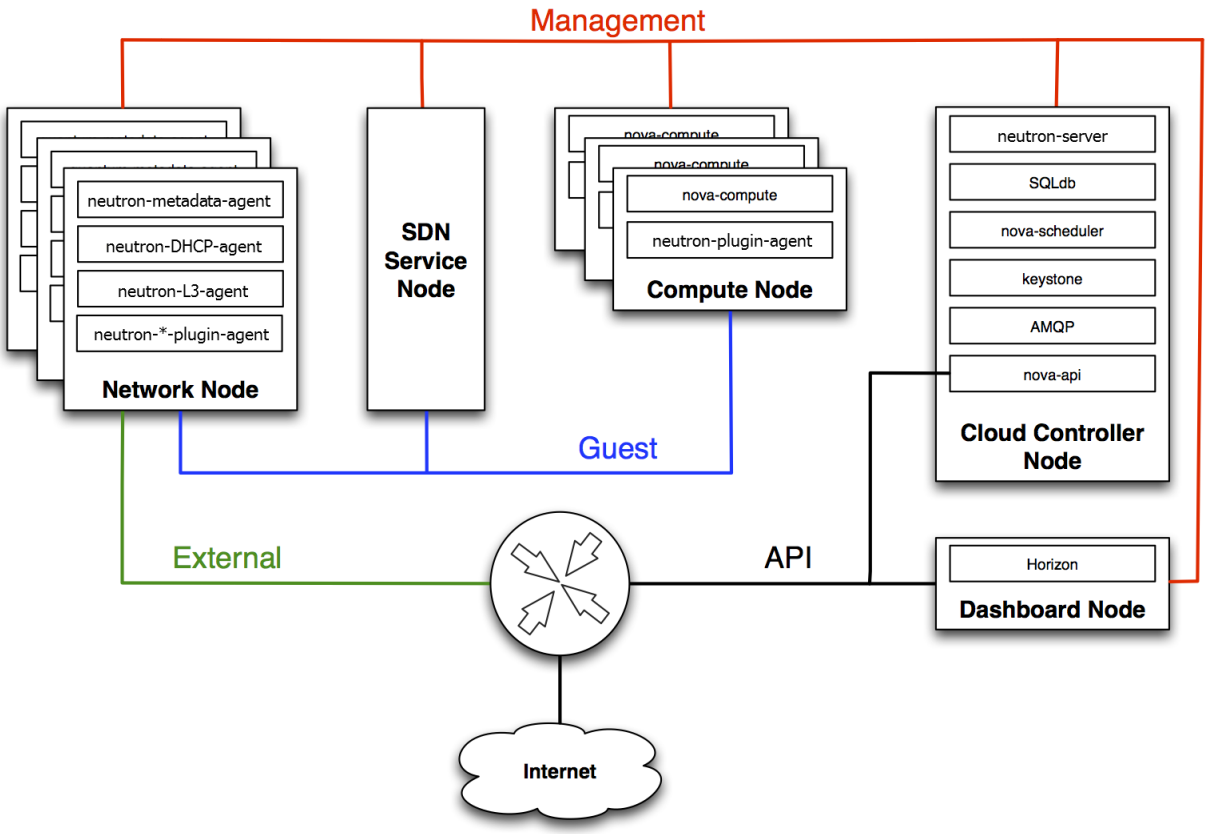

Figure 4. 11: API network [20]

4.2.1. Network types

The network types in OpenStack neutron is broadly classified into two types:

- Provider networks
- Self-service networks

Provider networks

Provider networks offer layer-2 connectivity to instances with optional support for DHCP and metadata services. it connects to the existing layer-2 physical networks or VLAN in the data center,

typically using VLAN (802.1q) tagging to identify and separate them. The OpenStack user with a member role cannot create or configure the network topologies in the provider network. The provider network can only be set up by an OpenStack admin who has access to manage the actual physical networks in the data center. This is because configuring the provider networks requires configuration changes in the physical network infrastructure level.

- **Routed provider networks**
  It offers layer-3 connectivity to instances. These networks map to existing layer-3 networks in the data center. More specifically, the network maps to multiple layer-2 segments, each of which is essentially a provider network. Each has a router gateway attached to it which routes traffic between them and externally. The networking service does not provide routing.

Self-service networks

Self-service networks enable the users with member role to create their own network topology within the user's project without involving OpenStack administrator's help. The users can create as many virtual networks as they liked (not exceeding the project quota limit), interconnected with virtual routers and connected to an external network. Also, self-service networks offer VxLAN/GRE encapsulation protocols. In that way, the user can create a private tunnel within the OpenStack nodes. This will allow the user to setup their own subnets and can be linked to the external or provider networks using an OpenStack router. By default, the self-service networks create in any specific tenant is entirely isolated and are not shared with other tenants in OpenStack. With the help of network isolation and overlay technologies, the user can create multiple private (self-service) networks in the same tenant and can also define their own subnets, even if that subnet ranges overlap with the subnet of another tenant network topology.

4.2.2. OpenStack Layer 3 routing

OpenStack networking [21] provides routing services for project networks. without a router, instances in a project networks are only able to communicate with one another over a shared L2 broadcast domain. Creating a router and assigning it to a tenant network allows the instances in that network to communicate with other project networks or upstream if the gateway is defined for the router.

- **Centralized Routing**
  Originally, neutron was designed with a centralized routing model where a project's virtual routers, managed by the neutron L3 agent, are all deployed in a dedicated node or cluster of nodes. This means that each time the routing function is required (east/west, floating IPs or SNAT), traffic would traverse through a dedicated. This introduced multiple challenges and resulted in sub-optimal traffic flows. For example:

i. Traffic between instances flows through a controller node. When two instances need to communicate with each other using L3, traffic has to pass the controller node. Even if the instances are scheduled on the same compute node, traffic still has to leave the compute node, flow through the controller, and route back to the compute node.
ii. Instances with floating IPs receive and send packets through the controller node. The external network gateway interface is available only at the controller node, so whether the traffic is originating from an instance, or destined to an instance from the external network, it has to flow through the controller node. Consequently, in the large environment, the controller node is subject to the heavy traffic load. This would affect performance and scalability.

To better scale L3 agent, neutron can use the L3 HA (High Availability) feature, which distributes the virtual routers across multiple nodes. Moreover, Distributed Virtual Routing (DVR) offers an alternative routing design, which intends to isolate the failure domain of the controller node and optimize network traffic deploying the L3 agent and schedule routers on every compute node.

4.3. Vhost-net/virtio-net Architecture

Virtio was developed as a standardized open interface for virtual machines (VMs) to access simplified devices such as block devices and network adaptors. Virtio-net is a virtual Ethernet card. the virtio specification is based on two elements: devices and drivers. The hypervisor exposes the virtio devices to the guest through several transport methods. The most common transport method is PCI or PCIe bus which expose a virtio device through virtual PCI port. In the VM world, the hypervisor captures accesses to the memory range like physical PCI hardware and performs device emulation, exposing the same memory layout that a real machine would have and offering the same responses. The following building blocks explains the environment to how virtio is connected:

- **KVM:** Kernel Based Virtual machine that allows Linux to playing a role as a hypervisor. So, a host machine can run multiple isolated virtual environments called guests. KVM is running in the host kernel space.
- **QEMU:** A hosted virtual machine monitor that through emulation, provides a set of different hardware and device models for the guest machine. QEMU can be used with KVM to run a virtual machine at near-native speed leveraging hardware extensions. It creates for each guest VM, so if it creates for N VMs, there will have N qemu processes which libvirt communicates with each of them.
- **Libvirt:** an interface that translates XML-formatted configurations to QEMU CLI calls. It is running in the host user space.

