

Sistema Distribuido Master-Slave con Ubicación Semántica

Abel Ponce González C411
Richard Alejandro Matos Arderí C411

December 9, 2025

Abstract

Este documento presenta la especificación arquitectónica de un sistema distribuido de búsqueda de documentos basado en arquitectura Master-Slave con elección dinámica de líder. Se detalla la arquitectura del sistema, la organización y roles de sus componentes, los mecanismos de ubicación de recursos mediante vectorización semántica, las estrategias de tolerancia a fallos con algoritmo Bully para elección de líder, el sistema de replicación por afinidad semántica, y los aspectos de seguridad y comunicación del diseño.

Contents

1	Introducción	4
1.1	Descripción del sistema	4
1.2	Objetivos del diseño	4
2	Arquitectura del Sistema	4
2.1	Estilos arquitectónicos	4
2.1.1	Arquitectura en Capas (Layered Architecture)	4
2.1.2	Arquitectura Publish-Subscribe para Eventos	5
2.1.3	Elementos de arquitectura Peer-to-Peer	5
2.1.4	Estilo Master-Slave con promoción dinámica	6
2.2	Modelo arquitectónico: Master-Slave con elección dinámica	6
2.2.1	Características del sistema	6
2.3	Distribución del sistema	7
2.3.1	Distribución jerárquica Master-Slave sobre Docker Swarm	7
2.4	Topología de red: Estrella con redundancia	8
2.4.1	Definición de la topología	8
2.4.2	Propiedades de la topología Master-Slave	8
3	Roles y organización funcional	9
3.1	Diagrama de roles e interacciones	9
3.2	Responsabilidades de cada rol	10

4	Despliegue y distribución de servicios con Docker Swarm	10
4.1	Arquitectura de redes Docker Swarm	10
4.1.1	Modos de endpoint para Service Discovery	10
4.2	Comandos de despliegue con Docker Swarm	11
4.3	Ventajas del diseño	12
5	Procesos y patrones de diseño	12
5.1	Tipos de procesos en el sistema	13
5.2	Arquitectura interna de un Slave	13
5.3	Patrones de diseño aplicados	13
5.4	Modelo de concurrencia	14
6	Comunicación entre nodos	14
6.1	Modelo de comunicación por capas	14
6.2	Tipos de comunicación	15
6.2.1	Comunicación Master-Slave (RPC y REST)	15
6.2.2	Comunicación Slave-Slave	15
6.2.3	Comunicación Cliente-Sistema	15
6.3	Protocolo de mensajería	15
6.4	Algoritmo de elección de líder: Bully	16
7	Coordinación y sincronización	17
7.1	Modelo de coordinación	17
7.1.1	Desacoplamiento temporal y referencial	17
7.2	Sincronización de acciones	17
7.2.1	Protocolo de sincronización para operaciones distribuidas	17
7.3	Toma de decisiones distribuidas	18
7.3.1	Algoritmo Bully para elección de líder	18
7.3.2	Consenso Raft-Lite para operaciones críticas	18
8	Localización y nombrado de datos / recursos	18
8.1	Nombrado basado en DNS de Docker Swarm	19
8.2	Localización semántica con TF-IDF + MinHash	19
8.3	Particionamiento con VP-Tree y Centroides	19
8.3.1	Estructura del VP-Tree	19
8.3.2	Centroides de partición	20
8.3.3	LSH para agrupamiento de documentos similares	20
8.4	Rebalanceo activo de particiones	21
8.5	Estrategias de localización de datos	23
8.5.1	Estrategia 1: Búsqueda semántica centralizada con TF-IDF	23
8.5.2	Estrategia 2: Asignación por afinidad semántica	23
9	Distribución, replicación y consistencia de datos	24
9.1	Estrategia de replicación por afinidad semántica	24
9.2	Modelo de consistencia	24

9.2.1	Garantías de durabilidad	25
10	Tolerancia a fallos, robustez y dinámicas de red	25
10.1	Modelo de fallos	25
10.2	Dinámicas de red en Docker Swarm	25
10.3	Detección de fallos mediante Heartbeat	26
10.4	Protocolo de incorporación de nuevo nodo (JOIN)	27
10.5	Manejo de fallo de nodo	27
10.6	Métricas de resiliencia	28
11	Seguridad, autenticación y autorización	28
11.1	Seguridad en Docker Swarm	28
11.2	Autenticación y autorización a nivel de aplicación	29
11.3	Modelo de seguridad por capas	29
11.4	Protocolo de autenticación	30
11.5	Control de acceso basado en permisos	30
11.6	Sistema de reputación contra nodos maliciosos	30
11.7	Mitigación de ataques comunes	31
12	Análisis de Calidad del Sistema	31
12.1	Propiedades del diseño	31
12.2	Escalabilidad del diseño	31
12.3	Métricas de rendimiento esperadas	32
13	Conclusión	32

1 Introducción

1.1 Descripción del sistema

El sistema propuesto es una plataforma distribuida para almacenamiento y búsqueda de documentos utilizando una arquitectura Master-Slave con elección dinámica de líder. El sistema emplea vectorización semántica para ubicación de recursos, lo que permite localizar documentos basándose en similitud de contenido en lugar de funciones hash. Esta aproximación proporciona búsquedas más relevantes y una distribución de datos más inteligente basada en afinidad de contenido.

1.2 Objetivos del diseño

- Proporcionar alta disponibilidad mediante elección automática de líder ante fallos del Master
- Garantizar redundancia de datos mediante replicación basada en afinidad semántica
- Proporcionar mecanismos eficientes de localización de recursos mediante vectorización semántica
- Implementar tolerancia a fallos con detección mediante heartbeats y recuperación automática
- Distribuir carga entre Slaves mediante balanceo basado en carga y afinidad de contenido
- Permitir acceso distribuido mediante DNS con múltiples puntos de entrada

2 Arquitectura del Sistema

2.1 Estilos arquitectónicos

Según Tanenbaum & Van Steen (2017), los sistemas distribuidos pueden clasificarse según diferentes estilos arquitectónicos. DistriSearch combina múltiples estilos para lograr sus objetivos:

2.1.1 Arquitectura en Capas (Layered Architecture)

El sistema implementa una **arquitectura en capas** donde cada capa proporciona servicios a la capa superior y consume servicios de la capa inferior:

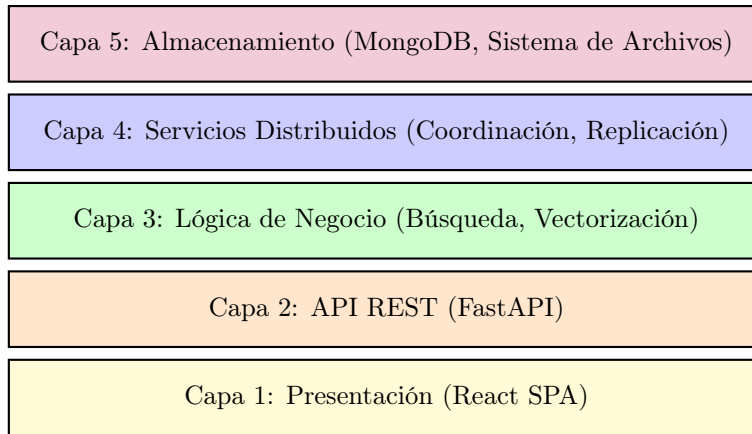


Figure 1: Arquitectura en capas del sistema

2.1.2 Arquitectura Publish-Subscribe para Eventos

Para la comunicación asíncrona entre componentes, el sistema utiliza el patrón **Publish-Subscribe** (Tanenbaum & Van Steen, 2017):

- **Eventos de heartbeat:** Los Slaves publican su estado; el Master y otros Slaves suscritos detectan fallos
- **Eventos de replicación:** Cuando se sube un documento, se publica evento para que réplicas lo reciban
- **Eventos de rebalanceo:** Al añadir/remover nodos, se notifica a todo el clúster

Este patrón proporciona **desacoplamiento temporal y referencial**: los publicadores no necesitan conocer a los suscriptores, y la comunicación puede ocurrir de forma asíncrona.

