



**Escuela de
Ingeniería y Arquitectura**
Universidad Zaragoza

Tp6-2

Despliegue de raft en kubernetes

30221 - Sistemas Distribuidos

Aarón Ibáñez Espés, 779088
Ángel Espinosa Gonzalo 775750

Índice

1. Resumen	2
2. Puesta en marcha de Kubernetes	2
3. Despliegue en kubernetes	2
4. Cambios realizados en el código	3
5. Validación Experimental	4
5.1. Test de funcionamiento	4
5.1.1. soloArranqueYparadaTest1	4
5.1.2. elegirPrimerLiderTest2	4
5.1.3. falloAnteriorElegirNuevoLiderTest3	4
5.1.4. tresOperacionesComprometidasEstable	5
5.2. Test de replicación y concurrencia	5
5.2.1. AcuerdoAPesarDeDesconexionesDeSeguidor	5
5.2.2. SinAcuerdoPorFallos	5
5.2.3. SometerConcurrentementeOperaciones	6
6. Conclusiones	6
7. Anexo	7

1. Resumen

Este documento se presenta una memoria del trabajo tp6-2 de la asignatura. En el se ha desplegado el sistema de almacenamiento clave/valor basado en Raft desarrollado durante las practicas 4 y 5 en Kubernetes.

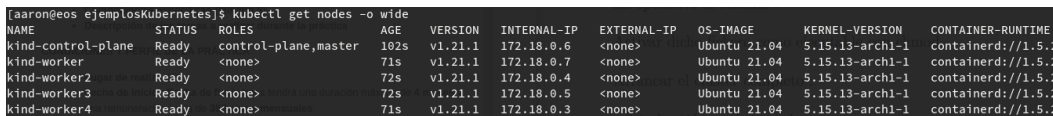
En este trabajo se ha puesto en marcha un cluster de kubernetes de de 4 nodos, en un solo ordenador mediante técnicas de contenedores anidados dentro de contenedores, utilizando la herramienta kind sobre Docker.

2. Puesta en marcha de Kubernetes

Inicialmente se ha instalado Docker en el ordenador. Se ha optado por una configuración de funcionamiento para Kubernetes de contenedor sobre contenedor que en lugar utilizar máquinas virtuales o físicas utiliza contenedores para albergar los nodos de Kubernetes. Además, se han incluido las herramientas kind y kubectl en el PATH del equipo, las cuales se utilizan para la creación y gestión del cluster de kubernetes.

Una vez se ha completado la configuración inicial se lanza el script *kind-with-registry.sh*, el cual se encarga de crear el registro local y poner en marcha 5 nodos de Kubernetes, un master y 4 workers, cada uno en un contenedor de docker.

Tras esto se comprueba con el comando *kubectl get nodes -o wide*, que se han creado correctamente.



NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION	CONTAINER-RUNTIME
kind-control-plane	Ready	control-plane,master	102s	v1.21.1	172.18.0.6	<none>	Ubuntu 21.04	5.15.13-arch1-1	containerd://1.5.2
kind-worker	Ready	<none>	71s	v1.21.1	172.18.0.7	<none>	Ubuntu 21.04	5.15.13-arch1-1	containerd://1.5.2
kind-worker2	Ready	<none>	72s	v1.21.1	172.18.0.4	<none>	Ubuntu 21.04	5.15.13-arch1-1	containerd://1.5.2
kind-worker3	Ready	<none>	72s	v1.21.1	172.18.0.5	<none>	Ubuntu 21.04	5.15.13-arch1-1	containerd://1.5.2
kind-worker4	Ready	<none>	71s	v1.21.1	172.18.0.3	<none>	Ubuntu 21.04	5.15.13-arch1-1	containerd://1.5.2

Figura 1: Nodos de kind

3. Despliegue en kubernetes

Para la ejecución de la aplicación Golang en Kubernetes se ha utilizado como imagen base de los contenedores de Docker una imagen mínima de Linux, Alpine en el caso de los test y una imagen scratch para los nodos de Raft.

