Universidad
Carlos III de Madrid

Universidad Carlos III

Sistemas Distribuidos

Curso 2024-25

EJERCICIO 2

06/04/2025

Inés Fuai Guillén Peña 100495752@alumnos.uc3m.es | g82

Claudia Fernández Menéndez 100495919@alumnos.uc3m.es | g82

Universidad
Carlos III de Madrid

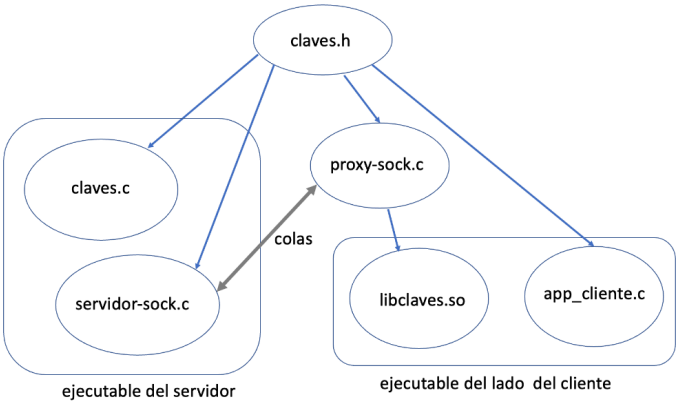
Introducción

El propósito de este ejercicio es desarrollar e implementar un servicio distribuido utilizando **sockets TCP**. Este servicio permitirá el almacenamiento de tuplas de datos compuestas por una clave y múltiples valores asociados a ella.

Para lograrlo, se ha diseñado una aplicación que incluye un servidor responsable de gestionar y almacenar las claves, junto con una API que facilitará a los procesos cliente el acceso a los servicios proporcionados por el sistema.

Diseño

La aplicación debe seguir la siguiente estructura:



Proxy

El `proxy-sock.c` será la API que permitirá al cliente comunicarse con el servidor. Es el responsable de ofrecer la interfaz a los clientes y se encarga de implementar los servicios (funciones) contactando con el servidor anterior. Aquí se implementarán las funciones que enviarán los mensajes correspondientes al servidor.

Servidor

En el archivo `claves.c`, se implementarán las funciones que se nos piden.

El `servidor-sock.c` es el responsable de las comunicaciones con la parte cliente. Este se encargará de recibir los mensajes de los clientes y crear un hilo para manejarlos. Se creará un hilo por cada mensaje recibido y cada uno llamará a las funciones correspondientes dependiendo del valor del campo *operacion*, incluido en el mensaje. Este devolverá el valor del resultado: 0 si todo ha ido bien o -1 si ha ocurrido algún error.

Los archivos de proxy y servidor están relacionados mediante la cabecera `claves.h`, donde se definirán los prototipos de las funciones llamadas por el cliente, y `common.h`, donde se definen las funciones encargadas de enviar y recibir los sockets.

Cliente

El `app_cliente.c` representará a un cliente, que solo llamará la función, o funciones, que quiera.

El archivo `libclaves.so` será la biblioteca que utilizarán las aplicaciones de usuario para usar el servicio.

Todos los ficheros de los clientes tienen en su cabecera `proxy-sock.h`, donde se definen las funciones implementadas en `proxy-sock.c`

Implementación

En primer lugar, desarrollamos el código de `claves.c`. Aquí creamos las funciones *set_value*, *get_value*, *modify_value*, *delete_key*, *exist* y *destroy* siguiendo los requisitos del enunciado. Considerando que las tuplas se almacenan en una lista enlazada, es importante señalar que la función *set_value* inserta la tupla recibida como parámetro al inicio de la lista. Esta decisión se tomó porque, no solo el orden de las tuplas del cliente no es relevante, sino también porque de esta forma es más eficiente.

En segundo lugar, implementamos `servidor-sock.c`. Aquí abrimos un socket en un bucle infinito, esperando conexiones entrantes de los clientes. Cuando un cliente se conecta y envía una peticion, el servidor recibe dicho mensaje, lo interpreta y llama a la función *processRequestThread*, que se encarga de llamar a la función que corresponde dependiendo del valor introducido como *operacion*.

