Instalación

El proceso instalar un cluster de kubernetes en “**bare metal**”, es decir, sobre servidores, sin ningún tipo de plataforma cloud como GCP, Amazon AWS o Azure, es el siguiente:

1. Instalar nodos (al menos un control plane y un worker)

2. Iniciar el cluster (se ejecuta en el primer control plane)

3. (Opcional, solo para alta disponibilidad) Unir control plane adicionales al cluster

4. Unir worker (al menos uno) al cluster

Nodo (común para control plane y worker)

Estas instrucciones detallan como instalar un nodo de kubernetes, por tanto es COMÚN para todos los nodos, con independencia de su posterior rol dentro del cluster (control plane o worker).

1. Verificar que el nodo cumple todos los siguientes requisitos:

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

2. Instalar un runtime

IMPORTANTE: leer la siguientes consideraciones

- Kubernetes requiere container runtime que siga especificación CRI (CRI-O, containerd, etc.)
- Docker Engine requiere dockershim, que ha sido eliminado, por lo que mejor NO usar Docker Engine como container runtime. Ver Container runtimes. Por curiosidad, el reemplazo (de dockershim) se llama “cri-dockerd”
- Para instalar containerd usaremos los paquetes, que los distribuye Docker Inc.
- Habrá que instalar luego los plugins CNI, que NO vienen en los paquetes que distribuye Docker Inc.

2.1. Instalar containerd

Este ejemplo es para ubuntu

```
sudo apt-get remove docker docker-engine docker.io containerd runc
```

```
sudo apt-get update
```

```
sudo apt-get install \
    ca-certificates \
    curl \
    gnupg \
```

```
    lsb-release
```

```
sudo mkdir -p /etc/apt/keyrings
```

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor
-o /etc/apt/keyrings/docker.gpg
```

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list
> /dev/null
```

```
sudo apt-get update
```

IMPORTANTE: NO instalar docker engine, solo containerd

```
sudo apt install containerd.io
```

Verificar:

```
systemctl status containerd
```

```
q
```

2.2. Redirigir IPv4 y permitir a iptables ver el trafico “bridged”

```
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF
```

```
sudo modprobe overlay
sudo modprobe br_netfilter
```

sysctl params required by setup, params persist across reboots

```
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables  = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
EOF
```

Apply sysctl params without reboot

```
sudo sysctl --system
```

2.3. Configurar systemd cgroup driver

2.3.1. Copia de seguridad y gener archivo nuevo de trinca

```
sudo cp /etc/containerd/config.toml /etc/containerd/config.toml.bak
sudo containerd config default | sudo tee /etc/containerd/config.toml
```

2.3.2. Reemplazar:

```
sudo sed -i 's/SystemdCgroup \= false/SystemdCgroup \= true/g'
/etc/containerd/config.toml
```

Esto es el equivalente a:

```
sudo vim /etc/containerd/config.toml
```

Y dejar esta linea tal que así:

```
            # anyadido
            #SystemdCgroup = false
            SystemdCgroup = true
```

2.3.3. Reiniciar el servicio:

```
sudo systemctl restart containerd
```

3. Instalar plugins CNI

En este ejemplo la arquitectura es ARM

```
sudo mkdir -p /opt/cni/bin/
```

Escoger de:

https://github.com/containernetworking/plugins/releases

OJO: en este ejemplo la arquitectura es ARM

```
sudo wget
https://github.com/containernetworking/plugins/releases/download/v1.1.1/cni-
plugins-linux-arm64-v1.1.1.tgz
```

```
sudo tar Cxzvf /opt/cni/bin cni-plugins-linux-arm64-v1.1.1.tgz
```

```
sudo systemctl restart containerd
```

4. Instalar kubeadm, kubelet and kubectl

```
sudo apt-get update
sudo apt-get install -y apt-transport-https ca-certificates curl
```

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

```
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
```

```
sudo apt-get update
sudo apt-get install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

5. Configurar driver cgroup

5.1. Configurar container runtime cgroup driver

Ya hecho en pasos anteriores, en este caso containerd

5.2. Configurar kubelet cgroup driver

https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/configure-cgroup-driver/#configuring-the-kubelet-cgroup-driver

Entiendo que NO es necesario porque desde la versión 1.22 configurará por defecto systemd

Primer control plane

Este paso solo se tiene que hacer **UNA VEZ POR CLUSTER**, y se ejecutará en aquel nodo, ya instalado, que vaya a ser el **PRIMER CONTROL PLANE**.

Sin alta disponibilidad

1. Conectarse al servidor

```
ssh k8s2
```

2. Iniciar el cluster. Ejecutar:

```
sudo kubeadm init \
  --control-plane-endpoint "k8s2.local:6443" \
  --pod-network-cidr=10.244.0.0/16 \
  --upload-certs \
  --v=5
```

Comentarios:

- Aunque no sea alta disponibilidad es conveniente usar 'control-plane-endpoint' por si más adelante quiere usarse
- Debe ser un nombre que todos los nodos (control plane y workers) resuelvan, o bien la IP privada del control plane
- El puerto es el 6443, es en el que escucha el servicio 'kube-apiserver'
- El parámetro 'pod-network-cidr' es requerido por flannel, y le pasamos el rango de IPs de los

PODS es el '10.244.0.0/16'

3. Anotar de la salida del comando anterior los comandos para unir control planes y workers al cluster:

3.1. Anotar el comando para unir un control plane:

```
sudo kubeadm join \
  k8s2.local:6443 \
  --token r6mawr.wsgooc91h45a55i8 \
  --discovery-token-ca-cert-hash
sha256:94e15af3476146deb36c0f9a53f33a9ea3c441470a2f9e3aefdf538ca7fb4443 \
  --control-plane \
  --certificate-key
7b04bab444d4a5515c514a8b8eeb7a2df6629ee80d46b7f1ed5bd4b1aa3d80ed \
  --v=8
```

Opciones:

- Añadir 'sudo' al comando
- Añadir '–v' para ver más cosas en la salida del comando cuando se ejecute

3.2. Anoter el comando para unir un worker:

```
sudo kubeadm join \
  k8s2.local:6443 \
  --token 9sljzd.hm5i967gzi802cah \
  --discovery-token-ca-cert-hash
sha256:0a3ac17a0a2c093aca67b2cdc8c99e3cc9c374a48fe762385faf705932785de8 \
  --v=8
```

Opciones:

- Añadir 'sudo' al comando
- Añadir '–v' para ver más cosas en la salida del comando cuando se ejecute

4. Instalar plugin CNI, en este ejemplo “flannel”. Ejecutar:

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

5. Configurar archivos para poder usar 'kubectl'. Ejecutar:

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

```
sudo mkdir -p /root/.kube
sudo cp -i /etc/kubernetes/admin.conf /root/.kube/config
```

6. Comprobar:

```
kubectl get nodes
```

Salida esperada similar a:

```
NAME   STATUS   ROLES                  AGE    VERSION
k8s2   Ready    control-plane,master   22h    v1.21.1
```

Con alta disponibilidad

En este ejemplo tendremos 3 nodos que actuarán como control plane:

DNS	IP	Puerto	Comentario
k8s.local	192.168.95.69	8443	'kube-apiserver' cluster.
k8s2.local	192.168.95.71	6443	Primer control plane
k8s3.local	192.168.95.72	6443	Control plane adicional
k8s3.local	192.168.95.73	6443	Control plane adicional

Antes de empezar para cada uno de los tres control plane confifurar balanceador capa 4 kube-apiserver

1. Conectarse al servidor

```
ssh k8s2
```

2. Iniciar el cluster. Ejecutar:

```
sudo kubeadm init \
  --control-plane-endpoint "k8s.local:8443" \
  --pod-network-cidr=10.244.0.0/16 \
  --upload-certs \
  --v=5
```

Comentarios:

- El parámetro 'control-plane-endpoint' debe ser un nombre que todos los nodos (control plane y workers) resuelvan, no se recomienda usar una dirección IP. Es el nombre DNS del balanceador capa 4 kube-apiserver
- El puerto es el 8443, es en el que escucha balanceador capa 4 kube-apiserver 'kube-apiserver'
- El parámetro 'pod-network-cidr' es requerido por flannel, y le pasamos el rango de IPs de los PODS es el '10.244.0.0/16'

3. Anotar de la salida del comando anterior los comandos para unir control planes y workers al cluster:

3.1. Anotar el comando para unir un control plane:

```
sudo kubeadm join \
```

```
  k8s.local:8443 \
  --token r6mawr.wsgooc91h45a55i8 \
  --discovery-token-ca-cert-hash
sha256:94e15af3476146deb36c0f9a53f33a9ea3c441470a2f9e3aefdf538ca7fb4443 \
  --control-plane \
  --certificate-key
7b04bab444d4a5515c514a8b8eeb7a2df6629ee80d46b7f1ed5bd4b1aa3d80ed \
  --v=8
```

Opciones:

- Añadir 'sudo' al comando
- Añadir '–v' para ver más cosas en la salida del comando cuando se ejecute

3.2. Anoter el comando para unir un worker:

```
sudo kubeadm join \
  k8s.local:8443 \
  --token 9sljzd.hm5i967gzi802cah \
  --discovery-token-ca-cert-hash
sha256:0a3ac17a0a2c093aca67b2cdc8c99e3cc9c374a48fe762385faf705932785de8 \
  --v=8
```

Opciones:

- Añadir 'sudo' al comando
- Añadir '–v' para ver más cosas en la salida del comando cuando se ejecute

4. Instalar plugin CNI, en este ejemplo “flannel”. Ejecutar:

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

5. Configurar archivos para poder usar 'kubectl'. Ejecutar:

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

```
sudo mkdir -p /root/.kube
sudo cp -i /etc/kubernetes/admin.conf /root/.kube/config
```

6. Comprobar:

```
kubectl get nodes
```

Salida esperada similar a:

```
NAME   STATUS   ROLES                  AGE    VERSION

```

```
k8s2   Ready    control-plane,master   22h    v1.21.1
```

Unir nodos al cluster

Una vez se ha iniciado el cluster en el primer control plane hay que añadir nodos al cluster:

- Control plane. Solo en alta disponibilidad. Si no vamos a configurar alta disponibilidad, un control plane por cluster de kubernetes es suficiente.
- Workers. Al menos uno.

Control plane (solo alta disponibilidad)

Solo debemos añadir control plane adicionales al cluster si vamos a configurar alta disponibilidad.

1. Conectarse al control plane:

```
ssh k8s3
```

2. Ejecutar el comando obtenido en el paso 3.1. de primer control plane con alta disponibilidad con las siguientes modificaciones:

- Si hemos configurado alta disponibilidad con kube-vip el directorio '/etc/kubernetes/manifests' no estará vacío, y el comando 'sudo kubeadm join' fallará. Por ese motivo, y solo en este caso, debemos añadir al comando 'sudo kubeadm join' el parámetro '–ignore-preflight-errors=DirAvailable–etc-kubernetes-manifests'
- Añadir 'sudo'
- (Opcional) Añadir '–v=8' para que la salida del comando muestre más información

```
sudo kubeadm join \
  k8s.local:8443 \
  --token r6mawr.wsgooc91h45a55i8 \
  --discovery-token-ca-cert-hash
sha256:94e15af3476146deb36c0f9a53f33a9ea3c441470a2f9e3aefdf538ca7fb4443 \
  --control-plane \
  --certificate-key
7b04bab444d4a5515c514a8b8eeb7a2df6629ee80d46b7f1ed5bd4b1aa3d80ed \
  --ignore-preflight-errors=DirAvailable--etc-kubernetes-manifests \
  --v=8
```

3. Pasos opcionales. Los pasos que se listan a continuación son opcionales, y solo son necesarios si queremos usar el comando 'kubectl' desde el segundo y subsiguientes control plane, lo que es acertado, pues igual nos hace falta si el primer control plane se cae por el motivo que sea.

3.1. Configurar archivos para poder usar 'kubectl'. Ejecutar:

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

```
sudo mkdir -p /root/.kube
sudo cp -i /etc/kubernetes/admin.conf /root/.kube/config
```

3.2. Comprobar:

```
kubectl get nodes
```

Salida esperada similar a:

```
NAME   STATUS   ROLES                  AGE    VERSION
k8s2   Ready    control-plane,master   22h    v1.21.1
k8s3   Ready    control-plane,master   22h    v1.21.1
```

4. Repetir los pasos 1 a 3 con el resto de control plane que queramos añadir al cluster.

Worker

Al menos un nodo debe tener el rol de worker. Las instrucciones son casi idénticas se haya configurado o no alta diponibilidad, pero se muestran de forma separada para hacerlas más legibles.

Sin alta disponibilidad

1. Conectarse al worker:

```
ssh k8s3
```

2. Ejecutar el comando obtenido en el paso 3.2. de primer control plane sin alta disponibilidad con las siguientes modificaciones:

- Añadir 'sudo'
- (Opcional) Añadir '–v=8' para que la salida del comando muestre más información

```
sudo kubeadm join \
  k8s2.local:6443 \
  --token 9sljzd.hm5i967gzi802cah \
  --discovery-token-ca-cert-hash
sha256:0a3ac17a0a2c093aca67b2cdc8c99e3cc9c374a48fe762385faf705932785de8 \
  --v=8
```

3. Comprobar

3.1. Conectarse al control plane:

```
ssh k8s2
```

3.2. Ejecutar:

```
kubectl get nodes
```

Salida esperada similar a:

```
NAME   STATUS   ROLES                  AGE     VERSION
k8s2   Ready    control-plane,master   2d18h   v1.21.1
k8s3   Ready    <none>                 46h     v1.21.1
```

4. (Opcional) Repetir los pasos 1 a 4 con el resto de workers que se quiera unir al cluster de kubernetes.

Con alta disponibilidad

1. Conectarse al worker:

