# Sistema Distribuido P2P con Topología Hipercubo

Abel Ponce González C411
Richard Alejandro Matos Arderí C411

November 30, 2025

**Abstract**

Este documento presenta la especificación arquitectónica de un sistema distribuido peer-to-peer descentralizado basado en topología hipercubo. Se detalla la arquitectura del sistema, la organización y roles de sus componentes, los procesos y patrones de diseño implementados, los mecanismos de comunicación y coordinación, el sistema de nombrado y localización de recursos, las estrategias de consistencia y replicación, los mecanismos de tolerancia a fallos y los aspectos de seguridad del diseño.

# Contents

# 1 Introducción

## 1.1 Descripción del sistema

El sistema propuesto es una plataforma distribuida peer-to-peer para almacenamiento y consulta de información de forma descentralizada. Utiliza una topología de red overlay tipo hipercubo que estructura la organización lógica de los nodos, proporcionando rutas eficientes de comunicación, redundancia y tolerancia a fallos.

## 1.2 Objetivos del diseño

- Eliminar puntos únicos de fallo mediante una arquitectura completamente descentralizada
- Garantizar alta disponibilidad de datos mediante replicación estratégica
- Proporcionar mecanismos eficientes de localización de recursos sin índices centralizados
- Implementar tolerancia a fallos con recuperación automática
- Asegurar comunicaciones mediante protocolos de seguridad robustos

# 2 Arquitectura del Sistema

## 2.1 Estilo arquitectónico

### 2.1.1 Clasificación según estilos arquitectónicos

El sistema sigue un **estilo arquitectónico peer-to-peer estructurado** que se caracteriza por una organización simétrica de componentes con capacidades equivalentes. Los componentes principales son nodos peer que exponen interfaces bien definidas para operaciones de almacenamiento, consulta y routing. Cada componente encapsula datos y ofrece métodos sin revelar su implementación interna, siguiendo el estilo basado en objetos. Esta encapsulación permite que los componentes sean reemplazables mientras mantengan la interfaz estándar del sistema.

La comunicación entre componentes se realiza mediante conectores que incluyen sockets TCP/UDP para comunicación punto a punto confiable, mecanismos de mensajería asíncrona para coordinación entre nodos, y streaming de datos para transferencias grandes. La configuración del sistema está determinada por una topología overlay hipercubo que define las conexiones lógicas entre nodos, creando una red estructurada con un índice distribuido basado en direcciones binarias.

## 2.2 Modelo arquitectónico: P2P estructurado descentralizado

### 2.2.1 Características del sistema P2P

El sistema implementa una arquitectura peer-to-peer **estructurada** pura sin componentes centralizados. La característica fundamental es la simetría de roles: cada nodo peer actúa

simultáneamente como cliente y servidor (patrón servant). Esto significa que un mismo nodo puede iniciar consultas hacia otros peers (actuando como cliente) y al mismo tiempo atender solicitudes de otros nodos (actuando como servidor). No existe distinción funcional entre los nodos; todos tienen capacidades equivalentes para almacenar fragmentos de datos y réplicas, procesar consultas tanto propias como de otros nodos, reenviar peticiones según algoritmos de routing, participar en protocolos de replicación y consistencia, y mantener conexiones activas con sus vecinos lógicos en la topología.

El sistema utiliza un índice distribuido semánticamente libre donde cada dato se asocia con una clave única derivada de las direcciones binarias en el hipercubo. El sistema emplea direccionamiento directo basado en la estructura del hipercubo, donde cada nodo es responsable de mantener pares (clave, valor) que corresponden a su región del espacio de direcciones. Esto crea un índice distribuido sin necesidad de un servidor central de índices, aprovechando las propiedades geométricas del hipercubo para routing eficiente.

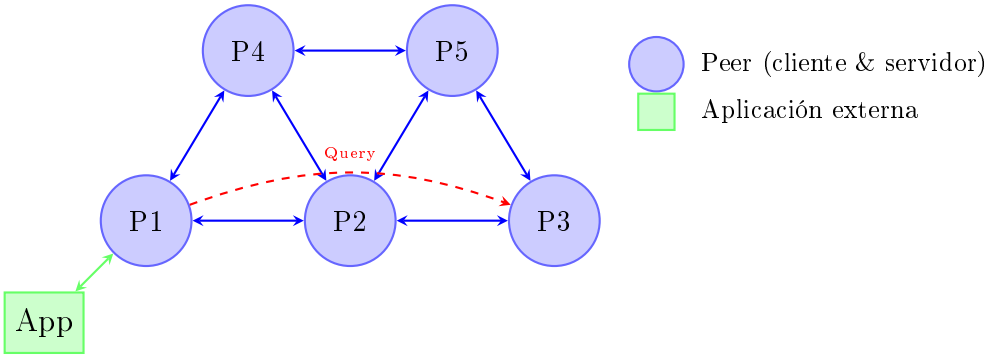


Figure 1: Arquitectura P2P: cada peer actúa como cliente y servidor. P1 puede consultar a P3 (cliente) mientras atiende peticiones de otros (servidor)

