

# Trabajo Administración de Sistema - Kubernetes

Hayk Kocharyan  
757715@unizar.es

14 de mayo de 2020

## Índice

<b>1. Resumen</b>	<b>1</b>
<b>2. Introducción</b>	<b>1</b>
<b>3. Arquitectura y despliegue de Kubernetes</b>	<b>1</b>
3.1. Pods . . . . .	1
3.2. Nodos . . . . .	2
3.3. Labels y Logging . . . . .	2
3.4. API Server . . . . .	3
3.5. Deployment . . . . .	3
3.6. Services . . . . .	4
3.7. Namespaces . . . . .	6
3.8. Volumenes, PVC, PV y StorageClass . . . . .	6
<b>4. Vagrant</b>	<b>7</b>
<b>5. Provision</b>	<b>8</b>
<b>6. Problemas</b>	<b>9</b>
<b>Referencias</b>	<b>11</b>

## 1. Resumen

En esta primera parte del proyecto de Kubernetes se ha llevado una toma de contacto con este orquestador de contenedores. Se han realizado despliegues de servicios básicos con el principal objetivo de conocer las herramientas básicas de Kubernetes, al igual que comprender la arquitectura que este esconde por debajo y de la que los desarrolladores se olvidan pero que un administrador no puede obviar.

## 2. Introducción

Para llevar a cabo esta primera parte de la tarea se ha realizado una previa documentación de Kubernetes desde diferentes fuentes. Una de estas ha sido las diapositivas de clase proporcionadas por el profesor, otra fuente de información ha sido una serie de tutoriales que explican los conceptos generales y un uso básico de estos [2], y también se han visualizado los dos primeros videos de uso mas detallado de este orquestador [1]. Para completar los conocimientos y la toma de contacto se ha seguido el guion proporcionado por el profesor [5]. Las primeras pruebas realizadas se llevan a cabo con la herramienta *Minikube*, un administrador de MV donde corre un *cluster* de Kuberentes pero de un único nodo. Para el seguimiento del guion [5] se ha hecho uso de Vagrant para desplegar 3 MV y desplegar en estas un *cluster* de Kubernetes con 2 *nodos* y un *maestro*<sup>1</sup>. A lo largo del seguimiento de este guion se lanzan diferentes tipos de *pods* en los nodos, estos pods contienen diferentes tipos de contenedores. Se trabaja con la replicación y disponibilidad de servicios que nos ofrece este orquestrador.

## 3. Arquitectura y despliegue de Kubernetes

### 3.1. Pods

Para comenzar vamos a explicar conceptos de Kubernetes (**K8s a partir de ahora**) que son imprescindibles de conocer.

En primer lugar, explicaremos la unidad más pequeña de K8s, que es el **Pod**. Un *pod* corre contenedores, puede correr desde un único contenedor a varios contenedores. Cada *pod* tiene una IP asignada dentro de esa subred del nodo, sin embargo, esta IP no es fija, ya que si el *pod* cae, cuando vuelva a estar operativo se le habrá asignando una nueva IP. Los contenedores que hay dentro de un *pod*, pueden contactar unos con otros a través de **localhost**.

Los pods se crean a partir de un fichero de configuración con extensión **.yaml**, estos ficheros contienen todo lo necesario para poner en marcha el pod dentro de un nodo con sus contenedores. Podemos tener ficheros de configuración muy básicos en el que simplemente le indicamos que solo queremos un contenedor con un *CentOS*, hasta ficheros en los que indicamos el número de replicas, los volúmenes y PVCs a usar, diferentes contenedores y muchos otros detalles.

Los comandos mas comunes para poner en marcha un *pod* son:

```
kubectl apply -f pod_file.yaml  
kubectl apply -f https://dirección.del.pod/
```

Para trabajar con los pods existen unos comando que se deben conocer:

```
kubectl get pods  
# si queremos más información  
kubectl get pods -o wide  
# tambien se puede la abreviación  
kubectl get po  
# para obtener monitorización de los pods usaremos  
kubectl top pods  
# podemos obtener mas información sobre un pod con:  
kubectl describe pod nombrePod  
# este comando nos da mucha información, desde el nodo en el que  
# está corriendo y su ip interna, la IP del pod, sus contenedores y
```

<sup>1</sup>En K8s se evita hacer uso de workers y masters, en su lugar se usan los términos maestro y nodos (todos los nodos que no actúan como maestro.)

```

# muchos otros datos .
# adicionalmente podemos obtener información del pod
# en forma de fichero yaml con:
kubectl get pod nombrePod -o yaml

```

Como datos adicionales al crear un contenedor en un *pod*, podemos asignarle datos de CPU y de memoria. Para la CPU se lleva una unidad métrica particular de K8s, en concreto **un cpu de k8s** es el equivalente a 1 vCPU/Core en la nube, o a 1 *HyperThread* ® en físico. Por último, resaltar que si al instanciar un *pod* añadimos la opción **-record**, al consultar los detalles de un *pod* con la opción *describe*, podremos observar en el mapa de *annotations* una serie de registro de eventos que han ocurrido en el *pod*.

