Diseño e Implementación de un Sistema FTP Distribuido

Adrian Hernández Castellanos - C412
Laura Martir Beltrán - C411

Resumen

Este informe presenta el diseño e implementación de un sistema FTP distribuido conforme a la especificación RFC 959. El sistema está desplegado sobre contenedores Docker, donde cada nodo tiene capacidad completa para atender clientes FTP y, simultáneamente, participar del clúster de almacenamiento distribuido.

La arquitectura empleada es Peer-to-Peer estructurada con liderazgo dinámico mediante el algoritmo Bully. El nodo líder centraliza todas las operaciones críticas de coordinación: serialización de escrituras, gestión de locks, decisión de replicación y recuperación ante fallas.

Se adopta un modelo de consistencia eventual controlada: las escrituras se sincronizan de forma estricta y las lecturas pueden responder con versiones recientes aún no propagadas completamente, lo que permite un balance entre rendimiento e integridad.

Se utilizan locks centralizados, replicación de archivos y un sistema de proxy binario para asegurar que cualquier cliente pueda acceder a cualquier recurso del sistema desde cualquier nodo, de forma completamente transparente.

Índice

1. Arquitectura **3**
1.1. Organización General 3
1.2. Componentes por Nodo 3
1.3. Puertos y Canales 4
1.4. Distribución de nodos en los hosts 4

2. Procesos **4**
2.1. Modelo de Ejecución y patrón de diseño 5

3. Comunicación **6**
3.1. Canales 6
3.2. Proxy de Descargas 6

4. Coordinación **6**
4.1. Elección de Líder 7
4.2. Locks 7

5. Localización y nombrado **8**

6. Replicación y Escrituras **9**
6.1. Factor de Replicación 9
6.2. Flujo de STOR 9
6.3. APPE y STOU 10

7. Tolerancia a Fallas **10**
7.1. Respuesta a Errores 11
7.2. Nivel de Tolerancia a Fallos 11
7.3. Manejo de Fallos Parciales y Nodos Dinámicos 12
7.4. Relación con el Modelo de Consistencia 12

8. Seguridad **13**
8.1. Seguridad en la Comunicación 13
8.2. Seguridad desde el Diseño 13
8.3. Autenticación y Autorización 14

9. Principios de Diseño del Sistema **14**

1. Arquitectura

Esta sección describe la estructura general del sistema FTP distribuido, detallando el modelo arquitectónico adoptado, la organización de los nodos y los componentes que los conforman. Se explica cómo la combinación de un esquema peer-to-peer con liderazgo dinámico permite balancear simplicidad, tolerancia a fallas y coordinación eficiente, manteniendo transparencia total para los clientes.

1.1. Organización General

El sistema adopta una arquitectura **Peer-to-Peer estructurada con liderazgo dinámico**. Todos los nodos del clúster son funcionalmente equivalentes, lo que significa que:

- Cada nodo puede recibir conexiones de clientes FTP usando el protocolo estándar.
- Todos mantienen un log local de operaciones que permite reconstruir el estado global del sistema.
- Todos pueden participar de comunicaciones distribuidas utilizando sockets TCP.

Aunque todos los nodos son iguales en capacidades, uno de ellos asume el rol de **Líder**. Este rol se obtiene dinámicamente mediante el algoritmo de elección Bully. El líder cumple un papel fundamental dentro del sistema porque:

- Garantiza que las operaciones de escritura se realicen de manera serializada, evitando inconsistencias.
- Administra los locks distribuidos para asegurar exclusión mutua sobre archivos.
- Decide qué nodos almacenarán cada réplica.
- Coordina los procedimientos de recuperación ante fallas y sincronización.

Centralizar estas funciones permite mantener un diseño simple, confiable y fácilmente verificable sin recurrir a mecanismos de consenso más complejos.

1.2. Componentes por Nodo

Cada nodo ejecuta internamente dos componentes lógicos principales:

1. **FTP Core:** Implementa completamente el protocolo RFC 959, atendiendo a los clientes externos. Se encarga del procesamiento directo de comandos como STOR, RETR, LIST, RNFR, etc.
2. **Sidecar de Gestión:** Servicio interno dedicado exclusivamente a la coordinación del clúster distribuido. Sus responsabilidades principales incluyen:
    - Ejecución del algoritmo Bully para elección de líder.