## 2.3 Distribución del sistema

### 2.3.1 Distribución horizontal

El sistema emplea **distribución horizontal** (horizontal distribution), donde los nodos peer están físicamente distribuidos en múltiples máquinas pero mantienen capacidades equivalentes. Cada nodo opera sobre su propio subconjunto del conjunto completo de datos, sin que exista una jerarquía vertical de capas entre los peers. Todos los nodos ejecutan el mismo software y pueden asumir cualquier rol necesario, lo que garantiza simetría funcional en el sistema. Esta distribución horizontal contrasta con la distribución vertical (donde diferentes capas de aplicación residen en diferentes máquinas), enfatizando la naturaleza descentralizada de la arquitectura P2P.

## 2.4 Topología de red overlay: Hipercubo estructurado

### 2.4.1 Definición de la topología

La red lógica (overlay) se estructura como un hipercubo de dimensión $d$. En esta topología, cada nodo posee una dirección binaria única de $d$ bits que determina su posición en el espacio

lógico. Los vecinos lógicos de un nodo son aquellos cuya dirección difiere en exactamente un bit, lo que significa que cada nodo mantiene $d$ conexiones directas. El número máximo de nodos que puede soportar el sistema es $N = 2^d$, y la distancia máxima entre cualquier par de nodos (diámetro de la red) es también $d$.

Esta estructura presenta propiedades clave que la hacen ideal para sistemas distribuidos. La **conectividad** múltiple garantiza redundancia, ya que existen varias rutas alternativas entre cualquier par de nodos. La **escalabilidad** se manifiesta en el crecimiento logarímico del diámetro respecto al número de nodos, lo que mantiene las latencias manejables incluso con miles de nodos. La **simetría** asegura que todos los nodos tienen el mismo grado de conectividad, evitando cuellos de botella. Finalmente, la **tolerancia a fallos** se logra porque la red permanece conectada incluso ante fallos de nodos individuales, gracias a las rutas alternativas disponibles.

### 2.4.2 Ejemplo: Hipercubo de dimensión 3

Un hipercubo de dimensión 3 ($d = 3$) tiene $2^3 = 8$ nodos, cada uno con una dirección binaria de 3 bits (000, 001, 010, ..., 111). Cada nodo tiene exactamente 3 vecinos (uno por cada bit que puede cambiar).

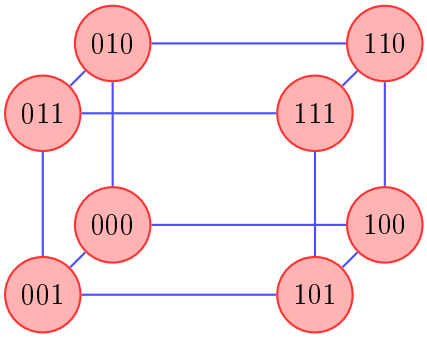


Figure 2: Hipercubo de dimensión 3: cada nodo tiene 3 vecinos (difieren en 1 bit)

### 2.4.3 Propiedades matemáticas del hipercubo

Para un hipercubo de dimensión $d$:

- **Número de nodos**: $N = 2^d$
- **Grado de cada nodo**: $d$ (cada nodo tiene $d$ vecinos)
- **Diámetro de la red**: $d$ (máximo número de saltos entre dos nodos cualesquiera)
- **Número total de aristas**: $\frac{N \cdot d}{2} = d \cdot 2^{d-1}$
- **Conectividad**: $d$ (número de caminos disjuntos entre nodos)

**Ejemplo de routing**: Para enviar un mensaje del nodo 000 al nodo 111:

1. Comparar direcciones bit a bit: 000 vs 111 $\rightarrow$ difieren en 3 bits

2. Ruta posible: 000 → 100 → 110 → 111 (3 saltos)

3. Existen múltiples rutas alternativas de longitud 3

# 3 Roles y organización funcional

En el sistema P2P propuesto, el concepto de "rol" es fluido debido a la naturaleza simétrica de la arquitectura. El componente fundamental es el **nodo peer**, que simultáneamente actúa como cliente y servidor. Cuando un peer necesita localizar un dato que no posee localmente, asume el rol de **cliente** iniciando consultas hacia otros peers. Al mismo tiempo, ese mismo peer puede estar atendiendo solicitudes de otros nodos, actuando como **servidor**. Esta dualidad cliente-servidor es la esencia del modelo peer-to-peer y permite que el sistema funcione sin depender de servidores dedicados.

