Distributed Systems Administration

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Adding complete High Availability + Prometheous to the K3s cluster

Francisco Javier Pizarro Martínez 821259

07/06/2023

Abstract

With the K3s and a complex already working properly we can improve it by adding some monitoring system and also by providing a complete High Availability.

In order to setup the monitoring system we'll be using Prometheus, also to install it we will use Helm.

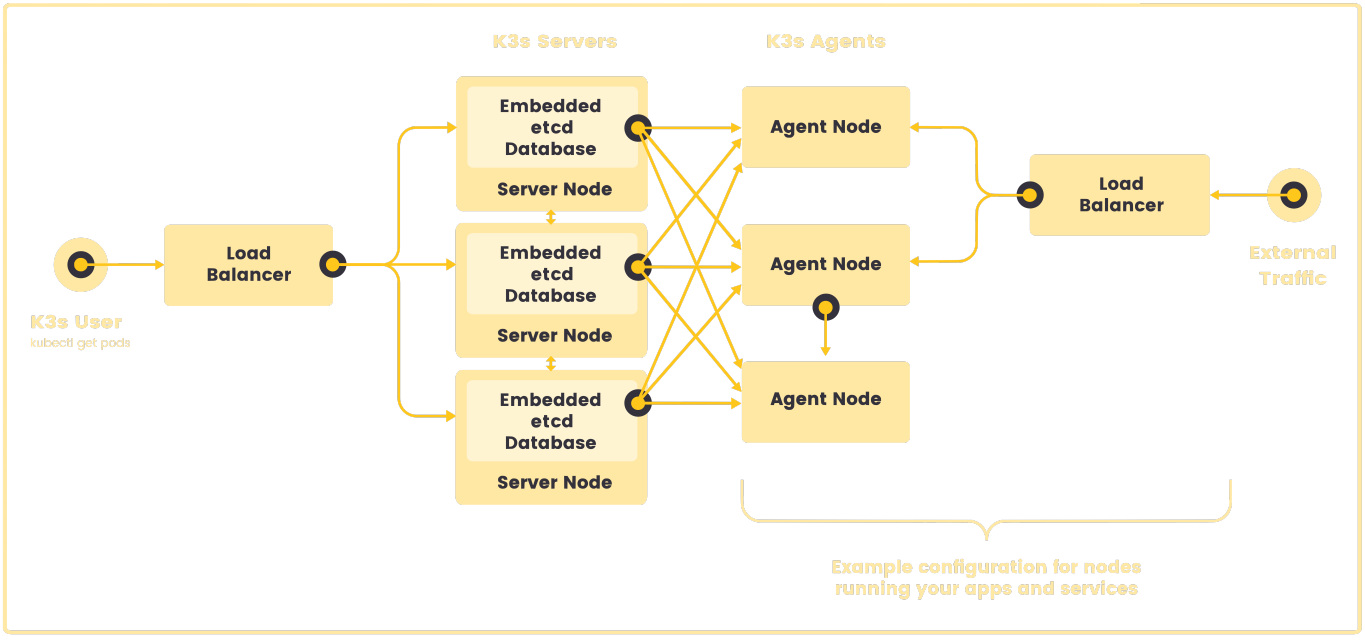
The complete High Avaiability can be reached by performing the following tasks:

- Adding more masters and setting them propperly
- Creating a Virtual Ip/ Using a Load Balancer to be able to reach all the masters even if some of them fail/crash
- Adding HA to the cluster DNS system

The fun part of adding the HA factor to the cluster is playing around with it while we are taking down some nodes, also we can check the status with the Prometheous dashboard we had set up previously.

The test that will be done to prove the HA is taking down the first master(m) and also take down a random worker(w?), with both nodes down we'll create the ceph cluster developed in the last part.