The virtio-networking has two layers:

- **Control plane:** it used for exchanging negotiation between the host and guest both for establishing and terminating the data plane. The control plane for virtio is implemented in the qemu process based on the virtio specification.
- **Data plane:** the virtio driver must be able to allocate memory regions that both the hypervisor and the devices can access for reading and writing e.g. via memory sharing. It called data plane which is used for transferring the packets between host and guest. Vhost protocol enables the data plane than going directly from kernel host by passing the qemu process.

Vhost-net is the backend running in the host kernel space, and virtio-net is the frontend running in the guest kernel space. The data plane communication, receive and transmit is accomplished through dedicated queues. To each guest, we can associate a number of virtual CPUs (vCPUs) and RX/TX queues are created per CPU [22].

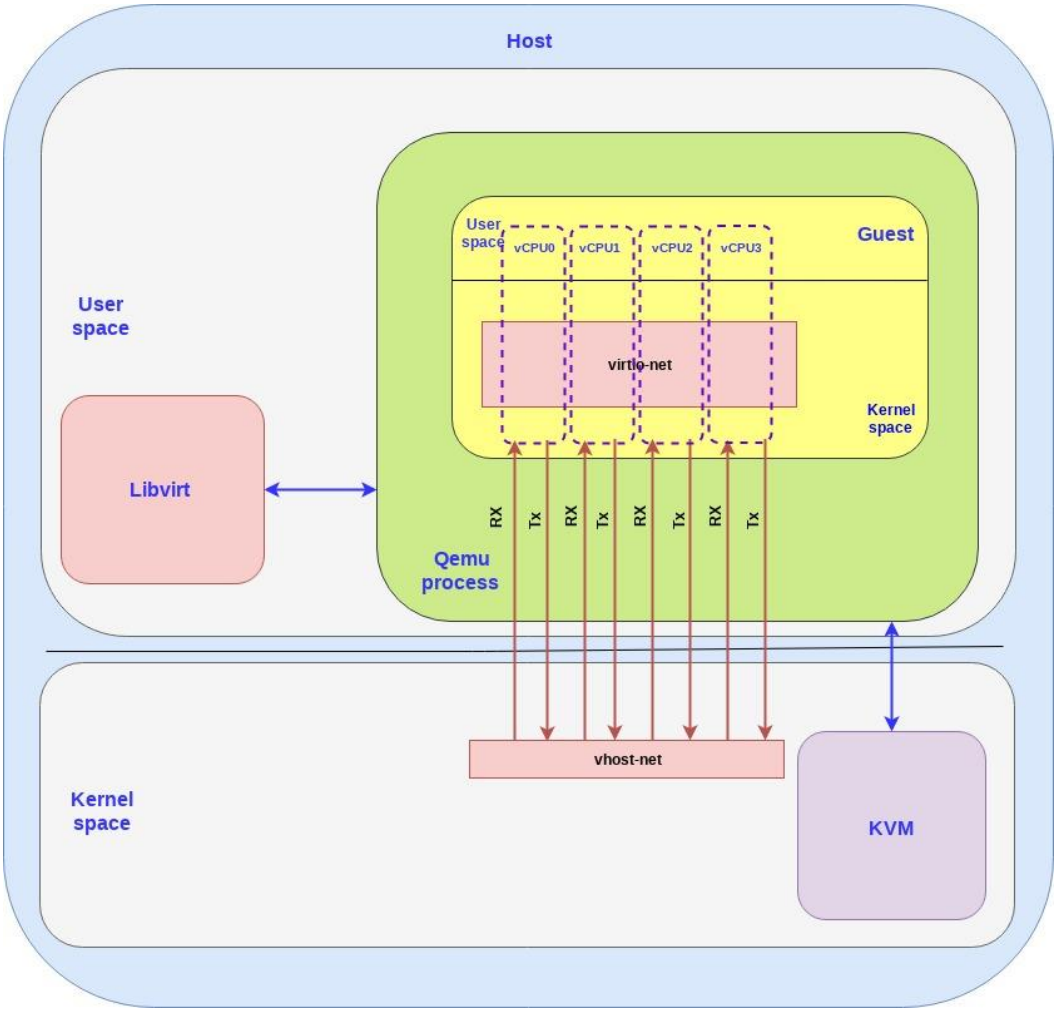

Figure 4. 12: Vhost-net/virtio-net Architecture [22]

Chapter 5
Network traffic flows patterns analysis

5.1. Network Traffic flows on OpenStack

5.1.1. Provider network traffic flow

In this part, it would be observed that the Provider network traffic flows of VMs on different scenarios. North-South traffic which pass between an instance and external network like Internet. Also, East-West traffic traverse between instances on the same network or different networks. In all cases, the physical network infrastructure manages to switch and to route among provider networks and external networks such as the Internet. In each scenario, one or more of the following components would be referenced:

- Provider network 1 (VLAN)
    - VLAN ID 101
    - IP address range 203.0.113.0./24
    - Gateway (resides on physical infrastructure) Ip address 203.0.113.1
- Provider network 2 (VLAN)
    - VLAN ID 102
    - IP address range 192.0.2.0/24
    - Gateway Ip address 192.0.2.1

Scenario 1 (Same provider network, Different compute nodes)

Instances can interact directly between compute node which are on the same provider network which sending layer 2 packets.

- Instance VM4 resides on compute node 1 and uses provider network 1.
- Instance VM5 resides on compute node 2 and uses provider network 1.

1. The instance VM4 sends the packet from its interface to Linux bridge2 through veth pair and Hypervisor handles this connection to tap4 port on Linux Bridge 2.
2. On the Linux bridge 1 security rules and firewalling would be handled.
3. The Linux bridge2 transfer packet to VLAN sub-interface port eth1.102 and it sends to the physical network interface eth1 and physical network interface eth1 adds VLAN tag eth1.102 So, the interface eth1 sends the packet to physical infrastructure switch.
4. The switch forwards the packet from compute node 1 to compute node 2

5. In the compute node 2, the physical network interface eth1 eliminates the tag 102 from the packet and send it to the VLAN sub-interface port on the Linux Bridge3.
6. On Linux Bridge3, the security rules and firewalling would be handled, then the packet sends to the instance VM2 from tap port on Linux bridge3 to VM2 interface through veth.

Scenario 2 (Different network, Same compute nodes)

In this case, VM instances would be communicated through the physical router due to using different provider networks.

