Propuesta para el Diseño de una Red Social Distribuida: Distnet

Funcionalidades

Distnet, es una red social distribuida para la comunicación que sigue la filosofía en la cual se basa Twitter. En ella, los usuarios que estén autenticados podrán publicar mensajes, volver a publicar aquellos de otros usuarios que consideren interesantes, así como elegir quiénes serán sus amigos. Con este fin, se implementará una infraestructura distribuida para el manejo de usuarios y la autenticación.

Solución

Arquitectura

¿Por qué hacer un sistema *distribuido*?

La adopción de una arquitectura distribuida en el desarrollo de *Distnet* (una red social) ofrece múltiples ventajas tanto técnicas como prácticas, especialmente en lo que respecta a la escalabilidad, la disponibilidad y la autonomía del usuario.

En cuanto a la escalabilidad, un sistema distribuido proporciona ventajas dado que las redes sociales manejan volúmenes de datos extraordinarios, desde mensajes y perfiles hasta interacciones como "me gusta" (re-publicar mensajes, en el caso de *Distnet*). Una arquitectura distribuida permite gestionar esta gran cantidad de información de manera eficiente, por lo que al repartir los datos entre múltiples nodos, se evitan cuellos de botella y se facilita un crecimiento horizontal, en el que se añaden más nodos según la demanda.

Otro aspecto que se ve beneficiado con este tipo de arquitectura es la disponibilidad continua, la cual resulta imprescindible para cualquier red social. Una arquitectura distribuida asegura esta característica mediante la redundancia y replicación de datos. Al mantener copias de la información en distintos nodos, la plataforma se conserva operativa incluso ante el fallo de uno o varios componentes. Esto evita interrupciones perceptibles para los usuarios.

El rendimiento general también se ve favorecido por una arquitectura distribuida. Los usuarios se conectan a nodos geográficamente cercanos, lo que reduce la latencia en comparación con un único servidor centralizado. Además, las operaciones complejas, como análisis de datos o búsquedas, se pueden distribuir entre varios nodos, incrementando la velocidad y el rendimiento.

La protección de los datos y la privacidad de los usuarios también se refuerzan. Al dispersar la información de los usuarios, se reduce el riesgo de que una sola entidad pueda acceder a ella, minimizando las posibilidades de vigilancia masiva o violaciones de la privacidad. Además, en redes sociales descentralizadas, la confianza de los usuarios aumenta al no haber un solo controlador que pueda manipular la información o censurar contenidos.

En resumen, la arquitectura distribuida no solo es una solución técnica avanzada, sino una estrategia esencial para el desarrollo de redes sociales escalables, seguras y centradas en el usuario. Al abordar los desafíos de la escalabilidad, la disponibilidad, la descentralización y la eficiencia, esta arquitectura se posiciona como una base sólida para las plataformas sociales del futuro.

Comparación entre Chord, Pastry y Kademlia.

• Chord: Una de las arquitecturas más tempranas para redes P2P (peer-to-peer) y DHT (Distributed Hash Table). Se centra en la distribución eficiente de datos usando un anillo virtual. Cada nodo tiene un identificador único (ID) en el anillo y mantiene información sobre sus sucesores y predecesores.
• Pastry: Arquitectura DHT que organiza los nodos en un espacio de ID multidimensional (normalmente con una base de 2^b). Utiliza una tabla de ruteo organizada en prefijos comunes para encaminar mensajes. Cada nodo conoce un conjunto de otros nodos basados en sus IDs, facilitando rutas eficientes.
• Kademlia: Arquitectura, también DHT, que utiliza una métrica XOR para calcular distancias entre nodos. Los nodos mantienen tablas de enrutamiento organizadas en "buckets" (cubetas) basadas en la distancia XOR con respecto a su propio ID. Se caracteriza por su robustez y eficiencia en redes con alta tasa de cambio de nodos.

