Proyecto Final de Sistemas Distribuidos

July 3, 2022

Integrantes:
Alben Luis Urquiza Rojas
Frank Abel Blanco Gómez
Karel Díaz Vergara

1 Buscador Distribuido

1.1 Arquitectura

El sistema tiene dos tipos de nodos:

1- **Client**: Es el encargado de hacer las peticiones que son de dos tipos: la primera una búsqueda a partir de un string para obtener posibles resultados de documentos que puedan ser relevantes, y una segunda que a partir de un identificador de un documento que puede ser relevante solicitarle al server que lo reportó que se lo envíe.

2- **Server**: Es el encargado de recibir las peticiones de los clientes y responder los resultados de vuelta al cliente.

1.2 Descubrimiento de la red

Para descubrir la red, cada servidor en la red hace ping cada cierto tiempo mediante grupos multicast enviando el string "s", luego los clientes que también están en la red reciben el ping y guardan en una lista de "servidores activos" la dirección IP y el tiempo del último ping. Para esta funcionalidad utilizamos la biblioteca **socket** de python y el protocolo UDP. Si algún

servidor se desconecta de la red, los clientes dejarán de recibir respuesta del ping, y luego de que pase un tiempo predeterminado del último ping que se hizo desde este servidor, este se elimina de la lista de servidores activos.

1.3 Comunicación entre el cliente y el servidor

Cuando un cliente realiza una petición del primer tipo, la envía a todos los servidores que tiene en su lista de servidores activos, espera por la respuesta de cada uno de estos y muestra los resultados. El mensaje enviado es simplemente un string *query* con el que el servidor realizará la búsqueda. La repuesta del servidor será entonces una lista con los posibles documentos relevantes, cada uno tiene un *ranking*, un *id* y una *descripción del texto*. Una vez recibidas todas las listas de cada uno de los servidores, las unimos todas en una misma lista y las ordenamos según el *ranking*.

Para las peticiones del segundo tipo solo se le envía al servidor que respondió con el documento deseado, en este caso, lo que se envía es el **id** con el cual el servidor puede identificar el documento. La respuesta de este es el documento completo por lo que el cliente podrá guardarlo luego.

Para establecer esta comunicación utilizamos la biblioteca **zmq** de python que que tiene una clase **socket** también y puede establecer la comunicación mediante el protocolo TCP. Por cada server se crea un hilo por el cual se establece el intercambio de información. En el cliente se crea un **zmq.socket** de tipo REQ (request) y en el lado del servidor otro de tipo REP (reply). El protocolo TCP garantiza que (si el servidor está activo) llegue la información que se transmite por el socket, además el tipo de socket REQ se queda esperando hasta que recibe una respuesta, y el socket REP siempre responde de vuelta al socket que le hizo una petición.

1.4 Tolerancia a fallas

Para tener una "buena tolerancia a fallas" intentamos hacer cada nodo lo más independiente posible. Es decir, los servidores son todos independientes por lo que todos tienen la misma función dentro de la red, si uno cae, el resto de ellos sigue funcionando sin ningún tipo de problema.

Otro tipo de inconveniente sería si un servidor cae y formaba parte de los servidores activos de un cliente. En este caso los clientes que tenían a este

servidor dejarán de recibir el ping que este hacía y después de un determinado tiempo sin recibirlo serán eliminados de los servidores activos.

También si un servidor cae en el momento en que un cliente está haciéndole una petición, después de un tiempo predeterminado el cliente dejará de esperar su respuesta y asumirá que el servidor se desconectó de la red y el resto de la red seguirá funcionando correctamente.

Por otra parte tuvimos en cuenta que podía ser posible que un cliente podría recibir el mismo documento de varios servidores, para ello la descripción de los documentos que recibe el cliente se *hashea* y se guarda en un diccionario donde la clave será el resultado de esta operación, luego al procesar otro documento si el hash es una clave del diccionario entonces asumimos que el documento es el mismo. De esta manera evitamos tener documentos repetidos (o sea, con el mismo hash) en la unión de los documentos devueltos de cada servidor, un posible inconveniente de este mecanismo es que puede haber documentos distintos con el mismo hash (colisiones), sin embargo es muy poco probable que ocurra una colisión.

1.5 Correr el proyecto

Para correr el proyecto primeramente tenemos que crear la imagen de Docker, para ello utilizamos el comando:

```
docker build -t distributed .
```

Luego solo quedaría correr el comando:

```
docker-compose up -d
```

De esta manera se iniciarán tres contenedores, correspondientes a tres computadoras, cada una con un cliente y un servidor.

Para crear un nuevo cliente y un nuevo servidor ejecutamos respectivamente los comandos:

```
docker run --network dis distributed python run_client.py --ip
<ip>
docker run --network dis distributed python run_server.py --ip
<ip> --collection <collection_name>
```

Para hcer una petición, necesitamos entrar en un contenedor que esté corriendo, ejecutando:

```
docker exec −it <container_name> sh
```

Luego para hacer la petición de primer tipo:

```
curl http://127.0.0.1:8000/query?value=<query_name>
```

La petición de segundo tipo:

```
curl http://127.0.0.1:8000/document/<document_id>
```

Para detener un servidor/cliente:

```
docker stop <container_name>
```