Informe de Diseño: Sistema de Búsqueda Distribuido

1. Arquitectura

Diseño del Sistema

El sistema sigue una arquitectura **P2P (Peer-to-Peer) Estructurada con Coordinador Dinámico**. Aunque todos los nodos tienen la capacidad de comportarse como pares (almacenando y procesando datos), el sistema elige dinámicamente un líder (Coordinador) para tareas de gestión del clúster.

Organización

El sistema se organiza como un anillo lógico utilizando **Consistent Hashing**. Esto permite una distribución uniforme de la carga y de los datos entre los nodos disponibles sin necesidad de una tabla centralizada estática.

Roles del Sistema

1. **Peer Node (Nodo Par):**
   - Almacena fragmentos del índice invertido y archivos.
   - Responde a consultas de búsqueda locales.
   - Participa en la replicación de datos.
2. **Coordinador (Líder):**
   - Elegido dinámicamente entre los pares.
   - Monitorea la salud del clúster (Heartbeats).
   - Gestiona la entrada de nuevos nodos y el balanceo de carga inicial.
3. **Cliente:**
   - Entidad externa que se conecta a cualquier nodo (o al coordinador) para realizar búsquedas o solicitar indexación de archivos.

Distribución en Docker

El sistema se despliega sobre una red virtual de Docker (`bridge` o `overlay`). Cada nodo es un contenedor independiente que expone:

- Un puerto TCP para comandos y transferencia de datos.
- Un puerto UDP para descubrimiento (Multicast) y heartbeats.

---

2. Procesos

Tipos de Procesos

Cada nodo del sistema (`search-node`) ejecuta un proceso principal en Python (`main_distributed.py`). Dentro de este proceso, se utiliza un modelo de concurrencia basado en **Hilos (Threading)**.

Organización de Procesos

- **Main Thread:** Inicia el servidor y la lógica principal.
- **TCP Server Thread:** Un servidor `ThreadingTCPServer` que genera un hilo nuevo por cada conexión de cliente o nodo entrante para manejar comandos (RPC simulado).
- **Heartbeat Monitor Thread:** Un hilo en segundo plano (`HeartbeatMonitor`) que envía y verifica señales de vida de otros nodos.
- **Discovery Thread:** Un hilo dedicado al descubrimiento de nodos mediante escaneo de subred TCP y conexión periódica a peers conocidos.
- **Subnet Scanner Thread:** Escanea la subred local en paralelo buscando nodos activos en el puerto 5000.

Patrón de Diseño

Se utiliza el patrón **Reactor/Event-Driven** para la red (a través de `socketserver`) combinado con **Worker Threads** para tareas de larga duración (como la indexación o la propagación de réplicas) para no bloquear el hilo principal de comunicación.

---

3. Comunicación

Tipo de Comunicación

La comunicación es **100% TCP**, optimizada para fiabilidad y compatibilidad con Docker:

1. **TCP (Sockets):** Para operaciones críticas que requieren fiabilidad (Indexar, Buscar, Replicar, Votar en elección). Se utiliza un protocolo de mensajes basado en **JSON**.
2. **TCP (IP Cache Discovery):** Para descubrimiento automático de nodos mediante escaneo de subred y registro bidireccional.
3. **TCP (Health Check):** Para verificar la salud de los nodos (heartbeat).

Comunicación Cliente-Servidor

El cliente se conecta vía TCP enviando un objeto JSON con el formato:

```
{ "command": "search", "query": "termino", "args": {...} }
```

El servidor procesa y responde con un JSON.

Comunicación Servidor-Servidor

Los nodos se comunican entre sí para:

- **Replicación:** Transferencia de datos a los nodos sucesores en el anillo.
- **Elección:** Intercambio de mensajes del algoritmo Bully.

- **Consulta Distribuida:** Propagación de la búsqueda a otros nodos si es necesario.

---

4. Coordinación

Toma de Decisiones Distribuidas (Elección de Líder)

Se implementa el **Algoritmo Bully**.

- Cuando un nodo detecta que el coordinador ha caído (timeout de heartbeat), inicia una elección.
- Los nodos con ID más alto tienen prioridad.
- El ganador se anuncia a todos los demás como el nuevo coordinador.