Característica	Chord	Pastry	Kademlia
Espacio de ID	Anillo virtual unidimensional	Espacio multidimensional (base 2^b)	Espacio de ID unidimensional
Organización de nodos	Anillo con sucesores y predecesores	Tablas de enrutamiento por prefijo común	“Buckets” por distancia XOR
Métrica de distancia	Distancia en el anillos	Prefijo Común	Distancia XOR
Ruteo	Salto al sucesor más cercano a la clave	Ruteo por prefijos comunes	Ruteo basado en distancia XOR
Complejidad de Ruteo	O(log N)	O(log N)	O(log N)
Robustez	Buena pero susceptible a fallas de nodos en anillos	Buena, rutas alternativas si falla un nodo	Muy robusta, buen manejo de churn
Escalabilidad	Buena pero puede requerir ajustes al crecer	Muy buena, diseñada para gran escala	Muy buena especialmente en entornos dinámicos
Simplicidad	Relativamente simple de implementar	Un poco más compleja que Chord	Considerada una de las más simples
Eficiencia	Buena	Generalmente más eficiente en rutas	Muy eficiente, especialmente con XOR métrica

Arquitectura	Ventajas	Desventajas
Chord	• Fácil de entender e implementar. • Buena escalabilidad • Rutas relativamente cortas	• Puede ser vulnerable a fallas de nodos en el anillos, especialmente si se concentran en un sector • Necesita un proceso de

		estabilización constante.
Pastry	• Escalabilidad superior a Chord. • Rutas más cortas y eficientes que Chord en muchos casos • Mayor tolerancia a fallos por su estructura de prefijos	• Implementación más compleja que Chord • Mantenimiento de tablas de enrutamiento requiere más recursos
Kademlia	• Muy robusta ante cambios de nodos (churn) • Ruteo eficiente y simple • Escalabilidad excelente • Amplia adopción y madurez	• El mecanismo XOR podría crear situaciones donde una búsqueda no alcance el nodo más cercano en términos geográficos

¿Por qué usamos Chord?

Para implementar la arquitectura distribuida de *Distnet*, se decidió usar Chord ya que se distingue por su simplicidad y diseño bien definido. En primer lugar, su mecánica es más sencilla y estructurada en comparación con alternativas como Kademlia y Pastry. Esto facilita su implementación, comprensión y depuración. Además, su algoritmo de ruteo, aunque básico, resulta efectivo, pues emplea una estructura de anillo ordenado donde cada nodo conoce a su sucesor y tiene un conjunto de nodos en su "tabla de dedos", reduciendo así la complejidad de la gestión del ruteo.

En segundo lugar, garantiza un balance de carga adecuado. La distribución de claves, ya sean tweets o perfiles, se realiza de forma uniforme entre los nodos mediante una función hash consistente. Esto, a su vez, minimiza la formación de cuellos de botella en nodos concretos, ya que la asignación de claves es automática e independiente de factores externos.

Otro aspecto favorable es su escalabilidad sencilla. Al igual que Kademlia y Pastry, Chord ofrece una complejidad de búsqueda logarítmica, lo que permite su expansión a millones de nodos. Además, su estructura de anillo facilita la incorporación gradual de nuevos nodos sin requerir complicadas reconfiguraciones.

Asimismo, Chord posee una tolerancia a fallos robusta. Su estructura de anillo y el seguimiento de sucesores aseguran que siempre existan rutas alternativas, facilitando la gestión de desconexiones o fallos en los nodos. Si un nodo falla, su sucesor puede tomar el control de las claves que gestiona, disminuyendo el impacto de la incidencia.

Además, se destaca por su consistencia y replicación sencilla. Chord permite replicar datos en nodos sucesores de manera eficiente, asegurando la disponibilidad de la información incluso ante el fallo de un nodo específico. Los datos asociados a una clave siempre se mapean al nodo responsable o a sus réplicas, eliminando de esta forma cualquier ambigüedad en la búsqueda.

Por último, Chord resulta más apropiado para sistemas descentralizados. No requiere un punto central de coordinación ni una jerarquía, lo que lo convierte en una opción ideal para

sistemas distribuidos. Además, la incorporación de nodos es sencilla, ya que cada nodo puede conectarse al anillo con un conocimiento mínimo de la red existente, simplificando su despliegue. Su enfoque, a diferencia de Pastry que utiliza rutas basadas en prefijos con mayor complejidad, emplea un modelo de ruteo en un espacio lineal.

