Distributed Tag-Base System

Descripción

El Sistema de Ficheros con Etiquetas Distribuido es un sistema distribuido que permite a los usuarios almacenar y recuperar archivos utilizando etiquetas. Está diseñado para ser escalable y tolerante a fallos. El sistema está construido utilizando una arquitectura peer-to-peer, donde cada nodo en la red actúa como cliente y servidor. Esto permite un sistema descentralizado que puede manejar grandes cantidades de datos y tráfico. El sistema está escrito en `Python`.

Instalación

Paquetes de `Python` utilizados

- `pickle`
- `umgspack`
- `rpyc`
- `asyncio`

Protocolo Kademlia

El sistema utiliza el protocolo `Kademlia` para comunicación peer-to-peer y almacenamiento de la tabla hash distribuida (`DHT`).

El protocolo `Kademlia` es un protocolo de tabla hash distribuida que permite a los nodos en una red localizar y recuperar datos de manera eficiente. Utiliza una estructura de árbol binario para organizar los nodos en la red, donde cada nodo es responsable de un subconjunto del espacio de ficheros. Los nodos se comunican entre sí para compartir información sobre las llaves de las que son responsables, lo que permite el enrutamiento y búsqueda eficientes de datos.

En el Sistema de Etiquetas Distribuido, cada nodo en la red utiliza el protocolo Kademlia para almacenar y recuperar archivos y sus etiquetas asociadas. Cuando se carga un archivo en la red, se genera una clave hash, que luego se utiliza para determinar qué nodo en la red es responsable de almacenar el archivo. El nodo almacena el archivo y sus etiquetas asociadas en su almacenamiento local.

Cuando un usuario busca archivos según etiquetas, el cliente envía una solicitud a uno de los nodos en la red para buscar archivos con la etiqueta especificada. El nodo luego utiliza el

protocolo `Kademlia` para localizar los nodos que son responsables de las claves asociadas con la etiqueta. El nodo luego envía solicitudes a estos nodos para recuperar los archivos y sus etiquetas asociadas. Una vez que se han recuperado los archivos y las etiquetas, el nodo los envía al cliente.

Tolerancia a fallos

El protocolo `Kademlia` asegura la tolerancia a fallos al permitir que los nodos se unan y abandonen la red de manera dinámica sin afectar el sistema en general.

Cuando un nodo se une a la red, contacta a otros nodos en la red para determinar su posición en la estructura de árbol binario. Luego, el nodo almacena información sobre los otros nodos que ha contactado en su almacenamiento local. Esta información incluye los IDs de nodo, las direcciones IP y los números de puerto de los otros nodos.

Si un nodo falla o se vuelve inaccesible, los otros nodos en la red aún pueden localizar y recuperar datos utilizando el protocolo `Kademlia` para enrutar alrededor del nodo fallido. Cuando un nodo necesita localizar una clave que no está en su almacenamiento local, envía una solicitud al nodo que está más cerca de la clave de los que conoce. Si ese nodo no tiene la información que se está buscando envía la solicitud al nodo más cercano a la clave que él conoce, y así sucesivamente, hasta que se encuentra la clave.

La tolerancia a fallos se logra mediante la replicación de archivos y sus etiquetas asociadas en varios nodos en la red. Esto asegura que si un nodo falla o se vuelve inaccesible, los archivos y etiquetas aún pueden ser recuperados de otros nodos en la red.

Arquitectura

El sistema consta de los siguientes componentes:

- Nodos: Cada nodo en la red es responsable de almacenar y brindar archivos. Los nodos se comunican entre sí para compartir información sobre los archivos que tienen y las etiquetas asociadas con ellos.
- Cliente: El cliente es responsable de interactuar con el usuario y enviar solicitudes a los nodos en la red. El cliente puede cargar, descargar, eliminar y buscar archivos según etiquetas en la red.
- Parser de comandos: Es responsable de analizar la entrada del usuario y generar un comando que puede ser ejecutado por el cliente.

Estructura general

El sistema utiliza un cliente-servidor bidireccional que utiliza RPyC y TCP para enviar archivos.

Cada nodo que se une a la red de intercambio de archivos arranca el cliente-servidor bidireccional, lanzando tanto un servidor como un cliente, este último solo cuando es necesario. El usuario solicita o publica un archivo a través del cliente. Para publicar un archivo en la red, el cliente utiliza la API de Kademlia. Después de la publicación, los nodos conectados a la red Kademlia pueden solicitar a Kademlia el/los nodo(s) que alojan el archivo deseado.

Para solicitar la ubicación de un archivo deseado, el cliente utiliza la API de Kademlia para obtener el archivo especificando su etiqueta. El cliente recibirá la dirección IP y el puerto de otro servidor en la red que tenga ese valor. Una vez que el cliente recibe el par de dirección IP y puerto adecuado para descargar el archivo deseado, se comunica con el servidor al que pertenece ese par. En el caso de que varios nodos alojen el mismo archivo, el cliente recibirá una lista de pares de compañeros para contactar en caso de que uno falle.