- Envío y recepción de mensajes RPC.
- Propagación y sincronización del índice global de metadatos.
- Coordinación de transferencias entre nodos durante operaciones de réplica.

Ambos componentes se ejecutan en forma de **hilos dentro de un único proceso Python**. Esta decisión elimina la necesidad de IPC (comunicación interna entre procesos) complejo, favorece la eficiencia en memoria compartida y facilita la sincronización interna.

1.3. Puertos y Canales

La arquitectura establece canales separados de comunicación:

- **Puerto 21**: Canal estándar FTP de control para clientes.
- **Rango PASV**: Transferencias FTP cliente-servidor.
- **Puerto 2123**: Canal RPC dedicado a mensajes de control entre nodos.
- **Puerto 2124**: Canal de transferencia binaria directa entre nodos durante replicaciones.

Esta separación asegura que el tráfico de datos no interfiera con la coordinación de control del sistema.

1.4. Distribución de nodos en los hosts

El proyecto se ejecutará en una red overlay de Docker Swarm a la cual pertenecerán dos hosts reales que contendrán múltiples contenedores (nodos) en ejecución. Se recomienda inicializar el sistema con no menos de 5 nodos para mantener la tolerancia a fallos requerida. Al menos uno de los dos hosts debe ejecutar como mínimo 3 nodos, para garantizar el funcionamiento del sistema ante particionamientos de la red por desconexiones de los hosts. No obstante, si este requisito no se cumple el sistema seguirá funcionando pero puede ocurrir pérdida de información.

2. Procesos

En esta sección se presenta el modelo de ejecución del sistema, enfocándose en la organización interna de los procesos y hilos que operan en cada nodo. Se analiza cómo la concurrencia, el procesamiento asíncrono y la separación de responsabilidades permiten atender múltiples clientes, coordinar el clúster y manejar fallas sin comprometer la consistencia ni el rendimiento.

2.1. Modelo de Ejecución y patrón de diseño

El sistema adopta un patrón de diseño distribuido con arquitectura de procesamiento asíncrono y concurrente basado en hilos dentro de cada nodo. Cada nodo ejecuta un único proceso que cuenta con los siguientes hilos concurrentes:

- **FTP Listener:** Atiende conexiones de clientes FTP, generando hilos por sesión.
- **Cluster RPC Server:** Recibe y procesa mensajes de coordinación entre nodos, tales como solicitudes de lock, información de replicación, propagación de logs de operaciones y reconstrucción del estado global.
- **Discovery-Bully Thread:**
  1. Consulta periódicamente el DNS de Docker para conocer la lista de nodos activos.
  2. Ejecuta rondas del algoritmo Bully.
  3. Verifica la disponibilidad del líder mediante timeouts controlados.

En primer lugar, desde la perspectiva funcional, el protocolo FTP mantiene explícitamente el modelo cliente–servidor, donde cualquier nodo del clúster puede desempeñar el rol de servidor FTP frente a clientes externos (por ejemplo, FileZilla o bibliotecas como ftplib). Los clientes se conectan a un nodo indistintamente, sin conocimiento de la estructura distribuida interna, garantizando así transparencia del sistema.

En el ámbito interno del clúster, sin embargo, el diseño evoluciona hacia un esquema peer-to-peer coordinado por líder, en el cual todos los nodos tienen capacidades equivalentes, pero uno de ellos asume dinámicamente el rol de coordinador mediante el algoritmo de elección Bully. Este líder es responsable de tomar decisiones distribuidas, como la asignación de réplicas, control de locks distribuidos y validación de actualizaciones críticas, mientras que el resto de nodos actúan como ejecutores de las operaciones delegadas.

