Sistemas distribuidos: ejercicio 2

Sockets

Belén Gómez Arnaldo 100472037

Ana Maria Ortega Mateo 100472023

7 de abril de 2024

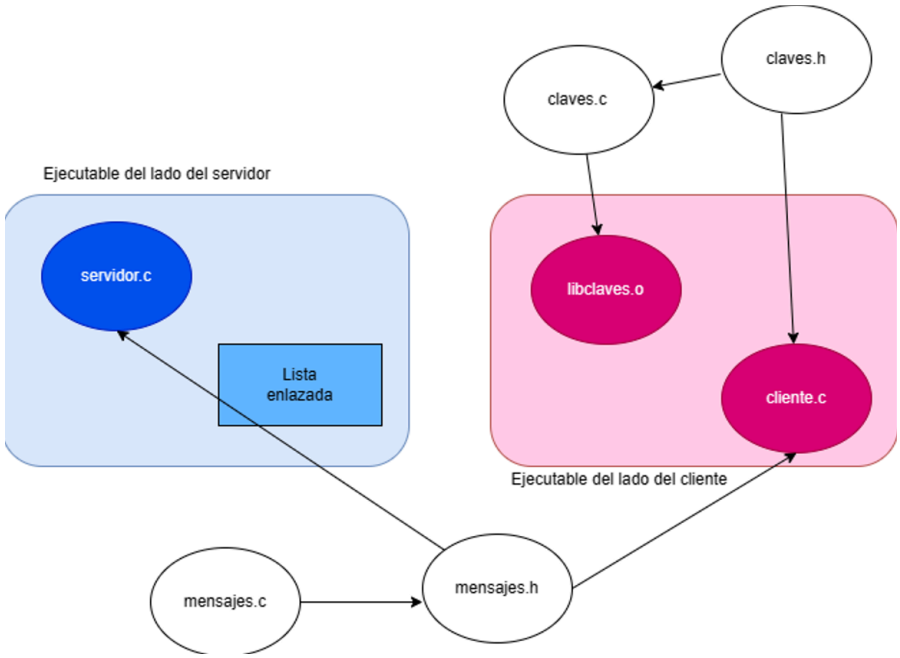
¿Dejamos la misma estructura que antes?

Descripción de la arquitectura.

La arquitectura de nuestro código se divide en dos partes fundamentales; el lado del servidor y el lado del cliente.

- Servidor: el servidor es el encargado de gestionar las peticiones de los clientes y se encarga de almacenar la información que se va añadiendo según las acciones que los clientes realizan. En “servidor.c”, se encuentra la implementación de todas las funciones. En la cabecera de “servidor.h” definimos la estructura de una lista enlazada, que es la que utiliza el servidor para ir guardando las tuplas de los clientes según las acciones realizadas.
- Cliente: el cliente está compuesto por una biblioteca (“libclaves.so”), por dos archivos de código (“clienteX.c” y “claves.c”) y una cabecera de “claves.h” donde se definen todas las funciones que el cliente puede realizar. En “clienteX.c” no se implementa nada de código de las funciones, sirve solamente para que el cliente pueda escribir las funciones que quiere realizar, junto con los parámetros necesarios para realizar la función. En claves.c es donde se realiza la lógica de las funciones por parte del cliente y donde se crea el socket y se envían los datos al servidor. La biblioteca sirve para generar el ejecutable y así poder enlazar claves.c con el cliente.

El cliente y el servidor acceden a un archivo de “mensajes.c”, donde se implementan las funciones necesarias para enviar y recibir mensajes a través de los sockets. También se implementan las funciones para obtener las variables de entorno. En la cabecera “mensajes.h” se incluye la definición de las funciones y una estructura que utiliza el servidor para almacenar todos los valores que tiene que devolver la función get_value. Este es el diagrama de la arquitectura:



Descripción de los componentes.

