Diseño del Sistema Distribuido: Spotify Clone

Autores:
Yesenia Valdés Rodríguez C411
Olivia Ibañez Mustelier C411

Índice

Introducción

En este informe se presenta la arquitectura distribuida utilizada en el proyecto *SpotifyClone*, basada en un modelo *leader-workers* respaldado por el algoritmo de consenso Raft para la elección y mantenimiento del líder. El líder coordina todas las operaciones de escritura sobre los nodos de almacenamiento y distribuye la carga de las operaciones de lectura entre los *workers*. Dado que la aplicación presenta una baja frecuencia de escrituras y una alta demanda de lecturas, el diseño prioriza la consistencia del sistema por encima de la disponibilidad. El proyecto está implementado en Python 3.11, empleando Django REST para la comunicación cliente-servidor y Pyro (RPC) para la comunicación entre servidores.

1. Arquitectura

La arquitectura del proyecto se organiza en dos componentes principales: la parte cliente y el conjunto de servidores. Ambos elementos cooperan para ofrecer un sistema distribuido tolerante a fallos, escalable y con una interfaz clara basada en servicios REST.

1.1. Cliente

La parte cliente está implementada en *Vue.js* y su función principal es interactuar con la API mediante peticiones HTTP. Este componente:

- Envía solicitudes y procesa las respuestas provenientes del servidor.
- Ensambla los fragmentos de audio recibidos mediante *streaming* y los reproduce en tiempo real.
- Lista las canciones disponibles.
- Permite crear nuevos artistas, álbumes y canciones a través de la interfaz gráfica.

El cliente se comunica exclusivamente mediante una interfaz uniforme basada en REST, lo que garantiza desacoplamiento y simplicidad en la interacción.

1.2. Arquitectura del Servidor

Los servidores siguen inicialmente una arquitectura por capas, organizada de la siguiente manera:

- **Modelos**: representan las entidades almacenadas en la base de datos.
- **Serializadores**: transforman los modelos a formatos intercambiables a través de la API y viceversa.
- **Vistas o Controladores**: exponen los servicios mediante *endpoints* REST y coordinan el acceso a la lógica de negocio.

Cada capa se sitúa sobre la anterior y actúa como intermediaria entre el cliente y los datos persistidos, logrando una separación clara de responsabilidades.

1.3. Coordinación Distribuida con Raft

Además de la estructura en capas, el backend incorpora una arquitectura distribuida basada en coordinación entre nodos empleando el protocolo *Raft*. Cada servidor funciona como un nodo dentro de un clúster y participa en:

- Un proceso continuo de **elección de líder**.
- Gestión del **estado replicado** entre nodos.
- Mecanismos de **tolerancia a fallos** y **consenso**.

Cuando un nodo es elegido líder, ejecuta en un hilo independiente el servicio **LeaderManager**, encargado de:

- Gestionar la replicación de archivos.
- Mantener la consistencia fuerte entre los nodos.
- Procesar todas las **operaciones de escritura**.
- Administrar el estado compartido del clúster.

Asimismo, las **operaciones de lectura** se distribuyen entre los nodos trabajadores mediante un esquema de balanceo de carga:

- Las lecturas se procesan en los *workers*, permitiendo escalabilidad horizontal.
- Las escrituras se gestionan exclusivamente en el líder, garantizando consistencia estricta.

Un aspecto relevante es que los serializadores y vistas no acceden directamente a la base de datos local de cada nodo. En su lugar, se comunican con la capa distribuida gestionada por Raft para obtener datos autorizados y consistentes, es decir, cada nodo worker utiliza las funciones RPC para procesar las peticiones que le sean asignadas manteniendo así una consistencia funcional a nivel de código. Esto evita estados divergentes y asegura coherencia incluso frente a fallos parciales del sistema.

1.4. Arquitectura Híbrida del Sistema

La integración de estos componentes da lugar a una arquitectura híbrida con las siguientes características:

- **REST**: para la comunicación uniforme entre cliente y servidor.
- **Arquitectura por capas**: que organiza el backend en diferentes niveles de responsabilidad.
- **Coordinación distribuida (Raft)**: para la elección de líder, consenso y replicación consistente.
- **Servicios internos especializados**: como el *LeaderManager*, encargado de mantener la coherencia del sistema distribuido.

En conjunto, esta arquitectura proporciona un sistema robusto, escalable y tolerante a fallos, con una interfaz clara para los clientes y un backend capaz de coordinar múltiples nodos de forma eficiente.

2. Gestión de Procesos y Concurrencia

La gestión de procesos dentro del sistema se basa en el uso de **hilos de ejecución** (*threads*) para paralelizar tareas que deben ejecutarse en segundo plano. Estas tareas incluyen la coordinación entre nodos para asegurarse de que exista un líder, el monitoreo del estado del clúster para saber si los nodos están vivos y las operaciones del servicio *LeaderManager* que incluyen la replicación de archivos, balanceador de carga y coordinador. Mientras estas actividades ocurren en hilos independientes, la API principal se mantiene **síncrona**, procesando una petición a la vez por cada nodo servidor.