```
ssh k8s5
```

2. Ejecutar el comando obtenido en el paso 3.2. de primer control plane con alta disponibilidad con las siguientes modificaciones:

- Añadir 'sudo'
- (Opcional) Añadir '–v=8' para que la salida del comando muestre más información

```
sudo kubeadm join \
  k8s.local:8443 \
  --token 9sljzd.hm5i967gzi802cah \
  --discovery-token-ca-cert-hash
sha256:0a3ac17a0a2c093aca67b2cdc8c99e3cc9c374a48fe762385faf705932785de8 \
  --v=8
```

3. Comprobar

3.1. Conectarse a cualquiera de los control plane:

```
ssh k8s4
```

3.2. Ejecutar:

```
kubectl get nodes
```

Salida esperada similar a:

```
NAME   STATUS   ROLES                  AGE     VERSION
k8s2   Ready    control-plane,master   2d18h   v1.21.1
k8s3   Ready    control-plane,master   2d18h   v1.21.1
k8s4   Ready    control-plane,master   2d18h   v1.21.1
k8s5   Ready    <none>                 46h     v1.21.1
```

4. (Opcional) Repetir los pasos 1 a 4 con el resto de workers que se quiera unir al cluster de kubernetes.

Balanceador capa 4 kube-apiserver

https://github.com/kubernetes/kubeadm/blob/master/docs/ha-considerations.md#kube-vip

Paso requerido para alta disponibilidad.

Vamos a crear un balanceador de carga capa 4 para el servicio kube-apiserver, en este caso mediante un pod.

El objetivo es crear una IP virtual o flotante ('192.168.95.69' en este ejemplo) asociada a un nombre DNS ('k8s.local' en este ejemplo) que irá pasando de un control plane a otro en caso de que el control plane que tenga la IP virtual se caiga.

Las peticiones TCP que lleguen a ese nombre ('k8s.local') y ese puerto ('8443' en este ejemplo) serán dirigidas al puerto TCP 6443 del control plane que en ese momento tenga la IP virtual. En ese puerto, TCP 6443, es en el que escucha el servicio 'kube-apiserver'.

Todos los nodos (control plane o workers) deben resolver el nombre que elijamos para el balanceador de carga, en este ejemplo 'k8s.local'.

Todos los miembros de ese balanceador de carga, los 3 control plane en este ejemplo, deben ser capaces de resolver el nombre DNS de todos y cada uno de sus miembros, ya que usaremos nombres DNS.

En este ejemplo tendremos 3 nodos que actuarán como control plane:

DNS	IP	Puerto	Comentario
k8s.local	10.0.0.200	8443	'kube-apiserver' cluster.
k8s2.local	10.0.0.2	6443	Primer control plane
k8s3.local	10.0.0.3	6443	Control plane adicional
k8s3.local	10.0.0.4	6443	Control plane adicional

Oracle cloud

OJO: una vez pasa el período de prueba gratix NO se pueden modificar, por lo que es MUY arriesgado confiar en este servicio. Por ejemplo si el puerto del ingress controller cambia del 32386 no se pueden añadir/eliminar backends al backend set

Oracle cloud ofrece por la patilla y para siempre un balanceador de capa 4 con IP pública y estática.

Resumen de máquinas, nombres, IPs, etc. (las IPs públicas están cambiadas)

DNS	IP pública	IP privada	Comentario
k8s.legido.com	1.2.3.4	10.0.0.194	Balanceador de carga. Escucha (listener) puerto 6443
k8s1	1.2.3.5	10.0.0.2	Primer control plane
k8s2	1.2.3.6	10.0.0.3	Segundo control plane
k8s3	1.2.3.7	10.0.0.4	Tercer control plane

Creamos:

1. Un load balancer con su correspondiente security list
2. Un listener que escuche en el puerto 6443 oara el servicio 'kube-apiserver'

Haproxy y keepalived como servicios

1. Keepalived

1.1. Instalar

```
sudo apt install keepalived
```

1.2. Crear:

```
sudo vim /etc/keepalived/keepalived.conf
```

Con el siguiente contenido:

```
! /etc/keepalived/keepalived.conf
! Configuration File for keepalived
global_defs {
    router_id LVS_DEVEL
}
vrrp_script check_apiserver {
  script "/etc/keepalived/check_apiserver.sh"
  interval 3
  weight -2
  fall 10
  rise 2
}

vrrp_instance VI_1 {
    state MASTER
    interface enp0s3
    virtual_router_id 51
    priority 101
    authentication {
        auth_type PASS
        auth_pass 42
    }
    virtual_ipaddress {
        10.0.0.200
    }
    track_script {
        check_apiserver
    }
}
```

Explicación:

- STATE: MASTER para master, BACKUP para el resto
- INTERFACE: enp0s3
- ROUTER_ID: 51
- PRIORITY: 101 para el master, 100 para el resto
- AUTH_PASS: 42
- APISERVER_VIP: 10.0.0.200

1.2. Crear:

```
sudo vim /etc/keepalived/check_apiserver.sh
```

Con el siguiente contenido:

```
#!/bin/sh

APISERVER_VIP="10.0.0.200"
APISERVER_DEST_PORT="8443"

errorExit() {
    echo "*** $*" 1>&2
    exit 1
}

curl --silent --max-time 2 --insecure
https://localhost:${APISERVER_DEST_PORT}/ -o /dev/null || errorExit "Error
GET https://localhost:${APISERVER_DEST_PORT}/"
if ip addr | grep -q ${APISERVER_VIP}; then
    curl --silent --max-time 2 --insecure
https://${APISERVER_VIP}:${APISERVER_DEST_PORT}/ -o /dev/null || errorExit
"Error GET https://${APISERVER_VIP}:${APISERVER_DEST_PORT}/"
fi
```

2. haproxy

2.1. Instalar

```
sudo apt install haproxy
```

2.2. Crear:

```
sudo vim /etc/haproxy/haproxy.cfg
```

Con el siguiente contenido:

```
# /etc/haproxy/haproxy.cfg
#---------------------------------------------------------------------
# Global settings
#---------------------------------------------------------------------
global
    log /dev/log local0
    log /dev/log local1 notice
```

```
    daemon

#---------------------------------------------------------------------
# common defaults that all the 'listen' and 'backend' sections will
# use if not designated in their block
#---------------------------------------------------------------------
defaults
    mode                    http
    log                     global
    option                  httplog
    option                  dontlognull
    option http-server-close
    option forwardfor       except 127.0.0.0/8
    option                  redispatch
    retries                 1
    timeout http-request    10s
    timeout queue           20s
    timeout connect         5s
    timeout client          20s
    timeout server          20s
    timeout http-keep-alive 10s
    timeout check           10s

#---------------------------------------------------------------------
# apiserver frontend which proxys to the control plane nodes
#---------------------------------------------------------------------
frontend apiserver
    bind *:8443
    mode tcp
    option tcplog
    default_backend apiserver

#---------------------------------------------------------------------
# round robin balancing for apiserver
#---------------------------------------------------------------------
backend apiserver
    option httpchk GET /healthz
    http-check expect status 200
    mode tcp
    option ssl-hello-chk
    balance     roundrobin
        server k8s1 k8s1.local:6443 check
        server k8s2 k8s2.local:6443 check
        server k8s3 k8s3.local:6443 check
```

3. Arrancar servicios:

```
sudo systemctl enable haproxy --now
sudo systemctl enable keepalived --now
```

IMPORTANTE: no va a funcionar hasta que el backend (apiserver) empiece a escuchar en el puerto

6443

Kube-vip

Antiguo No funciona en Oracle OCI

DNS	IP	Puerto	Comentario
k8s.local	192.168.95.69	8443	'kube-apiserver' cluster.
k8s2.local	192.168.95.71	6443	Primer control plane
k8s3.local	192.168.95.72	6443	Control plane adicional
k8s3.local	192.168.95.73	6443	Control plane adicional

1. Conectarse al control plane. Ejecutar:

```
ssh k8s2
```

2. Ejecutar:

```
sudo mkdir -p /etc/kube-vip
sudo vim /etc/kube-vip/config.yaml
```

Con el siguiente contenido, ajustando:

- localPeer. Id lo que queramos, address la IP privada del control plane.
- remotePeers. Igual que 'localPeer' pero para el resto de control planes.
- vip. La misma para todo el cluster.
- startAsLeader. Para el primer control plane 'true', para el resto 'false'.
- interface. En este ejemplo se tiene solo una interfaz, por tanto la 'eth0'.
- loadBalancers. El cluster escucha en el puerto 8433, cada uno de los control plane escucha en el puerto 6443

```
localPeer:
  id: k8s2.local
  address: 192.168.95.71
  port: 10000
remotePeers:
- id: k8s3.local
  address: 192.168.95.72
  port: 10000
- id: k8s4.local
  address: 192.168.95.73
  port: 10000
vip: 192.168.95.69
gratuitousARP: true
singleNode: false
startAsLeader: true
interface: eth0
loadBalancers:
- name: API Server Load Balancer
```

```
  type: tcp
  port: 8443
  bindToVip: false
  backends:
  - port: 6443
    address: 192.168.95.71
  - port: 6443
    address: 192.168.95.72
  - port: 6443
    address: 192.168.95.73
```

3. Ejecutar

```
docker run -it --rm plndr/kube-vip:0.1.1 /kube-vip sample manifest \
    | sed "s|plndr/kube-vip:'|plndr/kube-vip:0.1.1'|" \
    | sudo tee /etc/kubernetes/manifests/kube-vip.yaml
```

Resultado esperado similar a:

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  name: kube-vip
  namespace: kube-system
spec:
  containers:
  - command:
    - /kube-vip
    - start
    - -c
    - /vip.yaml
    image: 'plndr/kube-vip:0.1.1'
    name: kube-vip
    resources: {}
    securityContext:
      capabilities:
        add:
        - NET_ADMIN
        - SYS_TIME
    volumeMounts:
    - mountPath: /vip.yaml
      name: config
  hostNetwork: true
  volumes:
  - hostPath:
      path: /etc/kube-vip/config.yaml
    name: config
status: {}
```

IMPORTANTE: cuando se tengan que unir el segundo y tercer control plane al cluster (con el

comando 'sudo kubeadm join') hay que pasar el parámetro:

```
  --ignore-preflight-errors=DirAvailable--etc-kubernetes-manifests
```

4. Repetir los pasos 1 a 4, ajustando los parámetros que se indica en cada paso, hasta que no queden más control plane por añadir al balanceador de carga de capa 4.

Cuando se inicie el cluster (primer control plane) o se una el control plane al cluster (segundo y subsiguientes control planes) se iniciará un pod que gestionará la IP virtual ('192.168.95.69' en este ejemplo).

Deployment

Vamos a desplegar un POD con la imagen https://hub.docker.com/r/containous/whoami whoami.

1. Conectarse al master

```
ssh master
```

2. Ejecutar:

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-whoami
  labels:
    app: deployment-whoami
spec:
  replicas: 1
  selector:
    matchLabels:
      app: deployment-whoami
  template:
    metadata:
      labels:
        app: deployment-whoami
    spec:
      containers:
      - name: whoami
        image: containous/whoami
        ports:
        - containerPort: 80
EOF
```

3. Comprobar

```
kubectl get deployments
```

Resultado esperado similar a:

```
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
deployment-whoami   1/1     1            1           40h
```

Service

Un Service expone un Deployment de forma que se pueda acceder al 'Deployment' desde fuera del cluster de Kubernetes.

Requisitos:

- Un Deployment con 'metadata.name' igual a 'deployment-whoami'

1. Conectarse al master

```
ssh master
```

2. Ejecutar:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: service-whoami
spec:
  selector:
    app: deployment-whoami
  ports:
    - protocol: TCP
      port: 80
EOF
```

3. Comprobar

```
kubectl get services --field-selector metadata.name=service-whoami
```

Resultado esperado similar a:

```
NAME             TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
service-whoami   ClusterIP   10.103.221.128   <none>        80/TCP    42h
```

Metallb

Se trata de un balanceador de carga por software para instalaciones “bare metal”, es decir, aquellas

que NO se producen en ningún proveedor en la nube, como Amazon AWS, Google Cloud Services o Azure.

Es necesario instalarlo para poder instalar un servicio de tipo “LoadBalancer” y que obtenga una “EXTERNAL-IP”.

Más información https://metallb.universe.tf/ aquí.

En este ejemplo vamos a usar:

- Layer 2
- IPs privadas

Layer 2

Requisitos:

- Tener un rango de IPs **que no se usen**. Es decir, no vale ni las que tienen los nodos del cluster, ni las que usa flannel. En este ejemplo tenemos:
- 192.168.95.68
- 192.168.95.69

1. Conectarse al master

```
ssh master
```

2. Instalar. Ejecutar:

```
kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.9.6/manifests/namespace
.yaml
```

```
kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.9.6/manifests/metallb.y
aml
```

Solo la primera vez:

```
kubectl create secret generic -n metallb-system memberlist --from-
literal=secretkey="$(openssl rand -base64 128)"
```

3. Configuración. Ajustar:

- Nombre del pool de direcciones. En este ejemplo “default”
- IPs. En este ejemplo “192.168.95.68” y “192.168.95.69”
- Protocolo. En este ejemplo “layer2”

Ejecutar:

```
cat <<EOF | kubectl apply -f -
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
      - 192.168.95.68-192.168.95.69
EOF
```

4. Probar

4.1. Crear un “Service”, no importa que apunte a un “Deployment” valido. Ajustar 'metadata.annotations.metallb.universe.tf/address-pool' para que coincida con el valor del nombre del pool de direcciones, en este ejemplo “default”.

