Sistema de ficheros distribuido para AutoGOAL

Daniel Cárdenas Cabrera
Lia Zerquera Ferrer
Javier Oramas López
Facultad de Matemática y Computación, Universidad de La Habana, Cuba

19 de julio de 2023

1. Introducción

Este trabajo utiliza como punto de partida Kademlia[1]. Kademlia es una tabla de hash distribuida, que permite que millones de ordenadores se organicen automáticamente en una red, se comuniquen con otros ordenadores de la red y compartan recursos.
Se hace uso de varios enfoques e ideas propuestas por Kademlia, muchas de ellas se usan como base para ampliar las funcionalidades del proyecto y otras son utilizadas de la misma manera que Kademlia.

Una vez que se tiene un sistema de ficheros se integra a AutoGOAL para resolver problemas de aprendizaje de máquina de una manera más eficiente en cuanto a rendimiento, uso de la red y almacenamiento.

2. ¿Por qué Kademlia?

Una de las ventajas de Kademlia es la eficiencia de su algoritmo de enrutamiento, lo que permite una comunicación eficiente entre los nodos de la red. El prefijo binario de los ID de kademlia hace que se envíe la información hacia el nodo más cercano, minimizando la cantidad de saltos entre nodos y la latencia de la red en general.

Otra de las ventajas de este protocolo es la capacidad de manejar fallos de nodos y particionado de la red. El mecanismo de refrescado de Kademlia garantiza que se mantengan actualizadas las tablas de ruta de los nodos, manteniendo así la conectividad en la red y evitando la pérdida de datos dentro de lo posible.

Los datos pueden estar dispersos en múltiples nodos de la red. Kademlia utiliza una tabla hash distribuida para mantener un registro de la ubicación de los datos en la red. Esto permite que los nodos encuentren de manera eficiente la ubicación de los datos que necesitan procesar o analizar. Además, la arquitectura de Kademlia garantiza la tolerancia a fallos, lo que significa que si un nodo se

desconecta o falla, los datos todavía estarán disponibles en otros nodos de la red. Esta capacidad de almacenamiento y recuperación distribuida de datos de Kademlia es especialmente útil para los sistemas que suelen manejar grandes volúmenes de datos que necesitan ser procesados en paralelo. Al distribuir los datos entre varios nodos, se puede lograr un procesamiento y análisis más rápido y eficiente.

2.1. Ejemplos reales del uso de kademlia

Entre las compañías que utilizan este protocolo se encuentran:

- Storj [4]: Plataforma de almacenamiento en la nube que en su versión 3 utilizó una versión modificada del protocolo para la implementación de un sistema con capacidades similares a un DNS [3]
- Protocolo Ethereum [5]: El protocolo Ethereum utiliza una versión ligeramente modificada de kademlia, manteniendo el método de identificación con XOR y los K-buckets [3]
- The InterplanetaryFileSystem[6]: En la implementación de IPFS, el NodeID contiene un mapa directo hacia los hash de archivos de IPFS. Cada nodo también almacena información sobre dónde obtener el archivo o recurso.[3]

3. Descripción del sistema

Al igual que muchos sistemas de tabla de hash distribuida las llaves se almacenan en 160 bits, cada uno de los ordenadores participantes en la red tiene un ID y los pares (llave,valor) son almacenados en nodos con ID's çercanos", tomando como métrica de distancia la métrica XOR propuesta por Kademlia. Al igual que Kademlia se tratan los nodos como hojas de un árbol binario, en el que la posición de cada nodo viene determinada por el prefijo único más corto de su ID. Dichos nodos almacenan información de contacto entre sí para enrutar los mensajes de consultas

Se usa como base el protocolo de Kademlia que garantiza que cada nodo conoce al menos un nodo de cada uno de sus subárboles (si contiene algún nodo); con esta garantía cualquier nodo puede localizar a otro nodo por su ID.

Se cuenta con un módulo de persistencia llamado *PersistentStorage* que funciona de la siguiente manera:

Utiliza las rutas:

- *static/metadata* los nombres de los ficheros en esta ruta representan el hash de unos datos que se dividieron en varios chunks de tamaño máximo

1000kb, contienen como valor listas de python guardadas con pickle que tienen los hashes de cada chunk obtenido al picar unos datos determinados.

- *static/keys* los nombres de los ficheros en esta ruta representan todos los hashes de los datos almacenados, ya sea correspondiente a un dato completo o a un chunk de este, contienen como valor los hashes correspondientes en bytes.
- *static/values* los nombres de los ficheros en esta ruta representan los hashes de todos los chunks que se han almacenado, excluyendo hashes de datos sin picar.
- *timestamps* los nombres de los ficheros en esta ruta representan todos los hashes de los datos almacenados al igual que la ruta de keys pero contienen como valor un diccionario de python guardado con pickle que tiene como llaves *date* con el valor *datatime.now()*, *republish* con un valor booleano, y *last write* que es un *datetime* que representa la última vez q se se sobreescribioél archivo. Esto se usa para llevar un registro de la última vez que se accedió a 'una llave, para saber si es necesario que el nodo que la contiene la republique porque es accedida con frecuencia y en caso de partición de la red mantener el valor escrito mas reciente y no la ultima información accedida, de este modo garantizando consistencia eventual.