- Instance VM1 resides on compute node 1 and uses provider network 1.
- Instance VM3 resides on compute node 1 and uses provider network 2

1. The instance VM1 sends the packet from its interface to Linux bridge1, and from VLAN sub-interface eth1.101 on Linux Bridge 1 the packet would be transferred to physical network interface eth1.
2. The physical network interface eth1, adds VLAN tag 101 to the packet and forwards it to the physical network infrastructure switch.
3. The switch removes VLAN tag 101 from the packet and forwards it to the router. Then, the router routes the packet from provider network 1 to provider network 2 and forwards the packet to the switch.
4. The switch adds VLAN tag 102 to the packet and passes it to compute node 1.
5. On the compute node 1, the Physical network interface eth1 removes VLAN tag 102 from the packet and sends it to the VLAN sub-interface port eth1.102 on the Linux Bridge2. Then, the packet would be sent to instance VM3 from the tap port on the Linux Bridge2.

Scenario 3 (Same network, same compute node)

Instances communicate via the Linux bridge on the same compute node.

- Instance VM1 resides on compute node 1 and uses network provider 1
- Instance VM2 resides on compute node 1 and uses network provider 1

Instance VM1 sends a packet to instance VM2 through same Linux bridge.

The following steps involve the compute node 1:

1. The instance VM1from its interface sends the packet to the tap port of Linux Bridge1 through veth pair. The packet includes destination MAC layer of instance VM2 because the destination instance VM2 resides on the same network.
2. Security group rules on the provider bridge handle firewalling and connection tracking for the packet.
3. The Linux bridge1 sends the packet to the instance VM2 interface through veth pair because both instance VMs exists in the same L2 broadcast domain

Scenario 4: (Floating IP/Fixed IP traffic)

- Instance VM6 resides on compute node 2 and uses network provider 1
- Instance VM6 sends a packet to a host on the Internet.

1. The instance VM6 from its interface forwards the packet to the Linux Bridge 4, through veth connect to tap6 port on Linux Bridge 4.
2. On the Linux Bridge 4, the security group rules and firewalling would be managed. The VLAN sub-interface eth1.101 on Linux Bridge 4 forwards the packet to the physical network interface eth1. The physical network interface eth1 adds VLAN tag 101 to the packet and forwards it to the physical network infrastructure switch.
3. The switch removes the VLAN tag 101 from the packet and sends it to the router.
4. The router routes the packet from provider network 1 to the external network.

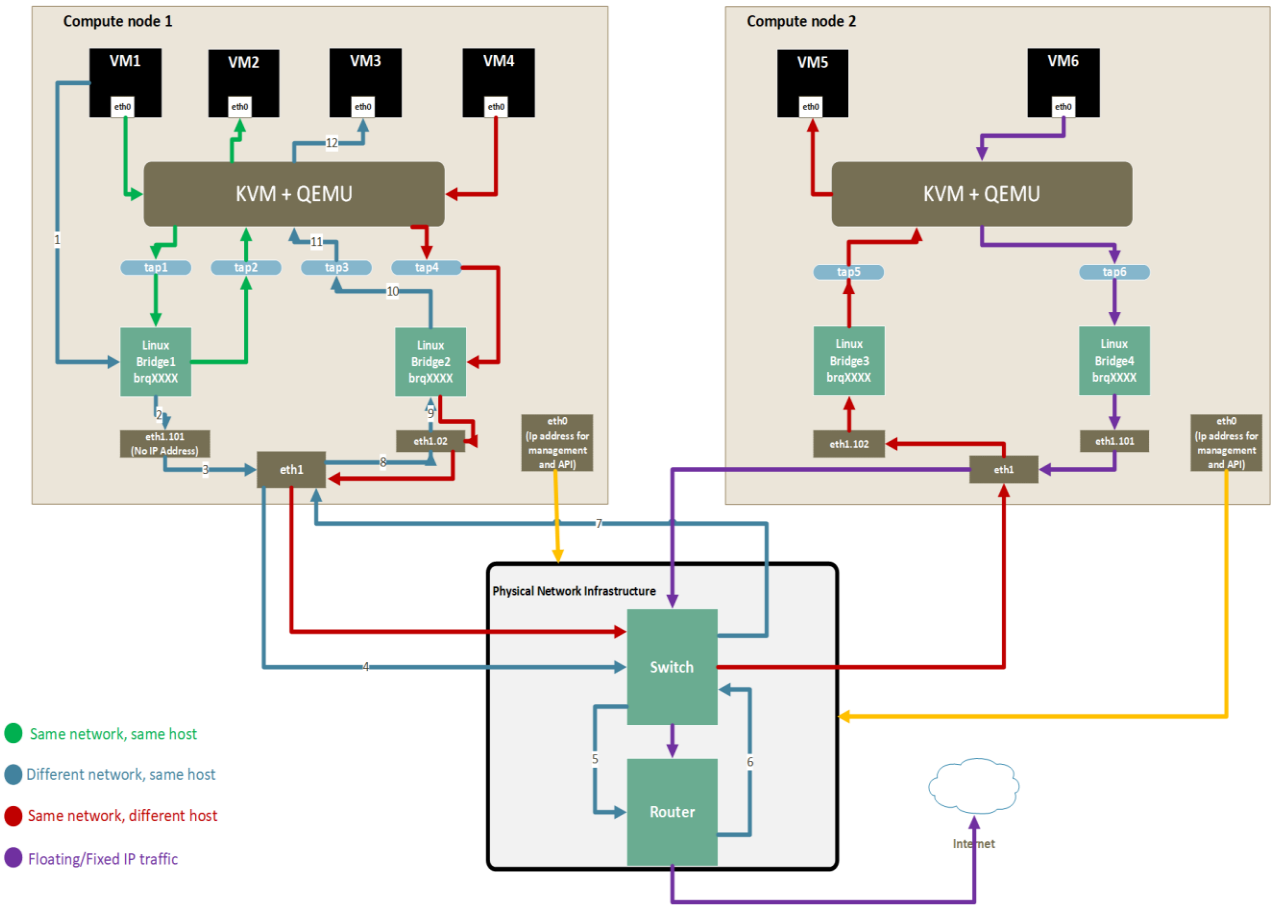

Figure 5. 1: Provider network traffic flow

5.1.2 Self- service Network traffic flow

It would be investigated the flow of network in some different scenarios. North-south network traffic travels between an instance and external network such Internet. East-west network traffic travels between instances on the same or different networks. In all cases, the physical network infrastructure handles switching and routing among provider networks and external networks such as the Internet. In each scenario refers to one or more of the following components:

- Provider network (flat network)
- Self-service network 1 (VXLAN)
  - VXLAN ID 101 (tagged)
- Self-service network 2 (VXLAN)
  - VXLAN ID 102 (tagged)
- Self-service router
  - Gateway on the provider network