2.1. Modelo de Ejecución

El esquema de concurrencia utilizado se estructura de la siguiente manera:

- Las **tareas críticas de fondo** (elección de líder, replicación, mantenimiento del estado del clúster, coordinación en caso de fallo) se ejecutan en hilos separados para no bloquear la API.
- La **API REST** opera de forma síncrona, atendiendo una solicitud a la vez en el hilo principal del proceso.
- La concurrencia se implementa mediante hilos estándar de Python, evitando enfoques basados en *event loops* o procesos independientes.

Este enfoque facilita la separación entre la lógica del clúster (paralela) y la atención al cliente (serializada), reduciendo la complejidad general del diseño.

2.2. Comportamiento de los Hilos en Python

En Python, la ejecución de hilos está regulada por el **Global Interpreter Lock (GIL)**, que limita la ejecución simultánea de bytecode a un solo hilo por proceso. Sin embargo, para sistemas cuyo trabajo principal es el acceso a red o disco, el GIL no suele ser un obstáculo significativo, debido a que:

- Durante operaciones de entrada/salida (I/O), como lectura de archivos, escritura en disco o comunicación de red, **los hilos liberan el GIL**.
- Al liberar el GIL, otros hilos pueden continuar su ejecución mientras uno espera la finalización de una operación bloqueante.
- Esto permite un grado efectivo de paralelismo para cargas de **I/O intensivo**, que es precisamente el caso del proyecto.

2.3. Ventajas del Enfoque Basado en Hilos y API Síncrona

El uso de hilos para las tareas de fondo combinado con una API síncrona aporta beneficios importantes:

- **Facilidad de implementación**: el modelo de hilos es sencillo de programar y depurar respecto a arquitecturas asíncronas o basadas en procesos.

- **Aprovechamiento eficiente para I/O**: debido a que los hilos liberan el GIL durante operaciones de I/O, se obtiene paralelismo efectivo en la mayoría de las tareas críticas del sistema.
- **Aislamiento de responsabilidades**: los hilos de fondo ejecutan procesos del clúster sin interferir con la gestión de peticiones HTTP.
- **Menor complejidad de sincronización**: al procesar una petición a la vez por nodo, se reducen las condiciones de carrera en la API.
- **Estabilidad del sistema**: la lógica secuencial de las vistas evita errores típicos de modelos altamente concurrentes.

Estas ventajas hacen que el enfoque sea adecuado para un sistema distribuido donde las operaciones pesadas son mayormente I/O y donde cada nodo tiene responsabilidades bien delimitadas.

2.4. Desventajas y Limitaciones del Modelo

A pesar de sus beneficios, el enfoque presenta limitaciones que deben considerarse:

- **Baja concurrencia en la API**: al manejar solicitudes de forma estrictamente síncrona, cada nodo atiende solo una petición a la vez.
- **Posibilidad de bloqueos en el hilo principal**: operaciones de escritura remota o esperas de red pueden retrasar todas las solicitudes del nodo.
- **Dependencia alta del balanceo de carga**: si el líder distribuye mal las lecturas, los nodos pueden saturarse fácilmente.
- **Escalabilidad limitada**: bajo cargas intensivas de CPU, el GIL impediría adquirir verdadero paralelismo.
- **Complejidad añadida en sincronización interna**: aunque menor que en sistemas asíncronos totales, la comunicación entre hilos aún requiere mecanismos de bloqueo.

En general, este modelo de concurrencia es adecuado para un sistema dominado por operaciones de entrada/salida, pero requeriría una revisión más profunda si en el futuro se buscan mayores niveles de concurrencia o procesamiento intensivo dentro de cada nodo.

3. Comunicación en el Sistema

El sistema implementa dos mecanismos principales de comunicación: la interacción **cliente-servidor**, basada en el protocolo HTTP, y la comunicación **servidor-servidor**, implementada mediante llamadas a procedimientos remotos (RPC). Cada una de estas formas de comunicación cumple un propósito específico dentro de la arquitectura y presenta ventajas y consideraciones particulares.

3.1. Comunicación Cliente-Servidor

La comunicación entre el cliente y los servidores se realiza mediante el protocolo **HTTP**, siguiendo el estilo arquitectónico REST. Este enfoque establece una interfaz uniforme y desacoplada entre ambas partes, donde el cliente puede interactuar con la API enviando solicitudes y recibiendo respuestas en formatos estructurados como JSON.

Las características más relevantes de esta comunicación son:

- **Estilo REST**: define una interfaz consistente y autodescriptiva.
- **Simplicidad**: los clientes realizan peticiones mediante métodos estándar como `GET`, `POST`, `PUT` y `DELETE`.
- **Desacoplamiento**: el cliente no necesita conocer la lógica interna del servidor, solo los puntos de entrada de la API.
- **Compatibilidad**: HTTP es ampliamente soportado y facilita la integración con herramientas externas.

