DistriSearch

Sistema de Búsqueda de Archivos Distribuido

Informe Técnico del Proyecto
Sistemas Distribuidos

Autores:

Nombre Apellido 1
Nombre Apellido 2
Nombre Apellido 3

Universidad de La Habana
Facultad de Matemática y Computación

Noviembre 2025

Índice

1. Resumen Ejecutivo

DistriSearch es un sistema distribuido de búsqueda e indexación de archivos que permite a múltiples nodos compartir y buscar archivos de manera eficiente. El sistema está diseñado para escalar desde pequeños clusters de 3-4 nodos hasta deployments empresariales de más de 100 nodos.

1.1. Objetivos del Sistema

- Proporcionar búsqueda full-text distribuida sobre archivos compartidos
- Garantizar alta disponibilidad mediante replicación de datos
- Soportar la adición y eliminación dinámica de nodos
- Mantener consistencia eventual de los datos
- Proveer una interfaz web intuitiva para usuarios

1.2. Tecnologías Principales

- **Backend**: Python (FastAPI) con arquitectura async/await
- **Frontend**: Streamlit para interfaz de usuario
- **Base de Datos**: MongoDB con GridFS para archivos grandes
- **Mensajería**: Redis para pub/sub distribuido
- **Contenedores**: Docker y Docker Compose
- **Monitoreo**: Prometheus y Grafana (opcional)

2. Arquitectura del Sistema

2.1. Problema de Diseño

El diseño del sistema enfrenta varios desafíos fundamentales:

1. **Escalabilidad**: ¿Cómo soportar desde 3 hasta 100+ nodos sin degradación significativa?
2. **Disponibilidad**: ¿Cómo mantener el servicio operativo ante fallas de nodos?
3. **Consistencia**: ¿Cómo garantizar que las búsquedas retornen resultados actualizados?
4. **Particionamiento**: ¿Cómo distribuir los datos entre nodos de manera eficiente?

2.2. Organización del Sistema Distribuido

El sistema sigue una arquitectura **híbrida** que combina elementos de:

- **Cliente-Servidor**: Los usuarios interactúan con el frontend que actúa como cliente del backend.
- **Peer-to-Peer**: Los nodos agentes se comunican entre sí para replicación y coordinación.
- **Coordinador Dinámico**: Un nodo es elegido líder mediante el protocolo Raft.