Para el desarrollo de esta práctica, hemos partido del ejercicio evaluable 1 y hemos modificado la comunicación mediante colas por sockets. Por ello, las funciones pedidas como por ejemplo, get value o init, son las mismas, se manejan mediantes listas enlazadas. La única diferencia se encuentra en el proceso de envío y recepción de mensajes. También se han modificado los mutex, en vez de tratar como sección crítica toda una función, ahora tratamos como sección crítica todas aquellas partes del código donde se accede a un recurso global. Por ejemplo, si un hilo está modificando la lista enlazada, se protegerá está sección para que ningún otro hilo pueda hacer una modificación de la lista simultáneamente. Se ha hecho solo en las partes que se acceden a estos recursos, no en toda la función.

Nuestro código parte de tres componentes principales:

1. ClienteX.c : El cliente podrá introducir cualquier acción de las que están definidas y se ejecutará las funciones correspondientes a dicha acción.
2. Claves.c : Se encarga de manejar las acciones que realiza el cliente y mandarle la petición al servidor. Está compuesta por seis funciones diferentes; init, set_value, get_value, modify, delete y exist. Para establecer la comunicación con el servidor y poder mandar y recibir mensajes mediante los sockets, todas comparten los siguientes pasos:
    - Se definen las variables de los sockets y las variables para el envío de los mensajes (dependerá de la función).

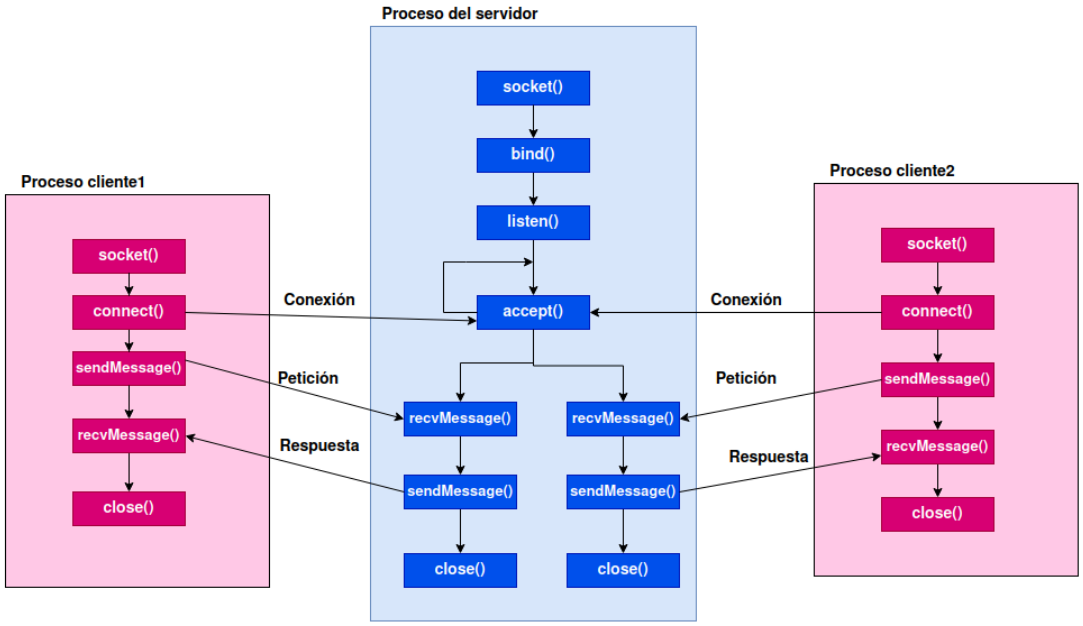
- Se obtienen las variables de entorno con la función ObtenerVariablesEntorno.Esta función, obtiene las variables de entorno IP_TUPLAS y PORT_TUPLAS y las guarda en las variables globales definidas en mensajes.h (componente que comparten tanto cliente como servidor).
- Creamos el socket TCP, con sus características SOCK_STREAM, comunicación orientada a la conexión.
- Obtenemos la dirección IP con nuestra función ObtenerIP(), que simplemente devuelve la IP anteriormente almacenada en la variable global.
- Se inicializa la estructura del socket del servidor para su posterior uso en la conexión.
- Obtenemos el puerto con la función ObtenerPuerto(), que devuelve el puerto almacenado en la variable global y lo convertimos en entero mediante strtol en vez de atoi, para poder hacer manejo de error en caso de que haya sucedido algún error en la conversión.
- Se establece la conexión mediante connect y se procede al envío de los mensajes. Antes de enviar un entero con la función sendMessage()(escribir mensaje en el socket), se convierte a htonl, para asegurar que el entero sea interpretado correctamente por el otro sistema. El vector de dobles se envía haciendo un bucle en función de N y mandando valor por valor.
- Tras el envío de mensajes, el cliente espera a que el servidor le envíe una respuesta a través de recieveMessage()(leer mensaje del socket). Dependiendo de la función esperará solo una respuesta o más, como en el caso de get_value que esperará los valores de la clave que se ha buscado sino ha habido ningún error.
- Finalmente se cierra el socket y nos aseguramos de pasar los valores de tipo entero con ntonl() para obtener su valor en formato entero.