Cuando se recibe un par (clave, valor) en formato bytes, el módulo PersistentStorage codifica la clave utilizando $base64.urlsafe_b64encode$ para obtener un string que se puede utilizar como nombre de archivo. Luego, se escribe un archivo con ese nombre en las rutas de keys y values, donde se guarda la clave en bytes en el archivo de keys y el valor en bytes en el archivo de values. En el caso de que el par a almacenar sea metadata, el valor en bytes se escribe en la ruta de metadata. En ambos casos, también se crea un archivo en la ruta de timestamps con el nombre correspondiente.
Antes de escribir un valor o metadata, se crea un diccionario con las llaves:

- *integrity*: Se utiliza para saber si el fichero esta corrupto, en caso de ser así, no se devuelve nunca.
- *value*: Valor a escribir
- *integrity-date*: Momento en q se setea por primera vez la integridad. En caso de que pase un tiempo determinado desde esta fecha e integrity este en False, se borra automáticamente el fichero.
- *key-name*: String de la llave q representa un fichero, se utiliza para poder devolver los nombres originales de los ficheros si se ejecuta el comando *ls* en el cliente.
- *last-write*: Momento en el que se escribió por última vez el fichero. Se utiliza para manejar los casos en que luego de una partición, si se escribió en las dos particiones, valores con iguales llaves, al recuperarse la red, se mantengan los valores escritos más recientemente.

El protocolo de Kademlia contiene cuatro RPCs : PING , STORE , FIND-NODE y FIND-VALUE[2]. Esta propuesta además de usar estos cuatro RPCs, implementa CONTAINS, BOOTSTRAPPABLE-NEIGHBOR, GET, GET-FILE-CHUNKS, UPLOAD-FILE, SET-KEY, CHECK-IF-NEW-VALUE-EXIST, GET-FILE-CHUNK-VALUE, GET-FILE-CHUNK-LOCATION, FIND-CHUNK-LOCATION y FIND-NEIGHBORS

- *GET*: Obtiene la información identificada con la llave de un archivo, devuelve una lista con las llaves de cada chunk.
- *UPLOAD-FILE*: Sube el archivo al sistema de ficheros, lo divide en chunks y guarda la metadata del archivo con la forma de unificar todos los chunks.
- *GET-FIND-CHUNK-VALUE*: Devuelve el valor asociado al chunk.
- *GET-FIND-CHUNK-LOCATION*: Recibe la llave de un chunk y devuelve una tupla $(ip, puerto)$ donde se encuentra.
- *FIND-NEIGHBORS*: Devuelve los vecinos al nodo acorde a la cercanía según la métrica *XOR*.

Además, los servidores utilizan los siguientes RPCs:

- *CONTAINS*: Detecta si una llave está en un nodo, esto se utiliza tanto para la replicación de la información como para encontrar si un nodo tiene la información que se desea.
- *GET-FILE-CHUNKS*: Obtiene la lista de ubicaciones de los chunks de la información.
- *SET-KEY*: Guarda un par $(llave, valor)$ en el sistema.
- *DELETE* : Elimina un fichero de la red
- *GET-ALL-FILE-NAMES* : Realiza la función del comando *ls* en linux, devuelve una lista con todos los ficheros en el sistema(todas las llaves de metadata)

Se utilizó la misma estructura de tablas de enrutamiento que Kademlia, la tabla de enrutamiento es un árbol binario cuyas hojas son k-buckets. Cada k-bucket contiene nodos con algún prefijo común en sus ID. El prefijo es la posición del k-bucket en el árbol binario, por lo que cada k-bucket cubre una parte del espacio de ID y, juntos, los k-buckets cubren todo el espacio de ID de 160 bits sin solaparse. Los nodos del árbol de enrutamiento se asignan dinámicamente, según sea necesario.

Para garantizar la correcta replicación de los datos, los nodos deben volver a publicar periódicamente las llaves. De lo contrario, dos fenómenos pueden hacer que fallen las búsquedas de llaves válidas. En primer lugar, algunos de los k

nodos que obtienen inicialmente un par llave-valor cuando se publica pueden abandonar la red. En segundo lugar, pueden unirse a la red nuevos nodos con ID más cercanos a alguna llave publicada que los nodos en los que se publicó originalmente el par llave-valor. En ambos casos, los nodos con el par llave-valor deben volver a publicar para asegurarse de que está disponible en los k nodos más cercanos a la llave.