### 3.2. Nodos

También vamos a entender que ocurre en un nodo. Un nodo es una máquina, física o virtual, donde correrán en su interior uno o varios *pods*. Un concepto muy importante es el de **Kubelet** que es el proceso que corre en cada uno de los nodos del cluster y se encarga de comunicarse con la API del maestro para registrar al nodo y para otras tareas como avisar de la caída de un *pod*. K8s comprueba que el nodo está vivo y es correcto (corren en él los servicios necesario), en ese caso este nodo se considera elegible para correr *pods* en su interior. Cada nodo dispone de dos IPs, uno interno para comunicarse con los pods y otro externo, de cara al resto de nodos. Como podemos ver en la figura 1 existe un servicio *Kube Proxy* en el nodo que es la encargada de tramitar las conexiones entrantes y salientes de este.

Con los nodos, como con los *pods*, podemos ejecutar los comandos que hemos mencionado anteriormente pero sustituyendo el término *pod* por *node* o *no*.

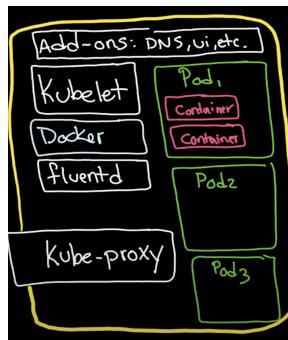


Figura 1: Posibles componentes en un nodo con varios *pods* [2]

### 3.3. Labels y Logging

Para facilitar trabajo a la hora de administrar los nodos y *pods*, podemos hacer uso de los *labels*. Este herramienta es muy útil para agrupar *pods*. De esta manera al crear un *pod* podemos añadirle al comando la opción **-l** con la que podemos darle una *label*. Lo más habitual es hacer uso de las *labels* con la jerarquía de clave-valor, por ejemplo, la *label env=producción*, o *env=desarrollo*, de esta manera sepáramos los *pods* para que se ejecuten en producción o en desarrollo, y así luego poder realizar consultas y motorización sobre un grupo de etiquetas. Esta etiqueta también se puede añadir en el fichero yaml, lo que esto nos limita la flexibilidad debido a que esa configuración solo será valida para esa etiqueta y si queremos darle otra, deberemos modificar el contenido. Sin embargo, si la etiqueta se la asignamos cuando vayamos a crear el *pod*, podemos usar el mismo fichero **yaml** para diferentes pods. Con los *labels* podemos conseguir ejecutar comando como el siguiente:

```

kubectl get pods -l 'env in (production)'
# obtenemos los pods de nuestro namespace default que estén con
# el label env=production

```

Por otro lado tenemos el comando *Kubectl logs*, que podemos aplicar al contenedor de un *pod* para que nos de información.

El motor que esté ejecutando el contenedor (comunmente docker) se encarga de guardar toda la salida para *stderr* y *stdout* en un fichero y asociarlo al contenedor, de esta forma es siempre accesible. Desde k8s nos recomiendan gestionar un *logrotate*, ya que es posible que tengamos una gran cantidad de salida en forma de *logs* y esto ocupe demasiada memoria.

Algo muy destacable es que podemos obtener *feedback* de componentes del sistema como es el *Kube-DNS*, para ello debemos ejecutar:

```
kubectl logs -n=kube-system nombre_pod_dns nombre_contendor_dns
```

Cabe destacar que *kube-proxy* y *scheduler* corren en contendores, pero por ejemplo, *kubelet* y el gestor de contendores no lo hacen, estos son procesos del nodo.

Una implementación de la arquitectura que podemos citar, es la de disponer de un nodo que se encargue de realizar los *logrotate* y mantener todo el *log* en un volumen.

### 3.4. API Server

A través de una proxy a la API Server podemos obtener información del *cluster*, esto se realiza primeramente abriendo un puerto *HTTP* con el siguiente comando:

```
kubectl proxy --port=8080
# con este comando creamos un servidor proxy o una puerta de
# enlace nivel aplicación entre la API Kubernetes y localhost
```

Ejecutando el comando de arriba podemos hacer un curl a la dirección *http://localhost:8080/api/* y obtener información de todo el *cluster*, o también podemos acceder a los pods *http://localhost:8080/api/v1/namespaces/default/pods*.

### 3.5. Deployment

Para entender lo que es un *deployment* vamos a poner un diagrama 2 que explica el despliegue de los diferentes elementos de K8s.

