Ejercicio Evaluable 2 - Comunicación por Sockets

Adrián Martínez Cruz 100451213

César López Navarro 100451326

Grupo 86

1. Introducción

En este proyecto se ha desarrollado un sistema distribuido que involucra un servidor y un cliente interconectados mediante sockets TCP. Esto implica que ambas partes pueden comunicarse simultáneamente. Los clientes realizan solicitudes al servidor, que las recibe y las procesa de manera concurrente utilizando hilos y mutex. Para asegurar el funcionamiento adecuado de la aplicación, es crucial que el servidor y el cliente utilicen el mismo puerto para la comunicación a través de los sockets.

2. Comunicación entre servidor y cliente - Sockets TCP

Como se ha mencionado previamente, la comunicación entre las partes se realizará mediante sockets TCP en C. El servidor estará a la espera de conexiones continuamente, mientras que los clientes establecerán conexiones para enviar sus solicitudes.

Servidor

El servidor, en su función main, ejecutará un bucle while(true) para escuchar constantemente las conexiones entrantes de los clientes. Para ello, creará un socket de tipo STREAM (TCP) y de la familia ipv4, que escuchará en el puerto especificado al ejecutar el programa. Se utilizará INADDR_ANY para permitir la escucha en todas las direcciones del puerto, permitiendo así que cualquier cliente se conecte.

Una vez establecidas las conexiones, se inicializan las variables necesarias para procesar las solicitudes de manera concurrente, incluyendo hilos, mutex, malloc y variables condicionales. Estos hilos serán independientes para evitar dependencias entre ellos al finalizar la ejecución de una solicitud. Los mutex sirven para proteger la variable local del socket del cliente y los malloc para otorgarle memoria dinámica a ésta y así evitar condiciones de carrera.

El bucle *while* esperará la llegada de conexiones, y cuando un cliente se conecte, los datos de su solicitud se almacenarán en una estructura auxiliar en C llamada *message*. Los parámetros de la solicitud, separados por "\n" y almacenados en un fichero llamado *data.txt*, se leerán utilizando un buffer auxiliar que se reiniciará para cada argumento. El uso de ficheros como estructura para la comunicación se ha elegido por la mayor familiarización de los desarrolladores con estas estructuras. Se ha decidido implementar una comunicación mediante cadenas de caracteres, para que la comunicación por sockets sea independiente del lenguaje de programación utilizado.

Una vez leídos los parámetros, se procesarán en un hilo independiente mediante la función tratar_peticion(void *sc). Este hilo generará copias locales del descriptor del socket y de los datos de la solicitud para evitar interferencias con otros hilos. Se utilizarán mutex para asegurar la ejecución sin fallos de memoria y evitar conflictos de datos y concurrencia paralela. Al finalizar la ejecución, se enviarán los resultados al cliente y se cerrará el socket cliente.

Proxy

Los archivos en lenguaje C para los clientes tan solo contienen la interfaz entre el cliente y el canal de comunicación mediante funciones incluidas en el proxy. La implementación de esta comunicación viene desarrollada en *libclaves.so*.

En cada función se sigue la misma estructura:

1. Se llama a la función connect_to_server(), donde se busca conectar el proxy con el servidor. Se obtienen los valores de las variables de entorno IP_TUPLAS Y PORT_TUPLAS. Si estas no tienen valores, se finaliza el programa. Después se genera el socket del cliente y se conecta al servidor.
2. En caso necesario, se comprueba la validez de los argumentos *value1* y *N_value2*.
3. Se envían todos los datos en una cadena de caracteres al socket del servidor.
4. Se espera a recibir respuesta por parte del servidor.
5. Una vez se recibe esa respuesta, el servidor cierra el descriptor del socket del cliente.
6. Por último la función devolverá el resultado de la operación enviado por el servidor.

Para llevar a cabo todo este proceso, se ha creado un archivo de cabecera *sockets.h* que contiene las funciones *send_message(), receive_message()* y *readLine()*.

3. Pruebas

Para comprobar la validez de este proyecto, se han generado cuatro archivos de tipo cliente que mandarán simultáneamente varias peticiones al servidor.

En el anterior proyecto de Colas de Mensajes ya se comprobó la validez de las funciones que ejecuta el servidor al recibir las peticiones, así que en este proyecto se hará mayor énfasis en la concurrencia del programa. Para ello se ha creado un script que ejecuta todos los clientes al mismo tiempo.

4. Compilar y generar los ejecutables

Para la compilación y generación de ejecutables y librerías se ha utilizado un Makefile. Será necesario ejecutar en terminal en el directorio *ejercicio_evaluable_2* dos comandos en dos terminales. En la primera:

```
./run-server.sh
```

Y en la segunda:

```
./run-clients.sh
```

El primer script corresponde al ejecutable del servidor, y se acompaña de un parámetro que será el número de puerto del socket a ser creado.

El segundo script ejecuta otros scripts de clientes que dan valores a IP_TUPLAS y PORT_TUPLAS al mismo tiempo, o casi al mismo tiempo, para poder validar la concurrencia de la aplicación.