Ejecutar:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: test-service-canberemoved
  annotations:
    metallb.universe.tf/address-pool: default
spec:
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: nonexistantdeployment
  type: LoadBalancer
EOF
```

4.2. Comprobar:

```
kubectl get services --field-selector metadata.name=test-service-
canberemoved
```

Resultado esperado similar a:

```
NAMESPACE   NAME                         TYPE           CLUSTER-IP
EXTERNAL-IP     PORT(S)        AGE
default     test-service-canberemoved   LoadBalancer   10.108.71.69
192.168.95.69   80:31652/TCP   3m52s
```

El campo 'EXTERNAL-IP' tiene asignada una IP del pool de direcciones.

4.3. Limpieza

```
kubectl delete services --field-selector metadata.name=test-service-
canberemoved
```

Cambiar 'externalTrafficPolicy'

Fuente

Requisitos:

- Tener instalado nginx controller

1. Conectarnos al master

```
ssh k8s2
```

2. Editar el servicio, en nuestro caso se llama 'ingress-nginx-controller'. Ejecutar:

```
kubectl edit services -n ingress-nginx ingress-nginx-controller
```

3. Cambiar:

```
externalTrafficPolicy: Cluster
```

Por:

```
externalTrafficPolicy: Local
```

IMPORTANTE: NO hace falta hacer un rollout restart de los pods, los cambios los toma inmediatamente.

4. Comprobar. En mi caso accediendo al nombre DNS público:

https://example.com

El valor de 'X-Forwarded-For' ha cambiado:

- Cluster ⇒ aparecía la IP de la interfaz de flannel del nodo que estaba ejecutando el pod 'ingress-nginx-controller-7c6b6c6fb8-bvvcf', en este caso '10.244.1.1'
- Local ⇒ aparece la IP privada del edge router. Todavía tengo que afinar, debería aparecer la IP pública del cliente que hizo la petición HTTPS.

Addons

Addons

Cert manager

Cert-manager es un addon que sirve para generar certificados SSL Letsencrypt.

Para instalarlo:

1. Conectarse al master

```
ssh master
```

2. Instalar los manifiestos de la versión marcada como 'Latest release', en este ejemplo la 'v1.8.1':

```
kubectl apply -f
https://github.com/jetstack/cert-manager/releases/download/v1.8.1/cert-manag
er.yaml
kubectl apply -f
https://github.com/jetstack/cert-manager/releases/download/v1.8.1/cert-manag
er.crds.yaml
```

3. Probar

3.1. Ejecutar:

```
watch -n 5 kubectl get pods --namespace cert-manager
```

Hastq que el pod 'webhook' esté corriendo. Salida esperada similar a:

```
NAME                                       READY   STATUS    RESTARTS   AGE
cert-manager-7dd5854bb4-zz8wj              1/1     Running   0          71s
cert-manager-cainjector-64c949654c-j7s9h   1/1     Running   0          71s
cert-manager-webhook-6b57b9b886-9ghg5      1/1     Running   0          71s
```

Suele tardar menos de un minuto.

3.2. Crear:

- Namespace
- Issuer
- Certificate

Ejecutar:

```
cat <<EOF > test-resources.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: cert-manager-test
---
apiVersion: cert-manager.io/v1
kind: Issuer
```

```
metadata:
  name: test-selfsigned
  namespace: cert-manager-test
spec:
  selfSigned: {}
---
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: selfsigned-cert
  namespace: cert-manager-test
spec:
  dnsNames:
    - example.com
  secretName: selfsigned-cert-tls
  issuerRef:
    name: test-selfsigned
EOF
```

3.3. Cargar el manifiesto creado en el paso anterior.

IMPORTANTE: esperar un par de minitos desde el paso anterior, o de otra forma podemos obtener un error:

```
x509: certificate signed by unknown authority
```

Yo no fui capaz de resolver ese error.

Ejecutar:

```
kubectl apply -f test-resources.yaml
```

3.4. Ver el certificado. Ejecutar:

```
kubectl describe certificate -n cert-manager-test | grep Message
```

Resultado esperado similar a:

```
Message:               Certificate is up to date and has not expired
```

3.5. Limpieza. Ejecutar:

```
kubectl delete -f test-resources.yaml
```

Issuer

Se trata de un custom resource creado durante el proceso de instalación del Addon llamado cert-manager.

Requisitos:

- Tener instalado cert-manager

Para instalarlo:

1. Conectarse al control plane

```
ssh k8s2
```

2. Ejecutar:

Ajustando:

- email. Si es del dominio 'example.com' no generará las claves tras llamar a letsencrypt

```
cat <<EOF | kubectl apply -f -
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: letsencrypt-production
  namespace: default
spec:
  acme:
    # Staging
    #server: https://acme-staging-v02.api.letsencrypt.org/directory
    # Production
    server: https://acme-v02.api.letsencrypt.org/directory
    # Email address used for ACME registration
    email: letsencrypt@example.com
    # Name of a secret used to store the ACME account private key
    privateKeySecretRef:
      name: letsencrypt-production
    # Enable the HTTP-01 challenge provider
    solvers:
    # An empty 'selector' means that this solver matches all domains
    - selector: {}
      http01:
        ingress:
          class: nginx
EOF
```

En este ejemplo estamos usando la API de producción de letsencrypt:

https://acme-v02.api.letsencrypt.org/directory

Pero podemos usar si queremos la de staging:

https://acme-staging-v02.api.letsencrypt.org/directory

3. Comprobar. Ejecutar:

```
watch -n 5 kubectl get issuers.cert-manager.io
```

Resultado esperado similar a:

```
NAME                      READY   AGE
letsencrypt-production    True    43h
```

Tarda menos de un minuto.

Edge router

Se trata de un servidor completamente por fuera de kubernetes, pero se menciona aquí para completar el ejemplo.

Se encarga de escuchar peticiones en los puertos TCP 80 y TCP 443 y dirigirlas al cluster kuberntes. Los nombres DNS de los servicios con certificados SSL válidos, por ejemplo con letsencrypt, deben resolver a la IP pública del edge router. Por ejemplo “example.com” debe resolver a la IP pública del edge router, de forma que la petición finalmente le llegue al cluster de kubernetes a través del siguiente esquema:

[Cliente] ⇒ example.com ⇒ [Edge router] ⇒ [Worker] ⇒ [Ingress route] ⇒ [Service] ⇒ [Deployment]

Requisitos:

- IP pública
- IP privada en el mismo rango que los nodos del clúster de kubernetes (“192.168.95.0/24” en este ejemplo)
- (Opcional) Docker instalado. Usamos docker para hacer más rápido y sencillo el despliegue de nginx

NodePort (usar este)

Requisitos:

- IPs privadas de los workers del clúster
- Puertos en los que escuchan los Nodeport tanto HTTP como HTTPS. Ver paso 8 de Nginx ingress Nodeport daemonset
- Docker

Bare metal

Estas instrucciones sob para montar completamente por fuera de kubernetes un balanceador de carga capa 7.

TODO: refinar, porque cuando jugué con OCI me di cuenta que capa 4 funciona perfectamente, no hace falta liarse con proxy_pass y capa 7

1. Acceder al servidor

```
ssh k8s1
```

2. Crear el archivo 'default.conf':

```
vim default.conf
```

Con el siguiente contenido:

```
proxy_set_header X-Real-IP       $proxy_protocol_addr;
proxy_set_header X-Forwarded-For $remote_addr;
# To avoid 404. Credits:
# https://serverfault.com/a/407983/570054
proxy_set_header Host $host:$server_port;

server {
        listen 80;
        location / {
                proxy_pass http://k8s;
        }
        error_page 404 /404.html;
        location = /404.html {
                root /usr/share/nginx/html/;
                internal;
        }
}
```

3. Crear el archivo 'nginx.conf':

```
vim nginx.conf
```

Con el siguiente contenido:

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;


events {
    worker_connections  1024;
}


http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;
```

```
    log_format  main  '$remote_addr - $remote_user [$time_local] "$request"
'
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;

    #gzip  on;

    include /etc/nginx/conf.d/*.conf;

    upstream k8s {
        # IP: kubernetes node private IP. Port: nginx ingress controller in
NodePort port for HTTP traffic
        server 192.168.95.74:31104;
        server 192.168.95.75:31104;
        server 192.168.95.76:31104;
    }
}

stream {

    server {
        listen 443;

        proxy_pass k8s_https;
    # Requires 'use-proxy-protocol' to 'true' in configmap of nginx ingress
controller
    #
https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/co
nfigmap/#use-proxy-protocol
    proxy_protocol on;

    }

        upstream k8s_https {
                # IP: kubernetes node private IP. Port: nginx ingress
controller in NodePort port for HTTP traffic
                server 192.168.95.74:30893;
                server 192.168.95.75:30893;
                server 192.168.95.76:30893;
        }

    log_format basic '$remote_addr - [$time_local] '
                       '$status '
                       '';
```

```
    access_log  /var/log/nginx/access.log  basic;

}
```

Modificar:

- IPs y puertos de los workers del clúster de kubernetes. En este ejemplo las IPs son:
  - 192.168.95.74
  - 192.168.95.75
  - 192.168.95.76
- Puerto en el que escucha el servicio NodePort para las conexiones HTTP. En este ejemplo '31104'
- Puerto en el que escucha el servicio NodePort para las conexiones HTTPS. En este ejemplo '30893'

4. Arrancar el contenedor docker, ajustando la ruta a los archivos creados en el paso anterior:

```
docker run \
  --name edge-router \
  --v /home/debian/nginx.conf:/etc/nginx/nginx.conf:ro \
  -v /home/debian/default.conf:/etc/nginx/conf.d/default.conf \
  -p 80:80 \
  -p 443:443 \
  -d nginx
```

5. Probar

Para esta prueba necesitamos los siguientes requisitos:

- Ingress de nginx instalado y configurado así
- Nombre DNS (en este ejemplo 'example.com') apuntando a la IP pública del servidor
- Ingress con letsencrypt y nombre de host 'example.com'

5.1. Pronbar que el contenedor está corriendo:

```
docker logs -f edge-router
```

5.2. Abrir un navegador y acceder a la URL configurada en el ingress route, en este ejemplo “example.com”:

```
https://example.com
```

Resultado esperado similar a:

```
Hostname: deployment-whoami-6b6dc7c84-tbv7d
IP: 127.0.0.1
IP: 10.244.2.4
RemoteAddr: 10.244.2.8:42052
GET / HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101
```

```
Firefox/78.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.5
Upgrade-Insecure-Requests: 1
X-Forwarded-For: 8.8.8.8
X-Forwarded-Host: example.com
X-Forwarded-Port: 443
X-Forwarded-Proto: https
X-Real-Ip: 8.8.8.8
X-Request-Id: 4d47507a04aa3e8ab3647c5e03983e41
X-Scheme: https
```

Donde '8.8.8.8' es la IP pública desde donde hemos lanzado la petición.

Oracle

Requisitos:

- Un load balancer con su correspondiente security list

Creamos dos listeners, todo ello capa 4:

- Uno que escuche en el puerto 80
- Otro que escuche en el puerto 443

LoadBalancer

AVISO: funciona, pero no devuelve la IP origen. Igual cambiando algo se puede conseguir

En nuestro caso se van a redirigir todas las peticiones no a un servicio de tipo 'NodePort' como se menciona aquí, sino a la “EXTERNAL-IP” (en nuestro ejemplo una IP privada, la '192.168.95.68') de un servicio de tipo 'LoadBalancer', este.

Esquema:

[Cliente DMZ] ⇒ https://example.com ⇒ [Edge router] ⇒ http://192.168.95.68 ⇒ [ingress-nginx-controller] ⇒ [ingress-whoami] ⇒ [service-whoami] ⇒ [deployment-whoami]

Si lo hiciéramos con servicios de tipo NodePort deberíamos estar continuamente tocando la configuración del edge router para especificar cada puerto, pues cada servicio tomaría un puerto distinto.

Nuestro edge router:

- Tiene una IP pública.
- Tiene una IP privada del rango de IPs del pool de IPs de metallb.
- Tiene instalado docker (porque el servicio nginx va a ser creado mediante un contenedor).