Este modelo permite detectar rápidamente fallas sin necesidad de procesos más complejos de implementar, como un heartbeat continuo de alta frecuencia, pero aumenta el tiempo de incertidumbre ante una falla de un nodo, por lo que se puede dar el caso de que un mensaje se dirija a un nodo que está caído antes de que el sistema lo reconozca como caído. Estos errores son manejados por el sistema mediante el uso de *timeouts* en todas las comunicaciones RPC. Ante la falta de respuesta de un nodo, la solicitud se aborta y se notifica al módulo de descubrimiento, el cual invalida dicho nodo en la topología activa. Dependiendo del tipo de operación, la petición es reintentada contra otros nodos disponibles o reenviada al líder para recalcular la acción necesaria.

En caso de que el nodo no respondiente sea el líder, el sistema bloquea temporalmente las operaciones de escritura que requieran coordinación centralizada, inicia el proceso de elección Bully y, una vez designado un nuevo líder activo, reintenta automáticamente las operaciones pendientes garantizando que no se ejecuten duplicados ni inconsistencias.

Gracias a este mecanismo, la posible entrega de mensajes a nodos caídos no compromete la integridad del sistema, ya que ningún estado crítico es modificado

hasta recibir confirmaciones válidas, y toda falla es absorbida mediante reintentos controlados, reconfiguración dinámica del clúster y recuperación coordinada.

3. Comunicación

Toda comunicación se implementa mediante **sockets TCP escritos en Python**, evitando dependencias externas innecesarias.

3.1. Canales

1. **RPC:** Mensajes JSON sobre TCP para intercambiar información de control: elecciones, coordinación, locks y sincronización de metadatos.
2. **Raw Data:** Streams binarios sin codificación para transmitir archivos, evitando sobrecostos por serialización.

La comunicación servidor–cliente sigue el estándar RFC959 con una capa de encriptación incorporada que garantiza la confidencialidad e integridad de la información intercambiada. Por otro lado, la comunicación servidor–servidor se realiza mediante sockets TCP directos utilizando un protocolo propio ligero basado en mensajes JSON para el intercambio de metadatos, coordinación de elecciones, gestión de locks y control de replicación, complementado con canales binarios dedicados para la transferencia eficiente de datos sin procesar entre nodos del clúster. Trae como desventaja la falta de seguridad entre mensajes servidor-servidor y como ventaja una implementación sencilla y un sistema de mensajería funcional y rápido. Podemos afrontar la desventaja a través de mecanismos de seguridad para mantener la red interna inaccesible desde el exterior.

No es necesaria la comunicación entre procesos, ya que solo existe un proceso por contenedor, ejecutando múltiples hilos que se comunican entre sí haciendo llamadas RPC solo en casos necesarios.

3.2. Proxy de Descargas

Cuando un nodo recibe una petición RETR y no posee el archivo localmente:

- Solicita autorización al Líder.
- Abre un socket contra algún nodo que tenga la réplica.
- Reenvía directamente los bloques recibidos al cliente.

El archivo nunca es almacenado temporalmente en el nodo intermediario, reduciendo uso de disco y latencia.

4. Coordinación

Esta sección aborda los mecanismos que permiten mantener una visión coherente del sistema distribuido, aun en presencia de múltiples nodos y fallas parciales. Se describen las estrategias de elección de líder, control de locks y toma de decisiones centralizadas que garantizan la serialización de operaciones críticas y la integridad del estado global del sistema.

4.1. Elección de Líder

Se emplea el algoritmo **Bully**, donde:

- El nodo con IP de mayor prioridad disponible es líder.
- Ante pérdida de comunicación detectada mediante timeout se inicia un proceso de elección.
- El nodo electo recopila los logs de todos los nodos, reconstruye el estado global del sistema (estructura de directorios y ubicación de réplicas) y asume inmediatamente la coordinación.

La decisión de realizar las identificaciones de los nodos por IP y no asignar un ID a cada nodo independiente produce simplicidad de implementación del protocolo de elección de líder, aunque no es recomendable normalmente en entornos donde las direcciones IP puedan cambiar constantemente, ya que esto llevaría a realizar múltiples procesos de elección de líder por cambios de IP. Una solución para esto (en caso de que sea requerido) podría ser realizar esta misma implementación sobre nodos con IDs asignadas.

4.2. Locks

Toda operación de escritura requiere obtener previamente un **lock centralizado del líder**.