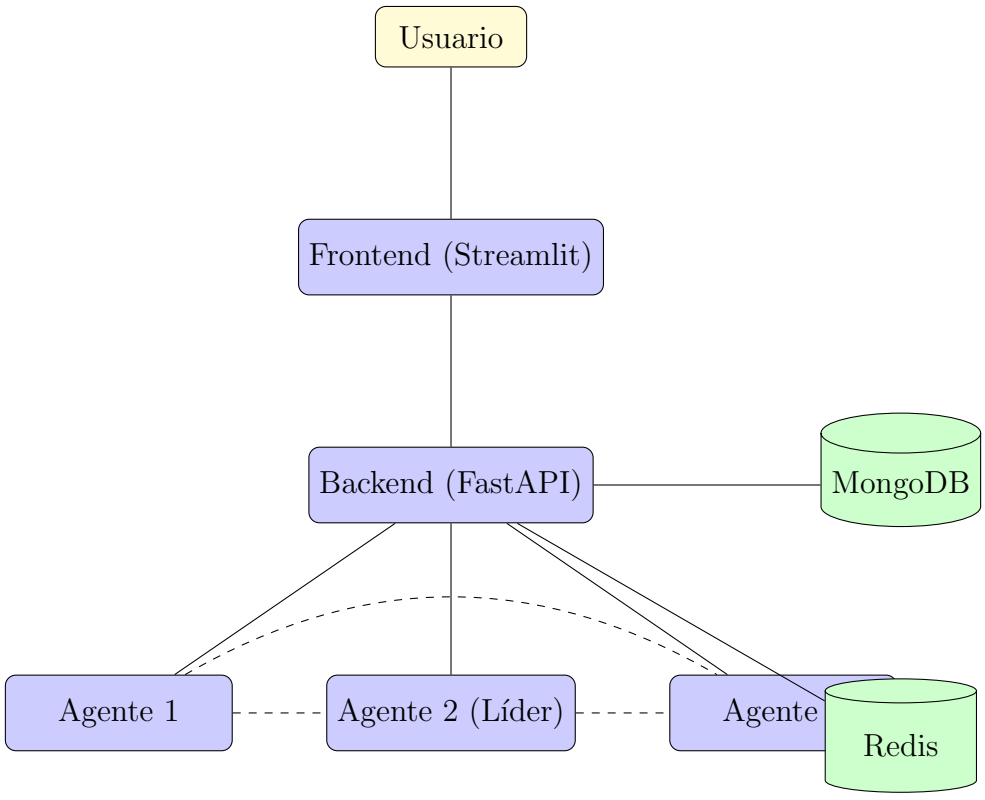

Figura 1: Arquitectura general del sistema DistriSearch

2.3. Roles del Sistema

El sistema define los siguientes roles para los nodos:

Rol	Responsabilidades	Instancias
`COORDINATOR`	Elección de líder, coordinación de transacciones, gestión de membresía del cluster	1 (elegido)
`WORKER`	Almacenamiento de archivos, indexación local, respuesta a búsquedas, replicación	N
`GATEWAY`	Punto de entrada para clientes, balanceo de carga, routing de peticiones	1+
`HYBRID`	Combina funciones de WORKER y GATEWAY (modo por defecto)	N

Cuadro 1: Roles de nodos en el sistema

2.4. Distribución de Servicios en Redes Docker

El sistema utiliza una arquitectura de **doble red** para seguridad y aislamiento:

2.4.1. Red Interna (`internal_network`)

Red privada para comunicación entre servicios del backend:

- Backend API
- MongoDB
- Redis
- Agentes
- Prometheus (monitoreo)

Características:

- No expuesta al exterior
- Comunicación segura entre contenedores
- DNS interno de Docker para descubrimiento

2.4.2. Red de Cliente (`client_network`)

Red para comunicación con usuarios finales:

- Frontend (Streamlit)
- Backend API (puerto expuesto)
- Grafana (monitoreo, opcional)

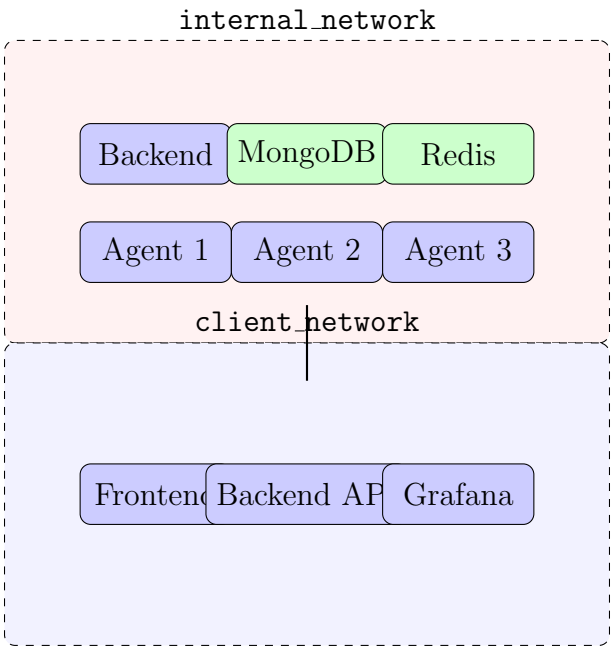

Figura 2: Distribución de servicios en redes Docker

3. Procesos del Sistema

3.1. Problema de Procesos

La cuestión fundamental es: *¿Cuántos programas o servicios necesita el sistema y cómo se organizan?*

