Práctica 6

Kubernetes y Raft

Javier Herrer Torres (NIP: 776609)
Javier Fuster Trallero (NIP: 626901)

Sistemas Distribuidos
Grado en Ingeniería Informática

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza
Curso 2020/2021

1. Objetivos

Los objetivos de esta práctica son:

1. La utilización de Kubernetes como plataforma de ejecución del sistema de almacenamiento clave/valor basado en vistas.
2. Puesta en marcha de un servicio replicado basado en el modelo de Raft [1] y [2].
3. Implementación y puesta en marcha de un servicio distribuido y tolerante a fallos de semáforos.

2. Ejecución del servicio de almacenamiento primario/copia, basado en vistas, en Kubernetes

2.1. Metodología

Se ha adaptado el código Golang de la práctica nº 5, ya que la puesta en marcha de nodos distribuidos es realizada por Kubernetes, en lugar de ssh. Por ello, se ha eliminado de los tests la puesta en marcha y parada de los procesos distribuidos. Se han compilado los códigos del servidor de vistas, el servidor de almacenamiento y el de clientes y pruebas en varios ejecutables. Posteriormente, se ha creado un contenedor Docker para cada uno, y se han puesto accesibles en un registro local de contenedores de Docker para que puedan ser obtenidos por Kubernetes.

Los **Dockerfile** de los contenedores incluyen la imagen que se usa para ese contenedor, se ha empleado *alpine* al ser más completo y permitir realizar más interacciones útiles para la depuración (`FROM alpine`). Después se incluye el binario anteriormente compilado mediante `COPY`. Y, por último, se expone el puerto que corresponda (el mismo que en el fichero de configuración) con `EXPOSE`.

La ejecución de cada tipo nodo ejecutable Golang de la aplicación se efectúa en su tipo de Pod específico.

- **Pods básicos** para el gestor de vistas y para los clientes del servidor de almacenamiento, ya que no se necesitan funcionalidades de tolerancia a fallos.
- **Deployment** (con ReplicaSets subyacentes) para los servidores de almacenamiento basado en una gestión estilo Puppet de puesta en marcha de un ReplicaSet con 3 réplicas.

Además, los pods de la aplicación necesitan ser *descubiertos* por los otros pods. Para ello, se crea un servicio (*Service*) que actúa como DNS y se utilizan selectores y etiquetas para ubicar los pods en la misma red (ver figura 1). Esto permite obviar el problema de los puertos compartidos, ya que cada pod tiene los suyos propios.

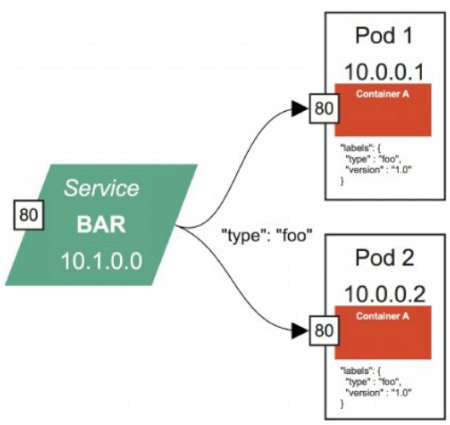

Figura 1: Ejemplo de Servicio en Kubernetes (selectores y etiquetas)

2.2. Resultados

Se ha comprobado que Kubernetes automatiza la recuperación de réplicas caídas, y que, además, la aplicación las incorpora al grupo de réplicas disponibles dentro de ella. La operativa de cara a los clientes sigue siendo idéntica excepto por la utilización de un nombre de dominio en lugar de direcciones IP.

3. Servicio clave-valor replicado basado en Raft

3.1. Metodología

Se ha hecho uso del recurso de Kubernetes **StatefulSet** para la aplicación *kvdb* partiendo de 3 réplicas.

La mayor dificultad del fichero de configuración reside en el apartado `command` de la especificación del Pod. Se han seguido los siguientes pasos:

1. Obtener el número (ordinal) del pod aplicando una expresión regular.
2. Solicitar a cada nodo Raft el líder del servicio replicado. Para ello se, iterará sobre un número N de réplicas (si no existe, *netcat* devolverá un error).
3. Si se obtiene el líder, se deja de iterar y se ejecuta el comando incluyendo su dirección para añadir el pod al cluster.
4. Si han terminado las N iteraciones y no se ha obtenido respuesta, se creará un nuevo *cluster* sin incluir la opción `-j`.

