Sistema Distribuido Master-Slave con Ubicación Semántica

Abel Ponce González C411
Richard Alejandro Matos Arderí C411

December 5, 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

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 Estilo arquitectónico

2.1.1 Clasificación según estilos arquitectónicos

El sistema sigue un **estilo arquitectónico Master-Slave** con capacidad de promoción dinámica. En este modelo, un nodo asume el rol de Master (coordinador) mientras los demás actúan como Slaves (trabajadores). La característica distintiva es que cualquier Slave puede convertirse en Master mediante un proceso de elección cuando el Master actual falla, eliminando así el punto único de fallo típico de arquitecturas centralizadas.

Los componentes principales son:

- **Nodos Slave**: Unidades autónomas que almacenan documentos, procesan búsquedas locales y mantienen su propia instancia de MongoDB. Cada Slave incluye tanto backend (API FastAPI) como frontend (interfaz de usuario).
- **Master**: Un Slave que asume responsabilidades adicionales de coordinación: mantiene el índice de ubicación semántica global, balancea carga entre Slaves, coordina replicación y enruta queries distribuidas.

- **Sistema DNS**: Proporciona resolución de nombres con múltiples entradas para failover automático.

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 embeddings semánticos generados mediante sentence-transformers. Cada documento se representa como un vector de 384 dimensiones que captura su significado semántico. El Master mantiene un índice de ubicación que mapea embeddings 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

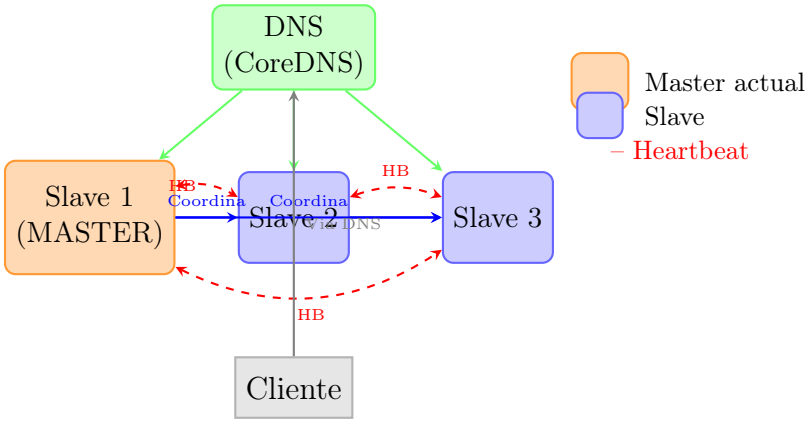


Figure 1: Arquitectura Master-Slave: Slave 1 actúa como Master. Los heartbeats detectan fallos. DNS permite acceso a cualquier nodo.

2.3 Distribución del sistema

2.3.1 Distribución jerárquica Master-Slave

El sistema emplea una **arquitectura Master-Slave** donde existe un nodo coordinador (Master) y múltiples nodos trabajadores (Slaves). 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.

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

- **Backend**: API REST para procesamiento de consultas y gestión de documentos
- **Frontend**: Interfaz web Streamlit 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

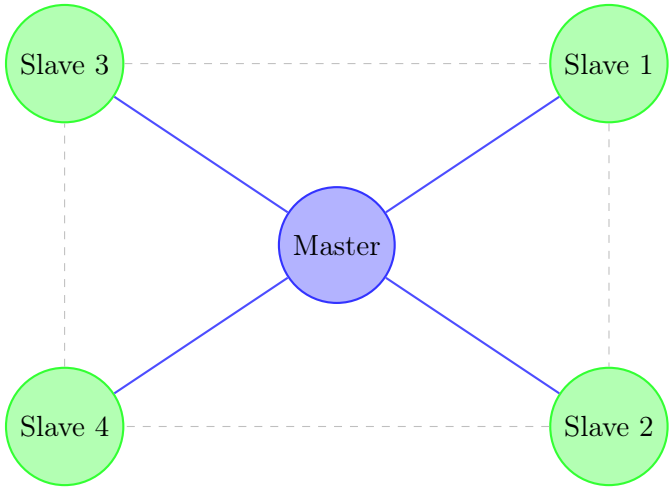


Figure 2: Topología Master-Slave: líneas sólidas = comunicación primaria, líneas punteadas = heartbeat y elección