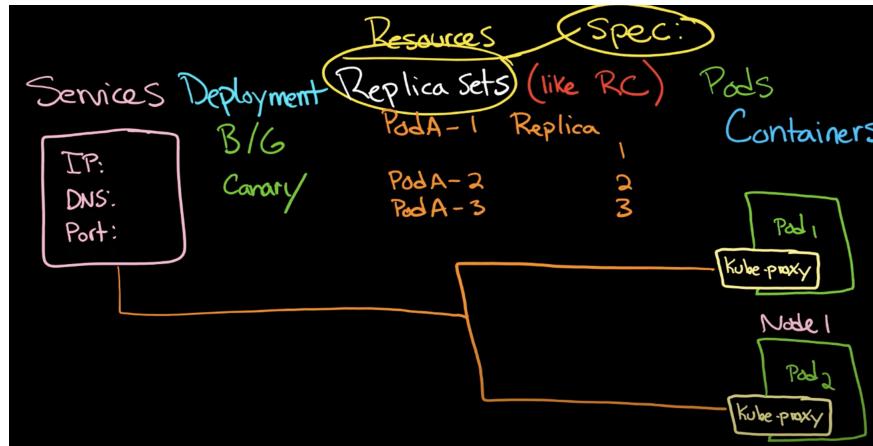


Figura 2: Orden de despliegue de los elementos de K8s [2]

En la figura 2 podemos observar que el elemento más grande son los *services*, que engloban todo. Lo siguiente más pequeño que tenemos son los *deployment*, estos contienen un *replica set* en el que especificamos el número de replicas que queremos de un *pod* específico.

Ahora podemos explicar lo que es un *deployment*. Este es una especie de supervisor para los *pods*. Con los *deployments* obtenemos control sobre los *pods*, podemos realizar *rollbacks*, al igual que *rolled out*. Además, cuando creamos un *deployment*, se crea también un *replica controller*, que es un elemento más de k8s que se encarga de mantener el número de réplicas especificado en el fichero **yml** siempre constante, sin tener pérdidas, y cuando un *pod*, este se encarga de levantar otro idéntico. Vamos a ver un ejemplo:

En esta imagen podemos observar en las tres terminales de abajo los siguientes datos:

```

hay@MacBook-Pro-de-Hayk: ~/Desktop/admin2/kubernetes/Proyecto2Parte1/vagrantk3s
$ k get all
NAME           TYPE     CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   <none>        443/TCP    19h
+ vagrantk3s   KubeConfig   ClusterIP   <none>        443/TCP    19h
service/kubernetes   ClusterIP   <none>        443/TCP    19h
+ vagrantk3s   kubectl apply -f https://raw.githubusercontent.com/openshift-evangelists/kbe/master/specs/deployments/d10.yaml
deployment.apps/sise-deploy created
+ vagrantk3s   kubectl apply -f up.yaml
deployment.apps/sise-deploy configured
+ vagrantk3s   k get all
NAME           READY   STATUS    RESTARTS   AGE
pod/sise-deploy-f696887f-8qjtm   1/1    Running   0          47s
pod/sise-deploy-f696887f-qszmx   1/1    Running   0          46s
NAME           TYPE     CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   <none>        443/TCP    19h
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/sise-deploy   2/2    2           2           2m2s
NAME           DESIRED  CURRENT   READY   AGE
replicaset.apps/sise-deploy-f696887f   2      2      2      47s
replicaset.apps/sise-deploy-68d9c95c7b   0      0      0      2m2s
+ vagrantk3s   k get po -w
NAME           READY   STATUS    RESTARTS   AGE
sise-deploy-f696887f-8qjtm   1/1    Running   0          26s
sise-deploy-f696887f-8qjtm   1/1    Running   0          26s
sise-deploy-f696887f-8qjtm   0/1    Pending   0          0s
sise-deploy-f696887f-8qjtm   0/1    Pending   0          0s
sise-deploy-f696887f-8qjtm   0/1    ContainerCreating   0          0s
sise-deploy-f696887f-8qjtm   1/1    Running   0          1s
sise-deploy-f696887f-8qjtm   1/1    Terminating   0          76s
sise-deploy-f696887f-8qjtm   1/1    Terminating   0          76s
sise-deploy-f696887f-qszmx   0/1    Pending   0          0s
sise-deploy-f696887f-qszmx   0/1    Pending   0          0s
sise-deploy-f696887f-qszmx   0/1    ContainerCreating   0          0s
sise-deploy-f696887f-qszmx   1/1    Running   0          0s
sise-deploy-f696887f-qszmx   1/1    Terminating   0          76s
sise-deploy-f696887f-qszmx   1/1    Terminating   0          76s
sise-deploy-f696887f-qszmx   0/1    Terminating   0          107s
sise-deploy-f696887f-qszmx   0/1    Terminating   0          107s
sise-deploy-f696887f-qszmx   0/1    Terminating   0          108s
sise-deploy-f696887f-qszmx   0/1    Terminating   0          108s
sise-deploy-f696887f-qszmx   0/1    Terminating   0          2m
sise-deploy-f696887f-qszmx   0/1    Terminating   0          2m
+ k get rs -w
NAME           DESIRED  CURRENT   READY   AGE
sise-deploy-68d9c95c7b   2      2      2      18s
sise-deploy-f696887f   1      0      0      0s
sise-deploy-f696887f   1      0      0      0s
sise-deploy-f696887f   1      1      0      0s
sise-deploy-f696887f   1      1      1      1s
sise-deploy-f696887f   1      1      1      1s
sise-deploy-f696887f   1      2      2      76s
sise-deploy-f696887f   2      1      1      1s
sise-deploy-f696887f   1      2      2      76s
sise-deploy-f696887f   1      2      2      76s
sise-deploy-f696887f   1      1      1      76s
sise-deploy-f696887f   2      1      1      1s
sise-deploy-f696887f   2      2      1      1s
sise-deploy-f696887f   2      2      2      1s
sise-deploy-f696887f   0      1      1      76s
sise-deploy-f696887f   0      1      1      76s
sise-deploy-f696887f   0      0      0      76s
+ k get deployments.apps -w
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
sise-deploy   2/2    2           2           12s
sise-deploy   2/2    2           2           75s
sise-deploy   2/2    2           2           75s
sise-deploy   2/2    0           2           75s
sise-deploy   2/2    1           2           75s
sise-deploy   3/2    1           3           26s
sise-deploy   2/2    1           2           76s
sise-deploy   2/2    2           2           76s
sise-deploy   3/2    2           3           76s
sise-deploy   2/2    2           2           76s