Es importante mencionar que en este punto implementamos un mecanismo de hilos y mutex para asegurar la sincronización entre las operaciones del servidor. Dado que múltiples clientes pueden hacer peticiones concurrentes, utilizamos mutex para bloquear la sección crítica cuando se modifican o leen los datos. Esto previene condiciones de carrera y asegura que las operaciones sobre las tuplas se realicen de manera segura. Cada vez que se procesa una solicitud, se crea un hilo para manejarla, y al final del procesamiento de la solicitud, el hilo se cierra correctamente.

En cuanto al código del archivo `proxy-sock.c`, aquí se implementan las funciones que servirán de API para el cliente para enviar las peticiones al servidor. Todas estas funciones siguen la misma estructura: devuelven la respuesta generada por la función *send_req_to_server*, que es el responsable de comunicarse con el servidor. En esta función, primero obtenemos el puerto y la dirección IP del servidor para, a continuación, resolver dicha dirección, de esta forma admitimos nombres de host como *localhost* y direcciones IP. Después, establecemos la conexión mediante un socket TCP y enviamos la estructura de la petición. Por último, esperamos a recibir la respuesta, la procesamos y devolvemos el valor de la respuesta al cliente.

En el archivo `common.c` implementamos las funciones que nos permitirán enviar y recibir los sockets entre el servidor y el proxy.

Finalmente, en los archivos `app-clientes-X.c` definimos los valores de la petición y llamamos a la función que queremos.

Compilación

Para la compilación del programa utilizamos un archivo *Makefile*. Simplemente ejecutando el comando *make* en la terminal se generan dos ejecutables: el ejecutable del servidor que implementa el servicio y el ejecutable de cliente (o ejecutables, en caso de haber varios) obtenido a partir del archivo app-cliente-X.c (X haciendo referencia al número del cliente). También se genera la biblioteca dinámica *libclaves.so*.

Para ejecutar al cliente tenemos primero que ejecutar el servidor (*./servidor-sock <PUERTO>*) en una terminal, y el cliente (*./env IP_TUPLAS=localhost PORT_TUPLAS=<PUERTO> ./app-cliente-X*) en otra, donde:

- *<PUERTO>:* Corresponde al número de puerto, un número de 4 dígitos, que debe ser igual para los clientes y el servidor.
- *X:* Corresponde al número del cliente.

Casos de prueba

Hemos desarrollado varios clientes para realizar las pruebas de funcionamiento del sistema. A continuación, explicaremos con más en detalle en qué consiste cada cliente, el comportamiento esperado y los resultados obtenidos.

Ninguno de los casos de prueba funcionará si el servidor no está activo en otra terminal o si el número de puerto del servidor no coincide con el introducido por el cliente.

Nota: Para ir revisando el correcto funcionamiento del código, hemos añadido mensajes de depuración tanto en la terminal del servidor como en la del cliente para saber dónde estamos en todo momento (CLAVES y SERVIDOR para `servidor-sock.c` *y PROXY para* `app-cliente-X.c`*). También hay mensajes de depuración para las funciones sendMessage() y recvMessage(), para asegurar que se recibe lo mismo que se envía y viceversa. Se identifican en terminal porque empiezan por "---".*

Cliente 1: Prueba básica de inserción y obtención (set_value, get_value)

Este cliente trata de insertar una tupla mediante *set_value,* para la que establecerá una *key* identificadora. Posteriormente, hará un *get_value* con dicho key para recuperar ese valor y consultar la tupla recién insertada.

Resultados: Podemos observar los dos accesos al servidor, que envía las respuestas que recibirá el cliente. Ambas operaciones se realizan correctamente.

Cliente 2: Prueba básica de modificación de valores

Este cliente realizará tres funciones: primero llamará a *set_value* para crear e insertar una tupla. A continuación, realizará cambios en la tupla recién creada mediante *modify_value*. Finalmente, realiza un *get_value* para recuperar la tupla y verificar que efectivamente se haya modificado el valor.

Resultados: Se observan los tres accesos al servidor y tres diferentes respuestas que devuelve el proxy. No hay ningún error inesperado, las operaciones se realizan correctamente.