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

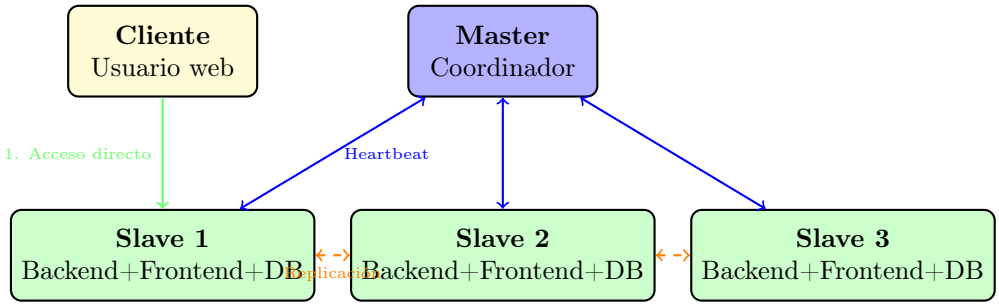


Figure 3: Arquitectura Master-Slave: clientes acceden directamente a Slaves, Master coordina ubicación y replicación

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 Streamlit), 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 distrisearch.local a cualquier Slave saludable.

Table 1: Responsabilidades por rol en arquitectura Master-Slave

4 Despliegue y distribución de servicios con Docker

El despliegue se implementa usando contenedores Docker con una red compartida para el clúster Master-Slave:

4.1 Arquitectura de redes Docker

cluster-network Red bridge donde residen el Master, los Slaves y sus bases de datos MongoDB. Permite comunicación directa entre todos los componentes del clúster.

dns-network Red opcional para CoreDNS que resuelve `distrisearch.local` a los Slaves disponibles.

Cada Slave expone sus puertos de backend (8000) y frontend (8501) al host, permitiendo acceso directo desde usuarios externos.

4.2 Comandos de despliegue

1. Crear red del clúster:

```
docker network create cluster-network
```

2. Desplegar Master:

```
docker run -d --name master --network cluster-network \
  -e NODE_ROLE=master \
  -e NODE_ID=master-001 \
  master-image:latest
```

3. Desplegar Slaves con MongoDB local:

```
docker run -d --name mongo-slave1 --network cluster-network \
  mongo:latest
docker run -d --name slave-001 --network cluster-network \
  -e NODE_ROLE=slave \
  -e NODE_ID=slave-001 \
  -e MASTER_HOST=master \
  -p 8000:8000 -p 8501:8501 \
  slave-image:latest
```

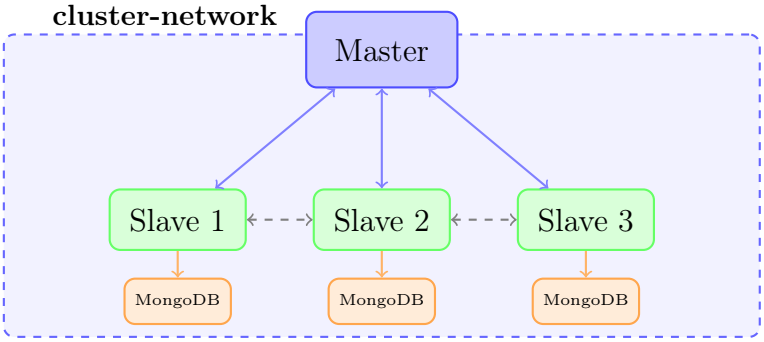


Figure 4: Clúster Master-Slave: Master coordina, cada Slave tiene MongoDB local, líneas punteadas = heartbeat

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. Los **Slaves** ejecutan: un servidor FastAPI para el backend, un servidor Streamlit para el frontend, un servicio de heartbeat UDP para detección de fallos, y un cliente MongoDB para persistencia local. El **Master** ejecuta adicionalmente: el índice semántico de ubicación, el balanceador de carga, y el coordinador de replicación.

En términos de concurrencia, el sistema utiliza `asyncio` de Python para manejar múltiples conexiones y operaciones I/O sin bloqueos. 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.

