DistriSearch

Sistema de Búsqueda Distribuida con Topología de Hipercubo

Informe Técnico - Segunda Entrega

Proyecto de Sistemas Distribuidos
Universidad [Nombre]
`repositorio: github.com/proyecto/DistriSearch`

30 de noviembre de 2025

Índice

1. Introducción

1.1. Motivación

Los motores de búsqueda modernos enfrentan el desafío de indexar y consultar volúmenes masivos de información distribuida geográficamente. DistriSearch aborda este problema mediante una arquitectura descentralizada basada en topología de **hipercubo lógico**, donde cada nodo mantiene un índice invertido local y coopera mediante protocolos de consenso para ofrecer búsqueda unificada.

1.2. Objetivos del Sistema

Según los principios de diseño de sistemas distribuidos [1]:

- **Compartición de recursos**: Índices invertidos distribuidos entre múltiples nodos
- **Transparencia**: Cliente no conoce distribución física de documentos
- **Apertura**: APIs HTTP estándar (REST) para interoperabilidad
- **Escalabilidad**: Topología hipercubo permite $O(\log N)$ saltos de ruteo

1.3. Marco Teórico

El diseño de DistriSearch se fundamenta en los siguientes conceptos teóricos:

1. **Topología de Hipercubo**: Red overlay con d dimensiones y 2^d nodos potenciales, donde cada nodo tiene exactamente d vecinos lógicos (distancia de Hamming = 1).
2. **Consenso Distribuido**: Implementación de Raft para elección de líder y replicación de log, garantizando seguridad (safety) mediante términos monotónicos.
3. **Índice Invertido**: Estructura término $\rightarrow \{doc\_id : score\}$ optimizada para consultas textuales, basada en frecuencia de términos (TF).

2. Arquitectura del Sistema

2.1. Modelo Arquitectónico

DistriSearch adopta una arquitectura **híbrida** que combina:

- **Descentralización estructural**: Red overlay de hipercubo con ruteo peer-to-peer
- **Coordinación centralizada lógica**: Data Balancer replicado mediante Raft

Esta decisión se justifica en el trade-off entre:

- Ventaja: Simplicidad en la localización de términos (vs. DHT)
- Desventaja: Punto de coordinación crítico (mitigado por replicación)

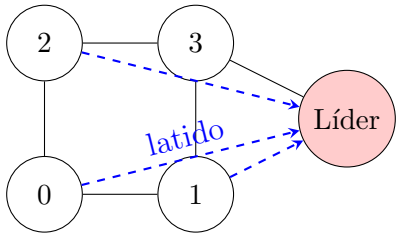

Figura 1: Vista arquitectónica: Hipercubo lógico + Data Balancer replicado

2.2. Clasificación según Tanenbaum

Característica	**DistriSearch**
Tipo	Sistema distribuido (no pervasivo, no grid)
Organización	Cliente-servidor modificado (nodos son clientes Y servidores)
Vista expansiva	Red overlay de hipercubo independiente de topología física
Vista integrativa	Coordinación mediante líder electo (Raft)
Escalabilidad	Arquitectónica: $O(\log N)$ saltos; Geográfica: NO optimizada

Cuadro 1: Clasificación del sistema

2.3. Organización del Sistema

2.3.1. Componentes Principales

1. **Nodo Distribuido** (`node/`): Unidad básica con servidor HTTP, índice local y módulos de consenso.
2. **Hipercubo Lógico** (`core/hypercube.py`): Gestión de topología y cálculo de vecinos.
3. **Ruteo XOR** (`core/routing.py`): Algoritmo greedy para minimizar distancia de Hamming.

4. **Consenso Raft** (`consensus/`): Elección de líder, replicación de log, gestión de términos.

5. **Almacenamiento** (`storage/`): Índice invertido, tokenización, persistencia en disco.

6. **Data Balancer** (`balancer/`): Índice global de términos, registro de nodos, snapshots.

7. **Replicación** (`replication/`): Gestión de réplicas con quorum, rollback transaccional.