2.1.3 Elementos de arquitectura Peer-to-Peer

Aunque el sistema es principalmente Master-Slave, incorpora elementos **P2P** (Tanenbaum & Van Steen, 2017):

- **Comunicación Slave-Slave:** Los Slaves se comunican directamente para heartbeats y replicación sin pasar por el Master
- **Elección de líder distribuida:** Cualquier Slave puede iniciar una elección (algoritmo Bully)
- **Tolerancia a particiones:** Slaves en una partición de red pueden seguir operando entre sí

2.1.4 Estilo Master-Slave con promoción dinámica

El estilo principal es **Master-Slave** donde un nodo coordinador (Master) dirige a múltiples trabajadores (Slaves). La característica distintiva es que cualquier Slave puede convertirse en Master mediante elección cuando el Master actual falla, eliminando el punto único de fallo.

Los componentes principales son:

- **Nodos Slave:** Unidades autónomas que almacenan documentos, procesan búsquedas locales y mantienen su propia instancia de MongoDB. Cada Slave incluye backend (API FastAPI) y frontend (React).
- **Master:** Un Slave con responsabilidades adicionales de coordinación: mantiene el índice VP-Tree global, balancea carga, coordina replicación y enruta queries.
- **Load Balancer:** Distribuye tráfico entrante entre Slaves disponibles (Nginx/Traefik).

2.2 Modelo arquitectónico: Master-Slave con elección dinámica

2.2.1 Características del sistema

El sistema implementa una arquitectura **Master-Slave con failover automático**. A diferencia de arquitecturas centralizadas tradicionales, el rol de Master no está fijado a un nodo específico sino que puede migrar dinámicamente entre los Slaves candidatos cuando ocurre una falla.

Ubicación de recursos por vectorización semántica: En lugar de utilizar funciones hash para determinar dónde almacenar documentos (como en DHT), el sistema emplea vectores TF-IDF combinados con firmas MinHash. Cada documento se representa mediante un vector TF-IDF sparse y una firma MinHash de 128 hashes que permite estimar similitud Jaccard eficientemente. El Master mantiene un índice de ubicación que mapea perfiles TF-IDF a los Slaves que contienen documentos similares, permitiendo:

- Routing inteligente de queries a Slaves con contenido relevante
- Selección de nodos para replicación basada en afinidad de contenido
- Balanceo de carga considerando tanto capacidad como especialización de contenido

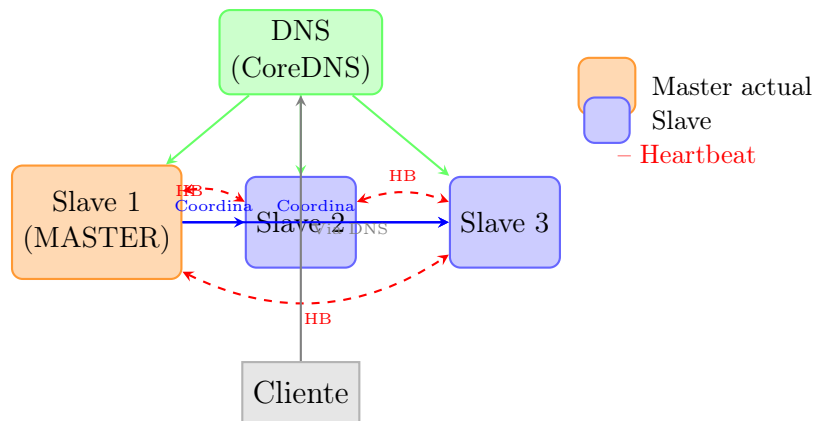


Figure 2: Arquitectura Master-Slave: Slave 1 actúa como Master. Los heartbeats detectan fallos. DNS permite acceso a cualquier nodo.

2.3 Distribución del sistema

2.3.1 Distribución jerárquica Master-Slave sobre Docker Swarm

El sistema emplea una **arquitectura Master-Slave** desplegada sobre **Docker Swarm**, el orquestador nativo de Docker que proporciona clustering, scheduling y service discovery automático. A diferencia de sistemas P2P puros, esta arquitectura proporciona un punto de coordinación centralizado que simplifica la gestión del clúster mientras mantiene la escalabilidad horizontal a través de los Slaves.

Docker Swarm aporta las siguientes características fundamentales (Tanenbaum & Van Steen, 2017):

- **Redes Overlay:** Comunicación multi-host mediante VXLAN, encapsulando tráfico Ethernet sobre UDP/IP
- **Service Discovery:** DNS interno en 127.0.0.11 que resuelve nombres de servicio a IPs virtuales
- **Load Balancing:** Routing mesh con IPVS para distribución de carga entre réplicas
- **Ingress Network:** Red overlay especial para tráfico externo con balanceo automático

Cada **Slave** es un nodo completo que integra:

- **Backend:** API REST para procesamiento de consultas y gestión de documentos
- **Frontend:** Aplicación web React servida mediante Nginx para interacción con usuarios
- **Base de datos:** Instancia MongoDB local para almacenamiento de documentos
- **Servicios de clúster:** Heartbeat, participación en elecciones, replicación

El **Master** coordina el clúster manteniendo:

- Índice semántico de ubicación de recursos
- Balanceador de carga entre Slaves
- Coordinador de replicación
- Enrutador de consultas

2.4 Topología de red: Estrella con redundancia

2.4.1 Definición de la topología

La red se estructura como una **topología en estrella** donde el Master actúa como nodo central y los Slaves se conectan directamente a él. Sin embargo, para garantizar tolerancia a fallos, todos los Slaves mantienen conexiones entre sí para:

- **Heartbeat:** Detección de fallos mediante UDP broadcast
- **Elección de líder:** Algoritmo Bully para elegir nuevo Master
- **Replicación directa:** Sincronización de datos entre réplicas

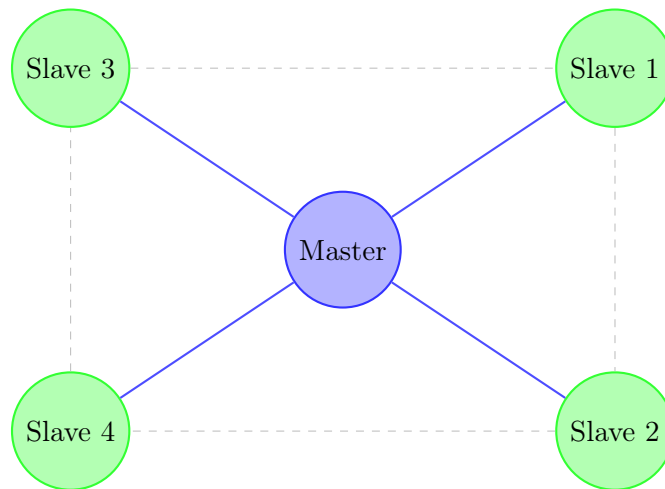


Figure 3: Topología Master-Slave: líneas sólidas = comunicación primaria, líneas punteadas = heartbeat y elección

2.4.2 Propiedades de la topología Master-Slave

Para un clúster con N Slaves:

- **Latencia de consulta:** $O(1)$ saltos (Master enruta directamente al Slave apropiado)
- **Escalabilidad:** Lineal con el número de Slaves
- **Tolerancia a fallos:** El sistema continúa operando si falla el Master (elección automática)

- **Consistencia:** Eventual, con replicación configurable (factor por defecto: 2)

Flujo de consulta típico:

1. Cliente envía consulta al Master
2. Master calcula embeddings semánticos de la consulta
3. Master identifica Slaves con contenido semánticamente relevante
4. Master enruta la consulta a los Slaves seleccionados
5. Slaves ejecutan búsqueda local y devuelven resultados
6. Master agrega y ordena resultados finales

3 Roles y organización funcional

En el sistema Master-Slave propuesto, los roles están claramente diferenciados para optimizar la coordinación y el procesamiento distribuido. El **Master** actúa como coordinador central del clúster, mientras que los **Slaves** son los nodos trabajadores que almacenan datos y procesan consultas.

El **Master** mantiene el índice semántico global de ubicación de recursos. Cuando recibe una consulta, calcula el embedding semántico y determina qué Slaves contienen información relevante. El Master no almacena documentos directamente sino metadatos sobre qué contenido tiene cada Slave.

Los **Slaves** son nodos autónomos que integran backend, frontend y base de datos. Cada Slave puede atender usuarios directamente a través de su interfaz web, procesar consultas locales y participar en la replicación de datos. Esta arquitectura permite que los Slaves operen de forma independiente incluso si el Master falla temporalmente.

