# Practice M2: Cluster Setup and Management

For the purpose of this practice, we will assume that we are working on a machine with either a Windows 10/11 or any recent Linux distribution and there is a local virtualization solution (like VirtualBox, Hyper-V, VMware Workstation, etc.) installed

***Please note that long commands may be hard to read here. To handle this, you can copy them to a plain text editor first. This will allow you to see them correctly. Then you can use them as intended***

## Part 1: Basic Cluster Installation

In this part we will focus on the creation of simple yet working **Kubernetes** cluster

### Preparation

We can consult the requirements here:

<https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#before-you-begin>

#### Basic settings

Let’s assume that we have a virtual machine with **Debian 10** installed (basic / minimal)

We will use it to prepare our golden image, that will be used for the creation of the cluster

Log on to the machine (we will assume that we are working with the **root** user)

Check if the **br\_netfilter** module is loaded

**lsmod | grep br\_netfilter**

If not, try to load it

**modprobe br\_netfilter**

Then prepare a configuration file to load it on boot

**cat << EOF | tee /etc/modules-load.d/k8s.conf**

**br\_netfilter**

**EOF**

Adjust a few more network-related settings

**cat << EOF | tee /etc/sysctl.d/k8s.conf**

**net.bridge.bridge-nf-call-ip6tables = 1**

**net.bridge.bridge-nf-call-iptables = 1**

**net.ipv4.ip\_forward = 1**

**EOF**

And then apply them

**sysctl --system**

Check which variant of **iptables** is in use

**update-alternatives --query iptables**

And switch it to the legacy version

**update-alternatives --set iptables /usr/sbin/iptables-legacy**

As a final general step, turn off the SWAP both for the session and in general

**swapoff -a**

**sed -i '/swap/ s/^/#/' /etc/fstab**

#### Container runtime

We will use **Docker** and will follow the steps from the official documentation:

<https://docs.docker.com/engine/install/debian/>

Update the repositories information

**apt-get update**

And install the required packages

**apt-get install ca-certificates curl gnupg lsb-release**

Download and install the key

**curl -fsSL https://download.docker.com/linux/debian/gpg | gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg**

Add the repository

**echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/debian $(lsb\_release -cs) stable" | tee /etc/apt/sources.list.d/docker.list > /dev/null**

Install the required packages

**apt-get update**

**apt-get install docker-ce docker-ce-cli containerd.io**

#### Container runtime configuration

We will refer to this source:

<https://kubernetes.io/docs/setup/production-environment/container-runtimes/>

Create the configuration folder if does not exist

**mkdir /etc/docker**

Then create the configuration file with the following content

**cat <<EOF | tee /etc/docker/daemon.json**

**{**

**"exec-opts": ["native.cgroupdriver=systemd"],**

**"log-driver": "json-file",**

**"log-opts": {**

**"max-size": "100m"**

**},**

**"storage-driver": "overlay2"**

**}**

**EOF**

Reload and restart the service

**systemctl enable docker**

**systemctl daemon-reload**

**systemctl restart docker**

#### Kubernetes components

We will refer to this source:

<https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#installing-kubeadm-kubelet-and-kubectl>

Install any packages that may be missing

**apt-get update**

**apt-get install -y apt-transport-https ca-certificates curl**

Download and install the key

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

Add the repository

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

Update repositories information

**apt-get update**

Check available versions of the packages

**apt-cache madison kubelet**

Should we want to install the latest version, we may use (skip it for now)

**apt-get install -y kubelet kubeadm kubectl**

For a particular version we should use (execute this one)

**apt-get install kubelet=1.21.6-00 kubeadm=1.21.6-00 kubectl=1.21.6-00**

Then exclude the packages from being updated

**apt-mark hold kubelet kubeadm kubectl**

#### Template preparation

Turn off the machine

**shutdown now**

Using the virtualization solution techniques create a template of this machine or its virtual disk

### Cluster creation

We will create a small cluster with three nodes. One will be part of the control plane and the rest will handle any work

#### Virtual infrastructure

Using the virtualization solution techniques create three identical virtual machines each with

