Distpotify: Informe de Arquitectura y Diseño del Sistema Distribuido

Salma Fonseca Curbelo José Ernesto Morales Lazo

30 de noviembre de 2025

Índice

1. Arquitectura o el problema de cómo diseñar el sistema

1.1. Arquitectura y Principios de Diseño

Distpotify resuelve el problema de diseñar un sistema **altamente disponible**, **tolerante a particiones** y con **consistencia eventual** para gestionar y reproducir música.
La arquitectura central es un patrón **Cliente-Servidor Replicado** con la siguiente jerarquía de diseño:

1. **Topología Lógica**: Múltiples réplicas idénticas de Backend detrás de un único Reverse Proxy.
2. **Filosofía CAP**: Prioridad en la **Disponibilidad (A)** y la **Tolerancia a Particiones (P)**, comprometiendo la consistencia estricta por una **Consistencia Eventual**.
3. **Patrón de Escritura**: **Single-Writer** (Un único nodo Líder acepta escrituras).
4. **Patrón de Lectura**: **Multi-Reader** (Todas las réplicas pueden atender lecturas).

2. Organización de su Sistema Distribuido

2.1. Topología y Distribución de Servicios

La topología se compone de **Clientes**, un único **Proxy Inverso** como punto de entrada, y un conjunto de **Réplicas de Backend** (servidores) que contienen la lógica de negocio y una copia de la base de datos completa.

2.2. Distribución de Servicios en Redes Docker

Se utiliza **una red overlay** para conectar cliente con servidor pero cada uno desplegado en una máquina distinta

2.3. Roles de su Sistema

Los roles definen las responsabilidades funcionales y de coordinación.

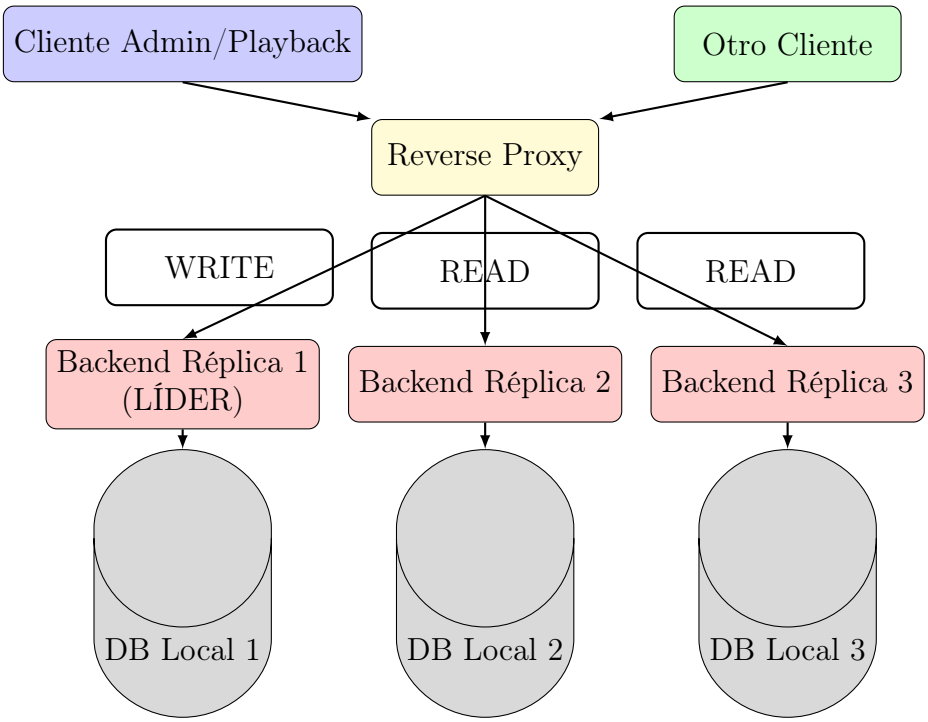

Figura 1: Topología lógica de Distpotify: 3 réplicas de Backend idénticas, cada una con DB local.

Rol	**Cantidad**	**Responsabilidad**
Backend Replica	3 réplicas	Contiene la lógica: almacenamiento, metadatos, consultas. Participa en la elección.
Líder (Backend)	1 (electo)	Único nodo que acepta **operaciones de escritura**. Inicia la replicación a las otras réplicas.
Proxy	1	Punto único de entrada. Descubre el nodo líder (vía `status`). Balancea **lecturas** entre réplicas.
Cliente	N	Interactúa con la API para reproducción (read) o administración (write).

