30221/39521 - Sistemas Distribuidos
Universidad de Zaragoza, curso 2024/2025

Práctica 5: **Kubernetes y Raft**

Resumen

En esta práctica, se plantea utilizar Kubernetes para poner en funcionamiento un servicio replicado basado en Raft.

Este trabajo incluye redactar una memoria, escribir comandos, manifiestos, código fuente y elaborar un juego de pruebas. El texto de la memoria y el código deben ser originales. Copiar supone un cero en la nota de prácticas.

1. Objetivos de la práctica

- El desarrollo completo de la recuperación de estado de una réplica de Raft, trás una caida.
- El despliegue del servicio de almacenamiento clave/valor basado en Raft, de la práctica 5, en Kubernetes.
- La comprobación del correcto funcionamiento de la recuperación completa del estado de una réplica en el contexto de su despliegue adecuado en Kubernetes.
- **Opcional** Modificación tanto del servidor rpc (main.go) como de la llamada *CallTimeout*, para utilizar el protocolo TLS en las llamadas.

Cada uno de los servidores (réplicas) de almacenamiento debe ejecutarse en un nodo (worker) Kubernetes diferente.

2. Ejecución del servicio de almacenamiento, basado en Raft, en Kubernetes

Se plantea diseñar la ejecución del servicio clave/valor de la práctica nº 5, en Kubernetes.

Previamente, se debe completar y comprobar la restitución completa de estado en un nodo Raft recuperado en ejecuciones locales sin Kubernetes.

Para ello, en primer lugar, se debería adaptar el código Golang de la práctica nº 5, ya que la puesta en marcha de nodos distribuidos será realizada por Kubernetes, en lugar de ssh. Para ello se compilarán tanto el código de servidor de almacenamiento y el de clientes y pruebas en varios ejecutables. Posteriormente se creará un contenedor Docker para cada uno, y se pondrá accesible en un registro local de contenedores de Docker para que puedan ser obtenidos por Kubernetes. Se deberá decidir los puertos y como pasar las direcciones de red y puertos a cada ejecutable.

En segundo lugar, la ejecucion de cada tipo nodo ejecutable Golang de la aplicación debería efectuarse en su tipo de Pod específico. Para la puesta en marcha de estos Pods hay que decidir que tipos de controladores de ejecución en Kubernetes es más adecuado, dadas la necesidades de descubrimiento y tolerancia a fallos, entre : Pods básicos, Deployments(con ReplicaSets subyacentes) y StatefulSets. Tener presente que, salvo en los Pods mas básicos y por configuración explicita y especifica, el resto de controladores de Kubernetes tienen incorporado las funcionalidades de tolerancia a fallos.

Se necesita reflexionar sobre las necesidades de los nodos distribuidos de la aplicación, tanto en tolerancia a fallos como para que puedan ser *descubiertos* por los otros nodos. De esta forma, podreis determinar cual de los controladores es el más adecuado para cada tipo de nodo distribuido que funciona en vuestra aplicación.

Algunas sugerencias para una solución :

- Comprobar que vuestra aplicación clave/valor recupera una réplica caida, con restitución completa de estado(tanto registro de Raft como máquina de estados de la aplicación) en las réplicas recuperadas. Una posibilidad, para ayudar en el test, es modificar la llamada rpc emphObtenerEstadoNodo, para obtener el registro Raft y el mapa clave/valor.

- Decidir el controlador para la puesta en marcha de los clientes del sistema de almacenamiento.

- Plantear qué controlador se necesita para la puesta en marcha de las réplicas

de almacenamiento (inicialmente con réplicas de Kubernetes). La mayor parte de pruebas es suficiente con 3 réplicas. En esta etapa podeis probar eliminar Pods de almacenamiento para ver como Kubernetes regenera el Pod eliminado y vuestro código Golang repara vuestro sistema e integra el nuevo Pod/réplica en vuestra aplicación.

- Comprobar que Kubernetes automatiza la recuperación de replicas caidas, con restitución de estado en ellas.

Para la parte opcional (incorporación de comunicación segura mediante cifrado a las llamadas RPC, utilizando el protocolo TLS) podeis generar la pareja de claves pública/-privada y el certificdo digital autofirmado (sin autorida certificadora externa) mediante el comando *openssl* en Linux. Y podeis la clave privada y el certificado, sea en 2 ficheros, sea en 2 variables en el programa. Teneis disponible algunos ejemplos en el directorio *AyudaTLS* que acompaña a este guión para comunicación TCP. Las modificaciones a realizar para utilizar con RPC son mínimas. En el cliente rpc, en lugar de llamar directamente con *rpc.Dial*, se podría utilizar rpc.NewClient con algún elemento adicional. Para el servidor RPC, la modificación es más simple.

