**Networking Simulation Final Report**

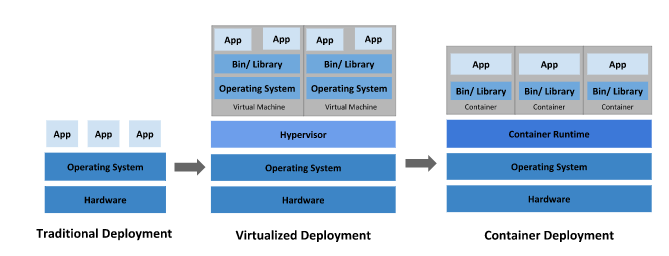
**A Multi-Client**

Operatings system : Ubuntu 20.04

Docker Version:19.03.8

Kubernetes: v1.20.4

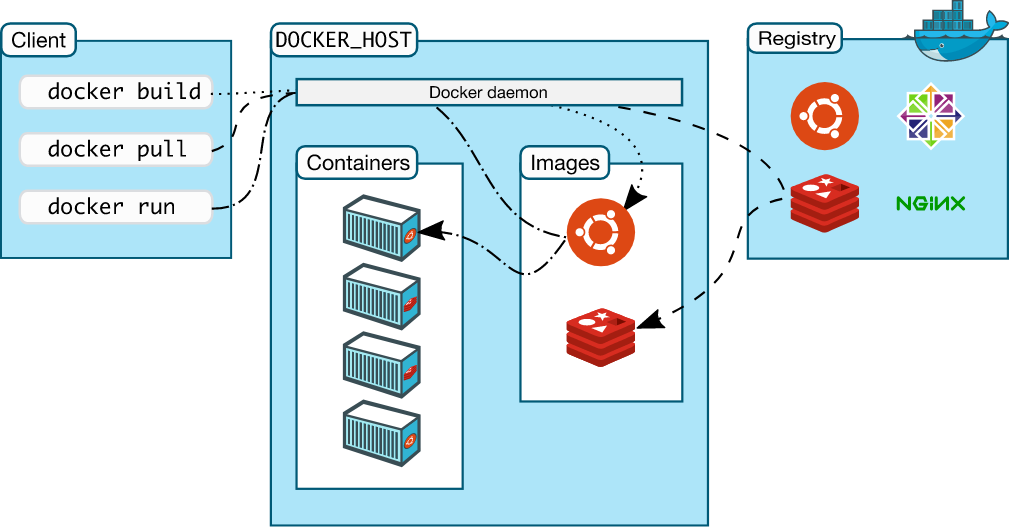
Supporting api: python 3, ssh, ansible, go



Why kubernetes for the network simulation?

Containers provide an easy way to bundle and run applications, along with kubernetes which is an open-source Container orchestration-system. Kubernetes provides a management system for containers that ensure to complete observation of the container environment. It has the functionality to scale, ensure there is no failover of your application, and provides the necessary deployment configurations, this ensures there is no downtime on your containers.[0]