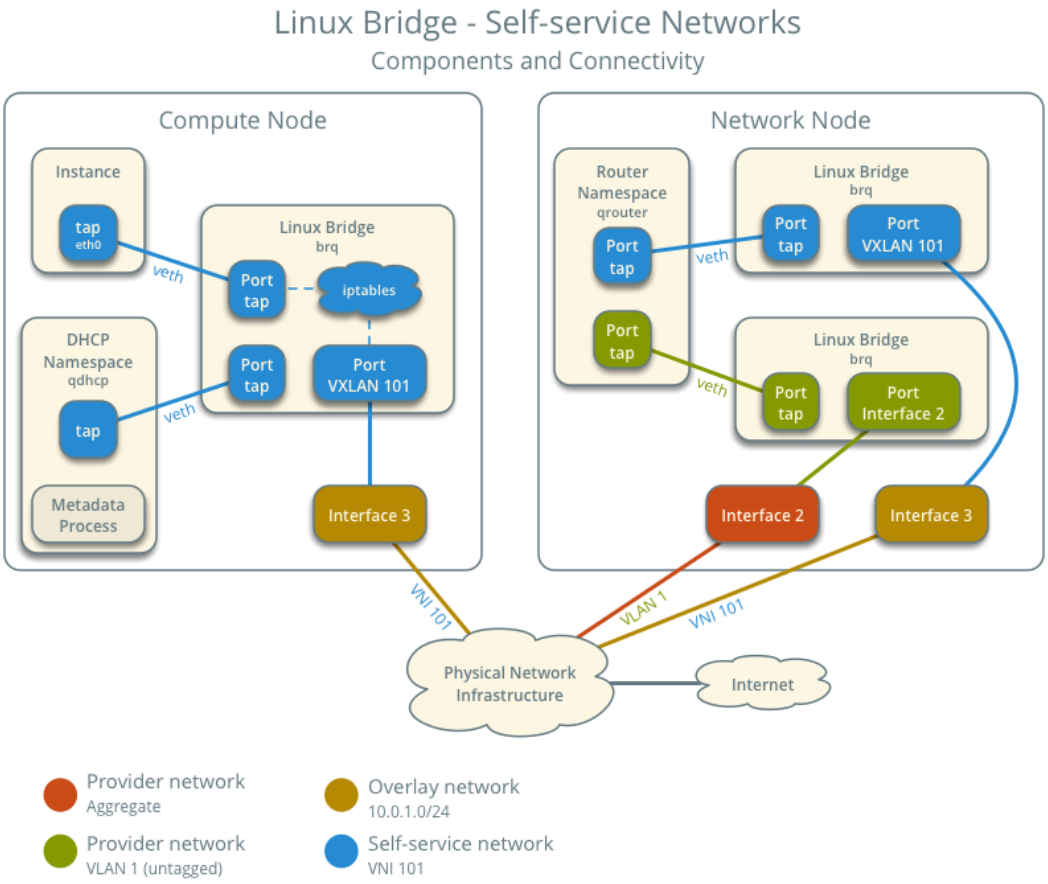

Figure 5. 2: Self-service network traffic flow [23]

Scenario 1 (Same network, different compute nodes)

Instances on the same network communicate directly between compute node containing those instances.

- Instance VM5 resides on compute node 2 and uses self-service network 1.
- Instance VM4 exists on compute node 1 and uses self-service network 1.
- Instance VM5 sends a packet to instance VM4.

The following steps include compute node 2:

1. the instance VM5 interface forwards the packet to the self-service bridge instance port via veth pair.
2. security group rules on the self-service bridge handle firewalling and connection tracking for the packet.
3. The self-service bridge forwards the packet to the VXLAN interface which wraps the packet using VNI 101.

4. For the underlaying physical interface for the VXLAN interface forwards the packet to compute node 1 via the overlay network.

The following steps involve compute node 1:

1. The underlaying physical interface for the VXLAN interface forwards the packet to the VXLAN interface which unwraps the packet.
2. Security group rules on the self-service bridge handle firewalling and connection tracking for the packet.
3. The self-service bridge instance port forwards the packet to the instance VM4 interface via veth pair.

Scenario 2 (Different network, same compute nodes)

Instances using a fixed IPv4/floating IP address to communicate via router on the network node. The self-service networks must reside on the same router.

- Instance VM3 resides on compute node 1 and uses self-service network 1
- Instance VM4 resides on compute node 1 and uses self-service network 2
- Instance VM3 sends a packet to instance VM4.

Both instances reside on the same compute node to illustrate how VXLAN enables multiple overlays to use the same layer-3 network.

The following steps associate compute node 1:

1. The instance VM3 interface forwards the packet to the self-service bridge instance port via veth pair.
2. Security group rules on the internal bridge handle firewalling and connection tracking for the packet
3. The self-service bridge forward packet to the VXLAN interface which wraps the packet using VNI 101.
4. The underlaying physical network interface for the VXLAN interface forwards the packet to the network node via overlay network.

The following steps involve the network node:

1. The underlying physical interface for the VXLAN interface forwards the packet to the VXLAN interface which unwraps the packet.
2. The self-service bridge router port forwards the packet to the self-service network 1 interface in the router namespace.
3. The router sends the packet to the next-hop IP address, typically the gateway IP address on self-service network 2, via the self-service network 2 interface.

4. The router forwards the packet to the self-service network 2 bridge router port.
5. The self-service network 2 bridge forwards the packet to VXLAN interface which wraps the packet using VNI 102.
6. The physical network interface for the VXLAN interface sends the packet to the compute node via the overlay network.

The following steps involve the compute node 1:

1. the underlaying physical network interface for the VXLAN interface sends the packet to the VXLAN interface which unwraps the packet.
2. Security group rules on the self-service bridge handle firewalling and connection tracking for the packet.
3. The self-service bridge instance port forwards the packet to the instance VM4 interface via veth pair.

Scenario 3: (Same network, same compute node)

Instances communicate via internal bridge on the same compute node.

- Instance VM1 resides on compute node 1 and uses self-service network 1
- Instance VM2 resides on compute node 1 and uses self-service network 1

Instance VM1 sends a packet to instance VM2 through the same internal bridge.

The following steps involve the compute node 1:

1. The instance VM1 interface sends the packet to the internal bridge instance port through veth pair. The packet includes destination MAC layer of instance VM2 because the destination instance VM2 resides on the same network.
2. Security group rules on the provider bridge handle firewalling and connection tracking for the packet.
3. The internal bridge sends the packet to the instance VM2 interface through veth pair because both instance VMs exists in the same L2 broadcast domain.

Scenario 4: (Floating IP/ Fixed IP traffic)

- Instance VM1 resides on compute node 1 and uses self-service network 1
- Instance VM1 sends a packet to a host on the Internet.

The following steps involve compute node 1:

1. The instance VM1 interface forwards the packet to the self-service bridge instance port via veth pair.

2. Security group rules on the self-service bridge handle firewalling and connection tracking for the packet
3. The self-service bridge forwards the packet to the VXLAN interface which wraps the packet using VNI 101.
4. The underlaying physical interface for the VXLAN interface forwards the packet to the network node via the overlay network.

The following steps involve the network node:

1. The underlaying physical interface for the VXLAN interface forwards the packet to the VXLAN interface which unwraps the packet.
2. The self-service bridge router port forwards the packet to the self-service network interface in the router namespace.
    - For IPv4, the router performs SNAT on the packet which changes the source IP address to the router IP address on the provider network and sends it to the gateway IP address on the provider network and sends it to the gateway interface on the provider network.
3. The router forwards the packet to the provider bridge router port.
4. The flat sub-interface port on the provider bridge forwards the packet to the physical network interface.
5. The provider physical network interface forwards the packet to the internet via network infrastructure.