3. Servidor.c : En servidor.c se realiza la implementación de funciones para gestionar las peticiones y almacenar las tuplas.La parte de hilos, mutex y las funciones en sí, son las mismas que en el anterior ejercicio, el único cambio es el explicada anteriormente. En el servidor primero se comprueba si los argumentos pasados por la terminal son dos o no, si al servidor no se le asigna un puerto el código no funcionará. Posteriormente se asignan las variables para el socket y se crea el socket TCP, el cual está orientado a la conexión(sd = socket(AF_INET, SOCK_STREAM, 0). Configuramos el socket y le asignamos el puerto, cogiendo el segundo argumento pasado por terminal(argv[1]), se pasa a entero con strtol para poder hacer manejo de errores. Con bind, asociamos la dirección al socket y se hace un listen() para preparar el socket a aceptar conexiones. Después hacemos un bucle while, para que cada vez que llegue una petición por parte de ese cliente y se acepta la conexión , se crea un socket de la conexión aceptada. Finalmente, el hilo

correspondiente atiende las peticiones que el cliente envía y a la función atender petición se le pasa el socket de la conexión aceptada para poder hacer la recepción y envío de mensajes necesarios para las funciones.

En atender_peticion , se utiliza un mutex y una variable de condición para proteger la copia del socket y se definen las variables para recibir el mensaje y enviarlo. Siempre espera a recibir una operación que indica que función se tiene que realizar y dependiendo de la función esperará a recibir más mensajes o no. Para recibir el vector, primero espera a recibir N, y en función de N (pasándolo con ntohl a entero) va esperando a recibir cada elemento i del vector de dobles. Tras haber realizado las funciones, se realiza un send de la respuesta pasándolo a htonl para asegurar que se manda correctamente. En caso de función get, es decir, si la operación es igual a 2 y no ha habido ningún error, se manda valor1, N y valor2. Para ello, hemos tenido que guardar los valores en una estructura para que quedaran almacenadas tras la función y poderlas mandarlas al cliente, siguiendo el orden de los mensajes que él espera recibir. Por último se cierra el socket creado para ese cliente.

Se tiene en cuenta todos los errores posibles que se realizan durante todo este proceso(tanto en cliente como en servidor) y si es necesario se cierran los sockets correspondientes. Si sucede un error en el servidor, se devuelve en la respuesta al cliente un -1. Para que quede más claro los pasos que realiza cada componente para la comunicación entre sockets, se ha hecho el siguiente diagrama:



Ejecución.

Al ejecutar el comando make se crean archivos ejecutables del cliente y el servidor. Para ejecutar el servidor solo es necesario escribir en la terminal el comando "./servidor PUERTO", siendo PUERTO el número del puerto que se quiera especificar. Es importante que este puerto sea el mismo que se indique para los clientes, sino habrá errores de conexión.