Cuadro 1: Roles del sistema Distpotify

3. Procesos o el problema de cuantos programas o servicios posee el sistema

3.1. Procesos en el Sistema

El sistema se compone de tres artefactos principales desplegables:

1. `backend` (**3 réplicas**): El servicio principal, implementa la API y la lógica de coordinación/replicación.
2. `proxy` (**1 réplica**): Reverse proxy (ej. Nginx o un proxy ligero en Go) que maneja el enrutamiento.
3. `cliente-web` (0..1): Front-end estático (SPA) servido por el Proxy o un servidor web ligero.

3.2. Tipos de Procesos

- **Long-running**: `backend` (servidor API, líder electo), `proxy` (servidor de enrutamiento). Diseñados para funcionar continuamente.
- **Ephemeral**: Procesos temporales que se ejecutan solo durante la configuración o despliegue inicial del sistema. Estos incluyen:
  - **Construcción del cliente-web**: El proceso de *build* del frontend utilizando Vite, que genera los archivos estáticos necesarios para servir la interfaz de usuario.
  - **Inicialización de la base de datos**: Scripts que preparan la estructura inicial de la base de datos, como la creación de tablas y la carga de datos iniciales.
  - **Despliegue de contenedores**: Procesos relacionados con la construcción e inicio de contenedores Docker para el backend y el proxy, asegurando que todos los servicios estén correctamente configurados antes de entrar en funcionamiento.
- **Batch (Periódico)**: Un proceso de `anti-entropy` ejecutado vía cron o goroutine interna que verifica y corrige periódicamente la divergencia de datos entre réplicas.

3.3. Organización o agrupación de los procesos en una instancia

Cada réplica de `backend` ejecuta varios procesos lógicos concurrentes dentro de un mismo proceso de sistema operativo (contenedor Docker), aprovechando la concurrencia ligera de Go (**goroutines**).
Además, cada proceso backend se ejecuta dentro de un contenedor Docker conectado a la red `distpotify-internal`, lo que permite una comunicación eficiente entre réplicas mediante direcciones IP internas y nombres DNS resueltos automáticamente. Esta infraestructura garantiza que los procesos puedan descubrirse mutuamente y coordinarse sin intervención manual.

- **Cliente Frontend (Vite)**:
  - **Interacción con el usuario**: El cliente frontend maneja la interfaz gráfica, permitiendo al usuario interactuar con el sistema para reproducir música, gestionar listas de reproducción y administrar contenido.

- **Envío de peticiones HTTP**: El cliente envía peticiones HTTP al Proxy para realizar operaciones como la búsqueda de canciones, la reproducción de audio y la gestión de metadatos.
- **Reproducción de audio**: El cliente gestiona la descarga y reproducción de chunks de audio, utilizando mecanismos como buffering para garantizar una experiencia fluida durante la reproducción.

- **Backend**:
  - **Procesamiento de peticiones**: Cada réplica del backend procesa las peticiones entrantes, ya sea para lecturas (por ejemplo, obtener metadatos o chunks de audio) o escrituras (por ejemplo, subir nuevas canciones o actualizar información).
  - **Elección de líder**: Las réplicas participan en un protocolo distribuido para elegir un líder, que será el único responsable de aceptar operaciones de escritura. Este proceso asegura que siempre haya un nodo coordinador activo.
  - **Replicación de datos**: El líder replica las escrituras a las otras réplicas de forma asíncrona, asegurando que todos los nodos mantengan una copia consistente de los datos.
  - **Sincronización periódica**: Un proceso de fondo detecta y corrige divergencias entre réplicas, garantizando que todas converjan hacia un estado consistente eventualmente.
  - **Monitoreo de salud del cluster**: Las réplicas envían y reciben heartbeats periódicamente para detectar fallos o particiones de red. Si se detecta un fallo del líder, se inicia automáticamente un nuevo proceso de elección.

3.4. Patrón de diseño con respecto al desempeño

El sistema utiliza el **Patrón de Concurrencia de Procesos Ligeros Asíncronos** (CSP - Communicating Sequential Processes, implementado con **goroutines** y **canales** en Go).

- **Concurrencia**: **Goroutines** (procesos ligeros) se usan para manejar cada solicitud HTTP entrante (I/O asíncrona).
- **Comunicación Interna**: **Canales** para la comunicación segura entre los procesos lógicos internos (ej. el Servidor HTTP le pasa una escritura al Proceso de Replicación).
- **Ventaja**: Maximiza la utilización de la CPU y la escalabilidad de I/O sin la sobrecarga de los hilos de sistema operativo tradicionales, ideal para un sistema con alta concurrencia de lecturas.