Cliente 3: Prueba de concurrencia

Este cliente trata de poner a prueba el funcionamiento de nuestro sistema en términos de concurrencia. Simula la ejecución de dos clientes diferentes, que tratan de hacer una operación *set_value* y *modify_value* al mismo tiempo. Las operaciones irán dirigidas a valores de *key* diferentes, por lo que antes de lanzar los hilos de los clientes se hará un *set_value* con un valor de *key* 10, que es el que se usará por el cliente que quiere modificar. De esta manera podemos asegurarnos de que la concurrencia funciona, pues los dos "clientes" intentarán acceder al servidor y hacer consultas al mismo tiempo.

Resultados: Primero se observa la ejecución del primer *set_value.* A continuación, si seguimos el orden de lo impreso por terminal, vemos cómo comienza la ejecución de *cliente1,* que aparentemente para y empieza *cliente2,* que termina su ejecución y luego aparece la mitad restante de *cliente1,* que ya termina su ejecución. Después, se devuelve la respuesta correspondiente a *cliente2.* Esto parece indicar que el primero en ejecutarse suele ser el cliente 2, que bloquea el mutex de la función *resolve_ip().* Por eso la ejecución del primer cliente aparentemente para de forma brusca y continúa más tarde. Esto indica que la concurrencia funciona correctamente.

Cliente 4: Prueba de eliminación de tupla

Este cliente creará una tupla con *set_value,* que posteriormente eliminará mediante el uso de *delete_key.* Finalmente, llama a *get_value* para comprobar si la eliminación de la tupla ha sido exitosa, en cuyo caso saltará un error con respecto a que la tupla a consultar no existe.

Resultados: Podemos ver tres respuestas del proxy para el cliente, siendo la última de ellas "-1". El cliente finaliza por un error. Esto indica el correcto funcionamiento de nuestro cliente, que ha eliminado una tupla que posteriormente está intentando recuperar mediante el *get_value.*

Cliente 5: Prueba de consulta de tupla inexistente

Este cliente trata de confirmar si una tupla que no existe existe mediante *exist.* Buscamos forzar que el resultado sea 0, lo cual significa que, en efecto, no existe dicha tupla.

Resultados: El proxy devuelve 0 al cliente. Esto implica que ha saltado un error en la función *exist,* devolviéndonos el resultado esperado. Se puede ver que entra correctamente en la función y devuelve lo esperado.

Cliente 6: Prueba de operaciones en claves múltiples

Este cliente hará dos *set_value* seguidos. Con él buscamos comprobar que dicha función funciona correctamente y que permite la inserción de varias tuplas.

Resultados: En efecto, vemos que se llama a *set_value* dos veces. Para la segunda los *prints* observados en pantalla son ligeramente diferentes, pues en vez de crear un hilo se utiliza el ya existente. Sin embargo, por el terminal del servidor si que observamos lo esperado, por lo que consideramos que el resultado es correcto.

Cliente 7: Prueba de error en el servidor (respuestas del servidor)

Este cliente trata de insertar una tupla mediante *set_value,* en la que el valor de *v2* es inválido para forzar un error. Con él, pretendemos depurar la detección de errores del propio *set_value.*

Resultados: Al tratar de insertar la tupla con valores incorrectos, vemos como se entra en la función y como la respuesta que recibe el servidor del proxy es -1, por lo que el funcionamiento es correcto.

Cliente 8: Prueba de eliminación de todas las tuplas

Usamos este cliente como un *hard-reset*, en el que simplemente llamamos a *destroy* para eliminar todas las tuplas creadas por los distintos clientes.

Resultados: En efecto, se destruyen todas las tuplas, por lo que podemos volver a generar tuplas nuevas con las keys de las antiguas y no saltaría ningún error.

Observaciones

Al ejecutar un cliente que crea una tupla, esta se mantiene siempre que el servidor permanece encendido. Esto implica que, por ejemplo, si ejecutamos *app-cliente-1* varias veces, la primera se hará correctamente pero las siguientes darán error, pues se intenta insertar una tupla que ya ha sido insertada.

Si llamamos a *app-cliente-8,* podemos volver a ejecutar a los clientes que insertan tuplas sin darnos error. Lo mismo ocurre si cerramos el servidor y lo volvemos a abrir, todas las tuplas que hubiera habrán sido eliminadas y nos dejará volver a insertar.