5.1 Arquitectura interna de un Slave

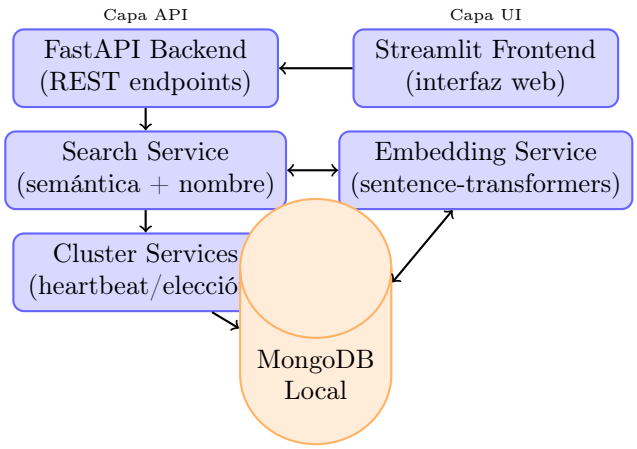


Figure 5: Arquitectura de un Slave: integra backend, frontend, búsqueda semántica y servicios de clúster

5.2 Patrones de diseño aplicados

Event-Driven Architecture — Cada componente reacciona a eventos (llegada de mensajes, timeouts, cambios de estado)

Message Passing — Comunicación entre componentes mediante colas de mensajes asíncronas

Reactor Pattern — Event loop que multiplexea I/O de múltiples sockets

Strategy Pattern — Algoritmos de routing y replicación intercambiables

Observer Pattern — Notificación de cambios de estado de red a componentes interesados

5.3 Modelo de concurrencia

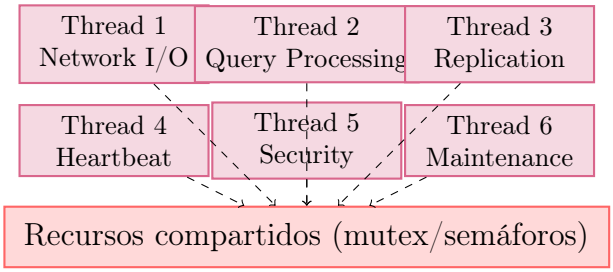


Figure 6: Modelo de threads para manejo concurrente de operaciones

6 Comunicación entre nodos

6.1 Modelo de comunicación por capas

El sistema implementa una **arquitectura en capas** para la comunicación que separa las responsabilidades en tres niveles. 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 como registro de documentos, consultas y replicación. Para heartbeats entre nodos se emplea UDP, optimizado para baja latencia y tolerancia a pérdidas ocasionales. Esta capa también gestiona conexiones persistentes mediante pools de conexiones HTTP.

La **capa de red** utiliza routing IP estándar sobre la red Docker interna. El dominio `distrisearch.local` se resuelve mediante CoreDNS a cualquier Slave saludable, proporcionando descubrimiento automático de servicios.

6.2 Tipos de comunicación

6.2.1 Comunicación Master-Slave

Los Slaves se comunican con el Master mediante REST API para: registrar nuevos documentos (el Master actualiza su índice semántico), recibir instrucciones de replicación, y reportar estado de salud. El Master inicia comunicación con Slaves para: enrutar consultas a nodos relevantes, coordinar replicación, y verificar disponibilidad.

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. Esta comunicación no pasa por el Master, garantizando continuidad ante fallos del coordinador.

6.2.3 Comunicación Cliente-Sistema

Los usuarios acceden directamente a cualquier Slave mediante su interfaz web Streamlit. El DNS resuelve `distrisearch.local` a Slaves disponibles, proporcionando balanceo de carga implícito. No existe gateway centralizado; cada Slave es autónomo para atender usuarios.

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 2: 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 $\{P_1, ..., P_n\}$ 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

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 3: 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

Protocolo de timestamps de Lamport: El sistema utiliza relojes lógicos para establecer un ordenamiento parcial de eventos distribuidos. Cada nodo mantiene un contador L_i que se incrementa en cada evento local. Cuando un nodo envía un mensaje, primero incrementa

su reloj ($L_i := L_i + 1$) y adjunta este timestamp al mensaje. Al recibir un mensaje con timestamp T, el nodo receptor actualiza su reloj tomando el máximo entre su valor actual y el recibido, luego lo incrementa: $L_i := \max(L_i, T) + 1$. Este mecanismo garantiza que si un evento a causalmente precede a un evento b, entonces el timestamp de a será menor que el de 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 $node_1 < node_2$.

7.3 Toma de decisiones distribuidas

7.3.1 Consenso para operaciones críticas

Escenario: Decidir qué nodo se encarga de una tarea (ej: convertirse en coordinador de zona).

