TP Kubernetes

Kubernetes is an open-source container management system, freely available. It provides a platform for automating deployment, scaling, and operations of application containers across clusters of hosts. Kubernetes gives you the freedom to take advantage of on-premises, hybrid, or public cloud infrastructure, releasing organizations from tedious deployment tasks.

In this pratical class, we are going to:

- setup multi-node Kubernetes Cluster on Ubuntu 20.20 server;

- deploy an application and manage it on our deployed Kubernetes;

Prerequisites

- Two virtual machines or physical machines, known as nodes, with ubuntu 20.20 server installed.
- A static IP address (masterIP) is configured on the first nodes (master node) and also a static IP (slaveIP) is configured on the second instance (slave node).
- Minimum 4 GB RAM and 2 vCPU per node.
- root password is setup on each instance "toto".

Connection to your nodes

Use the following commands to access your server :

The master node port is 130XX and the slave node 130XX+1

```
ssh ubuntu@147.127.121.1 -p 130XX #connection to master node
ssh ubuntu@147.127.121.1 -p 130XX+1 #connection to slave node
```

Where XX=01-40. This will give you acces to a VM with IP address 192.168.27.(XX+10) and the password is "*toto*"

```
sudo bash
apt-get update -y # On both node
```

Node configurations

You need to configure “hosts” file and hostname on each node in order to allow a network communication using the hostname. You begin by setting the master and slave node names.

On the master node run:

```
hostnamectl set-hostname master
```

On the slave node run :

```
hostnamectl set-hostname slave
```

You need to configure the hosts file. Therefore, run the following command on both nodes:

```
echo "slaveIP   slave" >> /etc/hosts
echo "masterIP  master" >> /etc/hosts
```

You have to disable swap memory on each node. kubelets do not support swap memory and will not work if swap is active. Therefore, you need to run the following command on both nodes:

```
swapoff -a
```

Docker installation

Docker must be installed on both the master and slave nodes. You start by installing all the required packages.

```
wget -qO- https://get.docker.com/ | sh
```

Kubernetes installation

Next, you will need to install: kubeadm, kubectl and kubelet on both nodes.

```
modprobe br_netfilter
```

```
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
```

```
sysctl --system
```

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |
sudo apt-key add -
```

```
cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
```

```
apt-get update
apt-get install -y  kubelet kubeadm kubectl
```

Good, all the required packages are installed on both servers.

We need to configure kubernetes on both nodes to use the correct docker driver.

As a root user, open the file *"/etc/systemd/system/kubelet.service.d/10-kubeadm.conf*" and edit the "KUBELET_CONFIG_ARGS" configuration, i.e add " --cgroup-driver=cgroupfs " as an argument, then reload systemd

```
systemctl daemon-reload
```

NB: This operation has to be carry-out on both nodes, i.e slave and master

Master node configuration

Now, it's time to configure Kubernetes master Node. First, initialize your cluster using its private IP address with the following command:

```
rm /etc/containerd/config.toml
systemctl restart containerd
kubeadm init --pod-network-cidr=10.244.0.0/16
```

Note:

- *pod-network-cidr* = specify the range of IP addresses for the pod network.

You should see the following output:

```
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/
```

```
Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.27.11:6443 --token 81gol4.fcbhty15p7x5vbeg \
    --discovery-token-ca-cert-hash sha256:8a8e4f06afdcdddca4f8a64f19fb2c9dd79f2ccea32a1ef24c13f2d9c70341ad
```

You have to save the 'kubeadm join ... ...' command. The command will be used to register new worker nodes to the kubernetes cluster.

To use Kubernetes, you must run some commands as shown in the result (as root).

```
mkdir -p $HOME/.kube
cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
chown $(id -u):$(id -g) $HOME/.kube/config
```

We can check the status of the master node by running the following command:

```
kubectl get nodes
kubectl get pods --all-namespaces
```

you can observe form the above output that the master node is listed as not ready. This is because the cluster does not have a Container Networking Interface (CNI).
Next, deploy the flannel network to the kubernetes cluster using the kubectl command.

```
kubectl apply -f https://raw.githubusercontent.com/coreos
/flannel/master/Documentation/kube-flannel.yml
```

Wait for a minute and check kubernetes node and pods using commands below.

```
kubectl get nodes
kubectl get pods --all-namespaces
```

You should see the following output:

```
root@tuvm1:/home/ubuntu# kubectl get pods --all-namespaces
NAMESPACE     NAME                              READY   STATUS    RESTARTS   AGE
kube-system   coredns-f9fd979d6-rmrsz           1/1     Running   0          2m33s
kube-system   coredns-f9fd979d6-xqf7t           1/1     Running   0          2m33s
kube-system   etcd-master                       1/1     Running   0          2m41s
kube-system   kube-apiserver-master             1/1     Running   0          2m41s
kube-system   kube-controller-manager-master    1/1     Running   0          2m41s
kube-system   kube-flannel-ds-7mwrx             1/1     Running   0          78s
kube-system   kube-proxy-2md8m                  1/1     Running   0          2m33s
kube-system   kube-scheduler-master             1/1     Running   0          2m41s
root@tuvm1:/home/ubuntu#
```

And you will get the 'kube-scheduler-master' node is running as a 'master' cluster with status 'ready'.

Kubernetes cluster master initialization and configuration has been completed.

Slave node configuration

Next, we need to log in to the slave node and add it to the cluster. Remember the join command in the output from the Master Node initialization command and issue it on the Slave Node as shown below:

```
sudo kubeadm join ``xxx``.``xxx``.``xxx``.``xxx``:6443 --token
wg42is.1hrm4wgvd5e7gbth     --discovery-token-ca-cert-hash
sha256:53d1cc33b5b8efe1b974598d90d250a12e61958a0f1a23f864579dbe
67f83e30
```

Once the Node is joined successfully, you should see the following output:

```
This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

root@tuvm2:/home/ubuntu#
```

Now, go back to the master node and run the command “kubectl get nodes” to see that the slave node is now ready (You may wait for some second for the node to be in ready state):

```
kubectl get nodes
```

Apache2 deployment (some fun)

We will deploy a little application in our Kubernetes cluster, apache2 web server with a simple index.php application. We will use the YAML template.

Create a new directory named 'apache' and go to that directory.

```
mkdir -p apache/
cd apache/
```

You should create a container image named apache-image from a Dockerfile, and this image should host a php application named app.php.

Dockerfile content

```
FROM php:7.2-apache
COPY ./app.php /var/www/html/
```

Here is the content of app.php

```
<!DOCTYPE html>
 <html>
     <head>
         <title>TP Cloud EC2</title>
     </head>
     <body>
         <h1>It works!</h1>

 <?php
 echo gethostname();
 ?>

     </body>
 </html>
```

I considered you can create apache-image container image based on what we did on previous lab-works. Create the image on the slave node only. After the creation on the node, use this set of commands to create a Docker Registry and register you Docker image. On the slave.

```
docker tag apache-image localhost:5000/ubuntu
docker run -d -p 5000:5000 --restart=always --name local-
registry registry:2
docker push localhost:5000/ubuntu
```

NB: this docker image should be create on both nodes, slave and master.

Now, you have to create the apache Deployment YAML file 'apache-deployment.yaml' and paste the following content.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache-deployment
spec:
  selector:
    matchLabels:
      run: apache-app
  replicas: 2
  template:
    metadata:
      labels:
```

```
        run: apache-app
    spec:
      containers:
      - name: apache-container
        image: localhost:5000/ubuntu
        ports:
        - containerPort: 80
```

Note:

- We're creating a new 'Deployment' named 'apache-deployment'.
- Setup the app label as 'apache-app' with 2 replicas.
- The 'apache-deployment' will have containers named 'apache-container', based on 'apache-image' docker image, and will expose the default HTTP port 80.

You can submit the deployment by running the kubectl command below.

```
kubectl create -f apache-deployment.yaml
```

After creating a new 'apache-deployment', check the deployments list inside the cluster.

```
kubectl describe deployment apache-deployment
```

Check the nodes the Pod is running on:.

```
kubectl get pods -l run=apache-app -o wide
```

Check your pods' IPs:

```
kubectl get pods -l run=apache-app -o yaml | grep podIP
```

We need to create a new service for our 'apache-deployment'. Therefore, create a new YAML file named 'apache-service.yaml' with the following content.

```
apiVersion: v1
kind: Service
metadata:
  name: apache-service
  labels:
    run: apache-app
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: apache-app
```

Note:

- We're creating a new kubernetes service named 'apache-service'.
- The service belongs to the app named ''apache-app' based on our deployment 'apache-deployment'.

To list the pods created by the deployment:

```
kubectl get pods -l app=apache-app
```

Create the Kubernetes service using the kubectl command below.

```
kubectl create -f apache-service.yaml
```

Now check all available services on the cluster and you will get the 'apache-service' on the list, then check details of service.

```
kubectl get service
kubectl describe service apache-service
```

Accessing the service

```
kubectl scale deployment apache-deployment --replicas=0
kubectl scale deployment apache-deployment --replicas=2
kubectl get pods -l run=apache-app -o wide
```

Copy the clusterIP and use it to access your application, index.php

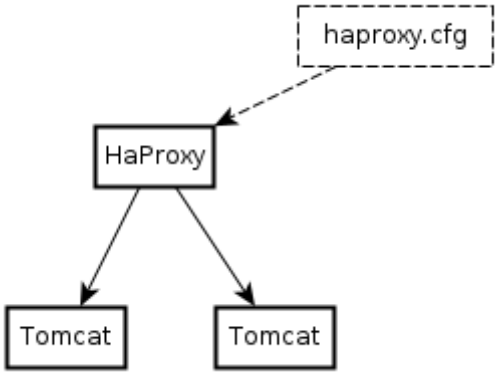
The command should be run from the slave node, not the master. This is a limitation of flannel that just allows accessing the application from slave nodes.

```
curl <clusterIP>/index.php
```

You can observe that there is a load balancing by app-service on the containers deployed.

Apache Deployment

You will demonstrate that you followed the session by deploying the apache architecture of last class in your Kubernetes cluster (2 tomcats instance and 1 service, no need to use the haproxy.cfg file).



Good luck!