El contenedor base para crear los pods de la aplicación está basado en la imagen de contenedor *alpine*. Como herramientas cliente de la aplicación se utiliza *netcat* para conectar con los distintos nodos Raft y ejecutar determinados

comandos que sirven para comprobar el correcto funcionamiento. Esos comandos se encuentran especificados tanto en el `README.md` del *framework* como en el de la aplicación ejemplo *kvdb*.

3.2. Resultados

Se han realizado pruebas de incremento y decremento del número de réplicas Raft, mediante Kubernetes:

```
$ kubectl scale statefulsets raft --replicas=<nº-replicas>
```

Se ha comprobado que la aplicación Golang las detecta y las incorpora al grupo de nodos Raft de forma correcta. También se ha comprobado la recuperación correcta de fallos de nodos participantes eliminando directamente los Pods mediante:

```
$ kubectl delete pod <nombre pod>
```

, e incorporando nuevas réplicas creadas automáticamente por Kubernetes (StatefulSet) en la aplicación clave/valor basada en Raft.

4. Servicio distribuido y tolerante a fallos de semáforos

4.1. Metodología

Se ha inspirado en los 3 ejemplos de la aplicación del framework Raft *Uhaha* que se incluye en el repositorio para implementar un servicio distribuido y tolerante a fallos de mutexes que garantiza exclusión mutua entre procesos distribuidos y que utiliza Raft para la tolerancia a fallos.

Para ello, se han definido dos operaciones básicas: **wait(semaforo)** y **signal(semaforo)**. Estas son las operaciones que tiene disponibles un cliente del servicio. Al realizar wait(semaforo), el método devuelve un canal sobre el que quedarse bloqueado, a la espera de una llamada TCP que lo despierte. Cuando se realice una operación signal(semaforo), el servidor avisará a un proceso de su cola FIFO para que se desbloquee.

4.2. Resultados

Debido a la implentación de Raft, durante las primeras pruebas realizadas, se obtuvieron unos resultados extraños, debido a que todos los nodos intentaban aplicar sobre su máquina de estados las funciones programadas, pero debido a que dichas funciones implicaban trabajar síncronamente con los clientes, se producían errores críticos como fallos en llamadas TCP a un cliente que ya no estaba escuchando o envíos múltiples de señales de desbloqueo.

Estos se solucionaron despertando únicamente con la primera señal que llegase e ignorando las demás.

Una vez resueltos estos contratiempos, el sistema funciona correctamente incluso ante caídas de un servidor. Tiene un problema asociado al concepto de semáforo, y es que no es resistente ante un fallo del cliente, ya que nunca quedaría liberado el semáforo. Esto se podría solucionar permitiendo accesos temporizados a la sección crítica o comprobando periódicamente que el cliente que lo está utilizando sigue vivo.

5. Conclusiones

Tras la realización de esta práctica, se ha observado que el despliegue en un servicio como Kubernetes de aplicaciones distribuidas es un paso muy natural y relativamente sencillo aún a pesar de la gran complejidad que implica manejar correctamente esta operativa. Kubernetes permite escalar horizontalmente los servicios de una manera muy sencilla, llegando a permitir automatizar el proceso de escalado en función a los recursos disponibles.

Si se compara esto con la operativa en anteriores prácticas en las que se debía realizar la puesta en marcha (manualmente o mediante un script) por ssh o implementar el controlador de recursos, se observa la gran capacidad de Kubernetes como *sistema operativo distribuido*, ya que proporciona una capacidad similar a las llamadas al sistema

Referencias

[1] Josh Baker. *Uhaha: High Availability Framework for Happy Data*. 2020. URL: https://github.com/tidwall/uhaha.

[2] Diego Ongaro y John Ousterhout. "In Search of an Understandable Consensus Algorithm". En: (2013). URL: https://raft.github.io/raft.pdf.