# Entrega Práctica 5 - Sistemas distribuidos

**Kubernetes y Raft**

**Autores:**

Fernando Pastor Peralta (897113)

Javier Murillo Jimenez (897154)

**Fecha de entrega:** 27/11/2025

**Grupo:** Pareja:1-6

# Índice

# Introducción:

El objetivo de esta práctica es desplegar el servicio de almacenamiento distribuido clave/valor basado en el protocolo de Raft diseñado en la práctica anterior, dentro de un cluster de Kubernetes, garantizando que cada réplica se ejecuta en un nodo worker diferente y que, ante la caída de una réplica o un nodo completo del clúster, Kubernetes la rearma automáticamente y Raft es capaz de restaurar su estado previo sin ninguna pérdida de datos.

Para ello hemos tenido que adaptar el código de la práctica anterior para permitir la conexión mediante DNS en el servicio de kubernetes en un entorno de contenedores.

# Diseño de nuestra solución:

## Adaptación del servidor Raft a Kubernetes

Para adaptar el código de la práctica anterior, hemos hecho los cambios necesarios para que los nodos se conecten correctamente al servicio DNS que hemos creado (en svraft_go.yaml), en el cliente poniendo explícitamente la dirección DNS de cada uno de los servidores raft, y en los servidores raft comprobando que el parámetro dado es un nombre de servidor esperado, construyendo su dirección y la de los demás conociendo de antemano el subdominio de DNS y el número de nodos del sistema.

Después, hemos compilado el código .go del servidor y del cliente, y hemos obtenido sus ejecutables en la carpeta raft/.

Posteriormente, hemos creado un Dockerfile para crear las imágenes de cada uno, y los ponemos accesibles en un registro local (localhost:5001) de contenedores de Docker para que puedan ser obtenidos por Kubernetes (el registro y kind fueron creados mediante el fichero shell proporcionado “kind-with-registry.sh”). En estos Dockerfile montamos la distribución Alpine Linux que correrá sobre docker y decidimos ficheros donde el contenedor arrancará y trabajará.

```shell
#build crea la imagen docker y push la mete en el registro local
   #---Imagen del servidor Raft---
   docker build -t localhost:5001/srvraft -f Dockerfile.srvraft .
   docker push localhost:5001/srvraft
```

```
#---Imagen del cliente Raft ---
docker build -t localhost:5001/cltraft -f Dockerfile.cltraft .
docker push localhost:5001/cltraft
```

Los Dockerfiles y .yaml estarán disponibles en el anexo.

## Ejecución de cada tipo de nodo en su Pod específico

Esta segunda parte se puede ver reflejada en los .yaml, donde aplicaremos estos manifiestos de configuración para kubernetes. Para el cliente (cltraft_go.yaml) hemos decido usar un Pod básico ya que no requiere tolerancia a fallos, además nos conectamos directamente como clientes, tomando automáticamente la imagen del cliente del registro local. En cambio en las réplicas Raft (srvraft_go.yaml), hemos usado un StatefulSet ya que necesitamos una identidad estable y única para cada nodo, un DNS fijo (servicio en puerto 6000) y tolerancia a fallos. Los siguientes apply lanzan directamente los diferentes pods:

```shell
kubectl apply -f srvraft_go.yaml
kubectl apply -f cltraft_go.yaml
```

## Estructura del sistema desplegado en Kubernetes

Nuestra solución se organiza en tres componentes:

- 3 réplicas Raft del servidor de almacenamiento, ejecutadas mediante un StatefulSet.
- Un Service headless para DNS.
- Un pod que ejecuta el cliente.

El StatefulSet nos proporciona nombres DNS estables para cada réplica, lo cual es necesario para que cada nodo pueda construir las direcciones RPC de los otros. El Service headless permite que Kubernetes registre cada Pod en el DNS sin la necesidad de asociar una IP virtual.

## Contenedores Docker

Para aislar los ejecutables y permitir su despliegue en Kubernetes, hemos generado dos imágenes Docker independientes:

- Imagen del servidor Raft, construida a partir de Alpine Linux y conteniendo únicamente el ejecutable “srvraft”.
- Imagen del cliente automático, también basada en Alpine, que ejecuta el binario “cltraft”.

Ambas imágenes, encontradas en el anexo, se han construido estaticamente a partir del código Golang (compilado con “CGO_ENABLED = 0”) y se han subido al registro local “localhost:5001” que Kubernetes usa como repositorio de contenedores. Esto garantiza que los nodos del clúster puedan descargar las imágenes sin necesidad de repositorios externos.