3. Notas sobre la puesta en marcha

En los anexos A y B teneis detalles sobre la puesta en marcha de Kubernetes y la ejecución de aplicaciones en Kubernetes.

Seguir las guías de diseño e implementación de las 2 prácticas anteriores en lo que respecta al código Golang.

Para la puesta en marcha de Kubernetes y de aplicaciones en Kubernetes, teneis disponibles diferentes ficheros que acompañan a este guión. Abordan la puesta en marcha de un cluster Kubernetes de 4 nodos, en un solo ordenador, mediante técnicas de contenedores anidados dentro de contenedores, mediante la herramienta *kind* [3] sobre *Docker*. Además, incluyen ejemplos de implementación y de uso de los tipos de controladores Kubernetes aconsejados para diferentes aspectos del despliegue de Raft y sus clientes. En los *anexos* de este guión se explican algunos de estos aspectos.

También teneis disponibles algunas referencias bibliográficas a elementos significativos de Kubernetes que pueden ser útiles en esta práctica.

La ejecución de vuestras aplicaciones, implementadas en Golang, será interna a Kubernetes. Es decir, que todos los nodos distribuidos Golang (incluidos clientes de almacenamiento y pruebas) se ejecutarán en Pods de Kubernetes, asegurando la comunicación

completa entre ellos.

3.1. Validación

Adaptar las consideraciones de validación de las 2 prácticas anteriores a este caso, adaptando el código de tests que sean convenientes.

El código/modulo de tests se puede ejecutar desde otro Pod que no tiene requerimientos de tolerancia a fallos ni descubrimiento. Para obtener las sálidas de tests se puede utilizar la funcionalidad/comando *"kubectl logs ..."* sobre ese Pod, en validaciones automáticas, o *"kubectl exec nomPod -ti ..."* para validaciones interactivas.

El desarrollo y la validación se puede efectuar sobre vuestro propio ordenador y será, a diferencia de prácticas anteriores, local en una sola máquina física. Para ello es necesario disponer, como mínimo, de *25 GB de espacio en disco, 3 GB de RAM.*

4. Criterios de Evaluación

La realización de las prácticas se puede realizarse por 1 persona o por parejas, pero los dos componentes de la pareja deberán entregarla de forma individual.

- La memoria debería incluir una explicación del diseño y desarrollo de la puesta en marcha de las aplicaciones en Kubernetes.
- **Cada una de las réplicas debe ejecutarse en un nodo (worker) Kubernetes diferente**.

5. Entrega y Defensa

Se debe entregar, tanto la memoria (longitud máxima de 7 páginas y anexos de 15 páginas máximo) como los ficheros de código fuente, en un solo fichero en formato tar.gz a través de moodle2 en la actividad habilitada a tal efecto. La fecha límite es el *día anterior a la sexta sesión de prácticas*, y la evaluación se realizará en la sexta sesión de prácticas.

La entrega debe contener los diferentes ficheros de código Golang, shell y manifiestos Kubernetes, y la memoria, en formato pdf. El nombre del fichero tar.gz debe indicar

apellidos del alumno y nº de práctica. *Aquellos alumnos que no entreguen la práctica a través de moodle2 no serán calificados.*

Referencias

[1] https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-debian-10

[2] https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

[3] https://kind.sigs.k8s.io/docs/user/quick-start/

[4] https://kind.sigs.k8s.io/docs/user/local-registry/

[5] https://kubernetes.io/docs/concepts/

[6] https://kubernetes.io/docs/tasks/

[7] https://kubernetes.io/docs/reference/kubectl/cheatsheet/

[8] https://www.mirantis.com/blog/introduction-to-yaml-creating-a-kubernetes-deployment/

[9] https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/

[10] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

[11] https://kubernetes.io/docs/tasks/run-application/run-stateless-application-deployment/

[12] https://blog.openshift.com/kubernetes-statefulset-in-action/

[13] https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

[14] https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/

A. Anexo : Puesta en marcha de Kubernetes

En primer lugar, se necesita un ordenador con sistema Linux (Debian, Ubuntu, Manjaro, ArchLinux, etc) que disponga de un minimo de 25 Gb de disco y 4Gb de Ram.

Deberá *instalarse*, previamente en dicho ordenador, una versión > v1.12 del software "Docker". Para ello podeis utilizar la referencia [1].

La configuración de funcionamiento que se propone, para Kubernetes, es la que se conoce como contenedor sobre contenedor (docker over docker). Es decir, que en lugar de VMs nativas o máquinas físicas, se utilizan contenedores para albergar los nodos Kubernetes (con una versión de Ubuntu ejecutandose dentro). La ejecución será más ágil y el uso de recursos más bajo.