El líder mantiene:

- Tabla activa de locks por archivo.
- Cola de solicitudes en espera.
- Timeouts automáticos que liberan locks en casos de falla.

El sistema previene bloqueos permanentes mediante timeouts globales: si una operación no se completa dentro del tiempo esperado o el nodo falla, el líder detecta la situación mediante la falta de confirmación y libera automáticamente el lock, permitiendo que otros nodos continúen con sus operaciones. Este esquema evita bloqueos permanentes (deadlocks) y garantiza que una falla parcial no detenga indefinidamente el progreso del sistema, manteniendo la consistencia incluso en presencia de errores o desconexiones inesperadas.

La toma de decisiones distribuidas se implementa mediante un esquema de coordinación centralizada lógica, donde todos los nodos conservan capacidad de operación autónoma, pero delegan aquellas decisiones que afectan al estado global del sistema al nodo líder. Operaciones como la asignación de réplicas, otorgamiento de locks, validación de escrituras y resolución de conflictos de metadatos se solicitan siempre al líder. El líder evalúa el estado reportado por los nodos y emite resoluciones que luego se propagan al resto del clúster mediante difusión de actualizaciones. Este enfoque evita inconsistencias derivadas de decisiones locales contradictorias, mantiene una visión coherente del sistema y permite que el consenso práctico se alcance sin recurrir a protocolos complejos de múltiples rondas, reduciendo la sobrecarga de coordinación mientras se preserva la consistencia operativa del sistema distribuido.

5. Localización y nombrado

Cada nodo mantiene un **log secuencial de operaciones** que registra todas las acciones realizadas sobre archivos y directorios. El líder, al ser elegido, recopila estos logs de todos los nodos y reconstruye el **estado global del sistema** mediante la fusión ordenada por timestamp de todas las operaciones registradas. Esta estructura resuelve el problema fundamental de *nombrado y localización* dentro del sistema distribuido: determinar de manera única qué recurso se solicita, dónde se encuentra almacenado físicamente y cómo acceder a él desde cualquier nodo del clúster.

El estado reconstruido mantiene la siguiente información por cada archivo:

$$Ruta \rightarrow \{tipo, timestamp, size, hash, [ListaNodosRplica]\}$$

Y para cada directorio:

$$Ruta \rightarrow \{tipo, timestamp, [ListaNodosContenedores]\}$$

Desde el punto de vista de la **identificación**, todos los datos se referencian mediante rutas FTP estándar, que actúan como identificadores lógicos únicos del sistema. Estas rutas son independientes de la ubicación física del archivo y permiten que los clientes interactúen con el sistema de forma transparente, sin conocer en qué nodo reside realmente cada recurso.

En cuanto a la **ubicación**, la tabla asocia cada identificador lógico con la lista de nodos que almacenan las réplicas del archivo, logrando un mapeo centralizado a nivel lógico.

Respecto a la **localización**, el líder utiliza localmente esta tabla para resolver peticiones entrantes: al recibir una solicitud de lectura o descarga, identifica de inmediato el nodo origen más adecuado y establece la conexión necesaria.

Esta estructura permite:

- Resolver comandos LIST y NLST consultando al líder, quien posee el estado global reconstruido más actualizado.
- Determinar en tiempo constante qué nodos contienen un archivo específico mediante consulta al estado global del líder.
- Registrar todas las operaciones localmente en el log, permitiendo recuperación ante fallas y reconstrucción consistente del sistema.

Las modificaciones se originan en cualquier nodo que reciba una operación de un cliente, pero deben ser **validadas y coordinadas por el líder** antes de su ejecución. El líder valida la consistencia global, decide esquemas de replicación y coordina la ejecución. Una vez autorizada, la operación se ejecuta en los nodos correspondientes, cada uno registrándola en su log local. Posteriormente, cuando el líder es reelegido o debe reconstruir el estado, fusiona todos los logs ordenándolos por timestamp para obtener una visión consistente y coherente del sistema distribuido.