En resumen, se elige Chord debido a su simplicidad, que lo hace más fácil de implementar y mantener que Kademlia y Pastry. Su balance de carga distribuye los datos equitativamente, evitando puntos de saturación. Su escalabilidad permite su expansión manteniendo la complejidad logarítmica. También se destaca su resiliencia, ya que tanto sucesores como la replicación manejan fallos con eficacia. Finalmente, es ideal para sistemas descentralizados, al no requerir coordinación central ni estructuras complejas.

Nuestra implementación.

Almacenamiento Distribuido

Se propone implementar un sistema de almacenamiento distribuido basado en una Tabla de Hash Distribuida (DHT) utilizando el algoritmo Chord. Este enfoque asegura una distribución equitativa de los datos y permite la escalabilidad del sistema. Cada nodo en el anillo de Chord tendrá un identificador único generado mediante una función de hash, lo que será aprovechado para gestionar la ubicación de los datos relacionados con los usuarios.

Dado que las redes sociales operan principalmente mediante usuarios registrados que intercambian información, se usará la misma función de hash para derivar el identificador del nodo y para los nombres de usuario (nicks). Esto asegura que:

- Los datos asociados a un usuario (como publicaciones, seguidores, y credenciales) se almacenan en el mismo nodo responsable de ese segmento en el anillo.
- La información de usuarios con identificadores en diferentes segmentos del anillo se distribuye en diferentes nodos, promoviendo la descentralización.

Esta estrategia permite que, a medida que la cantidad de usuarios crezca, la carga de almacenamiento y procesamiento en los nodos se balancee automáticamente, favoreciendo la escalabilidad de la red.

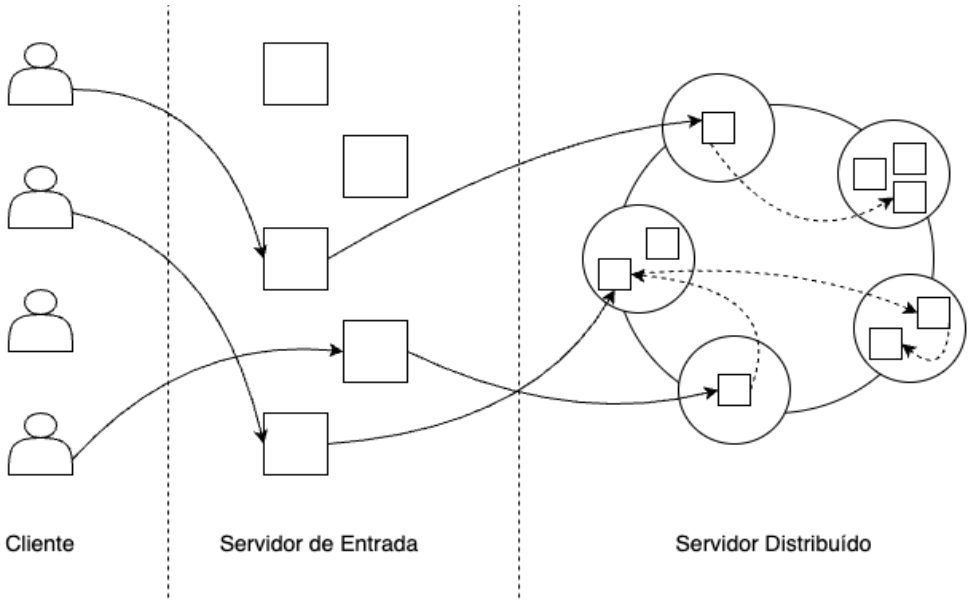
Componentes y arquitectura del sistema

La arquitectura de “Distnet” está diseñada para equilibrar simplicidad y eficiencia, y consta de tres componentes principales:

- **Cliente**: Representa la parte del sistema que interactúa con el usuario. Los clientes se conectan a la red a través de diversos dispositivos, como portátiles y teléfonos inteligentes. Su función es interactuar con el sistema, enviando solicitudes y recibiendo respuestas para tareas como publicar mensajes o recuperar contenido.
- **Servidor de Entrada**: Actúa como un nodo de puerta de enlace, facilitando el acceso de los clientes a la red. Agiliza el descubrimiento de nodos y enruta las solicitudes hacia la

parte correspondiente del sistema. Este componente garantiza una experiencia fluida tanto para los usuarios como para los nuevos nodos.