3.2. Tipos de Procesos

El sistema cuenta con los siguientes tipos de procesos:

Proceso	**Tipo**	**Descripción**
Backend API	Servidor HTTP	Procesa peticiones REST, coordina con agentes
Frontend	Servidor Web	Interfaz de usuario Streamlit
Agente	Daemon	Escanea carpetas, indexa archivos, sirve descargas
MongoDB	Base de Datos	Almacenamiento persistente
Redis	Message Broker	Pub/Sub para eventos distribuidos
Scanner (dentro del Agente)	Worker	Escaneo periódico de archivos
Message Bus Worker	Background Task	Procesa cola de eventos

Cuadro 2: Tipos de procesos en el sistema

3.3. Organización de Procesos por Instancia

3.3.1. Instancia Backend (1 contenedor)

- Proceso principal: Uvicorn (servidor ASGI)
- Workers internos (asyncio tasks):
  - Message Bus processor
  - Cluster manager heartbeat
  - Anti-entropy loop
  - Checkpoint scheduler

3.3.2. Instancia Agente (N contenedores)

- Proceso principal: FastAPI server
- Workers internos:
  - File scanner (periódico)
  - Uploader service
  - Raft election loop
  - Replication worker

3.4. Patrón de Diseño para Desempeño

El sistema utiliza un patrón **async/await con event loop** basado en:

1. **Asyncio (Python)**:
   - Event loop único por proceso
   - Coroutines para I/O no bloqueante
   - Tasks para operaciones concurrentes
2. **FastAPI con Uvicorn**:
   - Servidor ASGI asíncrono
   - Múltiples workers opcionales
   - Manejo eficiente de conexiones HTTP
3. **Hybrid Threading para CPU-bound**:
   - ThreadPoolExecutor para operaciones de I/O de archivos
   - ProcessPoolExecutor para indexación pesada (opcional)

```python
async def search_files(query: str) -> List[FileResult]:
    # Crear tareas para buscar en paralelo
    tasks = [
        search_in_node(node, query)
        for node in cluster_nodes
    ]

    # Ejecutar en paralelo y agregar resultados
    results = await asyncio.gather(*tasks, return_exceptions=True
        )

    return merge_results(results)
```

Listing 1: Ejemplo de patrón async/await

4. Comunicación

4.1. Problema de Comunicación

El desafío principal es: *¿Cómo enviar información de manera eficiente y confiable entre los componentes del sistema?*

4.2. Tipos de Comunicación Utilizados

Tipo	Uso	Justificación
REST/HTTP	Cliente-Servidor	Simplicidad, compatibilidad, stateless
Redis Pub/Sub	Eventos distribuidos	Desacoplamiento, broadcast eficiente
RPC (JSON-RPC)	Servidor-Servidor	Llamadas síncronas entre nodos
HTTP Streaming	Descarga de archivos	Transferencia eficiente de archivos grandes

Cuadro 3: Tipos de comunicación en el sistema

4.3. Comunicación Cliente-Servidor

4.3.1. Frontend ↔ Backend

- **Protocolo**: HTTP/HTTPS REST
- **Formato**: JSON
- **Autenticación**: JWT Bearer tokens
- **Endpoints principales**:
  - `POST /api/auth/login` - Autenticación
  - `GET /api/search?q=...` - Búsqueda de archivos
  - `GET /api/files/{file_id}/download` - Descarga
  - `GET /api/nodes` - Estado del cluster

4.4. Comunicación Servidor-Servidor

4.4.1. Backend ↔ Agentes

- **Protocolo**: HTTP REST + Redis Pub/Sub
- **Operaciones REST**:
  - Registro de nodo
  - Indexación de archivos
  - Solicitud de descarga

- Health checks

- **Eventos Pub/Sub**:
  - `node_joined` / `node_left`
  - `file_indexed` / `file_deleted`
  - `leader_elected`
  - `replication_required`