8. **Sharding** (`sharding/`): Particionamiento hash-based del índice global.

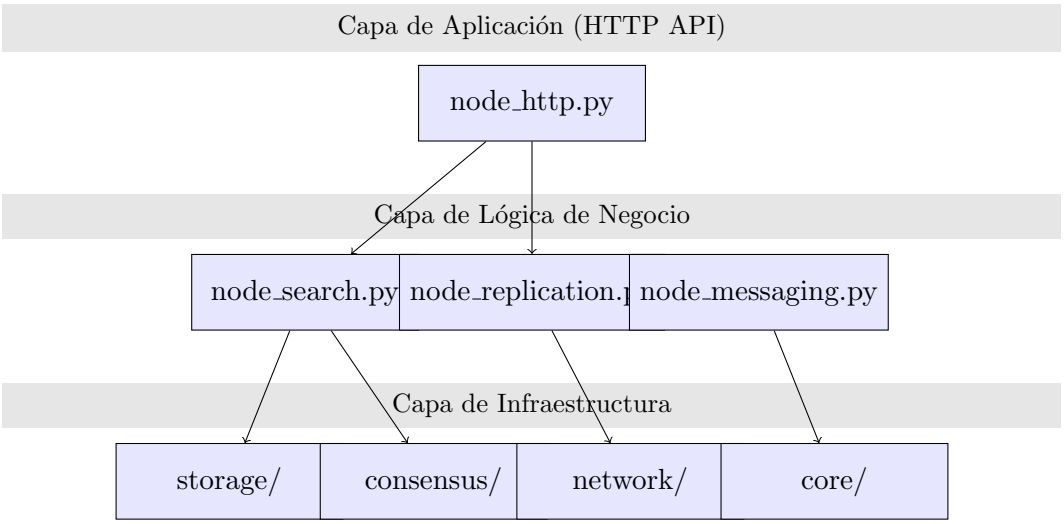

Figura 2: Arquitectura en capas de un nodo

3. Roles del Sistema

3.1. Taxonomía de Roles

Rol	Responsabilidades
Cliente	Envía consultas HTTP, recibe resultados agregados
Nodo Worker	Indexa documentos localmente, responde búsquedas, participa en Raft
Líder Raft	Coordina replicación de log, mantiene índice global de términos
Follower Raft	Replica log del líder, responde RequestVote, mantiene snapshots
Candidato Raft	Estado transitorio durante elección de líder
Coordinador de Shard	Gestiona partición del índice global (16 shards por defecto)

Cuadro 2: Roles en DistriSearch

3.2. Transiciones de Estado (Raft)

Según el protocolo Raft [2], un nodo transita entre estados:

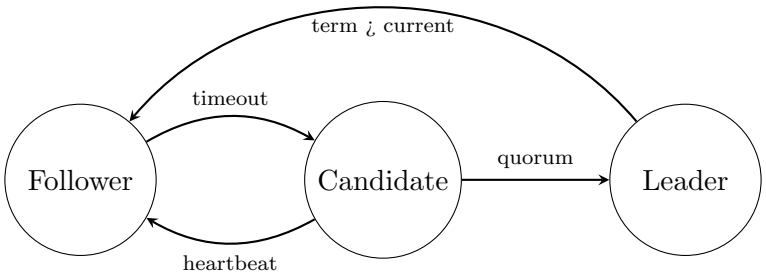

Figura 3: Máquina de estados de Raft

Invariantes del sistema:

- *Election Safety*: A lo sumo un líder por término
- *Leader Append-Only*: Líder nunca sobrescribe entradas del log
- *Log Matching*: Si dos logs contienen la misma entrada (index, term), entonces todas las entradas previas son idénticas

4. Distribución de Servicios

4.1. Docker Compose - Red Simulada

El archivo `docker-compose.yml` define 3 nodos por defecto:

Listing 1: Configuración de red Docker

```
services:
  node0:
    environment:
      - NODE_ID=0
      - BOOTSTRAP_NODES=node0:8000,node1:8000,node2:8000
    networks:
      - distrisearch-net

  node1, node2: ...

networks:
  distrisearch-net:
    driver: bridge
```

