Práctica 5:
SISTEMAS DISTRIBUIDOS

SERVICIO DE ALMACENAMIENTO BASADO EN RAFT EN KUBERNETES

Jorge Hernández Aznar (872838) y Paula Blasco Díaz (874320)

ÍNDICE

1. Introducción

El objetivo de esta última práctica es implementar un servidor con funcionalidad de almacén de pares clave/valor con réplicas basado en Raft a través de Kubernetes. Kubernetes es una herramienta que permite automatizar el despliegue de aplicaciones en múltiples máquinas mediante contenedores de Docker. Para ello, se ha de adaptar el código desarrollado en las dos anteriores prácticas para que pueda funcionar en un cluster local en lugar de en las máquinas raspberries.

2. Diseño e Implementación

2.1 Implementación de cliente y servidor en Golang

Inicialmente, el fichero *cmd/srvraft/main.go* con la declaración del servidor ha sido modificado de modo que las direcciones de los distintos nodos sean las correspondientes al cluster local en lugar de las direcciones de los nodos raft de las máquinas raspberries junto con sus puertos. De acuerdo a esto, cada nodo deberá recibir como primer argumento su nombre dentro de la red, que serán raft-0, raft-1 y raft-2 respectivamente.

Seguidamente, en el módulo *internal/raft/raft.go* con implementación de raft no ha habido modificaciones realizadas, a excepción de un sleep introducido en la función *NuevoNodo()* para poder estabilizar la ejecución de la aplicación y garantizar que todos los procesos necesarios están activos cuando empiece la comunicación entre ellos.

Finalmente, se ha creado el archivo *pkg/cltraft/cltraft.go* para desempeñar el papel de cliente debido a que no se dispone de una batería de pruebas que use ssh para comprobar el funcionamiento. Este cliente realiza operaciones de escritura y lectura de forma intermitente cada 2 segundos y con un almacén interno comprueba que los valores son los esperados. Asimismo se asegura de que la operación haya sido sometida al nodo que desempeñe el papel de líder.

2.2 Organización de la aplicación mediante Kubernetes

Para el correcto despliegue de la aplicación en Kubernetes es necesario en primer lugar definir las imágenes que se usarán para los contenedores de docker. Para ello se crean dos dockerfiles para el cliente y el servidor que son los siguiente respectivamente.

```
FROM alpine
COPY cltraft /usr/local/bin/cltraft
EXPOSE 7000

FROM scratch
COPY srvraft /usr/local/bin/srvraft
EXPOSE 6000
```

Las imágenes usadas son alpine y scratch basadas en los ejemplos proporcionados de dockerfiles, en ambos casos se copia el binario ejecutable y se establecen los puertos en los que estarán habilitados.

Para garantizar el despliegue de la aplicación basada en el algoritmo Raft en Kubernetes, se ha utilizado un fichero de configuración YAML denominado statefulset_go.yaml. Este archivo define la creación de un conjunto de recursos, que incluyen un servicio para habilitar la comunicación entre nodos y un StatefulSet para manejar las réplicas de los nodos Raft, así como un Pod adicional para el cliente que interactúa con el sistema. A continuación, se describen las principales secciones del fichero.

```
apiVersion: v1
kind: Service
metadata:
  name: raft-service
  labels:
    app: rep
spec:
  clusterIP: None
  selector:
    app: rep
  ports:
  - port: 6000
    name: servidor-port
    protocol: TCP
    targetPort: 6000
```

En este primer apartado se puede observar la declaración del servicio raft-service. Este servicio permite que los Pods del StatefulSet se comuniquen entre sí y habilita su descubrimiento dentro del clúster. Al no definir ninguna IP del cluster el dns de cada pod establecerá su dirección como "*<pod-name>.<service-name>*", por ejemplo "*raft-0.raft-service*". El puerto indicado es el 6000 y coincide con el declarado en el dockerfile del servidor.

```
kind: StatefulSet
apiVersion: apps/v1
metadata:
  name: raft
spec:
  serviceName: raft-service
  replicas: 3
  podManagementPolicy: Parallel
  selector:
    matchLabels:
      app: rep
  template:
    metadata:
      labels:
        app: rep
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: servidor
        image: localhost:5001/servidor:latest
```

```
        env:
        - name: MISUBDOMINIODNS
          value: raft-service.default.svc.cluster.local
        - name: MINOMBREPOD
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        command:
        - srvraft
        - $(MINOMBREPOD)
        ports:
        - containerPort: 6000
```