Kademlia está estructurado para tener su propio protocolo que consta de un conjunto de instrucciones RPC sobre UDP.

Consistencia en la Red

El sistema primero transmite los archivos a través de TCP hacia los nodos de almacenamiento destino. Luego, los archivos se almacenan en un archivo JSON para garantizar su persistencia en caso de que el nodo o la red se reinicie no se pierda información. Finalmente, se actualiza la ubicación de los archivos en los nodos del protocolo Kademlia para que se pueda consultar qué archivos están disponibles y en qué nodos se encuentran.

De esta manera si la operación de transmitir el archivo, que es la más tardía, falla nunca la red crea que contiene dicho archivo en el nodo que falló. Para mantener la consistencia entre el almacenamiento, el estado del json y los nodos de la red se implementó un recolector de basura.

El recollector de basura funciona de la siguiente manera:

- Se revisa cada archivo en el directorio "secure" y se verifica si está almacenado en el nodo.
- Si el archivo no está en el nodo, se elimina del sistema de archivos y se elimina su entrada en el state.jon.
- Si el archivo está en el diccionario, se actualiza la lista de etiquetas en el state.json con la lista de etiquetas almacenadas en nodo.

Flujo de datos:

- Los archivos se cargan en los nodos de la red y se almacenan de manera distribuida.
- Las etiquetas se asocian con los archivos y se almacenan de manera distribuida.
- Cuando un usuario busca archivos según etiquetas, el cliente envía una solicitud a uno de los nodos en la red para buscar archivos con la etiqueta especificada. El nodo luego devuelve una lista de archivos que coinciden con la etiqueta.
- Cuando un usuario descarga un archivo, el cliente envía una solicitud al nodo que tiene el archivo. El nodo luego envía el archivo al cliente.

Almacenamiento de datos

Cuando un cliente desea subir un archivo a la red, primero se calcula el hash del archivo, que se forma a partir del nombre del archivo y su contenido. Esta técnica proporciona flexibilidad al sistema de ficheros al garantizar que los archivos con nombres diferentes pero contenido idéntico tengan el mismo hash. A continuación, se lleva el hash de cada etiqueta asociada al archivo al espacio de identificadores de Kademlia. El sistema busca los nodos más cercanos en este espacio de identificadores (a lo sumo k nodos), que se consideran propietarios de la etiqueta, y se les envía tanto el archivo como las etiquetas. Los nodos almacenan los archivos y las etiquetas asociadas a ellos. Las etiquetas se almacenan en un diccionario donde se guardan las etiquetas y el conjunto de identificadores de los archivos asociados a cada una. Por otro lado, los archivos se almacenan en el sistema de archivos local del nodo y, a nivel de la red distribuida, se almacenan en un diccionario que tiene como llave el identificador del archivo y como valor una tupla que almacena las etiquetas asociadas al archivo, el hash de su contenido y su nombre.

Búsqueda de datos

Para buscar los valores asociados a una etiqueta en la red, el cliente comienza llevando al espacio de identificadores de Kademlia la etiqueta deseada y buscando los nodos más cercanos en la red. A partir de estos nodos, se explora la red utilizando un sistema de búsqueda en profundidad para encontrar los valores asociados a la etiqueta. Cuando se encuentra un valor asociado a la etiqueta, se almacena en una lista de valores encontrados. Después de explorar la red, se devuelve la lista de valores encontrados. Este proceso se puede extender para la búsqueda de valores asociados a una consulta compleja de etiquetas. En este caso, se buscan los valores asociados a cada una de las etiquetas de la consulta y luego se combinan según los operadores lógicos de la consulta.

Para buscar valores en la red se implementó la clase ValueSpiderCrawl. La clase ValueSpiderCrawl tiene un método llamado find() que inicia la búsqueda de un valor. La búsqueda implica contactar a los nodos en la red y pedirles el valor asociado con una cierta clave. La búsqueda se realiza de manera en profundidad, comenzando desde los nodos más cercanos a la clave y expandiéndose hacia afuera. La clase ValueSpiderCrawl lleva un registro de

los nodos que han sido contactados y los nodos que aún no han sido contactados. Si se contacta a un nodo y tiene el valor asociado con la clave, el valor se almacena en una lista llamada found_values. El nodo que tenía el valor también se almacena en una lista llamada found_nodes. Si se contacta a un nodo pero no tiene el valor, entonces los nodos más cercanos a ese nodo se añaden a la lista de nodos por visitar. Una vez que se han contactado todos los nodos, se analiza la lista found_values para determinar qué valor está asociado con la clave. Si no se encuentra ningún valor, la búsqueda se expande al siguiente conjunto de nodos. En general, la clase ValueSpiderCrawl proporciona una manera de buscar valores en una red Kademlia contactando a los nodos de manera en profundidad y manteniendo un registro de los nodos que han sido contactados y los valores asociados con la clave.

Para la búsqueda de nodos en la red se implementó la clase NodeSpiderCrawl, que realiza un proceso similar, pero en lugar de almacenra los valores encontrados, almacena los nodos encontrados.