Algoritmo de elección de líder (simplificado):

1. Cuando se detecta ausencia de coordinador, cada nodo calcula prioridad: $P_i = h(\text{node\_id}_i, \text{load}_i, \text{uptime}_i)$
2. Cada nodo anuncia su prioridad a sus vecinos
3. Nodo con mayor prioridad y confirmación de $\geq 2/3$ vecinos se designa líder
4. Líder anuncia su rol a toda la red mediante flooding

Consenso para cambios de configuración:

- Cambios críticos (ej: modificar factor de replicación) requieren consenso
- Se usa votación de mayoría simple o algoritmo Raft simplificado
- Solo se aplica cambio si $> 50\%$ de nodos activos aprueban

8 Localización y nombrado de datos / recursos

Cada documento en el sistema tiene un identificador único (UUID generado al subir) y un **embedding semántico** de 384 dimensiones que representa su contenido. El mecanismo de localización se basa en **similitud semántica** en lugar de funciones hash, lo que permite ubicar documentos según su significado y no según propiedades arbitrarias de su identificador.

El Master mantiene un **índice de ubicación semántica** que mapea cada Slave a un perfil semántico agregado de sus documentos. Cuando llega una consulta, el Master calcula su embedding y determina qué Slaves tienen contenido semánticamente similar.

8.1 Estrategias de localización de datos

8.1.1 Estrategia 1: Búsqueda semántica centralizada

Algorithm 2 Búsqueda semántica en Master

Input: *query* (texto de consulta), *top_k* (número de resultados)
Output: lista de documentos relevantes
q_embedding ← calcular_embedding(*query*)
slaves ← ordenar_slaves_por_similitud(*q_embedding*)
results ← []
for *slave* en *slaves*[: 3] **do** ▷ Top 3 Slaves más relevantes
    local_results ← enviar_query(*slave*, *query*)
    results ← *results* ∪ *local_results*
end for
return ordenar_por_relevancia(*results*)[:*top_k*]

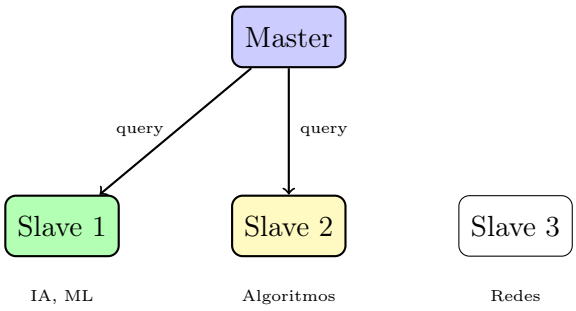
8.1.2 Estrategia 2: Asignación por afinidad semántica

Asignar cada documento al Slave cuyo perfil semántico es más similar:

$$\text{slave\_destino} = \arg\max_{s \in \text{slaves}} \cos(embedding_{doc}, profile_s)$$

donde cos es la similitud coseno entre vectores.

Ejemplo: Un documento sobre "algoritmos de ordenamiento" tiene un embedding 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 será semánticamente cercano, por lo que el nuevo documento se almacena allí.



Consulta "sorting algorithms" → Slave 2 (más relevante)

Figure 7: Ubicación semántica: Master enruta consultas a Slaves con contenido similar

9 Distribución, replicación y consistencia de datos

El sistema implementa replicación de datos donde cada documento puede tener k réplicas en diferentes Slaves (por defecto $k = 2$) para garantizar tolerancia a fallos y alta disponibilidad.

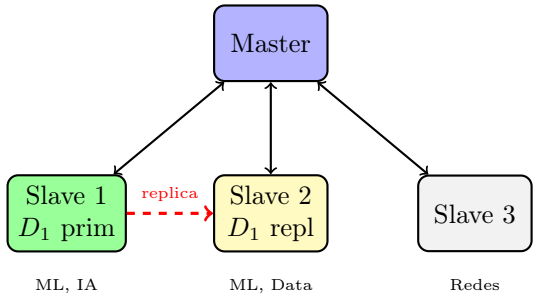
El coordinador de replicación en el Master selecciona Slaves con **afinidad semántica**: documentos similares se replican en Slaves que ya contienen contenido relacionado, mejorando la localidad de las consultas.

Respecto a la consistencia, los documentos son inmutables una vez subidos, simplificando el modelo al eliminar la necesidad de sincronizar actualizaciones.