- **Servidor Distribuido**: Una estructura de servidor consolidada que integra dos capas funcionales:
    - **Capa Chord**: Gestiona la tabla de dispersión distribuida (DHT) utilizando el protocolo Chord, asegurando un enrutamiento eficiente y la localización de recursos dentro de la red.
    - **Capa Procesamiento**: Maneja funcionalidades específicas de la aplicación, como el almacenamiento de datos de los usuarios, la gestión de tuits y el soporte para solicitudes concurrentes mediante multihilo.



Justificación de la Arquitectura

- Escalabilidad: La implementación distribuida permite añadir nuevos nodos fácilmente, garantizando un balance de carga dinámico a medida que crecen los usuarios y los datos.
- Tolerancia a fallos: La naturaleza descentralizada asegura que, en caso de fallo de un nodo, el sistema pueda redistribuir sus responsabilidades sin afectar significativamente el funcionamiento global.
- Optimización del almacenamiento: Asociar datos de usuarios a segmentos específicos del anillo minimiza el tráfico entre nodos y mejora el rendimiento.

Roles del sistema

El diseño de “Distnet” se fundamenta en una arquitectura distribuida que asigna roles específicos a cada uno de los componentes para garantizar la eficiencia, escalabilidad y seguridad. A continuación, se detallan las funcionalidades de cada componente y las razones detrás de su diseño.

Cliente (Client)

El cliente es el punto de interacción más cercano al usuario final. Este componente proporciona la interfaz necesaria para que los usuarios puedan comunicarse con los servicios generales de Distnet.

- Funcionalidad principal:
- Establece conexiones con los servidores de entrada (EntryServer) para realizar acciones como el envío y recepción de publicaciones, registro, autenticación y consulta de información.
- Justificación del diseño: Centralizar las interacciones del usuario en un único componente facilita la implementación de una interfaz de usuario intuitiva y reduce la complejidad en el lado del cliente.

Servidor de Entrada

El EntryServer es un componente intermedio crítico que actúa como mediador entre los clientes y los servidores de procesamiento, garantizando tanto la seguridad como la fiabilidad en la red.

- Responsabilidades clave:
- Recibe y valida las solicitudes de los clientes antes de reenviarlas al servidor correspondiente.
- Minimiza el contacto directo entre los clientes y los servidores internos, dificultando posibles ataques o accesos no autorizados.
- Detecta fallos en la red y notifica a los clientes en caso de problemas.
- Facilita la integración de nuevos nodos en el sistema distribuyendo los contactos iniciales necesarios para que el algoritmo Chord inserte un servidor correctamente.
- Justificación del diseño: Al aislar las interacciones externas del núcleo de la red, se mejora significativamente la seguridad y el control del tráfico en el sistema.

Servidor Distribuido

Capa Chord

El ChordServer es el componente encargado de mantener la estructura de la Dispersión de Hash Distribuida (DHT) y garantizar el correcto funcionamiento del anillo.

- Funciones principales:
- Almacena y actualiza constantemente las tablas de referencia (finger tables) necesarias para localizar recursos dentro del sistema.
- Colabora estrechamente con los TweeterServers para identificar la ubicación de datos específicos en el anillo.
- Justificación del diseño: La implementación de Chord asegura una búsqueda eficiente de datos mediante la estructura de DHT, lo que permite una escalabilidad prácticamente ilimitada y una rápida redistribución de recursos en caso de cambios en la red.

Capa de Procesamiento

El TweeterServer gestiona la lógica del sistema y garantiza la consistencia de los datos relacionados con los usuarios y sus interacciones.