* 2 vCPU
* 2 GB+ RAM

Connect them in a way that will allow for Internet access and easier communication with and between them. External/bridged mode will be the best option

During the demo, we will use **192.168.81.0/24**. You should adjust the commands to match your setup

#### Preparation

Start all nodes

Log on the first one and set

* Its IP address, for example **192.168.81.211/24**
* Ifs **FQDN**, for example **node-1.k8s**
* Its **/etc/hosts** file:

**echo "192.168.81.211 node-1.k8s node-1" | tee -a /etc/hosts**

**echo "192.168.81.212 node-2.k8s node-2" | tee -a /etc/hosts**

**echo "192.168.81.213 node-3.k8s node-3" | tee -a /etc/hosts**

Repeat the above steps on the other two machines

#### Cluster initialization (node-1)

Initialize the cluster with

**kubeadm init --apiserver-advertise-address=192.168.81.211 --pod-network-cidr 10.244.0.0/16**

Installation will finish relatively quickly

Copy somewhere the **join** command

To start using our cluster, we must execute the following

**mkdir -p $HOME/.kube**

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

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

Let's check our cluster nodes (just one so far)

**kubectl get nodes**

Note that it appears as **not ready**

Check the pods as well

**kubectl get pods -n kube-system**

Hm, most of the pods are operational, but there is one pair that is not (**CoreDNS**)

Let's check why the node is not ready

**kubectl describe node node-1**

Scroll to top and look for **Ready** and **KubeletNotReady** words

It appears that there isn't any (POD) network plugin installed

We can check here:

<https://kubernetes.io/docs/concepts/cluster-administration/addons/>

And get further details form here:

<https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/#pod-network>

Check here for a list of plugins here

<https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-implement-the-kubernetes-networking-model>

It appears, that by installing a pod network plugin, we will solve both issues

Let's install a POD network plugin

For this demo, we will use the **Flannel** plugin

More information here: <https://github.com/flannel-io/flannel#flannel>

Install it

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

We can watch the progress with:

**kubectl get pods --all-namespaces -w**

After a while both **Flannel** and **CoreDNS** will be fully operational

Press **Ctrl + C** to stop the monitoring

Check again the status of the node

**kubectl get nodes**

It should be operational and ready as well

#### Join nodes (node-2 and node-3)

Log on to **node-2**

Remember the join command that we copied earlier, now it is the time to use it

It should have the following structure:

**kubeadm join [IP]:6443 --token [TOKEN] --discovery-token-ca-cert-hash sha256:[HASH]**

Join the node to the cluster (yours may be different)

**kubeadm join 192.168.81.211:6443 --token 8qu2va.le6ndhtt9mdpbmow \**

**--discovery-token-ca-cert-hash sha256:9d2642aeda7a1c210b26db639bbf0272e4bfa59b895904162b948c055cb39402**

Repeat the same on **node-3**

#### Finalization

Return on **node-1**

And check nodes

**kubectl get nodes**

Show cluster information

**kubectl cluster-info**

Wouldn't it be nice if we were able to control our new server from our host?

Indeed, it would be 😉

Close the session to **node-1**

Navigate to our home folder (on our host) and then to the **.kube** folder

Copy the configuration file (use your actual master/node-1 IP address here)

**scp root@192.168.81.211:/etc/kubernetes/admin.conf .**

Backup the existing configuration if any

**mv ~\.kube\config ~\.kube\config.bak**

Make the copied file the active configuration

**mv .\admin.conf ~\.kube\config**

Ask for cluster information but this time from the host

**kubectl cluster-info**

Check version of our **kubectl**

**kubectl version --client**

And compare it with the one of the cluster

**kubectl version**

As we said last time, +/-1 minor version is acceptable

### Post installation activities

#### Dashboard Installation

Check the latest version and any installation instructions here:

<https://github.com/kubernetes/dashboard>

Deploy the **Dashboard**

**kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.4.0/aio/deploy/recommended.yaml**

Check the pods

**kubectl get pods --all-namespaces**

Try to access the **Dashboard**

**kubectl proxy**

Use this **URL**

<http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/>

We cannot log in as we do not have any valid way of doing it

Stop the **Dashboard** proxy with **Ctrl + C**

Create a file **dashboard-admin-user.yml** with the following content

**apiVersion: v1**

**kind: ServiceAccount**

**metadata:**

**name: admin-user**

**namespace: kubernetes-dashboard**

Create one more file **dashboard-admin-role.yml** with the following content

**apiVersion: rbac.authorization.k8s.io/v1**

**kind: ClusterRoleBinding**

**metadata:**

**name: admin-user**

**roleRef:**

**apiGroup: rbac.authorization.k8s.io**

**kind: ClusterRole**

**name: cluster-admin**

**subjects:**

**- kind: ServiceAccount**

**name: admin-user**

**namespace: kubernetes-dashboard**

Apply both files

**kubectl apply -f dashboard-admin-user.yml**

**kubectl apply -f dashboard-admin-role.yml**

Now, we can list the available secrets

**kubectl -n kubernetes-dashboard get secret**

Identify the one with name ***admin-user-token-xxxxx*** and ask for its details

**kubectl -n kubernetes-dashboard describe secret *admin-user-token-wtpbm***

Copy the token field data

Start the proxy again with

**kubectl proxy**

Navigate to the same **URL**

<http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/>

Use the token from earlier

Explore the **Dashboard**

Once done, close the browser tab and stop the proxy with **Ctrl + C**

#### Deploy a two-pod application

Deploy the producer pod + service (backend part) that we used in the previous module (M1)

**kubectl apply -f producer-pod.yml**

**kubectl apply -f producer-svc.yml**

Let's spin another one to act as an observer

**kubectl apply -f observer-pod.yml**

And connect to it

**kubectl exec -it observer-pod -- sh**

Install the **curl** command

**apk add curl**

Now, check if the service is accessible by name (**producer**)

**curl http://producer:5000**

Now, try the other names (service + namespace & FQDN) of the service

**curl http://producer.default:5000**

**curl http://producer.default.svc.cluster.local:5000**

Notice the name of the pod

Exit the observer session

**exit**

Delete the **producer** pod

**kubectl delete -f producer-pod.yml**

And spin up a deployment with 3 replicas

**kubectl apply -f producer-deployment.yml**

Check the pods

**kubectl get pods**

Open a session to the "**observer**"

**kubectl exec -it observer-pod -- sh**

Now, check if the service is accessible by name (producer)

**curl http://producer:5000**

Re-execute a few times and pay attention to the pod name

Close the session

**exit**

Deploy the consumer pod + service (frontend part)

**kubectl apply -f consumer-pod.yml**

**kubectl apply -f consumer-svc.yml**

Check the pods and services

**kubectl get pods**

**kubectl get services**

Open a browser tab to the IP address of one of the nodes + port 30001

For example, navigate to <http://192.168.81.211:30001>

Refresh a few times and pay attention to the IDs on top and bottom of the page

Try with another (owned by other node) IP address

For example, navigate to <http://192.168.81.213:30001>

Refresh a few times. It is working :)

Delete the consumer pod

**kubectl delete -f consumer-pod.yml [--wait=false]**

Create the consumer deployment

**kubectl apply -f consumer-deployment.yml**

Open a browser tab to the IP address of one of the nodes + port 30001

For example, navigate to <http://192.168.81.211:30001>

Okay, our first manually created cluster is working like a charm. Good work 😊

## Part 2: Cluster Management and Upgrade

### Nodes management

Check the pods distribution with

**kubectl get pods -o wide**

Make sure that there are pods on **node-3** (you may need to further scale one of the deployments)

Turn off the **node-3** virtual machine

Check the status of the nodes

**kubectl get nodes**

Check the distribution of the pods

**kubectl get pods -o wide**

Check that the application is working as expected

Hm, the application is working but it appears that the cluster is thinking that some of the pods are working even if the node is missing