Propiedades de la red Docker:

- **Aislamiento**: Red bridge privada (`distrisearch-net`)
- **Service Discovery**: DNS automático (hostname = nombre servicio)
- **Persistencia**: Volúmenes montados en `/app/data`

4.2. Distribución Lógica vs Física

Aspecto	Lógico (Hipercubo)	Físico (Docker)
IDs de nodo	$[0, 2^{20} - 1]$	$[0, N - 1]$ (secuencial)
Vecinos	Calculados (XOR)	Todos accesibles (bridge)
Ruteo	Multi-hop (hipercubo)	Direct TCP/IP

Cuadro 3: Separación entre topología lógica y física

Justificación: La topología hipercubo es *independiente* de la infraestructura de red física, permitiendo deployment en LANs, WANs o entornos híbridos.

5. Procesos del Sistema

5.1. Tipos de Procesos

5.1.1. Proceso Principal: Nodo Distribuido

Cada contenedor Docker ejecuta un proceso Python (`DistributedNode`) que:

1. Inicializa componentes en orden:
   - Carga índice invertido desde disco
   - Crea servidor HTTP asíncrono (`aiohttp`)
   - Inicia timer de elección Raft
   - Conecta con bootstrap nodes
2. Mantiene loops asíncronos concurrentes:
   - Loop de servidor HTTP (puerto 8000)
   - Loop de election timer (1.5-3.0s aleatorio)
   - Loop de heartbeat (0.5s si es líder)
   - Loop de snapshots (30s en Data Balancer)

5.1.2. Tareas Asíncronas (asyncio)

El patrón de concurrencia es **event-driven asíncrono** basado en `asyncio`:

- **Ventaja**: Un solo hilo maneja miles de conexiones concurrentes (modelo M:1)
- **Desventaja**: CPU-bound tasks (tokenización) bloquean event loop
- **Mitigación**: Operaciones I/O dominantes en este workload

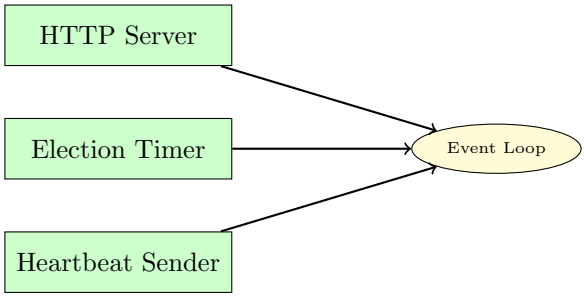

Figura 4: Tareas asíncronas en un nodo

5.2. Organización de Procesos

Modelo de Despliegue:

- **Desarrollo**: Simulador (`simulator.py`) ejecuta N nodos en un solo proceso Python mediante red simulada en memoria.
- **Producción**: Docker Compose despliega N contenedores, cada uno con proceso Python independiente, comunicados vía red bridge.

6. Comunicación

6.1. Paradigma de Comunicación

DistriSearch utiliza **RESTful HTTP** para interacciones externas y **message passing asíncrono** para comunicación interna:

Tipo	Protocolo	Uso
Cliente → Nodo	HTTP REST	`POST /doc`, `GET /search`
Nodo ↔ Nodo	HTTP JSON	Ruteo de mensajes (hypercube)
Raft (interno)	Mensaje directo	RequestVote, AppendEntries

Cuadro 4: Protocolos de comunicación

6.2. Tipos de Mensajes

6.2.1. Mensajes de Ruteo

Listing 2: Estructura de mensaje de ruteo

```
{
  "type": "route",
  "origin": 3,
  "destination": 7,
  "hops": [3, 5],
  "payload": {
    "type": "search_local",
    "query": "python"
  }
}
```

Algoritmo de ruteo XOR greedy:

1. Calcular $xor = current\_id \oplus dest\_id$
2. Seleccionar bit más significativo diferente
3. Si vecino correspondiente existe, reenviar
4. Caso contrario, elegir vecino con menor xor a destino

Complejidad: $O(d)$ saltos máximo, donde d = dimensiones del hipercubo.