- Funciones principales:
- Realiza operaciones como autenticación de usuarios, verificación de publicaciones, y gestión de seguidores.
- Almacena y recupera datos en la base de datos distribuida (Tweets, ReTweets, usuarios, tokens, etc.).
- Réplica datos entre servidores del mismo nodo para asegurar la redundancia y tolerancia a fallos.
- Se comunica con otros servidores Tweeter para intercambiar información o transferir datos en caso de cambios en la red.
- Interacciones clave:
- Con el EntryServer: Maneja las solicitudes de los usuarios transmitidas por el EntryServer.
- Con el ChordServer: Coordina la ubicación y recuperación de datos dentro de la DHT.
- Justificación del diseño: La distribución de datos entre múltiples TweeterServers asegura la redundancia, facilita la escalabilidad y optimiza el uso de recursos.

Procesos

En el diseño de nuestra red social distribuida, hemos optado por implementar un servidor multihilo para atender eficientemente a múltiples clientes de forma simultánea. Esta decisión se basa en la necesidad de manejar un gran número de conexiones concurrentes y optimizar el uso de recursos del sistema.

Diseño del Servidor Multihilo

Hemos encapsulado la funcionalidad básica de un servidor multihilo en la clase `MultiThreadedServer`. Esta clase abstracta proporciona una base robusta para implementar diferentes servicios en nuestra red social. Las características principales incluyen:

1. Configuración Flexible: Al instanciar el servidor, se puede especificar el número de hilos, el puerto de escucha y un delegado que define el comportamiento específico para atender a cada cliente.

2. Manejo Automático de Conexiones: El servidor escucha continuamente en el puerto especificado y asigna cada nueva conexión a un hilo disponible.

3. Abstracción de Complejidad: Los desarrolladores pueden centrarse en la lógica específica del servicio sin preocuparse por los detalles de la gestión de hilos y conexiones.

Justificación del Enfoque Multihilo

Elegimos un enfoque multihilo en lugar de multiproceso por las siguientes razones:

1. Eficiencia en la Comunicación: Los hilos comparten el mismo espacio de memoria, facilitando la comunicación y el intercambio de datos entre ellos.

2. Menor Sobrecarga: La creación y gestión de hilos es generalmente más ligera que la de procesos separados.

3. Facilidad de Implementación: Go ofrece herramientas integradas para la programación multihilo (goroutines, Channels, Select, Mutex) que simplifican el desarrollo.

Mecanismos de Sincronización

Para garantizar la integridad de los datos compartidos entre hilos, aprovecharemos los recursos que brinda el lenguaje Golang:

1. Mutexes: para bloquear el acceso concurrente a recursos compartidos.

2. Colas Thread-Safe: Implementamos una cola multiproductor-multiconsumidor para gestionar eficientemente las tareas entre los hilos.

3. Canales (Channels): para enviar señales (un valor booleano, un struct vacío, etc.) desde una gorutina a otras, indicando que algo ha sucedido, como una solicitud de detención. También sirven para coordinar acciones, por ejemplo, esperar a que una gorutina termine antes de continuar con otra tarea.

Gestión de Recursos

Para optimizar el rendimiento, implementamos:

1. Pool de Hilos (Goroutines): Reutilizamos un conjunto fijo de goroutines para atender las conexiones, evitando la sobrecarga de crear nuevos hilos constantemente.

2. Cola de Tareas: Utilizamos una cola para distribuir las conexiones entrantes entre los hilos disponibles. Para ello, usamos Channel como Cola de Tareas: el canal se convierte en la cola donde se encolan (valga la redundancia) las conexiones entrantes (o la información de conexión).

Manejo de Peticiones Asíncronas

Para manejar peticiones que requieren respuestas asíncronas, usamos los Channels y las goroutines de Go:

1. Goroutines para Lanzar Tareas Asíncronas:

- Concepto: La idea fundamental es lanzar una nueva goroutine cada vez que se necesita realizar una operación asíncrona.
- Implementación: Simplemente antepones go a la llamada de la función que quieres ejecutar de forma asíncrona.

2. Canales para Recibir Resultados Asíncronos:

- Concepto: Los canales se usan para enviar resultados de las tareas asíncronas al código que las invocó.
- Implementación: Se crea un canal del tipo de dato que se recibirá de la tarea asíncrona. La goroutine asíncrona enviará el resultado al canal cuando termine. La goroutine que invoca la tarea espera en el canal hasta que reciba el resultado.

Este diseño permite una gestión eficiente de peticiones que pueden tomar tiempo en procesarse, manteniendo la capacidad de respuesta del sistema.