Cuando un peer almacena datos, responde peticiones y participa en el routing de mensajes, está ejerciendo su rol de servidor. Cuando ese mismo peer busca información, solicita datos a otros nodos o envía consultas a través de la red, está ejerciendo su rol de cliente. Es importante destacar que estas funciones no son excluyentes sino concurrentes: un peer puede estar procesando múltiples peticiones entrantes (servidor) mientras simultáneamente envía sus propias consultas a otros nodos (cliente).

Adicionalmente, el sistema posee un **balanceo de carga** que esta lógica no constituye un servidor central sino un algoritmo que cada peer utiliza para tomar decisiones sobre cómo distribuir sus peticiones entre vecinos, considerando factores como carga actual, latencia y disponibilidad. Esto lo asume un pequeño conjunto de peers tomando responsabilidades adicionales de coordinación de manera redundante, sin romper la descentralización del sistema.

## 3.1 Diagrama de roles e interacciones

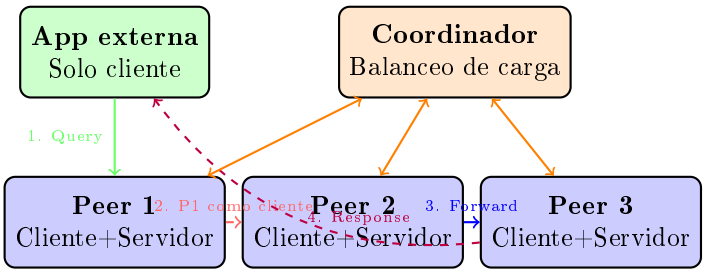


Figure 3: Interacción: app externa (solo cliente) vs peers (cliente y servidor). P1 actúa como servidor para app, y como cliente al consultar P2

### 3.2 Responsabilidades de cada rol

| Rol | Responsabilidades |
|---|---|
| **Peer** | **Como servidor:** Almacenar fragmentos de datos y réplicas, procesar consultas locales de otros peers o apps externas, reenviar consultas según algoritmo de routing, mantener conexiones con vecinos lógicos, participar en protocolos de replicación, reportar métricas de carga y salud.<br>**Como cliente:** Iniciar búsquedas de datos no disponibles localmente, formular consultas hacia otros peers, manejar timeouts y reintentos, procesar respuestas recibidas. |
| **Aplicación externa** | Formular consultas de datos sin ofrecer capacidad de servidor, conectarse al gateway o directamente a peers designados, manejar timeouts y reintentos, procesar respuestas recibidas. No almacena datos ni participa en routing. |
| **Coordinador** | Monitorear carga de peers, distribuir consultas según políticas de balanceo, detectar peers sobrecargados o caídos, ajustar dinámicamente asignación de tráfico, recopilar estadísticas del sistema. |

Table 1: Responsabilidades por rol: peers tienen doble naturaleza (cliente y servidor)

## 4 Despliegue y distribución de servicios con Docker

El despliegue se implementa usando contenedores Docker con dos redes separadas:

### 4.1 Arquitectura de redes Docker

**p2p-internal** Red bridge interna donde residen los peers del hipercubo. Los peers se comunican entre sí para routing, replicación y búsqueda distribuida.

**frontend-net** Red bridge pública conectada al gateway/balanceador. Expuesta al host para acceso externo (puerto 8080).

El gateway actúa como puente entre ambas redes: conectado a `frontend-net` para recibir peticiones externas y a `p2p-internal` para distribuirlas a los peers.

### 4.2 Comandos de despliegue

**1. Crear redes Docker:**

```
docker network create p2p-internal
docker network create frontend-net
```

**2. Desplegar peers en red interna:**

```
docker run -d --name peer-000 --network p2p-internal \
  -e PEER_ID=000 peer-image:latest
docker run -d --name peer-001 --network p2p-internal \
  -e PEER_ID=001 peer-image:latest
docker run -d --name peer-010 --network p2p-internal \
  -e PEER_ID=010 peer-image:latest
docker run -d --name peer-011 --network p2p-internal \
  -e PEER_ID=011 peer-image:latest
```

3. **Desplegar gateway conectado a ambas redes:**

```
docker run -d --name gateway --network frontend-net \
  -p 8080:8080 gateway-image:latest
docker network connect p2p-internal gateway
```