Un aspecto crítico es la **elección de líder**: si el Master falla, los Slaves ejecutan el algoritmo Bully para elegir un nuevo Master entre los candidatos elegibles. Esto garantiza la continuidad del servicio sin intervención manual.

3.1 Diagrama de roles e interacciones

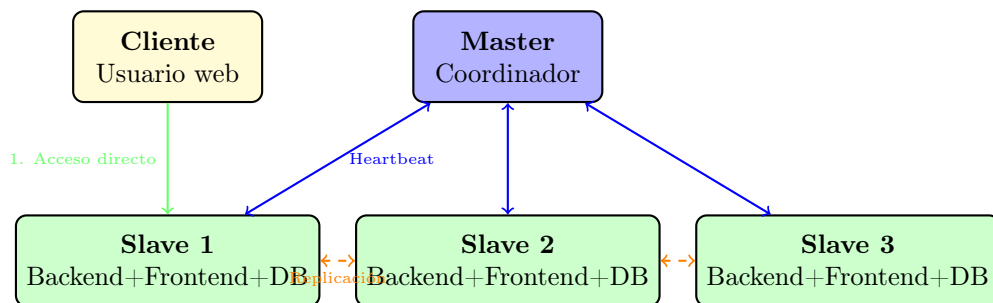


Figure 4: Arquitectura Master-Slave: clientes acceden directamente a Slaves, Master coordina ubicación y replicación

3.2 Responsabilidades de cada rol

Rol	Responsabilidades
Master	Mantener índice semántico de ubicación de recursos, recibir registros de nuevos documentos, calcular embeddings y determinar relevancia semántica, enrutar consultas a Slaves apropiados, coordinar replicación, monitorear salud del clúster mediante heartbeats, detectar fallos de nodos.
Slave	Almacenar documentos en MongoDB local, servir interfaz web (frontend React con Nginx), procesar consultas de búsqueda locales, participar en elecciones de líder (algoritmo Bully), enviar heartbeats al Master, recibir y aplicar replicaciones de otros Slaves.
Cliente	Acceder a cualquier Slave disponible vía web, realizar búsquedas semánticas y por nombre, subir y descargar documentos, el DNS resuelve <code>distrisearch.local</code> a cualquier Slave saludable.

Table 1: Responsabilidades por rol en arquitectura Master-Slave

4 Despliegue y distribución de servicios con Docker Swarm

El despliegue se implementa usando **Docker Swarm** como plataforma de orquestación, aprovechando sus capacidades nativas de networking, service discovery y load balancing:

4.1 Arquitectura de redes Docker Swarm

distrisearch-overlay Red overlay creada con driver `overlay` que permite comunicación multi-host. Utiliza VXLAN para encapsular tráfico entre nodos físicos del Swarm.

ingress Red overlay especial gestionada automáticamente por Swarm para tráfico externo. Implementa routing mesh con IPVS para balanceo de carga.

4.1.1 Modos de endpoint para Service Discovery

Docker Swarm ofrece dos modos de endpoint para resolución de servicios:

- **VIP (Virtual IP)**: Modo por defecto. El DNS interno resuelve el nombre del servicio a una IP virtual única. El balanceador IPVS distribuye el tráfico entre réplicas.
- **DNSRR (DNS Round Robin)**: El DNS retorna todas las IPs de las réplicas. El cliente decide a cuál conectarse.

El sistema utiliza **VIP** para los Slaves (balanceo automático) y permite acceso directo mediante tasks individuales: `slave.1.distrisearch-overlay`, `slave.2.distrisearch-overlay`, etc.

4.2 Comandos de despliegue con Docker Swarm

1. Inicializar Swarm y crear red overlay:

```
docker swarm init --advertise-addr <IP_MANAGER>
docker network create --driver overlay \
  --attachable distrisearch-overlay
```

2. Desplegar stack con docker-compose.swarm.yml:

```
docker stack deploy -c docker-compose.swarm.yml distrisearch
```

Configuración de servicio Master en docker-compose.swarm.yml:

```
master:
  image: distrisearch/master:latest
  networks:
    - distrisearch-overlay
  deploy:
    replicas: 1
    placement:
      constraints: [node.role == manager]
```

3. Configuración de Slaves con MongoDB en Swarm:

```
slave:
  image: distrisearch/slave:latest
  networks:
    - distrisearch-overlay
  environment:
    - MASTER_HOST=master # Resuelto por DNS interno
  deploy:
    replicas: 3
    endpoint_mode: vip    # Balanceo automático
  ports:
    - target: 8000
      published: 8000
      mode: ingress      # Routing mesh
```

```
mongo:
  image: mongo:7
  networks:
    - distrisearch-overlay
  deploy:
    replicas: 3
    placement:
      max_replicas_per_node: 1
```

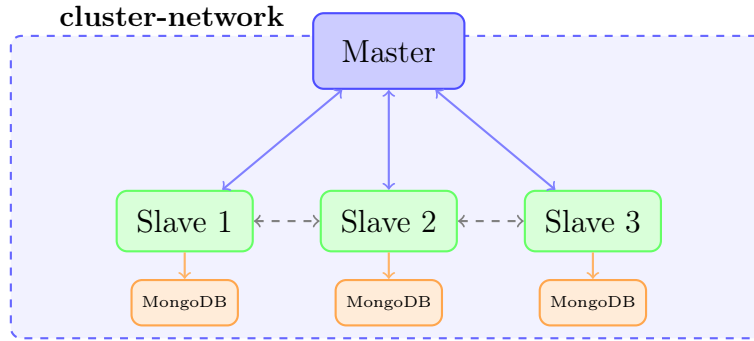


Figure 5: Clúster Master-Slave: Master coordina, cada Slave tiene MongoDB local, líneas punteadas = heartbeat

4.3 Ventajas del diseño

- **Simplicidad:** Una sola red para todo el clúster
- **Escalabilidad:** Nuevos Slaves se registran automáticamente con el Master
- **Tolerancia a fallos:** Si el Master falla, un Slave es elegido nuevo líder (algoritmo Bully)
- **Acceso directo:** Usuarios acceden a cualquier Slave sin necesidad de gateway

5 Procesos y patrones de diseño

Cada nodo del clúster ejecuta procesos especializados según su rol, siguiendo los principios de diseño de sistemas distribuidos modernos (Tanenbaum & Van Steen, 2017). Los **Slaves** ejecutan: un servidor FastAPI para el backend (procesos asíncronos con `uvicorn`), un servidor Nginx sirviendo la aplicación React para el frontend, un demonio de heartbeat UDP para detección de fallos, y un cliente Motor (async MongoDB) para persistencia local. El **Master** ejecuta adicionalmente: el índice semántico de ubicación basado en TF-IDF + MinHash, el balanceador de carga, y el coordinador de replicación.

5.1 Tipos de procesos en el sistema

Proceso	Tipo	Descripción
FastAPI Server	Asíncrono (asyncio)	Event loop principal, maneja múltiples conexiones HTTP concurrentes
React/Nginx	Estático + SPA	Aplicación React compilada servida por Nginx, cliente-side rendering
Heartbeat Daemon	Thread dedicado	UDP broadcaster/listener en hilo separado
Replication Worker	Asíncrono (asyncio)	Tareas de fondo para sincronización de réplicas
MongoDB Client	Asíncrono (Motor)	Pool de conexiones async para operaciones I/O

Table 2: Tipos de procesos por componente

En términos de concurrencia, el sistema utiliza `asyncio` de Python para manejar múltiples conexiones y operaciones I/O sin bloqueos, siguiendo el modelo de programación basada en eventos (Tanenbaum & Van Steen, 2017). Los heartbeats se procesan en un hilo separado usando UDP para minimizar latencia. El patrón adoptado combina arquitectura dirigida por eventos (event-driven) con paso de mensajes, ideal para sistemas distribuidos donde la latencia de red domina el tiempo de ejecución.