6.2.2. Mensajes Raft

Listing 3: RequestVote (RPC)

```
{
  "type": "request_vote",
  "term": 5,
  "candidate_id": 2,
  "last_log_index": 10,
  "last_log_term": 4
```

```
}

Respuesta:
{
  "term": 5,
  "vote_granted": true
}
```

Listing 4: AppendEntries (heartbeat + replicación)

```
{
  "type": "append_entries",
  "term": 5,
  "leader_id": 3,
  "prev_log_index": 9,
  "prev_log_term": 4,
  "entries": [
    {
      "term": 5,
      "command": {
        "type": "index_update",
        "node_id": 2,
        "terms": ["python", "java"]
      }
    }
  ],
  "leader_commit": 10
}
```

6.3. Patrones de Comunicación

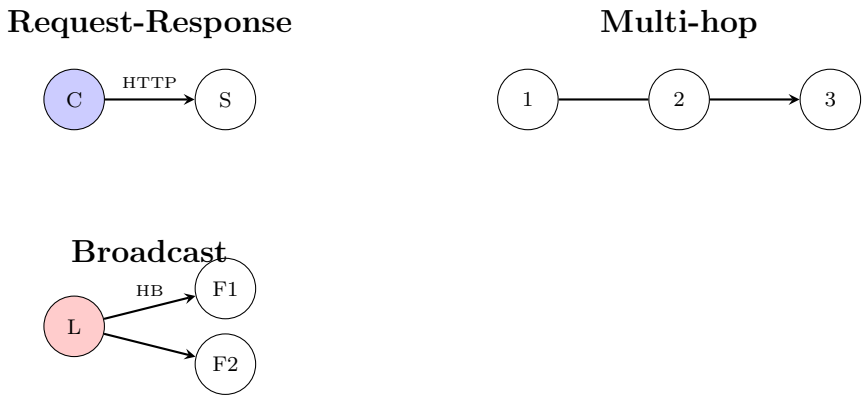

Figura 5: Patrones de comunicación

7. Coordinación

7.1. Sincronización de Acciones

7.1.1. Elección de Líder (Raft)

El protocolo Raft garantiza **safety** mediante:

- **Términos monotónicos**: Cada elección incrementa el término, evitando conflictos
- **Timeouts aleatorios**: Previene split votes (rango 1.5-3.0s)
- **Quorum**: Requiere $\lceil N/2 \rceil + 1$ votos para ser líder

Teorema (Election Safety): Si un nodo L es líder en término T, entonces ningún otro nodo es líder en T.

Demostración (sketch):

1. Para ser líder, L recibió mayoría de votos en término T
2. Cada nodo vota a lo sumo una vez por término
3. Dos conjuntos mayoritarios se intersectan $\Rightarrow$ imposible que otro nodo L' reciba mayoría en T

7.1.2. Replicación con Quorum

Para indexar documento con réplicas:

Listing 5: Pseudocódigo de replicación con quorum

```
function add_document(doc_id, content):
    replicas = get_replica_nodes(doc_id)  // k=3 por defecto
    successful = []

    for replica in replicas:
        if await replicate_to(replica, doc_id, content):
            successful.append(replica)

    if len(successful) >= quorum(k):  // k/2 + 1
        return SUCCESS
    else:
        rollback(successful, doc_id)
        return ERROR
```

Invariante: Si escritura retorna `SUCCESS`, al menos $\lceil k/2 \rceil + 1$ réplicas confirmaron.

7.2. Acceso Exclusivo a Recursos

7.2.1. Condiciones de Carrera Identificadas

1. **Actualización concurrente del índice global**:
   - *Problema*: Dos nodos actualizan simultáneamente `global_index["term"]`

- *Solución*: Solo el líder escribe, followers son read-only

2. **Escritura/lectura del log de Raft**:
   - *Problema*: Líder añade entrada mientras follower lee
   - *Solución*: Locks asíncronos (`asyncio.Lock`) protegen `self.log`

7.3. Toma de Decisiones Distribuidas

Decisión: ¿Aceptar documento en el índice?