Para ello, se utilizará las herramientas *kind* [3] para la puesta en marcha del cluster y *kubectl* [7] para su gestión. Ambas están disponibles, para su ejecución, en todas las máquinas del laboratorio 1.02. Para utilizarlos en vuestros propios ordenadores (portátiles y demás), podeis obtener ambos ejecutables, de kind y kubectl, mediante la herramienta *scp* desde :

central.cps.unizar.es:/misc/usuarios/unai/SistDist/

Ubicar ambos ejecutables en un directorio del PATH de ejecución de vuestra propia máquina.

La puesta en marcha de la versión 1.19.1 de Kubernetes se realizará con el programa shell contenido en el fichero *"kind-with-registry.sh"* que acompaña a este guión y que se ha adaptado a partir de [4]. Este programa permite instalar, configurar y poner en marcha un conjunto de 5 nodos Kubernetes(1 master y 4 workers), cada uno en un contenedor de Docker. Además, pone en marcha un registro para repositorio de contendores local en Docker en *localhost:5000* para su uso por Kubernetes.

La operativa es sencilla :

Activar dicho fichero como ejecutable con chmod.

Arrancar el cluster completo :

```
$ ./kind-with-registry.sh
```

Se utilizará *kubectl* como herramienta ya configurada para comunicar con el cluster Kubernetes.

Comprobar el funcionamiento de los nodos Kubernetes con :

```
$ kubectl get nodes -o wide
```

Para ver los contenedores que ejecutan estos nodos de Kubernetes (en lugar de máquinas físicas) :

```
$ docker ps -a -s
```

Ya teneis funcionando el cluster de Kubernetes !

Comandos adicionales que os pueden ser utiles son :

- Eliminación de un nodo del cluster kubernetes : `kubectl delete node <node name>`
- Eliminación del cluster entero : `kind delete <cluster name>`

Un resumen de comandos Kubernetes podeis encontrar en [7].

Si habeis terminado de utilizar el cluster, podeis eliminarlo por completo con el comando :

```
$ kind delete cluster
```

B. Anexo: Ejecución de aplicaciones Golang en Kubernetes

Como imagen base se utilizará, en el caso de un cliente, una imagen de contenedor Docker vacía, es decir, de tamaño cero como la imagen *"scratch"* del repositorio de Docker, como en el caso del ejemplo de *servidor* del modulo golang *conversa* que teneis disponible, o una imagen mínima de Linux como la de *"alpine"* (5MB), como en el caso del ejemplo de *cliente* del mismo modulo *conversa*.

En primer lugar, en lugar de utilizar solo comandos como se vió en ejercicios, se utilizan ficheros declarativos de recursos, denominados manifiestos, y controladores a ejecutar. Son los ficheros que tienen el sufijo *yaml*, ya que utilizan el formato yaml para codificar la configuración de carga de recursos y ejecución de aplicaciones en Kubernetes. En la referencia [8] se provee una explicación de su utilización.

En segundo lugar, para nombrar nodos distribuidos de vuestros programas Golang, utilizaremos en unos casos el nombre completo (FQDN) DNS y en otros casos la @ IP (como en anteriores prácticas). El problema es que la asignación de @ IP a Pods no es

predecible, pero el nombre DNS, en cambio, lo podemos fijar a priori para descubrirlo fácilmente. En los ejemplos se ve la utilización del nombre DNS en arranque de *Pods básicos* y con controlador *StatefulSet*. En cambio, utilizamos IPs para ejecucion con controlador *Deployment*. Los nombres DNS solo se resuelven, a IPs, dentro de los Pods de Kubernetes.

En los ficheros adjuntos teneis ejemplos para cada uno de los tipos de ejecución que son utiles en esta práctica :

- `pods_go.yaml` : Ejecuta 3 pods sin tolerancia a fallos y dandoles de alta el nombre completo DNS a uno de ellos, el servidor. Es decir que se pueden contactar por el nombre DNS completo con él. Documentacion en [9].

- `deploy_go.yaml` : Ejecución de 3 réplicas con controlador *Deployment*. No hay nombres DNS y las IPs de Pods se conocen solo a posteriori, en el arranque. Documentación en [10] y [11].

- `statefulset_go.yaml` : Puesta en marcha de réplicas con controlador *StatefulSet*. Tenemos un nombre DNS que podemos conocer a priori, definido por el nombre del *StatefulSet*. Documentación en [13] y [14].