A continuación, se define el servidor como un StatefulSet que permite que haya unicidad de pods y orden de puesta en marcha. Se establecen 3 réplicas y se vincula con el servicio raft-service recién declarado. Además se selecciona la imagen creada en docker con el dockerfile y se declara el comando a ejecutar de *srvraft* para arrancar los nodos pasando como argumento el nombre del nodo.

```
apiVersion: v1
kind: Pod
metadata:
  name: client
spec:
  restartPolicy: OnFailure
  containers:
  - name: cliente
    image: localhost:5001/cliente:latest
    command:
    - cltraft
    ports:
    - containerPort: 7000
```

Por último, se declara el pod del cliente de forma análoga al servidor se selecciona la imagen del cliente y se ejecuta el comando *cltraft*. También se establece el rearranque en caso de fallo y el puerto siendo el 7000 al igual que en el dockerfile del cliente.

3. Pruebas y verificación

3.1 Despliegue de la aplicación

El despliegue de la aplicación ha sido llevado a cabo en un máquina física personal con Ubuntu 24.04. Ha sido necesario instalar las herramientas *docker*, *kind* y *kubectl*. Para automatizar el despliegue se ha creado el script *restart_distributed_system.sh*. Este se muestra fragmentado a continuación.

```
#!/bin/bash

echo "Eliminando cluster..."
docker stop kind-worker
docker stop kind-worker{2,3,4}
docker stop kind-control-plane
docker stop kind-registry
```

```
docker rm kind-worker
docker rm kind-worker{2,3,4}
docker rm kind-control-plane
docker rm kind-registry

echo "Creando cluster..."
./kind-with-registry.sh
```

Primero se detienen y eliminan los contenedores previos y se ejecuta el script proporcionado en los ejemplos *kind-with-registry.sh* para iniciar de nuevo el cluster.

```
echo "\nRecompilando los ejecutables del cliente y servidor..."
rm Dockerfiles/cliente/cltraft
rm Dockerfiles/servidor/srvraft
CGO_ENABLED=0 go build -o Dockerfiles/servidor/srvraft cmd/srvraft/main.go
CGO_ENABLED=0 go build -o Dockerfiles/cliente/cltraft pkg/cltraft/cltraft.go

echo "Creando las imagenes en Docker..."
docker build Dockerfiles/servidor/. -t localhost:5001/servidor:latest
docker push localhost:5001/servidor:latest
docker build Dockerfiles/cliente/. -t localhost:5001/cliente:latest
docker push localhost:5001/cliente:latest
```

A continuación se compilan los ejecutables del servidor y del cliente con sus respectivas rutas y se crean las imagenes de docker.

```
echo "Ejecutando Kubernetes..."
kubectl delete statefulset raft
kubectl delete pod client
kubectl delete service raft-service
kubectl create -f statefulset_go.yaml
```

Finalmente, se eliminan los pods y servicios previos y se procede a crear la aplicación definida en el fichero *statefulset_go.yaml*.

3.1 Resultados obtenidos

Tras ejecutar el script *restart_distributed_system.sh* podemos ver que los pods son creados correctamente mediante el comando *kubectl get pods*.

```
NAME     READY   STATUS    RESTARTS   AGE
client   1/1     Running   0          31m
raft-0   1/1     Running   0          31m
raft-1   1/1     Running   0          31m
raft-2   1/1     Running   0          31m
```

Además, mediante el comando *kubectl logs client*, podemos acceder a los logs internos del nodo en el cual obtenemos que el flujo de acciones es el esperado, escribiendo y leyendo de los pods de forma correcta.

```
Operación de escritura: clave = a, valor = 1
Operación 1 sometida a 2
Operación de lectura: clave = a
Valor correcto leído: a = 1
Operación de escritura: clave = b, valor = 2
```

```
Operación 1 sometida a 2
Operación de lectura: clave = b
Valor correcto leído: b = 2
Operación de escritura: clave = c, valor = 3
Operación 1 sometida a 2
Operación de lectura: clave = c
Valor correcto leído: c = 3
Operación de escritura: clave = d, valor = 4
Operación 1 sometida a 2
Operación de lectura: clave = d
Valor correcto leído: d = 4
```

Finalmente podemos observar que si eliminamos manualmente un pod con *kubectl delete pod raft-2*, y a continuación observamos de nuevo los pods el nodo se reinicia automáticamente.

```
NAME     READY   STATUS    RESTARTS   AGE
client   1/1     Running   0          34m
raft-0   1/1     Running   0          34m
raft-1   1/1     Running   0          34m
raft-2   1/1     Running   0          11s
```

4. Referencias

Documentación de Kubernetes:
https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/
https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/
https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/
https://kubernetes.io/docs/tutorials/kubernetes-basics/scale/scale-intro/
https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/

ChatGPT OpenAI:
https://chat.openai.com