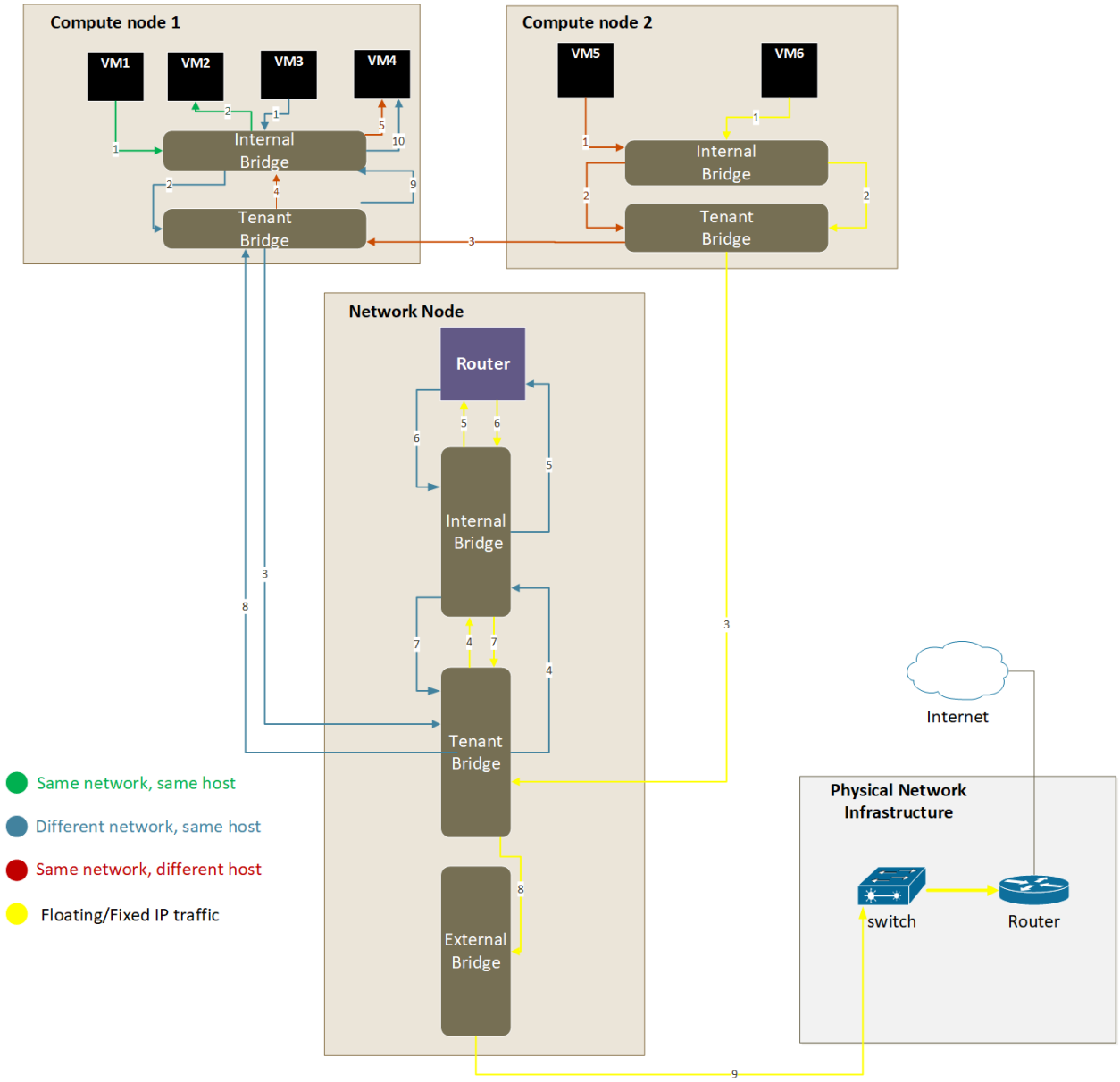

Figure 5. 3: Self-service network traffic flow

Concisely, the OpenStack network packets come in the form of centralized networking and its dependencies on multiple networking components for communication. The VM traffics are required to be routed via the network node. In the production environment, network node is implemented independently than a compute node where VMs are run, so this centralize routing introduces bottleneck which causes significant problems like high latency and throughput degradation. There are some solutions such as Distribution Virtual Routing (DVR) which allows L3 routers directly on compute nodes. Consequently, instances traffic North-South, or East-West could traverse without first requiring routing through Network node.

5.2. KubeVirt network traffic flow

In this section, it would be routing of VMs traffic flows through KubeVirt solution which offers to deploy virtual machines in Pods. It is utilized flannel CNI (Container Network Interface) plugin for network overly on Kubernetes.

- **Cni0:** is a Linux bridge device which all veth (virtual ethernet) devices can connect to this bridge. Moreover, all Pods on the same node can communicate with each other through cni0 bridge.
- **Flannel.<vni>:** it creates as a VXLAN device when VXLAN backend is used by flannel. The VXLAN VNI is set to 1 by default. It has VXLAN UDP port number 8472 and the nolearning tag which disables source-address learning. It means that Unicast with static L3 entries for the peers is used. The VXLAN flannel.1 is connected to physical network device eth0 for sending traffic via physical network.
    - **Flanneld:** It is run as an agent on each node which is responsible for managing a unique subnet on each host, distributing IP address to each Pod on its host and mapping routes from one Pod to another, even if it resides on different hosts. Also, it populates node ARP table. Each flanneld agent provides this information to a centralized etcd database, so other agents on hosts can route packets to other Pods within flannel network.

Scenario 1(Same Subnet, Same Node)

From the VM level, The VM1 sends the packet to bridge of Virtual launcher pod from interface eth0 that connects to br1 bridge through vnet0 in Pod level which resides in virtual launcher pod. Moreover, eth0 in Pod level has no IP address and br1 is in self-assigned ranged. So, the Pod has no network access. The packet contains the destination MAC layer of VM2 due to destination VM2 exists on the same subnet. From Pod level interface eth0 packet sends to cni0 bridge through veth1. Then, cni0 bridge transfer packet to VM2 to Pod level and VM level respectively.

Scenario 2(Different Subnet, Different Node)

From the VM level, VM4 sends packet through its interface eth0 to br1 bridge of pod virtual launcher by vnet0. From Pod level, br1 forwards the packet to cni0 bridge via veth2. Then it transfers to device flannel.1 based on routing table entry 10.244.0.0 10.244.0.0 255.255.255.0 flannel.1. The flannel.1 uses 10.244.0.0's MAC address 1a:2e:95:6d:18:40 which is populated by flanneld as the destination MAC for inner ethernet packet. The flannel.1 needs to gain VXLAN tunnel endpoint (VTEP)'s destination IP address to send it out. By searching for 1a:2e:95:6d:18:40 MAC address from Bridge in fdb which is flannel database, it could be observed that flannel.1 has

the IP address of Node1. Thus, the wrapped VXLAN packet is sent to Node1. The packet will traverse to virtual launcher Pod 2 and VM2 through applying the reversed packet processing logic.