Para desplegar el proyecto se ha utilizado un controlador *Statefulset*, detrás de un *Headless Service* encargado de la identidad de red de los pods, con el *Headless Service* no hay balanceador de carga y no se asigna una ip al cluster, *cluster-IP*. Con respecto al servicio DNS el selector del service modifica la configuración DNS para devolver registros A (direcciones IP) que apuntan directamente a los Pods que respaldan el Service.

La decision de utilizar un controlador Statefulset se ha tomado debido a que se tiene un nombre DNS conocido, lo que proporciona identificadores de red estables y únicos para los pods. Cada pod tiene su propio identificador persistente que mantiene a lo largo de cualquier re-programación. Esto es indispensable para la identificación de los pods en caso de caídas. Además, este controlador permite un despliegue y escalado ordenado y controlado y almacenamiento estable y persistente. Para el despliegue del pod donde se han lanzado los test, se ha optado por el uso del mismo controlador bajo el mismo *Headless Service*.

Como paso previo al despliegue, se han compilado los códigos de raft y de los test para generar un ejecutable utilizando el flag de compilación *CGO_ENABLED=0*, el cual compila estáticamente los programas y evita que se necesiten librerías dinámicas. En el caso de los test el comando de compilacion es ligeramente diferente al no tratarse de un modulo main, este es *CGO_ENABLED=0 go test -c*.

Para el despliegue se ha requerido de los siguientes ficheros:

- **Dockerfile y buildPushRaft.sh:** El fichero Dockerfile se utiliza para compilar la imagen de docker que almacena un nodo de Raft. Además, se cuenta con el script *buildPushRaft.sh* el cual automatiza el proceso de construcción y posterior subida de la imagen al registro local.
- **Dockerfile y buildPushTest.sh:** Al igual que en el punto anterior, el fichero Dockerfile se utiliza para compilar la imagen de docker que contenga los test y el script *buildPushTest.sh* para automatizar el proceso de construcción y push al registro local.
- **raft.test.stateful.yaml y deployRaft.sh** En este fichero de formato yaml se han desplegado dos *Statefulsets* bajo el mismo *Headless Service*, uno para cada uno de los códigos mencionados anteriormente. Este proceso se ha automatizado en el script *deployRaft.sh*.

El contenido de los ficheros mencionados se encuentra disponible en el Anexo.

Tras finalizar el proceso de despliegue, se comprueba con el comando *kubectl get pods -o wide* que todos los pods se han lanzado correctamente y que cada uno de ellos ha sido asignado a un nodo diferente del cluster de kind.

```
[aaron@eos ejemplosKubernetes]$ ./deployRaft.sh
service "raftdns" deleted
statefulset.apps "nodoraft" deleted
statefulset.apps "rafttests" deleted
service/raftdns created
statefulset.apps/nodoraft created
statefulset.apps/rafttests created
[aaron@eos ejemplosKubernetes]$ kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP            NODE          NOMINATED NODE   READINESS GATES
nodoraft-0    1/1     Running   0           36s   10.244.4.3    kind-worker   <none>            <none>
nodoraft-1    1/1     Running   0           36s   10.244.1.3    kind-worker3   <none>            <none>
nodoraft-2    1/1     Running   0           36s   10.244.2.3    kind-worker2   <none>            <none>
rafttests-0   1/1     Running   0           15s   10.244.3.3    kind-worker4   <none>            <none>
[aaron@eos ejemplosKubernetes]$
```

Figura 2: Descripción de los nodos

4. Cambios realizados en el código

Para adaptar el sistema de almacenamiento clave/valor finalizado en la practica 5, se han tenido que modificar ciertos aspectos de Raft y de los test.