4.4.2. Agente ↔ Agente

Comunicación P2P para coordinación:

- **Raft Consensus**:
  - `RequestVote` RPC
  - `AppendEntries` RPC (heartbeats + log replication)
- **Replicación**:
  - `POST /api/replication/write`
  - `GET /api/replication/read/{resource_id}`
  - `GET /api/replication/merkle/root`

4.5. Comunicación entre Procesos

Dentro de un mismo contenedor, los procesos se comunican mediante:

- **Message Bus interno**: Cola de eventos con prioridades
- **Shared memory**: Para métricas y estado
- **asyncio.Queue**: Para tareas asíncronas

```python
# Publicar evento
await message_bus.publish(Event(
    event_type=EventType.FILE_INDEXED,
    data={"file_id": file_id, "node_id": node_id},
    priority=EventPriority.NORMAL
))

# Suscribirse a eventos
@message_bus.subscribe(EventType.FILE_INDEXED)
async def on_file_indexed(event: Event):
    await update_search_index(event.data)
```

Listing 2: Ejemplo de Message Bus

5. Coordinación

5.1. Problema de Coordinación

El desafío es: *¿Cómo poner de acuerdo a todos los servicios para actuar de manera coherente?*

5.2. Sincronización de Acciones

5.2.1. Relojes Lógicos

El sistema implementa dos tipos de relojes lógicos:

1. **Reloj de Lamport**: Para ordenamiento causal simple
   - Cada evento local incrementa el contador
   - Al recibir mensaje: $C_{local} = max(C_{local}, C_{recibido}) + 1$
2. **Vector Clocks**: Para detectar concurrencia
   - Cada nodo mantiene contador por cada nodo conocido
   - Permite detectar eventos concurrentes vs. causalmente relacionados

5.2.2. Elección de Líder (Raft)

- **Estados**: Follower, Candidate, Leader
- **Términos**: Número monotónicamente creciente para detectar líderes obsoletos
- **Timeout aleatorio**: 150-300ms para evitar split votes
- **Log replication**: Entradas confirmadas cuando mayoría replica

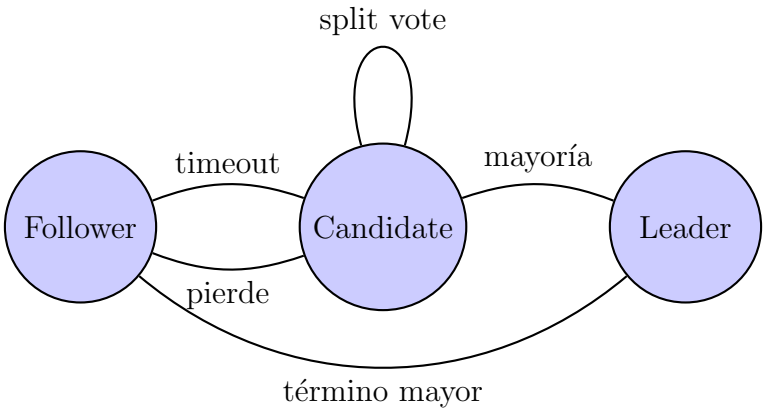

Figura 3: Transiciones de estado en Raft

5.3. Acceso Exclusivo a Recursos

5.3.1. Distributed Mutex (Ricart-Agrawala)

Para operaciones que requieren exclusión mutua:

1. Nodo solicita acceso enviando timestamp a todos
2. Otros nodos responden si no están solicitando o tienen timestamp mayor
3. Se otorga acceso cuando se reciben respuestas de mayoría
4. Al liberar, se envían respuestas diferidas

5.3.2. Condiciones de Carrera Manejadas

- **Escrituras concurrentes**: Resueltas con Vector Clocks + Last-Write-Wins
- **Registro de nodos**: Protegido por lock distribuido
- **Elección de líder**: Términos de Raft previenen múltiples líderes