# Manifiestos Kubernetes

Hemos diseñado los siguientes manifiestos:

1. Pod del cliente

Ejecuta el binario “cltraft” con política “Never” para que Kubernetes no lo vuelva a crear en caso de que este muera.

2. Service headless

Define el servicio “raft-service”, necesario para que Kubernetes registre las réplicas Raft en su DNS interno. Ponemos “clusterIp: None” para que Kubernetes no le asigne una IP virtual y en cambio cada uno obtiene un nombre DNS propio y estable.

3. StatefulSet de las réplicas

El StatefulSet crea tres réplicas del servidor Raft “srvraft”, cada una ejecutándose en un contenedor diferente. Mediante la variable de entorno “MINOMBREPOD”, cada nodo obtiene su nombre dentro del clúster y, a partir de él, calcula las direcciones del resto de las réplicas.

Los manifiestos están incluidos en el anexo.

# Pruebas de validación

Para las pruebas de validación creamos un script en bash que se basa en mirar si los logs del cliente indican que las operaciones han sido exitosas. El script realiza las siguientes pruebas:

1. Despliegue y operativa normal: tras limpiar el entorno levantamos servidores y cliente, ese cliente hace 4 lecturas y 4 escrituras (una de ellas sobre misma clave para confirmar que se cambia correctamente el valor), si todo sale bien el cliente imprime en uno de sus strings “ÉXITO”, cuando esto sucede el script en bash lo detecta y da la primera prueba como correcta.

2. Matamos al Pod del nodo raft líder: tras analizar los log para identificar el nodo líder, lo eliminamos forzosamente para verificar que Kubernetes lo regenerara y que Raft es capaz de elegir un nuevo líder y seguir operando correctamente con el cliente.

3. Prueba de tolerancia a fallos de un nodo (caída de un Docker): En bash identificamos el nodo (contenedor Docker) que está ejecutando el nuevo líder y lo detenemos (simulando un corte de red) para verificar que el sistema continúa funcionando correctamente con la mayoría restante. Luego restauramos el nodo.

# Anexo

## A. Dockerfiles

### a. Dockerfile.cltraft

```
None
#Montamos la distribución Alpine Linux en nuestra imagen.

FROM alpine

#A partir de ahora, todo lo que se haga irá a /usr/local/bin.

WORKDIR /usr/local/bin

#La aplicación dentro del contenerdor escucha el puerto 6000.

EXPOSE 6000

#Copiamos el ejecutable del servidor Raft.

COPY srvraft /usr/local/bin/srvraft

#Arrancar el servidor.

ENTRYPOINT ["/usr/local/bin/srvraft"]
```

### b. Dockerfile.srvraft

```
None
#Montamos la distribución Alpine Linux en nuestra imagen.

FROM alpine

#A partir de ahora, todo lo que se haga irá a /usr/local/bin.

WORKDIR /usr/local/bin

#Copiamos el ejecutable del servidor Raft.

COPY cltraft /usr/local/bin/cltraft

#Arrancar el servidor.

ENTRYPOINT ["/usr/local/bin/cltraft"]
```

## B. Yaml

### a. cltraft_go.yaml (Pod del cliente)

```
None
#Cliente automático, conectado al servicio Raft
apiVersion: v1
kind: Pod
metadata:
  name: clientraft
spec:
  restartPolicy: Never
  containers:
  - name: cltraft
    image: localhost:5001/cltraft:latest
    ports:
    - containerPort: 7000
```

### b. srvraft_go.yaml (Service headless y StatefulSet)

```
None
#Service "headless" para que Kubernetes dé de alta los Pods en el DNS.
apiVersion: v1
kind: Service
metadata:
  name: raft-service
  labels:
    app: raft
spec:
  clusterIP: None
  selector: #tiene que coincidir con label definido en los Pods.
```

```
    app: raft
  ports:
  - port: 6000
    name: raft-port
    protocol: TCP
    targetPort: 6000
---
#StatefulSet de réplicas de Raft
kind: StatefulSet
apiVersion: apps/v1
metadata:
  name: raft
spec:
  serviceName: raft-service
  replicas: 3
  podManagementPolicy: Parallel
  selector:
    matchLabels:
      app: raft          # tiene que corresponder a
.spec.template.metadata.labels
  template:
    metadata:
      labels:
        app: raft
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: srvraft
```

```
        image: localhost:5001/srvraft:latest
        env:
        - name: MINOMBREPOD  #primera réplica raft-0, segunda raft-1, etc...
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        command:
        - /usr/local/bin/srvraft
        - $(MINOMBREPOD)
        ports:
        - containerPort: 6000
```

