Casa abierta al tiempo

UNIVERSIDAD AUTÓNOMA METROPOLITANA

UNIDAD CUAJIMALPA

LICENCIATURA EN TECNOLOGÍAS Y SISTEMAS DE INFORMACIÓN.

PROYECTO DE SISTEMAS DISTRIBUIDOS:

STREAMING P2P

UNIDAD DE ENSEÑANZA

SISTEMAS DISTRIBUIDOS

Dr. Guillermo Monroy

Alumno:
Luis Antonio Salinas Mata

1. Introducción

En este documento se detalla la implementación de un sistema P2P (Peer-to-Peer) simulado para el intercambio de fragmentos de vídeo. El objetivo principal es construir un sistema distribuido que permita a diferentes "nodos" virtuales intercambiar partes de un video, siguiendo las especificaciones de una arquitectura monolítica modular.

Este proyecto fue desarrollado utilizando el framework **Spring Boot**, demostrando la capacidad de simular la lógica de un sistema distribuido en un entorno controlado y empaquetado con **Docker**.

2. Descripción del Problema

El desafío principal consiste en diseñar un sistema que gestione un video grande y lo divida en fragmentos más pequeños para su distribución. La clave es que los "nodos" no poseen el video completo, sino solo una parte de sus fragmentos.

La comunicación y la coordinación entre estos nodos deben ser eficientes, sin una autoridad central que dirija todas las transferencias, emulando así la naturaleza descentralizada de una red P2P.

3. Solución del problema

La solución propuesta es una aplicación de Spring Boot que simula el comportamiento de una red P2P. Se implementó una arquitectura **monolítica modular** donde los "nodos" son representados por servicios internos.

La aplicación divide un video en 10 fragmentos al inicio, los distribuye entre 3 nodos virtuales y utiliza una API REST para simular las solicitudes de intercambio. Para notificar la disponibilidad de nuevos fragmentos, se implementó un sistema **Pub/Sub** usando los eventos internos de Spring. Finalmente, toda la aplicación se empaqueta en un **único contenedor de Docker**, facilitando su despliegue y portabilidad.

4. Desarrollo del Sistema

El sistema se compone de varios módulos de Spring Boot que trabajan en conjunto. A continuación, se detalla la función de cada uno y se incluyen fragmentos de código para su comprensión.

a) Módulos de la Aplicación

- **NodeController.java**: Expone la API REST que permite interactuar con el sistema. Es el punto de entrada para solicitar la transferencia de fragmentos y para consultar el estado de los nodos.
- **NodeService.java**: Contiene la lógica interna de cada nodo virtual. Se encarga de solicitar fragmentos, verificar su disponibilidad y gestionar la colección de fragmentos que posee.
- **FragmentManagerService.java**: Es la capa de gestión central de los fragmentos. Al iniciar la aplicación, lee un video de prueba, lo divide en 10 partes y las distribuye aleatoriamente entre los nodos.
- **EventPublisherService.java**: Implementa el patrón Pub/Sub de Spring. Publica eventos de disponibilidad de fragmentos cada vez que un nodo adquiere uno nuevo.
- **FragmentAvailabilityListener.java**: Actúa como un "oyente" que reacciona a los eventos publicados, simulando que otros nodos se enteran de que un fragmento ya está disponible en la red.
- **Dockerfile**: Define las instrucciones para empaquetar la aplicación en un contenedor de Docker, utilizando una imagen base de OpenJDK.

