SISTEMAS DISTRIBUIDOS

PRÁCTICA 5

874055 - Ariana Porroche Llorén
871627 - Irene Pascual Albericio

ÍNDICE:

INTRODUCCIÓN:

En esta práctica, hemos llevado a cabo el algoritmo de Raft integrándolo en un entorno de Kubernetes, donde desplegábamos y gestionábamos un sistema distribuido con tolerancia a fallos basado en un servicio de almacenamiento clave-valor.

Empezamos poco a poco, probando primero los ejemplos que nos proporcionaron. Esto nos sirvió para entender cómo funciona Kubernetes y cómo se despliegan correctamente las aplicaciones distribuidas. Una vez que ya lo entendimos, pasamos a trabajar con nuestro propio código de Raft.

Montamos un clúster con cuatro pods: tres que ejecutaban el código de Raft y uno para el cliente. Respecto a las 3 réplicas (los 3 pods primeros), decidimos usar un StatefulSet, ya que nos permite guardar el estado de cada pod y asignarles un nombre DNS fijo que conocemos de antemano. Esto lo implementamos en archivo statefulset_go.yaml.

El otro pod (el cliente), se mantiene activo durante un tiempo prolongado gracias al comando sleep. Esto ayuda para que podamos utilizar el pod para ejecutar el main.go del cliente sin que este se finalice.

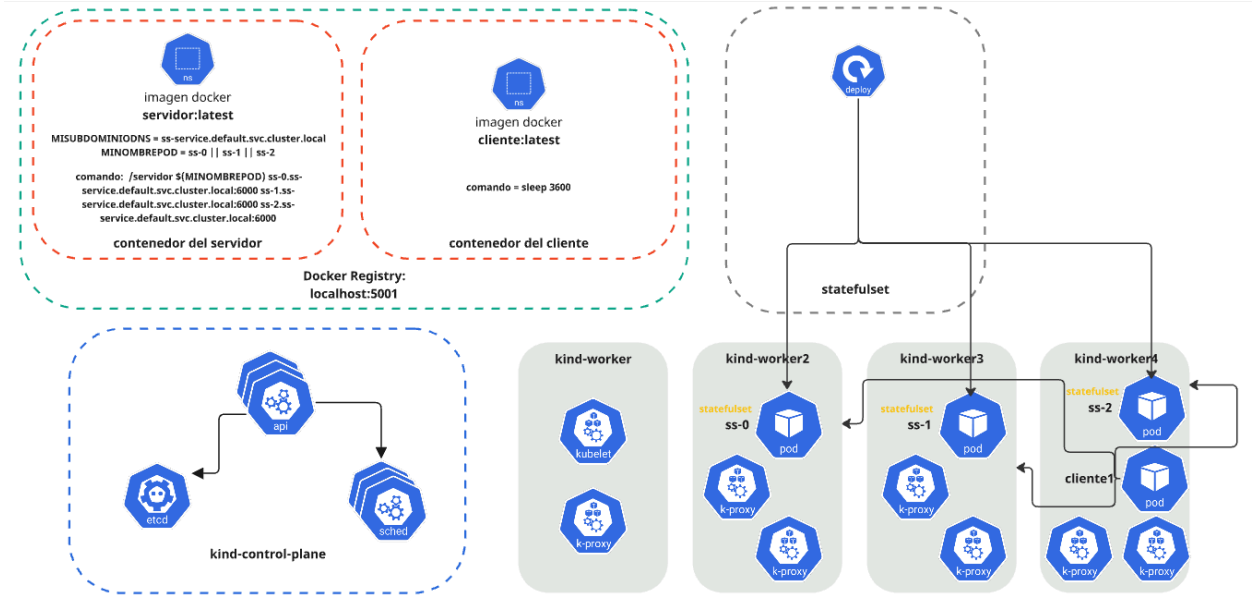
Luego, nos pusimos con el cliente, donde incorporamos un archivo main.go que básicamente seguía el esquema de los archivos test para poder verificar que se cumple todo. Con este cliente hicimos varias pruebas, como:

- Comprobar que se había elegido un líder
- Verificar que Raft podía recuperar el estado tras una caída.
- Introducir entradas en el sistema y asegurarnos de que se procesaban correctamente.

Para analizar cómo funcionaba todo, usamos los logs de los pods. Modificamos el código de Raft para que redirigiera sus mensajes a la salida estándar, lo que nos facilitó mucho las cosas. Al final, cuando todo funcionó perfectamente, creamos un script para automatizar los comandos y simplificar la ejecución.

En las siguientes secciones explicaremos con más detalle cómo lo hicimos, qué problemas encontramos y cómo los resolvimos los mismos.

ESQUEMA:



DISEÑO Y DESARROLLO:

Cluster de kubernetes

Primeramente, para poner en marcha el cluster de Kubernetes, hemos ejecutado el script que nos proporcionáis con el comando:

bash kind-with-registry.sh

Para comprobar que se habían creado correctamente los 3 workers y el cliente, ejecutamos el comando:

kubectl get nodes

La salida que obtuvimos fue la siguiente, donde podemos comprobar que todos los nodos están listos:

```
irene@pop-os:~/Descargas/MaterialDeAyudaParaAlumno/raft5$ kubectl get nodes
NAME                 STATUS   ROLES           AGE   VERSION
kind-control-plane   Ready    control-plane   13m   v1.31.2
kind-worker          Ready    <none>          13m   v1.31.2
kind-worker2         Ready    <none>          13m   v1.31.2
kind-worker3         Ready    <none>          13m   v1.31.2
kind-worker4         Ready    <none>          13m   v1.31.2
```

Cluster de Raft

Dentro de los 3 tipos de nodos que podíamos utilizar, hemos utilizado las réplicas con controlador StatefulSet para los nodos raft, ya que conservan el estado tras la caída de una réplica. Para el cliente, como en nuestro caso no requerimos de tolerancia a fallos, utilizamos un Pod normal.