Esta comunicación es ideal para la interacción usuario-aplicación, ya que proporciona una capa clara y estable para realizar operaciones de consulta, administración de entidades y reproducción de contenido.

3.2. Comunicación Servidor-Servidor

Los nodos del clúster se comunican entre sí mediante un sistema de **Remote Procedure Calls** (RPC). Esta forma de comunicación permite a un servidor invocar directamente funciones expuestas por otro nodo, como si se tratara de una llamada local, abstrayendo la complejidad del transporte.

El uso de RPC es fundamental para implementar la lógica distribuida del sistema, en particular las operaciones relacionadas con:

- **Elección de líder** mediante el protocolo Raft.
- **Replicación de archivos y estados**.
- **Sincronización de metadatos** entre nodos.
- **Transmisión de operaciones de escritura** desde el líder hacia las réplicas.

Entre las características clave de la comunicación RPC se encuentran:

- **Llamadas directas a funciones remotas**: lo que simplifica la implementación de algoritmos de consenso y sincronización.
- **Bajo acoplamiento lógico**: aunque los nodos colaboran estrechamente, la interfaz RPC define puntos de interacción bien delimitados.
- **Eficiencia en la comunicación interna**: RPC suele ser más ligero que HTTP para operaciones repetidas entre nodos.
- **Soporte para operaciones síncronas**: especialmente importante para la semántica de consistencia fuerte del líder.

3.3. Comparación y Complementariedad

Ambos mecanismos de comunicación coexisten y se complementan dentro del sistema:

- La comunicación **HTTP** está orientada a la interacción humana o de aplicaciones externas, donde la claridad, estandarización y robustez son esenciales.
- La comunicación **RPC** está orientada a la coordinación distribuida entre nodos, donde la baja latencia y la capacidad de invocar operaciones remotas son críticas.

El resultado es un modelo híbrido que aprovecha las fortalezas de ambos enfoques: simplicidad y compatibilidad para la comunicación con clientes, y eficiencia y control para la coordinación interna del clúster distribuido.

4. Coordinación del Sistema Distribuido

La coordinación entre nodos es un componente esencial del sistema y se implementa mediante el protocolo **Raft**, el cual proporciona mecanismos de consenso, elección de líder y mantenimiento de un estado global consistente. En este proyecto, Raft no solo coordina el liderazgo, sino que también sirve como base para la gestión del almacenamiento distribuido y el control de versiones de los archivos.

4.1. Elección de Líder

El sistema utiliza el proceso de elección de líder de Raft para determinar cuál nodo actúa como coordinador. El líder resultante se convierte en la autoridad para todas las operaciones de escritura y la administración del estado global.

Este mecanismo asegura que siempre exista un único líder válido y actualizado, incluso ante fallos o desconexiones temporales.

4.2. Funcionamiento de Raft

El procedimiento básico del consenso distribuido puede entenderse de manera progresiva a medida que la red crece y los nodos interactúan. En un estado inicial, cuando el sistema está compuesto por un solo nodo, dicho nodo se considera automáticamente líder, pues no existe competencia ni necesidad de votar. Al incorporarse un segundo nodo a la red, ambos intercambian información para sincronizar su estado interno —en particular, el *término* (*term*) actual, que identifica la época lógica del sistema—. Mientras el líder envía mensajes de mantenimiento (*heartbeats*) al nuevo integrante, este reconoce su rol (ya que su término es menor o igual que el del líder, y el mensaje de mantenimiento indica que proviene del líder), entonces dicho nodo actualiza su término según el del líder y permanece como *follower*.

Cada nodo mantiene internamente dos elementos fundamentales: su **término actual** (que indica en qué época lógica se encuentra) y un **registro interno de logs** (que contiene las operaciones pendientes de compromiso y sirve para comparar qué nodo está más actualizado). A medida que se integran más nodos, el sistema se mantiene estable siempre que estos reciban comunicación periódica del líder. Sin embargo, cuando un nodo deja de recibir *heartbeats* dentro de un intervalo determinado —ya sea por fallos de comunicación, caídas temporales o desincronización— asume que el líder ha dejado de estar disponible y pasa al estado de *candidate*. En este estado inicia un nuevo *término*, incrementando su contador de época, y envía solicitudes de voto (*RequestVote*) al resto de los nodos.

Cada nodo que recibe una solicitud de voto sigue reglas estrictas: solo puede votar una vez por término y únicamente si considera que el candidato está suficientemente actualizado. El nodo que se convierte en candidato también emite su propio voto para sí mismo al iniciar el proceso electoral. Para decidir si un candidato está actualizado, se evalúan dos criterios esenciales: (1) que el candidato posea un término mayor o igual que el del votante, y (2) que el registro de logs del candidato esté al menos tan completo como el del nodo votante. Si cualquiera de estas condiciones no se cumple, la solicitud de voto es denegada.