We will come back to this in a later module

Power on the node and wait for it to become ready

Check again pods distribution

Some of the pods (the ones that were running on **node-3**) are being restarted

Check the application. It should be working

There is another, more gallant, way to remove a node from the cluster for maintenance

We can first mark the node as not schedulable, so it won't receive any new work

**kubectl cordon node-3.k8s**

Check nodes

**kubectl get nodes**

Then check how the pods are distributed

**kubectl get pods -o wide**

And try to scale up for example the producer deployment to **5**

**kubectl get deployments**

**kubectl edit deployment producer-deploy**

**kubectl get deployments**

No pods should land on **node-3**

**kubectl get pods -o wide**

Scale down the producer deployment back to **3**

**kubectl edit deployment producer-deploy**

**kubectl get deployments**

**kubectl get pods -o wide**

As the cordon action is included in the drain action, we may continue or uncordon it first

**kubectl uncordon node-3.k8s**

Next, we can drain the node. This will remove all work from it

**kubectl drain node-3.k8s --ignore-errors --ignore-daemonsets --delete-local-data –force**

And check what happened

**kubectl get nodes**

**kubectl get pods -o wide**

Now, we can safely do our maintenance tasks and once done, and the node is up and running, we can inform the cluster

**kubectl uncordon node-3.k8s**

And again, check what is going on

**kubectl get nodes**

**kubectl get pods -o wide**

Hm, it seems that the workload is unbalanced. We will accept it for now, but will come back to it in a later module

Let’s clean up a bit

**kubectl delete -f observer-pod.yml**

**kubectl delete -f consumer-svc.yml**

**kubectl delete -f consumer-deployment.yml**

**kubectl delete -f producer-svc.yml**

**kubectl delete -f producer-deployment.yml**

And check that they all are gone

**kubectl get pods,services**

### etcd backup and restore

Let's create a snapshot of the **etcd** database

Log on to the control plane node

Execute the following to create a snapshot

**ETCDCTL\_API=3 etcdctl snapshot save /tmp/etcd-snapshot.db**

If the **etcdctl** binary appears to be missing, then install it

*For example, on* ***Debian****/****Ubuntu****, we can use the following*

***apt-get update***

***apt-get install etcd-client***

Then repeat the backup try

**ETCDCTL\_API=3 etcdctl snapshot save /tmp/etcd-snapshot.db**

If we receive an error again and if reads ***"Error: rpc error: code = Unavailable desc = transport is closing"*** then we must authenticate first

For this, we must change the above command to

**ETCDCTL\_API=3 etcdctl --endpoints=https://127.0.0.1:2379 \**

**--cacert=<trusted-ca-file> --cert=<cert-file> --key=<key-file> \**

**snapshot save /tmp/etcd-snapshot.db**

Where **trusted-ca-file**, **cert-file** and **key-file** can be obtained from the description of the **etcd** pod

We can get them from

**cat /etc/kubernetes/manifests/etcd.yaml**

They are:

**--trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt**

**--cert-file=/etc/kubernetes/pki/etcd/server.crt**

**--key-file=/etc/kubernetes/pki/etcd/server.key**

Then, the final backup command becomes:

**ETCDCTL\_API=3 etcdctl --endpoints=https://127.0.0.1:2379 \**

**--cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key \**

**snapshot save /tmp/etcd-snapshot.db**

Now, everything should work as expected. Check the snapshot

**ls -al /tmp/etcd\***

As we know, **etcd** holds the state of the cluster

So, now that we have it as of now, if a change occurs, we can bring everything back as it was at the time of the snapshot

Let's simulate this by starting a new pod

**kubectl apply -f observer-pod.yml**

**kubectl get pods**

Now, let's restore the database using the snapshot we made earlier to a new folder

**ETCDCTL\_API=3 etcdctl snapshot restore /tmp/etcd-snapshot.db --data-dir /var/lib/etcd-restore**

Next, we must instruct the **etcd** to use the restored data

Edit the **/etc/kubernetes/manifests/etcd.yaml** file and change the **etcd-data** volume to point to the new place (**/var/lib/etcd-restore**)