Para ello, modificamos el fichero **statefulset_go.yaml**, añadiendo la creación del pod del cliente y los comandos a ejecutar por el servidor:

```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: cliente1
spec:
  restartPolicy: Never
  containers:
  - name: cliente
    image: localhost:5001/cliente:latest
    command: ["/bin/sh"]
    args: ["-c", "sleep 3600;"]
    ports:
    - containerPort: 7000
```

```yaml
command:
- /servidor
- $(MINOMBREPOD)   #para lanzar el numero de replica correcta
- ss-0.ss-service.default.svc.cluster.local:6000
- ss-1.ss-service.default.svc.cluster.local:6000
- ss-2.ss-service.default.svc.cluster.local:6000
```

Una vez definidos los pods que vamos a crear, ejecutamos el script **creacion.sh**, el cual se encarga de limpiar todo lo creado anteriormente y crear el nuevo clúster y las réplicas raft. Este script realiza las siguientes acciones:

Eliminar el clúster anterior, para lo que ejecuta la instrucción:

./eliminar_cluster.sh &>/dev/null

Este script elimina el cluster de kubernetes creado anteriormente (primero para todos los nodos y luego los elimina):

```
docker stop kind-worker
docker stop kind-worker2
docker stop kind-worker3
docker stop kind-worker4
docker stop kind-control-plane
docker stop kind-registry
docker rm kind-registry
docker rm kind-control-plane
docker rm kind-worker
docker rm kind-worker2
docker rm kind-worker3
docker rm kind-worker4
```

A continuación, volvemos a crear el clúster con el comando:

./kind-with-registry.sh

Ahora eliminamos los ficheros ejecutables del servidor y del cliente:

rm Dockerfiles/servidor/servidor
rm Dockerfiles/cliente/cliente

Compilamos el main tanto del servidor como del cliente y lo movemos al respectivo directorio en Dockerfiles:

CGO_ENABLED=0 go build -o ../../Dockerfiles/servidor/servidor main.go
CGO_ENABLED=0 go build -o ../../Dockerfiles/cliente/cliente main.go

Ahora creamos el contenedor del servidor y lo subimos al repositorio local:

docker build . -t localhost:5001/servidor:latest
docker push localhost:5001/servidor:latest

Hacemos lo mismo con el cliente:

docker build . -t localhost:5001/cliente:latest
docker push localhost:5001/cliente:latest

Ya por último, elimina cualquier pod que pudiera estar creado y crea los 3 pods de las réplicas y el pod del cliente:

kubectl delete statefulset ss &>/dev/null
kubectl delete pod cliente1 &>/dev/null
kubectl delete service ss-service &>/dev/null
kubectl create -f statefulset_go.yaml

Con este script, ya tendríamos en marcha todo el cluster de raft, con los 3 nodos raft funcionando y el cliente también. Para comprobar esto, ejecutamos el siguiente comando y obtenemos la siguiente salida:

kubectl get pods

```
irene@pop-os:~/Descargas/MaterialDeAyudaParaAlumno/raft5$ kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
cliente1   1/1     Running   0          13m
ss-0       1/1     Running   0          13m
ss-1       1/1     Running   0          13m
ss-2       1/1     Running   0          13m
```

Para comprobar que las 3 réplicas raft ya han elegido un líder, el cual les está mandando latidos de corazón, miramos los logs de cada pod:

kubectl logs ss-0
kubectl logs ss-1
kubectl logs ss-2

Ahora queremos ejecutar el cliente, el cual va a comprobar que efectivamente se haya elegido un líder, someterá una entrada de escritura, hará caer a un nodo de los que no son líder, y entonces someterá otra operación de lectura. Ya que hemos utilizado pods con estado para las réplicas raft, cuando vuelva la réplica caída, podremos observar en los logs como recupera el estado anterior, en el cual ya tenía sometida la operación y entonces someterá la nueva. Para ejecutar el cliente, utilizamos el comando:

kubectl exec -ti cliente1 -- /bin/sh

Este comando nos abre una terminal en el cliente, y entonces le tenemos que mandar ejecutar el main, para lo que usamos:

./cliente

Podemos comprobar cómo hemos obtenido los resultados que queríamos, que se ha elegido correctamente al líder, y se han sometido correctamente las operaciones:

```
irene@pop-os:~/Descargas/MaterialDeAyudaParaAlumno/raft5$ kubectl exec -it cliente1 -- /bin/sh
/ # ./cliente
Entrando en el main del cliente
reply de  0 ReplyMandato:  1 es Lider:  false
reply de  1 ReplyMandato:  1 es Lider:  true
Encontrado el líder 1
Ya se ha elegido el líder:  1
Vamos a someter la operación 1 al líder 1
Vamos a someter la operación 1 al líder 1
Operación 1 sometida correctamente, valor devuelto: 5
Vamos a matar a la réplica 0
Vamos a someter la operación 2 al líder 1
Vamos a someter la operación 2 al líder 1
Operación 2 sometida correctamente, valor devuelto: 5
```

Para escribir el main del cliente, hemos utilizado funciones muy similares a las del test, como pruebaUnLider(), comprometerEntrada() o pararNodoNoLider().