Instalación:

1. Conectarnos al edge router

```
ssh edgerouter
```

2. Crear el siguiente archivo, ajustando la IP a la que le ha sido asignada al servicio 'ingress-nginx-controller' como 'EXTERNAL-IP', en este ejemplo '192.168.95.68':

```
vim nginx.conf
```

Con el siguiente contenido:

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;


events {
    worker_connections  1024;
}


http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format main '$remote_addr - [$time_local] '
                        '"$request" $status $body_bytes_sent '
                        '"$http_referer" "$http_user_agent"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;

    #gzip  on;

    include /etc/nginx/conf.d/*.conf;
}

stream {

    map $ssl_preread_server_name $targetBackend {
        ~^(.*)$ 192.168.95.68:443;
    }
```

```
    server {
        listen 443;

        proxy_pass $targetBackend;
        ssl_preread on;
    }

    log_format basic '$remote_addr - [$time_local] '
                     '$status '
                     '';

    access_log  /var/log/nginx/access.log  basic;
}
```

3. Crear el siguiente archivo, ajustando la IP a la que le ha sido asignada al servicio 'ingress-nginx-controller' como 'EXTERNAL-IP', en este ejemplo '192.168.95.68':

```
vim default.conf
```

Con el siguiente contenido:

```
proxy_set_header X-Real-IP       $proxy_protocol_addr;
proxy_set_header X-Forwarded-For $remote_addr;

server {
    listen 80;
    location / {
            #proxy_pass http://192.168.95.68;
            proxy_pass http://php;
        proxy_set_header Host            $host;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
        error_page 404 /404.html;
        location = /404.html {
                root /usr/share/nginx/html/;
                internal;
        }
}
```

4. Arrancar el contenedor, ajustando las rutas a los archivos de configuración, ejecutando:

```
docker run \
  --name reverse-proxy \
  -v /home/debian/nginx.conf:/etc/nginx/nginx.conf:ro \
  -v /home/debian/default.conf:/etc/nginx/conf.d/default.conf \
  -p 80:80 \
  -p 443:443 \
  -d nginx
docker logs -f reverse-proxy
```

Redundancia

Requisitos:

- Dos servidores
- Cada uno de los cuales tiene una IP pública
- Existe una tercera IP pública (“virtual” o “flotante”)
- Las 3 IPs públicas (las de cada servidor y la flotante) están en la misma subred

1. Instalar los dos servidores con NodePort

2. Instalar y configurar en ambos servidores Keepalived

Ingress

Según su propia definición:

```
An API object that manages external access to the services in a cluster,
typically HTTP.
```

Nginx

Ingress nginx

Nginx NodePort daemonset (usar este)

Con este procedimiento podremos obtener la IP de origen en los pods, con las otras aproximaciones todavía no lo he conseguido.

Características:

- Usaremos NodePort, de forma que todos los nodos, incluidos el master, expondrán a la DMZ un puerto para HTTP (el mismo en todos los nodos), y otro para HTTPS.
- Usaremos daemonset, de forma que por cada nodo worker, en este ejemplo 3, se creará un pod.
- Usaremos “externalTrafficPolicy: Local” para preservar la IP origen. La explicación detallada se puede encontrar aquí y aquí

1. Conectarse al master

```
ssh k8s2
```

2. Obtener el archivo:

```
wget
```

```
https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/baremetal/deploy.yaml
```

3. Forzar que los pods solo envien tráfico a los deployments, según su ingress route, que estén corriendo en el mismo worker node. Añadir 'spec.externalTrafficPolicy: Local'. Editar:

```
vim deploy.yaml
```

Y cambiar:

```
# Source: ingress-nginx/templates/controller-service.yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
  labels:
    helm.sh/chart: ingress-nginx-3.30.0
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/version: 0.46.0
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/component: controller
  name: ingress-nginx-controller
  namespace: ingress-nginx
spec:
```

Por:

```
# Source: ingress-nginx/templates/controller-service.yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
  labels:
    helm.sh/chart: ingress-nginx-3.30.0
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/version: 0.46.0
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/component: controller
  name: ingress-nginx-controller
  namespace: ingress-nginx
spec:
  # this setting is t make sure the source IP address is preserved.
  externalTrafficPolicy: Local
```

4. **AVISO** En la última versión ya es “NodePort”, pero dejo esto documentado por si Ascaso

Reemplazar para el tipo de balanceo para el servicio

```
vim deploy.yml
```

Y cambiar:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/component: controller
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
    app.kubernetes.io/version: 1.2.0
  name: ingress-nginx-controller
  namespace: ingress-nginx
spec:
  externalTrafficPolicy: Local
  ports:
  - appProtocol: http
    name: http
    port: 80
    protocol: TCP
    targetPort: http
  - appProtocol: https
    name: https
    port: 443
    protocol: TCP
    targetPort: https
  selector:
    app.kubernetes.io/component: controller
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/name: ingress-nginx
  type: LoadBalancer
```

Por:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/component: controller
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
    app.kubernetes.io/version: 1.2.0
  name: ingress-nginx-controller
  namespace: ingress-nginx
spec:
  externalTrafficPolicy: Local
  ports:
```

```
  - appProtocol: http
    name: http
    port: 80
    protocol: TCP
    targetPort: http
  - appProtocol: https
    name: https
    port: 443
    protocol: TCP
    targetPort: https
  selector:
    app.kubernetes.io/component: controller
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/name: ingress-nginx
  # anyadido
  #type: LoadBalancer
  type: NodePort
```

5. Instalar. Ejecutar:

```
kubectl apply -f deploy.yaml
```

6. Probar

6.1. Ejecutar:

```
kubectl get pods \
-n ingress-nginx \
-l app.kubernetes.io/name=ingress-nginx \
-o wide \
--watch
```

Resultado esperado similar a:

```
ingress-nginx-controller-55bc4f5576-78rmm   1/1     Running     0
45h   10.244.5.3   k8s7     <none>           <none>
ingress-nginx-controller-55bc4f5576-f27v5   1/1     Running     0
45h   10.244.3.3   k8s5     <none>           <none>
ingress-nginx-controller-55bc4f5576-v7f6h   1/1     Running     0
45h   10.244.4.3   k8s6     <none>           <none>
ingress-nginx-controller-6f7c7fb6f8-xvzcs   0/1     Pending     0
45h   <none>       <none>   <none>           <none>
```

IMPORTANTE: debe haber al menos una réplica por cada node worker.

6.2. Verificar que efectivamente cada servicio (ingress controller) corre en un node worker distinto. Ejecutar:

```
kubectl get nodes
```

Resultado esperado similar a:

```
NAME   STATUS   ROLES                  AGE     VERSION
k8s2   Ready    control-plane,master   2d19h   v1.21.1
k8s3   Ready    control-plane,master   2d19h   v1.21.1
k8s4   Ready    control-plane,master   2d18h   v1.21.1
k8s5   Ready    <none>                 46h     v1.21.1
k8s6   Ready    <none>                 46h     v1.21.1
k8s7   Ready    <none>                 45h     v1.21.1
```

Tenemos pues tres workers:

- k8s5
- k8s6
- k8s7

Todo correcto, cada worker tiene al menos un pod corriendo.

6.3. Comprobar la versión. Ejecutar:

```
POD_NAMESPACE=ingress-nginx
POD_NAME=$(kubectl get pods -n $POD_NAMESPACE -l
app.kubernetes.io/name=ingress-nginx --field-selector=status.phase=Running -
o jsonpath='{.items[0].metadata.name}')
kubectl exec -it $POD_NAME -n $POD_NAMESPACE -- /nginx-ingress-controller --
version
```

Resultado esperado similar a:

```
-------------------------------------------------------------------------------
---
NGINX Ingress controller
  Release:       v0.46.0
  Build:         6348dde672588d5495f70ec77257c230dc8da134
  Repository:    https://github.com/kubernetes/ingress-nginx
  nginx version: nginx/1.19.6

-------------------------------------------------------------------------------
---
```

7. Obtener los puertos expuestos. Esto es necesario para instalar y configura el edge router. Ejecutar:

```
kubectl get services -n ingress-nginx --field-selector
metadata.name=ingress-nginx-controller
```

Resultado esperado similar a:

```
NAME                       TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
ingress-nginx-controller   NodePort   10.110.176.170   <none>
80:31104/TCP,443:30893/TCP   55m
```

Por lo tanto en este ejemplo:

- HTTP ⇒ 31104
- HTTPS ⇒ 30893

8. **AVISO** Este paso está deprecated. Solo hay que hacerlo si se va a usar un edge router con balanceador capa 7. NO hay que hacerlo, eso lo hice en un primer intento. El balanceador que hay que poner en el edge router tiene que ser capa 4, por tanto NO hay que habilitar proxy protocol.

Editar ConfigMaps para habilitar 'use-proxy-protocol'

Actualizar nginx ingress controller

1. Comprobar si es posible una actualización

1.1. Obtemer la versión actual del nginx ingress controller

```
kubectl describe daemonsets.apps -n ingress-nginx ingress-nginx-controller
| grep vers
```

Resultado esperado similar a:

```
                        app.kubernetes.io/version=1.2.0
```

Por tanto la versión actual es “1.2.0”

1.2. Determinar si hay una nueva versión:

https://github.com/kubernetes/ingress-nginx#changelog

La hay, “v1.2.1”, compatible con las siguientes versiones de kubectl:

1.22, 1.21, 1.20, 1.19

```

```

1.3. Determinar la versión actual de kubectl:

```
kubectl version --short
```

Resultado esperado similar a:

```
Client Version: v1.24.2
Kustomize Version: v4.5.4
Server Version: v1.24.2
```

Tenemos kubectl versión “1.24.x”, que está por encima de “1.23”, asumo que podemos ir hacia delante.

2. Actualizar la versión de la imagen del daemonset (recordad que cambiamos el “Deployment” por “Daemonset”)

2.1. Comprobar que la imagen de docker (tag) existe, y obtener el SHA256, todo esto desde la máquina local

```
docker pull registry.k8s.io/ingress-nginx/controller:v1.2.1
```

Resultado esperado similar a:

```
v1.2.1: Pulling from ingress-nginx/controller
8663204ce13b: Pull complete
897a18b2d257: Pull complete
3cb02f360cf3: Pull complete
2b63816a7692: Pull complete
d61ce16aa3b6: Pull complete
4391833fbf2c: Pull complete
4f4fb700ef54: Pull complete
bb397308bcd5: Pull complete
803395581751: Pull complete
153d402a7263: Pull complete
c815f058cf7b: Pull complete
a872540e4aca: Pull complete
4972574251d0: Pull complete
30197fe775a6: Pull complete
b059831ea274: Pull complete
Digest:
sha256:5516d103a9c2ecc4f026efbd4b40662ce22dc1f824fb129ed121460aaa5c47f8
Status: Downloaded newer image for registry.k8s.io/ingress-
nginx/controller:v1.2.1
registry.k8s.io/ingress-nginx/controller:v1.2.1
```

La imagen existe, y el SHA256 es:

```
sha256:5516d103a9c2ecc4f026efbd4b40662ce22dc1f824fb129ed121460aaa5c47f8
```

2.2. Limpieza

```
docker image rm registry.k8s.io/ingress-nginx/controller:v1.2.1
```

3. (k8s server) Actualizar la versión de la imagen del daemonset (paso 1.2.) y el SHA256 (paso 2.1.)

3.1. Actualizar el daemonset:

```
kubectl set image daemonsets/ingress-nginx-controller \
  controller=registry.k8s.io/ingress-
nginx/controller:v1.2.1@sha256:5516d103a9c2ecc4f026efbd4b40662ce22dc1f824fb1
29ed121460aaa5c47f8 \
  -n ingress-nginx
```

3.2. Verificar:

```
kubectl get pods -n ingress-nginx -o wide
```

ETA: 3'

Resultado esperado similar a:

```
NAME                                   READY   STATUS      RESTARTS   AGE
IP            NODE
   NOMINATED NODE   READINESS GATES
ingress-nginx-admission-create-zxqvn   0/1     Completed   0          139m
10.244.0.75   k8s1
   <none>           <none>
ingress-nginx-admission-patch-c9t4w    0/1     Completed   1          139m
10.244.0.74   k8s1
   <none>           <none>
ingress-nginx-controller-6thbj         1/1     Running     0          2m8s
10.244.0.77   k8s1
   <none>           <none>
ingress-nginx-controller-tv9zd         1/1     Running     0          63s
10.244.1.16   k8s3
   <none>           <none>
ingress-nginx-controller-zgfw5         1/1     Running     0          95s
10.244.2.21   k8s2
   <none>           <none>
```

Nginx LoadBalancer

En este ejemplo asumimos que tenemos un LoadBalancer habilitado, porque hemos instalado un balanceador de carga por software.

1. Conectarse al master:

```
ssh master
```

2. Obtener el manifiesto:

```
wget
https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v0.46.
0/deploy/static/provider/baremetal/deploy.yaml
```

3. Editarlo:

```
vim deploy.yaml
```

Y substituir “type: NodePort”:

```
# Source: ingress-nginx/templates/controller-service.yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
```

```
  labels:
    helm.sh/chart: ingress-nginx-3.30.0
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/version: 0.46.0
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/component: controller
  name: ingress-nginx-controller
  namespace: ingress-nginx
spec:
  type: NodePort
```

por “type: LoadBalancer”:

```
# Source: ingress-nginx/templates/controller-service.yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
  labels:
    helm.sh/chart: ingress-nginx-3.30.0
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/instance: ingress-nginx
    app.kubernetes.io/version: 0.46.0
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/component: controller
  name: ingress-nginx-controller
  namespace: ingress-nginx
spec:
  type: LoadBalancer
```

4. Instalar el manifiesto:

```
kubectl apply -f deploy.yaml
```

5. Probar.

5.1. Ejecutar:

```
kubectl get pods -n ingress-nginx \
-l app.kubernetes.io/name=ingress-nginx --watch
```

Hasta que el 'controller' esté corriendo:

```
NAME                                        READY   STATUS      RESTARTS
AGE
ingress-nginx-admission-create-jw7f8        0/1     Completed   0
4d18h
ingress-nginx-admission-patch-b99q4         0/1     Completed   0
4d18h
ingress-nginx-controller-55bc4f5576-77bh8   1/1     Running     2
```

```
4d18h
```

5.2. Ejecutar:

```
kubectl get services --field-selector metadata.name=ingress-nginx-controller
-n ingress-nginx
```

El resultado esperado es similar a:

```
NAME                       TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)                      AGE
ingress-nginx-controller   LoadBalancer   10.109.52.158   192.168.95.68
80:31936/TCP,443:31079/TCP   4d18h
```

Vemos que ha tomado una 'EXTERNAL-IP' del pool de direcciones definida en el pool de direcciones de metallb.

5.3. Ejecutar:

```
POD_NAMESPACE=ingress-nginx
POD_NAME=$(kubectl get pods -n $POD_NAMESPACE -l
app.kubernetes.io/name=ingress-nginx --field-selector=status.phase=Running -
o jsonpath='{.items[0].metadata.name}')
kubectl exec -it $POD_NAME -n $POD_NAMESPACE -- /nginx-ingress-controller --
version
```

Resultado esperado similar a:

```
-------------------------------------------------------------------------------
---
NGINX Ingress controller
  Release:       v0.46.0
  Build:         6348dde672588d5495f70ec77257c230dc8da134
  Repository:    https://github.com/kubernetes/ingress-nginx
  nginx version: nginx/1.19.6