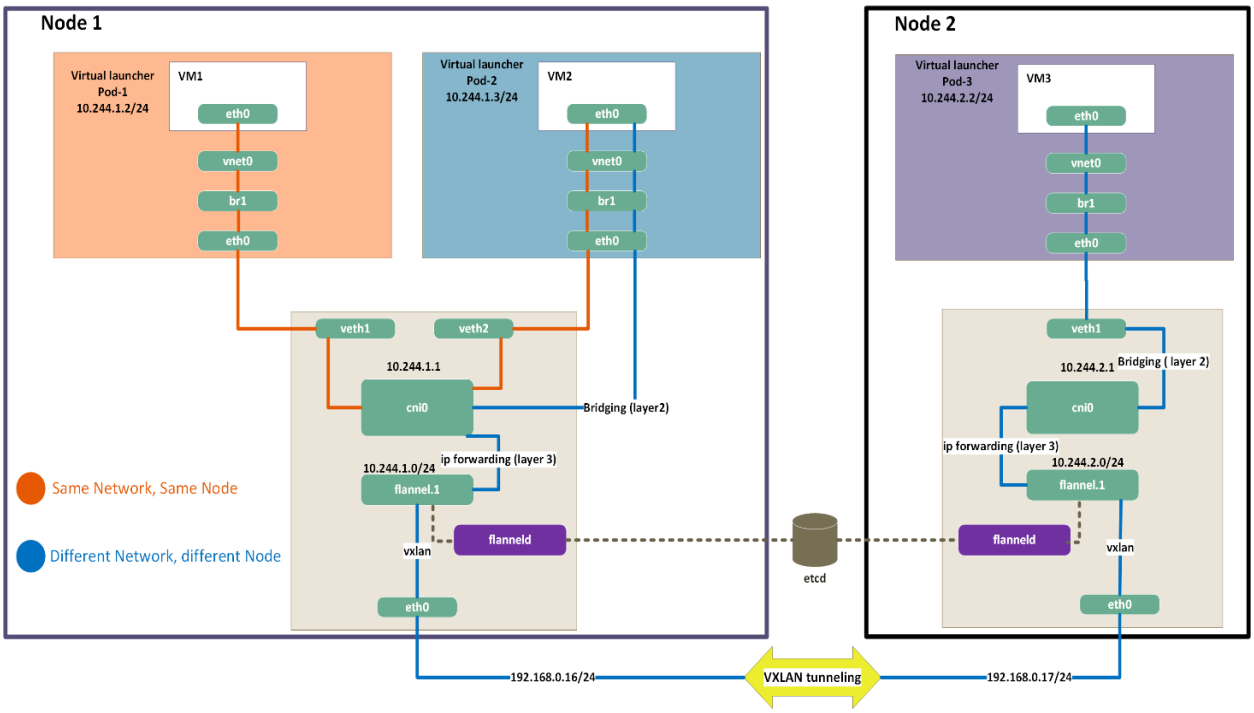

Figure 5. 4: KubeVirt network traffic flow

Chapter 6
Network performance measurements

6.1. OpenStack Network performance measurements

In our production environment, it is used hardware and software based on the below list:

Hardware components

- 4 server DELL R740x with 4 x 1GB/s eth NIC, 2 x 10GB/s eth NIC 4
- 1 switch Cisco SG350X-24 10GB/s
- 1 switch Cisco SG350XG-24D 1GB/s

Software

- Ubuntu server 18.04 LTS
- OpenStack-Ansible
- OpenStack 18.1.5 Rocky
- Hypervisor: KVM + QEMU
- Neutron agent plugin: ML2
- Mechanism driver: Linux bridge

Performance measurement tools

- Iperf3 for throughput
- qperf for latency

the measurements are taken with four different scenarios which are based on TCP throughput and latency. One pair of instances VM runs at a time for measuring each parameter. Also, all the tests are performed on the self-service network and the flat provider network is employed for external connections. Moreover, in our OpenStack installation, it is utilized Layer 3 HA (High Availability) mechanism using Virtual Router Redundancy Protocol (VRRP) via keepalived and provides failover of routing for self-service networks. also, it is used active/passive configuration for maintaining redundant instances when active service fails. It is defined three network nodes in our server nodes. Furthermore, three servers are configured as the controller, compute, and network node, and one server just works as a compute node. By default, the MTU size of self-service networks which are connected to qrouter is 1450 bytes, but the MTU size of provider network is set to 1500 bytes.

The performance test is done with default parameters with below commands:

$ iperf3 -f -s

$ iperf3 -f -c VM-IP -o 15 -i 1 -t 60

Scenario 1 (Same network, different compute nodes)

The packet includes destination MAC layer of instance because the destination instance resides on the same network. So, the packet would be sent to the destination through overlay network. Each packet between instances on different hosts is encapsulated on one host and sent to the other host through the VXLAN tunnel. In our production OpenStack servers are connected back-to-back through 10 GB/s NICs. Thus, the network overlay would be established through these back-to-back physical network interfaces. the figure 6.1 demonstrates that around 40 percent drops in throughput could be observed relatively physical machines.

Scenario 2 (Different network, same compute nodes)

In this case, the packet transfer to the network node from underlaying physical network interface through overlay network because the VMs connect to different self-service networks. it is seen that two different throughputs are gained. If the qrouter resides on the same compute nodes where VMs are. From figure 6.1, it could be seen that very high throughput and low latency because it is used centralized routing and the virtual router resides in the same compute node. So, all the process of routing and the packets transferred through bridge and virtual router on the same compute node. In this case, it does not go to the underlaying physical network interface. On the other hand, If the virtual router exists on different compute nodes where VMs are, it could be observed that the throughput decreased dramatically in compare to the first observation. Also, the throughput is lower than scenario 1, because it must perform one other step which is routing the packet for changing vni tag.

Scenario 3 (Same network, same compute node)

In this scenario, it could be seen that based on figure 6.1 very high throughput and low latency than other cases because the packet passes through Linux bridge and it does not depend on virtual router and underlay network. Also, the throughput relies on the virtio device that the hypervisor is used, so it exposes through the virtual PCI port bus.

Scenario 4 (Floating IP/ Fixed IP traffic)

In this case, it could be observed that very low throughput and high latency compared to other scenarios due to packet transfers from VM to the physical underlaying network interface and then goes to network node via overlay network. Because the packet is sending to an external network such as the internet, it should perform SNAT on the packet on the virtual router and changing the

source IP address with the gateway IP address of the provider network. So, the router forwards the packet to the provider bridge router port and sends it to the physical network interface. Thus, the packet transfers from the physical network interface to the Internet. All these steps from overlay to the underlaying network would be degraded the throughput and increase latency considerably.