Para ejecutar los clientes, tras hacer el make, solo hace falta poner esta línea en la terminal: "env IP_TUPLAS=direccion_ip PORT_TUPLAS=PUERTO ./clienteX", siendo PUERTO el mismo puerto que en el servidor, direccion_ip la dirección del servidor y X el número del cliente que se quiere ejecutar. Los códigos de los clientes se pueden cambiar para hacer diferentes funciones, nosotras hemos creado el código que consideramos adecuado para hacer una buena batería de pruebas. Si se quiere cambiar el código de los clientes hay que ejecutar después el comando "make". Para ejecutar varios clientes de forma concurrente basta con usar el comando "env IP_TUPLAS=direccion_ip PORT_TUPLAS=PUERTO clienteX & env IP_TUPLAS=direccion_ip PORT_TUPLAS=PUERTO clienteX".

Pruebas.

Comportamiento secuencial

Para comprobar el funcionamiento secuencial del programa usaremos los archivos de *cliente1.c y cliente2.c* . Cuando se llama a cualquiera de las funciones se comprueba el valor que devuelven. En caso de que sea -1 se imprimirá un mensaje de error. Para comprobar el estado del servidor después de cada función se han incluído mensajes de error y mensajes para imprimir la lista cuando se modifica. De esta forma es fácil comprobar el estado del servidor y la lista. Se han hecho las siguientes pruebas, en cliente1 y cliente2:

- En cliente1 se comprueba todas las funciones(init, set_value, modify, get_value, delete y exist), comprobando que se realizan las funciones correctamente y se muestra por la terminal los valores esperados. Realizamos en primer lugar un init, después dos set values, comprobamos con get value la clave añadida anteriormente y comprobamos los valores devueltos. Hacemos un modify para comprobar que se cambian los valores en la lista enlazada y un exist para comprobar que la clave existe. Finalmente eliminamos una de las tuplas y comprobamos con otro exist que ya no se encuentra almacenada.
- En cliente2 realizamos pruebas para la depuración de errores y poder comprobar que son detectados correctamente.

Comportamiento concurrente

Para comprobar el comportamiento concurrente, lanzamos el primer cliente (cliente1) y hemos creado otro, en este caso cliente3, que se ejecutará de manera concurrente. Ambos realizan distintas acciones, si un cliente realiza el set de una clave, el otro puede acceder a ella mediante un get. De la misma manera, si un cliente elimina una clave, el otro cliente no podrá acceder a ella. En el enunciado no se especifica el orden exacto en el que se tienen que hacer las funciones de los clientes. No se especifica que se tenga que intercalar o que se tenga que hacer primero todo un cliente y después el siguiente. Por ello, no hemos definido un orden concreto, pero sí hemos utilizado mutex para realizar las funciones de forma atómica y mantener la consistencia del sistema. Como hemos explicado antes, en vez de hacer un mutex sobre todas las funciones, lo hacemos cuando cada hilo tiene que acceder a una variable compartida, como la lista enlazada.

Conexión

Además de las pruebas para las funciones, hemos realizado las siguientes pruebas para comprobar errores de conexión o errores en las variables de entorno.

- Puerto distinto en el cliente y el servidor. Hemos ejecutado el servidor con este comando "./servidor 4500" y el cliente con este "env IP_TUPLAS= loclahost PORT_TUPLAS=4400 cliente1". El cliente y el servidor utilizan puertos distintos, por lo que no se conectan. En el cliente se pueden ver errores de conexión y al servidor no le llega ninguna petición.
- No especificar las variables de entorno. Si ejecutamos el cliente con este comando "./cliente1.c", como no estamos definiendo ni la dirección ip del servidor ni el puerto, se produce un error en las variables de entorno y no se ejecuta ninguna de las funciones.
- Ejecutar el servidor sin puerto: si ejecutamos el servidor con este comando "./servidor" dará un error en los argumentos ya que se tiene que especificar también el puerto.
- Funcionamiento concurrente: si al ejecutar el código lo hacemos con este comando en los clientes "env IP_TUPLAS= loclahost PORT_TUPLAS=4500 ./cliente1 & ./cliente3" se ejecuta bien el cliente1 pero el cliente3 dará errores al no haber especificado las variables de entorno.