- En el main de Raft se mantiene el paso por parámetro de las direcciones de los nodos, sin embargo estas pasan a ser el nombre DNS de cada uno en vez de su dirección ip, ya que esta es dinámica y cambia cada vez que se relanza el nodo. Así mismo, en vez de pasar como parámetro el numero de replica, se pasa el nombre de la misma, el cual se procesa y se obtiene el índice de nodo.
- Por otra parte, se ha eliminado el despliegue de Raft por ssh ya que al utilizar Kubernetes, los nodos ya se encuentran levantados, y en caso de caída, estos se levantan automáticamente, por lo que han desaparecido las funciones *startDistributedProcesses* y *startNodeDistributed*.

5. Validación Experimental

En este apartado se pone a prueba el despliegue realizado anteriormente con las pruebas que se habían implementado para las practicas 4 y 5. Para ello se lanzan los 3 nodos de Raft y además, un cuarto pod con los test previamente comentados.

Para comenzar las pruebas se accede al nodo donde se encuentran almacenadas con el comando `kubectll exec rafttests-0 -ti - sh` y se lanza el ejecutable `testintegracionraft1.test`.

Se ha tomado la decisión de no lanzar las pruebas directamente al crear el nodo ya que sino en cuanto estas acabasen el nodo se caería y se volvería a levantar, estando en una continua ejecución de las pruebas.

5.1. Test de funcionamiento

En este conjunto inicial de test se evalúa que las funciones básicas de raft funcionan correctamente. Se han hecho pruebas para comprobar el estado inicial de los nodos, la elección y reelección de líder en caso de caída y pruebas de someter operaciones.

5.1.1. soloArranqueYparadaTest1

Este es el primer test, el mas sencillo y en el simplemente se evalúa que los nodos se han lanzado correctamente.

```
/ # testintegracionraft1.test
TestPrimerasPruebas/T1:soloArranqueYparada .....
..... TestPrimerasPruebas/T1:soloArranqueYparada Superado
```

Figura 3: Resultado del test

5.1.2. elegirPrimerLiderTest2

En este test se comprueba que tras iniciar los nodos Raft, se ha elegido solamente un líder. Para ello se comprueba el estado de cada uno de los nodos y se cuenta el numero de nodos que han contestado líder.

```
TestPrimerasPruebas/T2:ElegirPrimerLider .....
Probando lider en curso
..... TestPrimerasPruebas/T2:ElegirPrimerLider Superado
```

Figura 4: Resultado del test

5.1.3. falloAnteriorElegirNuevoLiderTest3

En este test se comprueba que cuando se ha caído el líder, se realizan elecciones y un nuevo líder toma el relevo. Para ello se lanzan la tres replicas de Raft y se comprueba que en el primer mandato hay solamente un líder. Si es así, se desconecta el nodo líder y se comprueba que se ha iniciado un nuevo proceso de elecciones y se ha encontrado otro líder.

```
TestPrimerasPruebas/T2:ElegirPrimerLider .....
Probando lider en curso
..... TestPrimerasPruebas/T2:ElegirPrimerLider Superado
```

Figura 5: Resultado del test

5.1.4. tresOperacionesComprometidasEstable

En este test se evalúa que se consiguen comprometer 3 operaciones seguidas en 3 entradas diferentes en un entorno distribuido sin fallos y con un líder estable.

```
TestPrimerasPruebas/T2:ElegirPrimerLider .....  
Probando líder en curso  
..... TestPrimerasPruebas/T2:ElegirPrimerLider Superado
```

Figura 6: Resultado del test

5.2. Test de replicación y concurrencia

Tras haber probado que los nodos funcionan correctamente, que se inicia un proceso de elección y se elige un nuevo líder en caso de caída del líder y que en un entorno distribuido sin fallos se comprometen operaciones de forma correcta, se han implementado las siguientes pruebas, en las que evalúa el correcto funcionamiento del algoritmo en un entorno distribuido con fallos de nodos y concurrencia a la hora de someter operaciones.