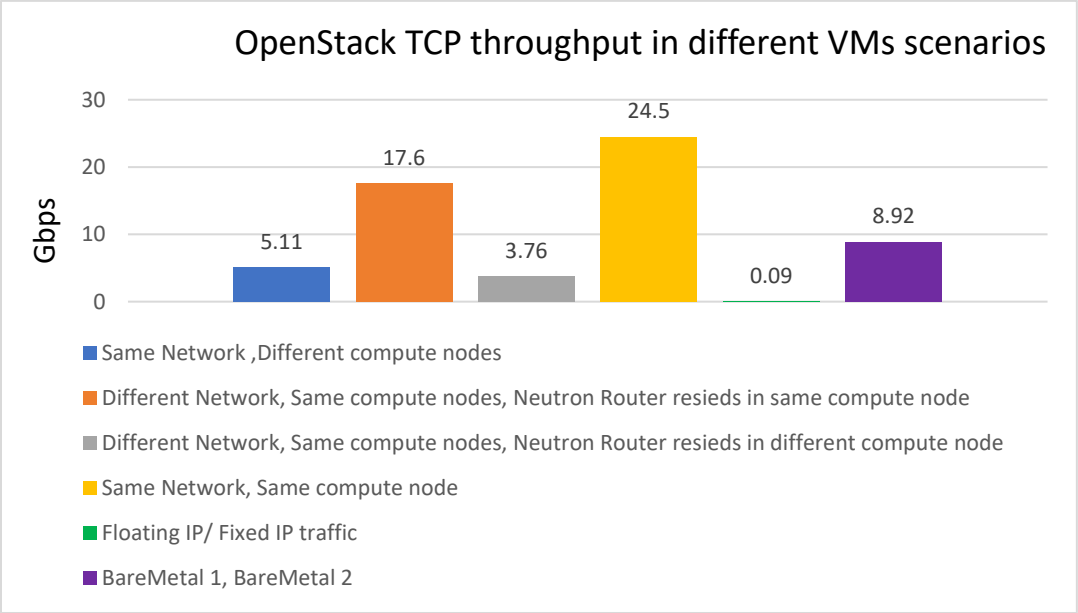

Figure 6. 1: OpenStack TCP throughput in different VMs scenarios

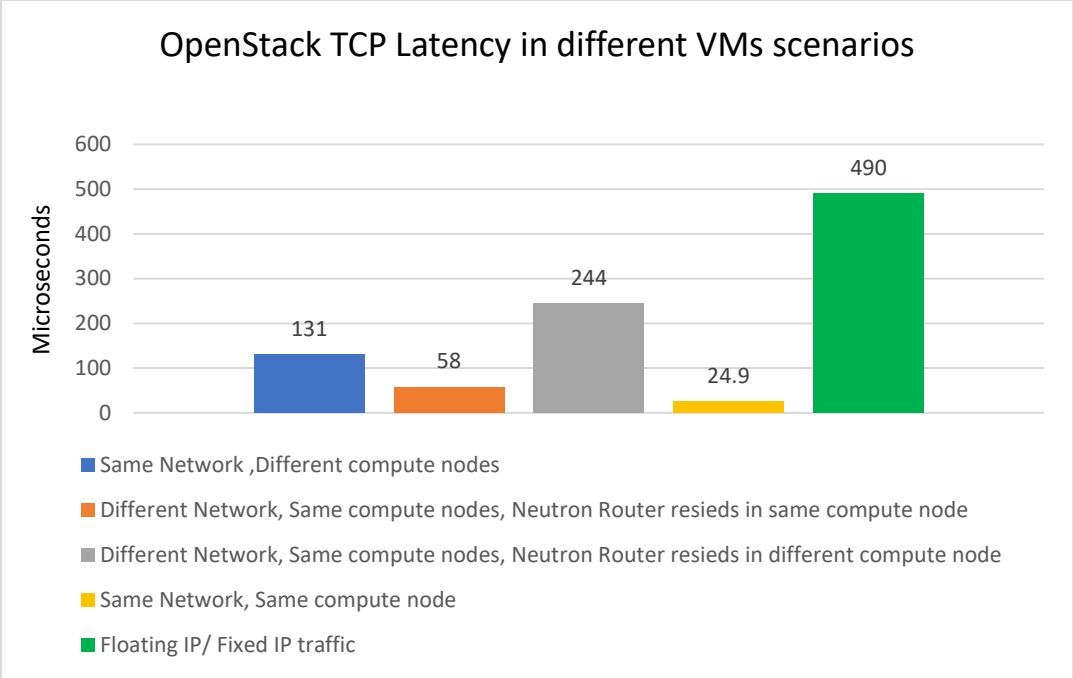

Figure 6. 2: OpenStack TCP Latency in different VMs scenarios

6.2. KubeVirt network Performance measurements

The software which is used in our testing environment is based on the following:

- Kubernetes v1.16.1
- KubeVirt v0.22.0
- Container Network Interface (CNI) plugin: flannel
- Hypervisor: QEMU

In our Kubernetes production cluster, it is used 1GB/s NIC for internet and 40GB/s NICs and a Cisco Switch 40GB/s is for communicating of nodes that are employed. Because of sharing the production cluster with other people, it was not to be able to change the configuration and CNI plugin selected the 1GB/s NIC as main NIC for assigning the IP address to its master and worker nodes, its nodes traffic passing from 1GB/s NIC. So, we could not test on the 40GB/s NIC cards.

The performance tests are performed based on two different scenarios:

Scenario 1(Same Subnet, Same Node)

In this case, figures 6.3 and 6.4 show that high throughput and low latency because two VMs reside on the same subnet in which the traffic flows pass through cni0 Linux bridge in the same node. Also, the throughput depends on the bus speed that the hypervisor uses the virtio device that exposes it to the virtual PCIe port.

Scenario 2 (Different Subnet, Different Node)

In this scenario, the packet sends to destination VM through VXLAN tunneling because the VMs reside on different subnets. The flannel.1 bridge would be sent the packet to the destination VM based on the VXALN tunnel endpoint (VETP) IP address of the destination which gains from the flannel database. It would be observed that VM throughput on KubeVirt is very good which is almost near the physical machine throughput with very low virtual machine overhead on network performance.