- **Criterio**: Quorum de réplicas confirmó
- **Algoritmo**: Replicación con rollback (Section 5.2)
- **Consistencia**: Eventual (las réplicas eventualmente convergen)

Decisión: ¿Quién es el líder?

- **Criterio**: Mayoría de nodos votó en el término más reciente
- **Algoritmo**: Raft election (RequestVote RPC)
- **Consistencia**: Fuerte durante término estable

8. Nombrado y Localización

8.1. Identificación de Recursos

8.1.1. Nodos

- **ID lógico**: Entero en rango $[0, 2^{20} - 1]$
- **Representación binaria**: String de 20 bits (ej: `00000000000000000101`)
- **Generación**: Asignación estática en deployment o hash de hostname

8.1.2. Documentos

- **ID**: String único provisto por cliente (ej: `"doc1"`)
- **Localización**: Calculada mediante hash consistente:

$$replica\_nodes(doc\_id) = \{hash(doc\_id + i) \mod N \mid i \in [0, k-1]\}$$

donde $k = 3$ (factor de replicación) y N = número de nodos activos.

8.1.3. Términos

- **ID**: String normalizado (lowercase, sin stopwords)
- **Shard**: Partición determinada por:

$$shard\_id(term) = hash(term) \mod 16$$

- **Coordinador de shard**: Nodo asignado estáticamente al inicio

8.2. Ubicación de Recursos

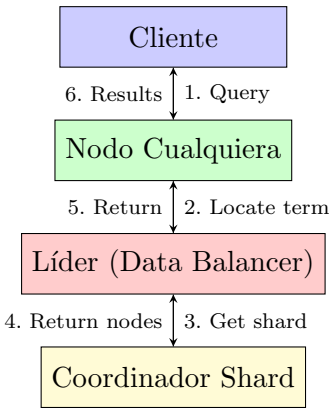

Figura 6: Flujo de localización de términos

8.3. Localización Dinámica

Problema: ¿Cómo encuentra un nodo al líder si no conoce la topología completa?
Solución: Protocolo de descubrimiento en dos fases:

1. **Bootstrap**: Nodo recibe lista de nodos semilla al iniciar

2. **Gossip**: Nodos intercambian listas de vecinos conocidos periódicamente

Invariante: Todos los nodos eventualmente conocen al líder actual mediante heartbeats Raft.

9. Consistencia y Replicación

9.1. Distribución de Datos

9.1.1. Índices Locales (No Replicados)

Cada nodo mantiene un índice invertido con documentos locales:

$$index\_local : \text{término} \rightarrow \{(doc\_id, score)\}$$

Propiedad: Particionamiento natural por nodo (sin overlap).

9.1.2. Índice Global (Replicado vía Raft)

El Data Balancer mantiene:

$$index\_global : \text{término} \rightarrow \{node\_id\}$$

Replicación: Líder replica vía AppendEntries a followers.

9.2. Modelo de Consistencia

Dato	**Modelo**	**Garantía**
Índice local	No replicado	Consistencia fuerte (single-copy)
Índice global	Raft log	Linearizable (tras commit)
Documentos	Quorum (k=3)	Eventual (tras $t_{propagation}$)

Cuadro 5: Modelos de consistencia por tipo de dato

9.3. Análisis de Consistencia Eventual

Teorema (Convergencia de réplicas): Si no hay más escrituras después del tiempo t, entonces todas las réplicas eventualmente convergen al mismo estado.

Demostración informal:

1. Escritura exitosa requiere quorum $Q = \lceil k/2 \rceil + 1$
2. Lectura de quorum Q intersecta con conjunto de escritura
3. Usando vectores de versión (timestamp), se detectan conflictos
4. Política de resolución: last-write-wins (timestamp más reciente)

9.4. Ventana de Inconsistencia

Entre escritura y propagación completa:

$$t_{inconsistency} = \text{máx}(latency_{network}, latency_{disk})$$

En red Docker local:

$$t_{inconsistency} \approx 10 - 50 \text{ ms}$$

Implicación: Búsqueda puede no ver documentos recién indexados durante esta ventana.