Here is a picture that explains how the new architecture works:



Adding HA to the masters with etcd

The first step to add more masters is creating their corresponding VMs by adding them into our Vagrantfile like this:

```
MASTER = '192.168.0.49'
NODES = [
  { hostname: 'm', type: "master", ip: MASTER, mem: 1800, m: MASTER },
  { hostname: 'm2', type: "masterReplica", ip: '192.168.0.48', mem: 1800, m: MASTER },
  { hostname: 'm3', type: "masterReplica", ip: '192.168.0.47', mem: 1800, m: MASTER },
  { hostname: 'w1', type: "worker", ip: '192.168.0.41', mem: 1800, m: MASTER },
  { hostname: 'w2', type: "worker", ip: '192.168.0.42', mem: 1800, m: MASTER },
  { hostname: 'w3', type: "worker", ip: '192.168.0.43', mem: 1800, m: MASTER },
]
```

After successfully adding them to the Vagrantfile we need to change 2 more things now in our puppet manifest, the first one is adding the flag --cluster-init to the original master(really important step), the second step is adding the case of our special master replicas, this ones must have the server(original master) specified as the worker nodes.

Modify our puppet manifest:

```
   if $nodetype == 'master' {
    exec { 'install_k3s_master':
      command => "env INSTALL_K3S_SKIP_DOWNLOAD=true /vagrant/install.sh server --cluster-init --token
'wCdC16AlP8qpqqI53DM6ujtrfZ7qsEM7PHLxD+Sw+RNK2d1oDJQQOsBkIwy5OZ/5' --flannel-iface enp0s8 --bind-address $nodeip -
-node-ip $nodeip --node-name $hostname --disable traefik --node-taint k3s-controlplane=true:NoExecute",
      path    => '/bin:/usr/bin',
      require => File['/usr/local/bin/k3s'],
    }

    exec { 'copy_k3s_yaml':
      command     => 'cp /etc/rancher/k3s/k3s.yaml /vagrant',
      path        => '/bin:/usr/bin',
      refreshonly => true,
      subscribe   => Exec['install_k3s_master'],
    }
  } else {
      if $nodetype == 'masterReplica' {
      exec { 'install_k3s_master':
        command => "env INSTALL_K3S_SKIP_DOWNLOAD=true /vagrant/install.sh server --server https://$masterip:6443
--token 'wCdC16AlP8qpqqI53DM6ujtrfZ7qsEM7PHLxD+Sw+RNK2d1oDJQQOsBkIwy5OZ/5' --flannel-iface enp0s8 --bind-address
$nodeip --node-ip $nodeip --node-name $hostname --disable traefik --node-taint k3s-controlplane=true:NoExecute",
        path    => '/bin:/usr/bin',
        require => File['/usr/local/bin/k3s'],
      }
       #exec { 'install_k3s_master':
       # command => "curl -sfL https://get.k3s.io |
K3S_TOKEN='wCdC16AlP8qpqqI53DM6ujtrfZ7qsEM7PHLxD+Sw+RNK2d1oDJQQOsBkIwy5OZ/5' sh -s - server --server
https://$masterip:6443 --token 'wCdC16AlP8qpqqI53DM6ujtrfZ7qsEM7PHLxD+Sw+RNK2d1oDJQQOsBkIwy5OZ/5' --flannel-iface
enp0s8 --bind-address $nodeip --node-ip $nodeip --node-name $hostname --disable traefik --node-taint k3s-
controlplane=true:NoExecute",
       # path    => '/bin:/usr/bin',
       # require => File['/usr/local/bin/k3s'],
       #}


    } else {
      exec { 'install_k3s_agent':
        command => "env INSTALL_K3S_SKIP_DOWNLOAD=true /vagrant/install.sh agent --server https://$masterip:6443 -
-token 'wCdC16AlP8qpqqI53DM6ujtrfZ7qsEM7PHLxD+Sw+RNK2d1oDJQQOsBkIwy5OZ/5' --node-ip $nodeip --node-name $hostname
--flannel-iface enp0s8",
        path    => '/bin:/usr/bin',
        require => File['/usr/local/bin/k3s'],
      }
```

```
    }
  }
```

All the masters contribute to the correct work of ETCD which uses raft under the hood, this means that:

- It can still work while there is a majority of the masters alive(in this case 2).
- While there is at least 1 master aviable it can hold the current state of the system.
- One of the masters will be the leader choosen by the consensus algorithm.