Eliminación de datos

Para la eliminación de datos se realiza un proceso similar al descrito anterirmente de búsqueda de los datos en la red y de los nodos que los contiene. Se eliminan además los archivos de las etiquetas que tenían asociados para garantizar que se mantenga la consistencia en la información.

Lenguaje de Comandos

El sistema utiliza un lenguaje de comandos para que el usuario interactúe con el sistema. El lenguaje de comandos consta de un conjunto de instrucciones que el usuario puede utilizar para cargar, descargar, eliminar y buscar archivos según etiquetas en la red.

Se ha implementado un `DSL` utilizando la biblioteca `PLY` de `Python` (`Python` `Lex-Yacc`), que es una implementación de las herramientas `Lex` y `Yacc` para `Python` . `PLY` se utiliza para definir la gramática del lenguaje y generar un analizador sintáctico que puede analizar la entrada del usuario y generar un árbol de sintaxis abstracta (`AST`) que representa el comando.

El archivo `instruction_parser/ply_parser.py` define la gramática del lenguaje utilizando la sintaxis de `PLY` . La gramática se define en términos de reglas de producción que especifican cómo se deben combinar los tokens para formar comandos válidos. Este también define previamente el analizador léxico que se utiliza para dividir la entrada del usuario en tokens. El analizador léxico utiliza expresiones regulares para identificar los diferentes tipos de tokens en la entrada. Luego, se pasa la lista de tokens al analizador sintáctico, que utiliza la gramática definida para generar un `AST` que representa el comando.

El `AST` se utiliza para generar un objeto `Command`, que tiene dos atributos: `name` y `args`. `name` es una cadena que representa el nombre del comando (por ejemplo, "add", "delete", "list", etc.), y `args` es un diccionario que contiene los argumentos del comando.

Gramática

- S' -> input
- input -> instruction
- instruction -> add_inst
- instruction -> query_inst
- instruction -> add_tags_inst
- instruction -> delete_tags_inst
- add_inst -> ADD F file_list T tag_list
- query_inst -> inst Q tag_query_or_star
- add_tags_inst -> ADD_TAGS Q tag_query_or_star T tag_list
- delete_tags_inst -> DELETE_TAGS Q tag_query_or_star T tag_list
- inst -> DELETE
- inst -> LIST
- inst -> GET
- file_list -> FILENAME
- file_list -> FILENAME file_list
- tag_list -> WORD
- tag_list -> WORD tag_list
- tag_query_or_star -> STAR
- tag_query_or_star -> tag_query
- tag_query -> tag_query AND basic_tag_query
- tag_query -> tag_query OR basic_tag_query
- tag_query -> basic_tag_query
- basic_tag_query -> NOT basic_tag_query
- basic_tag_query -> WORD
- basic_tag_query -> LPAREN tag_query RPAREN

Instrucciones

add -f file-list -t tag-list

Copia uno o más ficheros hacia el sistema y estos son inscritos con las etiquetas contenidas en tag-list.

delete -q tag-query

Elimina todos los ficheros que cumplan con la consulta tag-query.

list -q tag-query

Lista el nombre y las etiquetas de todos los ficheros que cumplan con la consulta tag-query.

add-tags -q tag-query -t tag-list

Añade las etiquetas contenidas en tag-list a todos los ficheros que cumpan con la consulta tag-query.

delete-tags -q tag-query -t tag-list

Elimina las etiquetas contenidas en tag-list de todos los ficheros que cumplan con la consulta tag-query.

Funcionalidades adicionales

get -q tag-query

Descarga todos los ficheros que cumplan con la consulta tag-query. Los ficheros serán almacenados en la carpeta 'secure' del proyecto.

Consultas complejas de etiquetas

En la gramática, una consulta `tag_query` se define como una expresión booleana que combina etiquetas y operadores lógicos. Las etiquetas se representan como cadenas de texto, y los operadores lógicos se representan como caracteres clave (`and` : `&` , `or` : `||` , `not` : `~`) que se utilizan para combinar las etiquetas. También se utiliza como símbolo para agrupar los paréntesis.

La gramática permite la combinación de etiquetas utilizando los siguientes operadores lógicos:

`&` : combina dos etiquetas y devuelve los archivos que contienen ambas etiquetas. `||` : combina dos etiquetas y devuelve los archivos que contienen al menos una de las etiquetas. `~` : niega una etiqueta y devuelve los archivos que no la contienen.

Las consultas `tag_query` pueden ser simples o complejas, y pueden incluir cualquier número de etiquetas y operadores lógicos. Por ejemplo, una consulta simple podría ser `tag1` , que devuelve todos los archivos que contienen la etiqueta `tag1` . Una consulta más compleja podría ser `(`

`tag1 & tag2 ) || ~ tag3`, que devuelve todos los archivos que contienen las etiquetas `tag1` y `tag2`, o que no contienen la etiqueta `tag3`.

En resumen, las consultas `tag_query` de la gramática utilizada en `ply_parser.py` son expresiones booleanas que combinan etiquetas y operadores lógicos para buscar archivos que contengan ciertas etiquetas.