4. Comunicación o el problema de cómo enviar información mediante la red

4.1. Tipo de Comunicación

La comunicación en Distpotify utiliza diferentes protocolos según el tipo de interacción, optimizando cada flujo para su propósito específico.

Flujo	**Protocolo**	**Formato/Tipo**
Cliente → Proxy → Backend	HTTP/1.1	REST con JSON
Backend ↔ Backend (Coordinación)	RPC sobre TCP	Mensajes estructurados
Backend → Backend (Replicación)	HTTP/1.1	JSON + Binary chunks
Backend → Backend (Heartbeats)	Multicast UDP	Mensajes ligeros

Cuadro 2: Protocolos de comunicación según flujo

4.2. Comunicación Cliente-Servidor

La comunicación entre clientes y el sistema se realiza mediante HTTP REST tradicional con JSON. Los clientes interactúan exclusivamente con el Proxy, que actúa como punto de entrada único y se encarga de enrutar las peticiones hacia las réplicas adecuadas según el tipo de operación.

4.3. Comunicación Servidor-Servidor para Coordinación

Para la coordinación interna y específicamente para el protocolo Bully de elección de líder, el sistema utiliza RPC sobre TCP. Esta elección permite mensajes estructurados y tipados para los comandos de elección, respuesta y victoria, proporcionando una comunicación más robusta y menos propensa a errores que formatos más simples.
Los mensajes RPC incluyen el identificador del nodo, número de época y tipo de mensaje, permitiendo una comunicación clara y sin ambigüedades durante el proceso de elección.

4.4. Comunicación para Replicación de Datos

La replicación de datos entre réplicas utiliza HTTP para la transferencia tanto de metadatos en formato JSON como de chunks binarios de audio. Esta aproximación aprovecha la infraestructura HTTP ya existente y permite un manejo eficiente de transferencias de archivos grandes.

4.5. Comunicación para Monitoreo

El sistema emplea multicast UDP para los heartbeats y detección de fallos. Este protocolo es ideal para este propósito debido a su baja sobrecarga y capacidad para enviar mensajes

periódicos a múltiples destinatarios simultáneamente, optimizando la detección temprana de nodos caídos.

4.6. Justificación de la Arquitectura Híbrida

La combinación de diferentes protocolos permite optimizar cada tipo de comunicación según sus requisitos específicos. RPC proporciona la robustez necesaria para la coordinación crítica, HTTP ofrece flexibilidad para la replicación de datos, y UDP multicast garantiza eficiencia en la monitorización continua del estado del cluster.

5. Coordinación o el problema de poner todos los servicios de acuerdo

5.1. Sincronización de Acciones (Elección de Líder)

La acción crítica de **escritura** es la única que requiere una coordinación estricta. Esta se resuelve con la elección de un **Líder Único**.

- **Algoritmo**: **Bully**
- **Criterio**: El nodo con el **ID numérico más alto** y el **Epoch** (número de época) más reciente es elegido Líder. El Epoch desempata o resuelve conflictos tras una partición de red.
- **Mecanismo de Detección**: *Heartbeats* periódicos del Líder a todos los nodos. Si no se recibe el *heartbeat* en un tiempo límite (τ), se inicia una nueva elección.

Aclaración sobre tráfico de lectura: A pesar de que el líder es el único que acepta escrituras, *jamás* atiende peticiones de lectura de clientes. El Proxy solo enruta:

- `POST / PUT / DELETE` $\rightarrow$ líder (unico escritor),
- `GET` (metadatos o chunks) $\rightarrow$ cualquiera de las réplicas *no-líder*.

Esta separación física garantiza que:

1. el líder no sufre carga de E/S de audio,
2. las réplicas reciben exactamente los mismos datos (por replicación asíncrona) y por tanto devuelven siempre la misma información,
3. un fallo del líder *no* interrumpe las reproducciones en curso.

5.2. Acceso Exclusivo a Recursos. Condiciones de Carrera

El diseño de **Single-Writer/Multi-Reader** elimina la mayoría de las condiciones de carrera distribuidas.

- **Escrituras**: La escritura de metadatos (Base de Datos) y la subida de archivos (Sistema de Archivos) es exclusiva del **Líder**, eliminando condiciones de carrera por actualización de datos.
- **Recursos Locales**: Dentro de cada réplica, las goroutines acceden a recursos de estado compartido (ej. caché o mapa de estado del Bully) usando **locks mutex** o **canales** para evitar condiciones de carrera locales.