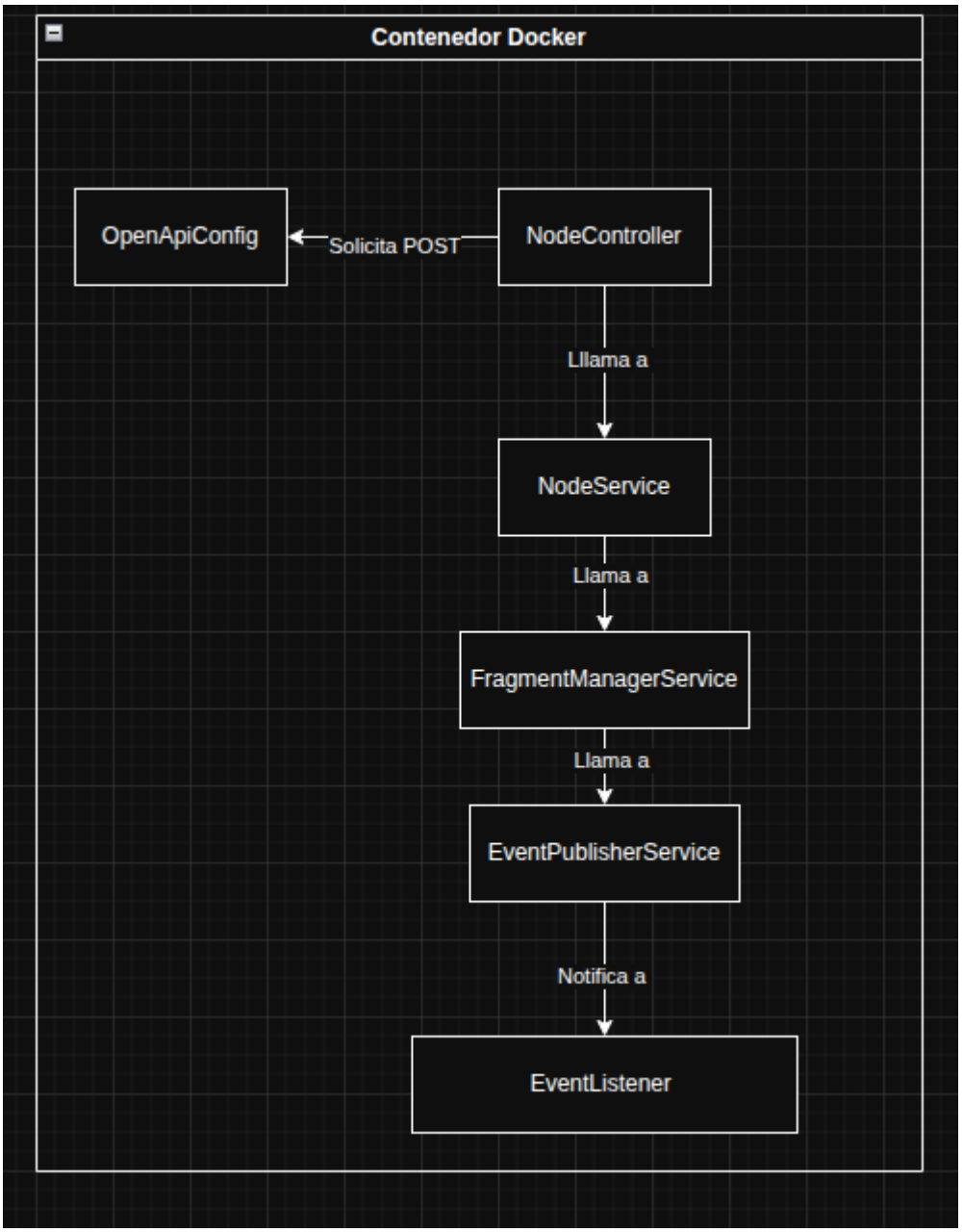

Imagen 1. Diagrama de lógica del programa.

