CURATOR/ZOOKEEPER

Desarrollo de Software Crítico

CRISTINA BARÓN SUÁREZ

ETSI INFORMÁTICA

Universidad de Málaga

ÍNDICE

ILUSTRACIONES

CONSTRUCCIÓN DEL SISTEMA

CLASE PRINCIPAL

Se ha creado una clase principal llamada Nodo. Esta dispone de 2 variables globales:

```
private static final String FILE = "medidas.txt";
public static final Path PATH_FILE = Paths.get("fichero/" + FILE);
```

Ilustración 1. Variables globales de la clase principal

- La ruta donde se encuentra dicho fichero.
- La variable de entorno para la conexión con ZooKeeper.

Una vez dentro del `main`, primero se debe de comprobar que dicho fichero del que hay que leer y en el que escribir existe. Por lo tanto, si no hay constancia de esto, se crea.

```
// Se necesita disponer de un fichero llamado "medidas.txt"
if (!Files.exists(PATH_FILE)) {
    // Crear el fichero si no existe
    try {
        Files.createFile(PATH_FILE);
    } catch (IOException e) { }
}
```

Ilustración 2. Creación del fichero "medidas.txt"

Posteriormente, pasamos a crear el cliente ZooKeeper con Curator. Para ello hemos creado la clase ElectionNode

```
ElectionNode nd = new ElectionNode();
nd.start();
```

Ilustración 3. Creación del ElectionNode

Dicha nueva clase tiene las variables globales:

- PATH_ZOOKEEPER: ruta en la que se encuentran los nodos.
- ZOOKEEPER_HOST: es una variable de entorno

Igualmente, se dispondrá del método `takeLeadership()` que será el que permita al líder leer, cada 5 segundos, las medidas que están insertando los seguidores en el fichero y enviarlas como petición a la API REST. Cómo se llevan a cabo estas funciones se verá en profundidad más adelante.

Además, mientras se esté en este método, el nodo será líder. Si se corta la conexión, se tardará medio minuto aproximadamente en seleccionar un nuevo líder que retoma la acción que estaba llevando a cabo el anterior. Para ello primero debe ser creada la conexión.

```
public class ElectionNode  extends LeaderSelectorListenerAdapter implements Closeable {

    LeaderSelector leaderSelector;

    private static final String PATH_ZOOKEEPER = "/zookeeper";
    public static final String ZOOKEEPER_HOST = System.getenv().getOrDefault("ZOOKEEPER_HOST", "localhost:2181");

    public ElectionNode() {
        CuratorFramework client = CuratorFrameworkFactory.newClient(ZOOKEEPER_HOST,
                new ExponentialBackoffRetry(1000, 3));
        client.start();

        leaderSelector = new LeaderSelector(client, PATH_ZOOKEEPER, this);
        leaderSelector.autoRequeue();
    }

    public void start() throws IOException {
        leaderSelector.start();
    }

    public void close() throws IOException {
        leaderSelector.close();
    }

    public boolean getImLeading() throws Exception {
        return leaderSelector.hasLeadership();
    }

    @Override
    public void takeLeadership(CuratorFramework client) throws Exception {
        while(true) {
            System.out.println("Soy lider");
            Operacion.lider();
            Thread.sleep(5000);
        }
    }
}
```

Ilustración 4. Clase ElectionNode

Continuando con el método main de la clase principal: según se sea líder o seguidor se realizará una acción u otra. Estas están recogidas en una nueva clase llamada Operación.

```
while(true) {
    if(!nd.getImLeading()) {     // seguidor
        System.out.println("Soy seguidor");
        Operacion.nodo();
        Thread.sleep(5000);
    }
}
```

Ilustración 5. Seguidor

En esta nueva clase se han creado otras 4 nuevas variables globales:

```
private static final int MIN = 60;
private static final int MAX = 199;
public static final String PRACTICA2_HOST = System.getenv().getOrDefault("PRACTICA2_HOST", "http://localhost:80");
private static final String URL_NUEVO = PRACTICA2_HOST + "/nuevo/";
```

Ilustración 6. Variables globales de la clase Operacion

- MIN y MAX representan los valores entre los que se encontrarán las medidas originadas por los seguidores.
- PRACTICA2_HOST: Variable de entorno a la que enviar peticiones HTTP.
- URL_NUEVO: URL con la que se desea establecer conexión y a la que el líder envía una petición. Coincide con la que se estableció en la práctica anterior.