6. Replicación y Escrituras

Esta sección aborda el problema fundamental de **consistencia y replicación**, es decir, cómo mantener múltiples copias de un mismo dato distribuidas entre diferentes nodos sin comprometer la integridad del sistema, garantizando tanto disponibilidad como tolerancia a fallas, y evitando inconsistencias derivadas de actualizaciones simultáneas o pérdidas parciales de información.

6.1. Factor de Replicación

Cada archivo se almacena en exactamente **3 nodos distintos**. Este factor de réplica representa un balance práctico entre:

- **Confiabilidad**: la pérdida de uno o incluso dos nodos no implica la pérdida del archivo. (Tolerancia a fallos de nivel 2)
- **Costo de almacenamiento**: evitar la sobre-replicación excesiva de datos.
- **Complejidad de coordinación**: a mayor número de copias, mayores desafíos de sincronización.

La **distribución de los datos** se realiza de forma controlada por el líder, quien selecciona los nodos destino considerando:

- Menor ocupación de disco, para balancear el uso global del almacenamiento.
- Disponibilidad activa y estado saludable del nodo, evitando asignar réplicas a componentes inestables.

Este enfoque introduce un reparto progresivo de la carga sin la necesidad de mecanismos sofisticados de rebalanceo automático como podría ser el hashing consistente usado en DHTs. Esto reduce la complejidad de implementación y los costos operativos sin sacrificar los niveles aceptables de disponibilidad y tolerancia a fallas.

6.2. Flujo de STOR

El proceso de carga de archivos se diseña para asegurar la **consistencia entre réplicas** frente a escrituras distribuidas:

1. El nodo que recibe la solicitud STOR actúa como intermediario y solicita al líder un plan de replicación.
2. El líder selecciona de manera determinística los nodos destino responsables de almacenar las copias del nuevo archivo.
3. Los datos se transmiten directamente a cada nodo mediante **Raw Sockets TCP**, evitando sobrecostos de serialización y permitiendo alto rendimiento.
4. Cada nodo confirma la correcta escritura local del archivo.
5. El líder valida las replicaciones antes de considerar la operación como completada.

6. Una vez validada la replicación, el líder actualiza su estado global en memoria y cada nodo que participó en la operación registra la acción en su log local, garantizando persistencia y capacidad de reconstrucción futura del estado.

Este esquema asegura que ninguna operación de escritura quede confirmada mientras el sistema no tenga certeza de que las réplicas están efectivamente persistidas y registradas en los logs locales, aumentando la confiabilidad de los datos tras una actualización y permitiendo recuperación completa ante fallos.

6.3. APPE y STOU

El comando APPE (append) implementa un sistema de **transacciones delta** que propaga únicamente el contenido añadido hacia las réplicas, evitando retransmitir el archivo completo.

El nodo receptor almacena temporalmente el contenido a añadir (delta), consulta al líder para obtener lock de escritura y la lista de réplicas, y luego:

1. Aplica el append sobre su copia local si es réplica.
2. Envía órdenes `APPEND_BLOCK` a las réplicas remotas indicando la ubicación del delta.
3. Cada réplica descarga el delta, aplica el append y confirma mediante `DELTA_CONFIRMATION`.
4. El nodo origen elimina el delta temporal tras recibir todas las confirmaciones o alcanzar un timeout de seguridad (90 segundos).

Este enfoque reduce drásticamente el tráfico de red ($n \times size\_delta$ vs $n \times size\_total$) y garantiza confirmación explícita de cada réplica antes de considerar la operación completada, manteniendo consistencia incluso ante fallos parciales.

El comando STOU (store unique) sigue el mecanismo de STOR, generando automáticamente un nombre único mediante sufijo aleatorio para evitar colisiones con archivos existentes.

7. Tolerancia a Fallas

La tolerancia a fallas constituye uno de los objetivos centrales del sistema distribuido propuesto: el esfuerzo de distribuir datos y servicios carecería de sentido si la caída de una única componente pudiera provocar la interrupción total del servicio o la pérdida de información. Por este motivo, el sistema se diseña bajo el principio de **continuidad operativa ante fallos parciales**, asumiendo que los errores son eventos normales y esperables en un entorno distribuido.