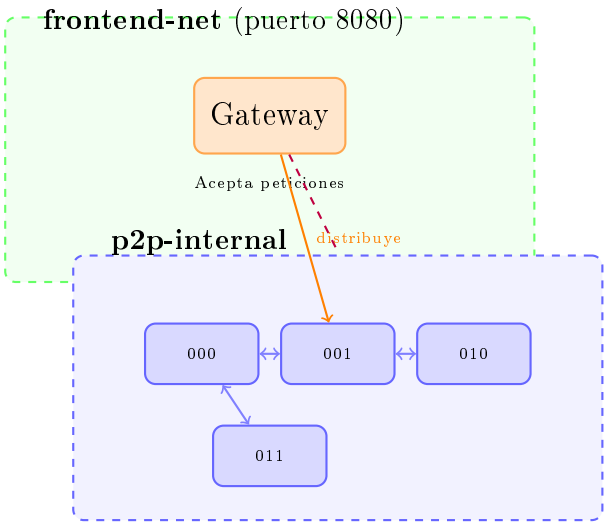


Figure 4: Distribución en dos redes Docker: gateway conecta frontend-net (pública) con p2p-internal (privada)

## 4.3 Ventajas del diseño

- **Aislamiento**: Peers protegidos en red interna sin exposición directa
- **Escalabilidad**: Nuevos peers se agregan solo a `p2p-internal`
- **Punto de entrada único**: Gateway gestiona autenticación y balanceo
- **Separación de responsabilidades**: Frontend vs lógica P2P distribuida

# 5 Procesos y patrones de diseño

Cada nodo peer ejecuta uno o más procesos que proporcionan servicios especializados: un servicio de almacenamiento gestiona los datos locales y réplicas, un servicio de red se encarga del routing y escucha de conexiones entrantes, y un servicio de manejo de peticiones procesa las solicitudes tanto propias como de otros nodos. Si el sistema implementa lógica de balanceo o coordinación, esta puede residir como un módulo separado o subproceso dentro de algunos peers, ejecutándose de forma redundante para evitar puntos únicos de fallo.

En términos de concurrencia, cada nodo utiliza hilos o programación asíncrona (async I/O) para manejar múltiples conexiones y peticiones simultáneas sin bloqueos. El sistema adopta un patrón de diseño basado en concurrencia, asincronía y arquitectura dirigida por eventos (event-driven) con paso de mensajes (message-passing). Este enfoque es especialmente adecuado para sistemas distribuidos de red donde la latencia variable y las operaciones I/O dominan el tiempo de ejecución.

## 5.1 Arquitectura interna de un nodo peer

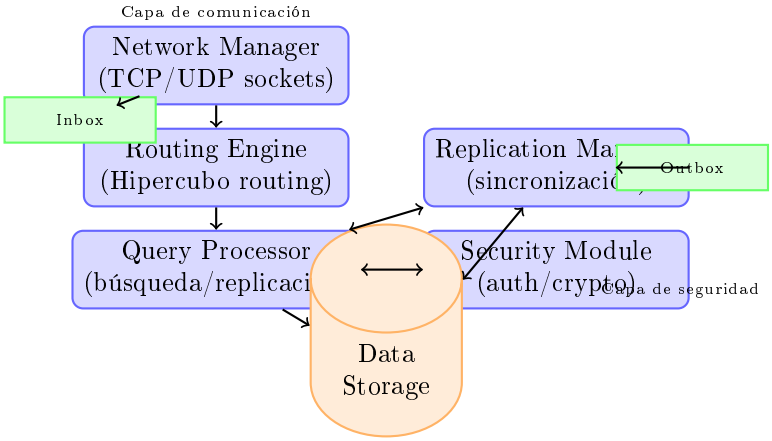


Figure 5: Arquitectura modular de un nodo peer: actúa como cliente (Query Processor inicia búsquedas) y servidor (Network Manager atiende peticiones)

## 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

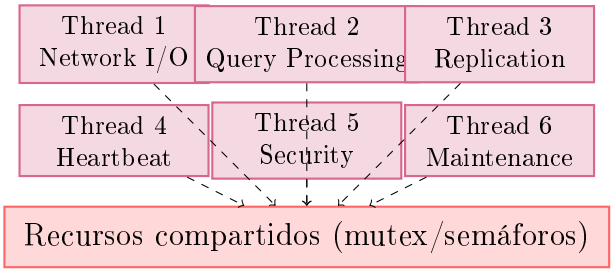


Figure 6: Modelo de threads para manejo concurrente de operaciones

## 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 `REQUEST`, `RESPONSE`, `REPLICATE` y `HEARTBEAT`. Esta capa contiene la lógica de routing y búsqueda basada en la topología hipercubo, así como la gestión de sesiones y estado de transacciones. Cada peer interpreta estos mensajes y toma decisiones sobre cómo procesarlos o reenviarlos.