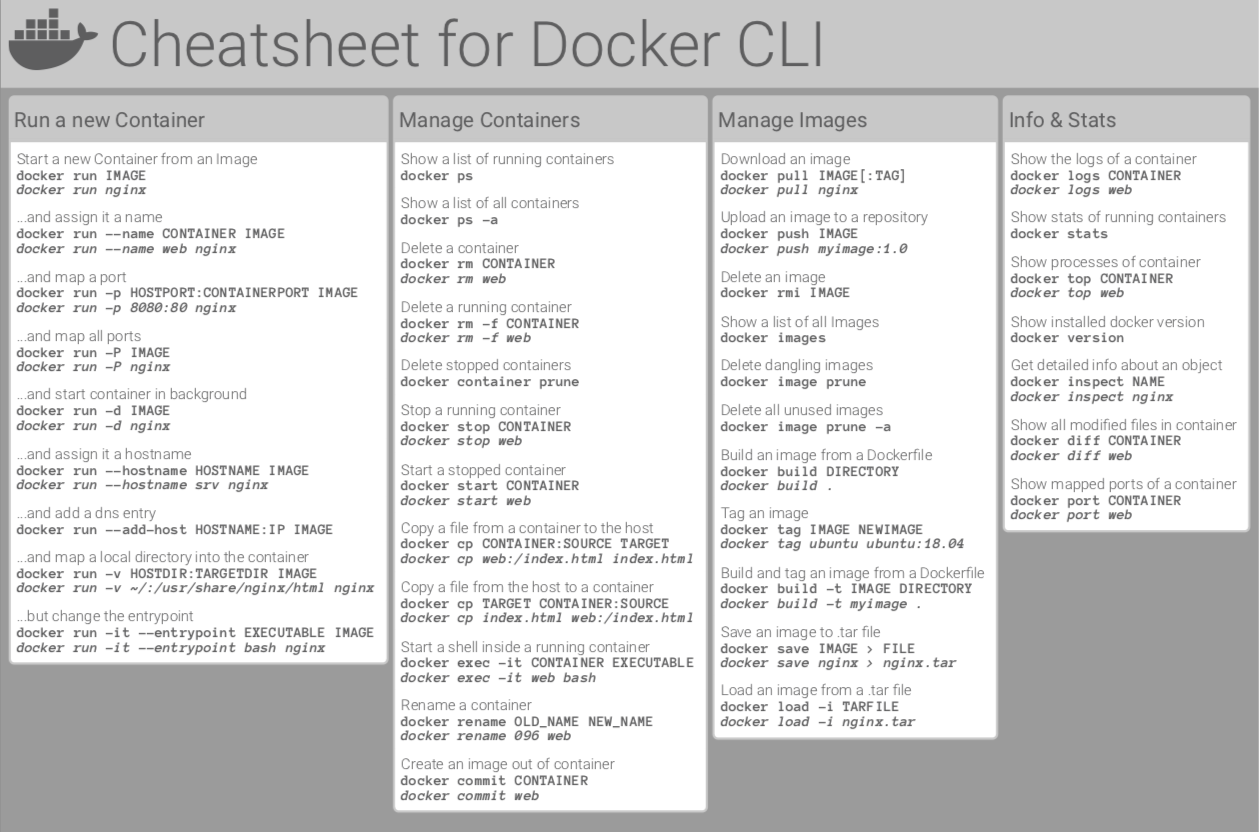
**Docker**



Docker is an open source containerized platform that allows the ability to quickly build, test, and deploy applications inside containers. A container represents a runtime for a single application and includes everything the software needs to run. Only sharing the OS Kernel, These isolated containers require far less amounts of resources. The container images on the other hand are a read-only template that the containers take the build instructions from. They do not have states.

Docker Installation

This tutorial covers how to install Docker on an Ubuntu 20.04 machine.  
<https://docs.docker.com/engine/install/ubuntu/>

It is beneficial to try out a few commands to get yourself familiar with it. As you test on how to build different containers, it is easy to see why they have become so popular in the cloud services. Containers can be quickly deployable on any one platform whether it be the server, the cloud environment or to the developer. Moreover with the integration of Kubernetes, docker is able to be easily scaled, orchestrated and deployed.

Next we need to configure docker to run on system start up and as a manage its permissions.  
  
- Run these commands to add docker to sudo, After restart to ensure proper memberships have been established. [2]

$ sudo groupadd docker

$ sudo usermod -aG docker $USER

* To verify run docker run hello-world

If an error occurs it is likely due to dockers configuration file permissions already created, to fix this issue run.

$ sudo chown "$USER":"$USER" /home/"$USER"/.docker -R

$ sudo chmod g+rwx "$HOME/.docker" -R

Lastly let's configure docker to run on system start-up

$ sudo systemctl enable docker.service

$ sudo systemctl enable containerd.service

For additional information and troubleshooting docker docs provides a [post installation guide](https://docs.docker.com/engine/install/linux-postinstall/) that goes more in depth.[3] There you can configure firewall access, set up ipv6 for the daemon or manage DNS servers for docker.

VrNetlabs

Notes: verify KVM is installed

VrNetlabs allows you to run different virtual routers within a containerized platform. Every router supported by vrnetlab has unique configuration and setup procedures. For the [OpenWRT](https://openwrt.org/) router, the vnetlab author created a makefile that will download the latest version of OpenWRT to the *vrnetlab/openwrt* directory and then build an OpenWRT Docker image.