Save and close the file. Wait a while for the changes to take place

Check again for the pod

**kubectl get pods**

No pod should be there as the restored state said so

### Upgrade a cluster

We will refer to these sources:

<https://kubernetes.io/docs/tasks/administer-cluster/cluster-upgrade/>

<https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/>

Let’s start the process (if you want, reboot all the nodes before continuing)

#### **Upgrade Control Plane nodes**

This we will do one node at a time

Check which the latest version

**apt update**

**apt-cache madison kubeadm**

We have only one control plane node, so we don’t have to choose. Continue on it

At the moment, the latest version is **1.22.3-00** so let’s use it

**apt-get update && \**

**apt-get install -y --allow-change-held-packages kubeadm=1.22.3-00**

Check that the new version is here

**kubeadm version**

Ask for the upgrade plan

**kubeadm upgrade plan**

Should we see any errors (okay, not in production), we may use the following

**kubeadm upgrade plan --ignore-preflight-errors=true**

Then initiate the actual upgrade

**kubeadm upgrade apply v1.22.3**

When asked for confirmation, do it

We may need to upgrade CNI provider plugin (not in our case), so we must consult with its documentation

*If we had other control plane nodes, then we must execute the following command on each one of them:*

***kubeadm upgrade node***

Drain the node

**kubectl drain node-1.k8s --ignore-daemonsets**

*Or if we see any errors that the process cannot be finished, execute this*

***kubectl drain node-1.k8s --ignore-errors --ignore-daemonsets --delete-local-data --force***

Upgrade the **kubelet** and **kubectl**

As at the moment, the latest version is **1.22.3-00**, we will execute this

**apt-get update && \**

**apt-get install -y --allow-change-held-packages kubelet=1.22.3-00 kubectl=1.22.3-00**

Restart the **kubelet** service

**systemctl daemon-reload**

**systemctl restart kubelet**

Uncordon the node

**kubectl uncordon node-1.k8s**

Check the cluster status

**kubectl get nodes**

#### Upgrade nodes

This part we execute one node at a time again

As at the moment, the latest version is **1.22.3-00**, we will execute

**apt-get update && \**

**apt-get install -y --allow-change-held-packages kubeadm=1.22.3-00**

Then the upgrade

**kubeadm upgrade node**

Drain the node (from the control plane node)

**kubectl drain node-2.k8s --ignore-daemonsets**

*Or if we see an error, execute*

***kubectl drain node-2.k8s --ignore-errors --ignore-daemonsets --delete-local-data --force***

Return on the node

Upgrade the **kubelet** and **kubectl**

As now, the latest version is **1.22.3-00**, we will execute

**apt-get update && \**

**apt-get install -y --allow-change-held-packages kubelet=1.22.3-00 kubectl=1.22.3-00**

Then, restart the **kubelet** service

**systemctl daemon-reload**

**systemctl restart kubelet**

And uncordon the node (from the control plane node)

**kubectl uncordon node-2.k8s**

While still on the control plane node, check the cluster status

**kubectl get nodes**

Repeat the procedure on the other node(s)

We did it! Our cluster is upgraded 😊

## Part 3: Highly-available Cluster

For this part we will need an extended setup

We will need three virtual machines for control plane nodes and one or more for nodes members of the cluster

In addition, we will need a machine to act as a load balancer

The following sources are used:

<https://kubernetes.io/docs/tasks/administer-cluster/highly-available-control-plane/>

<https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/>

<https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/>

<https://github.com/kubernetes/kubeadm/blob/main/docs/ha-considerations.md>

### Load Balancer

This is an intentionally over-simplified **HAProxy** setup which will act as a load balancer for the **Control Plane**

Install the required package

**apt-get update**

**apt-get install haproxy**

Then, edit the **/etc/haproxy/haproxy.cfg** file and add the following to the end

