Resumen de diseño - Sistema de Archivos Basado en Tags

1. Arquitectura

El sistema es un **filesystem distribuido basado en etiquetas** diseñado con una **arquitectura orientada a servicios (SOA) de estilo REST**, organizada en capas (cliente, lógica de negocio y almacenamiento de datos) y desplegada con **contenedores Docker** sobre una **overlay network attachable** de Docker Swarm (`dfs-net`).

Servicios principales:

- **CLI (capa de presentación)**: interfaz de usuario que corre como contenedor, monta un directorio local para subir/descargar archivos y se comunica con el sistema mediante una API REST expuesta por los Controllers.
- **Controllers (servicio de metadata / capa de lógica)**: proporcionan un servicio de metadata distribuido (usuarios, archivos, tags, ubicación de chunks). Existen múltiples instancias simétricas que replican su estado mediante **gossip protocol** y **vector clocks**, ofreciendo **eventual consistency** y alta disponibilidad. Además de la API REST hacia la CLI, utilizan **gRPC** para comunicarse con los Chunkservers en las operaciones de lectura/escritura de chunks, gestión de réplicas y recepción de heartbeats/estadísticas.
- **Chunkservers (servicio de almacenamiento / capa de datos)**: gestionan el almacenamiento físico de los **chunks** de los archivos (tamaño fijo, p.ej. 4 MB) y mantienen varias réplicas por chunk para tolerancia a fallos. Entre sí utilizan **gRPC streaming** para la **replicación asíncrona de chunks**, coordinada a partir de las decisiones de los Controllers.

El descubrimiento de servicios se realiza mediante **Docker DNS y network aliases** (`controller` , `chunkserver`), y el diseño se clasifica como **AP en el teorema CAP**, priorizando disponibilidad y tolerancia a particiones frente a consistencia fuerte.

---

2. Procesos

Tipos de procesos:

- **Controller Process**
  - Servidor **FastAPI** (HTTP/REST).
  - Módulo de **gossip** para replicar metadata entre controllers.

    - Hilos en background para: monitoreo de Chunkservers, resolución de conflictos y tareas de mantenimiento.
- **Chunkserver Process**
    - Servidor **gRPC** para lectura/escritura/replicación de chunks.
    - Hilos en background para heartbeats, replicación asíncrona y actualización del índice local de chunks.
- **CLI Process**
    - Contenedor interactivo que ejecuta comandos (`register` , `login` , `add` , `list` , `delete` , `add-tags` , `delete-tags`).
    - Usa HTTP/REST hacia cualquier Controller (descubierto por DNS).

Patrones de diseño respecto al rendimiento:

- Controllers: **async/await** con FastAPI, más hilos para gossip y mantenimiento.
- Chunkservers: **gRPC streaming** + hilos para replicación y heartbeats.
- CLI: cliente síncrono, simple request/response, pero con reintentos y balanceo simple vía DNS.

---

3. Comunicación

Tipos de comunicación:

- **CLI ↔ Controller**:
    - **HTTP/REST** sobre la overlay network.
    - Operaciones de autenticación y gestión de archivos (subida, búsqueda por tags, borrado, gestión de tags).
    - La CLI obtiene la lista de Controllers mediante DNS (`controller`) y hace **load balancing del lado cliente** (round-robin / aleatorio).
- **Controller ↔ Controller (gossip)**:
    - Comunicación periódica para **replicar metadata** y asegurar **eventual consistency**.
    - Se intercambian actualizaciones con **vector clocks** y, opcionalmente, resúmenes tipo **Merkle tree** para detectar diferencias.
- **Controller ↔ Chunkserver**:
    - **gRPC** para operaciones sobre chunks: `WriteChunk` , `ReadChunk` , `DeleteChunk` , `GetStats` .
    - Los Controllers eligen Chunkservers desde el alias DNS `chunkserver` y mantienen pools de conexiones.
- **Chunkserver ↔ Chunkserver**:

- **gRPC streaming** para replicación asíncrona de chunks (`ReplicateChunk`).
- Permite replicar datos en background sin bloquear la respuesta al cliente.

Dentro de cada proceso, la coordinación se hace por **memoria compartida + hilos**, no hay mecanismos extra de IPC locales.

---

4. Coordinación

La **coordinación global** se basa en:

- **Gossip protocol** entre Controllers para replicar y reconciliar metadata.
- **Vector clocks** para detectar actualizaciones concurrentes y decidir qué versión conservar (típicamente **last-write-wins**), mientras que para tags se hace un **merge** (unión de conjuntos).

Sincronización de acciones:

- En Controllers:
  - Cada Controller aplica escrituras en su base SQLite local y luego propaga cambios por gossip.
  - No hay leader: cualquiera acepta lecturas y escrituras, y la convergencia es eventual.
- En Chunkservers:
  - Cada escritura de chunk es **atómica a nivel local** (se verifica checksum y se escribe el archivo completo).
  - La replicación a otras réplicas es **asíncrona** y se reintenta en background.

Acceso exclusivo / race conditions:

- Cada chunk se escribe una sola vez (modelo inmutable), lo que simplifica la coherencia.
- La metadata se protege con transacciones SQLite y con la lógica de vector clocks para resolver conflictos.

Toma de decisiones distribuida:

- Colocación de chunks, disparo de re-replicación y resolución de ciertos conflictos se deciden de forma **local por cada Controller**, basándose en la información parcialmente replicada (vista local + gossip).

---

5. Nombrado y localización

Identificación:

- **Archivos**:
  - `file_id` (UUID), `name` (texto), `tags` (lista de strings), `owner_id`.
- **Chunks**:
  - `chunk_id` (UUID) y `(file_id, chunk_index)` para ordenarlos.
- **Servicios**:
  - `controller` y `chunkserver` como aliases DNS en la red Docker.