5.4. Toma de Decisiones Distribuidas

5.4.1. Two-Phase Commit (2PC)

Para transacciones que afectan múltiples nodos:

1. **Fase Prepare**: Coordinador pregunta a participantes si pueden commit
2. **Fase Commit/Abort**: Coordinador decide según votos

5.4.2. Quorum-based Decisions

Para operaciones de lectura/escritura:

- $W + R > N$ garantiza consistencia fuerte
- Quorum de escritura: $W = \lfloor N/2 \rfloor + 1$
- Quorum de lectura: $R = \lfloor N/2 \rfloor + 1$

5.5. Fencing para Split-Brain

Mecanismo para prevenir situaciones donde múltiples nodos creen ser líder:

- **Fencing Token**: Número monotónico asignado al convertirse en líder
- **Validación**: Operaciones rechazadas si token es obsoleto
- **Detección**: Verificación periódica de líderes múltiples
- **Resolución**: Líder con token mayor gana

6. Nombrado y Localización

6.1. Problema de Nombrado

El desafío es: *¿Dónde se encuentra un recurso y cómo llegar a él?*

6.2. Identificación de Datos y Servicios

6.2.1. Identificadores de Archivos

Cada archivo tiene un identificador único compuesto por:

```
file_id = sha256(node_id + path + content_hash)
```

- **Unicidad**: Garantizada por hash criptográfico
- **Determinístico**: El mismo archivo genera el mismo ID
- **Verificable**: Permite detectar modificaciones

6.2.2. Identificadores de Nodos

```
node_id = f"{hostname}-{uuid4()}"
```

- Único globalmente
- Persistente entre reinicios (almacenado en config)
- Legible para debugging

6.2.3. Identificadores de Servicios

```
service_id = f"{service_type}:{node_id}:{port}"
# Ejemplo: "search:node-alpha-1234:8001"
```

6.3. Ubicación de Datos y Servicios

6.3.1. Service Registry

Registro centralizado (con respaldo distribuido) de servicios:

Service ID	Host	Port	Status
search:node-1:8001	172.18.0.5	8001	HEALTHY
search:node-2:8001	172.18.0.6	8001	HEALTHY
upload:node-1:8002	172.18.0.5	8002	DEGRADED

Cuadro 4: Ejemplo de Service Registry

6.3.2. Consistent Hashing

Para determinar qué nodo almacena qué datos:

- Anillo virtual de 2^{32} posiciones
- Nodos colocados según hash de su ID
- Datos asignados al siguiente nodo en el anillo
- Nodos virtuales para balanceo uniforme

6.4. Localización de Datos y Servicios

6.4.1. Descubrimiento de Servicios

1. **DNS interno de Docker**: Resolución por nombre de contenedor
2. **Service Registry**: Consulta de servicios disponibles
3. **Health Checks**: Verificación periódica de disponibilidad

6.4.2. Localización de Archivos

Para encontrar un archivo específico:

1. Cliente envía búsqueda al backend
2. Backend consulta índice en MongoDB
3. Índice retorna `(file_id, node_id)` tuples
4. Backend consulta Service Registry para localizar nodo
5. Se establece conexión directa para descarga

7. Consistencia y Replicación

7.1. Problema de Consistencia

El desafío es: *¿Cómo garantizar que las copias de un dato sean coherentes?*

7.2. Distribución de Datos

7.2.1. Particionamiento

Los datos se distribuyen usando **Consistent Hashing**:

- Cada archivo pertenece a un nodo primario
- Réplicas se colocan en nodos siguientes del anillo
- Re-balanceo automático al agregar/eliminar nodos

7.2.2. Tipos de Datos

Tipo	Almacenamiento	Replicación
Metadatos de archivos	MongoDB	Síncrona
Contenido de archivos	GridFS + Nodos	Eventual
Índice de búsqueda	MongoDB	Síncrona
Estado del cluster	Redis + Memory	Eventual
Log de Raft	MongoDB	Por consenso