5.2.1. AcuerdoAPesarDeDesconexionesDeSeguidor

En este test se evalúa que se consigue acuerdos de varias entradas de registro a pesar de que un replica se desconecta del grupo, para ello, se comprometen 3 entradas con un nodo caído, tras esto se levanta el tercer nodo y se comprometen mas entradas. Esto permite comprobar que la replicación del log del líder se realiza de forma correcta en el nodo que se había caído, y que además de guardar las nuevas entradas, replica las anteriores de manera exitosa y en el orden correcto.

```
TestAcuerdosConFallos/T5:AcuerdoAPesarDeDesconexionesDeSeguidor_ .....  
Desconecto a: 0  
BIEN: 1  
BIEN: 2  
Reconectando 0  
BIEN: 0  
BIEN: 1  
BIEN: 2  
..... TestAcuerdosConFallos/T5:AcuerdoAPesarDeDesconexionesDeSeguidor_ Superado
```

Figura 7: Resultado del test

Como se puede ver en la Figura 7, la ejecución del test sigue los siguientes pasos

1. Se levantan las tres replicas de nodo Raft.
2. Se desconecta uno de los nodos seguidores, en este caso en nodo con id 0.
3. Se someten 3 operaciones con el nodo 0 desconectado y se comprueba que se han sometido correctamente.
4. Se reconecta el nodo 0 y se someten 3 operaciones más.
5. Finalmente se comprueba que las entradas se han comprometido las entradas y el log de todos los nodos esta correctamente replicado.

5.2.2. SinAcuerdoPorFallos

En este test se comprueba que no se consigue acuerdo de varias entradas al desconectarse 2 de los 3 nodos de Raft. Para ello, se lanzan las 3 replicas e inmediatamente se desconectan las dos en estado seguidor. Una vez se han desconectado, se envían unas peticiones para someter operación y se comprueba si estas han sido comprometidas. Tras esto se levantan las dos réplicas previamente desconectadas y nuevamente se intentan comprometer nuevas entradas, esta vez de forma correcta ya que hay mayoría de nodos levantados.

```

TestAcuerdosConFallos/T5:SinAcuerdoPorFallos_ .....
Desconecto a: 0      Se reconecta el nodo 0 y se someten 3 operaciones más.
Desconecto a: 2
El nodo 1 no tiene todas las entradas comprometidas no comprometido las
Reconectando 0      los nodos esta correctamente replicado.
Reconectando 2
BIEN: 0      5.2.2. SinAcuerdoPorFallos
BIEN: 1      En este test se comprueba que no se consigue acuerdo de varias entra
BIEN: 2      los 3 nodos de Raft. Para ello se hacen las 3 replicas e inmediatamente
..... TestAcuerdosConFallos/T5:SinAcuerdoPorFallos_ Superado

```

Figura 8: Resultado del test

Como se puede ver en la Figura 8, la ejecución del test sigue los siguientes pasos

1. Se levantan las tres replicas de nodo Raft.
2. Se desconectan los dos nodos seguidores, los nodos 0 y 2.
3. Se someten 3 operaciones con los nodos desconectados y se comprueba que no se ha podido comprometer la entrada.
4. Se reconectan los nodos desconectados y se someten 3 operaciones más.
5. Finalmente se comprueba que tanto las entradas pendientes como las nuevas han sido correctamente replicadas y comprometidas.

5.2.3. SometerConcurrentementeOperaciones

En este test se comprueba que se consiguen someter 5 operaciones cliente de forma concurrente y comprobar avance de índice.

```

TestAcuerdosConFallos/T5:SometerConcurrentementeOperaciones_ .....
BIEN: 0
BIEN: 1
BIEN: 2
..... TestAcuerdosConFallos/T5:SometerConcurrentementeOperaciones_ Superado
PASS

```

Figura 9: Resultado del test

Como se puede ver en la Figura 9, la ejecución del test sigue los siguientes pasos

1. Se levantan las tres replicas de nodo Raft.
2. Se someten 5 operaciones de forma concurrente.
3. Finalmente se comprueba que todas las entradas han sido correctamente replicadas y comprometidas.