Diseño de la Tabla Hash Distribuida (DHT) basada en Chord

Para garantizar la escalabilidad de nuestra red social distribuida, hemos decidido implementar una Tabla Hash Distribuida (DHT) basada en el protocolo Chord. Esta elección nos permite distribuir eficientemente la base de datos entre múltiples servidores y localizar rápidamente la información requerida para responder a las consultas de los usuarios.

Estructura y Funcionamiento

1. Organización en Anillo: Cada servidor recibe un identificador único (ID) y se organiza en un anillo virtual, ordenado en sentido horario.

2. Asignación de Datos: Cada dato se asocia a una clave numérica y se almacena en el servidor cuyo ID es el sucesor inmediato de esa clave en el anillo.

3. Caso Especial: El nodo con el ID más bajo es responsable de las claves menores que su ID y mayores que el ID máximo del anillo.

Optimización de Búsquedas

Para mejorar la eficiencia de las búsquedas, implementamos una estructura de datos llamada "Finger Table":

1. Finger Table: Cada nodo mantiene una tabla con log_2(N) entradas, donde N es el número máximo de nodos posibles.

2. Estructura de la Tabla: La i-ésima entrada contiene el sucesor del nodo actual más 2^{i-1} en el espacio de identificadores.

3. Proceso de Búsqueda: Las consultas se redirigen al nodo con el mayor ID menor que la clave buscada, reduciendo significativamente el número de saltos necesarios.

Inserción de Nuevos Nodos

La inserción de nuevos nodos se realiza de manera incremental:

1. Asignación de ID: Al nuevo nodo (N) se le asigna un ID basado en el hash SHA256 de su dirección IP.

2. Localización: Se identifica el sucesor (S) y el predecesor (P) del nuevo nodo en el anillo.

3. Actualización de enlaces: N se inserta entre P y S, actualizando sus referencias mutuas.

4. Actualización de Finger Tables: Las tablas de los nodos existentes se actualizan periódicamente para incluir el nuevo nodo cuando corresponda.

Manejo de Casos Especiales

Para resolver consultas sobre claves menores que el ID del nodo actual y su predecesor:

1. Extensión del Espacio de ID: Cada nodo responde por su ID original y por (ID + MAX_ID).

2. Búsqueda Unificada: Las consultas para claves menores se transforman sumando MAX_ID, permitiendo una búsqueda uniforme en todo el espacio de claves.

Proceso de Inserción de Nuevos Nodos

Para garantizar una integración suave de nuevos nodos, se seguirá el siguiente proceso:

1. El nuevo nodo contactará con un EntryServer para obtener información sobre el anillo Chord existente.
2. Se asignará un ID único al nuevo nodo y se determinará su posición en el anillo.
3. El nodo establecerá conexiones con sus vecinos inmediatos (sucesor y predecesor).
4. Se actualizarán las tablas de enrutamiento (fingertables) de los nodos afectados.
5. El nuevo nodo notificará su inserción exitosa al EntryServer y a su TweeterServer asociado.

Esta estrategia asegura una distribución equilibrada de la carga y mantiene la integridad de la estructura del anillo Chord.

Gestión de Datos y Réplicas

Para garantizar la disponibilidad y la integridad de los datos, se implementará un sistema de replicación:

1. Inserción como Nuevo Nodo: Se copiarán los datos correspondientes al rango de hash asignado.
2. Inserción como Réplica: Se replicarán todos los datos del nodo existente.

La transferencia de datos se realizará de manera ordenada, comenzando por la tabla de usuarios para mantener la integridad referencial. Los datos se transferirán en bloques de 20 filas para optimizar el proceso.

Comunicación

La arquitectura distribuida de *Distnet*, basada en tres componentes principales (Cliente, Servidor de Entrada y Servidor de Procesamiento), requiere un sistema de comunicación eficiente, robusto y escalable. Cada tipo de comunicación –cliente-servidor, servidor-servidor y entre procesos dentro de un nodo– debe abordarse con tecnologías y patrones específicos que garanticen una interacción confiable en el entorno distribuido. A continuación, se detallan las propuestas para gestionar la comunicación en cada nivel.