```

Figura 3: Simulación de caída y recuperación de pods

- **Terminal izquierda:** los pods y sus estados en tiempo real, vemos como los dos primeros pods en ejecutarse empiezan por **68d**
- **Terminal central:** los *replica set* que existen en tiempo real, el primero en ejecutarse es el **68d...**
- **Terminal derecho:** el *deployment* actual con su estado con respecto a los pods.

En la **terminal superior** podemos observar que en primer lugar se aplica un *k get all*<sup>2</sup>, para mostrar que no hay nada corriendo, a continuación se aplica un **d10.yaml** para lanzar un deployment, seguido de esto se hace una **up.yaml** que actualiza la versión del deployment anterior. Tras esto se realiza un *k get all* y se ve que los *pods* ejecutándose son empiezan por **f69** que no coincide con los dos *pods* primeros que teníamos corriendo. Esto es debido a que tras hacer el *update* de la versión del *deployment* se han tirado los *pods* que se estaban ejecutando y se han levantado dos nuevos que además pertenecen a otro grupo de replicas. En la terminal de la derecha podemos ver como en todo momento siempre hay al menos 2 pods ejecutándose, que son los que hemos exigido que existan, por lo que primero se pone en marcha uno nuevo, luego se tira el antiguo, se pone otro nuevo y se tira el antiguo restante.

### 3.6. Services

Como hemos explicado en la sección anterior, los servicios engloban toda la arquitectura de k8s. Cada servicio tiene una IP virtual, llamada VIP. Esta VIP sirve para redireccionar las conexiones a los *pods* que se encuentran en los nodos, todo esto a través de *kube-proxy*, un servicio que corre en todos los nodos.

Como hemos mencionado en puntos anteriores, la ip de un *pod* no es fija, es decir, puede variar si sufre una caída, por lo que si un usuario quiere acceder a un servicio no podemos proporcionarle la IP del *pod*, sino que hay que proporcionarle la del servicio.

<sup>2</sup>k es el alias de kubectl

```
vagrant@w1: ~
-A KUBE-SEP-3KSNWAQQUVVDJK3LB -p tcp -m tcp -j DNAT --to-destination 192.368.0.90:6443
-A KUBE-SEP-BPXAGC0DH4GUSROY -s 10.42.2.28/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-BPXAGC0DH4GUSROY -p tcp -m tcp -j DNAT --to-destination 10.42.2.28:9876
-A KUBE-SEP-HK04X4C0RQ1CVGVTB -s 10.42.1.4/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-HK04X4C0RQ1CVGVTB -p tcp -m tcp -j DNAT --to-destination 10.42.1.4:53
-A KUBE-SEP-LUUL1YJHCOVETXHE -s 10.42.1.4/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-LUUL1YJHCOVETXHE -p udp -m udp -j DNAT --to-destination 10.42.1.4:53
-A KUBE-SEP-NIXN0U7GR2H2SP4 -s 10.42.1.4/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-NIXN0U7GR2H2SP4 -p tcp -m tcp -j DNAT --to-destination 10.42.1.4:9153
-A KUBE-SEP-T6217T3YK1ZS0QX7 -s 10.42.1.3/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-T6217T3YK1ZS0QX7 -p tcp -m tcp -j DNAT --to-destination 10.42.1.3:443
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.1/32 -p tcp -m comment --comment "default/kubernetes:https cluster IP" -m tcp --dport 443 -j KUBE-MARK-MASQ
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.10/32 -p udp -m comment --comment "kube-system/kube-dns:cluster IP" -m udp --dport 53 -j KUBE-MARK-MASQ
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.10/32 -p udp -m comment --comment "kube-system/kube-dns:cluster IP" -m udp --dport 53 -j KUBE-SVC-TC0U7JCQXEZGVNU
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.10/32 -p tcp -m comment --comment "kube-system/kube-dns:tcp cluster IP" -m tcp --dport 53 -j KUBE-MARK-MASQ
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.10/32 -p tcp -m comment --comment "kube-system/kube-dns:tcp cluster IP" -m tcp --dport 53 -j KUBE-SVC-ERIFXISQEP7F07F4
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.10/32 -p tcp -m comment --comment "kube-system/kube-dns:metrics cluster IP" -m tcp --dport 9153 -j KUBE-MARK-MASQ
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.157.144/32 -p tcp -m comment --comment "kube-system/kube-dns:metrics cluster IP" -m tcp --dport 9153 -j KUBE-SVC-JD5M3NA414D0Y0P
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.157.144/32 -p udp -m comment --comment "kube-system/metrics-server: cluster IP" -m udp --dport 80 -j KUBE-MARK-MASQ
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.157.144/32 -p udp -m comment --comment "kube-system/metrics-server: cluster IP" -m udp --dport 80 -j KUBE-SVC-LCSQY6GVUZHJ6WZ
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.157.144/32 -p tcp -m comment --comment "default/simpleservice: cluster IP" -m tcp --dport 80 -j KUBE-MARK-MASQ
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.157.144/32 -p tcp -m comment --comment "default/simpleservice: cluster IP" -m tcp --dport 80 -j KUBE-SVC-EZC6WLOVQADP4IAW
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.157.144/32 -p tcp -m comment --comment "kubernetes service nodeports; NOTE: this must be the last rule in this chain" -m addrtary --dst-type LOCAL -j KUBE-NODEPORTS
-A KUBE-SVC-ERIFXISQEP7F07F4 -j KUBE-SEP-BPXAGC0DH4GUSROY
-A KUBE-SVC-EZC6WLOVQADP4IAW -j KUBE-SEP-BPXAGC0DH4GUSROY
-A KUBE-SVC-JD5M3NA414D0Y0P -j KUBE-SEP-NIXN0U7GR2H2SP4
-A KUBE-SVC-LCSQY6GVUZHJ6WZ -j KUBE-SEP-T6217T3YK1ZS0QX7
-A KUBE-SVC-NPXA40MPMTKRNQY -j KUBE-SEP-3KSNWAQQUVVDJK3LB
-A KUBE-SVC-TC0U7JCQXEZGVNU -j KUBE-SEP-LUUL1YJHCOVETXHE
COMMIT
# Completed on Tue May 12 20:06:59 2020
vagrant@w1:~$ sudo iptables-save | egrep "simpleservice|KUBE-SVC-EZC6WLOVQADP4IAW"
> <
vagrant@w1:~$
```