5.2 Arquitectura interna de un Slave

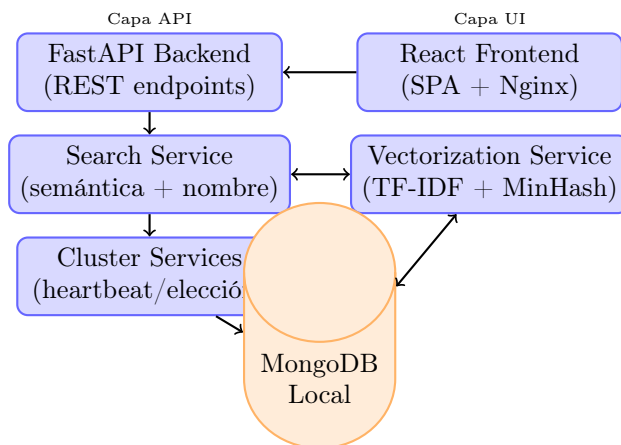


Figure 6: Arquitectura de un Slave: integra backend, frontend, búsqueda semántica y servicios de clúster

5.3 Patrones de diseño aplicados

Los siguientes patrones de diseño se aplican siguiendo las mejores prácticas de sistemas distribuidos (Tanenbaum & Van Steen, 2017):

Event-Driven Architecture — Cada componente reacciona a eventos (llegada de mensajes, timeouts, cambios de estado). FastAPI utiliza un event loop `asyncio` para multiplexar I/O.

Message Passing — Comunicación entre componentes mediante colas de mensajes asíncronas (`asyncio.Queue`).

Reactor Pattern — Event loop que multiplexea I/O de múltiples sockets usando `uvloop` o `selectors`.

Strategy Pattern — Algoritmos de routing y replicación intercambiables (VIP vs DNSRR, afinidad vs round-robin).

Observer Pattern — Notificación de cambios de estado de red a componentes interesados mediante callbacks `async`.

Circuit Breaker — Protección contra cascadas de fallos en llamadas entre servicios.

5.4 Modelo de concurrencia

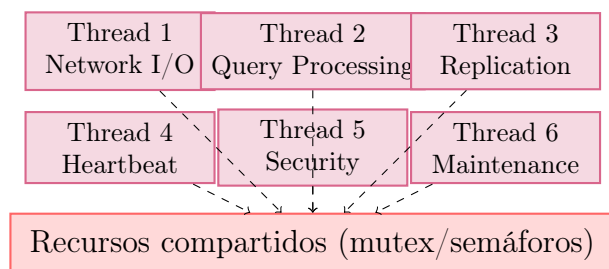


Figure 7: Modelo de threads para manejo concurrente de operaciones

6 Comunicación entre nodos

6.1 Modelo de comunicación por capas

El sistema implementa una **arquitectura en capas** para la comunicación siguiendo el modelo OSI adaptado a sistemas distribuidos (Tanenbaum & Van Steen, 2017). La **capa de aplicación** define un protocolo de mensajería de alto nivel con mensajes como `REGISTER`, `HEARTBEAT`, `ELECTION`, `REPLICATE` y `QUERY`. Esta capa contiene la lógica de coordinación Master-Slave, ubicación semántica de recursos y gestión de estado del clúster.

La **capa de transporte** utiliza:

- **HTTP/REST (TCP)**: Para operaciones de API síncronas (registro de documentos, consultas)
- **gRPC (HTTP/2)**: Para comunicación inter-nodo de alta eficiencia con Protocol Buffers

- **UDP**: Para heartbeats con baja latencia y tolerancia a pérdidas ocasionales

La **capa de red** utiliza las redes overlay de Docker Swarm con VXLAN. El **DNS interno de Docker** (127.0.0.11) resuelve nombres de servicio automáticamente:

- `master` → VIP del servicio Master
- `slave` → VIP balanceada entre réplicas de Slaves
- `tasks.slave` → Todas las IPs individuales (modo DNSRR)

6.2 Tipos de comunicación

6.2.1 Comunicación Master-Slave (RPC y REST)

Los Slaves se comunican con el Master mediante **REST API** (JSON sobre HTTP) para operaciones estándar y **gRPC** para operaciones de alto rendimiento:

- `POST /register`: Registrar nuevos documentos (el Master actualiza su índice TF-IDF)
- `gRPC ReplicateStream`: Streaming bidireccional para sincronización de réplicas
- `GET /health`: Health checks para el balanceador de Swarm

El Master utiliza el DNS de Docker Swarm para descubrir Slaves: `dns.resolver.query('tasks.slave', 'A')` retorna todas las IPs de réplicas activas.

6.2.2 Comunicación Slave-Slave

Los Slaves se comunican entre sí para: heartbeats UDP (detección de fallos), mensajes de elección (algoritmo Bully cuando el Master falla), y transferencia directa de réplicas mediante gRPC streams. Esta comunicación no pasa por el Master, garantizando continuidad ante fallos del coordinador.

6.2.3 Comunicación Cliente-Sistema

Los usuarios acceden al sistema a través del **ingress network** de Docker Swarm. El routing mesh recibe peticiones en cualquier nodo del Swarm y las enruta automáticamente a un Slave disponible mediante IPVS. Externamente se puede configurar un load balancer (HAProxy, Traefik) que resuelve `distrisearch.local`.

6.3 Protocolo de mensajería

Se definen los siguientes tipos de mensajes:

Mensaje	Descripción
REGISTER(doc_id, embedding)	Slave notifica al Master sobre nuevo documento con su embedding semántico
HEARTBEAT(node_id, status)	Slave envía estado de salud al Master y otros Slaves (UDP)
ELECTION(node_id)	Mensaje del algoritmo Bully: inicio de elección de nuevo Master
OK(node_id)	Respuesta en elección: nodo con ID mayor responde que tomará el liderazgo
COORDINATOR(node_id)	Anuncio del nuevo Master elegido a todos los Slaves
REPLICATE(doc_id, data, target)	Instrucción de replicar documento hacia Slave objetivo
QUERY(embedding, top_k)	Consulta semántica: Master enruta a Slaves con contenido similar

Table 3: Tipos de mensajes del protocolo Master-Slave

6.4 Algoritmo de elección de líder: Bully

Cuando un Slave detecta que el Master no responde (timeout en heartbeats):

Algorithm 1 Algoritmo Bully para elección de Master

- 1: **Input:** nodo actual P_i , conjunto de nodos $\{P_1, \dots, P_n\}$ ordenados por ID
 - 2: P_i envía mensaje ELECTION a todos P_j donde $j > i$
 - 3: **if** ningún P_j responde con OK en timeout **then**
 - 4: P_i se convierte en Master
 - 5: P_i envía COORDINATOR a todos los nodos
 - 6: **else**
 - 7: P_i espera mensaje COORDINATOR del nodo con mayor ID
 - 8: **end if**
-

Ejemplo: Si el Master (ID=100) falla y los Slaves tienen IDs 50, 60, 70:

1. Slave-50 detecta fallo, envía ELECTION a 60, 70
2. Slave-60 y 70 responden OK (tienen ID mayor)
3. Slave-60 envía ELECTION a 70
4. Slave-70 responde OK
5. Slave-70 no tiene nadie con ID mayor, se convierte en Master
6. Slave-70 envía COORDINATOR a todos: nuevo Master elegido

7 Coordinación y sincronización

7.1 Modelo de coordinación

La coordinación en sistemas distribuidos aborda dos problemas fundamentales: el ordenamiento de eventos y la toma de decisiones colectivas (Tanenbaum & Van Steen, 2017). El sistema DistriSearch implementa mecanismos específicos para cada uno.

7.1.1 Desacoplamiento temporal y referencial

El sistema implementa diferentes niveles de acoplamiento según el tipo de operación:

Operación	Acoplamiento temporal	Acoplamiento referencial
Consulta directa	Acoplado (síncrono)	Acoplado (peer específico)
Replicación	Desacoplado (asíncrono)	Acoplado (réplicas específicas)
Búsqueda flooding	Desacoplado	Parcialmente desacoplado
Heartbeat	Desacoplado	Acoplado (vecinos)

Table 4: Niveles de acoplamiento por tipo de operación

El sistema implementa dos modos principales de comunicación con diferentes características de acoplamiento. La **comunicación directa** es temporalmente acoplada, lo que significa que ambos nodos (emisor y receptor) deben estar activos simultáneamente para completar la interacción. También es referencialmente acoplada porque el emisor conoce explícitamente la identidad del receptor. Este modo se utiliza para consultas síncronas donde se espera una respuesta inmediata y para transferencias de datos que requieren confirmación.

Por otro lado, la **comunicación basada en eventos** es temporalmente desacoplada, permitiendo que publicación y suscripción ocurran en momentos diferentes sin requerir sincronía estricta. También es referencialmente desacoplada ya que los publicadores no necesitan conocer la identidad de los suscriptores. Este patrón se emplea para notificaciones de cambios de estado, detección de fallos y propagación de eventos que múltiples nodos pueden consumir.