Cuadro 5: Distribución de tipos de datos

7.3. Replicación

7.3.1. Factor de Replicación

- Configurable (por defecto: 3)
- Mínimo: $\lfloor N/2 \rfloor + 1$ para quorum
- Máximo: N (todos los nodos)

7.3.2. Estrategias de Replicación

1. **Síncrona (STRONG)**:
   - Espera confirmación de todas las réplicas
   - Mayor latencia, máxima consistencia
   - Usada para: metadatos críticos
2. **Quorum (QUORUM)**:
   - Espera confirmación de mayoría

- Balance entre latencia y consistencia
- Usada por defecto

3. **Asíncrona (EVENTUAL)**:
   - Confirma inmediatamente, replica en background
   - Mínima latencia, consistencia eventual
   - Usada para: contenido de archivos

7.4. Confiabilidad de Réplicas

7.4.1. Resolución de Conflictos

Cuando versiones divergen:

1. **Last-Write-Wins (LWW)**: Timestamp más reciente gana
2. **Vector Clocks**: Detecta si son concurrentes o causales
3. **Merge**: Para datos que soportan fusión (ej: CRDTs)

7.4.2. Anti-Entropy con Merkle Trees

Proceso periódico para reparar inconsistencias:

1. Cada nodo calcula hash de sus datos (Merkle root)
2. Compara roots con otros nodos
3. Si difieren, compara branches para identificar diferencias
4. Solo intercambia datos que difieren

```
async def anti_entropy(peer_node):
    my_root = get_merkle_root()
    peer_root = await peer.get_merkle_root()

    if my_root == peer_root:
        return  # Sincronizados

    # Encontrar diferencias
    diff_resources = await compare_merkle_branches(peer)

    for resource_id in diff_resources:
        await sync_resource(resource_id, peer)
```

Listing 3: Algoritmo de anti-entropy simplificado

8. Tolerancia a Fallas

8.1. Problema de Tolerancia

El desafío es: *¿Cómo mantener el sistema operativo cuando componentes fallan?*

8.2. Respuesta a Errores

8.2.1. Tipos de Fallas Manejadas

Tipo	Descripción	Respuesta
CRASH	Nodo deja de funcionar	Failover + Replicación
OMISSION	Mensajes perdidos	Retry + Timeout
TIMING	Respuestas lentas	Timeout + Circuit Breaker
BYZANTINE	Comportamiento arbitrario	No soportado

Cuadro 6: Tipos de fallas y respuestas

8.2.2. Circuit Breaker Pattern

Protege contra fallos en cascada:

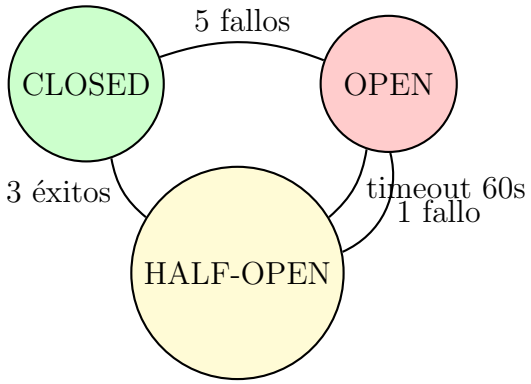

Figura 4: Estados del Circuit Breaker

8.3. Nivel de Tolerancia Esperado

- **Tolerancia a f fallas**: El sistema tolera $f = \lfloor (N-1)/2 \rfloor$ fallas simultáneas
- **Con N=5 nodos**: Tolera 2 fallas simultáneas
- **Con N=7 nodos**: Tolera 3 fallas simultáneas

8.3.1. Métricas de Confiabilidad

- **MTTF** (Mean Time To Failure): Tiempo promedio hasta falla
- **MTTR** (Mean Time To Repair): Tiempo promedio de recuperación