Localización de servicios:

- Dentro de la overlay network, la resolución `controller` y `chunkserver` devuelve **todas las instancias**.
- La CLI y los Controllers eligen uno de la lista para cada conexión.

Localización de datos:

- **Metadata** (usuarios, archivos, tags, ubicación de chunks):
  - Guardada en una base SQLite en cada Controller y replicada vía gossip.
- **Chunks**:
  - Almacenados en disco en varios Chunkservers (`/data/chunks/<chunk_id>.chk`).
  - Cada Chunkserver mantiene un índice local JSON para saber qué chunks posee.

Descubrimiento:

- **Service discovery**: siempre mediante Docker DNS (no hay Zookeeper, etcd, etc.).
- **Data discovery**:
  - Búsqueda de archivos por tags y usuario propietario usando SQL sobre la metadata local de cada Controller.
  - Selección de Chunkservers para lectura/escritura usando la tabla de colocación de chunks.

---

6. Consistencia y replicación

Distribución de datos:

- Metadata: **replicada en todos los Controllers** (copia completa, eventual consistency).
- Chunks: **particionados** entre Chunkservers con **R réplicas** por chunk (p.ej. R=3).

Replicación:

- **Metadata**:
  - Se replica vía gossip, con vector clocks para detectar qué versión es más reciente o si hay conflictos concurrentes.
- **Chunks**:
  - Replicación **asíncrona best-effort**:
    - El Controller escribe primero en un Chunkserver.
    - Luego ese Chunkserver replica a otros R−1 en background.
  - El valor de R es adaptativo: `R = min(3, número de Chunkservers disponibles)`.

Modelo de consistencia:

- **Metadata**:
  - **Eventual consistency** entre Controllers; lecturas pueden ver un estado ligeramente desfasado.
- **Datos (chunks)**:
  - También **eventual consistency**: una escritura se considera exitosa con 1 réplica, las demás llegan después.
  - Los chunks son inmutables, lo que evita conflictos de “escrituras concurrentes” sobre el mismo chunk.

Fiabilidad tras actualizaciones:

- Checksum (SHA-256) en cada chunk para asegurarse de que las réplicas no estén corruptas.
- Si una réplica falla/verifica mal, puede reemplazarse copiando desde otra réplica sana.
- La metadata de colocación se actualiza y se replica por gossip.

---

7. Tolerancia a fallos

Respuesta ante errores:

- **Fallo de Controller**:
  - Otros Controllers siguen atendiendo requests; la CLI los descubre vía DNS.
  - Gossip re-sincroniza metadata cuando el nodo vuelve.
- **Fallo de Chunkserver**:
  - Controllers detectan la ausencia de heartbeats.
  - Los chunks alojados ahí se marcan como **sub-replicados**.
  - Se dispara **re-replicación** desde réplicas sanas hacia otros Chunkservers.
- **Errores de escritura/lectura de chunk**:

- Si falla un Chunkserver, se reintenta en otra réplica.
- Si el checksum no coincide, se descarta la réplica corrupta y se re-replica.

Nivel esperado de tolerancia:

- Metadata: mientras quede al menos **1 Controller**, el sistema sigue respondiendo (aunque con menos capacidad).
- Datos: mientras quede al menos **1 réplica de cada chunk**, el archivo puede leerse.
- El sistema está pensado para soportar fallos frecuentes de nodos, siempre que no se pierdan todas las réplicas de un chunk.

Fallos parciales y nodos dinámicos:

- **Nodos temporalmente caídos**:
  - Se marcan como fallidos; al volver, sus índices y metadata se reconcilian con el estado actual.
- **Nuevos nodos**:
  - Un nuevo Chunkserver se registra mediante heartbeat y empieza a recibir nuevos chunks y re-replicación.
  - Un nuevo Controller se une a la overlay network, descubre a sus pares vía DNS y sincroniza metadata mediante gossip.

---

8. Seguridad

8.1 Seguridad en la comunicación

Diseño básico:

- Comunicación en **texto claro** dentro de la overlay network (HTTP para la CLI, gRPC para servicios internos).
- Se asume red interna relativamente confiable, pero para producción se recomienda:
- **HTTPS/TLS** para CLI ↔ Controllers.
- **mTLS** entre Controllers y Chunkservers.
- Opcional: cifrado de datos **end-to-end en la CLI** (los servidores sólo ven datos cifrados).

8.2 Seguridad en el diseño

- Mayor número de nodos ⇒ más superficie de ataque (cada Controller y Chunkserver es un punto a proteger).
- Riesgos principales:

- Intercepción de tráfico si no hay TLS.
- Acceso directo a discos de Chunkservers.
- Abuso de recursos (DoS, usuarios llenando el almacenamiento).

Mitigaciones sugeridas:

- Aislamiento de red (solo exponer Controllers necesarios).
- Validación de entradas y consultas parametrizadas en SQLite.
- Cuotas por usuario y limitación de tamaño de uploads.
- Auditoría y monitoreo centralizado.

8.3 Autorización y autenticación

- **Autenticación** basada en **API Key**:
  - `register` crea un usuario con password hash y API Key.
  - `login` genera una **nueva API Key**, invalidando la anterior.
  - La API Key se envía en `Authorization: Bearer <key>` en todas las operaciones de archivos.
- **Autorización** basada en **ownership**:
  - Cada archivo tiene un `owner_id`.
  - Todas las operaciones que leen o modifican archivos verifican que `owner_id` coincida con el usuario asociado a la API Key.
  - No hay acceso a archivos de otros usuarios.