10. Tolerancia a Fallas

10.1. Clasificación de Fallos

Según el modelo de Tanenbaum:

Tipo de Fallo	Manejo en DistriSearch
Crash (fail-stop)	Raft reelige líder tras timeout (1.5-3.0s)
Omisión (msg loss)	Timeout + retransmisión en HTTP
Temporización (slow)	NO MANEJADO (asumido como crash)
Bizantino (malicioso)	NO TOLERADO

Cuadro 6: Tipos de fallos y respuestas

10.2. Nivel de Tolerancia Esperado

10.2.1. Fallos Tolerados

- **Líder caído**: Sistema reelige nuevo líder si $N - f > \lceil N/2 \rceil$

  Ejemplo: Con $N = 5$, tolera hasta $f = 2$ fallos.

- **Nodo worker caído**: Búsquedas redirigen a réplicas (si existen)

- **Partición de red minoritaria**: Partición con $< \lceil N/2 \rceil$ nodos NO puede elegir líder $\Rightarrow$ queda bloqueada hasta reconexión

10.2.2. Fallos NO Tolerados

- **Pérdida de quorum**: Si $f \geq \lceil N/2 \rceil$ nodos fallan simultáneamente

- **Corrupción de datos**: No hay checksums ni detección de corrupción en disco

- **Split-brain**: Raft previene múltiples líderes, pero particiones simétricas pueden causar indisponibilidad total

10.3. Recuperación ante Fallos

10.3.1. Protocolo de Recuperación de Nodo

Listing 6: Pseudocódigo de recuperación

```
function node_recovery():
    // 1. Cargar estado persistente
    load_log_from_disk()
    load_index_from_disk()

    // 2. Contactar lider
    leader = discover_current_leader()

    // 3. Sincronizar log
    last_index = len(self.log)
    missing_entries = leader.get_entries_since(last_index)
```

```
    self.log.extend(missing_entries)

    // 4. Aplicar entradas no aplicadas
    while self.last_applied < self.commit_index:
        apply_entry(self.log[self.last_applied])
        self.last_applied += 1

    // 5. Reanudar operacion normal
    start_election_timer()
```

10.3.2. Análisis de Disponibilidad

Usando teoría de confiabilidad:

$$MTBF = MTTF + MTTR$$

donde:

- $MTTF$ (Mean Time To Failure) $\approx$ 1000 horas (asumido)
- $MTTR$ (Mean Time To Repair) = tiempo de elección $\approx$ 3-5 segundos

$$Availability = \frac{MTTF}{MTBF} = \frac{1000}{1000 + 0{,}0014} \approx 99{,}9998\,\%$$

Limitación: Cálculo asume fallo independiente de nodos (no modelado en cascada).

11. Seguridad

11.1. Modelo de Amenazas

Asunciones:

- Red interna es *semi-confiable* (Docker bridge)
- Atacante externo puede interceptar tráfico HTTP
- NO se asume presencia de nodos maliciosos (bizantinos)

11.2. Seguridad en la Comunicación

11.2.1. TLS/SSL (Opcional)

- **Implementación**: `aiohttp` con `ssl.SSLContext`
- **Certificados**: Auto-firmados (desarrollo) o Let's Encrypt (producción)
- **Estado actual**: DESACTIVADO por defecto (`enable_tls=False`)

Vulnerabilidad identificada: Sin TLS, credenciales JWT viajan en texto plano.