Figura 4: Iptables - redirección a endpoints

Podemos ver como en la linea seleccionada del centro, la ip **10.43.220.104** perteneciente al servicio, redirige las peticiones tcp por el puerto 80 a **KUBE-SVC-EZ...**, esta a **KUBE-SEP-BPX...** que curiosamente es la IP del *Pod*.

```
vagrant@m: ~
-A KUBE-SEP-3KSNWAQQUVVDJK3LB -p tcp -m tcp -j DNAT --to-destination 192.368.0.90:6443
-A KUBE-SEP-BPXAGC0DH4GUSROY -s 10.42.2.28/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-BPXAGC0DH4GUSROY -p tcp -m tcp -j DNAT --to-destination 10.42.2.28:9876
-A KUBE-SEP-E7NN9WTA3NO2BSM -s 10.42.1.16/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-E7NN9WTA3NO2BSM -p tcp -m tcp -j DNAT --to-destination 10.42.1.16:9876
-A KUBE-SEP-HK04X4C0RQ1CVGVTB -s 10.42.1.4/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-HK04X4C0RQ1CVGVTB -p tcp -m tcp -j DNAT --to-destination 10.42.1.4:53
-A KUBE-SEP-LUUL1YJHCOVETXHE -s 10.42.1.4/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-LUUL1YJHCOVETXHE -p udp -m udp -j DNAT --to-destination 10.42.1.4:53
-A KUBE-SEP-NIXN0U7GR2H2SP4 -s 10.42.1.4/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-NIXN0U7GR2H2SP4 -p tcp -m tcp -j DNAT --to-destination 10.42.1.4:9153
-A KUBE-SEP-T6217T3YK1ZS0QX7 -s 10.42.1.3/32 -j KUBE-MARK-MASQ
-A KUBE-SEP-T6217T3YK1ZS0QX7 -p tcp -m tcp -j DNAT --to-destination 10.42.1.3:443
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.1/32 -p tcp -m comment --comment "default/simpleservice: cluster IP" -m tcp --dport 80 -j KUBE-MARK-MASQ
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.1/32 -p tcp -m comment --comment "default/simpleservice: cluster IP" -m tcp --dport 80 -j KUBE-SVC-EZC6WLOVQADP4IAW
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.10/32 -p udp -m comment --comment "default/kubernetes:https cluster IP" -m udp --dport 443 -j KUBE-SVC-NPXA40MPMTKRNQY
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.10/32 -p udp -m comment --comment "kube-system/kube-dns:cluster IP" -m udp --dport 53 -j KUBE-MARK-MASQ
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.10/32 -p udp -m comment --comment "kube-system/kube-dns:metrics cluster IP" -m udp --dport 53 -j KUBE-SVC-TC0U7JCQXEZGVNU
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.10/32 -p tcp -m comment --comment "kube-system/kube-dns:tcp cluster IP" -m tcp --dport 53 -j KUBE-MARK-MASQ
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.10/32 -p tcp -m comment --comment "kube-system/kube-dns:tcp cluster IP" -m tcp --dport 53 -j KUBE-SVC-ERIFXISQEP7F07F4
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.0.10/32 -p tcp -m comment --comment "kube-system/kube-dns:metrics cluster IP" -m tcp --dport 9153 -j KUBE-MARK-MASQ
-A KUBE-SERVICES ! -s 10.42.0.0/16 -d 10.43.157.144/32 -p tcp -m comment --comment "kubernetes service nodeports; NOTE: this must be the last rule in this chain" -m addrtary --dst-type LOCAL -j KUBE-NODEPORTS
-A KUBE-SVC-ERIFXISQEP7F07F4 -j KUBE-SEP-BPXAGC0DH4GUSROY
-A KUBE-SVC-EZC6WLOVQADP4IAW -j KUBE-SEP-NIXN0U7GR2H2SP4
-A KUBE-SVC-JD5M3NA414D0Y0P -j KUBE-SEP-T6217T3YK1ZS0QX7
-A KUBE-SVC-LCSQY6GVUZHJ6WZ -j KUBE-SEP-TC0U7JCQXEZGVNU
-A KUBE-SVC-NPXA40MPMTKRNQY -j KUBE-SEP-3KSNWAQQUVVDJK3LB
-A KUBE-SVC-TC0U7JCQXEZGVNU -j KUBE-SEP-LUUL1YJHCOVETXHE
COMMIT
# Completed on Tue May 12 20:48:03 2020
```