7.2 Sincronización de acciones

7.2.1 Protocolo de sincronización para operaciones distribuidas

Relojes Lógicos de Lamport (1978): El sistema utiliza relojes lógicos para establecer un ordenamiento parcial de eventos distribuidos, fundamental para la consistencia de operaciones (Tanenbaum & Van Steen, 2017). Cada nodo mantiene un contador L_i que se incrementa en cada evento local según las siguientes reglas:

1. **Evento local:** $L_i := L_i + 1$
2. **Envío de mensaje:** Incrementar L_i , adjuntar timestamp $T = L_i$ al mensaje
3. **Recepción de mensaje** con timestamp T : $L_i := \max(L_i, T) + 1$

Este mecanismo garantiza la propiedad de **causalidad**: si un evento a causalmente precede a un evento b (notado $a \rightarrow b$), entonces $L(a) < L(b)$. Para resolver empates cuando dos eventos tienen el mismo timestamp, se utiliza el identificador del nodo como criterio de desempate: $(T_1, node_1) < (T_2, node_2)$ si $T_1 < T_2$, o si $T_1 = T_2$ y $node_1 < node_2$.

Aplicación en DistriSearch: Los timestamps de Lamport se utilizan para:

- Ordenar operaciones de replicación concurrentes
- Detectar mensajes de elección obsoletos
- Resolver conflictos en actualizaciones del índice semántico

7.3 Toma de decisiones distribuidas

7.3.1 Algoritmo Bully para elección de líder

El algoritmo Bully (Garcia-Molina, 1982) se utiliza para elegir un nuevo Master cuando el actual falla. Es un algoritmo de elección que garantiza que el nodo con mayor ID (prioridad) se convierte en líder.

7.3.2 Consenso Raft-Lite para operaciones críticas

Para operaciones que requieren acuerdo entre múltiples nodos, el sistema implementa una versión simplificada del algoritmo **Raft** (Ongaro & Ousterhout, 2014), adaptada para el contexto Master-Slave:

1. **Términos (Terms):** Cada período de liderazgo tiene un número de término monotonamente creciente
2. **Heartbeats del líder:** El Master envía heartbeats periódicos a todos los Slaves
3. **Timeout de elección:** Si un Slave no recibe heartbeat en $T_{election}$, inicia elección
4. **Votación:** Cada Slave vota por el primer candidato que solicita voto en un término
5. **Mayoría:** Candidato con $> N/2$ votos se convierte en líder

Consenso para cambios de configuración:

- Cambios críticos (ej: modificar factor de replicación) requieren consenso
- Se usa votación de mayoría simple: solo se aplica cambio si $> 50\%$ de nodos activos aprueban
- Los cambios se registran en un log replicado para recuperación ante fallos

8 Localización y nombrado de datos / recursos

El sistema de nombrado y localización combina dos mecanismos complementarios (Tanenbaum & Van Steen, 2017):

8.1 Nombrado basado en DNS de Docker Swarm

Docker Swarm proporciona un sistema de nombrado jerárquico automático:

- **Nombres de servicio:** `master`, `slave`, `mongo` resueltos por DNS interno
- **Nombres de tareas:** `slave.1`, `slave.2`, etc. para acceso a réplicas específicas
- **DNS queries:** `tasks.slave` retorna todas las IPs (DNSRR)

8.2 Localización semántica con TF-IDF + MinHash

Cada documento tiene un identificador único (UUID) y un **vector TF-IDF** que representa su contenido. A diferencia de embeddings de redes neuronales pre-entrenadas, TF-IDF ofrece:

- **Interpretabilidad:** Pesos directamente relacionados con términos del documento
- **Eficiencia:** Cálculo local sin dependencia de modelos externos
- **Extensibilidad:** Fácil actualización del vocabulario global

Para detección rápida de similitud, se utiliza **MinHash** (Broder, 1997) que genera firmas compactas de 128 hashes para cada documento, permitiendo estimar similitud Jaccard en $O(1)$.

El Master mantiene un **índice de ubicación semántica** que mapea cada Slave a un perfil TF-IDF agregado de sus documentos. Cuando llega una consulta, el Master calcula su vector TF-IDF y determina qué Slaves tienen contenido semánticamente similar usando similitud coseno.

8.3 Particionamiento con VP-Tree y Centroides

El sistema utiliza un **VP-Tree** (Vantage-Point Tree) para particionar el espacio semántico de documentos (Yianilos, 1993). A diferencia de estructuras como KD-Trees que trabajan bien en espacios de baja dimensión, VP-Tree escala eficientemente en espacios métricos de alta dimensión típicos de representaciones TF-IDF.

8.3.1 Estructura del VP-Tree

El VP-Tree divide recursivamente el espacio de documentos:

1. Se selecciona un **vantage point** (punto de referencia) como pivote
2. Se calcula la **distancia mediana** μ de todos los documentos al pivote
3. Los documentos con $d(x, vp) \leq \mu$ van al subárbol izquierdo
4. Los documentos con $d(x, vp) > \mu$ van al subárbol derecho
5. Se repite recursivamente hasta que cada hoja contiene $\leq \text{threshold}$ documentos

Algorithm 2 Construcción de VP-Tree para particionamiento

```
1: function BUILDVPTREE(documents)
2:   if  $|documents| \leq threshold$  then
3:     return Leaf(documents, centroid=COMPUTECENTROID(documents))
4:   end if
5:    $vp \leftarrow \text{SELECTVANTAGEPOINT}(\textit{documents})$   $\triangleright$  Maximiza varianza de distancias
6:    $distances \leftarrow [d(doc, vp) \text{ for } doc \text{ in } documents]$ 
7:    $\mu \leftarrow \text{median}(distances)$ 
8:    $left \leftarrow \{doc : d(doc, vp) \leq \mu\}$ 
9:    $right \leftarrow \{doc : d(doc, vp) > \mu\}$ 
10:  return Node( $vp, \mu, \text{BUILDVPTREE}(left), \text{BUILDVPTREE}(right)$ )
11: end function
```

8.3.2 Centroides de partición

Cada partición (hoja del VP-Tree) tiene un **centroide** que representa el perfil semántico agregado de sus documentos. El centroide se calcula como la media de los vectores TF-IDF normalizados:

$$\text{centroid}_p = \frac{1}{|D_p|} \sum_{d \in D_p} \frac{\mathbf{v}_d}{\|\mathbf{v}_d\|_2}$$

donde D_p es el conjunto de documentos en la partición p y \mathbf{v}_d es el vector TF-IDF del documento d .

Recomputación de centroides: Cuando la distribución de documentos en una partición cambia significativamente (drift semántico), se ejecuta un algoritmo de **k-medoids** para recomputar centroides óptimos:

Algorithm 3 Recomputación de centroides con k-medoids

```
1: function RECOMPUTECENTROIDS(partitions,  $k$ )
2:   for partition in partitions do
3:     if partition.drift_score > threshold then
4:        $medoids \leftarrow \text{KMEDOIDS}(\textit{partition.documents}, k = 1)$ 
5:        $\textit{partition.centroid} \leftarrow medoids[0]$ 
6:        $\textit{partition.drift\_score} \leftarrow 0$ 
7:     end if
8:   end for
9: end function
```

8.3.3 LSH para agrupamiento de documentos similares

Locality-Sensitive Hashing (LSH) se utiliza para pre-agrupar documentos similares antes de su asignación a particiones (Indyk & Motwani, 1998). LSH garantiza que documentos con alta similitud Jaccard tengan alta probabilidad de colisionar en el mismo bucket:

$$P[\text{hash}(a) = \text{hash}(b)] = J(a, b)$$

donde $J(a, b)$ es la similitud Jaccard entre documentos a y b .