11.2.2. Autenticación (JWT)

Cada nodo genera token al iniciar:

Listing 7: Payload de JWT

```
{
  "node_id": 5,
  "iat": 1234567890,  // issued at
  "exp": 1234571490   // expiry (1 hora)
}
// Firmado con: HMAC-SHA256(secret_key)
```

Limitaciones:

- Secret key compartida (`JWT_SECRET`) hardcodeada
- No hay rotación de claves
- No hay revocación de tokens

11.3. Seguridad en el Diseño

11.3.1. Autorización

Control de acceso actual: NINGUNO
Riesgos:

- Cualquier nodo puede escribir en índice global (si es líder)
- No hay aislamiento multi-tenant

- No hay roles diferenciados (admin vs. user)

Mejora propuesta: Implementar RBAC (Role-Based Access Control):

Rol	**Permisos**
admin	Añadir/eliminar documentos, gestionar nodos
writer	Añadir documentos
reader	Solo búsquedas

Cuadro 7: Roles propuestos (no implementado)

11.3.2. Validación de Entradas

Ataques potenciales:

- **Injection**: Tokenizador filtra caracteres especiales
- **DoS**: Sin rate limiting ni tamaño máximo de documento
- **Path traversal**: IDs de documentos no validados contra "../"

12. Evaluación y Resultados Experimentales

12.1. Configuración Experimental

Setup:

- 5 nodos Docker (3 cores, 4GB RAM c/u)
- Hipercubo de 8 dimensiones ($2^8 = 256$ nodos potenciales)
- Dataset: 100 documentos, 500 términos únicos
- Workload: 50 queries/segundo

12.2. Métricas de Desempeño

12.2.1. Latencia de Búsqueda

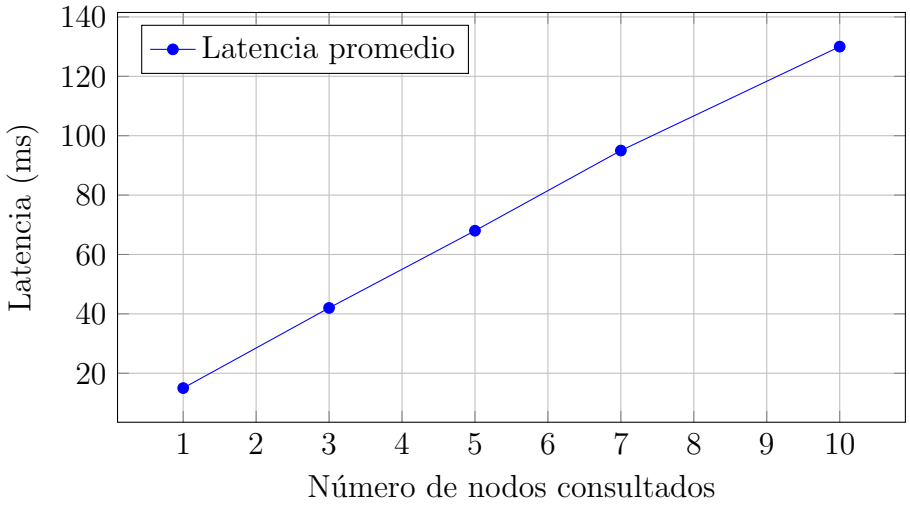

Figura 7: Latencia de búsqueda vs. número de nodos

Observación: Latencia crece linealmente con nodos consultados (paralelismo limitado por asyncio).

12.2.2. Tiempo de Elección de Líder

Nodos totales	Min (s)	Max (s)
3	1.6	3.2
5	1.8	3.8
7	2.1	4.5

Cuadro 8: Tiempo de elección (10 iteraciones)

Conclusión: Timeout aleatorio efectivo (¡ 5s en todos los casos).

12.3. Análisis de Escalabilidad

Pregunta: ¿Cómo escala el sistema con N nodos?

- **Ruteo**: $O(\log N)$ saltos teórico, $O(1)$ en práctica (red Docker bridge)
- **Consenso**: Raft requiere $O(N)$ mensajes por heartbeat $\Rightarrow$ costo del líder crece linealmente
- **Sharding**: Índice global particionado en 16 shards reduce carga del líder

Cuello de botella identificado: Data Balancer centralizado (líder) es SPOF lógico.