La **capa de transporte** utiliza sockets TCP para comunicación confiable en operaciones que requieren entrega garantizada, como transferencia de datos y replicación. Para mensajes de control de baja latencia que toleran pérdidas ocasionales, como heartbeats y descubrimiento de vecinos, se emplean sockets UDP. Esta capa también gestiona conexiones persistentes entre vecinos lógicos, manteniendo un pool de conexiones activas para minimizar la latencia de establecimiento.

La **capa de red** utiliza routing IP estándar sobre Internet. Es importante destacar que la topología overlay hipercubo es completamente independiente de la red física subyacente: dos nodos que son vecinos lógicos en el hipercubo pueden estar físicamente en diferentes continentes, mientras que nodos físicamente cercanos pueden no ser vecinos lógicos.

### 6.2 Tipos de comunicación

#### 6.2.1 Comunicación peer-to-peer directa

Los nodos se comunican directamente mediante sockets TCP/UDP sin intermediarios. Cada peer mantiene conexiones activas con sus $d$ vecinos lógicos determinados por la topología hipercubo, lo que garantiza que cualquier peer pueda ser alcanzado en máximo $d$ saltos. Para operaciones síncronas como consultas de datos, se utiliza un modelo petición-respuesta (request-reply) donde el peer cliente espera una respuesta del peer servidor. Para operaciones asíncronas como replicación de datos y notificaciones de cambios de estado, se emplea mensajería asíncrona donde el emisor no bloquea esperando confirmación inmediata.

### 6.2.2 Comunicación cliente-sistema

Las aplicaciones externas que no forman parte de la red P2P (y por tanto no actúan como servidores) pueden acceder al sistema de dos formas. Pueden conectarse a través de un gateway que actúa como punto de entrada y balanceador de carga, redirigiendo peticiones a la red P2P interna. También un cliente externo puede comunicarse directamente con cualquier peer específico. En ambos casos, el cliente externo solo actúa como cliente, a diferencia de los peers que simultáneamente son clientes y servidores.

## 6.3 Protocolo de mensajería

Se definen los siguientes tipos de mensajes:

| **Mensaje** | **Descripción** |
|---|---|
| `QUERY(key, ttl, origin)` | Búsqueda de dato con clave `key`, tiempo de vida `ttl`, nodo origen |
| `QUERY_HIT(key, data, holder)` | Respuesta con el dato encontrado y nodo que lo posee |
| `STORE(key, data, replicas)` | Almacenar dato con número de réplicas especificado |
| `REPLICATE(key, data, target)` | Replicar dato hacia nodo objetivo |
| `PING / PONG` | Verificación de disponibilidad entre vecinos |
| `JOIN(node_id, address)` | Notificación de unión de nuevo nodo |
| `LEAVE(node_id)` | Notificación de salida de nodo |

Table 2: Tipos de mensajes del protocolo P2P

## 6.4 Algoritmo de routing basado en distancia Hamming

Para enviar un mensaje desde el nodo $A$ al nodo $B$:

**Algorithm 1** Routing en hipercubo

**Input:** nodo actual *current*, nodo destino *target*
**Output:** siguiente salto en la ruta
$diff \leftarrow current \oplus target$ ▷ XOR bit a bit
**if** $diff = 0$ **then**
    **return** *target* ▷ Ya llegamos
**end if**
*bit* ← primer bit en 1 de $diff$ (de derecha a izquierda)
*next* ← *current* con el bit *bit* invertido
**return** *next*

**Ejemplo**: Para ilustrar el routing desde el nodo 000 al nodo 111, primero calculamos la diferencia: $diff = 000 \oplus 111 = 111$, lo que indica que difieren en los tres bits. El algoritmo invierte el bit menos significativo primero, moviendo de $000 \rightarrow 001$. Desde 001, la nueva diferencia es $diff = 001 \oplus 111 = 110$, por lo que invertimos el siguiente bit: $001 \rightarrow 011$. Finalmente, desde 011 calculamos $diff = 011 \oplus 111 = 100$ e invertimos el

bit más significativo: $011 \rightarrow 111$, alcanzando el destino en exactamente 3 saltos (igual a la distancia Hamming entre ambos nodos).

# 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 dato en el sistema tiene un identificador global único (ID o hash criptográfico) que lo distingue de otros elementos. El mecanismo de localización aprovecha la topología overlay del hipercubo combinada con lógica inteligente de búsqueda y routing. El sistema ofrece tres estrategias principales de localización.