Para la validación de las entradas comprometidas en un nodo se cuenta con la función *checkCommits*, la cual comprueba el numero de entradas comprometidas de cada nodo. Para ello utiliza el valor de *CommitIndex*, el cual indica el índice de la ultima entrada comprometida en el log.

6. Conclusiones

En este trabajo se ha comprobado como el servicio de almacenamiento clave/valor implementado en las practicas anteriores se puede desplegar en Kubernetes. Además, gracias a los controladores de Kubernetes y a Raft, se ha comprobado como el sistema es capaz de gestionar los fallos y mantener la consistencia de los datos en todo momento y mantener el almacen disponible.

7. Anexo

- Dockerfile de Raft

```
FROM alpine  
COPY main /usr/local/bin/nodoraft  
EXPOSE 6000
```

Figura 10: Dockerfile

- Script Build And Push de Raft

```
docker build . -t localhost:5000/raft:latest  
docker push localhost:5000/raft:latest
```

Figura 11: script de Raft

- Dockerfile de los test

```
FROM alpine  
COPY testintegracionraft1.test /usr/local/bin/testintegracionraft1.test  
EXPOSE 6000
```

Figura 12: Dockerfile

- Script Build And Push de los test

```
docker build . -t localhost:5000/test:latest  
docker push localhost:5000/test:latest
```

Figura 13: script de los test

- raft_test_stateful.yaml

```
docker build . -t localhost:5000/test:latest  
docker push localhost:5000/test:latest
```

Figura 14: Dockerfile

■ deployRaft.sh

```
apiVersion: v1
kind: Service
metadata:
  name: raftdns
  labels:
    app: rep
spec:
  clusterIP: None
  selector:      # tiene que coincidir con label definido en pod de StatefulSet
    app: rep    # Para dar de alta automaticamente en DNS a los PODS ligados
  ports:
    - port: 6000
      name: servidor-port
      protocol: TCP
      targetPort: 6000
---
kind: StatefulSet
apiVersion: apps/v1
metadata:
  name: nodoraft
spec:
  serviceName: raftdns
  replicas: 3
  podManagementPolicy: Parallel    # por defecto seria OrderedReady (secuencial)
  selector:
    matchLabels:
      app: rep    # tiene que corresponder a .spec.template.metadata.labels
  template:
    metadata:
      labels:
        app: rep
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: raft
          image: localhost:5000/raft:latest
          env:
            - name: MISUBDOMINIODNS
              value: raftdns.default.svc.cluster.local
            - name: MINOMBREPOD    # primera replica r-0, segunda r-1, etc
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
          command:
            - nodoraft
            - $(MINOMBREPOD)
            - nodoraft-0.raftdns.default.svc.cluster.local:6000
            - nodoraft-1.raftdns.default.svc.cluster.local:6000
            - nodoraft-2.raftdns.default.svc.cluster.local:6000
          ports:
            - containerPort: 6000
```

```

---
kind: StatefulSet
apiVersion: apps/v1
metadata:
  name: rafttests
spec:
  serviceName: raftdns
  replicas: 1
  podManagementPolicy: Parallel # por defecto seria OrderedReady (secuencial)
  selector:
    matchLabels:
      app: rep # tiene que corresponder a .spec.template.metadata.labels
  template:
    metadata:
      labels:
        app: rep
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: test
        image: localhost:5000/test:latest
        env:
          - name: MISUBDOMINIODNS
            value: raftdns.default.svc.cluster.local
          - name: MINOMBREPOD # primera replica r-0, segunda r-1, etc
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
        command:
          - sleep
          - "3600"
        ports:
          - containerPort: 6000

```

Figura 15: Fichero yaml para el despliegue