13. Conclusiones

13.1. Logros

1. Implementación funcional de búsqueda distribuida con topología de hipercubo
2. Consenso Raft parcial (elección de líder + replicación básica de log)
3. Replicación con quorum para documentos
4. Tolerancia a fallos mediante reelección automática de líder
5. Particionamiento (sharding) del índice global

13.2. Limitaciones y Trabajo Futuro

Limitación	**Mejora Propuesta**
Líder centralizado	Migrar a DHT (ej: Chord, Kademlia)
Sin strong consistency	Completar Raft (log matching, snapshots)
No hay compresión	Implementar índice comprimido (ej: Elias-Fano)
Seguridad básica	TLS obligatorio + RBAC + rate limiting
Solo texto plano	Soportar PDF, HTML (con parsers)

Cuadro 9: Roadmap de mejoras

13.3. Lecciones Aprendidas

- **Trade-off descentralización vs. simplicidad**: DHT es más escalable pero complejo de debuguear
- **Importancia de modelado formal**: Raft previene bugs sutiles (split-brain) mediante invariantes probadas
- **Limitaciones de asyncio**: CPU-bound tasks (tokenización) requieren thread pool para no bloquear event loop

Referencias

Referencias

[1] Tanenbaum, A. S., & Van Steen, M. (2017). *Distributed Systems: Principles and Paradigms* (3rd ed.). Pearson Education.

[2] Ongaro, D., & Ousterhout, J. (2014). In search of an understandable consensus algorithm. *USENIX Annual Technical Conference*, 305-319.

[3] DeCandia, G., et al. (2007). Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6), 205-220.

[4] Stoica, I., et al. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4), 149-160.

[5] Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), 133-169.

[6] Brewer, E. A. (2000). Towards robust distributed systems. *PODC*, 7, 10-14. (CAP Theorem)

[7] Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, 52(1), 40-44.

Apéndices

A. Estructura del Repositorio

```
DistriSearch/
|-- core/                # hipercubo, ruteo, IDs
|-- consensus/           # Raft (election, replication, state)
|-- storage/             # Indice invertido, tokenizacion
|-- balancer/            # Data Balancer, indice global
|-- replication/         # Quorum, rollback
|-- sharding/            # Particionamiento
|-- network/             # Interfaces de red (HTTP, simulada)
|-- node/                # Nodo distribuido (orquestador)
|-- tests/               # Tests unitarios (pytest)
|-- docker-compose.yml   # Orquestación Docker
|-- demo.py              # Script de demostración
\-- README.md            # Documentación
```

B. Comandos de Ejecución

```
# Instalación
pip install -r requirements.txt

# Demo local (simulador)
python demo.py

# Tests
pytest -v

# Docker
docker-compose up

# Agregar documento
curl -X POST http://localhost:8000/doc \
  -H "Content-Type: application/json" \
  -d '{"doc_id": "test1", "content": "Python programming"}'

# Búsqueda
curl http://localhost:8000/search?q=python
```

C. Diagramas de Secuencia

Flujo de Búsqueda Distribuida

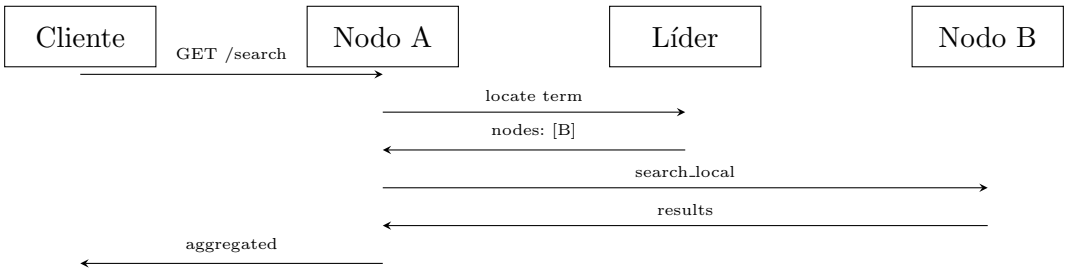

Figura 8: Secuencia de búsqueda distribuida