La **búsqueda mediante reenvío** funciona cuando un peer recibe una petición de dato que no posee localmente: reenvía la consulta a sus vecinos lógicos, quienes a su vez propagan la búsqueda hasta encontrar una réplica del dato solicitado. La **búsqueda dirigida** utiliza heurísticas para optimizar el proceso, realizando encaminamiento preferente hacia nodos "más cercanos" según distancia de Hamming, seleccionando nodos con menor carga actual, o priorizando nodos con alto historial de disponibilidad.

## 8.1 Estrategias de localización de datos

### 8.1.1 Estrategia 1: Búsqueda por inundación controlada (Flooding)

**Algorithm 2** Búsqueda por flooding con TTL

**Input:** *key* (clave del dato), *ttl* (tiempo de vida)
**Output:** dato encontrado o NULL
**if** dato con *key* está en almacenamiento local **then**
    **return** dato
**end if**
**if** $ttl \leq 0$ **then**
    **return** NULL
**end if**
Enviar `QUERY(`*key*`,` $ttl - 1$`)` a todos los vecinos
Esperar respuestas con timeout
**return** primera respuesta válida o NULL

### 8.1.2 Estrategia 2: Asignación determinística basada en hash

Asignar cada dato al nodo cuya dirección es más cercana al hash del dato:

$$\text{nodo\_destino} = \arg\min_{i \in \text{nodos}} d_H(\text{hash}(key) \mod 2^d, i)$$

donde $d_H$ es la distancia de Hamming.

**Ejemplo** (hipercubo $d = 3$): Consideremos un dato cuyo hash es 42. Para mapearlo al espacio de direcciones de dimensión 3, calculamos $42 \mod 8 = 2$, que en binario es 010. Por lo tanto, este dato se almacena preferentemente en el nodo con dirección 010, y sus réplicas se distribuyen en nodos vecinos para garantizar disponibilidad y tolerancia a fallos.

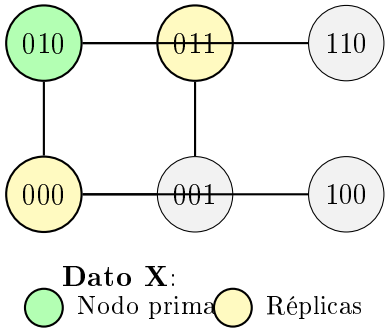


Figure 7: Distribución de dato X: nodo primario 010 con réplicas en vecinos 000 y 011

# 9 Distribución, replicación y consistencia de datos

El sistema implementa replicación de datos donde cada elemento puede tener $k$ réplicas en diferentes nodos (típicamente $k = 2$ o $k = 3$) para garantizar tolerancia a fallos y alta

disponibilidad. El protocolo de replicación funciona asignando, al momento de insertar un dato, las réplicas a nodos distintos que idealmente estén "lejanos" en la topología hipercubo. Esta distancia lógica maximiza la probabilidad de que las réplicas sobrevivan a fallos correlacionados.

Respecto a la consistencia, los datos son inmutables, entonces el problema se simplifica ya que no hay sincronización de actualizaciones.

## 9.1 Estrategia de replicación

Para un factor de replicación $k = 3$, al almacenar un dato en el nodo primario $N_p$, el sistema selecciona $k - 1 = 2$ nodos adicionales para almacenar réplicas. Los criterios de selección incluyen maximizar la distancia de Hamming entre réplicas para lograr dispersión geográfica lógica, preferir nodos con baja carga actual para balancear el almacenamiento, y priorizar nodos con alta disponibilidad histórica según métricas de uptime. Una vez seleccionados los nodos, el sistema envía mensajes `REPLICATE` y espera confirmación de almacenamiento antes de considerar la operación completa.

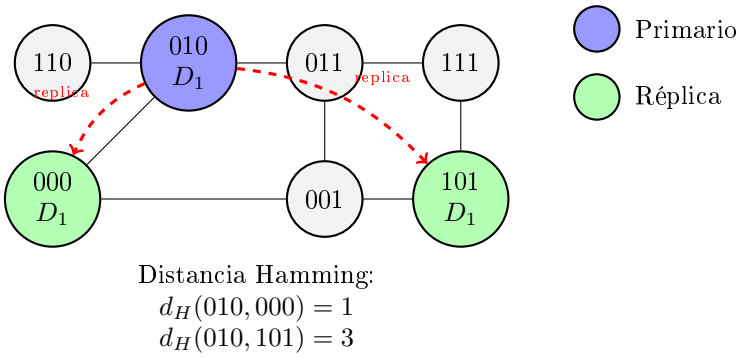


Figure 8: Replicación de dato $D_1$ con factor $k = 3$: nodo primario 010 y réplicas en 000 y 101