-------------------------------------------------------------------------------
---
```

Por tanto en este ejemplo las peticiones que lleguen a la IP “192.168.95.68” serán enrutaddas en función de las reglas que se definan en el ingress (ver más adelante).

Ejemplo ingress

Este ejemplo requiere:

- Un Service cuyo 'metadata.name' sea 'service-whoami'
- Tener instalado un Ingress controller, como Nginx

1. Conectarse al master

```
ssh master
```

2. Ejecutar:

```
cat <<EOF | kubectl apply -f -
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: nginx
  name: ingress-whoami
spec:
  rules:
  - host: example.com
    http:
      paths:
      - backend:
          service:
            name: service-whoami
            port:
              number: 80
        path: /
        pathType: Prefix
EOF
```

3. Verificar

```
kubectl get ingress --field-selector metadata.name=ingress-whoami
```

Resultado esperado similar a:

```
NAME             CLASS    HOSTS           ADDRESS          PORTS     AGE
ingress-whoami   <none>   k8s.kedu.coop   192.168.95.73   80, 443   42h
```

Con esta configuración:

- Las peticiones que le lleguen a la IP que tiene le servicio 'ingress-nginx-controller', en este ejemplo la '192.168.95.68' pasarán por las reglas definidas en este 'ingress'
- Si se recibe una petición HTTP en el puerto TCP 80 o TCP 443 con un 'header' con clave 'Host' y valor 'example.com' se enrutará al servicio con nombre 'service-whoami'

Ejemplo ingress con letsencrypt

Este ejemplo requiere:

- Un Service cuyo 'metadata.name' sea 'service-whoami'
- Tener instalado un Ingress controller, como Nginx

- Tener instalado el addon Cert-manager
- Tener instalado un issuer llamado 'letsencrypt-production'
- Edge router:
  - Nginx
  - Reverse proxy a ingress-nginx-controller 'EXTERNAL-IP'
  - Que no termine las conexiones SSL, es decr, que redirija en capa 4 lo que le llegue a TCP 443
- Un nombre DNS, en este ejemplo “example.com”, que apunte a la IP pública del edge router, en este ejemplo '8.8.8.8'

1. Conectarse al master:

```
ssh master
```

2. Ejecutar:

```
cat <<EOF | kubectl apply -f -
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    # Should match 'Issuer.metadata.name'
    cert-manager.io/issuer: letsencrypt-production
    kubernetes.io/ingress.class: nginx
  name: ingress-whoami
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: service-whoami
            port:
              number: 80
  tls:
  - hosts:
    - example.com
    secretName: quickstart-example-tls
EOF
```

3. Comprobar

3.1. Ejecutar:

```
watch -n 5 kubectl get certificate
```

El resultado esperado, tras aproximadamente un minuto, es:

```
NAME                     READY   SECRET                   AGE
quickstart-example-tls   True    quickstart-example-tls   63s
```

3.2. Se ha creado un secret llamado 'quickstart-example-tls'. Para ver lo que contiene ejecutar:

```
kubectl describe secrets quickstart-example-tls
```

Salida esperada similar a:

```
Name:         quickstart-example-tls
Namespace:    default
Labels:       <none>
Annotations:  cert-manager.io/alt-names: k8s.kedu.coop
              cert-manager.io/certificate-name: quickstart-example-tls
              cert-manager.io/common-name: k8s.kedu.coop
              cert-manager.io/ip-sans:
              cert-manager.io/issuer-group: cert-manager.io
              cert-manager.io/issuer-kind: Issuer
              cert-manager.io/issuer-name: letsencrypt-production
              cert-manager.io/uri-sans:

Type:  kubernetes.io/tls

Data
====
tls.crt:  5587 bytes
tls.key:  1675 bytes
```

3.3. Abrir un navegador e ir a:

https://example.com

Debería aparecer el contenido servido por el deployment 'deployment-whoami' con un certificado SSL válido.

Ejemplo ingress con modsecurity

Requisitos:

- Crear un servicio “service-echoserver”

```
cat <<EOF | kubectl apply -f -
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    #
https://awkwardferny.medium.com/enabling-modsecurity-in-the-kubernetes-ingress-nginx-controller-111f9c877998
```

```
    nginx.ingress.kubernetes.io/enable-modsecurity: "true"
    nginx.ingress.kubernetes.io/modsecurity-snippet: |
      SecRuleEngine On
      SecRequestBodyAccess On
      SecAuditEngine RelevantOnly
      SecAuditLogParts ABIJDEFHZ
      SecAuditLog /var/log/modsec_audit.log
      SecRule REQUEST_HEADERS:User-Agent \"fern-scanner\"
\"log,deny,id:107,status:403,msg:\'Fern Scanner Identified\'\"
  name: ingress-echoserver
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: service-echoserver
            port:
              number: 80
EOF
```

Probar:

```
curl http://example.com -k -H "user-agent: fern-scanner"
```

Respuesta esperada: 403

ConfigMaps

ConfigMaps

Editar ConfigMaps para habilitar 'use-proxy-protocol'

IMPORTANTE: si habilitamos proxy protocol mediante configmap lo haremos tanto para HTTP como para HTTPS. Eso quiere decir que el “edge router” que esté por delante del cluster de kubernetes debe tener esos protocolos activados. En el ejemplo que hay en esta página NO lo está (en el edge router) para HTTP, pero sigue funcionando porque la aplicación (whoami) redirige el tráfico de HTTP a HTTPS. Pero si se hace con curl (está documentado en esta página) fallará. Para que funcione (el curl), tal y como está documentado, habrá que deshabilitar momentáneamente proxy protocol, hacer la prueba con curl, y volver a habilitar. Ver 7194

En este ejemplo:

- Vamos a editar un ConfigMap
- Vamos a ver los cambios que se aplican inmediatamente en el pod

1. Identificar el ConfigMap. En nuestro caso queremos modificar este 'ConfigMap', por lo que filtramos por el 'Namespace' llamado 'ingress-controller':

```
kubectl get configmaps -n ingress-nginx
```

Resultado esperado similar a:

```
NAME                              DATA   AGE
ingress-controller-leader-nginx   0      5d
ingress-nginx-controller          0      5d
kube-root-ca.crt                  1      5d
```

En nuestro caso es ingress-nginx-controller .

2. Editamos el 'ConfigMap':

```
kubectl edit configmaps -n ingress-nginx ingress-nginx-controller
```

Vamos a seguir estas instrucciones, por lo que le vamos a añadir una sección 'data' que incluya use-proxy-protocol:

```
data:
  use-proxy-protocol: "true"
```

3. Comprobar

3.1. Buscar el pod que usa ese 'ConfigMap'. En este caso hacemos un poco de atajo:

```
kubectl get pods -n ingress-nginx
```

Resultado esperado similar a:

```
NAME                                        READY   STATUS      RESTARTS
AGE
ingress-nginx-admission-create-glxnx        0/1     Completed   0
12h
ingress-nginx-admission-patch-njb2p         0/1     Completed   0
12h
ingress-nginx-controller-598dd75597-lkpp8   1/1     Running     0
138m
ingress-nginx-controller-598dd75597-s9kwg   1/1     Running     0
137m
```

3.2. Ver los logs

```
kubectl logs -f -n ingress-nginx ingress-nginx-controller-598dd75597-lkpp8
```

Resultado similar a:

```
I0520 15:47:25.546777       9 event.go:282]
Event(v1.ObjectReference{Kind:"ConfigMap", Namespace:"ingress-nginx",
Name:"ingress-nginx-controller", UID:"e84d9bc4-bee4-47f0-a687-bac67a95c8d0",
APIVersion:"v1", ResourceVersion:"69461", FieldPath:""}): type: 'Normal'
reason: 'UPDATE' ConfigMap ingress-nginx/ingress-nginx-controller
I0520 15:47:25.549723       9 controller.go:146] "Configuration changes
detected, backend reload required"
I0520 15:47:25.618572       9 controller.go:163] "Backend successfully
reloaded"
I0520 15:47:25.623095       9 event.go:282]
Event(v1.ObjectReference{Kind:"Pod", Namespace:"ingress-nginx",
Name:"ingress-nginx-controller-598dd75597-lkpp8",
UID:"eda84159-3520-48be-8220-c4e0491afb07", APIVersion:"v1",
ResourceVersion:"60184", FieldPath:""}): type: 'Normal' reason: 'RELOAD'
NGINX reload triggered due to a change in configuration
```

3.3. Comprobar si se ha añadido un setting proxy_protocol por algún lado. Ejecutar:

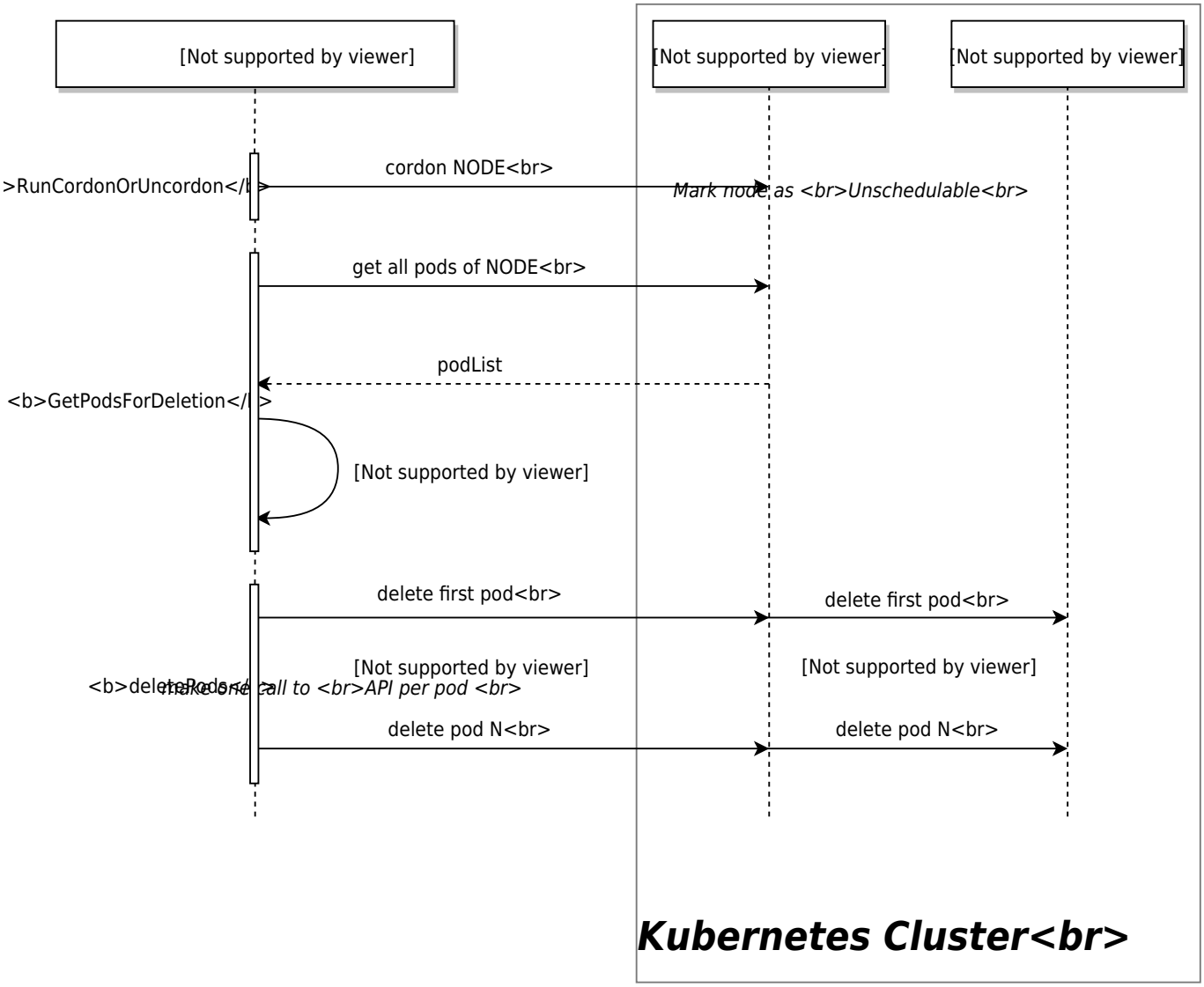
```
kubectl exec -ti -n ingress-nginx ingress-nginx-controller-598dd75597-lkpp8
-- cat /etc/nginx/nginx.conf | grep proxy_protocol | grep listen
```

Salida esperada similar a:

```
        listen 80 proxy_protocol default_server reuseport backlog=511 ;
        listen 443 proxy_protocol default_server reuseport backlog=511 ssl
http2 ;
        listen 80 proxy_protocol ;
        listen 443 proxy_protocol ssl http2 ;
```

Por tanto el 'ConfigMap' ha funcionado.

Desconectar nodo



Worker

1. Conectarnos a un control plane

```
ssh k8s2
```

2. Listar nodos:

```
kubectl get nodes
```

Resultado esperado similar a:

```
NAME   STATUS   ROLES                  AGE     VERSION
k8s2   Ready    control-plane,master   5d14h   v1.21.1
k8s3   Ready    control-plane,master   5d14h   v1.21.1
k8s4   Ready    control-plane,master   5d14h   v1.21.1
k8s5   Ready    <none>                 4d18h   v1.21.1
k8s6   Ready    <none>                 4d17h   v1.21.1
k8s7   Ready    <none>                 4d17h   v1.21.1
```

3. Vaciar de pods el worker node:

```
kubectl drain k8s7
```

Em este caso nos aparece el siguiente error:

```
node/k8s7 cordoned
error: unable to drain node "k8s7", aborting command...