Comunicación Cliente-Servidor

La interacción entre los clientes y el sistema se manejará mediante **REST APIs** implementadas sobre HTTP, utilizando JSON como formato de intercambio de datos. Este enfoque permite que los clientes (interfaces de usuario) envíen solicitudes al EntryServer para operaciones como autenticación, publicación de mensajes o consultas de contenido.
Ventajas de REST APIs:

- **Simplicidad**: REST es ampliamente utilizado, fácil de implementar y compatible con múltiples plataformas y dispositivos, incluyendo navegadores y aplicaciones móviles.
- **Escalabilidad**: Su arquitectura desacoplada permite integrar balanceadores de carga y distribuir el tráfico entre varios EntryServers.
- **Estandarización**: Al usar HTTP y JSON, se asegura que las solicitudes y respuestas sean legibles y comprensibles, facilitando la depuración y el desarrollo.

Comunicación Servidor-Servidor

En una arquitectura basada en Chord, los UnifiedServers deben interactuar entre sí para gestionar la estructura de la DHT (Distributed Hash Table) y compartir datos. Para esta comunicación, se propone el uso de **Remote Procedure Calls (RPC)** implementados con **gRPC** y **Protocol Buffers (Protobuf)**.
Razones para usar gRPC:

- **Eficiencia**: gRPC permite transferencias rápidas y compactas al utilizar Protobuf, que genera mensajes binarios más pequeños que JSON o XML.
- **Soporte de streaming**: La comunicación bidireccional de gRPC facilita operaciones como replicación de datos y actualizaciones en tiempo real.
- **Seguridad**: gRPC integra soporte nativo para conexiones seguras mediante TLS.
- **Compatibilidad**: Al ser multilenguaje, gRPC permite que los servidores se desarrollen en diferentes tecnologías, asegurando interoperabilidad.

Por ejemplo, cuando un nuevo nodo (UnifiedServer) se une al sistema, se conecta a un servidor existente mediante RPC para obtener información de sus sucesores y predecesores, además de replicar datos correspondientes a su rango de hashes.

Comunicación entre Procesos en el Servidor de Procesamiento

Para facilitar la interacción entre las capas del Servidor de Procesamiento, se propone usar un sistema de **colas de mensajes internas** basado en herramientas como **ZeroMQ** o **RabbitMQ**.
Ventajas de las colas de mensajes internas:

- **Asincronía**: Permiten que las capas trabajen de forma independiente, reduciendo bloqueos y mejorando la eficiencia.
- **Resiliencia**: Los mensajes se almacenan temporalmente en caso de que una de las capas esté ocupada o momentáneamente inactiva.
- **Modularidad**: Facilita el mantenimiento, ya que las capas pueden desarrollarse y actualizarse de manera independiente.

Por ejemplo, si la capa funcional necesita obtener información de la DHT para localizar un recurso o replicar datos, envía una solicitud a través de la cola y espera la respuesta en otro hilo.

En conclusión, el enfoque de comunicación propuesto para Distnet ha sido diseñado para garantizar que el sistema sea escalable, eficiente y resiliente frente a los desafíos que presenta una arquitectura distribuida compleja. Al emplear REST para la comunicación cliente-servidor, se asegura la simplicidad y estandarización, lo que facilita la integración con diferentes dispositivos y plataformas. Por otro lado, el uso de gRPC para la comunicación entre servidores permite transferencias rápidas, seguras y eficientes, fundamentales para mantener la consistencia y funcionalidad de la DHT en Chord.

La integración de colas de mensajes internas dentro del Servidor de Procesamiento refuerza la modularidad y asincronía, mejorando la capacidad del sistema para manejar múltiples operaciones de manera simultánea sin comprometer el rendimiento. Este enfoque no solo ofrece un sistema flexible que puede adaptarse a futuras expansiones o cambios tecnológicos, sino que también garantiza la resiliencia del sistema, permitiendo la continuidad operativa incluso ante fallos temporales o la adición de nuevos nodos. En definitiva, esta estrategia de comunicación establece una base sólida para el éxito y la sostenibilidad de Distnet como red distribuida.