Installing Vrnetlabs

To install vrnetlab, clone the [vrnetlab repository](https://github.com/plajjan/vrnetlab) from GitHub to your system. In this example, I cloned the repository to my home directory, as follows:

$ cd $HOME

$ git clone <https://github.com/plajjan/vrnetlab.git>

Change directory to *vrnetlab’s openwrt directory*:

$ cd ~/vrnetlab/openwrt

$ sudo make build

Note: Make build automatically pulls the image from the openWRT website and builds it into a docker image that later on can be deployed. If by chance it doesn’t, you will need to use:

$ wget <openwrt url link to download the image example: <https://downloads.openwrt.org/releases/18.06.9/targets/x86/64/openwrt-18.06.9-x86-64-combined-ext4.img.gz>>

Unzip the file using some software such as gunzip. Once you have the image in the openwrt directory then run:

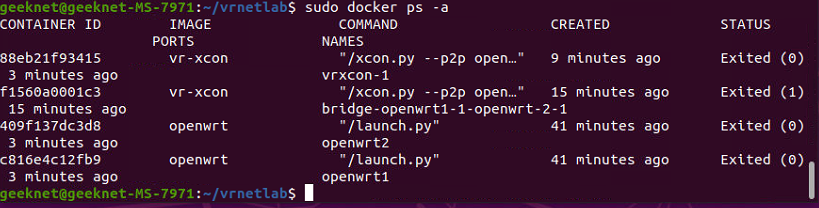
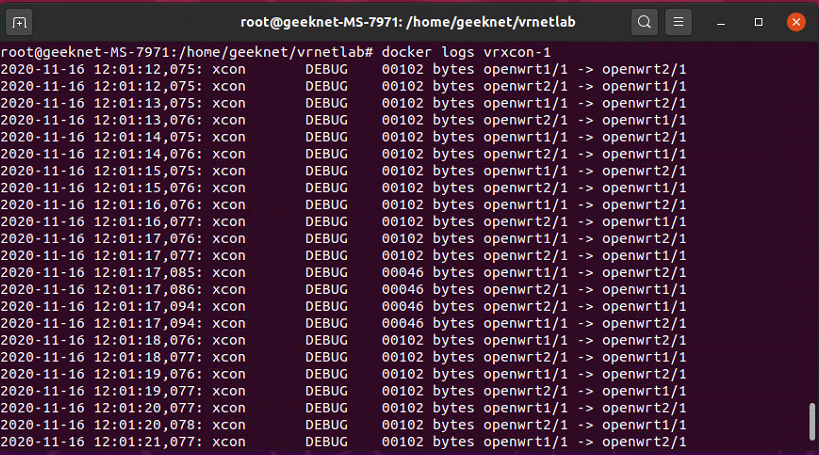
$ sudo make docker-image

This should produce a message stating that there was a successful build with a successful tag to vrnetlabs docker image. The docker image is named *vrnetlab/vr-openwrt:<some version>*. See to all docker images use:

$ sudo docker images

REPOSITORY TAG IMAGE ID CREATED SIZE

vrnetlab/vr-openwrt 18.06.2 0c0eef5fb556 20 seconds ago 545MB

If you want to test the connectivity and how vrnetlab’s openWRT routers work. You can follow [Brianlinkletter's walkthrough](https://www.brianlinkletter.com/2019/03/vrnetlab-emulate-networks-using-kvm-and-docker/)[4] that shows how to implement a virtual connection between two containers. Below is an example of how I deployed two openwrt containers from the image i pulled previously and then bridged them using the using xcon, which is another container that allows the connection between two vrnetlab containers through the routers tcp interface.

**Kubernetes**

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.[5]

Kubernetes deploys a control plane when you initialize the cluster, along with any workers nodes you may deploy with the control plane specified in your configuration file. Kubernetes is a fantastic orchestration tool that allows for testing out many networking provisions. Starting with the major components of the cluster will be the control plane, these components are stored inside the control plane as pods.