## 9.2 Protocolo de consistencia eventual

**Algorithm 3** Actualización con consistencia eventual

1: **Input:** *key*, *new_value*, *timestamp*
2: Actualizar valor local si *timestamp* es más reciente
3: Obtener lista de réplicas para *key*
4: **for** cada réplica $r$ en lista **do**
5: Enviar `UPDATE(`*key*`,` *new_value*`,` *timestamp*`)` a $r$
6: No esperar confirmación (asíncrono)
7: **end for**
8: Programar verificación de consistencia periódica

## 9.3 Modelo de consistencia

El sistema adopta un modelo de consistencia eventual con las siguientes características. Para operaciones de **lectura**, el sistema acepta respuestas de cualquier réplica disponible, lo que significa que un cliente puede obtener una versión que no sea la más reciente si otras réplicas aún no han sido actualizadas

La **convergencia** entre réplicas está garantizada eventualmente mediante timestamps lógicos y mecanismos de anti-entropy que periódicamente detectan y corrigen divergencias.

# 10 Tolerancia a fallos, robustez y dinámicas de red

El sistema proporciona soporte integral para fallos parciales, incluyendo nodos que se desconectan temporalmente, nodos que fallan permanentemente, y la reincorporación de nodos nuevos o recuperados. La topología de hipercubo es fundamental para la resiliencia ya que ofrece múltiples rutas alternativas entre cualquier par de nodos, proporcionando redundancia natural ante fallos de nodos o enlaces individuales.

Cuando un nodo nuevo se une a la red o un nodo existente sale, es necesario reconfigurar las conexiones lógicas del sistema. Esto implica asignar o actualizar la dirección del nodo en el hipercubo, establecer o terminar conexiones con sus vecinos lógicos determinados por diferencias de un bit, reasignar réplicas si el nodo que falla era responsable de datos críticos, y actualizar las tablas de routing para reflejar la nueva topología.

El manejo de errores se implementa mediante múltiples estrategias complementarias. Se configuran timeouts para detectar nodos no responsivos, se reintentan peticiones fallidas dirigiéndolas hacia vecinos alternativos en la topología, y se implementa fallback automático a réplicas cuando el nodo primario no está disponible.

## 10.1 Detección de fallos mediante Heartbeat

**Algorithm 4** 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)

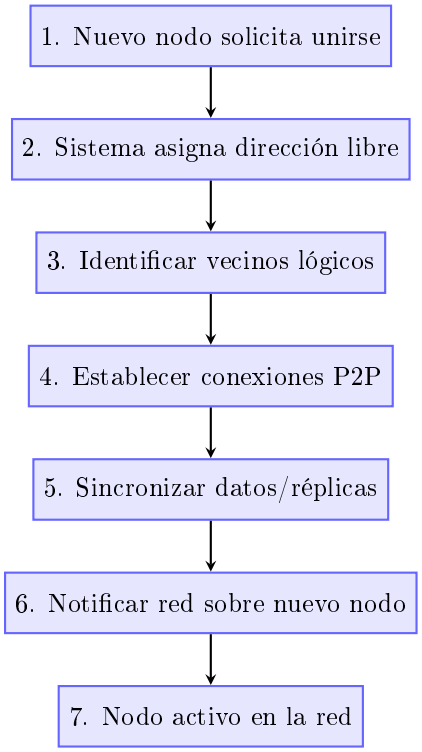


Figure 9: Proceso de incorporación de nuevo nodo a la red P2P

## 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

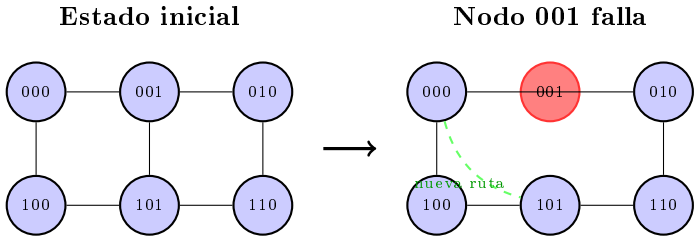


Figure 10: Reconfiguración de red tras fallo de nodo 001: rutas alternativas y reconexiones

### 10.4 Métricas de resiliencia

Para un hipercubo de dimensión $d$, el sistema presenta características de resiliencia cuantificables. La **conectividad de nodo** es igual a $d$, lo que significa que cada nodo puede tolerar hasta $d$ fallos de vecinos antes de quedar desconectado de la red. Existen $d!$ (factorial de $d$) rutas alternativas posibles entre cualquier par de nodos, proporcionando múltiples caminos de routing ante fallos. El **tiempo de detección de fallo** está determinado por el timeout de heartbeat, configurado en $T_{timeout} = 15s$. El **tiempo de recuperación** tras un fallo es del orden de $O(d \cdot T_{message})$, donde $T_{message}$ es el tiempo promedio de entrega de mensaje, ya que la información de reconfiguración debe propagarse a través de $d$ saltos máximo.