- **MTBF** (Mean Time Between Failures): $MTBF = MTTF + MTTR$
- **Availability**: $A = \frac{MTTF}{MTTF+MTTR}$

8.4. Fallos Parciales

8.4.1. Nodos Caídos Temporalmente

1. **Detección**: Heartbeats cada 5 segundos, timeout de 15 segundos
2. **Sospecha**: Nodo marcado como SUSPECT
3. **Confirmación**: Phi Accrual Failure Detector ($\phi > 8$)
4. **Failover**: Servicios migrados a otros nodos
5. **Recuperación**: Nodo puede re-unirse y sincronizar

8.4.2. Nodos Nuevos

1. Nuevo nodo contacta a nodo conocido (seed)
2. Recibe lista de miembros del cluster
3. Registra sus servicios en el Service Registry
4. Recibe datos asignados por Consistent Hashing
5. Comienza a participar en replicación

8.5. Checkpointing

8.5.1. Checkpoints Locales

- Guardado periódico del estado (cada 5 minutos)
- Almacenamiento en disco local + MongoDB
- Limpieza automática (máximo 5 checkpoints)

8.5.2. Checkpoints Distribuidos

Protocolo Chandy-Lamport simplificado:

1. Iniciador crea checkpoint local y notifica a todos
2. Cada nodo crea su checkpoint y responde
3. Checkpoint global completo cuando mayoría responde
4. Permite recuperación consistente del sistema completo

9. Seguridad

9.1. Problema de Seguridad

El desafío es: *¿Qué tan vulnerable es el diseño y cómo protegemos el sistema?*

9.2. Seguridad en la Comunicación

9.2.1. HTTPS/TLS

- Certificados SSL/TLS para todas las comunicaciones externas
- Generación automática de certificados self-signed para desarrollo
- Soporte para certificados de CA para producción

9.2.2. Comunicación Interna

- Red Docker aislada (`internal_network`)
- Tokens JWT para autenticación entre servicios
- Validación de origen para peticiones inter-nodo

9.3. Seguridad en el Diseño

9.3.1. Principios Aplicados

1. **Defensa en profundidad**: Múltiples capas de seguridad
2. **Mínimo privilegio**: Cada componente tiene permisos mínimos necesarios
3. **Fail-secure**: Ante errores, denegar acceso por defecto
4. **Separación de redes**: Cliente vs. interna

9.3.2. Vulnerabilidades Mitigadas

Amenaza	**Mitigación**
SQL/NoSQL Injection	Uso de ODM (Motor) con queries parametrizadas
XSS	Frontend Streamlit no renderiza HTML arbitrario
CSRF	Tokens JWT en headers, no cookies
DDoS	Rate limiting por IP y usuario
Man-in-the-Middle	TLS obligatorio en producción

Cuadro 7: Amenazas y mitigaciones

9.4. Autenticación y Autorización

9.4.1. Autenticación de Usuarios

1. Usuario envía credenciales (username + password)
2. Backend verifica contra hash bcrypt en MongoDB
3. Si válido, genera token JWT con claims:

```
{
    "sub": "user_id",
    "username": "john_doe",
    "roles": ["user", "admin"],
    "exp": 1732900800,  # 24h
    "iat": 1732814400
}
```

4. Token enviado en header: `Authorization: Bearer <token>`

9.4.2. Autenticación de Nodos

- Cada nodo tiene un `node_id` único
- Tokens de nodo para comunicación inter-nodo
- Validación de IP + node_id para registro

9.4.3. Sistema de Roles (RBAC)

Rol	Permisos	Descripción
`guest`	`read:files`	Usuario no autenticado
`user`	`read:*, write:files`	Usuario estándar
`admin`	`*:*`	Administrador
`node`	`internal:*`	Nodo del cluster