We can communicate with ETCD via API to check all this propperties.

Adding a Virtual IP to the HA masters with corosync

Ok so now we have 3 masters but how we can communicate with any one of them as if they were the same to us?

To solve this we use what is called Virtual IP, the idea is really simple this IP acts a Proxy that redirects us to one of the active configured machines.

In order to add the virtual ip we first need to config the tool by running certain commands, this can be achivied by adding this into the puppet manifest:

```
if $nodetype =~ /^master/ {
    if $nodetype == 'master' {
      exec { 'configure_corosynckey':
        command => "sudo corosync-keygen ; sudo cp /etc/corosync/authkey /vagrant/authkey",
        path    => '/bin:/usr/bin',
        require => Exec['install_k3s_master'],
        #require => Exec['configure_pacemaker'],
      }
    } else {
      exec { 'configure_corosynckey':
        command => "cp /vagrant/authkey /etc/corosync/authkey;chmod 400 /etc/corosync/authkey",
        path    => '/bin:/usr/bin',
        require => Exec['install_k3s_master'],
        #require => Exec['configure_pacemaker'],
      }
    }

    exec { 'configure_corosyncconfaddr':
      command => "sed -i 's/bindnetaddr: 127.0.0.1/bindnetaddr: 192.168.0.0/' /etc/corosync/corosync.conf",
      path    => '/bin:/usr/bin',
      require => Exec['configure_corosynckey'],
    }

    exec { 'configure_corosyncconfnodes':
      command => "echo 'nodelist {\n  node {\n    ring0_addr: 192.168.0.49\n    name: primary\n    nodeid: 1\n
}\n  node {\n    ring0_addr: 192.168.0.48\n    name: secondary\n    nodeid: 2\n  }\nnode {\n    ring0_addr:
192.168.0.47\n    name: third\n    nodeid: 3\n  }\n}' >> /etc/corosync/corosync.conf",
      path    => '/bin:/usr/bin',
      require => Exec['configure_corosyncconfaddr'],
    }
    exec { 'enable_corosync':
      command => 'systemctl enable corosync',
      path    => '/bin:/usr/bin',
      require => Exec['configure_corosyncconfnodes'],
    }

    exec { 'start_corosync':
      command => 'systemctl restart corosync',
      path    => '/bin:/usr/bin',
      require => Exec['enable_corosync'],
    }

    if $nodetype == 'master' {
      exec { 'configure_highavailability':
        command => 'sudo crm configure property stonith-enabled=false ; sudo crm configure primitive FAILOVER-ADDR
ocf:heartbeat:IPaddr2 params ip="192.168.0.50" nic="enp0s8" op monitor interval="10s"',
        #command => 'sudo crm configure property stonith-enabled=false ; sudo crm configure primitive FAILOVER-
ADDR ocf:heartbeat:FloatIP params ip="192.168.0.50" nic="enp0s8" op monitor interval="10s"',
        path    => '/bin:/usr/bin',
        require => Exec['enable_corosync'],
        #require => Exec['start_corosync'],
      }
    }
  }
```

The previous commands create and distribute a key that is used by the corosync cluster to autentificate the machines, also we add the nodes list into the configuration files, finally we add the resource of FAILOVER-ADDR that is our Virtual IP in 192.168.0.50

Adding loadbalancer to the connections between client and master

To solve some of the Virtual IP problems we set up a really basic nginx load balancer in the host machine, here is the configuration file:

```
nginx.conf
events {}

stream {
        upstream k3s_servers {
                server 192.168.0.49:6443;
                server 192.168.0.48:6443;
                server 192.168.0.47:6443;
        }

        server {
                listen 6443;
                proxy_pass k3s_servers;
        }
}
```

To start the load balancer we run the command `sudo nginx`.

To solve the default config IP problem we just use this command in the startup script so it modifies the default amster IP by localhost, which in this case points to our loadbalancer which redirects us to the K3s master nodes, the command we need to add is this one:

```
sed -i 's/https:\/\/192.168.0.49:6443/https:\/\/127.0.0.1:6443/g' ~/.kube/config
```

We use localhost instead of the client public IP because of some problem with permissions and certifications, but using this it stills works.

