Informe de Diseño: Sistema de Búsqueda Distribuido basado en Chord

Este documento detalla el diseño de un sistema de búsqueda distribuido que utiliza la arquitectura Chord para almacenar y recuperar archivos, gestionando eficientemente archivos con contenido idéntico pero nombres diferentes.

1. Arquitectura

1.1 Organización del Sistema Distribuido:

El sistema se basa en la arquitectura Chord, donde cada nodo tiene un identificador único (hash de su dirección IP). Los archivos se identifican mediante el hash de su contenido (SHA-256). Esta estrategia asegura unicidad, independientemente del nombre del archivo. Un mismo contenido, con nombres distintos, tendrá un único identificador.

1.2 Roles del Sistema:

* Clientes: Suben archivos, calculando el hash del contenido. Realizan búsquedas por nombre o extensión. Descargan los archivos del nodo responsable del hash de contenido.
* Servidores (Nodos Chord): Almacenan archivos y metadatos. Gestionan la pertenencia al anillo Chord (unión y salida de nodos). Procesan las solicitudes de clientes, incluyendo búsquedas, almacenamiento y recuperación. Realizan la replicación de datos.

1.3 Distribución de Servicios en Redes Docker:

Empleamos dos redes Docker separadas: una para clientes y otra para servidores, interconectadas por un router. La red de servidores aloja los nodos Chord, gestionando la comunicación directa entre ellos para el mantenimiento de la estructura del anillo, enrutamiento de claves y almacenamiento de datos. La red de clientes provee acceso independiente a los nodos, usando el router como intermediario. Los servicios de mantenimiento del anillo operan exclusivamente en la red de servidores.

2. Procesos

2.1 Tipos de Procesos:

Cada nodo (cliente o servidor) gestiona múltiples procesos concurrentes utilizando hilos.

2.2 Organización de Procesos:

* Cliente: Los procesos de subida, búsqueda y descarga operan en hilos separados, gestionando eventos de forma asíncrona, permitiendo a los usuarios continuar con otras acciones.
* Servidor: Un hilo principal gestiona el inicio del servidor, la aceptación de nuevas conexiones y el enrutamiento de solicitudes a hilos secundarios. Se usan hilos dedicados para manejar solicitudes de cliente (búsqueda, subida, descarga) de forma independiente. Hilos adicionales gestionan la comunicación entre nodos y el mantenimiento del anillo Chord. Se utiliza el modelo de hilos para un procesamiento concurrente más eficiente.

2.3 Patrón de Diseño:

Se utiliza un modelo asíncrono basado en hilos para la gestión de solicitudes de clientes y la comunicación entre nodos, mejorando la eficiencia y la capacidad de respuesta del sistema.

3. Comunicación

3.1 Tipo de Comunicación:

Se utiliza ZeroMQ (ZMQ) para la comunicación entre los clientes y los servidores, y entre los servidores entre sí.

3.2 Comunicación Cliente-Servidor:

Se emplea el patrón Request-Reply de ZMQ. El cliente envía una solicitud (subir, buscar, descargar) al servidor, y el servidor responde acordemente.

3.3 Comunicación Servidor-Servidor:

Se utiliza el patrón Request-Reply para operaciones internas de Chord (por ejemplo, comprobar la pertenencia de un archivo).

3.4 Comunicación entre Procesos:

Dentro de cada nodo, la comunicación entre hilos se realiza mediante estructuras de datos compartidas y mecanismos de sincronización (por ejemplo, colas de mensajes).

4. Coordinación

* Sincronización: La arquitectura Chord, al usar nodos independientes, no necesita sincronización global. La coordinación se da a nivel de nodo, utilizando mecanismos como locks para el acceso exclusivo a la base de datos local.
* Acceso Exclusivo a Recursos: Se utiliza `threading.Lock` en Python para evitar conflictos al acceder a los archivos de un nodo, garantizando integridad.
* Toma de Decisiones Distribuidas: La toma de decisiones se delega a cada nodo para su parte del anillo, simplificando el proceso y mejorando la eficiencia.

5. Nombrado y Localización

5.1 Identificación de Datos y Servicios:

Los archivos se identifican por el hash SHA-256 de su contenido. Los nodos por su hash de dirección IP. Los nombres de archivos y extensiones se almacenan en metadatos indexados por el hash del nombre.