Una vez que un cliente pide al sistema un determinado valor, se le devuelve una lista de las ubicaciones de los distintos chunks de datos (no necesariamente estos se encuentran en la misma PC) y se crea una conexión con la PC más cercana a la información de forma tal que la red no se sature innecesariamente. Luego el cliente unifica los datos y devuelve el valor almacenado. Una vez que un nodo envía cierta información, marca ese archivo como pendiente de republicar y actualiza su timestamp, de manera que se informa a cada uno de los vecinos en los que se debe replicar la información que esta fue accedida y no ha de ser eliminada.

Para algoritmo de republicación implementado se utiliza un Thread que se ejecuta cada un intervalo de tiempo i y consiste en lo siguiente:

- Recorrer todas las llaves en el storage cuyo timestamp sea mayor que un tiempo t y aquellas que tengan republish en True son republicadas por el nodo.
- Recorrer todas las llaves en el storage y verificar en la red cuántas réplicas puede encontrar de cada una, aquellas llaves que estén por debajo del factor de replicación especificado son republicadas por el nodo.Aquellas que estén por encima, se borran de los nodos más lejanos hasta igualar el factor de replicación especificado.

Cuando un servidor descubre un nuevo nodo, para cada clave almacenada, el sistema recupera los k nodos más cercanos. Si el nuevo nodo está más cerca que el nodo más alejado de esa lista, y el nodo para este servidor está más cerca que el nodo más cercano de esa lista, entonces el par clave/valor se almacena en el nuevo nodo.
Cuando un servidor inicia una conexión nueva con un cliente, inicia un Thread nuevo para manejar dicha conexión.
Cuando el cliente comienza a subir un archivo, los datos se dividen en chunks de máximo 1000kb y se almacenan en el sistema de ficheros con integrity en False. Luego, se almacena la metadata del archivo, que contiene la información necesaria para unificar los chunks y reconstruir el archivo original también con integrity en False. Una vez que se confirma que se pudieron almacenar todos los chunks y la metadata, comienza a confirmarse la integridad de cada chunk en la red y después la de la metadata. En caso de ocurrir algún error en cualquier parte del proceso, se borra todo lo almacenado, haciendo el efecto de un rollback.

4. Tolerancia a Fallas

Los nodos mantienen información sobre otros nodos cercanos en la red, almacenando estos nodos en su tabla de enrutamiento. Cada nodo tiene una lista de "k-buckets"que contienen los detalles de contacto de otros nodos en la red, clasificados según su proximidad en el espacio de identificación.

Cuando un nodo deja de responder o se desconecta, los otros nodos de la red detectan la falta de respuesta después de un período de tiempo determinado. En ese momento, se considera que el nodo fallido está inactivo.

Cuando un nodo detecta la inactividad de otro nodo, actualiza su tabla de enrutamiento eliminando la entrada del nodo fallido. Además, el nodo realiza una serie de acciones para mantener la conectividad y la redundancia en la red. Estas acciones incluyen:

- **Replicación de datos**: Si el nodo fallido almacenaba datos, otros nodos de la red pueden tomar la responsabilidad de mantener réplicas de esos datos. De esta manera, los datos permanecen disponibles incluso si el nodo original se desconecta. Los nodos que asumirán esta responsabilidad serán los que estén más cercanos utilizando la misma métrica de XOR con los IDs para mantener la red lo más optimizada posible.
- **Actualización de información de enrutamiento**: Los nodos que tenían al nodo fallido en su tabla de enrutamiento actualizan esa entrada eliminándola. De esta manera, los nodos evitan enviar mensajes o realizar acciones hacia un nodo que ya no está disponible.
- La expiración de contacto garantiza que la información almacenada en la red permanezca accesible incluso cuando los nodos individuales fallan. Al eliminar los nodos inactivos de los buckets, se asegura de que las rutas de enrutamiento se actualicen y se mantengan eficientes.

La expiración de contacto garantiza que la información almacenada en la red permanezca accesible incluso cuando los nodos individuales fallan. Al eliminar los nodos inactivos de los buckets, se asegura de que las rutas de enrutamiento se actualicen y se mantengan eficientes.

5. Autodescubrimiento

Para el mecanismo de auto descubrimiento se crea también un Thread que emite un latido en las direcciones de broadcast de cada una de las NICs del host, con un identificador de la fuente del broadcast e ip y puerto *"dfs ip puerto"*. Cada vez que un nuevo nodo entra a la red escucha los broadcast para encontrar vecinos. El mecanismo es accesible desde el cliente, haciendo transparente la conexión para el usuario.