Figura 5: Iptables - redirección a endpoints con dos pods idénticos

En esta imagen ocurre algo similar, pero el servicio redirige a un **KUBE-SVC-EZ..** que aparece dos veces, esto es debido a que hay dos *pods*, así que hay dos endpoint. Esto, implica que haya que balancear la carga.

### 3.7. Namespaces

Los *namespaces* son un medio para dividir los recursos del *cluster* entre múltiples usuarios. Esto es útil cuando tenemos muchos desarrolladores divididos en diferentes grupos como puede ser desarrolladores, administradores, testers...

Un caso útil de ejemplo es el despliegue de un *pod* en ambiente de desarrollo y de producción, o por ejemplo, lanzar un *pod* específico que realice tests pero el cual no debe verse por usuarios del grupo de desarrollo.

### 3.8. Volumenes, PVC, PV y StorageClass

En primer lugar hay que aclarar que los *pods* no almacenan datos, si fuese así cuando cayese un *pod*, el pod sustituto no tendría los datos por lo que se perdería la consistencia. Es aquí donde entran en juego los Voluemenes, PVC, PV, StorageClass y los almacenamientos físicos. Para entender los volúmenes vamos a introducir un diagrama que se observa en la figura 6 para ver al detalle todos los tipos posibles de volúmenes que existen.

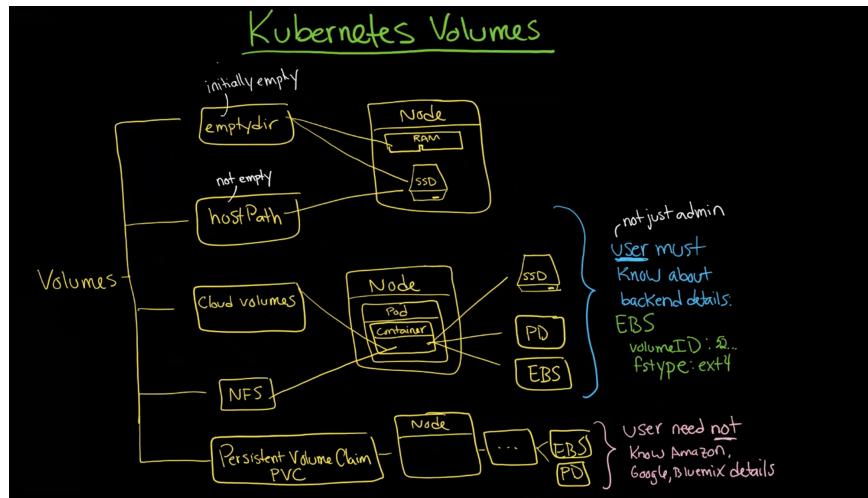


Figura 6: Tipos de volúmenes [2]

- **Host Path:** el nodo puede almacenar los datos en un volumen en ram o ssd que inicialmente estará vacío, este se encuentra en el nodo, por lo que si cae el nodo, se pierde todo.
- **Host Path:** Es similar al caso anterior pero con la excepción de que es un directorio existente y puede no estar vacío.
- **Cloud Volumes:** Especificamos el proveedor de *cloud* que nos ofrece el servicio de almacenamiento. El punto negativo es que el usuario debe conocer muchos detalles de backend, como tipo de sistema de ficheros, nombre de volumen, proveedor... tarea que es más de un administrador.
- **NFS:** Este caso es similar al anterior, es decir, el usuario necesita conocer muchos detalles.
- **PVC:** El nodo se conecta a una serie de abstracciones que son los que le dan el acceso a un almacenamiento en *cloud*, sin embargo, el usuario no debe conocer ningún detalle.

Para entender los PVC, PV y StorageClass vamos a ver el siguiente diagrama (figura 7).