**frontend kubernetes**

**bind 192.168.81.220:6443**

**option tcplog**

**mode tcp**

**default\_backend kubernetes-cp**

**backend kubernetes-cp**

**option httpchk GET /healthz**

**http-check expect status 200**

**mode tcp**

**option ssl-hello-chk**

**balance roundrobin**

**server node-1 192.168.81.221:6443 check fall 3 rise 2**

**server node-2 192.168.81.222:6443 check fall 3 rise 2**

**server node-3 192.168.81.223:6443 check fall 3 rise 2**

**frontend stats**

**bind 192.168.81.220:8080**

**mode http**

**stats enable**

**stats uri /**

**stats realm HAProxy\ Statistics**

**stats auth admin:haproxy**

Save and close the file

*Please note that you should adjust it to match your setup (names, ip addresses, etc.)*

Restart the service

**systemctl restart haproxy**

### Control Plane

*Before continuing make sure that all six machines have their* ***/etc/hosts*** *file adjusted*

***echo "192.168.81.221 node-1.k8s node-1" | tee -a /etc/hosts***

***echo "192.168.81.222 node-2.k8s node-2" | tee -a /etc/hosts***

***echo "192.168.81.223 node-3.k8s node-3" | tee -a /etc/hosts***

***echo "192.168.81.224 node-4.k8s node-4" | tee -a /etc/hosts***

***echo "192.168.81.225 node-5.k8s node-5" | tee -a /etc/hosts***

***echo "192.168.81.226 node-6.k8s node-6" | tee -a /etc/hosts***

Initialize the cluster (on the first control plane node)

**kubeadm init --control-plane-endpoint "192.168.81.220:6443" --upload-certs --pod-network-cidr 10.244.0.0/16**

Installation will finish relatively quickly

Copy somewhere the join command(s)

To start using our cluster, we must execute the following

**mkdir -p $HOME/.kube**

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

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

Let's check our cluster nodes (just one so far)

**kubectl get nodes**

Note that it appears as not ready. We know already what is causing this - the missing POD network plugin

Let's install a POD network plugin

For this demo, we will use the **Flannel** plugin

More information here: <https://github.com/flannel-io/flannel#flannel>

Install it

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

We can watch the progress with:

**kubectl get pods --all-namespaces -w**

After a while both **Flannel** and **CoreDNS** will be fully operational

Press **Ctrl + C** to stop the monitoring

Check again the status of the node

**kubectl get nodes**

It should be operational and ready as well

Join the rest of the control plane nodes (adjust and execute the following on all control plane nodes)

**kubeadm join 192.168.81.220:6443 --token ozo8xv.c5jz648l6tp50jqp \**

**--discovery-token-ca-cert-hash sha256:f7eff5b82343969492fb8f8f613dc7ff752dc2da06e5d79e69879d425e980121 \**

**--control-plane --certificate-key d6befa0a65edc6659e1bd56a4706de715c7e11499c2aaf0c4c3f5b7100e7f780**

Check the state of the control plane with

**kubectl get nodes**

**kubectl get nodes -o wide**

### Cluster Members

Now, join the other nodes using the command shown earlier (adjust it and execute it on all remaining nodes)

**kubeadm join 192.168.81.220:6443 --token ozo8xv.c5jz648l6tp50jqp \**

**--discovery-token-ca-cert-hash sha256:f7eff5b82343969492fb8f8f613dc7ff752dc2da06e5d79e69879d425e980121**

Check the state of the control plane with

**kubectl get nodes**

Wow, by now we should have a real cluster 😊

Now, we can do all the usual stuff:

* copy the configuration locally
* install the **Dashboard**
* spin up some workload 😊

If we continue with **NodePort** usage, we will soon see that we must use the IP address of the nodes and not the load balancer

We can correct this by changing the load balancer configuration **/etc/haproxy/haproxy.cfg**

And adding the following block

**frontend nodeport**

**bind \*:30000-32768**

**mode tcp**

**balance roundrobin**

**server node-1 192.168.81.221**

**server node-2 192.168.81.222**

**server node-3 192.168.81.223**

**server node-4 192.168.81.224**

**server node-5 192.168.81.225**

**server node-6 192.168.81.226**

Save and close the file

Restart the service

Check again, but this time use the load balancer IP address

You can also check the load balancer’s statistics page: <http://192.168.81.220:8080/>