Proyecto Final: Implementación del protocolo Chord

Materia: Telecomunicaciones y Sistemas Distribuidos (1968)

Profesores:

- Arroyo Marcelo Daniel
- Scilingo Gastón Maximiliano
- Tardivio María Laura

Integrantes:

- Gardiola Joaquin, DNI: 38418091
- Giachero Ezequiel Ignacio, DNI: 39737931

Descripción del proyecto

El proyecto consiste en una implementación del algoritmo Chord que fue brindado en pseudocódigo. El lenguaje de programación que elegimos fue Python, por su sencillez a la hora de crear y configurar sockets para conexiones entre distintas computadoras (reales o virtuales). El protocolo de transporte utilizado es TCP, que es el predeterminado de Python, sin embargo, por nuestra arquitectura de proyecto podría cambiarse por UDP.

También se extendió el algoritmo para realizar caching de consultas con un tiempo de vida determinado y replicación de la información en distintos nodos.

Las consultas son iniciadas por un nodo que se contacta a cualquier otro nodo del anillo. Este nodo es quien responde al primero con la información solicitada.

Estructura del proyecto

El proyecto consta de 5 archivos, donde cada uno se encarga de una funcionalidad en particular y solo uno es el encargado de comenzar la ejecución del nodo.

La estructura de los archivos es la siguiente:

```
Proyecto/
    ├── AuxFunctions.py
    ├── Menu.py
    ├── Node.py
    ├── README.md
    ├── SocketManager.py
```

El contenedor principal “Node.py” se encarga de la definición de la clase Node con todas sus funciones, que se encuentran especificadas en el protocolo y hace uso de los demás archivos para la implementación de las mismas. Además, la función *main* está definida en este.

Dentro de “AuxFunctions.py” se encuentran definidas funciones auxiliares como por ejemplo funciones para la conversión de diccionarios y nodos, obtener hashes a partir de datos, etc.

En “Menu.py” se describen las distintas opciones que un nodo puede realizar y se encarga de tomar los datos necesarios por consola para llevar a cabo la ejecución de las mismas.

El archivo “SocketManager.py” define una clase auxiliar para ayudar al manejo de sockets y facilitar el envío y recepción de información. También se encarga de codificar y decodificar los datos antes de enviarlos o recibirlos.

Inicialización del nodo

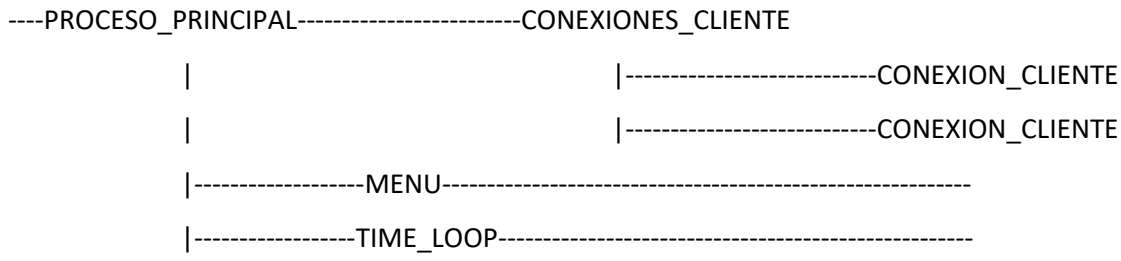
Un nodo es una clase en Python que contiene los atributos *id, ip, port* necesarios para identificar y conectar los nodos. Además, contiene la información de su nodo predecesor y sucesor a través de los atributos *pred* y *succ*. Dentro del atributo *finger_table* se almacena la información de los n sucesores. El atributo *hash_table* contiene los datos (*key, value*) almacenados en el nodo y en *cache* se guarda una información temporal de los datos consultados por el nodo.

Además, en la creación del nodo se inicializa el contador de reinicio de *cache*, la *finger_table* y se trata de crear un socket (asociado a la ip y puerto) en el que se van a conectar aquellos nodos que requieran datos locales.

Arquitectura del nodo

Luego de la inicialización se define la función *start*, que al ser invocada se crean los distintos threads para la ejecución del nodo. Esta función es invocada dentro de *main*.

Existen varios threads, el primero que se crea es el encargado de mostrar el menú y recibir las opciones elegidas por el usuario. El segundo thread se encarga de realizar las operaciones periódicas para la estabilización y limpieza de la cache del nodo. El proceso principal se queda esperando recibir conexiones entrantes y cada vez que recibe una, crea un thread dedicado a manejar el cliente que se conectó. De esta manera puede manejarse más de un nodo cliente conectado simultáneamente.



Manejo de mensajes

Todas las operaciones que se desean realizar en un nodo remoto, deberán crear un socket hacia ese y mandar un mensaje por este canal que contenga el nombre de la operación y los datos necesarios para realizar la misma. El nodo remoto recibirá este mensaje (*handle_client*), comprobará que operación ejecutar, la invocará con los parámetros pasados y devolverá el resultado dependiendo de si es necesario o no.

Para la codificación de los mensajes usamos la librería *json*. Internamente Python maneja los *json* como diccionarios, por tal motivo a la hora de procesar un mensaje hay que hacer ciertas transformaciones para obtener la información del mismo.

Estructura de replicación de nodos

Nosotros decidimos que la información de un nodo se encuentre replicada en los n nodos sucesores, sin embargo, para simplificar la complejidad elegimos n = 1. Para lograr esto, cada vez que se invoca la función *set* de la hash table la información se almacena no solo en el nodo correspondiente sino también en su sucesor.

De esta manera, si el nodo con datos replicados se desconecta la información queda almacenada en el nodo inicial, y si el nodo con la información original se desconecta los datos quedan en su sucesor. Esto asegura que no haya perdida de información y que se mantenga el invariante (*hash(datos.key) < nodo.id*).

Cada vez que un nodo entra (*join*) o sale (*leave*), se transmite la información de la réplica al nuevo nodo correspondiente.

Estructura de la cache

La caché se encuentra almacenada dentro de un atributo de la clase nodo y se definen distintas funciones para el manejo de la misma.

Cada vez que se realiza una consulta, el nodo que la realizó se fija si ya tiene los datos en la cache, de ser así, los retorna. En caso contrario busca la información en el nodo correspondiente y antes de retornarla la guarda en su cache.

Periódicamente (dentro de *time_loop*) se decrementa un contador y cuando este llega a cero se elimina el dato más antiguo de la cache y se vuelve a resetear el contador. De esta forma, se produce un ciclo que va limpiando la cache continuamente.