b) Explicación del Código

- **FragmentManagerService.java**: Muestra el método @PostConstruct que lee el archivo de video de recursos y lo divide.

```java
@PostConstruct
public void init() throws IOException {
    System.out.println("Iniciando la división del video y la distribución de fragmentos...");

    // 1. Dividir el video
    byte[] videoData = videoResource.getContentAsByteArray();
    int fragmentSize = videoData.length / 10;

    for (int i = 0; i < 10; i++) {
        int start = i * fragmentSize;
        int end = (i == 9) ? videoData.length : start + fragmentSize;
        byte[] fragmentData = Arrays.copyOfRange(videoData, start, end);

        Fragment fragment = new Fragment(fragmentData);
        fragment.setOriginalVideoId("video.mp4");
        allFragments.put(fragment.getId(), fragment);
    }

    System.out.println("Video dividido en " + allFragments.size() + " fragmentos.");

    // 2. Distribuir los fragmentos entre 3 nodos simulados
    List<String> fragmentIds = new ArrayList<>(allFragments.keySet());
    Collections.shuffle(fragmentIds); // Se mezclan los fragmentos de manera aleatoria

    // Creamos 3 nodos simulados
    String nodeId1 = "node-1";
    String nodeId2 = "node-2";
    String nodeId3 = "node-3";

    // Distribuimos los fragmentos entre los nodos.
    // Cada nodo tendrá una parte del video, pero le faltarán otras.
    nodeFragments.put(nodeId1, new ArrayList<>(fragmentIds.subList(0, 4))); // Fragmentos 0-3
    nodeFragments.put(nodeId2, new ArrayList<>(fragmentIds.subList(4, 7))); // Fragmentos 4-6
    nodeFragments.put(nodeId3, new ArrayList<>(fragmentIds.subList(7, 10))); // Fragmentos 7-9

    System.out.println("Fragmentos distribuidos entre los nodos: " + nodeFragments.keySet());
}
```

Imagen 2. Fragmento del codigo “FragmentManagerService.java”

- **NodeService.java:** Muestra el método requestFragment, que simula la transferencia de un fragmento.

```java
    /**
     * Simula la solicitud de un fragmento a otro nodo.
     */
    public Optional<Fragment> requestFragment(String sourceNodeId, String targetNodeId, String fragmentId) {
        Node targetNode = getNodeById(targetNodeId);
        if (targetNode != null && targetNode.hasFragment(fragmentId)) {
            System.out.println("Node " + sourceNodeId + " solicitando fragmento " + fragmentId + " a " + targetNodeId);

            Optional<Fragment> fragment = fragmentManagerService.getFragmentById(fragmentId);
            if (fragment.isPresent()) {
                Node sourceNode = getNodeById(sourceNodeId);
                if (sourceNode != null) {
                    sourceNode.addFragment(fragmentId);
                    System.out.println("Node " + sourceNodeId + " ha recibido el fragmento " + fragmentId);

                    // Publicamos el evento de disponibilidad.
                    eventPublisherService.publishFragmentAvailabilityEvent(fragmentId, sourceNodeId);
                }
                return fragment;
            }
        }
        System.out.println("Node " + targetNodeId + " no tiene el fragmento " + fragmentId + " o no existe.");
        return Optional.empty();
    }
```

Imagen 3. Fragmento de codigo del "NodeService.java"

`NodeController.java`: Muestra el endpoint POST que llama a `NodeService`.

```java
    @PostMapping("/{sourceNodeId}/request/{targetNodeId}/{fragmentId}")
    @Operation(summary = "Simula la solicitud de un fragmento de un nodo a otro.")
    public ResponseEntity<String> requestFragment(
            @PathVariable String sourceNodeId,
            @PathVariable String targetNodeId,
            @PathVariable String fragmentId) {

        System.out.println("----------------------------------------");
        System.out.println("Solicitud REST recibida:");
        System.out.println("  - Desde Node " + sourceNodeId);
        System.out.println("  - A Node " + targetNodeId);
        System.out.println("  - Fragmento " + fragmentId);


        // El método requestFragment devuelve Optional<Fragment>
        Optional<Fragment> fragmentResult = nodeService.requestFragment(sourceNodeId, targetNodeId, fragmentId);

        // Verificamos si el Optional contiene un fragmento
        if (fragmentResult.isPresent()) {
            // Si está presente, obtenemos el objeto Fragment y luego sus datos
            // byte[] fragmentContent = fragmentResult.get().getData();
            // Aunque no es necesario usar los datos en la respuesta, puedes acceder a ellos así.
            return ResponseEntity.ok("Transferencia exitosa. Fragmento " + fragmentId + " recibido por " + sourceNodeId);
        } else {
            return ResponseEntity.badRequest().body("Fallo en la transferencia. El nodo destino no tiene el fragmento.");
        }
    }
}
```