El diseño busca garantizar que cualquier fallo individual —ya sea de un nodo de almacenamiento, del líder coordinador o de una partición temporal de la red— pueda ser absorbido por el sistema sin comprometer la integridad de los datos ni interrumpir completamente el servicio a los clientes.

7.1. Respuesta a Errores

La detección de errores se realiza mediante dos mecanismos complementarios:

- **Discovery periódico**: cada nodo consulta el DNS de Docker para validar la presencia de otros nodos activos.
- **Timeouts de comunicación**: toda interacción RPC o solicitud de coordinación hacia el líder dispone de límites temporales que permiten inferir fallos cuando no se recibe respuesta.

Ante la detección de una falla:

- Si se trata de un nodo de almacenamiento, el líder marca sus réplicas como inactivas y ejecuta procesos de **re-replicación automática** para restaurar el factor de replicación definido.
- Los logs se actualizan para reflejar el nuevo estado del sistema.
- Si el nodo caído es el líder, se dispara inmediatamente un nuevo proceso de elección Bully que permite restaurar la coordinación sin intervención manual.
- Las operaciones que se encontraban en curso son reintentadas automáticamente una vez restablecida la coordinación global y reconstruido el estado general tras la caída del líder.

De esta forma, el sistema no requiere reinicio total ni intervención administrativa para recuperarse de fallos comunes, manteniendo disponibilidad operativa continua.

7.2. Nivel de Tolerancia a Fallos

El sistema está diseñado para tolerar:

- La caída simultánea de hasta **dos nodos de almacenamiento** por archivo sin pérdida de datos, gracias al uso de tres réplicas independientes.
- La pérdida completa del **nodo líder**, la cual se resuelve mediante elección automática sin impacto permanente sobre el servicio.
- **Particiones temporales de red**, en las que segmentos del sistema quedan aislados entre sí.

Durante una partición, cada segmento elegirá su propio líder y operará de forma autónoma bajo el principio de disponibilidad, aceptando escrituras localmente. Al momento de la reunificación:

- Se ejecuta una nueva elección global de liderazgo.
- El líder resultante recopila los logs de operaciones de todos los nodos de ambos segmentos.
- Los logs se fusionan ordenándolos por timestamp para reconstruir el estado global.

- Los conflictos de versiones se resuelven aplicando la política **Last Write Wins (LWW)**: la operación con timestamp más reciente prevalece.
- Se detectan inconsistencias entre el estado reconstruido y el estado físico de cada nodo.

Este enfoque permite priorizar la continuidad de servicio durante particiones, aceptando una consistencia eventual que se restablece automáticamente tras la recuperación de conectividad.

7.3. Manejo de Fallos Parciales y Nodos Dinámicos

El sistema asume fallos parciales como eventos esperados y los maneja de forma transparente:

- **Nodos caídos temporalmente**: cuando un nodo deja de responder es excluido del conjunto activo y sus responsabilidades de almacenamiento son replicadas en otros nodos. Si el nodo se reincorpora posteriormente, el líder solicita su log de operaciones, detecta inconsistencias comparándolo con el estado global y envía comandos de sincronización para descargar archivos faltantes. Una vez sincronizado, el nodo es incorporado paulatinamente en nuevas decisiones de replicación.
- **Nodos nuevos**: al integrarse al clúster, el líder solicita su log de operaciones (vacío si es completamente nuevo), lo integra al estado global reconstruido y envía comandos de sincronización para crear la estructura de directorios necesaria y replicar aquellos archivos que el líder determine necesarios para cumplir con el factor de réplica. El nodo registra todas estas operaciones en su log local.

La ausencia de rebalanceo automático masivo simplifica la operación del sistema y evita sobrecarga innecesaria de red y escritura, manteniendo un equilibrio razonable entre eficiencia, tolerancia a fallas y simplicidad de implementación.

7.4. Relación con el Modelo de Consistencia

El mecanismo de tolerancia a fallas se integra directamente con el modelo de consistencia del sistema, que combina:

- **Escrituras estrictamente serializadas**, coordinadas por el líder mediante locks centrales.
- **Lecturas eventualmente consistentes**, resueltas por el líder.
- **Exclusión mutua por archivo**, garantizada por locks.