## C. Ejecución y comandos utilizados

### a. Construcción y subida de imágenes

Shell

```
# Compilamos los ejecutables del cliente y el servidor
CGO_ENABLED=0 go build -o cltraft pkg/cltraft/main.go
CGO_ENABLED=0 go build -o srvraft cmd/srvraft/main.go
# Subimos las imágenes al kind-registry
#---Imagen del servidor Raft---
docker build -t localhost:5001/srvraft -f Dockerfile.srvraft .
docker push localhost:5001/srvraft
#---Imagen del cliente Raft ---
docker build -t localhost:5001/cltraft -f Dockerfile.cltraft .
docker push localhost:5001/cltraft
```

## b. Puesta en marcha del sistema

Shell

```
# Aplicamos los manifiestos de configuración de Kubernetes

kubectl apply -f srvraft_go.yaml

kubectl apply -f cltraft_go.yaml
```

## c. Pruebas

Shell

```
bash test.bash

# A continuación el test.bash

#!/bin/bash

# Configuración

CLIENTE="cltraft_go.yaml"

SERVIDOR="srvraft_go.yaml"

POD_CLI="clientraft"

# Función para ejecutar cliente y validar éxito.

verificar_cliente() {

    kubectl delete pod $POD_CLI --grace-period=0 --force 2>/dev/null

    kubectl apply -f $CLIENTE

    echo "   ...Esperando ejecución del cliente..."

    #El script se pausa aquí hasta que el pod acaba su trabajo (estado
Succeeded o Failed).

    while true; do

        #Obtenemos el estado actual (Running, Pending, Succeeded,
Failed...).

        PHASE=$(kubectl get pod $POD_CLI -o jsonpath='{.status.phase}'
2>/dev/null)

        # Si ya acabó (bien o mal), salimos del bucle.

        if [[ "$PHASE" == "Succeeded" || "$PHASE" == "Failed" ]]; then

            break
```

```
        fi
        #Si sigue corriendo, esperamos un poco.
        sleep 2
    done
    LOGS=$(kubectl logs $POD_CLI)
    if echo "$LOGS" | grep -q "ÉXITO"; then
        echo "   -> OK: Prueba superada.\n"
        echo "---------------------------------"
        kubectl get pods -o wide
        echo "---------------------------------"
        echo "$LOGS"
        echo "---------------------------------"
    else
        echo "   -> ERROR: El cliente falló."
        echo "$LOGS"
        exit 1
    fi
}
echo "=== INICIO TEST RAFT EN KUBERNETES ==="
echo "[1] Desplegando Servidores..."
kubectl delete pod --all --grace-period=0 --force 2>/dev/null
kubectl apply -f $SERVIDOR
echo "   Esperando a que los pods estén preparados..."
sleep 5
echo "[2] Prueba Básica (Escritura/Lectura)"
verificar_cliente
# Guardamos logs para buscar al líder
```

```
LOGS_PREVIOS=$(kubectl logs $POD_CLI)

#Buscamos el id del lider a matar en los logs de la anterior ejecución.

LIDER_ID=$(echo "$LOGS_PREVIOS" | grep -o 'lider=[0-9]*' | tail -1 | cut -d=
-f2)

echo "[3] Borrando Pod Líder (raft-$LIDER_ID)..."

kubectl delete pod raft-$LIDER_ID --grace-period=0 --force 2>/dev/null

echo "   Esperando regeneración..."

sleep 10

verificar_cliente # Comprobar que sigue funcionando

# Guardamos logs para buscar al líder

LOGS_PREVIOS=$(kubectl logs $POD_CLI)

#Buscamos el id del lider a matar en los logs de la anterior ejecución.

LIDER_ID=$(echo "$LOGS_PREVIOS" | grep -o 'lider=[0-9]*' | tail -1 | cut -d=
-f2)

NODO_LIDER=$(kubectl get pods -o wide | grep -E "raft-$LIDER_ID.*" | awk
'{print $7}' )

echo "[4] Parando contenedor Docker del nodo ($NODO_LIDER)..."

docker stop $NODO_LIDER

sleep 8

echo "   Probando con nodo caído..."

verificar_cliente

echo "[5] Restaurando nodo..."

docker start $NODO_LIDER

echo " "

kubectl get pods -o wide

echo "=== FIN ==="
```