Imagen 4. Fragmento de codigo del "`NodeController.java`"

5. Ejecuciones de Prueba

Se realizaron varias pruebas para verificar el funcionamiento del sistema.

a) Inicio de la Aplicación y Distribución

1. **Comando de inicio**: docker run -p 8080:8080 p2p-video-app
2. **Log de inicio**: El log muestra que la aplicación se inicia correctamente, el video se divide y los fragmentos se distribuyen entre los nodos.

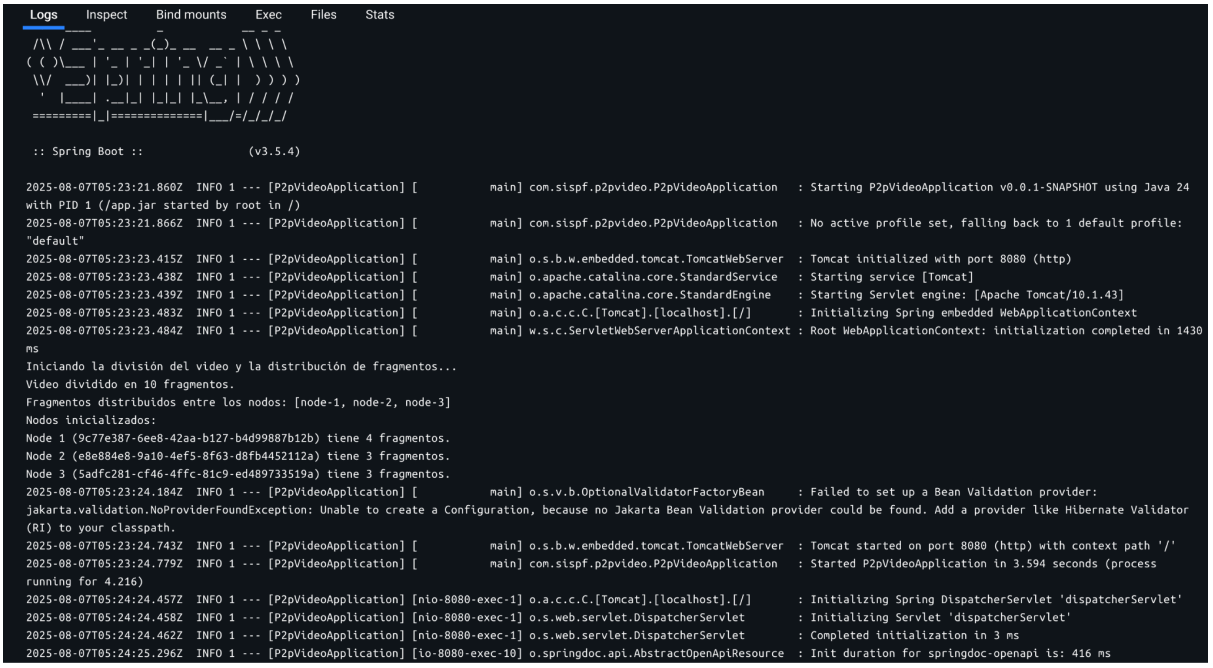

Imagen 5. Log de ejecución del programa

b) Transferencia Simulada de Fragmentos

1. **Consulta de Fragmentos Faltantes**: Se utilizó Swagger para consultar el nodo 1 (GET /api/nodes/{id_del_nodo_1}). Se identificó que le faltaba un fragmento que el nodo 2 (GET /api/nodes/{id_del_nodo_2}) sí tenía.