Cuadro 8: Roles y permisos

9.4.4. Rate Limiting

Protección contra abuso:

- **Por IP**: 100 requests/minuto
- **Por usuario**: 200 requests/minuto
- **Por endpoint crítico**: 10 requests/minuto (ej: login)
- Implementación con algoritmo Token Bucket

9.5. Auditoría

- Registro de todas las operaciones sensibles
- Almacenamiento en MongoDB (colección `audit_logs`)
- Campos: timestamp, user_id, action, resource, ip_address, result
- Retención: 90 días por defecto

10. Conclusiones

DistriSearch implementa una arquitectura distribuida robusta que aborda los principales desafíos de los sistemas distribuidos:

1. **Arquitectura**: Diseño híbrido cliente-servidor con coordinación P2P mediante Raft, soportando escalabilidad de 3 a 100+ nodos.
2. **Procesos**: Modelo async/await eficiente con FastAPI/Uvicorn, minimizando overhead de threads mientras mantiene alta concurrencia.
3. **Comunicación**: Combinación de REST para operaciones síncronas y Redis Pub/-Sub para eventos asíncronos, con RPC para coordinación entre nodos.
4. **Coordinación**: Implementación de Raft para consenso, relojes lógicos para ordenamiento, y mutex distribuido para exclusión mutua.
5. **Nombrado**: Sistema de identificadores únicos con Consistent Hashing para distribución y Service Registry para localización.
6. **Consistencia**: Modelo de consistencia configurable (eventual, quorum, fuerte) con resolución de conflictos mediante Vector Clocks y anti-entropy con Merkle Trees.
7. **Tolerancia a fallas**: Circuit breakers, failover automático, checkpointing distribuido, y fencing tokens para prevenir split-brain.
8. **Seguridad**: TLS para comunicaciones, JWT para autenticación, RBAC para autorización, y rate limiting para protección contra abuso.

10.1. Trabajo Futuro

- Implementar búsqueda semántica con embeddings
- Añadir soporte para Byzantine Fault Tolerance
- Optimizar anti-entropy con vectores de versión
- Implementar sharding automático basado en carga
- Añadir observabilidad completa con OpenTelemetry

A. Configuración de Docker Compose

```yaml
version: "3.8"

networks:
  internal_network:
    driver: bridge
    internal: true
  client_network:
    driver: bridge

services:
  backend:
    build: ./backend
    networks:
      - internal_network
      - client_network
    depends_on:
      - mongodb
      - redis

  frontend:
    build: ./frontend
    networks:
      - client_network
    ports:
      - "8501:8501"

  mongodb:
    image: mongo:6
    networks:
      - internal_network

  redis:
    image: redis:7-alpine
    networks:
      - internal_network
```

Listing 4: docker-compose.yml (extracto)

B. Estructura del Proyecto

```
DistriSearch/
    backend/
        core/           # Modulos del sistema distribuido
            config.py
            message_bus.py
            cluster_manager.py
            coordination.py
            replication.py
```

```
            fault_tolerance.py
            security.py
        routes/          # Endpoints API
        services/        # Logica de negocio
        main.py
    frontend/
        app.py
        pages/
        components/
    agent/
        agent.py
        scanner.py
        uploader.py
    deploy/
        docker-compose.yml
```

Referencias

[1] Diego Ongaro and John Ousterhout. *In Search of an Understandable Consensus Algorithm (Extended Version)*. Stanford University, 2014.

[2] Leslie Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*. Communications of the ACM, 1978.

[3] Giuseppe DeCandia et al. *Dynamo: Amazon's Highly Available Key-value Store*. SOSP '07, 2007.

[4] Ralph Merkle. *A Digital Signature Based on a Conventional Encryption Function*. CRYPTO '87, 1987.

[5] K. Mani Chandy and Leslie Lamport. *Distributed Snapshots: Determining Global States of Distributed Systems*. ACM TOCS, 1985.

[6] Michael T. Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2018.