SEGUIDOR

El código que permite al **seguidor** crear e introducir en el fichero "mediciones.txt" enteros aleatorios es el que sigue. Estas acciones se realizarán indefinidamente cada 5 segundos, o hasta que la hebra sea interrumpida.

```java
// Metodo para insertar una medida aleatorio en el fichero "medidas.txt"
public static void nodo() throws InterruptedException {
    Random rnd = new Random();
    // Crear valor aleatorio
    int medida = rnd.nextInt(MIN, MAX);
    System.out.println("Medida generada: " + medida);
    // Escribir en fichero
    insertarEnFichero(String.valueOf(medida));
}
```

Ilustración 7. Operaciones del seguidor

```java
// Inserta en el fichero la medida que se le pasa por parametro
private static void insertarEnFichero(String medida) {
    try(FileWriter fw = new FileWriter(String.valueOf(Nodo.PATH_FILE), true);
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter out = new PrintWriter(bw)) {
        out.println(medida);
    } catch (IOException e) {   }
}
```

Ilustración 8. Insertar medidas en el fichero "medidas.txt"

LÍDER

El código que permite al **líder** calcular la media de entre todas las medidas que los seguidores han originado hasta el momento es el que se muestra a continuación. Además, también enviará dicha media a la API REST creada en la práctica anterior. Estas acciones también se realizarán indefinidamente cada 5 segundos, o hasta que la hebra sea interrumpida.

```java
// Metodo para calcular la media de entre las medidas del fichero "medidas.txt"
public static void lider() throws IOException, InterruptedException {
    List<String> medidas = leerFichero();
    int sum = 0;
    for(String value : medidas) {
        sum += Integer.parseInt(value);
    }
    if (medidas.size() > 0) {
      int media = sum / medidas.size(); // int y no for porque es lo que espera la API REST
      System.out.println("Media de los valores: " + media);
      // Se envía la media como petición
      peticionHTTP(String.valueOf(media));
    }
}
```

Ilustración 9. Operaciones del líder

Para que el método anterior pueda adquirir las medidas que hay escritas en el fichero "medidas.txt", debe de leerlo. El siguiente método devuelve una lista de Strings en la que cada String es una línea de dicho fichero, es decir, un entero aleatorio.

```java
// Obtiene todas las medidas que hay en el fichero
private static List<String> leerFichero() {
    try {
        return Files.readAllLines(Nodo.PATH_FILE, StandardCharsets.UTF_8);
    } catch (IOException e) {
        System.out.println(e);
    }
    return new ArrayList<String>();
}
```

Ilustración 10. Leer medidas del fichero "medidas.txt"

Otro método que necesita el líder es el de `peticionHTTP()`. Este es el que abre la conexión y envía la petición con la medida que se le pasa por parámetro. Además, también se controla que la petición sea exitosa. En el caso de no serlo, notifica porqué puede haber sido.

```
private static void peticionHTTP(String medicion) throws IOException {
    // Se crea la URL
    URL url = new URL(URL_NUEVO + medicion);

    // Se abre la conexión con la URL
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();

    // El método de petición es GET
    conn.setRequestMethod("GET");

    // Chquear el resultado de la conexión
    int codigo = conn.getResponseCode();
    if(codigo == 500) {
        System.out.println("The server encountered an error while processing the request.");
    }else if(codigo == 404) {
        System.out.println("The requested resource was not found on the server.");
    }else if(codigo == 200){ // Peticion exitosa
        // Se lee la respuesta
        Reader r = new InputStreamReader(conn.getInputStream());
        BufferedReader rd = new BufferedReader(r);
        StringBuilder res = new StringBuilder();
        String linea;
        while ((linea = rd.readLine()) != null) {
            res.append(linea);
        }
        rd.close();

        // Imprimir la respuesta
        System.out.println(res);
    }else {
        System.out.println("Otro error de conexión.");
    }
}
```