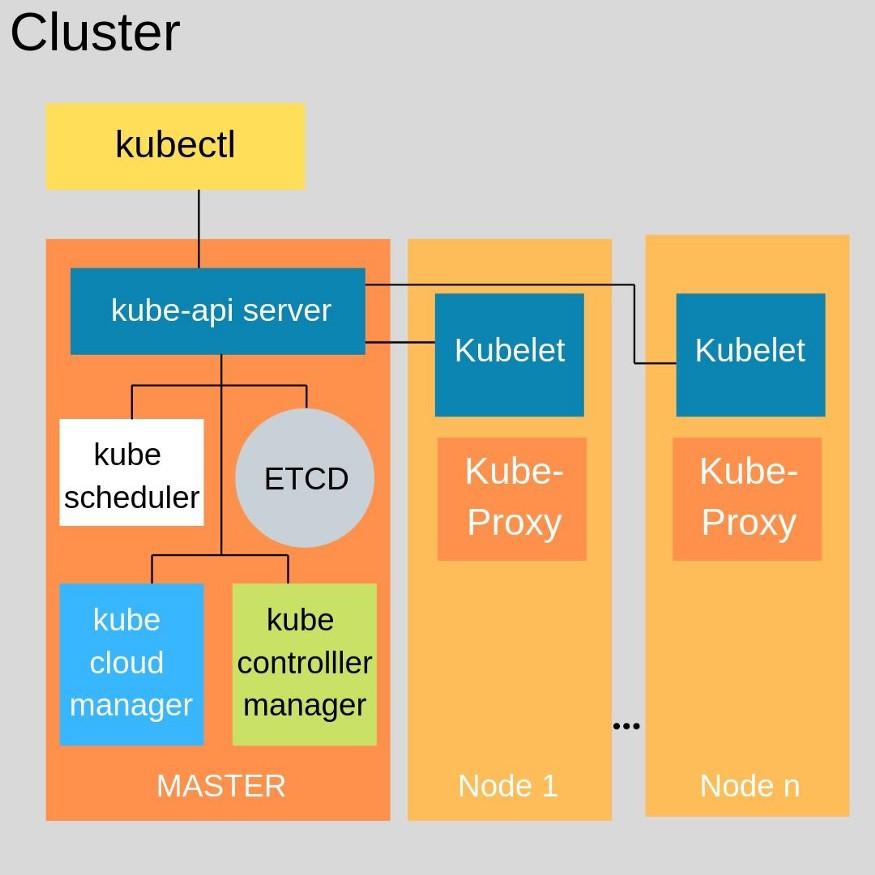
Control Plane Components

kube-apiserver

The Kubernetes API server validates and configures data for the api objects which include pods, services, replicationcontrollers, and others. The API Server services REST operations and provides the frontend to the cluster’s shared state through which all other components interact.

Kube-proxy

The Kubernetes network proxy runs on each node. This reflects services as defined in the Kubernetes API on each node and can do simple TCP,UDP stream forwarding or round robin TCP,UDP forwarding across a set of backends. Service cluster ips and ports are currently found through Docker-links-compatible environment variables specifying ports opened by the service proxy. There is an optional addon that provides cluster DNS for these cluster IPs. The user must create a service with the apiserver API to configure the proxy.



Kube-scheduler

The Kubernetes scheduler is a policy-rich, topology-aware, workload-specific function that significantly impacts availability, performance, and capacity. The scheduler needs to take into account individual and collective resource requirements, quality of service requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, deadlines, and so on. Workload-specific requirements will be exposed through the API as necessary.

kubelet

The kubelet is the primary “node agent” that runs on each node. The kubelet works in terms of a PodSpec. A PodSpec is a YAML or JSON object that describes a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms (primarily through the apiserver) and ensures that the containers described in those PodSpecs are running and healthy.

kube-controller-manager

The Kubernetes controller manager is a daemon that embeds the core control loops shipped with Kubernetes. In applications of robotics and automation, a control loop is a non-terminating loop that regulates the state of the system. In Kubernetes, a controller is a control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state.