Para cada uno de ellos, debe prepararse, previamente, cada contenedor que contiene cada ejecutable de vuestro código Golang que necesitais ejecutar en Pods (puestos en marcha por alguno de los recursos previamente comentados). Para evitar necesitar librerías dinámicas (glibc u otros), compilar estáticamente los programas Golang mediante el comando (en el ejemplo provisto, sería en los directorios *cliente* y *servidor* del modulo *conversa*) de golang que teneis disponible :

```
$ CGO_ENABLED=0 go build -o .  ./...
```

La preparación de cada contenedor se realiza mediante *Docker* [1] y ficheros de configuración *Dockerfile* [2]. Ejemplos básicos de estos ficheros de configuración para creación de nuevos tipos de contenedores podeis encontrar en el directorio *Dockerfiles*. Para crear los nuevo tipos de contenedores, que contienen vuestros ejecutables, os situais en cada directorio donde se ubica cada fichero Dockerfile, y el fichero ejecutable a incluir en el contenedor, y ejecutar por ejemplo:

```
$ docker build . -t localhost:5001/{servidor, cliente}:latest
```

Y posteriormente los subis al repositorio local para que esten accesible para Kubernetes con :

```
$ docker push localhost:5001/{servidor, cliente}:latest
```

Una vez creados los contenedores y subidos al registro, se puede realizar ya la puesta en marcha de aplicaciones que utilicen dichos contenedores como imagenes ejecutables, como se define en los 3 manifiestos de ejemplo previos: *pods_go.yaml*, *deploy_go.yaml* y *statefulset_go.yaml*. La puesta en marcha se puede realizar con los scripts shell : *go_pods.sh*, *go_deployment.sh*, *go_statefulset.sh*. Comenzar por *go_pods.sh*. Tener en consideración que el pod *c2* (cliente automático) puede penerse en funcionamiento antes que el pod *s1* (servidor). Eliminar el pod *c2*, y someter de nuevo el manifiesto *pods_go.yaml* con comando "*kubectl create*". Solo se creará de nuevo el pod *c2*, y mostrará un error no consecuente con los otros. Comprobar la salida del pod *c2*.

Para *manipular y depurar los Pods* podeis utilizar subcomandos de *kubectl* como : *exec* y *logs*.

En el caso de *exec*, permite entrar en un Pod (y su contenedor), y ejecutar un comando. En particular, se puede entrar con un shell como en el siguiente ejemplo :

```
$ kubectl exec c1 -ti -- sh
```

En el cual, nos introducimos en el Pod de nombre *c1* y ejecutamos un interprete shell. A través de él podemos comprobar la situación interna del contenedor utilizando comandos Unix en Alpine Linux (*ifconfig, nslookup, apk add procps, ps auxww*, etc). En particular podemos instalar software adicional con el gestor de paquetes de Alpine Linux (apk). Podeis buscar los comandos que os interesen en la web : `https://pkgs.alpinelinux.org/contents`. Y podeis obtener su ip (ifconfig) e invocar a programa "cliente" en interactivo (3 parámetros: @ cliente, @ servidor, y cualquier valor - para último parámetro -):

```
$  ./cliente  <IP_propia>:7000  <@ servidor>:<puerto servidor>  loquesea
```

En interactivo, este cliente se puede utilizar para comunicarse con cualquier tipo de despliegue del servidor, es decir, si el servidor se ejecuta solo como Pod, o Deployment, o StatefulSet. Sea con recurso Service sin cabeza (headless - campo clusterIP: None -) o IP de recurso Service con cabeza (sin campo clusterIP), es decir con IP o con nombre DNS de pods.

La obtención de la salida stdout, de la ejecución de terminal de un Pod, se realiza con *logs* de la siguiente forma :

```
$ kubectl logs s1
```

Ayuda para subcomandos de *kubectl* :

```
$ kubectl -h
```

```
$ kubectl exec -h
```

Podeis obtener información interna más detallada de cualquier recurso con :

```
$ kubectl describe -h
```

Existe la posibilidad de obtener diferentes datos de recursos de kubernetes mediante salida en formato json o yaml.

Por ejemplo, para obtener la especificación declarativa de un recurso Kubernetes especifico :

```
$ kubectl get deployment de -o yaml
```

Esta salida a stdout, con supresión de algunos campos y adaptación de los necesarios, puede servir para definir un fichero yaml declarativo para cargar de nuevo en Kubernetes.

Otro ejemplo, para obtener IPs de nodos Kubernetes :

```
$ kubectl get nodes -o jsonpath='{.items[*].status.addresses[?(@.type=="InternalIP")].address}'
```

Para que no os ocupe en exceso el disco, cuando termineis de utilizar las imagenes de contenedores creadas (residen habitualmente en /var/lib/docker), podeis suprimirlos del disco con el comando :

```
$ docker image rm ...
```