Luis Gómez-Manzanilla Nieto 100472006

Ignacio Fernández Cañedo 100471955

EJERCICIO EVALUABLE 2

ÍNDICE

ÍNDICE 1
INTRODUCCIÓN 1
ESTRUCTURA 2
DISEÑO 3
COMUNICACIÓN 5
PRUEBAS 6
IMPORTANTE 7

INTRODUCCIÓN

Este documento contiene la memoria del segundo ejercicio evaluable de la asignatura Sistemas Distribuidos. Este ejercicio trata de diseñar e implementar el mismo servicio distribuido del primer ejercicio evaluable que permite almacenar tuplas. En esta aplicación, a diferencia del ejercicio previo, se utilizarán sockets TCP en lugar de colas de mensajes.

ESTRUCTURA

La estructura del ejercicio es la siguiente:

claves.h: Este es el archivo cabecera que contiene la declaración de las funciones. Las funciones incluyen init, set_value, get_value, modify_value, exist y delete_key, que implementan las operaciones que puede realizar el cliente.

claves.c: Este archivo contiene las implementaciones de las funciones declaradas en claves.h.

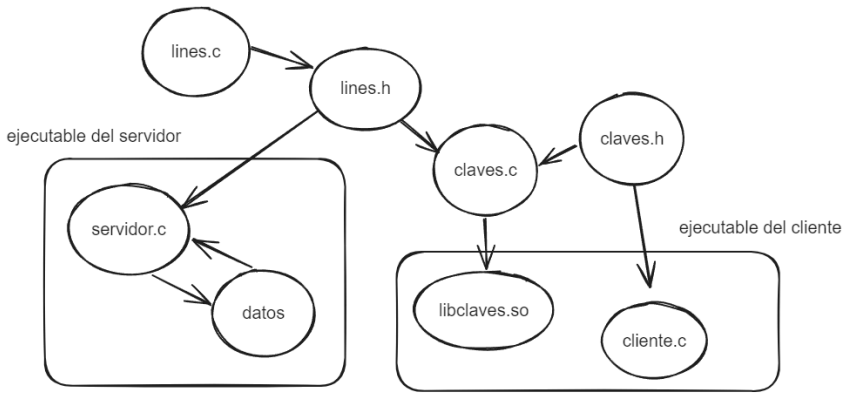
lines.h: Este es el archivo cabecera que contiene la declaración de las funciones de lines.c, sendMessage, recvMessage y readLines.

lines.c: Este archivo contiene funciones para enviar (sendMessage) y recibir (recvMessage) mensajes a través de un socket. También se incluye la función readLine que se utiliza en el caso en que una cadena de caracteres finalice con '\0', ya que no se sabe a priori su longitud y no se puede usar la función recvMessage para leerla.

Servidor.c: Este archivo contiene el código del servidor. El servidor se encarga de gestionar las operaciones de los clientes. Cuando se inicia, crea un socket para recibir peticiones de los clientes. Cada petición se maneja en un hilo separado. Las peticiones pueden ser para añadir una nueva tupla, obtener los valores de una tupla existente, modificar los valores de una tupla existente, comprobar si existe una tupla con una determinada clave o eliminar una tupla. Después de tratar cada petición, el servidor escribe todas las tuplas en un archivo.

cliente.c: Este archivo contiene el código del cliente. El cliente puede enviar peticiones al servidor mediante las distintas funciones declaradas en la cabecera.

datos: Fichero en el que se almacenarán todas las tuplas.



DISEÑO

Para almacenar los elementos clave-valor1-valor2 hemos decidido utilizar ficheros como estructura de almacenamiento de datos. A diferencia de las listas anidadas, gracias a utilizar ficheros en nuestra implementación, los datos se guardarán entre ejecuciones. Además, tenemos un vector donde se cargan las tuplas existentes de ejecuciones anteriores. Las consultas principalmente afectan al vector y posteriormente este vector se carga en el fichero.

En el caso de claves.c, las funciones incluidas en este archivo comienzan con la creación del descriptor del socket, así como la declaración de las estructuras correspondientes como hostent o sockaddr.

Primero, se configura la estructura *sockaddr_in* para la dirección del servidor. Esta estructura contiene detalles como la familia de direcciones (AF_INET), la dirección IP del servidor y el número de puerto del servidor, que se obtienen de variables de entorno utilizando getenv.

Posteriormente se realiza la conexión al servidor utilizando connect. Si se logra establecer la conexión y no da error, se envía la operación utilizando sendMessage dependiendo de la función. En el caso del Init, la operación es el 0. Si falla, se devuelve -1. Posteriormente se espera una respuesta del servidor con *recvMessage* y se maneja el error.

Finalmente, la función cierra el socket utilizando la llamada al sistema close y devuelve la respuesta recibida del servidor.

Respecto al archivo servidor.c, inicialmente se define el número máximo de tuplas a almacenar y se declaran varias variables globales. Las variables globales incluyen los mutex para la sincronización de hilos, el array de tuplas, el array de claves, el número actual de tuplas y el nombre del archivo donde se almacenan las tuplas.