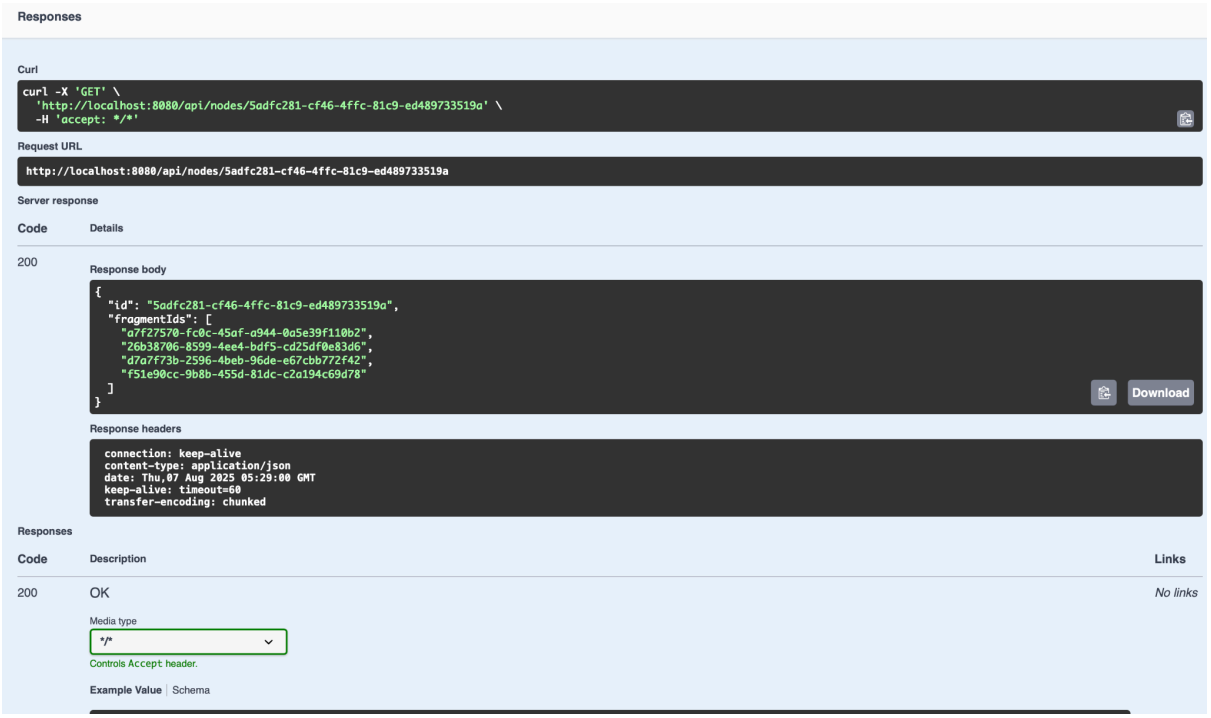

Imagen 6. Uso de swagger para consultar GET.

2. **Solicitud de Transferencia**: Se realizó una solicitud POST desde Swagger, con los IDs correspondientes.