Cada nodo posee un temporizador independiente y aleatorio —en nuestro caso entre 3 y 7 segundos—. Este temporizador se reinicia cada vez que el nodo recibe comunicación válida del

líder. Como los temporizadores son aleatorios, es muy probable que uno expire antes que los otros, provocando que ese nodo sea el primero en convertirse en candidato y solicitar votos, reduciendo así colisiones y empates en las elecciones.

El candidato que obtiene la mayoría absoluta de los votos (más de la mitad de los nodos del sistema, es decir, $\lfloor n/2 \rfloor + 1$ con n nodos totales) se convierte en el nuevo **líder**. Una vez elegido, comienza a enviar *heartbeats* periódicos para reafirmar su autoridad y evitar que surjan nuevas elecciones innecesarias. Si durante el proceso de votación varios nodos se convierten simultáneamente en candidatos y se produce una división de votos sin que ninguno alcance la mayoría, todos regresan al estado de *follower* cuando sus temporizadores expiran, iniciándose un nuevo término y repitiendo el proceso hasta que un único candidato consiga la mayoría.

En resumen, el procedimiento completo implica:

- Nodos que no reciben comunicación del líder pasan a estado de *candidate* y generan un nuevo término.
- Cada candidato solicita votos al resto de los nodos mediante mensajes *RequestVote* y emite un voto para sí mismo.
- Los nodos votan una única vez por término, siempre que el candidato tenga un término mayor o igual y un registro de logs tan completo o más que el suyo.
- El candidato que obtiene la mayoría se convierte en **líder** y envía *heartbeats* periódicos para mantener la estabilidad del sistema.
- Si ningún candidato obtiene la mayoría, los nodos regresan a *follower* y una nueva ronda de votaciones comenzará cuando expiren los temporizadores aleatorios (3–7 segundos).

Esta dinámica garantiza que, sin importar la cantidad de nodos o la incorporación progresiva de nuevos participantes, el sistema converge a un único líder válido por término, manteniendo la coherencia y robustez del consenso distribuido.

4.3. Log del Líder y Seguimiento del Estado de Almacenamiento

Una particularidad del proyecto es que el líder mantiene, dentro de su **log de Raft**, información adicional relacionada con la gestión del sistema de archivos distribuido. En este registro, el líder conserva:

- **La ubicación de almacenamiento de cada archivo** (qué nodos contienen sus réplicas).

- **El listado de nodos que cumplen funciones de base de datos**, es decir, los nodos autorizados para persistir metadatos y réplicas.
- **La versión de cada archivo**, asociada al término en el que se generó la última modificación.

De esta forma, el log del líder no solo cumple con la función clásica de consenso de Raft, sino que actúa como la fuente autorizada de información para gestionar todo el sistema de almacenamiento.

4.4. Sobre el *heartbeat* y la replicación del Log

El mecanismo de comunicación periódica que el líder envía a los demás nodos, conocido comúnmente como *heartbeat*, es en realidad un mensaje de **AppendEntries**. Este mensaje no solo cumple la función de indicar que el líder sigue activo y detectar que nodos siguen vivos, sino que también es el método mediante el cual se replica y mantiene actualizado el **log** en todos los nodos del sistema.

En cada *AppendEntries*, el líder incluye:

- Su **término actual**, indicando en qué época lógica se encuentra.
- Su **identidad** como líder legítimo del término.
- Las **entradas de su log** que deben ser replicadas en los seguidores.

De este modo, lo que llamamos *heartbeat* no es un simple mensaje vacío, sino una operación que también sirve para mantener sincronizados los registros internos de los nodos. Incluso cuando no hay nuevas entradas para agregar al log, el envío de un *AppendEntries* vacío (un *heartbeat*) reafirma la autoridad del líder y permite que los seguidores mantengan actualizado el puntero de coincidencia y la información de replicación.

Los nodos seguidores utilizan estos mensajes para:

- **Actualizar su log** con las entradas enviadas por el líder.
- **Confirmar** que el líder continúa disponible y en control del término.
- **Detectar incoherencias y actualizar** si las entradas no coinciden con su propio log.

Después de recibir un *AppendEntries*, cada seguidor envía una **respuesta positiva** (un acuse de recibo) si el log pudo aplicarse correctamente. Si un seguidor no responde dentro de un tiempo determinado, el líder asume que dicho nodo está temporalmente desconectado o inaccesible. En

estos casos, el líder continúa funcionando con el resto de nodos disponibles y seguirá intentando la replicación hasta que el nodo faltante vuelva a responder.

En conjunto, este mecanismo asegura que todos los nodos mantengan una copia coherente y actualizada del log, al mismo tiempo que proporciona la infraestructura fundamental para detectar fallos, mantener la autoridad del líder y sostener la consistencia del sistema distribuido.