El sistema implementa LSH con **bandas** (banding technique):

- Las 128 firmas MinHash se dividen en $b = 16$ bandas de $r = 8$ hashes cada una
- Dos documentos son **candidatos similares** si coinciden en al menos una banda completa
- Probabilidad de ser candidatos: $P = 1 - (1 - s^r)^b$ donde s es la similitud real
- Con $b = 16, r = 8$: documentos con $s \geq 0.5$ tienen $> 96\%$ de probabilidad de ser candidatos

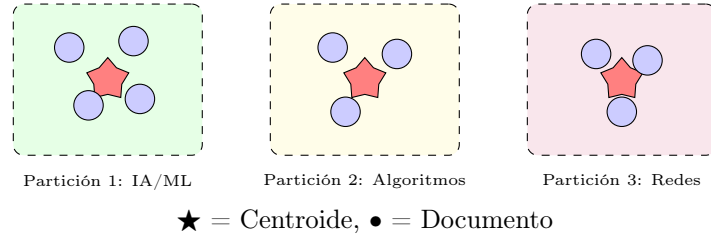


Figure 8: Particiones VP-Tree con centroides representando regiones semánticas

8.4 Rebalanceo activo de particiones

Cuando un nodo se une o abandona el clúster, el sistema ejecuta un **rebalanceo activo** para redistribuir documentos. El algoritmo utiliza la técnica "**Power of Two Choices**" con afinidad semántica:

Algorithm 4 Rebalanceo activo con Power of Two Choices

```
1: function REBALANCE(cluster, event)
2:   if event.type = NODE_JOIN then
3:     overloaded  $\leftarrow$  GETOVERLOADEDNODES(cluster)
4:     for node in overloaded do
5:       docs_to_migrate  $\leftarrow$  SELECTMIGRATIONCANDIDATES(node)
6:       for doc in docs_to_migrate do
7:         choice1, choice2  $\leftarrow$  RANDOMSAMPLE(cluster.nodes, 2)
8:         best  $\leftarrow$  node con menor carga Y mayor afinidad semántica con doc
9:         MIGRATE(doc, node, best)
10:      end for
11:    end for
12:  else if event.type = NODE_LEAVE then
13:    orphan_docs  $\leftarrow$  event.node.documents
14:    for doc in orphan_docs do
15:      target  $\leftarrow$  FINDBESTNODE(doc, cluster) ▷ Por afinidad
16:      RESTOREFROMREPLICA(doc, target)
17:    end for
18:  end if
19: end function
```

Métricas para selección de nodo destino:

- **Score de afinidad:** $\text{affinity}(\text{doc}, \text{node}) = \cos(\mathbf{v}_{\text{doc}}, \text{centroid}_{\text{node}})$
- **Score de carga:** $\text{load}(\text{node}) = \frac{\text{docs_count}}{\text{capacity}}$
- **Score combinado:** $\alpha \cdot \text{affinity} + (1 - \alpha) \cdot (1 - \text{load})$ con $\alpha = 0.6$

8.5 Estrategias de localización de datos

8.5.1 Estrategia 1: Búsqueda semántica centralizada con TF-IDF

Algorithm 5 Búsqueda semántica en Master usando TF-IDF + MinHash

```

1: Input: query (texto de consulta), top_k (número de resultados)
2: Output: lista de documentos relevantes
3:  $q\_tfidf \leftarrow \text{calcular\_tfidf}(query, \text{vocabulario\_global})$ 
4:  $q\_minhash \leftarrow \text{calcular\_minhash}(query)$ 
5:                                     ▷ Fase 1: Filtrado rápido con MinHash
6:  $candidatos \leftarrow \text{filtrar\_por\_minhash}(q\_minhash, \text{umbral\_jaccard} = 0.3)$ 
7:                                     ▷ Fase 2: Ranking preciso con TF-IDF
8:  $slaves \leftarrow \text{ordenar\_slaves\_por\_similitud\_coseno}(q\_tfidf, candidatos)$ 
9:  $results \leftarrow []$ 
10: for slave en  $slaves[:3]$  do                                     ▷ Top 3 Slaves más relevantes
11:    $local\_results \leftarrow \text{enviar\_query\_grpc}(slave, query)$ 
12:    $results \leftarrow results \cup local\_results$ 
13: end for
14: return  $\text{ordenar\_por\_relevancia}(results)[:top\_k]$ 

```

8.5.2 Estrategia 2: Asignación por afinidad semántica

Asignar cada documento al Slave cuyo perfil TF-IDF es más similar:

$$slave_destino = \arg \max_{s \in \text{slaves}} \cos(tfidf_{doc}, profile_s)$$

donde \cos es la similitud coseno entre vectores TF-IDF.

Ejemplo: Un documento sobre "algoritmos de ordenamiento" tiene un vector TF-IDF que el Master compara con los perfiles de cada Slave. Si Slave-2 ya contiene documentos sobre "estructuras de datos" y "complejidad algorítmica", su perfil TF-IDF tendrá términos similares, por lo que el nuevo documento se almacena allí.

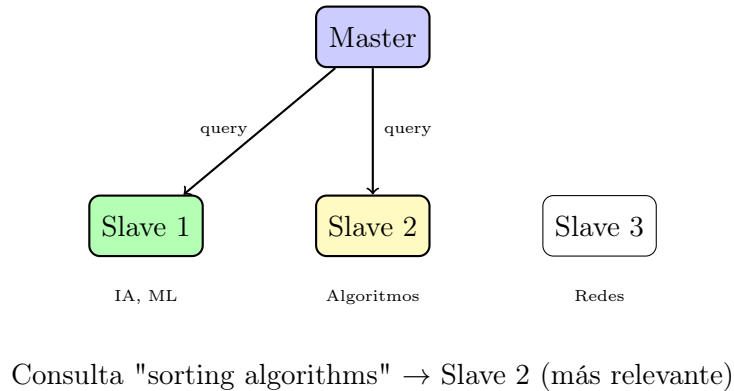


Figure 9: Ubicación semántica: Master enruta consultas a Slaves con contenido similar

9 Distribución, replicación y consistencia de datos

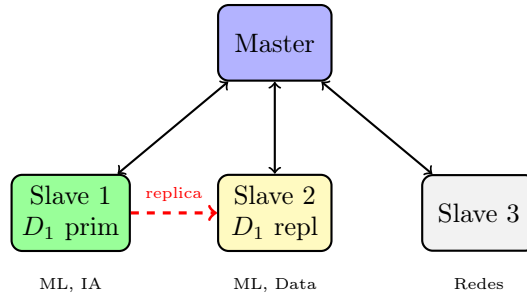
La replicación es fundamental para garantizar disponibilidad y tolerancia a fallos en sistemas distribuidos (Tanenbaum & Van Steen, 2017). El sistema implementa replicación donde cada documento puede tener k réplicas en diferentes Slaves (por defecto $k = 2$). El coordinador de replicación en el Master selecciona Slaves con **afinidad semántica basada en TF-IDF**: documentos similares se replican en Slaves que ya contienen contenido relacionado, mejorando la localidad de las consultas.

Respecto a la consistencia, los documentos son inmutables una vez subidos, simplificando el modelo al eliminar la necesidad de sincronizar actualizaciones (modelo WORM: Write Once, Read Many).

9.1 Estrategia de replicación por afinidad semántica

Para un factor de replicación $k = 2$, al almacenar un documento en el Slave primario S_p , el Master selecciona $k - 1 = 1$ Slaves adicionales para réplicas. Los criterios de selección son:

- **Afinidad semántica**: Preferir Slaves cuyo perfil TF-IDF sea similar al vector del documento
- **Carga actual**: Balancear distribución de almacenamiento entre Slaves (métrica: GB usados)
- **Disponibilidad**: Priorizar Slaves con historial de uptime alto (ventana de 24h)



D_1 (doc sobre ML) \rightarrow replicado a Slave 2 (perfil similar)

Figure 10: Replicación por afinidad semántica: documento ML replicado a Slave con perfil similar

9.2 Modelo de consistencia

El sistema adopta un modelo de **consistencia eventual** (Tanenbaum & Van Steen, 2017), priorizando disponibilidad y tolerancia a particiones (teorema CAP):

- **Escrituras**: Documentos nuevos se escriben primero en el Slave que recibe la subida (*primary*), luego se replican asincrónicamente a réplicas secundarias

- **Lecturas:** Cualquier Slave con una réplica puede responder consultas (*read-any*)
- **Convergencia:** Garantizada por la naturaleza inmutable de los documentos
- **Ventana de inconsistencia:** Típicamente $< 5s$ bajo condiciones normales de red

9.2.1 Garantías de durabilidad

- **At-least-once delivery:** Replicación con ACKs y reintentos automáticos
- **Quorum writes:** Opcionalmente, escritura confirmada solo si $\geq k/2 + 1$ réplicas confirman
- **Anti-entropy:** Reconciliación periódica entre réplicas usando MinHash para detección de divergencias

10 Tolerancia a fallos, robustez y dinámicas de red

La tolerancia a fallos parciales es una característica esencial de los sistemas distribuidos (Tanenbaum & Van Steen, 2017). El sistema DistriSearch proporciona soporte integral para: Slaves que se desconectan temporalmente, fallo del Master (recuperado mediante elección Bully), particiones de red, y reincorporación de nodos nuevos o recuperados.