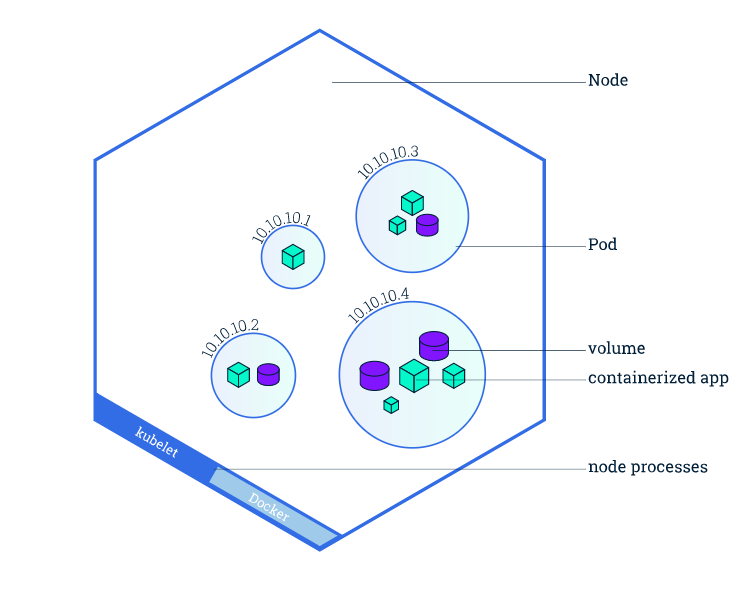
Etcd

etcd is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

### cloud-controller-manager

A Kubernetes [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

Getting to know your node



Within a cluster you can have numerous nodes, with one being a master, the rest being worker nodes, for the network simulation we will only be working with a single node because we aren't distributing this on a cloud service. Rather we are trying to test the topology and understand the network interface within a single node cluster. Thus we need the control plane(master node), and k8s-topo, this will be a kubernetes add-on we talk about later on, to build our network simulation through pods.

Lastly our final piece of the puzzle would be connecting our pods within our master node through meshnet-CNI which will deploy the virtual connections between the pods based on k8s-topo topology.

\*Important Note\*

|  |  |
| --- | --- |
| Ensure you have swapoff disabled  $ sudo swapoff -a | To permanently disable swapoff upon system start, run:  $ sudo sed -i '/ swap / s/^\(.\*\)$/#\1/g' /etc/fstab |

Why is disabling swap off important?

Support for swap is non-trivial. Guaranteed pods should never require swap. Burstable pods should have their requests met without requiring swap. BestEffort pods have no guarantee. The kubelet right now lacks the smarts to provide the right amount of predictable behavior here across pods.As of today, the scheduler does not understand swap configuration and usage to provide smarts for swap support.[8]

Local host installation of kubernetes

First lets install all the necessary packages needed for kubernetes

$ sudo apt-get update

$ sudo apt-get install -y apt-transport-https ca-certificates curl

Download the Google Cloud public signing key:

$ sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg https://packages.cloud.google.com/apt/doc/apt-key.gpg

Add the Kubernetes apt repository:

$ echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc/apt/sources.list.d/kubernetes.list

Install kubectl, kubelet, and kubeadm:

$ sudo apt-get install kubeadm kubelet kubectl

Once all of our necessary packages are installed we can begin to develop our cluster.

Initialize the cluster and its control plane by running the following:

$ kubeadm init

To make kubectl work for your non-root user, run these commands.:

mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown **$(**id -u**)**:**$(**id -g**)** $HOME/.kube/config

Note:

To create a proxy to your cluster run:

$ kubectl proxy --port=8080 &

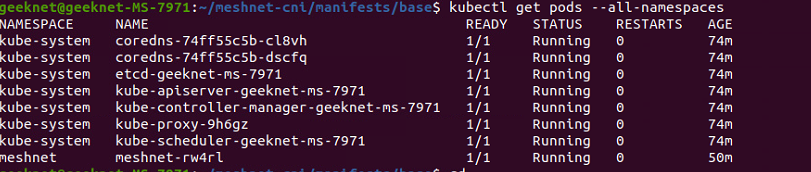
Kubernetes Cheat Sheet

<https://github.com/dennyzhang/cheatsheet-kubernetes-A4>

Once you have initialized your cluster and set up the configuration files we can run the following to see the pods we have at the moment.