Como podemos ver, cuando definimos un *pod*, especificamos el uso de un volumen que hará referencia a un PVC, que se define de igual manera con un fichero **.yaml**. Un volumen siempre está vinculado a un *PVC*, este es un link a un recurso donde se define al detalle el *Persistent Volume*. Dentro del *PVC*, definiremos tamaño, tipo (rw,ro, wo), el selector que hará referencia al *PV* y si queremos un *StorageClass*. Si hacemos la declaración del *Storage Class*, el *PVC* hará uso de este, sino usará un *PV*. Un *Storage Class* es una manera en la que los administradores de sistema describen las clases o tipos de almacenamiento que ofrecen a los desarrolladores, en cambio un *PV* viene

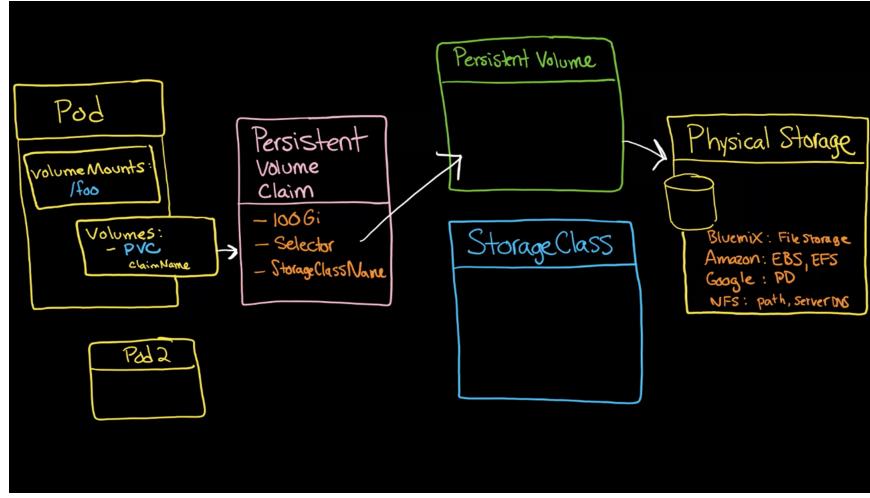


Figura 7: PVC, PV y StorageClass [2]

a ser una partición del almacenamiento del *cluster* que nos provee el administrador. Los PV se conectan a un almacenamiento físico que puede ser de un proveedor de *cloud* o ser un sistema de ficheros NFS.

## 4. Vagrant

En esta sección vamos a analizar el *script* usado con *Vagrant* [4] para lanzar las máquinas virtuales que formarán nuestro *cluster*.

Para empezar, diremos que *Vagrant* es una herramienta para crear entornos de desarrollo reproducibles, permitiendo crear y configurar máquinas virtuales a partir de ficheros de configuración.

Al comenzar, el *script* escrito en *ruby* comienza definiendo dos variables, **U** que corresponde a la imagen de *Ubuntu*, y **D**, que corresponde a la imagen de *Debian*.

A continuación, se define la IP del máster y una lista de tuplas de tipo *nombre nodo, tipo de nodo, ip de nodo, memoria, e ip del master*.

El código comienza con un bucle que realiza la configuración, como argumento se pone un 2 para indicar que usamos configuración de versiones 2.0.x, que es la última disponible a fecha de hoy. En el bucle, se itera la lista de tuplas de nodos y para cada nodo se realiza lo siguiente:

1. Definir la máquina con su nombre

```
config.vm.define node[:hostname] do |nodeconfig|
```

2. Se define el tipo de box que correrá en la máquina. Los *box* son los formatos de paquetes para Vagrant, como es el caso de Ubuntu bionic.

```
#definimos el box como U que es Ubuntu (por la variable previa)
nodeconfig.vm.box = U
#le asignamos el nombre de la tupla correspondiente del nodo
#Vagrant cambiará este nombre en /etc/hosts
nodeconfig.vm.hostname = node[:hostname]
```

3. Configuramos la red

```
# defininimos la red como pública
nodeconfig.vm.network :public_network,
# con bridge definimos la tarjeta de red que nos dará acceso a internet
```

```

bridge: "en0:Wi-Fi(Wireless)",
# asignamos la IP de su tupla
ip: node[:ip],
# definimos Network Interface Card type, en este caso viritio
# específico para VirtualBox.
nic_type: "virtio"

```

- Configuración del proveedor, en nuestro caso VirutalBox

```

nodeconfig.vm.provider "virtualbox" do |v|
  # personalizamos memoria, y cpus
  v.customize ["modifyvm", :id, "--memory", node[:mem], "--cpus", "1"]
  # le asignamos la Network Interface Card anterior
  v.default_nic_type = "virtio"

```

- Especificamos el tiempo por defecto a esperar para que se inicie la máquina

```

nodeconfig.vm.boot_timeout = 400

```

- Especificamos un *provisioner*, un mecanismo que nos permite instalar software cuando la máquina se crea.

```

# decimos que el provisioner será un script shell
nodeconfig.vm.provision "shell",
  # especificamos la ruta
  path: 'provision.sh',
  # escribimos los argumentos
  args: [ node[:hostname], node[:ip], node[:mem], node[:type] ]

```

- Si estamos con el máster, realizaremos algo más.

```

# lanzamos esto después de realizar un vagrant up
nodeconfig.trigger.after :up do |trigger|
  trigger.run = \
    # copiamos k3s.yaml en el fichero de kubeconfig
    {inline: "sh -c 'cp k3s.yaml /Users/hayk/.kube/config'"}

```

## 5. Provision

Provisión es el *script* que se ejecuta **dentro** de cada una de las máquinas virtuales.  
Vamos a explicar el *scirpt* paso a paso.