9.1 Estrategia de replicación

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 sea similar al embedding del documento
- **Carga actual**: Balancear distribución de almacenamiento entre Slaves
- **Disponibilidad**: Priorizar Slaves con historial de uptime alto



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

Figure 8: 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** con las siguientes características:

- **Escrituras**: Documentos nuevos se escriben primero en el Slave que recibe la subida, luego se replican
- **Lecturas**: Cualquier Slave con una réplica puede responder consultas
- **Convergencia**: Garantizada por la naturaleza inmutable de los documentos

10 Tolerancia a fallos, robustez y dinámicas de red

El sistema proporciona soporte integral para fallos parciales, incluyendo: Slaves que se desconectan temporalmente, fallo del Master (recuperado mediante elección Bully), y reincorporación de nodos nuevos o recuperados.

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.

Cuando un Slave nuevo se une a la red o un Slave existente falla:

- **Nuevo Slave**: Se registra con el Master enviando su perfil semántico inicial
- **Fallo de Slave**: El Master detecta ausencia de heartbeats y marca el nodo como no disponible
- **Fallo de Master**: Slaves detectan timeout, inician elección Bully, nuevo Master asume coordinación
- **Rerreplicación**: Si un Slave con réplicas falla, el Master coordina crear nuevas réplicas en otros Slaves

10.1 Detección de fallos mediante Heartbeat

Algorithm 3 Protocolo Heartbeat para detección de fallos

Constantes: $T_{heartbeat} = 5s$, $T_{timeout} = 15s$
while nodo está activo **do**
    for cada vecino v en lista de vecinos **do**
        Enviar `PING` a v
        if no se recibe `PONG` en $T_{timeout}$ **then**
            Marcar v como fallido
            Notificar a otros vecinos sobre fallo de v
            Intentar reconexión o buscar reemplazo
        end if
    end for
    Esperar $T_{heartbeat}$
end while

10.2 Protocolo de incorporación de nuevo nodo (JOIN)

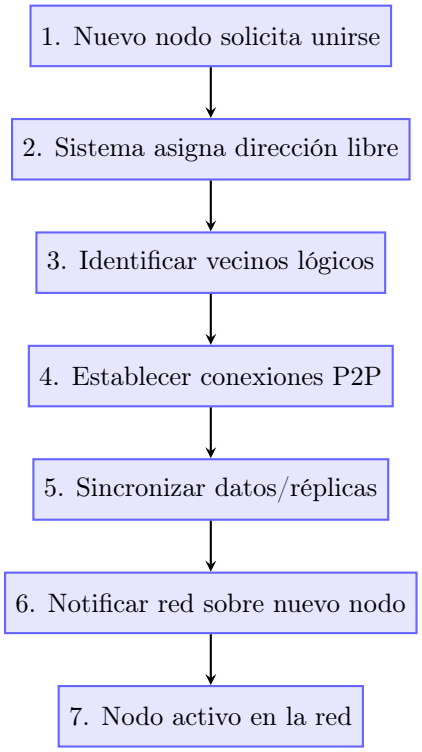


Figure 9: Proceso de incorporación de nuevo nodo a la red P2P

10.3 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

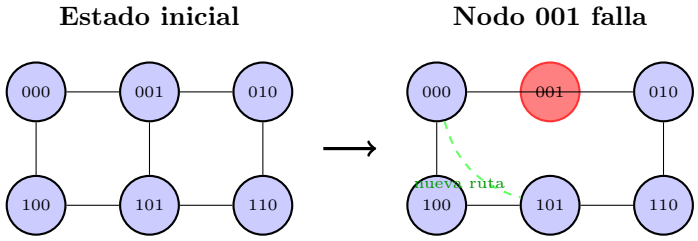


Figure 10: Reconfiguración de red tras fallo de nodo 001: rutas alternativas y reconexiones

10.4 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

El sistema implementa múltiples capas de seguridad para proteger tanto los datos como la integridad de la red. Cada nodo posee una identidad única basada en criptografía de clave pública (par de claves pública/privada), lo que permite autenticación mutua entre peers antes de establecer comunicación. Toda la comunicación entre nodos se realiza mediante canales cifrados usando TLS (Transport Layer Security), protegiendo los datos en tránsito contra escuchas y manipulación.