Sincronización

- **Heartbeats:** Los nodos envían pulsos periódicos. Si un nodo deja de responder, se marca como inactivo y se actualiza la lista de miembros (`membership list`).
- **Acceso a Recursos:** Cada nodo gestiona su propio índice local con bloqueos a nivel de hilo (`threading.Lock`) para evitar condiciones de carrera durante la escritura (indexación) y lectura (búsqueda) simultáneas.

---

5. Nombrado y Localización

Identificación

- **Nodos:** Se identifican por un ID único (ej. `node1`, `node2`) y su tupla de dirección `(IP, Puerto)`.
- **Datos (Archivos):** Se identifican por el hash de su contenido o nombre.

Ubicación (Consistent Hashing)

Para localizar dónde debe guardarse o buscarse un dato, se utiliza un **Anillo de Hash Consistente** (`ConsistentHashRing`).

1. Se calcula el hash del nombre del archivo.
2. Se ubica el nodo cuyo hash es igual o inmediatamente superior en el anillo.
3. Esto elimina la necesidad de un servidor de nombres centralizado para localizar datos.

Descubrimiento

Se utiliza **IP Cache Discovery**

1. **Escaneo de Subred:** Al iniciar, cada nodo escanea la subred local (ej. `10.0.1.0/24`) buscando otros nodos en el puerto 5000.
2. **Registro Bidireccional:** Cuando un nodo A contacta a un nodo B para obtener peers (`get_peers`), A envía su propia información. B registra a A automáticamente, creando un registro mutuo.
3. **Propagación de Peers:** Cuando B responde a A, incluye la lista de todos los nodos que conoce. A registra esos nodos y puede contactarlos directamente.
4. **Seed Nodes (Opcional):** Como respaldo, se pueden configurar nodos conocidos de antemano para acelerar el descubrimiento inicial.

---

6. Consistencia y Replicación

Distribución de Datos

Los archivos no se guardan en un solo nodo. El índice invertido se fragmenta (sharding) basado en los términos o documentos.

Replicación

Se utiliza una estrategia de **Replicación en Cadena** sobre el anillo de hash.

- **Factor de Replicación (N):** Configurable (ej. N=3).
- Un dato se guarda en el nodo propietario (determinado por el hash) y en sus `N-1` sucesores inmediatos en el anillo.

Consistencia (Quorum)

Para garantizar la fiabilidad, se implementa un sistema de **Quorum** (`quorum.py`):

- **Escritura (W):** La operación de indexado solo se confirma si al menos W réplicas confirman la escritura.
- **Lectura (R):** Se puede configurar para leer de varias réplicas y comparar versiones, asegurando que no se lean datos obsoletos.

---

7. Tolerancia a Fallas

Detección de Fallos

El módulo `HeartbeatMonitor` verifica constantemente la disponibilidad de los pares.

Respuesta a Errores

1. **Caída de un Nodo Par:**

- El sistema detecta la caída.
- Las peticiones de búsqueda se redirigen automáticamente a los nodos que poseen las réplicas de los datos del nodo caído (gracias al anillo de hash).

2. **Caída del Coordinador:**
   - Se dispara el algoritmo Bully.
   - Se elige un nuevo líder automáticamente.
   - El sistema sigue operando sin interrupción grave del servicio.
3. **Nodos Nuevos:**
   - Al entrar, se le asigna un rango del anillo de hash.
   - (Futuro/Mejora) Se transfieren las llaves correspondientes desde los nodos vecinos para balancear la carga.

---

8. Seguridad

Diseño Actual

El sistema confía en la seguridad perimetral proporcionada por la red de Docker.

- **Aislamiento:** Los nodos corren en una red privada interna. Solo los puertos necesarios se exponen al host.

Vulnerabilidades y Mejoras

- **Autenticación:** Actualmente no hay autenticación entre nodos; cualquier proceso en la red interna puede enviar comandos.
- **Encriptación:** La comunicación es texto plano (JSON).
- **Autorización:** No hay roles de usuario diferenciados (admin vs usuario).

Resumen de Tecnologías

- **Lenguaje:** Python 3.9+
- **Librerías Base:** `socket`, `threading`, `json`, `struct`.
- **Infraestructura:** Docker, Docker Networking.