4.5. El Término de Raft como Unidad Temporal

En Raft, como ya se explicó anteriormente, se tienen los **términos** que son una medida lógica del progreso del sistema. En este proyecto, el término se utiliza como **unidad temporal global** para coordinar y validar el estado distribuido. Esto permite:

- **Detectar y descartar información desactualizada** proveniente de nodos atrasados.
- Asociar cada operación del sistema a un punto temporal bien definido.
- Controlar versiones de archivos y metadatos mediante una noción estricta de orden.

El término se incrementa automáticamente durante elecciones de líder o cuando el sistema detecta inconsistencias en la comunicación. En este proyecto, además, se amplía el uso del término más allá de Raft para incluir operaciones regulares de lectura y escritura.

4.6. Actualización del Término en Operaciones de Lectura y Escritura

Cada vez que se realiza una operación de lectura o escritura, el sistema incrementa el término o lo utiliza para validar la operación. Esto aporta varias ventajas:

- **Consistencia temporal**: cada operación tiene un sello temporal derivado del término.
- **Control de versiones**: la versión de un archivo se liga al término en el que fue modificada.
- **Detección de obsolescencia**: si un nodo reporta una versión con un término inferior, se identifica automáticamente como archivo desactualizado.

De esta manera, el término funciona como un mecanismo global para asegurar que todas las operaciones se realizan sobre información vigente.

4.7. Sincronización, Replicación y Exclusión Mutua

El líder coordina la replicación de archivos y metadatos enviando entradas del log a los nodos seguidores. Una operación de escritura solo se confirma cuando todos los nodos del clúster replica la entrada correspondiente, garantizando así:

- **Consenso firme sobre el estado global**.
- **Exclusión mutua implícita**, ya que solo el líder puede aceptar modificaciones.
- **Orden global de las operaciones**, determinado por el log y el término asociado.

Estas propiedades permiten un alto grado de consistencia incluso en presencia de fallos parciales.

5. Nombramiento y Descubrimiento Automático de Nodos

El sistema implementa un mecanismo simple y automático de descubrimiento de nodos basado en las capacidades internas de **Docker Swarm**. Dado que los servidores se despliegan en una red *overlay*, cada nodo es registrado automáticamente en el **DNS interno de Docker**. Esto permite que cualquier nodo pueda resolver por nombre a los demás sin configuración manual adicional.

La arquitectura aprovecha este comportamiento para:

- Identificar dinámicamente los nodos activos del clúster.
- Resolver los nombres de servicio asignados a cada nodo.
- Permitir que Raft y los servicios internos se comuniquen sin necesidad de direcciones IP estáticas.

5.1. Distribución de Servicios en Docker Swarm

La arquitectura empleada en el proyecto se despliega sobre dos computadoras físicas que actúan como nodos de un mismo clúster Docker Swarm. Este entorno permite ejecutar los servicios distribuidos de Raft y los componentes del sistema sin necesidad de configuraciones manuales de red ni direcciones estáticas.

El proceso de integración al Swarm ocurre de la siguiente forma: una de las máquinas inicializa el clúster y se convierte automáticamente en *manager*. Esta máquina genera un identificador seguro que los demás nodos deben usar para unirse. La segunda computadora, conectada a la misma red física o virtual, emplea dicho identificador para incorporarse al clúster. Una vez admitida, pasa a

ser reconocida por el Swarm como un nodo *worker* o un nodo adicional con rol de *manager*, según la arquitectura requerida.

Una vez ambas máquinas forman parte del mismo Swarm, Docker crea de manera automática una red *overlay* distribuida. Esta red funciona como un segmento virtual que abarca todos los nodos del clúster, permitiendo que los contenedores ejecutados en cualquiera de las computadoras puedan comunicarse entre sí como si estuvieran en la misma red local, aun cuando se encuentren en máquinas separadas. La red *overlay* incluye también un sistema interno de resolución de nombres mediante DNS, gracias al cual cada servicio desplegado en el Swarm obtiene un nombre estable que todos los nodos pueden resolver sin conocer sus direcciones IP concretas.

Sobre esta infraestructura se monta el proyecto. Cada nodo físico ejecuta contenedores del backend distribuido, incluyendo las instancias participantes del protocolo Raft. El líder y los workers pueden ejecutarse en nodos distintos y, gracias al DNS interno del Swarm, cada uno puede descubrir al resto sin configuración adicional. Los servicios internos del sistema (como el coordinador, el replicador, el balanceador de carga y los procesos de Raft) interactúan a través de esta red compartida, lo que permite distribuir responsabilidades de forma transparente y tolerante a fallos.

Este esquema facilita no solo la comunicación entre nodos, sino también la redistribución de servicios cuando una máquina cae o se reincorpora. Docker Swarm se encarga automáticamente de mantener el estado deseado del clúster, garantizando que cada servidor del sistema distribuido ejecuta el conjunto de servicios definido por la arquitectura, sin requerir supervisión manual por parte del desarrollador.