Posteriormente se crean funciones para escribir y para leer las tuplas en un archivo de texto. Ambas funciones utilizan mutex para asegurarse que solo un hilo puede leer o escribir en el archivo a la vez.

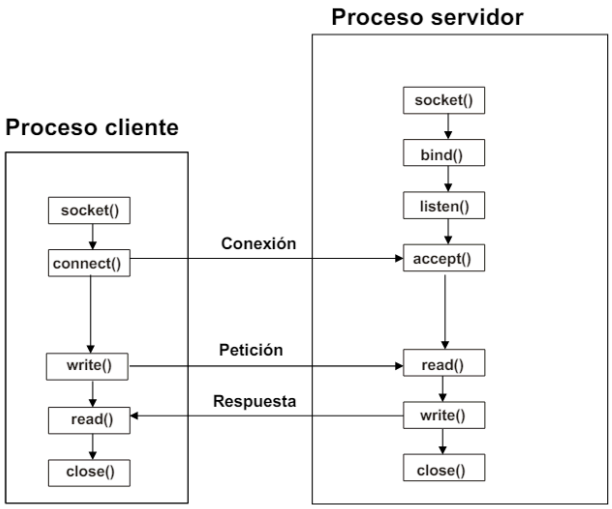
A continuación se declaran las funciones, en primer lugar se encuentra la función init, que inicializa el servidor, establece el nombre del archivo donde se almacenarán las tuplas y llama a las funciones para leer y escribir las tuplas. El resto de funciones bloquean los mutex al comenzar y se desbloquean al finalizar.

La función *tratar_peticion* se ejecuta en un hilo separado para cada petición del cliente. La función comienza extrayendo el descriptor del socket del argumento sockfd. Posteriormente recibe la operación del cliente mediante *recvMessage*, dependiendo del valor de la operación. La operación recibida está en formato de red, por lo que se utiliza ntohl para convertirla en formato de host. La función realiza diferentes acciones, cada valor de la operación corresponde a un comando distinto que a su vez corresponde a cada función. Después de procesar la operación solicitada, se almacena la respuesta en res y se envía al cliente con *sendMessage*, convirtiéndolo en representación de red con htonl. Finalmente, la función cierra el socket liberando los recursos y terminando la conexión con el cliente.

A continuación se encuentra la función main. Primero, comprueba el número de argumentos pasados. Posteriormente se inicializan los atributos del hilo que se utilizará para manejar las peticiones de los clientes. La función comienza creando un socket y permitiendo que este se reutilice utilizando *setsockopt*. Como hemos visto en la comunicación mediante sockets stream, se comienza haciendo un *bind()* para enlazar el socket a la dirección y el puerto. Después se realiza *listen()* para escuchar las conexiones entrantes y por último *accept()* dentro de un bucle infinito para aceptar las conexiones entrantes. Crea un hilo por cada conexión para ejecutar *tratar_petición*. Finalmente el socket se cierra al abandonar el bucle.

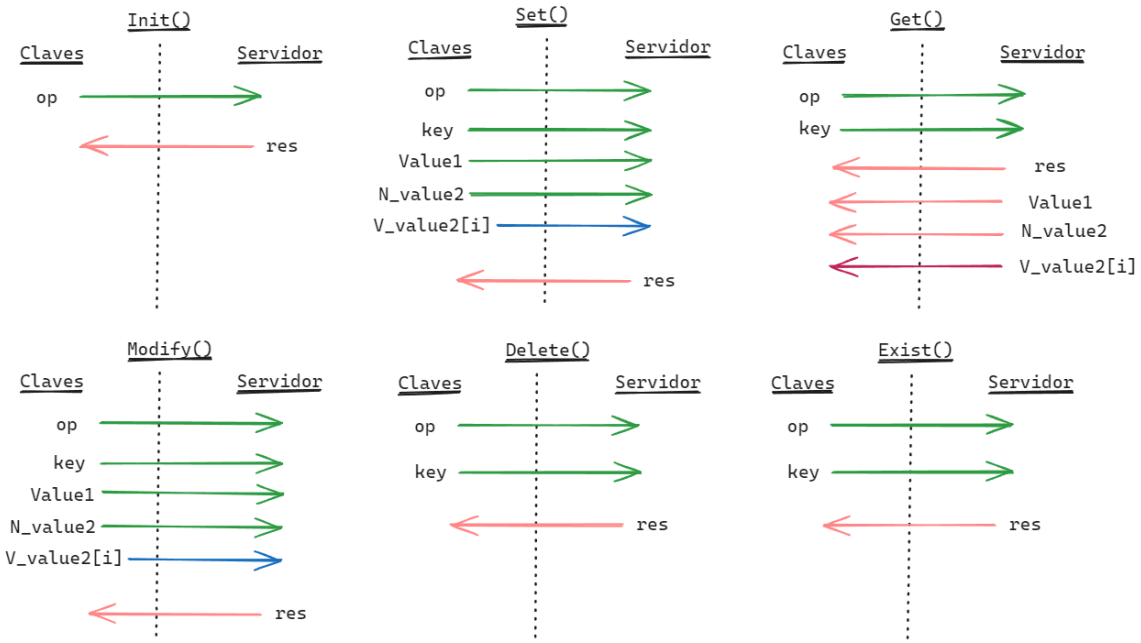
COMUNICACIÓN

La comunicación la realizamos usando sockets TCP ya que queremos que las conexiones sean fiables. El uso de esto sockets es el siguiente:



Dicha figura representa la manera de realizar una conexión entre el servidor y el cliente mediante sockets. En la práctica, el servidor tiene un socket principal y cuando recibe una petición (accept) genera un nuevo socket y crea un hilo hijo que maneja esa conexión.