There are pending nodes to be drained:
 k8s7
error: cannot delete DaemonSet-managed Pods (use --ignore-daemonsets to
ignore): kube-system/kube-flannel-ds-n47nd, kube-system/kube-proxy-cghwq
```

4. Reintentar el comando anterior añadiendo algunos parámetros:

```
kubectl drain k8s7 --ignore-daemonsets
```

Resultado esperado similar a:

```
node/k8s7 already cordoned
WARNING: ignoring DaemonSet-managed Pods: kube-system/kube-flannel-ds-n47nd,
kube-system/kube-proxy-cghwq
evicting pod ingress-nginx/ingress-nginx-controller-55bc4f5576-78rmm
evicting pod cert-manager/cert-manager-7dd5854bb4-td892
evicting pod cert-manager/cert-manager-webhook-6b57b9b886-pwhhq
evicting pod ingress-nginx/ingress-nginx-admission-patch-t4dm6
pod/ingress-nginx-admission-patch-t4dm6 evicted
pod/cert-manager-7dd5854bb4-td892 evicted
pod/cert-manager-webhook-6b57b9b886-pwhhq evicted
pod/ingress-nginx-controller-55bc4f5576-78rmm evicted
node/k8s7 evicted
```

En este punto:

- El worker 'k8s7' no acepta que se programen despliegues de nuevos pods
- Todavía está corriendo pods de tipo DaemonSet

5. (Opcional) Comprobar el estado de los nodos:

```
kubectl get nodes
```

Resultado esperado similar a:

```
NAME   STATUS                     ROLES                  AGE     VERSION
k8s2   Ready                      control-plane,master   5d15h   v1.21.1
k8s3   Ready                      control-plane,master   5d14h   v1.21.1
k8s4   Ready                      control-plane,master   5d14h   v1.21.1
k8s5   Ready                      <none>                 4d18h   v1.21.1
k8s6   Ready                      <none>                 4d18h   v1.21.1
```

```
k8s7   Ready,SchedulingDisabled   <none>                 4d17h   v1.21.1
```

6. (Opcional) Listar los pods que todavía están corriendo en el worker 'k8s7':

```
kubectl get pods --all-namespaces -o wide | grep k8s7
```

Resultado esperado similar a:

```
kube-system     kube-flannel-ds-n47nd                      1/1     Running
0          4d17h   192.168.95.76   k8s7     <none>           <none>
kube-system     kube-proxy-cghwq                           1/1     Running
0          4d17h   192.168.95.76   k8s7     <none>           <none>
```

7. Eliminar el nodo

```
kubectl delete node k8s7
```

Resultado esperado similar a:

```
node "k8s7" deleted
```

8. Comprobar:

kubectl get nodes -w

Después de un minuto aproximadamente habrá desaparecido:

```
NAME   STATUS   ROLES                  AGE     VERSION
k8s2   Ready    control-plane,master   5d15h   v1.21.1
k8s3   Ready    control-plane,master   5d15h   v1.21.1
k8s4   Ready    control-plane,master   5d14h   v1.21.1
k8s5   Ready    <none>                 4d18h   v1.21.1
k8s6   Ready    <none>                 4d18h   v1.21.1
```

Definir tiempo pod salta a otro worker

Por defecto un pod saltará a otro worker tras **5 minutos** de que el worker esté marcado como “NotReady”.

AVISO: el parámetro pod-eviction-timeout parece ser que no funciona. Ver #74651 y #7112

1. Conectarse al control plane

```
ssh k8s2
```

2. Realizar una copia de seguridad:

```
sudo cp /etc/kubernetes/manifests/kube-apiserver.yaml ~
```

3. Editar:

```
sudo vim /etc/kubernetes/manifests/kube-apiserver.yaml
```

Y añadir '.spec.containers.0.command':

```
    # anyadido
    #- --enable-admission-plugins=NodeRestriction
    - --enable-admission-plugins=NodeRestriction,DefaultTolerationSeconds
    - --default-not-ready-toleration-seconds=30
    - --default-unreachable-toleration-seconds=30
```

Ejemplo:

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint:
192.168.95.71:6443
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=192.168.95.71
    # anyadido
    - --default-not-ready-toleration-seconds=30
    - --default-unreachable-toleration-seconds=30
...
```

4. Repetir los pasos 1 a 3 para el resto de control plane, en el caso de que haya más por ser un entorno de alta disponibilidad

5. Comprobar

5.1. Comprobar que todos y cada uno de los control plane tienen un pod 'kube-apiserver' corriendo:

```
watch -n 5 "kubectl get pods  --all-namespaces  | grep kube-api"
```

Resultado esperado similar a:

```
kube-system     kube-apiserver-k8s2                          1/1     Running
0          43s
kube-system     kube-apiserver-k8s3                          1/1     Running
3          5d20h
```

```
kube-system     kube-apiserver-k8s4                          1/1     Running
3          5d20h
```

5.2. Identificar los pods que corren en un worker:

```
kubectl get pods -o wide
```

Resultado esperado similar a:

```
NAME                                    READY   STATUS    RESTARTS   AGE
IP            NODE   NOMINATED NODE   READINESS GATES
deployment-whoami-6b6dc7c84-2swm9       1/1     Running   0          101m
10.244.3.37   k8s5   <none>           <none>
```

5.3. Detener el worker donde está corriendo el pod

5.3.1. Conectarse al worker

```
ssh k8s5
```

5.3.2. Apagarlo

```
sudo shutdown -h now
```

5.3.3. Anotar la hora

```
date
```

5.4. Monitorizar nodos

```
watch -n 5 kubectl get nodes
```

Resultado esperado similar a **tras aproximadamente 30 segundos**:

```
NAME   STATUS     ROLES                  AGE     VERSION
k8s2   Ready      control-plane,master   5d22h   v1.21.1
k8s3   Ready      control-plane,master   5d22h   v1.21.1
k8s4   Ready      control-plane,master   5d22h   v1.21.1
k8s5   NotReady   <none>                 5d2h    v1.21.1
k8s6   Ready      <none>                 5d1h    v1.21.1
```

5.5. Monitorizar pods

```
watch -n 5 kubectl get pods -o wide
```

Resultado esperado similar a **tras aproximadamente 60 segundos**:

```
NAME                                    READY   STATUS        RESTARTS   AGE
IP            N
ODE   NOMINATED NODE   READINESS GATES
```

```
deployment-whoami-6b6dc7c84-2swm9       1/1     Terminating   0
104m   10.244.3.37   k
8s5   <none>           <none>
deployment-whoami-6b6dc7c84-mrjjw       1/1     Running       0          19s
10.244.4.31   k
8s6   <none>           <none>
```

LOCAL

Bajamos código fuente de kubernetes e instalamos:

```
# git clone https://github.com/kubernetes/kubernetes
# cd kubernetes
# make quick-release
```

Ahora para acabar de instalar hay que decir en que proveedor crea las máquinas virtuales. Lista sacada de https://get.k8s.io/

```
Google Compute Engine [default]
KUBERNETES_PROVIDER=gce

Google Container Engine
KUBERNETES_PROVIDER=gke

Amazon EC2
KUBERNETES_PROVIDER=aws

Libvirt (with CoreOS as a guest operating system)
KUBERNETES_PROVIDER=libvirt-coreos

Microsoft Azure
KUBERNETES_PROVIDER=azure-legacy

Vagrant (local virtual machines)
KUBERNETES_PROVIDER=vagrant

VMWare Photon Controller
KUBERNETES_PROVIDER=photon-controller

Rackspace
KUBERNETES_PROVIDER=rackspace

OpenStack-Heat
KUBERNETES_PROVIDER=openstack-heat
```

En nuestro caso elegimos vagrant que es virtualbox. Instalamos virtualbox:

```
# apt-get install vagrant virtualbox
```

Para que nos cree las dos máquinas virtuales de virtualbox y configure kubernetes ejecutamos este script que se baja los binarios:

```
# export KUBERNETES_PROVIDER=vagrant
# wget -q -O - https://get.k8s.io | bash
```

Nos dice que pongamos la ruta en el PATH y volvamos a ejecutarlo:

```
Add '/home/jose/kubernetes/kubernetes/client/bin' to your PATH to use newly-
installed binaries.
```

Lo ponemos y ejecutamos:

```
# export PATH=$PATH:/home/jose/kubernetes/kubernetes/client/bin
# wget -q -O - https://get.k8s.io | bash
```

Crea dos máquinas con fedora:

```
# VBoxManage list vms
"kubernetes_master_1486996921273_58743"
{0cae89a8-08f2-4c48-8f02-1ca5df33c5b9}
"kubernetes_node-1_1486997158445_96359" {11a27a45-3683-4173-
b746-2eb111ad7915}
```

La instalación acabo con:

```
Each machine instance has been created/updated.
  Now waiting for the Salt provisioning process to complete on each machine.
  This can take some time based on your network, disk, and cpu speed.
  It is possible for an error to occur during Salt provision of cluster and
this could loop forever.
Validating master
.................................
```

Parece que se queda colgado.....

Lo lanzo en casa y funciona:

```
Each machine instance has been created/updated.
  Now waiting for the Salt provisioning process to complete on each machine.
  This can take some time based on your network, disk, and cpu speed.
  It is possible for an error to occur during Salt provision of cluster and
this could loop forever.
Validating master
Validating node-1

Waiting for each node to be registered with cloud provider
Flag --api-version has been deprecated, flag is no longer respected and will
be deleted in the next release
Validating we can run kubectl commands.
```

```
No resources found.

Kubernetes cluster is running.

The master is running at:

  https://10.245.1.2

Administer and visualize its resources using Cockpit:

  https://10.245.1.2:9090

For more information on Cockpit, visit http://cockpit-project.org

The user name and password to use is located in /root/.kube/config

... calling validate-cluster
Found 1 node(s).
NAME                STATUS    AGE
kubernetes-node-1   Ready     38s
Validate output:
NAME                 STATUS    MESSAGE              ERROR
scheduler            Healthy   ok
controller-manager   Healthy   ok
etcd-0               Healthy   {"health": "true"}
etcd-1               Healthy   {"health": "true"}
Cluster validation succeeded
Done, listing cluster services:

Kubernetes master is running at https://10.245.1.2
Heapster is running at
https://10.245.1.2/api/v1/proxy/namespaces/kube-system/services/heapster
KubeDNS is running at
https://10.245.1.2/api/v1/proxy/namespaces/kube-system/services/kube-dns
kubernetes-dashboard is running at
https://10.245.1.2/api/v1/proxy/namespaces/kube-system/services/kubernetes-d
ashboard
Grafana is running at
https://10.245.1.2/api/v1/proxy/namespaces/kube-system/services/monitoring-g
rafana
InfluxDB is running at
https://10.245.1.2/api/v1/proxy/namespaces/kube-system/services/monitoring-i
nfluxdb

To further debug and diagnose cluster problems, use 'kubectl cluster-info
dump'.

Kubernetes binaries at /home/jose/kubernetes/kubernetes/cluster/
You may want to add this directory to your PATH in $HOME/.profile
Installation successful!
```

Arrancamos el proxy y ya podemos acceder a la UI del dashboard de Kubernetes:

```
# ./platforms/linux/amd64/kubectl proxy

Starting to serve on 127.0.0.1:8001
```

```
http://127.0.0.1:8001/ui
```

```
usuario: vagrant
contraseña: vagrant
```

MINIKUBE

https://kubernetes.io/docs/tasks/tools/install-minikube/

1. Instalamos kubectl

https://kubernetes.io/docs/tasks/tools/install-kubectl/#install-kubectl-binary-via-curl

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/li
nux/amd64/kubectl
```

Lo ponemos en el path:

```
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl
```

2. Instalamos minikube https://github.com/kubernetes/minikube/releases

Ojo que puede cambiar versión, id a web:

```
curl -Lo minikube
https://storage.googleapis.com/minikube/releases/v0.20.0/minikube-linux-amd6
4 && chmod +x minikube && sudo mv minikube /usr/local/bin/
```

Creamos el cluster. Primero crea la VM en virtualbox y luego la levanta y configura los servicios:

```
# minikube start

Starting local Kubernetes cluster...
Downloading Minikube ISO
84.07 MB / 84.07 MB [============================================] 100.00%
0s
Kubectl is now configured to use the cluster.
```

```
# vboxmanage list vms
```

```
"minikube" {e1e5e6f9-e3cc-437d-81f8-80a038f15ac5}
```

Para conectar a la VM de virtualbox llamada minikube:

```
ssh docker@192.168.99.100
docker/tcuser
```

Miramos que tiene desplegado:

```
# kubectl get pods --all-namespaces

NAMESPACE     NAME                              READY     STATUS
RESTARTS   AGE
kube-system   kube-addon-manager-minikube   0/1       ContainerCreating   0
3m
```

Instalamos el dashboard:

```
# kubectl create -f
https://rawgit.com/kubernetes/dashboard/master/src/deploy/kubernetes-dashboa
rd.yaml

deployment "kubernetes-dashboard" created
service "kubernetes-dashboard" created
```

```
# kubectl get services --all-namespaces

NAMESPACE     NAME                   CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
default       kubernetes             10.0.0.1     <none>        443/TCP
4m
kube-system   kubernetes-dashboard   10.0.0.66    <nodes>       80:32220/TCP
21s
```

Dashboard:

```
minikube dashboard
```

Te abre http://192.168.99.100:30000

También he hecho un tunel 8080:

```
ssh -L 8080:localhost:8080 docker@192.168.99.100
```

Url de acceso: http://localhost:8080/ui

Autoescalado

https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

http://blog.kubernetes.io/2016/07/autoscaling-in-kubernetes.html

https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/

https://github.com/kubernetes/heapster/blob/master/docs/influxdb.md

Autoescalamos la aplicación:

```
kubectl autoscale deployment apache --min=2 --max=5 --cpu-percent=20
```

```
deployment "apache" autoscaled
```

Vemos el autoescalado. Desde la UI se ve en deployments:

```
kubectl get hpa
```

```
NAME      REFERENCE           TARGETS          MINPODS   MAXPODS   REPLICAS
AGE
apache    Deployment/apache   <unknown> / 1%   2         5         2
6m
```

Con mas detalle:

```
kubectl describe hpa apache
```

```
Name:                           apache
Namespace:                      default
Labels:                         <none>
Annotations:                        <none>
CreationTimestamp:                  Fri, 23 Jun 2017 17:51:17 +0200
Reference:                      Deployment/apache
Metrics:                        ( current / target )
  resource cpu on pods  (as a percentage of request):   <unknown> / 1%
Min replicas:                       2
Max replicas:                       5
Events:
  FirstSeen LastSeen    Count    From                SubObjectPath    Type
Reason                Message
  --------- --------    -----    ----                -------------    ------
--    ------                -------
  6m        6m        1    horizontal-pod-autoscaler            Normal
SuccessfulRescale         New size: 2; reason: Current number of replicas
below Spec.MinReplicas
  5m        27s        12    horizontal-pod-autoscaler            Warning
FailedGetResourceMetric         unable to get metrics for resource cpu:
```

```
failed to get pod resource metrics: the server could not find the requested
resource (get services http:heapster:)
  5m        27s        12    horizontal-pod-autoscaler           Warning
FailedComputeMetricsReplicas    failed to get cpu utilization: unable to get
metrics for resource cpu: failed to get pod resource metrics: the server
could not find the requested resource (get services http:heapster:)
```

Daba el error:

```
kubectl describe hpa
```

```
35m     18s        72    horizontal-pod-autoscaler           Warning
FailedComputeMetricsReplicas    failed to get cpu utilization: missing
request for cpu on container httpd in pod default/httpd-796666570-2h1c6
```

El problema es que en el deployment hay que decir que quiere monitorizar, en mi caso dentro de resources he añaido las dos líneas de requests: cpu:400m

```
......
spec:
  containers:
    name: httpd
    image: httpd
    resources:
      requests:
        cpu:400m
......
```

Después da este error, pero tarda un poco en coger las métricas de heapster:

```
kubectl describe hpa
```

```
2m      1m        3    horizontal-pod-autoscaler           Warning
FailedComputeMetricsReplicas    failed to get cpu utilization: unable to get
metrics for resource cpu: no metrics returned from heapster
```

LOCAL kubeadm

Instalamos docker

```
apt-get install apt-transport-https ca-certificates curl gnupg2 software-
properties-common
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/debian $(lsb_release -cs) stable"
```

Local en Centos

https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/

Installing kubelet and kubeadm

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
        https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
setenforce 0
yum install -y docker kubelet kubeadm kubernetes-cni
systemctl enable docker && systemctl start docker
systemctl enable kubelet && systemctl start kubelet
```

Debian

```
apt-get update && apt-get install -y apt-transport-https
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add
-
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
# Install docker if you don't have it already.
apt-get install -y docker-engine
apt-get install -y kubelet kubeadm kubernetes-cni
```

Arrancamos kubeadm

```
kubeadm init
```

```
Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run (as a regular user):

  sudo cp /etc/kubernetes/admin.conf $HOME/
  sudo chown $(id -u):$(id -g) $HOME/admin.conf
  export KUBECONFIG=$HOME/admin.conf
```

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

You can now join any number of machines by running the following on each
node
as root:

  kubeadm join --token bad112.b2514df7739b0845 192.168.1.133:6443
```

Como usuario normal ejecutamos:

```
sudo cp /etc/kubernetes/admin.conf $HOME/
sudo chown $(id -u):$(id -g) $HOME/admin.conf
export KUBECONFIG=$HOME/admin.conf
```

Ahora podemos ver los pods:

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

Debug errores:

```
 sudo journalctl -r -u kubelet
```

Vemos que el pod de dns se queda colgado, hay que configurar la RED

Plugin RED https://www.weave.works/docs/net/latest/kube-addon/

```
kubectl apply -n kube-system -f
"https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 |
tr -d '\n')"
```

Ahora miramos que todos los pods están activos

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

```
NAMESPACE     NAME                              READY     STATUS    RESTARTS
AGE
kube-system   etcd-avtp110                      1/1       Running   0
26m
kube-system   kube-apiserver-avtp110            1/1       Running   0
26m
kube-system   kube-controller-manager-avtp110   1/1       Running   0
26m
kube-system   kube-dns-692378583-dn3qg          3/3       Running   0
26m
kube-system   kube-proxy-vdpmv                  1/1       Running   0
26m
kube-system   kube-scheduler-avtp110            1/1       Running   0
26m
```

```
kube-system   weave-net-1s7wd                    2/2       Running   0
2m
```

Vemos que el nodo está funcionando

```
kubectl get nodes
NAME      STATUS    AGE       VERSION
avtp110   Ready     25m       v1.6.6
```

Instalamos dashboard

```
kubectl create -f
https://rawgit.com/kubernetes/dashboard/master/src/deploy/kubernetes-dashboa
rd.yaml
```

Para mirar el estado:

```
kubectl describe services kubernetes-dashboard --namespace=kube-system
```

Arrancamos proxy para poder acceder:

```
kubectl proxy
Starting to serve on 127.0.0.1:8001
```

Ya se puede acceder en http://127.0.0.1:8001/ui

Al desplegar cualquier cosa me da error:

```
No nodes are available that match all of the following predicates::
PodToleratesNodeTaints (1).
```

Solución:

```
kubectl describe node
```

```
Name:           avtp110
Role:
Labels:         beta.kubernetes.io/arch=amd64
            beta.kubernetes.io/os=linux
            kubernetes.io/hostname=avtp110
            node-role.kubernetes.io/master=
Annotations:        node.alpha.kubernetes.io/ttl=0
            volumes.kubernetes.io/controller-managed-attach-detach=true
Taints:         node-role.kubernetes.io/master:NoSchedule
CreationTimestamp:  Sat, 24 Jun 2017 00:33:04 +0200
Phase:
Conditions:
  Type          Status    LastHeartbeatTime           LastTransitionTime
Reason              Message
  ----          ------    -----------------           ------------------
```

```
------                  -------
  OutOfDisk         False     Sat, 24 Jun 2017 01:00:18 +0200     Sat, 24
Jun 2017 00:33:04 +0200     KubeletHasSufficientDisk     kubelet has
sufficient disk space available
  MemoryPressure    False     Sat, 24 Jun 2017 01:00:18 +0200     Sat, 24
Jun 2017 00:33:04 +0200     KubeletHasSufficientMemory     kubelet has
sufficient memory available
  DiskPressure      False     Sat, 24 Jun 2017 01:00:18 +0200     Sat, 24
Jun 2017 00:33:04 +0200     KubeletHasNoDiskPressure     kubelet has no disk
pressure
  Ready         True     Sat, 24 Jun 2017 01:00:18 +0200     Sat, 24 Jun
2017 00:34:54 +0200     KubeletReady             kubelet is posting ready
status
Addresses:      192.168.1.133,192.168.1.133,avtp110
Capacity:
 cpu:       4
 memory:    20445912Ki
 pods:      110
Allocatable:
 cpu:       4
 memory:    20343512Ki
 pods:      110
System Info:
 Machine ID:            72d18515f62d4c7186558889b32712f9
 System UUID:           4C4C4544-0035-3010-8042-B8C04F504332
 Boot ID:           c529b79b-7f1c-4e1c-95f6-6187bdc57f6a
 Kernel Version:        4.9.0-3-amd64
 OS Image:          Debian GNU/Linux 9 (stretch)
 Operating System:      linux
 Architecture:          amd64
 Container Runtime Version: docker://1.11.2
 Kubelet Version:       v1.6.6
 Kube-Proxy Version:        v1.6.6
ExternalID:         avtp110
Non-terminated Pods:        (8 in total)
  Namespace         Name                          CPU Requests    CPU Limits
Memory Requests    Memory Limits
  ---------         ----                          ------------    ----------
---------------    -------------
  kube-system           etcd-avtp110                    0 (0%)        0 (0%)
0 (0%)        0 (0%)
  kube-system           kube-apiserver-avtp110                250m (6%)    0
(0%)        0 (0%)        0 (0%)
  kube-system           kube-controller-manager-avtp110            200m (5%)
0 (0%)        0 (0%)        0 (0%)
  kube-system           kube-dns-692378583-f0sk0            260m (6%)    0
(0%)        110Mi (0%)    170Mi (0%)
  kube-system           kube-proxy-1j5hd                0 (0%)        0 (0%)
0 (0%)        0 (0%)
  kube-system           kube-scheduler-avtp110                100m (2%)    0
(0%)        0 (0%)        0 (0%)
```

```
  kube-system           kubernetes-dashboard-2039414953-s1k99        0 (0%)
0 (0%)        0 (0%)        0 (0%)
  kube-system           weave-net-sw6k7                     20m (0%)    0
(0%)        0 (0%)        0 (0%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits    Memory Requests    Memory Limits
  ------------  ----------    ---------------    -------------
  830m (20%)    0 (0%)        110Mi (0%)    170Mi (0%)
Events:
  FirstSeen LastSeen    Count    From            SubObjectPath    Type
Reason            Message
  --------- --------    -----    ----            -------------    --------
------            -------
  27m       27m         1    kubelet, avtp110            Normal
Starting        Starting kubelet.
  27m       27m         1    kubelet, avtp110            Warning
ImageGCFailed        unable to find data for container /
  27m       27m         28    kubelet, avtp110            Normal
NodeHasSufficientDisk    Node avtp110 status is now: NodeHasSufficientDisk
  27m       27m         28    kubelet, avtp110            Normal
NodeHasSufficientMemory    Node avtp110 status is now:
NodeHasSufficientMemory
  27m       27m         28    kubelet, avtp110            Normal
NodeHasNoDiskPressure    Node avtp110 status is now: NodeHasNoDiskPressure
  27m       27m         1    kube-proxy, avtp110            Normal
Starting        Starting kube-proxy.
  25m       25m         1    kubelet, avtp110            Normal
NodeReady        Node avtp110 status is now: NodeReady
```

Nos fijamos en la línea:

```
Taints: node-role.kubernetes.io/master:NoSchedule
```

Lanzamos el comando:

```
kubectl taint nodes avtp110 node-role.kubernetes.io/master:NoSchedule-
node "avtp110" tainted
```

Para monitorización:

```
kubectl create -f
https://raw.githubusercontent.com/luxas/kubeadm-workshop/master/demos/monito
ring/heapster.yaml
```

Balanceador de carga

La gracia de Kubernetes es que no hay que poner ips, se configura por nombres.

Por ejemplo, deplegamos un apache y lo llamamos httpd. Si levantamos varios pods serán accesibles a sus ips por el puerto 80

Podemos crear un balanceador con el siguiente fichero apuntado a la aplicación httpd:

```
    {
      "kind": "Service",
      "apiVersion": "v1",
      "metadata": {
        "name": "ejemplo-balanceador"
      },
      "spec": {
        "ports": [{
          "port": 8080,
          "targetPort": 80
        }],
        "selector": {
          "app": "httpd"
        },
        "type": "LoadBalancer"
      }
    }
```

Lo desplegamos en kubernetes:

```
kubectl create -f balancecador.yaml
```

Vemos su configuración:

```
kubectl describe services ejemplo-balanceador
```

```
Name:           ejemplo-balanceador
Namespace:      default
Labels:         <none>
Annotations:        <none>
Selector:       app=httpd
Type:           LoadBalancer
IP:         10.96.97.216
Port:           <unset>    8080/TCP
NodePort:       <unset>    32249/TCP
Endpoints:      10.32.0.22:80,10.32.0.23:80
Session Affinity:   None
Events:         <none>
```

Si accedemos a http://10.96.97.216:8080 nos balancea entre los dos PODS que tenemos desplegados de apache

CLOUD

Sitios para instalar en cloud:
https://kubernetes.io/docs/setup/pick-right-solution/#hosted-solutions

Seguridad

https://kubernetes.io/docs/admin/authentication/

Balanceo de un nodo físico

https://kubernetes.io/docs/tasks/administer-cluster/cluster-management/

Si queremos parar un nodo por mantenimiento:

```
kubectl --server=192.168.2.1:8001 drain kubernetes2

node "kubernetes2" cordoned
error: DaemonSet-managed pods (use --ignore-daemonsets to ignore): kube-
proxy-648gj, weave-net-3z537
```

Parece que no funciona. Lo lanzamos con el comando:

```
kubectl  --ignore-daemonsets --server=192.168.2.1:8001 drain kubernetes2
node "kubernetes2" already cordoned
WARNING: Ignoring DaemonSet-managed pods: kube-proxy-648gj, weave-net-3z537

pod "servidorweb-3127718498-tpts3" evicted
pod "servidorweb-3127718498-3trvp" evicted
pod "heapster-57121549-zx3dt" evicted
pod "kubernetes-dashboard-2039414953-hsvkn" evicted
pod "apache-4284376775-7z96m" evicted
pod "kube-dns-692378583-x9kzv" evicted
node "kubernetes2" drained
```

Para volver a dejarlo funcionando:

```
kubectl   --server=192.168.2.1:8001 uncordon kubernetes2
```

Flecos

TODO: revisar si lo que hay aquí anotado se conserva o se suprime

Mesos fleet tutum grafana

Monitorización: http://www.sysdig.org/

System metrics: Datadog
Container logs: Papertrail

Ejemplo

En este ejemplo:

- Configuramos kubernetes en alta disponibilidad
- Exponemos un servicio a través de un certificado SSL válido
- El pod que recibe la petición muestra la IP pública del cliente

Máquinas:

DNS	IP privada	IP pública	Comentario
k8s.local	192.168.95.69	-	Cluster 'kube-apiserver'
k8s1.local	192.168.95.71	8.8.8.8	Edge router. La IP pública es figurada
k8s2.local	192.168.95.71	-	Primer control plane
k8s3.local	192.168.95.72	-	Control plane adicional
k8s3.local	192.168.95.73	-	Control plane adicional
k8s4.local	192.168.95.74	-	Worker
k8s5.local	192.168.95.75	-	Worker
k8s6.local	192.168.95.76	-	Worker

1. Crear el cluster

1.1. Instalar docker en todos los nodos, los tres control plane (k8s2.local, k8s3.local y k8s4.local) y los tres workers (k8s5.local, k8s6.local y k8s7.local).

1.2. Instalar paquetes de kubernetes en todos los nodos, los tres control plane (k8s2.local, k8s3.local y k8s4.local) y los tres workers (k8s5.local, k8s6.local y k8s7.local).

1.3. Crear un balanceador de kube-apiserver y unir los tres control plane al mismo.

1.4. Iniciar el cluster en el primer control plane

1.5. Unir al cluster el resto de control plane (k8s3.local y k8s4.local).

1.6. Unir al cluster los tres workers.

En este momento ya tenemos el cluster de kubernetes montado.

2. Exponer un servicio con certificado SSL válido y que muestre la IP pública del cliente

2.1. Instalar Nginx ingress con nodeport, afinitty y proxy protocol.

Como dice el paso 8, anotar los puertos HTTP y HTTPS que exponen cada uno de los nodos.

2.2. Instalar Cert-manager addon

2.3. Crear issuer para letsencrypt en producción

2.4. Crear un deployment que despliegue el contenedor “whoami”

2.5. Exponer el anterior deployment a través de un service.

2.6. Crear un ingress con letsencrypt para que un nombre DNS público, “example.com” llegue al servicio definido en el paso anterior.

En este punto si:

- Se deshabilita momentáneamente proxy protoco (revirtiendo este paso
- Se obtiene el worker donde está corriendo el pod 'whoami'
- Se obtiene la IP privada de ese worker
- Se obtiene el puerto en el que escucha el servicio HTTP Nodeport (ver paso 2.1. de estas instrucciones)
- Se le pasa la cabecera “Host” con el mombre adecuado (en este ejemplo “example.com”9

Se podría llegar al pod. Ejemplo:

```
_HOST=192.168.95.75
_PORT=32079
curl -L -s -i -H "Host: example.com" http://$_HOST:$_PORT
```

3. Configurar edge router

En este ejemplo vamos a instalar dos servidores, que van a compartir una IP flotante para tener redundancia.

3.1. Seguir estas instrucciones.

3.2. Crear un nombre DNS, “example.com”, que apunte a la IP pública flotante, por ejemplo '8.8.8.8'

4. Probar

https://example.com

Resultado esperado:

- Debería mostrar un certificado SSL válido
- Debería mostrar la IP pública del equipo que realizó la petición

Bash completion

https://kubernetes.io/es/docs/tasks/tools/included/optional-kubectl-configs-bash-linux/

```
source /usr/share/bash-completion/bash_completion
echo 'source <(kubectl completion bash)' >>~/.bashrc
```

```
sudo su
kubectl completion bash >/etc/bash_completion.d/kubectl
exit
```

```
exit
```

Desplegar pods en control plane

No es lo recomendable, pero si tenemos un cluster con 3 nodos (stacked etc), los 3 control plane, y sin workers, podemos desplegar pods en los control plane si les quitamos los taints que precisamente marcan que NO se puedan desplegar pods allí.

Para cada uno de los nodos del cluster, en este caso “k8s1”:

```
kubectl taint nodes k8s1 node-role.kubernetes.io/control-plane:NoSchedule-
kubectl taint nodes k8s1 node-role.kubernetes.io/master:NoSchedule-
```

Volúmenes

TODO

Persistentes

kadalu (usar este)

Fuente

1. Instalar cliente glusterfs.

TODO: igual solo con el cliente mejor, pero así funciona

```
sudo apt install -y glusterfs-server
```

2. Instalar kadalu

```
curl -fsSL
https://github.com/kadalu/kadalu/releases/latest/download/install.sh | sudo
bash -x
```

Comprobar:

```
kubectl kadalu version
```

Salida esperada similar a:

```
kubectl-kadalu plugin: 0.8.15
```

Listar pods:

```
kubectl get pods -n kadalu
```

Salida esperada similar a:

```
NAME                           READY   STATUS    RESTARTS   AGE
kadalu-csi-nodeplugin-h8nm8    3/3     Running   0          74s
kadalu-csi-nodeplugin-kfp5c    3/3     Running   0          74s
kadalu-csi-nodeplugin-sfgn9    3/3     Running   0          74s
kadalu-csi-provisioner-0       5/5     Running   0          63s
operator-57b47b555f-h6gdm      1/1     Running   0          75s
```

3. (En cada uno de los nodos de kubernetes) Crear directorio donde se almacenará el cluster de glusterfs

sudo mkdir -p /opt/kadalu/brick1/gv1

4. Crear el volumen del cluster de glusterfs

```
kubectl kadalu storage-add \
  storage-pool1 \
  --verbose \
  --type=Disperse \
  --data 2 \
  --redundancy 1 \
  --path k8s1:/opt/kadalu/brick1/gv1 \
  --path k8s2:/opt/kadalu/brick1/gv1 \
  --path k8s3:/opt/kadalu/brick1/gv1
```

Salida esperada similar a:

```
The following nodes are available:
  k8s1, k8s2, k8s3

Storage Yaml file for your reference:

apiVersion: "kadalu-operator.storage/v1alpha1"
```

```
kind: "KadaluStorage"
metadata:
  name: "storage-pool1"
spec:
  type: "Disperse"
  storage:
    - node: "k8s1"
      path: "/opt/kadalu/brick1/gv1"
    - node: "k8s2"
      path: "/opt/kadalu/brick1/gv1"
    - node: "k8s3"
      path: "/opt/kadalu/brick1/gv1"
  disperse:
    data: 2
    redundancy: 1

Is this correct?(Yes/No):
```

Teclear “Yes” y pulsar “Enter”

Salida esperada similar a:

```
Storage add request sent successfully
kadalustorage.kadalu-operator.storage/storage-pool1 created
```

Listar pods:

```
kubectl get pods -n kadalu -o wide
```

Salida esperada similar a:

```
NAME                          READY   STATUS    RESTARTS   AGE   IP
NODE   NOMINATED NODE   READINESS GATES
kadalu-csi-nodeplugin-h8nm8   3/3     Running   0          15m   10.244.2.13
k8s2   <none>           <none>
kadalu-csi-nodeplugin-kfp5c   3/3     Running   0          15m   10.244.0.26
k8s1   <none>           <none>
kadalu-csi-nodeplugin-sfgn9   3/3     Running   0          15m   10.244.1.6
k8s3   <none>           <none>
kadalu-csi-provisioner-0      5/5     Running   0          15m   10.244.0.25
k8s1   <none>           <none>
operator-57b47b555f-h6gdm     1/1     Running   0          15m   10.244.0.24
k8s1   <none>           <none>
server-storage-pool1-0-0      1/1     Running   0          83s   10.244.0.27
k8s1   <none>           <none>
server-storage-pool1-1-0      1/1     Running   0          82s   10.244.2.14
k8s2   <none>           <none>
server-storage-pool1-2-0      1/1     Running   0          82s   10.244.1.7
k8s3   <none>           <none>
```

5. Repetir para cada pod:

5.1. Crear PVC (Persistent Volume Claim):

IMPORTANTE: la documentación dice que “storageClassName:” debe ser “kadalu.disperse”, pero no, tiene que ser el nombre del volumen de glusterfs creado anteriormente.

```
cat <<EOF | kubectl apply -f -
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: persistent-volume-claim-1
spec:
  storageClassName: kadalu.storage-pool1
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
EOF
```

Un nuevo volumen persistente se ha creado:

```
kubectl get persistentvolume
```

Salida esperada similar a:

```
NAME                                       CAPACITY   ACCESS MODES   RECLAIM
POLICY   STATUS   CLAIM                              STORAGECLASS
REASON   AGE
pvc-5df9ed7d-e77d-4910-b73b-a6145efb9c96   1Gi        RWX            Delete
Bound    default/persistent-volume-claim-1   kadalu.storage-pool1
52s
```

Una nueva reclamación de volumen persistente se ha creado:

```
kubectl get persistentvolumeclaims
```

Salida esperada similar a:

```
NAME                        STATUS   VOLUME
CAPACITY   ACCE
SS MODES   STORAGECLASS           AGE
persistent-volume-claim-1   Bound    pvc-5df9ed7d-e77d-4910-b73b-
a6145efb9c96   1Gi        RWX
           kadalu.storage-pool1   22s
```

5.2. Crear el pod referenciando la reclamación de volumen persistente:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
```

```
metadata:
  name: nginx-test
spec:
  containers:
    - name: nginx-test
      image: nginx
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: nginx-test-volume
  volumes:
    - name: nginx-test-volume
      persistentVolumeClaim:
        claimName: persistent-volume-claim-1
EOF
```

Usando glusterfs externo

Requisitos:

- Un Cluster glusterfs funcionando. En este ejemplo los 3 nodos serán a la vez:
  - Control Plane
  - Workers
  - Nodos del cluster glusterfs

Fuente

1. Crear endpoints

- “ip” es la IP de cada nodo. En nuestro caso es el mismo que los nodos de kubernetes
- “port” un número arbitrario, “1” es suficiente

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Endpoints
metadata:
  name: glusterfs-cluster
subsets:
- addresses:
  - ip: 10.0.0.2
  ports:
  - port: 1
- addresses:
  - ip: 10.0.0.3
  ports:
  - port: 1
- addresses:
  - ip: 10.0.0.4
  ports:
  - port: 1
EOF
```

Comprobar:

```
kubectl get endpoints
```

Resultado esperado similar a:

```
NAME                ENDPOINTS                                   AGE
glusterfs-cluster   10.0.0.2:1,10.0.0.3:1,10.0.0.4:1   22s
```

2. Crear servicio

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: glusterfs-cluster
spec:
  ports:
  - port: 1
EOF
```

Comprobar:

```
kubectl get services
```

Resultado esperado similar a:

```
NAME                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
glusterfs-cluster   ClusterIP   10.105.75.133    <none>        1/TCP     51s
```

3. Crear pod

- Campo “path” debe ser el nombre de un volumen de glusterfs **previamente** creado

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: glusterfs
spec:
  containers:
  - name: glusterfs
    image: nginx
    volumeMounts:
    - mountPath: "/usr/share/nginx/html"
      name: glusterfsvol
  volumes:
  - name: glusterfsvol
    glusterfs:
      endpoints: glusterfs-cluster
      path: gv1
```

```
      readOnly: false
EOF
```

Comprobar (puede tardar 1 minuto):

```
kubectl get pods
```

Resultado esperado similar a:

```
NAME                                READY   STATUS    RESTARTS   AGE
glusterfs                           1/1     Running   0          13s
```

Resumen:

- El pod “glusterfs” tiene montado localmente un directorio, “/usr/share/nginx/html”
- En este directorio puede escribir
- Lo que escriba se escribirá en el volumen “gv1” del cluster glusterfs, que a su vez se escribirá en el brick “/opt/brick1/gv1” de cada uno de los nodos del cluster de glusterfs
- Se pueden montar más pods, o este mismo si se tiene que recrear (porque se arrancla con “replica 1” o similar), que escriban en ese volumen de glusterfs

TODO:

- Nótese que la provisión de volúmenes de glusterfs NO es dinámica. Primero se crea manualmente el volumen de glusterfs y luego se referencia, manualmente, en el pod de kubernetes
- Parece ser que para que se pueda provisionar dinámicamente parece ser que se requiere Heketi, que es un proyecto que está de capa caída

Cluster externo Glusterfs

Vamos a instalar un cluster de Glusterfs “externo”, o como servicio, en lugar de montarlo como daemonset, que parece que requiere de un servicio (heketi) que no parece tener mucho futuro.

1. (Cada nodo) Editar:

```
sudo vim /etc/hosts
```

Y añadir:

```
10.0.0.2    k8s1
10.0.0.3    k8s2
10.0.0.4    k8s3
```

2. (Cada nodo) Instalar:

```
sudo apt install -y glusterfs-server
```

```
sudo systemctl enable glusterd --now
```

```
sudo service glusterd status
```

```
q
```

3. (Desde un nodo, en este caso k8s1) Ejecutar:

```
sudo gluster peer probe k8s2
sudo gluster peer probe k8s3
```

4. (Desde otro nodo, en este caso k8s3) Ejecutar:

```
sudo gluster peer probe k8s1
```

5. (Desde cualquier nodo) Ejecutar:

```
sudo gluster peer status
```

6. Crear brick. Estas instrucciones se pueden ejecutar desde cualquier nodo del cluster

6.1. Ejecutar:

```
sudo mkdir -p /opt/brick1/gv1
```

WARNING: vamos a usar “force” porque usaremos la misma partición de sistema

Vamos a usar el modo disperse con redundancia 1, que significa que podemos perder un máximo de un nodo y todo seguirá funcionando.

```
sudo gluster volume create gv1 disperse 3 redundancy 1
k8s{1..3}:/opt/brick1/gv1 force
```

```
sudo gluster volume start gv1
```

```
sudo gluster volume info
```

From:
http://wiki.legido.com/ - **Legido Wiki**

Permanent link:
http://wiki.legido.com/doku.php?id=informatica:linux:kubernetes

Last update: **2023/09/25 07:05**