$ kubectl get pods --all-namespaces

You should notice the namespace kube-system which is essentially the namespace for the essential control plane components. You might also notice that the *coredns* are not yet running. This is because we currently do not have a Container Network Interface(CNI) deployed inside our cluster. As described before we will be working with Meshnet-CNI. This works well with our topology builder k8s-topo.



Kubernete objects

In order to provide kubernetes with an object you must provide k8s(kubernetes) with object specs that tell the orchestration system the information needed in order to create the object.

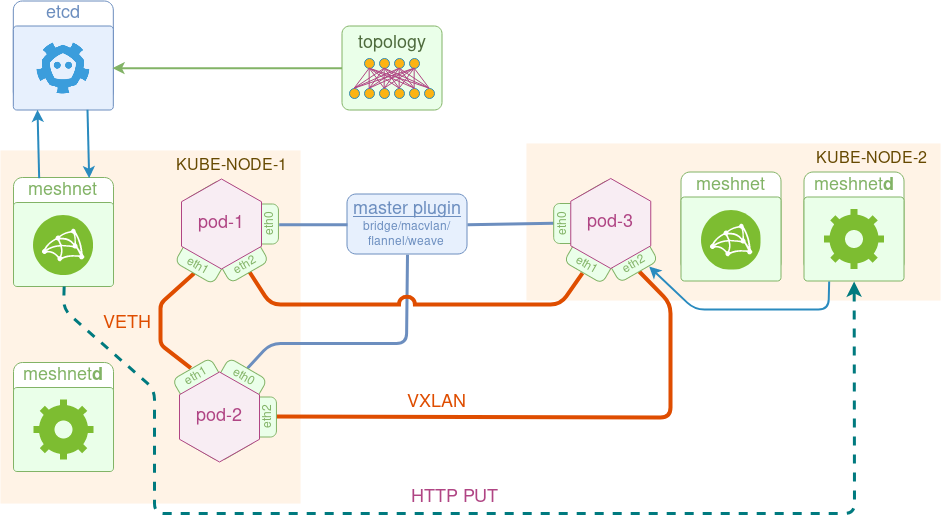
In order to create the desired object such as our meshnet-CNI or k8s-topo we need to send the kubernetes API some yaml file(these files are kubernetes main file input) with at most the required information:

* apiVersion - Which version of the Kubernetes API you're using to create this object
* kind - What kind of object you want to create
* metadata - Data that helps uniquely identify the object, including a name string, UID, and optional namespace
* spec - What state you desire for the object

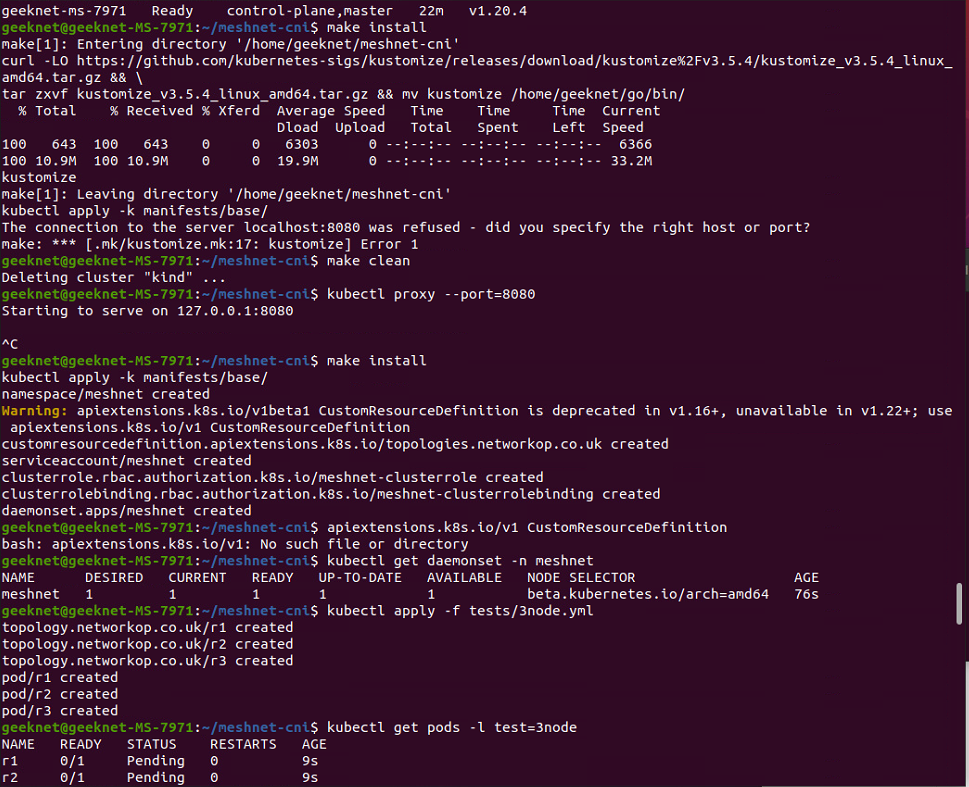
Kubectl takes the yaml file and sends it to the API to process the data. At which point the control plane will begin to implement the data you had specified.