Ilustración 11. Conexión y peticiones con la API REST de la práctica anterior

Además, se imprime el resultado de la página para ir comprobando en tiempo real si se ha producido alguna anomalía.

DESARROLLO DE STACK DOCKER

En primer lugar, se crea el Dockerfile:

```
FROM openjdk:17
WORKDIR /
ADD prPractica3.jar prPractica3.jar
# Crea el directorio para el fichero medidas.txt
RUN mkdir /fichero
# Define environment variable
ENV NAME PRACTICA2_HOST
ENV NAME ZOOKEEPER_HOST
CMD java -jar prPractica3.jar
```

Dicho archivo incluye una instrucción para crear el directorio en el que se creará el fichero “medidas.txt”.

Ilustración 12. Dockerfile

Posteriormente, se crea la imagen y se sube a https://hub.docker.com/ con los comandos explicados en la práctica anterior.

Posteriormente, se crea el *stack.yml*. Dicho docker stack creará el número de servicios que se le soliciten y cada uno se ejecutará con un número diferente de instancia al hacer el deploy.

Solo se necesitará una instancia tanto del servicio de REDIS como del que se creó en la práctica anterior, como del de ZooKeeper. Este último será el que dará servicio al creado en esta última entrega.

Por otra parte, el servicio correspondiente al de la práctica 2 es el único servicio que debe ser accesible desde el exterior. Necesita saber cómo disponer del servicio REDIS y lo hará por la variable de entorno.

En resumen, el servicio de la práctica que se está desarrollando requiere del de práctica 2 y del de ZooKeeper por lo que se necesitan variables de entorno que le digan cómo acceder a ellos. Aunque en principio no se ejecuta ninguna desde el inicio es de saber que cuando el usuario decida lanzar múltiples instancias todas compartirán un volumen común donde escribir datos. A dicho volumen se le ha llamado fichero.

```yaml
version: "3"

#Este docker stack creará cuatro servicios y cada uno se ejecutará con un número diferente de instancias al hacer el deploy
services:
  #REDIS solo debe haber una instancia
  redis:
    image: redis:latest
  #PRACTICA2 solo necesita una instancia, aunque podría haber más. Es el único servicio que debe ser accesible desde el exterior
  # necesita saber cómo disponer del servicio REDIS y lo hará por la variable de entorno
  practica2:
    image: cbaronsuarez/practica2:latest
    ports:
      - "4567:4567"
    environment:
      - REDIS_HOST=redis
    depends_on:
      - redis
    deploy:
      replicas: 1
#Zookeeper se usará para dar servicio a PRACTICA3. Solo debe haber una instancia.
  zookeeper:
    image: zookeeper
#PRACTICA3 requiere de los servicios de PRACTICA2 y ZOOKEEPER. Necesita variables de entorno que le digan como acceder a ellos.
# Aunque en principio no ejecutamos ninguna desde el inicio, debemos saber que cuando el usuario decida lanzar multiples instancias
# Todas compartirán un volumen común donde escribir datos. A este volumen le hemos llamado fichero
  practica3:
    image: cbaronsuarez/practica3
    depends_on:
      - zookeeper
      - practica2
    environment:
      - PRACTICA2_HOST=http://practica2:4567
      - ZOOKEEPER_HOST=zookeeper:2181
    deploy:
      replicas: 0
    volumes:
      - "fichero:/fichero"
volumes:
  fichero:
```