6. Consistencia y Replicación

La consistencia y la replicación son elementos fundamentales en el diseño del sistema. Dado que las operaciones de lectura superan ampliamente en volumen a las operaciones de escritura, el proyecto prioriza de forma explícita la **consistencia** por encima de la disponibilidad. Esta decisión garantiza que los usuarios siempre accedan a los datos más actuales, tanto en la recuperación de fragmentos de audio como en la consulta de metadatos asociados.

6.1. Modelo de Consistencia Adoptado

El sistema utiliza un esquema de **consistencia fuerte**, particularmente cercano al modelo de *consistencia linealizable*, donde todas las operaciones se ordenan según un reloj lógico global y todos los nodos observan ese orden. En esta arquitectura:

- Todas las operaciones de escritura se realizan exclusivamente en por el líder y este manda a replicar.

- El término de Raft funciona como una medida temporal para ordenar las operaciones.
- Una operación se considera completada únicamente cuando todas las réplicas confirman su actualización.

De este modo, cualquier cliente que lea un archivo o sus metadatos obtiene siempre la versión más reciente confirmada en el clúster.

6.2. Replicación de Datos

La replicación se utiliza tanto para mejorar la disponibilidad como para prevenir la pérdida de información ante fallos múltiples. El líder es el responsable de gestionar todo el proceso de replicación.

6.2.1. Replicación de Archivos mediante Shards

Los archivos de audio se dividen en **shards**, que son distribuidos entre los nodos del sistema. Cada shard se almacena en **K nodos distintos**, donde el valor de $K \geq 3$ garantiza que incluso si ocurren fallos simultáneos de nivel 2, ninguna parte del archivo se pierda.

El líder es responsable de:

- Dividir cada archivo en shards.
- Seleccionar los nodos responsables de almacenar cada shard, distribuyendo la carga.
- Enviar los shards mediante RPC a los nodos correspondientes.
- Registrar en su log la ubicación de cada shard, su versión y el término asociado.

Además, el líder mantiene para cada shard un registro del **último término en que cada nodo realizó una operación** sobre ese shard. Esta información permite:

- Detectar nodos que almacenan versiones obsoletas.
- Seleccionar nodos para lectura según su grado de actualización y carga.

6.2.2. Replicación de Metadatos

Los metadatos se almacenan en nodos especiales denominados **nodos Base de Datos**, cada uno con una copia completa de los datos en una base SQLite local. Existen exactamente **K nodos con este rol**, y el líder es siempre uno de ellos.

La replicación de metadatos incluye:

- Insertar o actualizar los datos primero en la base de datos del líder.
- Enviar mediante RPC los cambios a los demás nodos Base de Datos.
- Confirmar la operación únicamente si todos los nodos con el rol correspondiente responden con éxito.

Este modelo corresponde al *protocolo de escritura remota* utilizado para consistencia fuerte.

6.3. Manejo de Operaciones de Escritura

El líder gestiona todas las operaciones de escritura, tanto de archivos como de metadatos. El proceso general es el siguiente:

1. El líder recibe la solicitud de escritura.
2. Si la operación involucra metadatos, se realiza primero la inserción SQL local.
3. Si ocurre un error SQL (p. ej., violación de restricción), el cliente recibe inmediatamente un error.
4. Si la operación local es exitosa, el líder envía los cambios a los nodos base de datos mediante RPC.
5. Cada nodo replica la información y responde con `OK`.
6. Si todos los nodos responden satisfactoriamente, la operación se confirma y se responde `OK` al cliente.

6.3.1. Ante caídas de nodos

Si alguno de los nodos de base de datos (metadatos) se desconecta, el líder ejecuta en segundo plano con LeaderManager detecta cuantos nodos base de datos hay y en base a eso asigna los faltantes mandando a replicar su base de datos.

Si es un nodo que contiene shards, se aplica la misma lógica de replicación para restaurar las réplicas faltantes de los shards (fragmentos de las canciones).

Si el líder se desconecta, entonces Raft inicia un proceso de elección para seleccionar un nuevo líder, este toma la base de datos más actualizada y replica los shards y metadatos faltantes. Esto asegura que en caso de que se realice una escritura y se desconecte el líder y/u otro nodo base de datos pero la escritura fue completada en un 3er nodo base de datos, no se pierda la información.

Con este mecanismo, se garantiza tolerancia a fallos de nivel 2.

6.4. Control de Versiones mediante Términos de Raft

Cada archivo y metadato tiene asociado un **término de lectura y uno de escritura**, el de escritura funciona como versión global y el de lectura se usa para balanceo de carga. Este término se actualiza:

- En cada operación de escritura que es global y lectura, de tal forma que no siempre se lea del mismo nodo.
- En operaciones de borrado.