Meshnet-CNI

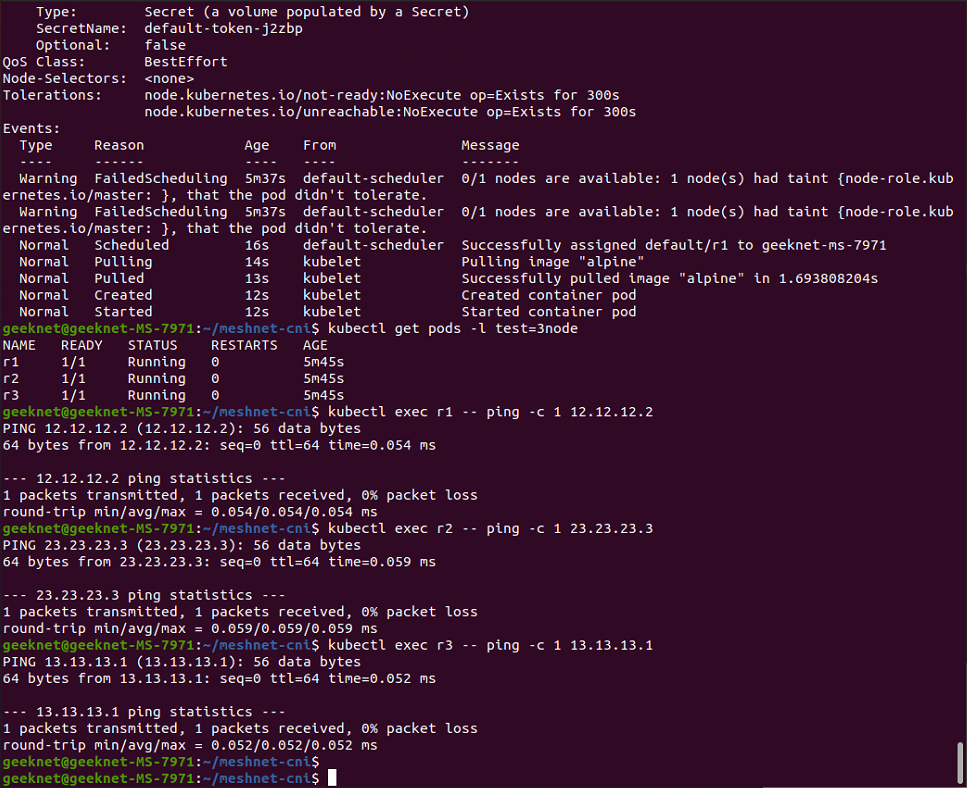
A CNI, Container Network Interface, is used to orchestrate the network between pods. It takes a container runtime and connects them to a network. This allows for the pods of our topology builder to communicate with one another through a virtual connection.[6]



For us this will be meshnet-CNI which creates arbitrary network topologies out of point-to-point links with the help of [koko](https://github.com/redhat-nfvpe/koko).[7] Interconnects these pods through the defined topology.