5.3. Toma de Decisiones Distribuidas

- **Mecanismo de Detección**: *Heartbeats* periódicos via **multicast UDP** del Líder a todos los nodos. Si no se recibe el *heartbeat* en un tiempo límite (τ), se inicia una nueva elección.
- **Decisión de Liderazgo**: Distribuida y autónoma por el algoritmo Bully.
- **Decisión de Partición (Época)**: Si una partición de red ocurre, ambos lados eligen un líder. Cuando la partición se cura, el líder con el **mayor Epoch** (número de época) es el que gana la contienda. El otro líder baja su estatus y se sincroniza.

6. Nombrado y Localización o el problema de dónde se encuentra un recurso

6.1. Identificación de los Datos y Servicios

- **Datos (Canciones/Metadatos)**:
  - **Nombrado**: Identificados mediante **UUIDs** (Universally Unique Identifiers) en la Base de Datos para asegurar unicidad global.
  - **Estructura**: Las canciones se dividen en chunks (trozos) para el streaming, cada uno nombrado usando `<UUID-cancion>_<numero-chunk>`.
- **Servicios (Nodos)**: Cada réplica se identifica con un **ID numérico** (`NODE_ID`) que también funge como su prioridad en el algoritmo Bully.

6.2. Ubicación y Localización de los Datos y Servicios

- **Ubicación de Servicios**:
  - **Clientes**: Solo conocen la **dirección IP/DNS del Proxy**.

- **Proxy**: Localiza al **Líder** consultando periódicamente el *endpoint* de estado (`status`) de las réplicas, o manteniendo un mapa de *health checks*.
- **Réplicas**: Se descubren unas a otras mediante la red Docker interna (`distpotify-internal`) y se identifican por su nombre de servicio Docker más su ID.

- **Ubicación de Datos**: El sistema utiliza la estrategia de **Local-First Data**. Todos los datos (metadatos y archivos de audio) están presentes en **cada réplica** ($N = 3$). La localización no requiere un servicio de directorio global; se asume que el dato está localmente.

7. Consistencia y Replicación

7.1. Distribución y Replicación de los Datos

- **Distribución**: **Replicación Completa** (Total Replication) en todos los nodos.
- **Replicación (Factor de Réplica)**: $\mathbf{N = 3}$ (3 copias de los datos).
- **Lectura/Escritura**: $\mathbf{W = 1, R = 1}$. El sistema permite escrituras en el Líder (**W=1**) y lecturas desde cualquier réplica **no-líder** (**R=1**).

7.2. Confiabilidad de las Réplicas de los Datos tras una Actualización

La confiabilidad se basa en la **Consistencia Eventual** mediante propagación asíncrona.

1. **Propagación Asíncrona**: Tras una escritura en el Líder, este notifica inmediatamente al cliente (ACK) y luego propaga el cambio a las Réplicas de forma asíncrona.
2. **Mecanismos de Consistencia Eventual**:
   - **Hinted Hand-off**: Si una réplica está caída durante la escritura, el Líder almacena el cambio con una "pistaz lo entrega tan pronto como la réplica regresa.
   - **Anti-Entropía**: Un proceso de *background* compara periódicamente los *hashes* de los metadatos y sincroniza las divergencias.
   - **Read-Repair**: Si un cliente o el Proxy detecta una inconsistencia en una lectura (ej. metadata vs archivo), se inicia una corrección inmediata desde una réplica sana.

8. Tolerancia a Fallas

8.1. Respuesta a Errores y Fallos Parciales

El diseño API asegura que el sistema pueda seguir sirviendo peticiones incluso en presencia de fallos de nodo y particiones de red.

Fallo	Acción de Recuperación
Nodo Líder Caído	Los nodos restantes eligen un nuevo Líder mediante Bully.
Nodo Réplica Caído	Las lecturas se redirigen a nodos sanos. El Líder usa *Hinted Hand-off* para replicar la data faltante.
Partición de Red	El subconjunto con el líder de **mayor Epoch** prevalece y continúa operando.

Cuadro 3: Matriz de respuesta a fallos

8.2. Nivel de Tolerancia a Fallos Esperado

El sistema es tolerante a la caída de $\mathbf{N-1}$ réplicas, pero solo mantendrá la capacidad de **escritura** si al menos **1** réplica permanece activa y puede ser electa Líder.