## 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

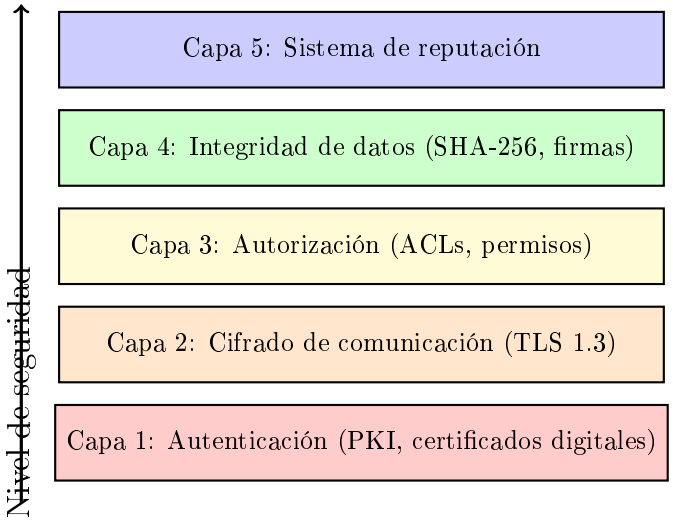


Figure 11: Arquitectura de seguridad por capas

## 11.2 Protocolo de autenticación

**Algorithm 5** 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

| **Dimensión** $d$ | **Nodos** ($2^d$) | **Diámetro** | **Vecinos/nodo** | **Escenario de uso** |
|---|---|---|---|---|
| 3 | 8 | 3 | 3 | Prototipo/Testing |
| 4 | 16 | 4 | 4 | Desarrollo |
| 5 | 32 | 5 | 5 | Despliegue piloto |
| 6 | 64 | 6 | 6 | Producción pequeña |
| 7 | 128 | 7 | 7 | Producción media |
| 8 | 256 | 8 | 8 | Producción grande |
| 10 | 1024 | 10 | 10 | Escala empresarial |

Table 6: Escalabilidad del hipercubo según dimensión

**Análisis**: La estructura hipercubo presenta características de escalabilidad excepcionales. El diámetro de la red crece logarítmicamente con el número de nodos según la fórmula $d = \log_2(N)$, lo que significa que duplicar el número de nodos solo incrementa el diámetro en uno. Cada nodo mantiene exactamente $\log_2(N)$ conexiones activas independientemente del tamaño total de la red, evitando que el overhead de conectividad crezca linealmente. Esto proporciona excelente escalabilidad para sistemas de hasta varios miles de nodos, ya que el overhead de comunicación permanece manejable para la mayoría de aplicaciones prácticas. Incluso con 1024 nodos (dimensión 10), cada nodo solo necesita mantener 10 conexiones y la distancia máxima entre nodos es de 10 saltos.

### 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 | $< 500$ms |
| 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 | $< 30$s |

Table 7: Métricas de rendimiento del sistema

## 13 Conclusión

Este documento ha presentado la especificación arquitectónica completa de un sistema distribuido P2P basado en topología hipercubo. El diseño prioriza cinco aspectos fundamentales para garantizar un sistema robusto y escalable. La **descentralización completa** elimina puntos únicos de fallo al distribuir uniformemente las responsabilidades entre todos los nodos peers, donde cada nodo actúa simultáneamente como cliente cuando solicita datos y como servidor cuando atiende peticiones de otros nodos. La **alta disponibilidad** se logra mediante replicación estratégica de datos en nodos lógicamente distantes y múltiples rutas alternativas que garantizan conectividad incluso ante fallos parciales. La **tolerancia a fallos** se implementa con detección automática de nodos caídos mediante heartbeats y mecanismos de recuperación que permiten reconexiones dinámicas. La **seguridad** abarca protocolos robustos de autenticación basados en PKI, cifrado de comunicaciones mediante TLS, y control de acceso granular con sistemas de reputación para mitigar nodos maliciosos. Finalmente, la **escalabilidad** se asegura mediante un crecimiento logarítmico de recursos por nodo, donde cada peer mantiene solo $\log_2(N)$ conexiones independientemente del tamaño total del sistema.

## Referencias


- "Peer-to-Peer (P2P) Architecture – GeeksforGeeks" — definición y características del modelo P2P.
- "Structured and Unstructured Peer-to-Peer Systems – GeeksforGeeks" — diferencias entre redes estructuradas y no estructuradas.
- Artículos sobre overlays basados en hipercubo para sistemas P2P / distribuidos.