5.2 Ubicación de Datos y Servicios:

Los archivos se localizan a través del hash del contenido, dirigiéndose al nodo responsable de la clave. Los metadatos se usan para localizar la clave del contenido a partir del nombre o extensión.

5.3 Localización de Datos y Servicios:

El enrutamiento en Chord facilita la localización eficiente de archivos y metadatos, utilizando la tabla de dedos (finger table) para la navegación en el anillo.

6. Consistencia y Replicación en Sistemas Distribuidos

6.1 Distribución de los Datos

La distribución de datos implica asignar partes del conjunto de datos total a diferentes nodos dentro del sistema distribuido. Esto se realiza para garantizar la escalabilidad y mejorar el rendimiento del sistema.

- Hashing distribuido: Los datos se asignan a nodos utilizando un algoritmo de hashing que distribuye uniformemente las claves en el sistema.
- Responsabilidad de los nodos: Cada nodo es responsable de un rango específico de claves, lo que asegura que los datos estén organizados y accesibles eficientemente.
- Equilibrio de carga: El algoritmo de distribución debe minimizar la carga desproporcionada en un nodo específico, asegurando un equilibrio adecuado en el sistema.

6.2 Replicación y Cantidad de Réplicas

La replicación consiste en mantener múltiples copias de los mismos datos en diferentes nodos para garantizar la disponibilidad y tolerancia a fallos.

Factores Clave:

- Cantidad de réplicas (`k`): Define cuántas copias de un dato se mantienen en el sistema. Un valor típico de `k` asegura la resiliencia ante fallos de múltiples nodos.
- Ubicación de réplicas: Las réplicas se distribuyen entre los nodos cercanos (por ejemplo, sucesores en un anillo Chord) para optimizar el acceso y minimizar la latencia.
- Costo de replicación: Aunque mejora la disponibilidad, mantener muchas réplicas puede aumentar los costos de almacenamiento y la complejidad del sistema.

6.3 Confiabilidad de las Réplicas de los Datos tras una Actualización

La confiabilidad de las réplicas implica garantizar que todas las copias de un dato sean consistentes tras cualquier operación de escritura o actualización.

Modelos de Consistencia:

- Consistencia eventual: En este modelo, las actualizaciones en un nodo eventualmente se propagan a todas las réplicas. Aunque permite inconsistencias temporales, es más eficiente en términos de latencia.
- Consistencia fuerte: Garantiza que todas las réplicas se actualicen antes de que la operación de escritura sea confirmada. Esto asegura que las lecturas sean consistentes en cualquier momento, pero puede aumentar la latencia.
- Consistencia basada en quorum: Combina flexibilidad y confiabilidad, exigiendo que las actualizaciones sean aplicadas a un subconjunto mínimo de réplicas (quorum) antes de considerarse exitosas.

Mecanismos de Sincronización:

- Protocolos de replicación activa: Todos los nodos aplican las actualizaciones simultáneamente.

- Protocolos de replicación pasiva: Un nodo primario aplica la actualización y luego la replica a los secundarios.
- Detección y recuperación de fallos: Si un nodo falla durante una actualización, el sistema debe reintentar o transferir los datos a un nodo de respaldo.

7. Tolerancia a Fallas en Sistemas Distribuidos

7.1 Respuesta a Errores

Un sistema distribuido debe ser capaz de detectar, manejar y recuperarse de errores sin afectar su funcionalidad general.

Estrategias:

- Detección de fallos: Uso de mecanismos como *heartbeats* (señales periódicas) para monitorear la conectividad entre nodos.
- Reintentos automáticos: Los nodos deben intentar reestablecer la comunicación o repetir operaciones fallidas.
- Failover: Transferir las tareas de un nodo fallido a otros nodos activos en el sistema.
- Replicación: Mantener múltiples copias de datos y servicios en diferentes nodos para garantizar disponibilidad.
- Reconfiguración dinámica: El sistema debe adaptarse automáticamente a la ausencia de nodos y redistribuir los datos o responsabilidades.

7.2 Nivel de Tolerancia a Fallos Esperado

El nivel de tolerancia a fallos necesario depende de los requisitos del sistema y el impacto que tendría un fallo en la experiencia del usuario.

Factores a Considerar:

- Disponibilidad requerida: ¿Cuánto tiempo puede permitirse el sistema estar inactivo?
- Cantidad de fallos soportados simultáneamente: Define cuántos nodos pueden fallar antes de que el sistema deje de ser funcional.
- Costo asociado: Incrementar la tolerancia a fallos (réplicas, nodos adicionales, etc.) implica mayores costos de infraestructura y operación.