Consistencia y Tolerancia a Fallos

Garantizar la consistencia y la tolerancia a fallos en la arquitectura de **Distnet** requiere una planificación cuidadosa, especialmente considerando que el sistema está diseñado para operar como una red distribuida basada en el protocolo Chord. A continuación, se detallan las estrategias implementadas para abordar los desafíos asociados con la replicación de datos, la confiabilidad, y la resiliencia frente a fallos.

Consistencia

En un sistema distribuido, mantener varias copias del mismo dato en diferentes nodos introduce el desafío de garantizar la consistencia. Para resolver este problema, **Distnet** emplea una estrategia de replicación controlada. Cada dato se asigna a un nodo primario y se replica en al menos dos o tres nodos sucesores dentro del anillo Chord. Esto asegura que, incluso si un nodo falla, las réplicas puedan responder a las solicitudes. La consistencia eventual es el modelo elegido, permitiendo que las actualizaciones sean rápidas en el nodo primario mientras las réplicas se sincronizan progresivamente. Las escrituras se aplican primero al nodo primario y luego se propagan a las réplicas, mientras que las lecturas acceden al nodo más cercano con una copia válida, optimizando la latencia.
La distribución de los datos en el anillo Chord se realiza de manera uniforme, aprovechando el hashing para asignar cada dato a un nodo específico. Esto no solo evita la sobrecarga en nodos individuales, sino que también facilita una búsqueda eficiente con una complejidad de O(log N), donde N es el número total de nodos en el sistema. Cuando un nodo se agrega o elimina, Chord redistribuye automáticamente los datos afectados, manteniendo un equilibrio dinámico.
La confiabilidad de las réplicas se refuerza mediante protocolos robustos de actualización y supervisión. Cada réplica envía confirmaciones al nodo primario después de recibir una actualización, asegurando que los cambios se apliquen correctamente. Además, los nodos sucesores realizan verificaciones periódicas para detectar y corregir inconsistencias, y se mantienen logs de operaciones recientes para facilitar la recuperación de datos en caso de fallo.

Tolerancia a Fallos

En cuanto a la tolerancia a fallos, el sistema está diseñado para resistir hasta fallos simultáneos de nodos, donde es el número de réplicas. Esto se logra gracias al uso de listas de sucesores extendidas, que permiten a los nodos mantener referencias a múltiples nodos vecinos. Si un nodo falla, estas referencias permiten redirigir las solicitudes a los nodos vivos más cercanos. Cuando un nodo se reincorpora o un nodo nuevo se une al sistema, se recalculan las réplicas y los datos se transfieren automáticamente, asegurando una integración fluida.
Para detectar y manejar errores, cada nodo supervisa periódicamente la conectividad con sus vecinos mediante mensajes de *ping*. Si un nodo no responde después de varios intentos, se considera fallido y los datos se redistribuyen automáticamente. Las operaciones interrumpidas se reintentan tras un breve retraso aleatorio, evitando la saturación de la red.
En conjunto, estas estrategias garantizan que el sistema sea altamente resiliente y confiable. La replicación adecuada, la supervisión constante y los mecanismos de recuperación automática aseguran que **Distnet** pueda operar eficientemente incluso en presencia de fallos parciales, ofreciendo una experiencia robusta y estable para los usuarios de esta red social distribuida.

Se implementarán las siguientes estrategias para mejorar la robustez del sistema:

1. Verificación de Integridad de Mensajes: Para detectar y descartar mensajes corruptos.
2. Envío Múltiple: Intentar conexiones con múltiples nodos para superar fallos transitorios.
3. Envío Persistente:
   - Relajado: Para actualizaciones no críticas.
   - Frecuente: Para actualizaciones críticas como la DHT del Chord.
4. Replicación de Datos: Para recuperarse de fallos permanentes de hardware.
5. Sugerencia de Componentes Activos: El proceso de "Stalking" para mantener una lista actualizada de nodos activos.
6. Tiempos de Espera: Para manejar conexiones rotas o nodos no respondientes.

Estas medidas tienen el objetivo de abordar los tres tipos principales de fallos: transitorios, permanentes e intermitentes, asegurando así la continuidad del servicio y la integridad de los datos.