10.1 Modelo de fallos

El sistema asume un modelo de **fail-stop** donde los nodos fallan deteniendo su ejecución completamente (no hay comportamiento bizantino). Los fallos se clasifican en:

- **Crash failures:** El nodo deja de responder y no se recupera
- **Omission failures:** Pérdida de mensajes (mitigado con reintentos)
- **Timing failures:** Respuestas fuera de timeout (detectado por heartbeats)

La arquitectura Master-Slave con elección de líder garantiza continuidad: si el Master falla, los Slaves detectan la ausencia de heartbeats y ejecutan el algoritmo Bully para elegir un nuevo coordinador.

10.2 Dinámicas de red en Docker Swarm

Cuando un Slave nuevo se une a la red o un Slave existente falla, Docker Swarm gestiona automáticamente parte del proceso:

- **Nuevo Slave:** Swarm lo añade a la red overlay, DNS interno actualiza registros automáticamente
- **Registro con Master:** El nuevo Slave envía su perfil TF-IDF inicial vía REST API

- **Fallo de Slave:** El Master detecta ausencia de heartbeats (timeout 15s) y marca el nodo como no disponible
- **Health checks de Swarm:** Docker también detecta fallos y puede reiniciar contenedores automáticamente
- **Fallo de Master:** Slaves detectan timeout, inician elección Bully, nuevo Master asume coordinación
- **Re-replicación:** Si un Slave con réplicas falla permanentemente, el Master coordina crear nuevas réplicas

10.3 Detección de fallos mediante Heartbeat

Algorithm 6 Protocolo Heartbeat para detección de fallos

```

1: Constantes:  $T_{heartbeat} = 5s$ ,  $T_{timeout} = 15s$ 
2: while nodo está activo do
3:   for cada vecino  $v$  en lista de vecinos do
4:     Enviar PING a  $v$ 
5:     if no se recibe PONG en  $T_{timeout}$  then
6:       Marcar  $v$  como fallido
7:       Notificar a otros vecinos sobre fallo de  $v$ 
8:       Intentar reconexión o buscar reemplazo
9:     end if
10:  end for
11:  Esperar  $T_{heartbeat}$ 
12: end while

```

10.4 Protocolo de incorporación de nuevo nodo (JOIN)

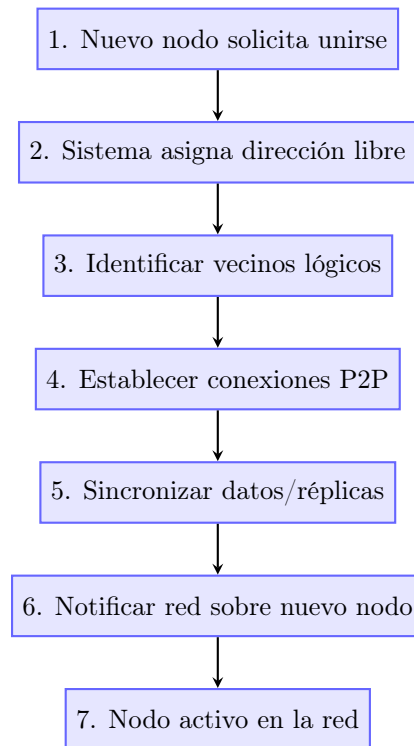


Figure 11: Proceso de incorporación de nuevo nodo a la red P2P

10.5 Manejo de fallo de nodo

Cuando un nodo N falla:

1. **Detección:** Vecinos detectan fallo por timeout de heartbeat
2. **Notificación:** Vecinos notifican al resto de la red
3. **Reconexión:** Vecinos de N se conectan entre sí para mantener conectividad
4. **Re-replicación:** Datos con réplicas en N se replican en otros nodos
5. **Actualización de rutas:** Tablas de routing se actualizan

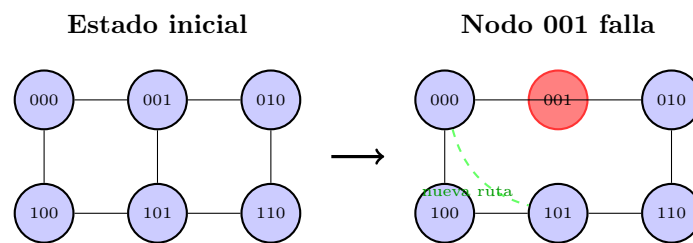


Figure 12: Reconfiguración de red tras fallo de nodo 001: rutas alternativas y reconexiones

10.6 Métricas de resiliencia

Para un clúster con N Slaves, el sistema presenta características de resiliencia cuantificables:

- **Tolerancia a fallos de Slaves:** El sistema permanece operativo mientras al menos un Slave esté activo
- **Tolerancia a fallo de Master:** El algoritmo Bully elige nuevo Master en $O(N)$ mensajes
- **Tiempo de detección de fallo:** Determinado por $T_{timeout} = 15s$ (3 heartbeats perdidos)
- **Tiempo de recuperación:** Elección Bully completa en $< 30s$ típicamente

11 Seguridad, autenticación y autorización

La seguridad en sistemas distribuidos abarca múltiples dimensiones: confidencialidad, integridad, disponibilidad y autenticación (Tanenbaum & Van Steen, 2017). El sistema DistriSearch implementa múltiples capas de seguridad aprovechando las capacidades nativas de Docker Swarm.

11.1 Seguridad en Docker Swarm

Docker Swarm proporciona mecanismos de seguridad integrados:

- **Cifrado de overlay networks:** Habilitado con `--opt encrypted`, utiliza IPsec para cifrar todo el tráfico entre nodos del Swarm
- **Mutual TLS (mTLS):** Comunicación entre nodos del Swarm automáticamente cifrada y autenticada
- **Docker Secrets:** Almacenamiento seguro de credenciales, API keys y certificados, accesibles solo por servicios autorizados
- **Rotación automática de certificados:** Swarm rota certificados TLS cada 90 días (configurable)

```
# Crear red overlay cifrada
docker network create --driver overlay --opt encrypted \
  distrisearch-secure

# Crear secret para API key
echo "my-api-key" | docker secret create api_key -

# Usar secret en servicio
docker service create --name slave --secret api_key \
  distrisearch/slave:latest
```

11.2 Autenticación y autorización a nivel de aplicación

Cada nodo posee una identidad única basada en criptografía de clave pública (par de claves pública/privada), lo que permite autenticación mutua entre nodos antes de establecer comunicación. El sistema implementa control de acceso mediante:

- **JWT tokens:** Para autenticación de clientes en la API REST
- **API Keys:** Para comunicación inter-servicio con Docker Secrets
- **ACLs:** Listas de control de acceso para operaciones administrativas

11.3 Modelo de seguridad por capas

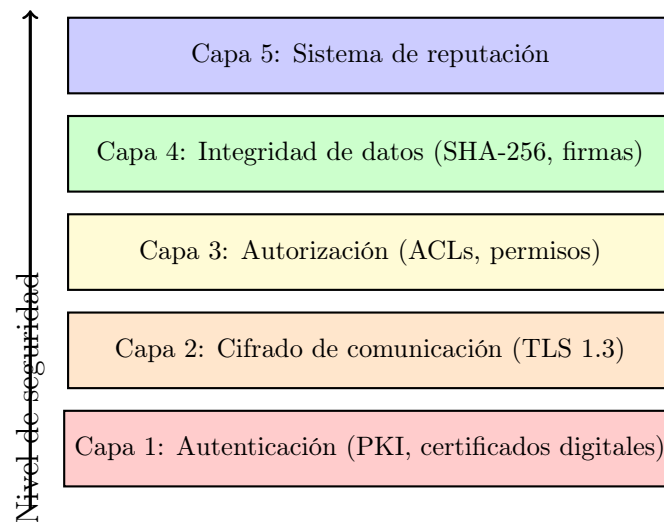


Figure 13: Arquitectura de seguridad por capas

11.4 Protocolo de autenticación

Algorithm 7 Autenticación mutua entre peers

```

1: Nodo A quiere conectarse con Nodo B
2: A genera nonce  $N_A$  aleatorio
3:  $A \rightarrow B$ : HELLO( $ID_A, N_A, Cert_A$ )
4: B verifica  $Cert_A$  con autoridad certificadora
5: if  $Cert_A$  es válido then
6:   B genera nonce  $N_B$ 
7:    $B \rightarrow A$ : CHALLENGE( $N_B, Sign_B(N_A), Cert_B$ )
8:   A verifica  $Cert_B$  y  $Sign_B(N_A)$ 
9:   if verificación exitosa then
10:     $A \rightarrow B$ : RESPONSE( $Sign_A(N_B)$ )
11:    B verifica  $Sign_A(N_B)$ 
12:    Conexión autenticada establecida
13:    Establecer canal TLS con claves de sesión
14:   end if
15: end if