Por otro lado, cada función implementada en claves.c realiza diferentes conexiones. Dichas conexiones están representadas en la siguiente figura. Cada diagrama está separado en dos zonas, la de claves y la del servidor que les representa, la flecha hace referencia a la conexión en sí, siendo el inicio de la flecha quien manda el mensaje y la punta quien lo recibe. El orden en el que aparece determina el orden de ejecución en el que se está realizando dichas conexiones en el código de la función. Por otro lado, como V_value2 es un vector, para enviarlo o recibirlo es necesario enviar cada uno de sus elementos, esto en el código lo hacemos con un bucle for, y en el diagrama lo representamos como una flecha de color diferente a las demas y siendo V_value2[i].



PRUEBAS

Para ejecutar el ejercicio, en primer lugar se deberá hacer un “make”, después, para ejecutar el servidor basta con escribir “./servidor”, si no deja se deberá escribir “LD_LIBRARY_PATH=. ./servidor”. Para ejecutar el cliente también se debe escribir “LD_LIBRARY_PATH=. ./cliente” en la consola. El uso de LD_LIBRARY_PATH=. es porque no suele coger bien la dirección de la biblioteca, también se puede evitar poniendo eso al ejecutar si se configura correctamente el path con esta línea EXPORT LD_LIBRARY_PATH=$LD_LIBRARY_PATH:. , si funciona correctamente, se puede omitir ese fragmento por lo que la ejecución sería “./cliente”.

Para realizar pruebas de concurrencia y del funcionamiento del servidor atendiendo peticiones de diferentes clientes, hemos creado dos ficheros de cliente, cliente.c y cliente2.c. Cada uno tiene su ejecutable correspondiente. La diferencia entre estos dos es que cliente es quien se tiene que ejecutar primero ya que es el que realiza el init y en cambio el cliente2 no realiza el init pero utiliza el resto de funciones.

Además, el archivo de cliente.c tiene implementadas todas las funciones para mostrar el correcto comportamiento secuencial (ejecutar las funciones y ver cómo se realizan correctamente). Esto lo hacemos llamando a las funciones y en caso de que se realicen correctamente y se reciba un 0, se imprimirá por pantalla una confirmación.

En cliente.c primero realizamos un init para inicializar el sistema. Posteriormente usamos set_value() para guardar dos tuplas diferentes. Hasta aquí podemos comprobar que se ha generado el archivo datos.txt y contiene las dos tuplas creadas. Posteriormente realizamos un get_value de la segunda tupla e imprimimos los elementos del vector obtenido para comprobar que son los mismos que hemos seteado. A continuación, modificamos la tupla que acabamos de obtener con modify_value() y volvemos a obtenerla e imprimirla para ver como ha cambiado. Por último, usaremos delete_key() de la primera tupla y a continuación usaremos exist() para comprobar si las dos tuplas que introducimos inicialmente existen. El resultado de esto será que la primera no existe pero la segunda si. Finalmente en datos.txt de las dos tuplas que hemos añadido inicialmente solo queda la segunda y tendrá los valores del vector modificados.

Por otro lado, el cliente2 está pensado para comprobar el comportamiento concurrente. Para ello, una vez compilado el proyecto, inicialmente ejecutaremos el cliente como se explica anteriormente para realizar el init. Posteriormente ejecutamos el siguiente comando “LD_LIBRARY_PATH=. ./cliente2 & LD_LIBRARY_PATH=. ./cliente2”, con este creamos dos procesos que ejecuta el cliente2 concurrentemente.

En este archivo lo que hacemos es set_value() primero dos valores, con key 3 y 4 respectivamente. En caso de que se realice correctamente se mostrará un mensaje de confirmación y en caso contrario uno de error. Como podemos observar al ejecutar, por cada set_value() aparecen los dos mensajes y esto se debe a que un hilo realiza el set correctamente y genera el mensaje correcto y el otro hilo como ya se ha realizado el set, este da error y se muestra el mensaje de error. Posteriormente, comprobamos la existencia de las dos claves introducidas y eliminamos la 3, solo se produce un mensaje de que se ha eliminado ya que en el otro no existe por lo que da error. Por último, se comprueba si la clave que hemos eliminado existe y como podemos observar se muestra dos veces que no existe la clave.

IMPORTANTE

No entendemos porqué pero al ejecutarlo en guernika no nos funciona pero en mi pc que usa WSL2 si, el código es el mismo.