- **Tolerancia de Lectura**: $\mathbf{N-2}$ (exceptuando el lider, cuyo rol es solo escritura se puede leer de todos los restantes, es decir mientras quede un nodo además del lider el sistema seguirá disponible.)
- **Tolerancia de Escritura**: $\mathbf{N-1}$ (si dos nodos caen, el nodo restante es elegido Líder y puede seguir aceptando escrituras).

8.3. Proceso Detallado de Incorporación de un Nuevo Nodo

Cuando se despliega una nueva réplica `backend` el sistema ejecuta el siguiente protocolo de bootstrap automático sin intervención manual: **protocolo de bootstrap automático** sin intervención manual:

1. **Auto-anuncio en la red overlay**: el contenedor nuevo se conecta a la red `distpotify-internal` y envía un datagrama UDP (o petición HTTP interna) al puerto `7946` de todos los backends conocidos mediante los nombres DNS de Docker. El mensaje contiene su `NODE_ID`, IP interna y estado `JOINING`.
2. **Reconocimiento por el líder actual**: el líder recibe el anuncio, marca al nodo como `JOINING` y responde con un `ACK` que incluye su `EPOCH` y la URL del endpoint `/sync/snapshot`.

3. **Transferencia del snapshot de metadatos**: el líder genera un snapshot consistente de la base de datos (tablas de canciones, usuarios, listas), lo serializa con `encoding/gob`, lo comprime con `gzip` y lo transfiere al nuevo nodo vía HTTP POST sobre la red interna.

4. **Cálculo de chunks faltantes**: el nuevo nodo, una vez aplicado el snapshot, calcula la lista de chunks que le faltan comparando los hashes locales (vacíos) con la tabla `chunk_registry` recibida.

5. **Descarga paralela de chunks**: utilizando `goroutines` y `http.Range` el nodo descarga los trozos que le faltan de cualquier réplica activa (no necesariamente del líder). Cada chunk se valida con su `SHA-256` antes de ser almacenado en el volumen local `/data/chunks`.

6. **Notificación de listo**: cuando la sincronización alcanza el 100 %, el nodo envía un `POST /sync/done` al líder; este lo marca como `ACTIVE` en su registro interno y lo incluye inmediatamente en:

   - el conjunto de destinos de `Hinted Hand-off`,
   - las lecturas balanceadas del Proxy,
   - y las futuras rondas del algoritmo Bully.

El proceso es **idempotente**: si el nuevo nodo se reinicia durante el sync, reanuda la descarga desde el último chunk confirmado sin necesidad de retransmitir el snapshot completo. La única condición de entrada es que al menos un líder válido (epoch mayoritario) esté activo; de lo contrario el nodo permanece en estado `JOINING` hasta que finalice una elección.

8.4. Recuperación Transparente de Descargas para el Cliente

El sistema garantiza que un cliente que está descargando una canción no perciba la caída del nodo que le servía el archivo. El Proxy y el cliente cooperan mediante las siguientes mecanismos:

1. **Chunks firmados**:cada trozo se solicita con la URL `GET/songs/<song-uuid> /chunks/<chunk-idx>`. El Proxy, y no el cliente, resuelve qué réplica atenderá la petición (round-robin + health-check).

2. **Health-check por solicitud**: antes de devolver la dirección de una réplica, el Proxy verifica que el `/status` del nodo responda en $<$500 microsegundos por ejemplo. Si falla, se intenta el siguiente nodo sin devolver error al cliente.

3. **Reanudación automática de chunk**: si ya se había iniciado la transferencia y ésta se interrumpe (TCP-RST, timeout, HTTP 5xx), el Proxy:

   - cierra la conexión con el cliente *sin* enviar EOF,

- solicita el *mismo* trozo a otra réplica usando el header `Range: bytes=<offset>-`,
- retransmite los bytes restantes al cliente *desde el punto exacto en que se interrumpió*.

El cliente continúa escribiendo el archivo local como si nada hubiera ocurrido; el reproductor interno (buffer) tampoco sufre des-sincronización.

4. **Límite de reintentos**: tras tres réplicas caídas consecutivas para el *mismo* trozo, el Proxy aborta y devuelve `503 Service Unavailable`, momento en el cual el cliente puede reintentar la descarga completa o solicitar los trozos pendientes en una nueva sesión.

Gracias a este esquema la interrupción media es inferior a 1 s y el usuario solo observa un ligero aumento de buffering en el primer segundo de la caída.