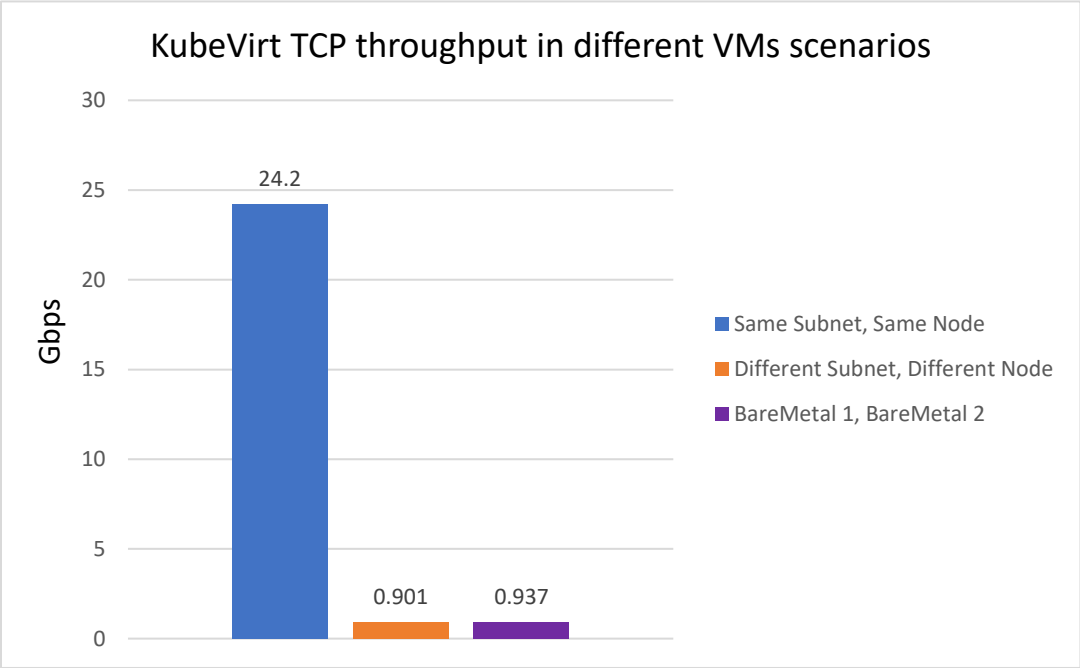

Figure 6. 3: KubeVirt TCP throughput in different VMs scenarios

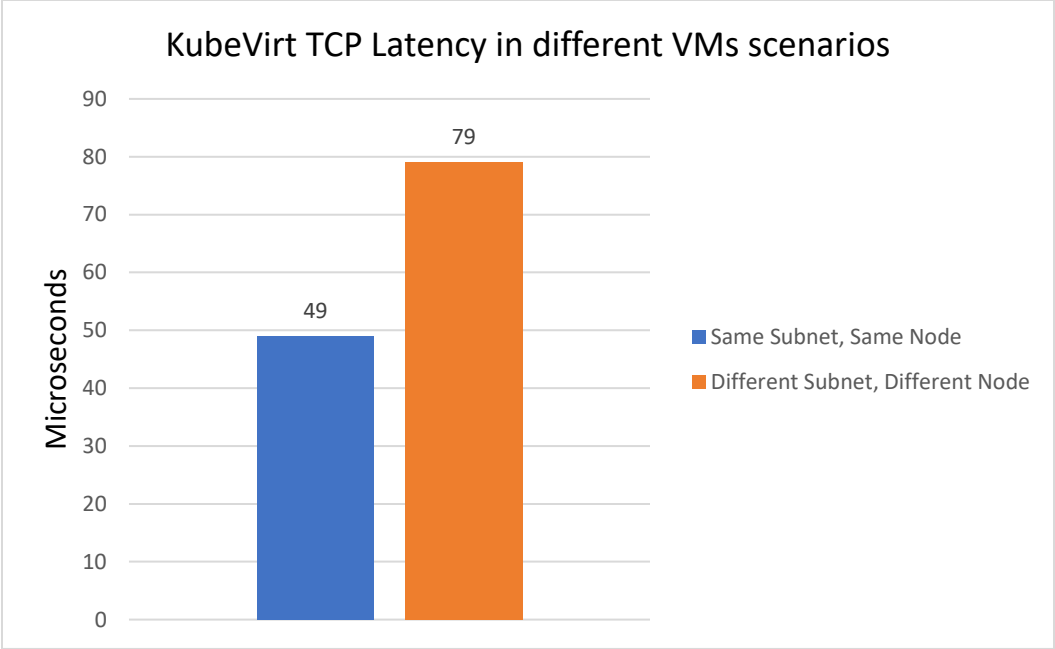

Figure 6. 4: KubeVirt TCP Latency in different VMs scenarios

.

Chapter 7
Conclusion

In this work, it is studied the general architecture and more concentrating on network architecture of Kubernetes, KubeVirt, and OpenStack. The traffic flow patterns on different VM scenarios both on OpenStack and KubeVirt platforms were investigated. Furthermore, it is performed network performance tests on different cases for finding out the effects of virtual machines overhead on network performance based on OpenStack and Kubevirt. The results showed that the location of virtual machines in term of compute node and network indicate different network performance. Therefore, when VMs reside on the same compute node and the same network, the network performance is better than other scenarios because the transmission path is shorter than other cases and it uses Layer 2 communicate. Each of these technologies has some advantages and disadvantages against others. It is observed that on OpenStack, we have lots of network overhead in case of VMs resides on different compute node which affects network throughput around fifty percent mitigation than network performance of physical machines. There are several solutions and different mechanisms that could tune the OpenStack network performance to improves the throughput up to near performance of BareMetal machine. One of the significant advantages of OpenStack rather than KubeVirt is the integrated web dashboard which provides complete visualized management services for both admins and users. Moreover, KubeVirt can provide virtual machines in the combination of virtualized workloads with new container workload on the one platform. But, one of the most limitations that could be seen on the KubeVirt is the difficult management of virtual machines and accessing each user from outside of Kubernetes cluster to VMs. On the other hand, it was observed that very good network throughput on the VMs of KubeVirt.

there is an alternative project called Kuryr [24] which is an OpenStack project that enables native Neutron-based networking in Kubernetes. With Kuryr-Kubernetes it is now possible to choose to run both OpenStack VMs and Kubernetes Pods on the same Neutron network. Using the Kuryr, COE can use neutron features. The OpenStack neutron manages all the network such as network port creation and load balance handling, and Kubernetes node allocate IPs after creation ports on the Kubernetes Pod based on the information assigned by the neutron. Moreover, it provides various network features such as VXLAN, VLAN, Flat based on virtual switches.

Bibliography

[1] (Kumar, Rakesh, et al. “Open source solution for cloud computing platform using OpenStack.” International Journal of Computer Science and Mobile Computing, vol. 3(5), pp. 89-98, May. 2014.).

[2] Bernstein, David. “Containers and cloud: From lxc to docker to kubernetes.” IEEE Cloud Computing, vol. 1(3), pp. 81-89, Sept. 2014.

[3] https://kubernetes.io.

[4] https://kubernetes.io/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive.

[5] https://kubernetes.io/docs/concepts/overview/components/#node-components

[6] https://medium.com/google-cloud/understanding-kubernetes-networking-pods-7117dd28727

[7] https://itnext.io/an-illustrated-guide-to-kubernetes-networking-part-1-d1ede3322727

[8] https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/

[9] M.Hausenblas, “Container Networking”, O'Reilly Media, May 2018

[10] https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/

[11] https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html

[12] https://metallb.universe.tf/

[13] https://kubernetes.io/blog/2018/05/22/getting-to-know-kubevirt/

[14] https://kubevirt.io/2018/KubeVirt-Network-Deep-Dive.html

[15] https://kubevirt.io/2018/containerized-data-importer.html

[16] https://www.learningjournal.guru/article/public-cloud-infrastructure/what-is-bastion-host-server/

[17] https://www.openstack.org/software/

[18] https://docs.openstack.org/cs/install-guide/get-started-logical-architecture.html

[19] https://access.redhat.com/documentation/en US/Red_Hat_Enterprise_Linux_OpenStack_Platform/6/html-single/Component_Overview/index.html#section-blockStorage

[20] https://docs.openstack.org/security-guide/networking/architecture.html

[21] https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/10/html/networking_guide/sec-dvr

[22] https://www.redhat.com/en/blog/introduction-virtio-networking-and-vhost-net

[23] https://docs.openstack.org/neutron/rocky/admin/deploy-lb-selfservice.html

[24] https://docs.openstack.org/kuryr-kubernetes/latest/index.html