El sistema implementa control de acceso mediante permisos que definen qué nodos pueden leer o escribir ciertos datos. Esto se logra usando firmas digitales para verificar autoriz ación, listas de control de acceso (ACLs) basadas en identidad de nodo, y certificados digitales que vinculan identidades con permisos específicos. Para mitigar la presencia de nodos maliciosos, el sistema emplea múltiples mecanismos: verificación de integridad mediante hashes criptográficos (SHA-256) y firmas digitales, sistemas de reputación que rastrean el comportamiento histórico de los nodos, validación de datos mediante consenso entre múltiples peers, limitación de privilegios siguiendo el principio de mínimo privilegio, y protección contra ataques Sybil mediante pruebas de identidad y restricción de nuevas incorporaciones.

11.1 Modelo de seguridad por capas

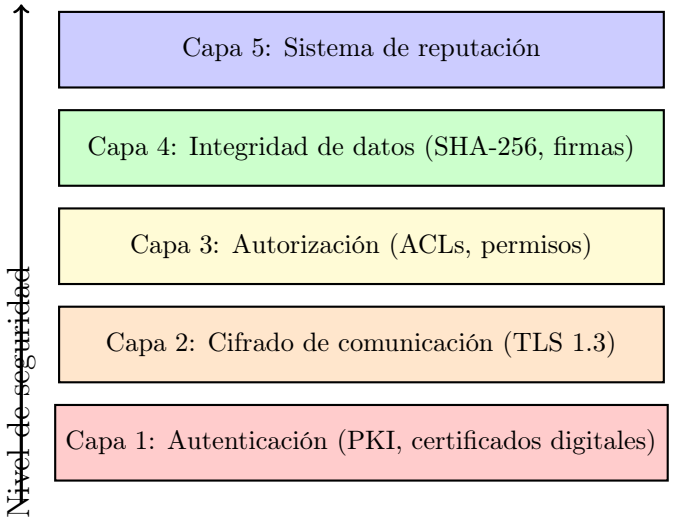


Figure 11: Arquitectura de seguridad por capas

11.2 Protocolo de autenticación

Algorithm 4 Autenticación mutua entre peers

Nodo A quiere conectarse con Nodo B
A genera nonce N_A aleatorio
A $\rightarrow$ B: `HELLO`(ID_A, N_A, $Cert_A$)
B verifica $Cert_A$ con autoridad certificadora
if $Cert_A$ es válido **then**
    B genera nonce N_B
    B $\rightarrow$ A: `CHALLENGE`(N_B, $Sign_B(N_A)$, $Cert_B$)
    A verifica $Cert_B$ y $Sign_B(N_A)$
    if verificación exitosa **then**
        A $\rightarrow$ B: `RESPONSE`($Sign_A(N_B)$)
        B verifica $Sign_A(N_B)$
        Conexión autenticada establecida
        Establecer canal TLS con claves de sesión
    end if
end if

11.3 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 4: Descriptor de seguridad de datos

11.4 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.

11.5 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 5: 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 6: 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 7: Métricas de rendimiento del sistema

13 Conclusión

Este documento ha presentado la especificación arquitectónica completa de un sistema distribuido Master-Slave con ubicación de recursos por vectorización semántica. El diseño prioriza cinco aspectos fundamentales:

Coordinación centralizada con tolerancia a fallos: El Master coordina el clúster manteniendo el índice semántico de ubicación 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.

Ubicación semántica de recursos: En lugar de funciones hash (DHT), el sistema utiliza embeddings de 384 dimensiones generados por sentence-transformers para representar documentos. Esto permite localización basada en similitud de contenido, mejorando la relevancia de búsquedas y la localidad de datos relacionados.

Alta disponibilidad: Cada Slave es autónomo con su propio backend, frontend y base de datos MongoDB. Los usuarios pueden acceder a cualquier Slave directamente, y la replicación por afinidad semántica garantiza que los datos sobrevivan fallos individuales.

Escalabilidad horizontal: Agregar Slaves es trivial: el nuevo nodo se registra con el Master, reporta su perfil semántico, y comienza a recibir consultas y datos. El Master actualiza dinámicamente su índice de ubicación.

Simplicidad operativa: La arquitectura Master-Slave es conceptualmente más simple que topologías P2P complejas como hipercubo, facilitando debugging, monitoreo y mantenimiento del sistema en producción.

Referencias

- Sentence-Transformers: documentación oficial para generación de embeddings semánticos.
- Algoritmo Bully: Garcia-Molina, H. "Elections in a Distributed Computing System" (1982).
- CoreDNS: documentación oficial para resolución DNS en entornos containerizados.
- FastAPI y Streamlit: frameworks utilizados para backend y frontend respectivamente.