```

11.5 Control de acceso basado en permisos

Cada dato tiene asociado un descriptor de seguridad:

Campo	Descripción
owner	ID del nodo propietario del dato
readers	Lista de nodos con permiso de lectura
writers	Lista de nodos con permiso de escritura
signature	Firma digital del propietario
hash	Hash SHA-256 para verificar integridad
timestamp	Marca temporal de creación/modificación

Table 5: Descriptor de seguridad de datos

11.6 Sistema de reputación contra nodos maliciosos

Cada nodo mantiene una tabla de reputación de sus vecinos que se actualiza continuamente según un modelo de promedio ponderado exponencial: $R_i(t+1) = \alpha \cdot R_i(t) + (1 - \alpha) \cdot B_i(t)$, donde $R_i(t)$ representa la reputación del nodo i en el tiempo t , $B_i(t)$ captura el comportamiento reciente tomando valor 1 para comportamiento bueno y 0 para comportamiento malo, y $\alpha = 0.9$ es el factor de decaimiento que da mayor peso al historial.

El sistema toma acciones diferenciadas según el nivel de reputación: nodos con $R_i > 0.8$ son considerados confiables y reciben prioridad alta en routing y almacenamiento; nodos con $0.5 < R_i \leq 0.8$ son tratados como normales sin privilegios especiales ni restricciones; nodos con $0.3 < R_i \leq 0.5$ son clasificados como sospechosos y sometidos a monitoreo aumentado;

finalmente, nodos con $R_i \leq 0.3$ son bloqueados y desconectados de la red para proteger la integridad del sistema.

11.7 Mitigación de ataques comunes

Ataque	Descripción	Mitigación
Sybil	Crear múltiples identidades falsas	PKI + verificación de identidad + límite de nodos por IP
Eclipse	Aislar nodo víctima	Diversificación de conexiones + monitoreo de conectividad
Man-in-the-Middle	Interceptar comunicaciones	TLS 1.3 + autenticación mutua
Data poisoning	Insertar datos corruptos	Firmas digitales + verificación de hash + reputación
DDoS	Saturar nodos con peticiones	Rate limiting + listas negras + sistema de reputación

Table 6: Ataques comunes y mecanismos de mitigación

12 Análisis de Calidad del Sistema

12.1 Propiedades del diseño

El sistema presenta 4 propiedades fundamentales que caracterizan su arquitectura. La **descentralización** garantiza que no existe un punto central de fallo, ya que todos los nodos poseen capacidades equivalentes y pueden asumir cualquier rol necesario. La **redundancia** se manifiesta en múltiples rutas alternativas entre nodos y en la replicación de datos a través de múltiples nodos. La **escalabilidad** se logra mediante un crecimiento logarítmico tanto de los recursos requeridos por nodo como del diámetro de la red respecto al número total de nodos. La **flexibilidad** permite la incorporación y salida dinámica de nodos sin interrumpir el servicio.

12.2 Escalabilidad del diseño

Slaves	Capacidad	Throughput	Escenario de uso
3	3 TB	100 qps	Prototipo/Testing
5	5 TB	200 qps	Desarrollo
10	10 TB	500 qps	Producción pequeña
20	20 TB	1000 qps	Producción media
50	50 TB	2500 qps	Producción grande
100	100 TB	5000 qps	Escala empresarial

Table 7: Escalabilidad del sistema Master-Slave según número de Slaves

Análisis: La arquitectura Master-Slave presenta escalabilidad lineal con el número de Slaves. Cada Slave agregado aumenta proporcionalmente la capacidad de almacenamiento y el throughput del sistema. El Master se convierte en potencial cuello de botella solo cuando el número de Slaves supera varios cientos, momento en que se pueden implementar técnicas de sharding del índice semántico o replicación del Master. Para la mayoría de casos de uso (hasta 100 Slaves), un solo Master es suficiente.

12.3 Métricas de rendimiento esperadas

Métrica	Descripción	Objetivo
Disponibilidad	Porcentaje de tiempo que el sistema está operativo	> 99.5%
Latencia de búsqueda	Tiempo promedio para localizar un dato	< 500ms
Tasa de éxito	Porcentaje de búsquedas exitosas	> 95%
Throughput	Consultas por segundo soportadas	> 1000 qps
Overhead de red	Mensajes adicionales vs óptimo teórico	< 3x
MTTR	Tiempo promedio de recuperación tras fallo	< 30s

Table 8: Métricas de rendimiento del sistema

13 Conclusión

Este documento ha presentado la especificación arquitectónica completa de un sistema distribuido Master-Slave desplegado sobre Docker Swarm con ubicación de recursos por vectorización semántica basada en TF-IDF y MinHash. El diseño prioriza cinco aspectos fundamentales según los principios de sistemas distribuidos (Tanenbaum & Van Steen, 2017):

Coordinación centralizada con tolerancia a fallos: El Master coordina el clúster manteniendo el índice semántico de ubicación basado en TF-IDF y el balanceo de carga. Si el Master falla, el algoritmo Bully garantiza elección automática de un nuevo líder entre los Slaves candidatos, eliminando puntos únicos de fallo permanentes. El consenso Raft-Lite se utiliza para operaciones críticas.

Ubicación semántica de recursos: En lugar de funciones hash (DHT) o embeddings de redes neuronales costosos, el sistema utiliza vectores TF-IDF combinados con firmas MinHash para representar documentos. Esto permite localización basada en similitud de contenido con alta eficiencia computacional, mejorando la relevancia de búsquedas y la localidad de datos relacionados.

Alta disponibilidad mediante Docker Swarm: El despliegue sobre Docker Swarm proporciona redes overlay con VXLAN, service discovery automático vía DNS interno, y balanceo de carga mediante routing mesh. Cada Slave es autónomo con su propio backend FastAPI, frontend React (servido por Nginx) y base de datos MongoDB.

Escalabilidad horizontal: Agregar Slaves es trivial gracias a Docker Swarm: `docker service scale slave=N`. El nuevo nodo se registra automáticamente en la red overlay, el DNS interno actualiza sus registros, y el Master incorpora el perfil TF-IDF del nuevo Slave.

Seguridad integrada: Docker Swarm proporciona mTLS automático entre nodos, cifrado opcional de overlay networks, y Docker Secrets para gestión segura de credenciales. La aplicación añade JWT para autenticación de clientes y ACLs para autorización.

Referencias

- Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed Systems: Principles and Paradigms* (4th ed.). Pearson Education.
- Lamport, L. (1978). “Time, Clocks, and the Ordering of Events in a Distributed System”. *Communications of the ACM*, 21(7), 558-565.
- Garcia-Molina, H. (1982). “Elections in a Distributed Computing System”. *IEEE Transactions on Computers*, C-31(1), 48-59.
- Ongaro, D., & Ousterhout, J. (2014). “In Search of an Understandable Consensus Algorithm (Raft)”. *USENIX Annual Technical Conference*.
- Broder, A. Z. (1997). “On the resemblance and containment of documents”. *Compression and Complexity of Sequences*, 21-29.
- Yianilos, P. N. (1993). “Data structures and algorithms for nearest neighbor search in general metric spaces”. *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 311-321.
- Indyk, P., & Motwani, R. (1998). “Approximate nearest neighbors: towards removing the curse of dimensionality”. *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC)*, 604-613.
- Docker, Inc. (2024). Docker Swarm networking documentation. <https://docs.docker.com/network/drivers/overlay/>
- FastAPI: Modern, fast web framework for building APIs with Python 3.7+. <https://fastapi.tiangolo.com/>
- React: A JavaScript library for building user interfaces. <https://react.dev/>