El término permite:

- Determinar si una versión almacenada por un nodo es **válida** u **obsoleta**.
- Coordinar la lectura correcta de shards en nodos que posean la versión más reciente, evitando sobrecargar al mismo nodo con peticiones de lectura.
- Mantener un historial temporal uniforme en todo el clúster.

Esto permite mantener una consistencia estricta en el acceso a los archivos.

7. Tolerancia a Fallos

La arquitectura distribuida del sistema está diseñada para garantizar un alto grado de tolerancia a fallos, preservando la consistencia y disponibilidad mínima requerida incluso ante la desconexión simultánea de nodos. Para ello, todos los datos —tanto archivos como metadatos— se encuentran replicados en un número suficiente de nodos de manera que la pérdida de hasta **dos nodos cualesquiera** no comprometa la integridad del sistema ni provoque pérdida irreversible de información.

7.1. Detección de Fallos mediante Heartbeats

El líder mantiene comunicación periódica con los nodos seguidores a través de *heartbeats*. Cuando uno de estos mensajes no es recibido dentro del intervalo esperado, el nodo se considera *fallido* o desconectado. Esta detección desencadena el proceso interno de recuperación y redistribución gestionado por el servicio **LeaderManager**.

7.2. Recuperación ante la Pérdida de un Seguidor

Cuando un nodo seguidor abandona la red, el *LeaderManager* realiza los siguientes pasos:

1. Identifica todos los **shards** y metadatos que estaban almacenados en el nodo caído, utilizando la información registrada en el log del líder.
2. Localiza en el resto del clúster las réplicas actualizadas de cada shard.
3. Selecciona uno o varios nodos destino que cumplirán la función de nueva réplica, manteniendo la invariante del sistema: **cada shard debe estar replicado en al menos K nodos**.
4. Replica los shards necesarios mediante RPC hacia los nodos seleccionados.
5. Actualiza el log del líder para reflejar la nueva distribución de almacenamiento.

Este proceso asegura que la caída de un seguidor se trate como un evento recuperable y que la resiliencia del sistema permanezca intacta.

7.3. Recuperación ante la Pérdida del Líder

La pérdida del líder constituye un caso especial dentro del sistema. Ante esta situación, Raft inicia de manera automática un proceso de **elección** para seleccionar un nuevo líder. Una vez elegido, el nuevo líder ya posee, mediante el log replicado, toda la información necesaria para continuar coordinando el clúster:

- La ubicación de cada shard.
- Las versiones actualizadas de los archivos.
- Los nodos responsables del almacenamiento de metadatos.

Por lo tanto, desde la perspectiva del sistema, la pérdida del líder equivale a la pérdida de un seguidor en lo que respecta a la reconstrucción de shards. El nuevo líder ejecuta un proceso de **estabilización** para asegurar que la replicación continúe cumpliendo las invariantes establecidas.

7.4. Recuperación de la Base de Datos del Líder

A diferencia de los seguidores, el líder debe almacenar una copia completa y actualizada de la base de datos. Cuando ocurre una elección y un nuevo nodo asume el rol de líder, este debe obtener la versión más reciente de la base de datos desde los nodos que cumplen el rol de **nodos Base de**

Datos. Cada uno de estos nodos mantiene una copia local cuyo estado está asociado a un **término** que sirve como versión global.

El proceso consiste en:

1. Identificar entre los nodos Base de Datos aquel con el **término más alto**.
2. Solicitar mediante RPC la transferencia completa de la base de datos.
3. Reemplazar la base local con la copia recibida.

Una vez concluido, el nuevo líder se encuentra completamente sincronizado y listo para retomar las operaciones de escritura y coordinación del clúster.

7.5. Mantenimiento de la Invariante del Sistema

Tanto la pérdida de seguidores como la pérdida del líder son gestionadas mediante mecanismos automáticos que garantizan la preservación de la siguiente invariante fundamental:

Todos los datos del sistema deben existir siempre en al menos K nodos actualizados, aun en presencia de fallos simultáneos de hasta dos nodos.

Esta propiedad asegura que el sistema puede continuar operando sin pérdida de datos y que la recuperación es siempre posible mediante las réplicas restantes, proporcionando tolerancia a fallos de grado 2.

8. Seguridad

El sistema incorpora un conjunto reducido pero efectivo de mecanismos de seguridad, seleccionados con el objetivo de proteger la integridad y confidencialidad de los datos sin añadir una complejidad innecesaria al diseño. Las medidas aplicadas corresponden a controles simples, ampliamente soportados y suficientes para el modelo de despliegue utilizado.

8.1. Cifrado de la Comunicación mediante TLS

Todas las comunicaciones entre el cliente y la API REST deben realizarse mediante **HTTPS**. El uso de TLS garantiza:

- Confidencialidad de los datos transmitidos.
- Integridad frente a modificaciones en tránsito.

- Autenticidad del servidor ante el cliente.