Ilustración 13. stack.yml

EJECUCIÓN

Para poder probar el funcionamiento del programa es necesario ejecutar los siguientes comandos:

```
docker swarm init
```

```
docker stack deploy -c stack.yml mystack
```

El comando siguiente ejecutará 4 veces el programa.

```
docker service scale mystack_practica3=4
```

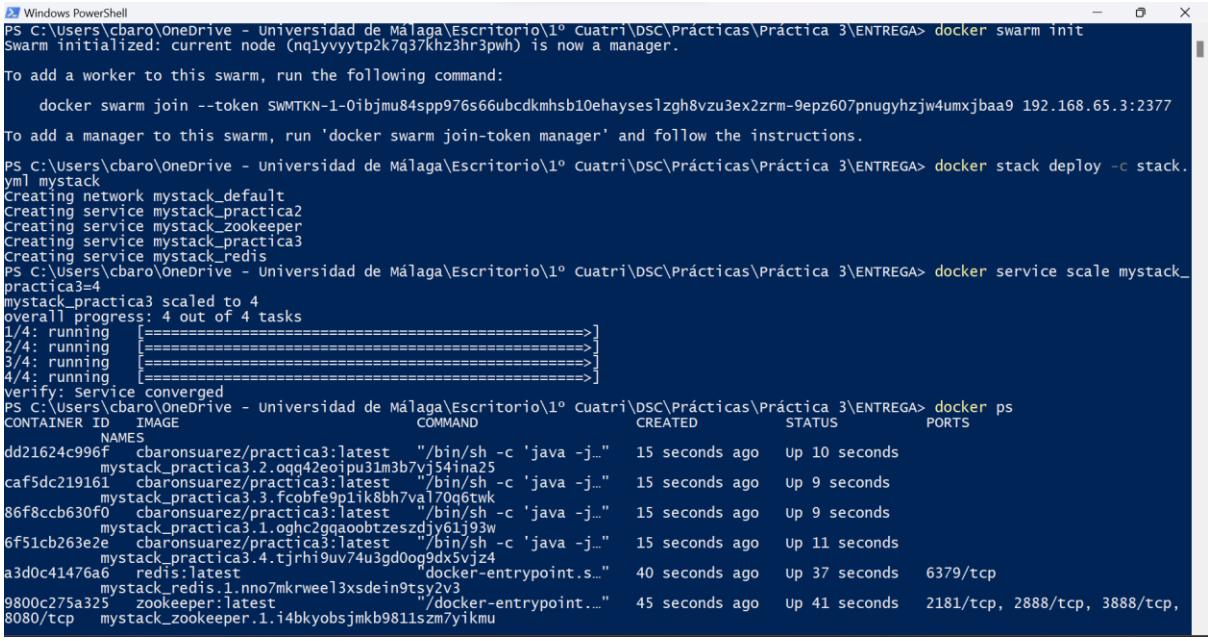

Ilustración 14. Comandos

Para poder ver las medidas que se están generando y las medias calculadas que se están enviando como petición podemos ejecutar los siguientes comandos:

```
docker ps
```

De los 4 contenedores que se han creado de la imagen correspondiente a esta práctica, seleccionamos sus ID y los insertamos en el siguiente comando, donde el ID es "..hash..". Uno de ellos será el master y el resto los esclavos.

```
docker logs ..hash..
```

Por ejemplo: `docker logs dd21624c996f`

Para el líder:

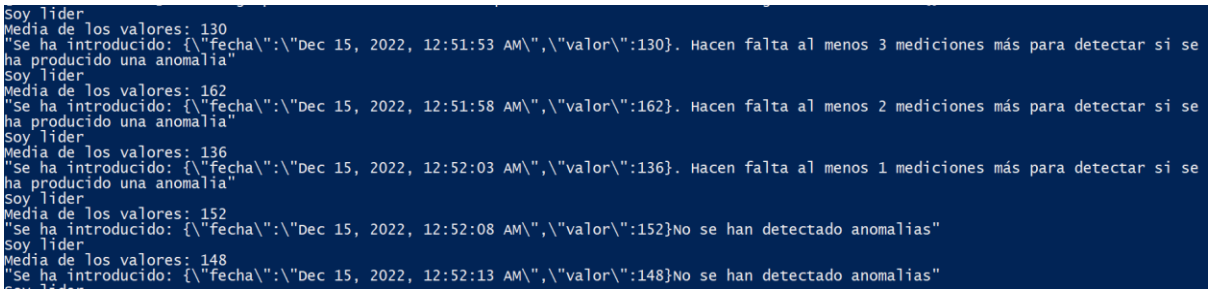

Ilustración 15. Ejecución líder

Para el seguidor:

```
Soy seguidor
Medida generada: 70
Soy seguidor
Medida generada: 186
Soy seguidor
Medida generada: 62
Soy seguidor
Medida generada: 149
Soy seguidor
Medida generada: 90
Soy seguidor
Medida generada: 64
Soy seguidor
Medida generada: 154
Soy seguidor
Medida generada: 73
Soy seguidor
Medida generada: 172
```

Ilustración 16. Ejecución seguidor

En el caso de querer ver el contenido del fichero en el que se están escribiendo y leyendo las medidas, se ejecuta el siguiente comando. "..hash.." puede corresponder a cualquiera de los cuatro IDs mencionados anteriormente.

```
docker exec -it ..hash.. cat /fichero/medidas.txt
```

Comprobamos que, efectivamente, se están enviando correctamente las peticiones: http://localhost:4567/listar

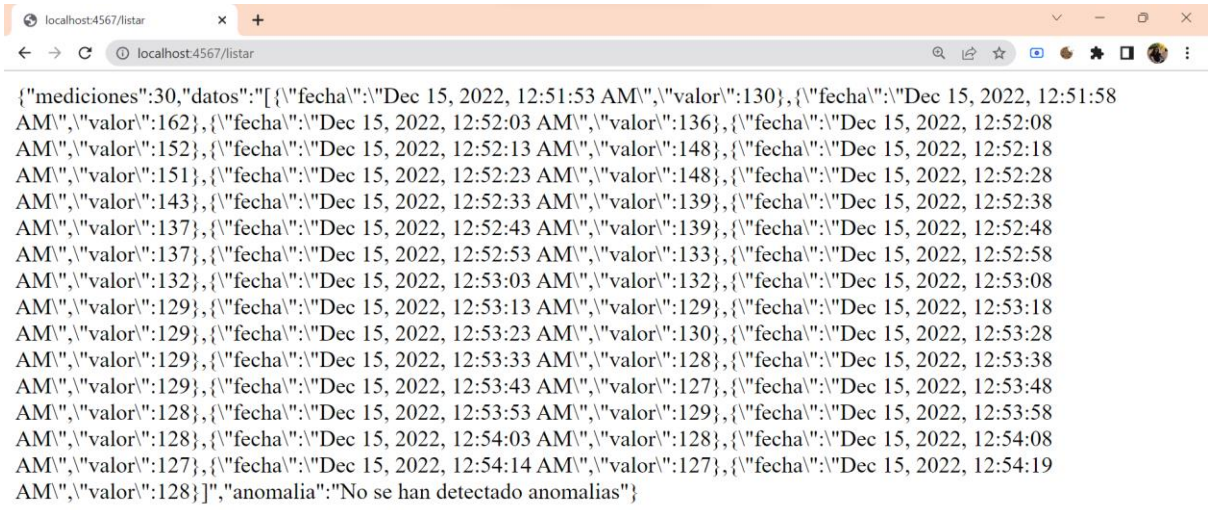

Ilustración 17. http://localhost:4567/listar