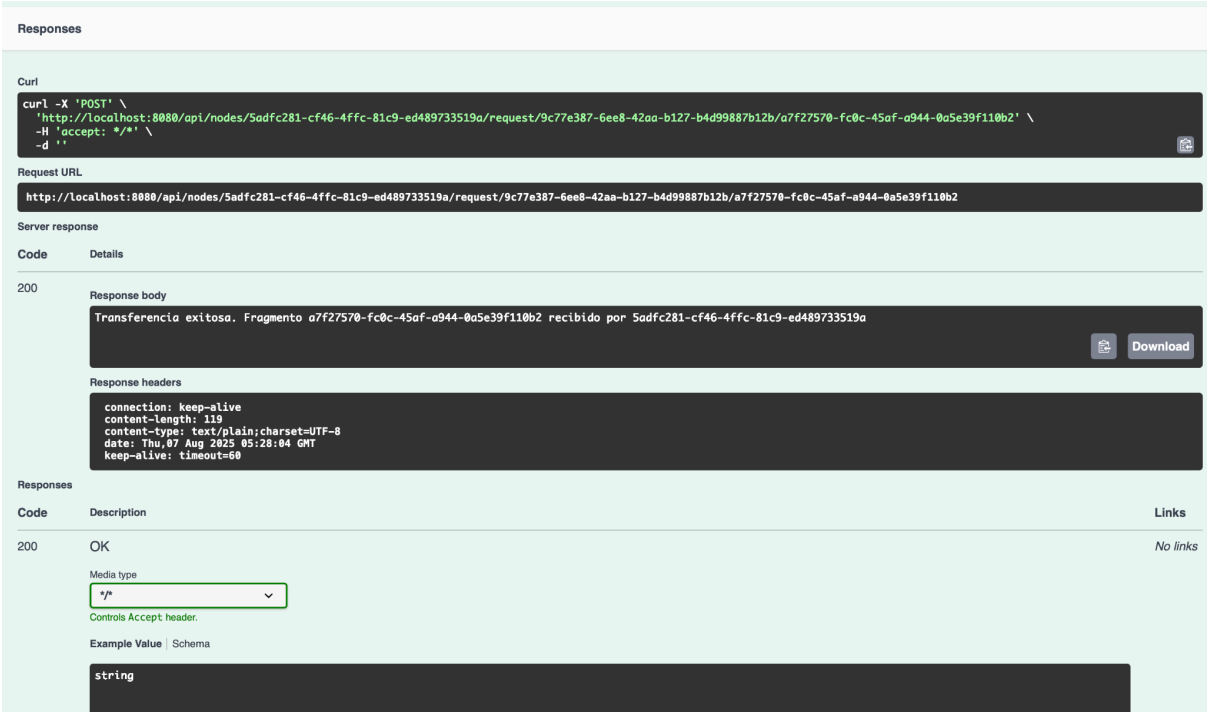

Imagen 7. Uso de swagger para la solicitud POST.

3. **Resultado en los Logs**: El log de Docker muestra el flujo de la transferencia simulada y la publicación del evento.

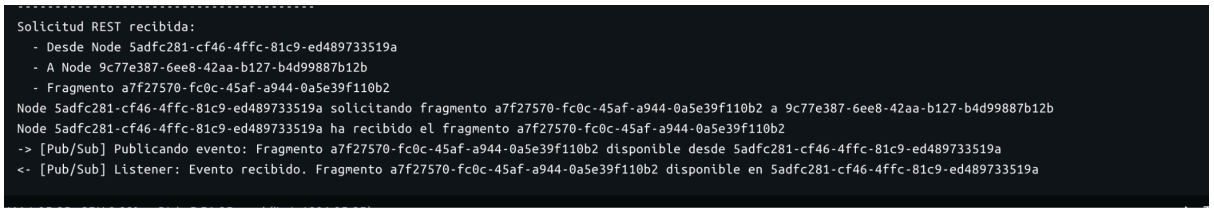

Imagen 8. Log Resultado

6. Conclusión

Se demostró cómo una arquitectura **monolítica modular** puede ser utilizada para simular un sistema distribuido de forma efectiva. La implementación del sistema de eventos de Spring como mecanismo de **Pub/Sub** fue clave para simular la comunicación asíncrona entre nodos.

Finalmente, el uso de **Docker** encapsula la aplicación de forma portable, lo que valida la solución como un sistema moderno y listo para el despliegue. El proyecto no sólo resolvió el problema del intercambio de fragmentos, sino que también sirvió como un excelente ejercicio para entender los principios de la arquitectura distribuida.