Este modelo prioriza la **integridad de los datos por encima de la disponibilidad absoluta**, permitiendo mantener el servicio operativo durante fallos parciales sin comprometer la corrección del estado global cuando el sistema converge nuevamente.

8. Seguridad

El diseño de la seguridad del sistema prioriza la protección de la integridad de los datos, la correcta autorización de las operaciones y la confidencialidad de la información transmitida, evitando introducir mecanismos de seguridad excesivamente complejos que no aporten beneficios proporcionales al nivel de riesgo real.

8.1. Seguridad en la Comunicación

Toda la comunicación **cliente–servidor** sigue el estándar RFC959 con una **capa de cifrado incorporada**, garantizando la confidencialidad e integridad de las transferencias de control y de datos frente a escuchas pasivas o modificaciones en tránsito. De esta manera se previene la exposición de credenciales, comandos FTP y contenido de archivos ante accesos no autorizados dentro de la red.

Por su parte, la comunicación **servidor–servidor** ocurre exclusivamente dentro de la red privada Docker, inaccesible desde el exterior. Dichas transmisiones se realizan mediante sockets TCP y protocolos propios basados en JSON para coordinación, locks, elecciones de líder y sincronización de metadatos, así como flujos binarios directos para replicación de archivos. Aunque esta comunicación interna no se cifra por defecto para reducir sobrecosto computacional, la segmentación de red y el aislamiento proporcionado por Docker constituyen una barrera efectiva frente a accesos externos, disminuyendo significativamente los vectores de ataque.

8.2. Seguridad desde el Diseño

El diseño del sistema incorpora mecanismos que refuerzan la seguridad estructural más allá del cifrado de comunicaciones:

- **Centralización lógica del control**: el rol del líder impide que nodos individuales tomen decisiones que puedan comprometer la integridad global, evitando escrituras inconsistentes o distribución arbitraria de datos.
- **Bloqueo centralizado**: previene condiciones de carrera y elimina la posibilidad de corrupción de archivos por accesos concurrentes no coordinados.
- **Autorización implícita de acciones internas**: únicamente los nodos reconocidos por el proceso de discovery forman parte del clúster operativo, cerrando el acceso a nodos externos no autorizados.
- **Aislamiento de red**: la arquitectura basada en contenedores limita la visibilidad del sistema al interior del entorno de ejecución, reduciendo drásticamente la superficie de ataque.

Estas decisiones arquitectónicas se orientan a proteger la consistencia de los datos y a garantizar que cualquier operación crítica pase por un canal de autoridad controlado, reduciendo riesgos tanto por errores accidentales como por comportamientos maliciosos dentro del sistema.

8.3. Autenticación y Autorización

La **autenticación de clientes** se gestiona conforme al estándar FTP, requiriendo credenciales válidas para el establecimiento de sesiones. Las credenciales son transmitidas bajo la capa cifrada de comunicación, evitando su interceptación en tránsito.

La **autorización** se aplica a nivel de comandos FTP: una vez autenticado, el cliente sólo puede ejecutar operaciones que correspondan a su perfil de permisos definidos en el servidor local, tales como lectura, escritura, modificación o borrado de archivos dentro de rutas específicas. Cada nodo aplica localmente esta política antes de autorizar cualquier operación que implique acceso a datos.

En el plano **interno del clúster**, los nodos no requieren autenticación explícita entre ellos debido al control estricto del entorno Docker y al aislamiento de red privada. El mecanismo de discovery funciona además como una validación implícita de identidad, permitiendo únicamente la comunicación entre instancias previamente definidas dentro del despliegue del sistema. La toma de decisiones críticas —locks, replicación y actualizaciones del esquema global— se valida siempre a través del líder, evitando que nodos no autorizados o en estado inconsistente puedan ejecutar acciones que comprometan la seguridad del sistema.

En conjunto, estos mecanismos establecen un modelo de seguridad proporcional al contexto de uso del sistema, equilibrando confidencialidad, protección de datos, control de acceso y simplicidad operativa sin introducir complejidad adicional innecesaria.