Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

sisdis-pr-raft

RAFT & Kubernetes

Autor 1:	Toral Pallás, Héctor - 798095
Autor 2:	Pardos Blesa, Javier - 698910
Grado:	Ingeniería Informática
Curso:	2022-2023

25 de noviembre de 2022

Índice

1. Introducción

En esta práctica se plantea realizar el despliegue del sistema de almacenamiento clave/valor desarrollado en las dos prácticas anteriores, para ello se va a hacer uso de Kubernetes; una plataforma de código abierto para automatizar la implementación, el escalado y la administración de aplicaciones en contenedores.

2. Arquitectura

La configuración de despliegue se puede encontrar en el fichero statefulset_go.yaml. Dentro de este fichero podemos encontrar tres declaraciones distintas: una para el cliente, otra para los nodos réplica y por último un servicio de tipo ClusterIp para dar de alta los nodos raft en el servicio de DNS.

Los nodos raft son de tipo StatefulSet debido a la necesidad de conocer durante todo su período de vida su dirección IP que posteriormente se da de alta en un servicio de DNS para que las 3 réplicas se puedan conocer entre si, además, el tipo StatefulSet da la opción de crear volúmenes para mantener la persistencia de la aplicación aunque en este caso hemos delegado tal responsabilidad en los nodos raft. De esta manera si un nodo cae, el propio sistema clave valor (cluster de nodos raft) será el encargado de darle la información al nodo caído. A continuación se puede ver el fragmento de declaración en yaml que generaría el volumen para cada pod de nodo raft.

```yaml
kind: StatefulSet
spec:
  template:
    spec:
      containers:
        - name: servidor
          volumeMounts:
          - mountPath: "/data"
            name: raft
  volumeClaimTemplates:
  - metadata:
      name: raft
    spec:
      accessModes:
      - ReadWriteOnce
      resources:
        request:
          storage: 5Gi
        storageClassName: do-block-storage
```

Figura 1: Declaración del volumen para los pods de los nodos raft.

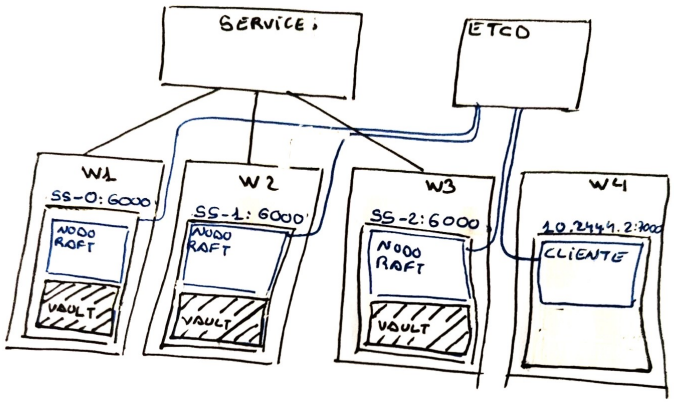

Figura 2: Arquitectura implementada para el despliegue en Kubernetes.

Ahora, mostramos como ha sido generada la arquitectura previamente mostrada en nuestro equipo para así poder comparar la.

```
$ kubectl get nodes -o wide
NAME                 STATUS   ROLES           AGE    VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION                     CONTAINER-RUNTIME
kind-control-plane   Ready    control-plane   2m11s  v1.25.3   172.20.0.6    <none>        Ubuntu 22.04.1 LTS   5.10.16.3-microsoft-standard-WSL2   containerd://1.6.9
kind-worker          Ready    <none>          100s   v1.25.3   172.20.0.3    <none>        Ubuntu 22.04.1 LTS   5.10.16.3-microsoft-standard-WSL2   containerd://1.6.9
kind-worker2         Ready    <none>          100s   v1.25.3   172.20.0.4    <none>        Ubuntu 22.04.1 LTS   5.10.16.3-microsoft-standard-WSL2   containerd://1.6.9
kind-worker3         Ready    <none>          99s    v1.25.3   172.20.0.2    <none>        Ubuntu 22.04.1 LTS   5.10.16.3-microsoft-standard-WSL2   containerd://1.6.9
kind-worker4         Ready    <none>          112s   v1.25.3   172.20.0.5    <none>        Ubuntu 22.04.1 LTS   5.10.16.3-microsoft-standard-WSL2   containerd://1.6.9
```

Figura 3: Workers generados con kind-with-registry.sh

```
$ kubectl get all -o wide
NAME        READY   STATUS    RESTARTS   AGE   IP           NODE           NOMINATED NODE   READINESS GATES
pod/c1      1/1     Running   0          38m   10.244.4.2   kind-worker4   <none>           <none>
pod/ss-0    1/1     Running   0          38m   10.244.1.2   kind-worker    <none>           <none>
pod/ss-1    1/1     Running   0          38m   10.244.2.2   kind-worker2   <none>           <none>
pod/ss-2    1/1     Running   0          38m   10.244.3.2   kind-worker3   <none>           <none>

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE   SELECTOR
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP    43m   <none>
service/ss-service   ClusterIP   None         <none>        6000/TCP   38m   app=rep

NAME                  READY   AGE   CONTAINERS   IMAGES
statefulset.apps/ss   3/3     38m   servidor     localhost:5001/servidor:latest`
```

Figura 4: Información sobre los pods y servicios en ejecución.

```
$ kubectl describe svc ss-service
Name:              ss-service
Namespace:         default
Labels:            app=rep
Annotations:       <none>
Selector:          app=rep
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                None
IPs:               None
Port:              servidor-port  6000/TCP
TargetPort:        6000/TCP
Endpoints:         10.244.1.2:6000,10.244.2.2:6000,10.244.3.2:6000
Session Affinity:  None
Events:            <none>
```

Figura 5: Información sobre el servicio especificado en statefulset_go.yaml

En esta imágen se puede ver como están conectados a este servicio los 3 pods servidores (ss-[i]).

3. Puesta en marcha

Compilación de ejecutables

```
CGO_ENABLED=0 go build -o cmd/srvraft/servidor cmd/srvraft/main.go
CGO_ENABLED=0 go build -o cmd/cltraft/cliente cmd/cltraft/main.go
```

Construcción de las imagenes de docker

```
docker build cmd/cltraft/ -t localhost:5001/cliente:latest
docker build cmd/srvraft/ -t localhost:5001/servidor:latest
```

Publicación de las imágenes al repositorio local

```
docker push localhost:5001/cliente:latest
docker push localhost:5001/servidor:latest
```

Despligue en el cluster

```
si no cluster:
    kind-with-registry.sh
fsi
./deployCluster.sh
```

Una vez se han desplegado los nodos raft y el cliente, se puede comprobar su estado y comenzar con la validación de la siguiente manera:

```
kubectl get pods            # Devuelve el estado de los pods
kubectl exec c1 -ti -- sh   # Abre una shell para el pod c1

# Dentro de cliente 1
cd usr/local/bin    # Direccion del ejecutable
./cliente           # Lanza el cliente de forma manual
```

4. Validación

Para verificar la correcta ejecución del algoritmo de consenso RAFT en el entorno creado a partir de Kubernetes, se ha realizado un cliente que implementa toda la operativa necesaria para comprobar el funcionamiento del sistema.

Este cliente es un programa interactivo por linea de comandos en el que el cliente puede hacer uso de la funcionalidades que se implementaron en el sistema RAFT en las últimas prácticas, como pueden ser:

- Obtener estado de un Nodo
- Obtener estado del Almacén
- Parar un Nodo
- Someter una operación (lectura / escritura)

El código se estructura en un main que contiene un switch que diferencia entre las distintas operativas.