### d. Eliminar todo

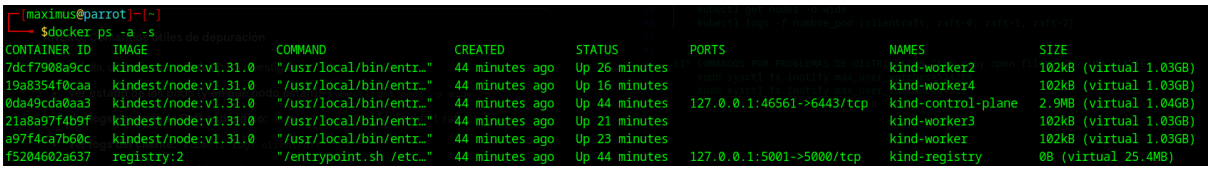
Shell

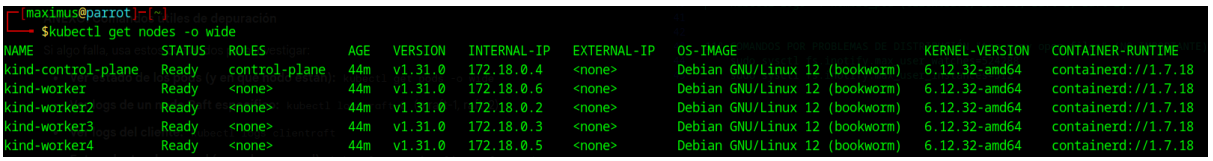
```
kind delete cluster

docker stop kind-registry

docker rm kind-registry
```

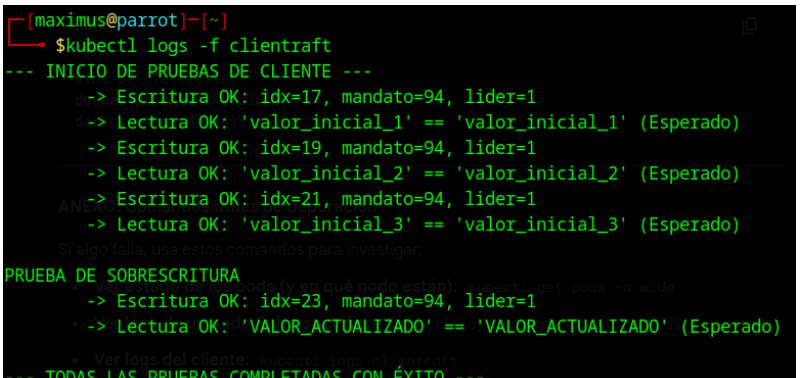
### e. Otros comandos

Shell

```
# Mostrar los Pods en ejecución

kubectl get pods -o wide

# Mostrar los contenedores Docker del sitema host

docker ps -a -s

# Mostrar todos los nodos del clúster Kubernetes

kubectl get nodes -o wide

# Mostrar logs en tiempo real

kubectl logs -f nombre_pod (clientraft, raft-0, raft-1, raft-2)
```