5.1. Desventajas de este enfoque

Como se utiliza broadcast para el autodescubrimiento esto solo es posible en una red local, o utilizando una (o varias) PC como puente entre distintas subredes, haciendo que si por alguna razón no existiese una forma de conectar localmente las computadoras, estas no serán capaces de encontrarse, no obstante como el sistema está pensado para conexión de distintas estaciones de trabajo se asume que estas estarán en la misma red local haciendo factible este enfoque.

6. Análisis del teorema CAP

El teorema CAP, también conocido como el teorema de Brewer, es un concepto fundamental en los sistemas distribuidos que establece que es imposible que un almacén de datos distribuido proporcione simultáneamente consistencia (C), disponibilidad (A) y tolerancia a particiones (P).

Consistencia (C): La consistencia se refiere a que todos los nodos en un sistema distribuido tengan la misma visión de los datos al mismo tiempo. En este sistema, lograr una consistencia estricta en todos los nodos no es una prioridad. En cambio, se garantiza en la consistencia eventual, lo que significa que con el tiempo, todos los nodos convergerán al mismo estado. Los nodos intercambian periódicamente información y actualizan sus tablas de enrutamiento para lograr esta convergencia.

Disponibilidad (A): La disponibilidad implica que el sistema permanezca receptivo y accesible para los usuarios incluso en presencia de fallos de nodos o particiones de red. EL sistema prioriza la disponibilidad asegurando que los nodos puedan seguir operando y proporcionando servicios incluso cuando algunos nodos estén no disponibles o inalcanzables. Esto se logra a través de la redundancia y la replicación de datos, donde se almacenan múltiples copias de datos en diferentes nodos.

Tolerancia a particiones (P): La tolerancia a particiones se refiere a la capacidad del sistema para continuar funcionando y proporcionar servicios a pesar de particiones o fallos en la red. El sistema está diseñado para ser tolerante a particiones, lo que permite que la red siga operando y mantenga su funcionalidad incluso cuando los nodos estén temporalmente desconectados o aislados debido a problemas de red.

Todos estos puntos se analizan asumiendo que no se sobrepasa la capacidad de recuperación del sistema, si se superase dicho punto el sistema comenzará a ver afectada la disponibilidad. Pero esto solo Ocurriría en caso de fallos simultáneos, puesto que el sistema es capaz de recuperarse de fallos eventuales sin ningún problema.

7. Cliente

Se desarrolló un cliente que cuenta con las siguientes características:

- Puede recibir puntos de entrada a la red, conocidos como *bootstrap nodes* para una conexión directa con el sistema de ficheros
- Posee un mecanismo de auto-descubrimiento recibiendo broadcast por todas las NICs de la pc, si no recibe ningún *bootstrap node* o no es capaz de conectar con ninguno puede usar esta funcionalidad para encontrar algún nodo automáticamente.
- Tiene la capacidad de al conectar con algún nodo descubir otros nodos a partir de los vecinos de este de manera automática.
- Maneja errores relacionados con inestabilidad de la red o caídas inesperadas de un nodo, permitiendo que el usuario establezca el número de reintentos que se debe hacer cuando se pierde la conexión con un nodo, posee un cola con los nodos conocidos que al agotarse el número de reintentos de conexión con un nodo, este es removido de la cola y se pasa al siguiente, es posible especificar también si se quiere que se utilice el mecanismo de auto-descubribimiento al quedarse la cola vacía.

8. Recomendaciones

- Dada la caída de un nodo de la red, mantener de alguna manera el id del nodo anterior para si su información se mantuviese aún en disco y actualizada cargar con este id puede ser más eficiente que iniciar como un nodo nuevo y comanzar nuevamente el proceso de balanceo de la red.
- En la implementación original de kademlia, la comunicación entre nodos se realiza mediante UDP. Dada la limitación de UDP para manejar grandes cantidades de información se cambió a TCP, pero esta es menos eficiente para las tareas que no sean transferencia de datos de almacenamiento. Implementar un doble protocolo de comunicación entre los nodos debería representar una mejoría en el rendimiento de la red.
- Cambiar el algoritmo de hash utilizado de sha1 a sha256 dado q sha1 ya no es considerado seguro y sus vulnerabilidades hacen mas fácil a los atacantes realizar acciones maliciosas
- Implementar un mecanismo de testing automático distribuido utilizando técnicas como chaos testing y swarm testing
- Implementar un sistema de autenticación para que a los servidores solo se puedan conectar clientes autorizados y estos solamente tengan acceso a los RPC necesarios

Bibliografía

[1] Kademlia : A peer-to-peer Information System based on de XOR metric

[2] Kademlia : A peer-to-peer Information System based on de XOR metric section 2.3

[3] https://www.storj.io/blog/a-brief-overview-of-kademlia-and-its-use-in-various-decentralized-platforms

[4] https://www.storj.io/

[5] https://ethereum.org/en/

[6] https://ipfs.tech/