Para ambientes de producción se recomienda el uso de certificados emitidos por autoridades confiables (p. ej., Let's Encrypt) y la deshabilitación de versiones obsoletas del protocolo TLS.

8.2. CORS y Protección CSRF

Dado que el cliente principal del sistema es una aplicación web, se habilitan dos mecanismos esenciales:

- **Política CORS restrictiva:** únicamente se permiten solicitudes desde el dominio autorizado del cliente. Esto evita que aplicaciones externas interactúen con la API sin autorización.
- **Protección CSRF:** las operaciones que modifican estado requieren tokens CSRF válidos, evitando ataques que intenten forzar acciones desde navegadores del usuario.

Ambas medidas resultan fundamentales para proteger la API expuesta de ataques comunes desde el contexto del navegador.

8.3. Validación y Saneamiento de Entradas

Para evitar vulnerabilidades internas derivadas de datos malformados o manipulados, el sistema aplica validación estricta en todas las entradas:

- Los **serializadores** de Django REST Framework actúan como primera línea de defensa, verificando tipos, formatos y rangos permitidos.
- Se realiza **escape de caracteres peligrosos** y se restringen rutas de archivos y parámetros sensibles.
- Se evita de forma explícita cualquier tipo de *path traversal*, inyección SQL o manipulación no autorizada de identificadores.

Este enfoque garantiza que tanto la API como los servicios internos operen únicamente sobre datos válidos y controlados.

8.4. Seguridad en la Comunicación Interna entre Nodos

Aunque en sistemas distribuidos es habitual asegurar la comunicación interna mediante mecanismos como autenticación mutua o cifrado adicional, en este caso el clúster se ejecuta sobre una **red interna de Docker Swarm**, específicamente una red *overlay*. En este tipo de red:

- Los puertos utilizados por el servicio RPC **no son accesibles desde el exterior**.
- Únicamente los contenedores pertenecientes al servicio tienen visibilidad de dichos puertos.
- El aislamiento es gestionado directamente por Docker, impidiendo que agentes externos puedan comunicarse con el sistema interno.

Por este motivo, añadir mecanismos complejos de autenticación o cifrado en la capa RPC **no aporta beneficios reales en este contexto**, dado que los nodos se encuentran completamente aislados y solo pueden comunicarse dentro de la red virtual controlada por el orquestador.

Conclusiones

El desarrollo del sistema *SpotifyClone* ha permitido integrar de manera efectiva múltiples conceptos fundamentales de los sistemas distribuidos, desde la coordinación entre nodos hasta la replicación consistente de datos y la tolerancia a fallos. A lo largo del diseño, se han priorizado la robustez, la coherencia del estado global y la simplicidad operativa, resultando en una arquitectura sólida y adecuada para cargas dominadas por operaciones de lectura.

En primer lugar, la utilización del protocolo **Raft** como mecanismo de consenso ha demostrado ser una solución eficaz para gestionar la elección de líder, mantener la consistencia del estado replicado y asegurar un comportamiento predecible ante fallos. Su integración con la capa de almacenamiento distribuido permite que el sistema preserve un orden global de operaciones basado en términos, lo que facilita el control de versiones y la detección de nodos desactualizados.

Asimismo, la arquitectura basada en un modelo *leader-workers* ha permitido separar con claridad las responsabilidades dentro del clúster. El líder coordina las escrituras y gestiona la replicación, mientras que los trabajadores atienden las operaciones de lectura, logrando un equilibrio entre consistencia fuerte y escalabilidad horizontal. Este modelo, combinado con el uso de hilos para tareas internas y una API síncrona para la atención a clientes, ha permitido mantener un diseño simple sin sacrificar rendimiento en un entorno mayoritariamente I/O.

En cuanto a la **tolerancia a fallos**, el sistema garantiza que la pérdida de hasta dos nodos no derive en pérdida de información, gracias a una replicación estratégica tanto de shards como de metadatos. El *LeaderManager* desempeña un papel crucial en la reconfiguración automática del

clúster, asegurando la preservación de la invariante de replicación y permitiendo una recuperación transparente ante fallos del líder o de seguidores.

El sistema también incorpora medidas de seguridad prácticas, enfocadas en la protección del tráfico externo mediante **TLS**, el control de acceso desde el cliente a través de **CORS y CSRF**, y una validación rigurosa de entradas en la API. A nivel interno, la seguridad se apoya en el aislamiento natural de la red *overlay* de Docker Swarm, lo que elimina la necesidad de mecanismos adicionales en la comunicación RPC entre nodos.

En conjunto, el proyecto logra un balance adecuado entre simplicidad, confiabilidad y rigor técnico. Si bien existen posibles mejoras futuras —como ampliar la concurrencia interna, optimizar el balanceo de carga o explorar replicación más sofisticada—, la arquitectura presentada constituye una base sólida, bien estructurada y capaz de responder de forma consistente ante fallos, manteniendo siempre la integridad y disponibilidad del sistema distribuido.