## D. La salida de nuestro script

Shell

```
 === INICIO TEST RAFT EN KUBERNETES ===
[1] Desplegando Servidores...
pod "clientraft" force deleted from default namespace
pod "raft-0" force deleted from default namespace
pod "raft-1" force deleted from default namespace
pod "raft-2" force deleted from default namespace
service/raft-service unchanged
statefulset.apps/raft unchanged
   Esperando a que los pods estén preparados...
[2] Prueba Básica (Escritura/Lectura)
pod/clientraft created
   ...Esperando ejecución del cliente...
   -> OK: Prueba superada.\n
---------------------------------
NAME         READY   STATUS      RESTARTS   AGE     IP            NODE
NOMINATED NODE   READINESS GATES
clientraft   0/1     Completed   0          2m18s   10.244.4.5
kind-worker2   <none>           <none>
raft-0       1/1     Running     0          2m24s   10.244.1.6
kind-worker    <none>           <none>
raft-1       1/1     Running     0          2m24s   10.244.3.3
kind-worker3   <none>           <none>
raft-2       1/1     Running     0          2m24s   10.244.2.14
kind-worker4   <none>           <none>
---------------------------------
--- INICIO DE PRUEBAS DE CLIENTE ---
       -> Escritura OK: idx=1, mandato=38, lider=1
```

```
       -> Lectura OK: 'valor_inicial_1' == 'valor_inicial_1' (Esperado)
       -> Escritura OK: idx=3, mandato=38, lider=1
       -> Lectura OK: 'valor_inicial_2' == 'valor_inicial_2' (Esperado)
       -> Escritura OK: idx=5, mandato=38, lider=1
       -> Lectura OK: 'valor_inicial_3' == 'valor_inicial_3' (Esperado)

PRUEBA DE SOBRESCRITURA
       -> Escritura OK: idx=7, mandato=38, lider=1
       -> Lectura OK: 'VALOR_ACTUALIZADO' == 'VALOR_ACTUALIZADO' (Esperado)

--- TODAS LAS PRUEBAS COMPLETADAS CON ÉXITO ---
---------------------------------
[3] Borrando Pod Líder (raft-1)...
pod "raft-1" force deleted from default namespace
   Esperando regeneración...
pod "clientraft" force deleted from default namespace
pod/clientraft created
   ...Esperando ejecución del cliente...
   -> OK: Prueba superada.\n
---------------------------------
NAME         READY   STATUS      RESTARTS   AGE     IP            NODE
NOMINATED NODE   READINESS GATES

clientraft   0/1     Completed   0          2m17s   10.244.3.4
kind-worker3   <none>           <none>

raft-0       1/1     Running     0          4m51s   10.244.1.6
kind-worker    <none>           <none>

raft-1       1/1     Running     0          2m27s   10.244.4.6
kind-worker2   <none>           <none>
```

```
raft-2       1/1     Running     0          4m51s   10.244.2.14
kind-worker4   <none>           <none>

---------------------------------

--- INICIO DE PRUEBAS DE CLIENTE ---

       -> Escritura OK: idx=9, mandato=93, lider=2

       -> Lectura OK: 'valor_inicial_1' == 'valor_inicial_1' (Esperado)

       -> Escritura OK: idx=11, mandato=93, lider=2

       -> Lectura OK: 'valor_inicial_2' == 'valor_inicial_2' (Esperado)

       -> Escritura OK: idx=13, mandato=93, lider=2

       -> Lectura OK: 'valor_inicial_3' == 'valor_inicial_3' (Esperado)


PRUEBA DE SOBRESCRITURA

       -> Escritura OK: idx=15, mandato=93, lider=2

       -> Lectura OK: 'VALOR_ACTUALIZADO' == 'VALOR_ACTUALIZADO' (Esperado)


--- TODAS LAS PRUEBAS COMPLETADAS CON ÉXITO ---

---------------------------------

[4] Parando contenedor Docker del nodo (kind-worker4)...

kind-worker4

   Probando con nodo caído...

pod "clientraft" force deleted from default namespace

pod/clientraft created

   ...Esperando ejecución del cliente...

   -> OK: Prueba superada.\n

---------------------------------

NAME         READY   STATUS      RESTARTS   AGE     IP            NODE
NOMINATED NODE   READINESS GATES

clientraft   0/1     Completed   0          5s      10.244.3.5
kind-worker3   <none>           <none>
```

```
raft-0       1/1     Running     0          5m4s    10.244.1.6
kind-worker    <none>           <none>

raft-1       1/1     Running     0          2m40s   10.244.4.6
kind-worker2   <none>           <none>

raft-2       1/1     Running     0          5m4s    10.244.2.14
kind-worker4   <none>           <none>

---------------------------------

--- INICIO DE PRUEBAS DE CLIENTE ---

       -> Escritura OK: idx=17, mandato=94, lider=1

       -> Lectura OK: 'valor_inicial_1' == 'valor_inicial_1' (Esperado)

       -> Escritura OK: idx=19, mandato=94, lider=1

       -> Lectura OK: 'valor_inicial_2' == 'valor_inicial_2' (Esperado)

       -> Escritura OK: idx=21, mandato=94, lider=1

       -> Lectura OK: 'valor_inicial_3' == 'valor_inicial_3' (Esperado)


PRUEBA DE SOBRESCRITURA

       -> Escritura OK: idx=23, mandato=94, lider=1

       -> Lectura OK: 'VALOR_ACTUALIZADO' == 'VALOR_ACTUALIZADO' (Esperado)


--- TODAS LAS PRUEBAS COMPLETADAS CON ÉXITO ---

---------------------------------

[5] Restaurando nodo...

kind-worker4


NAME         READY   STATUS      RESTARTS   AGE     IP            NODE
NOMINATED NODE   READINESS GATES

clientraft   0/1     Completed   0          5s      10.244.3.5
kind-worker3   <none>           <none>

raft-0       1/1     Running     0          5m4s    10.244.1.6
kind-worker    <none>           <none>
```

```
raft-1       1/1     Running     0          2m40s   10.244.4.6
kind-worker2   <none>           <none>

raft-2       1/1     Running     0          5m4s    10.244.2.14
kind-worker4   <none>           <none>

=== FIN ===
```