How flannel provides HA

The default backbone of flannel in K3s is vxlan which provides redundancy also uses the etcd provided by the cluster which in this case is also HA. Checking the HA with the kubectl commands:

```
kubectl get nodes
#NAME   STATUS   ROLES                       AGE     VERSION
#m      Ready    control-plane,etcd,master   10m     v1.20.6+k3s1
#m2     Ready    control-plane,etcd,master   9m42s   v1.20.6+k3s1
#m3     Ready    control-plane,etcd,master   8m43s   v1.20.6+k3s1
#w1     Ready    <none>                      6m15s   v1.20.6+k3s1
#w2     Ready    <none>                      7m      v1.20.6+k3s1
#w3     Ready    <none>                      6m13s   v1.20.6+k3s1
kubectl get pods -n kube-system
#NAME                                     READY   STATUS    RESTARTS   AGE
#coredns-854c77959c-p45n8                 1/1     Running   0          11m
#local-path-provisioner-5ff76fc89d-h7bmf   1/1     Running   0          11m
#metrics-server-86cbb8457f-v9kpk           1/1     Running   0          11m
kubectl cluster-info
#Kubernetes control plane is running at https://192.168.0.49:6443
#CoreDNS is running at https://192.168.0.49:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
#Metrics-server is running at https://192.168.0.49:6443/api/v1/namespaces/kube-system/services/https:metrics-
server:/proxy
```

Adding HA to the DNS

Adding a new default HA DNS has 3 really important parts which are setting up a deployment(with antiaffinity) to run the DNS, setting up a service to be able to comunnicate with this deployment and also setting the cluster default dns ip as the one we have defined in our service.

We need to add this flag into the puppet manifest, specificly in the exec that is runned on the original master to start k3s --cluster-dns=10.43.0.99

The kubernetes manifest that creates the DNS and the corresponding service has the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: coredns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
spec:
  replicas: 3
  selector:
    matchLabels:
      k8s-app: kube-dns
  template:
    metadata:
      labels:
        k8s-app: kube-dns
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: k8s-app
                    operator: In
                    values:
                      - kube-dns
              topologyKey: kubernetes.io/hostname
      containers:
        - name: coredns
          image: coredns/coredns:1.8.4
          args: ["-conf", "/etc/coredns/Corefile"]
          volumeMounts:
            - name: config-volume
              mountPath: /etc/coredns
              readOnly: true
      volumes:
        - name: config-volume
          configMap:
            name: coredns
            items:
              - key: Corefile
                path: Corefile
---
apiVersion: v1
kind: Service
metadata:
  name: kube-dns
  namespace: kube-system
  labels:
    k8s-app: kube-dns
spec:
  selector:
    k8s-app: kube-dns
  clusterIP: 10.43.0.99
  ports:
```

```
  - name: dns
    port: 53
    protocol: UDP
```

The DNS is needed since the start of the cluster so rigth after starting the cluster we must run this command:

```
kubectl delete deployment coredns -n kube-system;kubectl delete svc coredns -n kube-system;kubectl apply -f
coredns.yaml
```

To do it we use a startup script.

To test it we need to run the following commands:

```
kubectl get endpoints -n kube-system
kubectl get po -n kube-system -o wide
kubectl get svc -n kube-system -o wide
kubectl run dummy-pod --image=busybox --restart=Never --rm -it -- sh
  #$nslookup moodle.unizar.es
  #Server:         10.43.0.99
  #Address:        10.43.0.99:53
  #Non-authoritative answer:
  #Name:   moodle.unizar.es
  #Address: 155.210.10.103
```

HELM + Prometheus

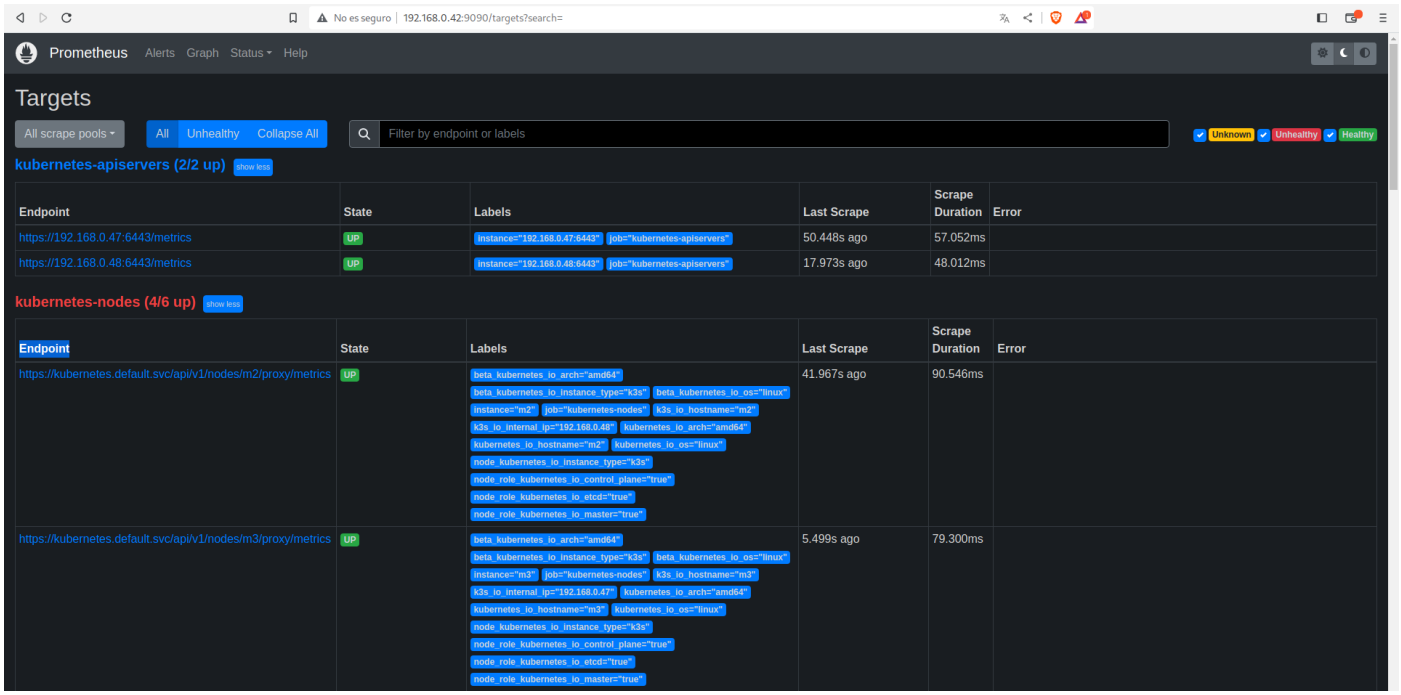
Before installing prometheus we need to install the Helm tool which is the packet manager for kubernets, after installing it we just need to add the Prometheus repo to our Helm list of repositories and install it. We can do this by executing the following commands:

```
wget https://get.helm.sh/helm-v3.12.0-linux-amd64.tar.gz -o helm
tar -zxvf helm-v3.12.0-linux-amd64.tar.gz
mv linux-amd64/helm /usr/local/bin/helm

helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update

#con el k3s corriendo
helm install prometheus prometheus-community/prometheus
kubectl get deployments
kubectl expose deployment prometheus-server -n default --type=LoadBalancer --name=prometheus-server-lb
kubectl get svc
```

After running the commands we should be able to access our web interface and it should look more or less like this:



Annex I Problems found along the project

The most problematic part was without any doubt creating the Virtual IP using corosync and pacemaker. This part had all kind of different errors from the installation proccess to the real use of this IP. The problems of this part were so bad that in order to have the final project working in time as an alternative solution i use a NGINX loadbalancer.

After setting up the load balancer a new problem appeared this was while interacting with the cluster via kubectl using the host machine IP, this caused a error in the certificates.The solution was as absurd as the problem, it was using localhost as destination IP.

A small problem found while adding more masters was the missing flag --cluster-init, without this flag the HA didnt work as expected and a lot of crazy things happen.