Diseño para tolerancia:

- Alta disponibilidad: Garantizar que el sistema esté operativo el mayor porcentaje de tiempo posible.
- Mecanismos de quorum: Asegurar que ciertas operaciones críticas solo se realicen si un número mínimo de nodos está disponible.
- Consistencia eventual o fuerte: Balancear entre la confiabilidad y el rendimiento según el caso.

7.3 Fallos Parciales

Nodos caídos temporalmente

- Manejo de nodos inactivos:
    - Marcar los nodos inactivos y evitar enviarles solicitudes hasta que vuelvan a estar activos.
    - Reasignar temporalmente las responsabilidades de los nodos caídos a sus vecinos o réplicas.
    - Reintentar la comunicación con el nodo caído tras intervalos definidos.
- Reconexión y reintegración:
    - Cuando un nodo vuelva a estar disponible, sincronizarlo con el estado actual del sistema.
    - Recuperar cualquier dato que haya perdido durante su inactividad.

Nodos nuevos que se incorporan al sistema

- Integración dinámica:
    - Un nuevo nodo debe ser capaz de unirse al sistema sin interrumpir su funcionamiento.
    - Redistribuir datos al nodo nuevo para balancear la carga.
    - Actualizar las tablas de enrutamiento de los nodos existentes para reflejar la incorporación del nuevo nodo.
- Estabilización:
    - Garantizar que las réplicas y los enlaces de enrutamiento se actualicen para incluir al nuevo nodo.
    - Verificar la consistencia del sistema tras la integración.

8. Seguridad en Sistemas Distribuidos

8.1 Seguridad con Respecto a la Comunicación

Las comunicaciones entre nodos en un sistema distribuido deben ser protegidas para evitar la interceptación o manipulación de datos.

Estrategias:

- Cifrado de datos en tránsito:
    - Utilizar protocolos seguros como TLS (Transport Layer Security) para cifrar todas las comunicaciones entre nodos.
    - Asegurar que las claves y certificados sean gestionados adecuadamente.
- Integridad de los datos:

    - Implementar algoritmos de hash (e.g., SHA-256) para garantizar que los datos no han sido modificados durante la transmisión.

- Prevención de ataques:
    - Ataques Man-in-the-Middle (MITM): Usar autenticación mutua entre nodos para prevenir interceptaciones.
    - Ataques de denegación de servicio (DoS): Implementar límites en las solicitudes aceptadas por nodo y mecanismos para detectar patrones anómalos de tráfico.

8.2 Seguridad con Respecto al Diseño

Un diseño seguro considera la protección contra vulnerabilidades en la arquitectura del sistema.

Principios de diseño seguro:

- Minimización de la superficie de ataque:
    - Reducir los puntos de entrada al sistema, exponiendo solo las interfaces esenciales.
    - Aislar servicios críticos en entornos controlados (e.g., contenedores o redes privadas).
- Defensa en profundidad:
    - Implementar capas de seguridad, como firewalls, redes privadas virtuales (VPN), y restricciones de acceso por rol.
- Auditoría y monitoreo:
    - Registrar las operaciones del sistema para detectar actividades sospechosas.
    - Analizar los registros de acceso para identificar posibles intentos de intrusión.
- Actualizaciones y parches:
    - Mantener el software actualizado para cerrar vulnerabilidades conocidas.

8.3 Autorización y Autenticación

La autorización y autenticación garantizan que solo los usuarios y nodos legítimos puedan acceder a los recursos del sistema.

Autenticación:

- Mecanismos de autenticación:
    - Uso de credenciales seguras como contraseñas, certificados digitales o autenticación basada en claves públicas y privadas.
    - Autenticación multifactor (MFA) para añadir una capa extra de seguridad.
- Gestión de identidad:
    - Implementar un sistema centralizado o distribuido para autenticar usuarios y nodos (e.g., OAuth, LDAP).

Autorización:

- Control de acceso:
    - Definir políticas claras para especificar qué usuarios o nodos pueden acceder a qué recursos.
    - Usar modelos como el control de acceso basado en roles (RBAC) para asignar permisos de manera estructurada.
- Revisión de permisos:
    - Implementar auditorías regulares para verificar que los permisos asignados sean los mínimos necesarios para cada usuario o nodo.