- Se define el nombre del nodo, y se modifica el fichero de hosts

```

# se copia el nombre del nodo en /etc/hostname
echo $1 > /etc/hostname
hostname $1

# se saca por salida estandar las ips correspondientes a cada máquinas
# y el fichero antiguo de hosts y todo esto se guarda en un nuevo fichero
{ echo 192.168.0.90 m; echo 192.168.0.93 w1; echo 192.168.0.94 w2
  echo 192.168.0.95 w3; cat /etc/hosts
} > /etc/hosts.new
# se borra el fichero nuevo, guardandolo en el original.
mv /etc/hosts{.new,}

```

- Se copia la herramienta k3s [3]

```
cp k3s /usr/local/bin/
```

### 3. Caso del máster

```
# ignoramos la instalación ya que disponemos del ejecutable
INSTALL_K3S_SKIP_DOWNLOAD=true \
# ejecutamos install.sh con argumento server
./install.sh server \
# token secreto usado por el maestro para unirse al cluster
--token "wCdC16AlP8qpqqI53DM6ujtrfZ7qsEM7PHLxD+Sw+RNK2d1oDJQQ0sBkIwy50Z/5" \
# un servicio de K3s que nos provee una capa de red de nivel 3 para K8s
--flannel-iface enp0s8 \
# dirección IP de bindeo de k3s
--bind-address $NODEIP \
# nombre del nodo y si ip que publicamos
--node-ip $NODEIP --node-name $HOSTNAME \
# desahabilitamos traefik que es una especie de proxy inverso
# o balanceador con funciones de capa 7, que realiza funciones
# de entrada de fuera de K8s
--disable traefik \
# deshabilitamos servicelb, un balancedor de carga de capa 3.
--disable servicelb \
# registramos kubelet
--node-taint k3s-controlplane=true:NoExecute
#--advertise-address $NODEIP
#--cluster-domain ''cluster.local''
#--cluster-dns "10.43.0.10"

# Copiamos el fichero yaml generado en /vagrant para
# que Vagrant pueda vincularlo con kubeconfig
cp /etc/rancher/k3s/k3s.yaml /vagrant
```

### 4. Caso de un nodo

```
#no instalamos k3s ya que disponemos del binario
INSTALL_K3S_SKIP_DOWNLOAD=true \
#ejecutamos el script con el argumento agent
./install.sh agent
# servidor al que meterse, es decir, dirección del cluster
--server https://$MASTERIP:6443 \
# token para registrarse en el cluster
--token "wCdC16AlP8qpqqI53DM6ujtrfZ7qsEM7PHLxD+Sw+RNK2d1oDJQQ0sBkIwy50Z/5" \
# ip y nombre del nodo que ofrecemos
--node-ip $NODEIP --node-name $HOSTNAME
--flannel-iface enp0s8
```

El *script* install.sh sirve para registrar kubelet como un proceso de systemd en nuestro caso .

## 6. Problemas

Los únicos problemas que se han encontrado han sido a la hora de configurar *Vagrant* para lanzar las máquinas. El problema tuvo dos partes, la primera era que la tarjeta de red que se proporcionaba en el *script* por parte del profesor, no cuadraba con la de mi equipo con *macOS*, por lo que tuve que modificarlo. La otra parte estaba en que las IPs proporcionadas por el profesor para las diferentes MV no funcionaba en mi red debido a que mi *router* trabaja con IPs en rango 192.168.0.x y las IPs proporcionadas eran de tipo 192.168.1.x.

Otro problema fue entender sin mucho conocimiento previo que hacían los *scripts* **provision.sh** e **install.sh**, cosa que se comprendió mejor tras leer la documentación de K3s[3] y de Vagrant[4].

## Referencias

- [1] K. en Español. Despliegue y uso de kubernetes. [https://www.youtube.com/channel/UCT83D9N048XeFc0ZQ3nc\\_9Q](https://www.youtube.com/channel/UCT83D9N048XeFc0ZQ3nc_9Q). Tutoriales de despliegue y explicación de conceptos.
- [2] I. F. FCI and C. F. Management. Tutoriales kubernetes. <https://www.youtube.com/channel/UCVjsceiYN0-5d4Ly93Lj0jQ>. Conceptos generales de kubernetes.
- [3] D. Project. K3s. <https://www.disasterproject.com/kubernetes-mas-simple-k3s/>.
- [4] H. Stack. Vagrant. <https://www.vagrantup.com/docs/index.html>.
- [5] O. Team. Practical introduction kubernetes. <https://kubernetesbyexample.com>. Guión de ejemplos.