In the images below I applied a topology defined by the three node yaml file. Which deployed three additional pods to my cluster.

Next we pinged the pods to verify that the connectivity between them was established.



Installing Meshnet-CNI

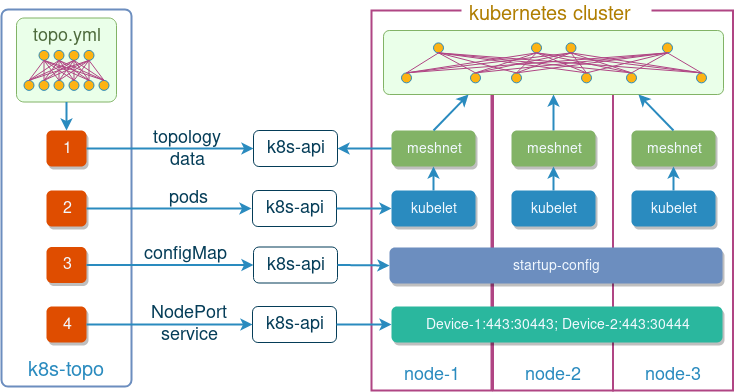
Retrieve the git repository from github by running the following:

$ cd $HOME

$ git clone https://github.com/networkop/meshnet-cni.git && cd meshnet-cni

Now we need to navigate to our manifests .yml file. This will give our cluster the object information to deploy the plugin. It will create a namespace for the meshnet interface, a resource definition for the network topology, privileges to meshnet to work with other resources and a daemonset with the plugin and configuration files.

$ kubectl apply -f /manifests/base/meshnet.yml

K8s-topo[8]

K8s-topo provides our CNI with the tology to build our network. It uses an arbitrary network topology builder for network simulations without our cluster. This also ties into our vrnetlabs container image, as we can use that image to create a virtual openwrt router within a pod. Once we have a topology defined with the router we can begin to test different configurations.

Installing k8s-topo and testing

$ cd $HOME

git clone https://github.com/networkop/k8s-topo.git && cd k8s-topo

Deploy k8s-topo pod

$kubectl create -f manifest.yml

Log into the k8s-topo pod.

$ kubectl exec --stdin --tty <pod name> -- /bin/sh

Create a random 20-node network topology( at this point you should be inside the k8s-topo pod.

/k8s-topo # ./examples/builder/builder 20 0

Total number of links generated: 19

Create the topology inside K8s

/k8s-topo # k8s-topo --create examples/builder/random.yml

Optionally, you can generate a D3.js network topology graph

/k8s-topo # k8s-topo --graph examples/builder/random.yml

View the generated topology graph at *http://<any\_k8s\_cluster\_node\_ip>:32000*

Verify that the topology has been deployed (from the master node)

/k8s-topo # kubectl get pods -o wide | grep qrtr

qrtr-1 1/1 Running 0 11s 10.233.65.231 node3 <none>

qrtr-10 1/1 Running 0 11s 10.233.65.234 node3 <none>

qrtr-11 1/1 Running 0 10s 10.233.66.246 node4 <none>

Destroy the topology and exit the pod by running:

/k8s-topo # k8s-topo --destroy examples/builder/random.yml  
/k8s-topo # exit

Extra steps.

Kubernetes Dashboard

<https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>

Creating a simple user to access the dashboard

<https://github.com/kubernetes/dashboard/blob/master/docs/user/access-control/creating-sample-user.md>

Appendices

[0]

<https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

[1]  
<https://docs.docker.com/get-started/overview/>

[2]

<https://dockerlabs.collabnix.com/docker/cheatsheet/>

[3]

<https://docs.docker.com/engine/install/linux-postinstall/>

[4]

<https://www.brianlinkletter.com/2019/03/vrnetlab-emulate-networks-using-kvm-and-docker/>

[5]

<https://kubernetes.io/docs/concepts/overview/>

[6]

<https://github.com/networkop/meshnet-cni>

[7]

<https://github.com/redhat-nfvpe/koko>

[8]

<https://github.com/networkop/k8s-topo>