MEMORIA EJERCICIO EVALUABLE 2

Universidad Carlos III

Grado Ingeniería Informática 2023-24

Procesadores del lenguaje (Grupo 81)

Práctica realizada por:

- **Tomás Mendizábal** (NIA: 100461170, Email: 100461170@alumnos.uc3m.es)
- **Alejandro Díaz Cuéllar** (NIA: 100472173, Email: 100472173@alumnos.uc3m.es)

Diseño elegido

Lado del cliente

El lado del cliente se divide en 2 archivos: cliente.c y claves.c. En el primero se encuentran todos los pedidos a realizar. Se destaca que en este archivo no hay ninguna funcionalidad implementada de manera que el cliente no sabe como realmente es la implementación solo puede hacer las peticiones.

En claves.c se encuentran implementadas las 6 funciones pedidas. Lo primero que se hace es abrir el socket del cliente usando la función clientSocket. Para poder hacerlo se tiene que disponer de las dos variables de entorno “IP_TUPLAS” y “PORT_TUPLAS”. Tanto la IP como el puerto deben coincidir con la IP del servidor y el puerto que se pasó como argumento al servidor. Si alguna no está definida, se da un error. A continuación, se comprueban errores de apertura del socket. Una vez abierto, se procede a enviar las peticiones a través de mensajes. Para el envío y la recepción de mensajes se utilizan las funciones sendMessage y recvMessage. Con el send se envía los códigos de operación y los componentes que vayan a ser utilizados (key, value1, N_value2, V_value2). En esta última parte también hay comprobaciones de errores. Existe el caso particular de la función get, que después de recibir la respuesta tiene que copiar todos los valores recibidos. Al terminar, se recibe el entero de respuesta y se cierra el socket.

Lado del servidor

Antes de explicar la funcionalidad del servidor se va detallar la forma de almacenamiento elegido y como se usa. Para el almacenamiento de las tuplas hemos decidido usar un archivo de texto binario. Sin embargo este archivo solo se usa para almacenamiento para alojar las tuplas cuando no está funcionando el servidor. Cuando se inicia el servidor todas las tuplas (si existe alguna) se cargan en un array dinámico de tuplas.

En el inicio de la función main se inicializa el array dinámico “almacen” y se guarda el número máximo de elementos. Luego se llama a la función *load* que lee (y crea si es necesario) los elementos del almacen.txt. Adicionalmente, se asigna la señal SIGINT a la función *close_server*. El siguiente bloque de código son todas las declaraciones de todas las variables que se van a utilizar (mutex, hilos, condit). Se procede a abrir el socket del servidor usando el puerto pasado como argumento. En este momento, se entra en un bucle infinito (hasta que se reciba la señal SIGINT). En dicho bucle se espera a recibir una petición con las funciones serverAccept y recvMessage. Si la petición se lee correctamente, se crea un hilo para tratarla. En este momento interviene el primer mutex. Este mutex se utiliza para que cada hilo copie el socket cliente de la variable global a una variable local.

La función del hilo se llama tratar_peticion es la que ejecuta cada hilo. Al acabar de copiar la variable global a la local se libera el primer mutex. Ahora, se llama a la función pedida en la petición. En todas las funciones se usa un segundo mutex. Este mutex es necesario para poder evitar las condiciones de carrera al leer y escribir en el almacén.

En init() se hace un lock se libera el almacén también se vacía el archivo de texto "almacen.txt". Al final se hace unlock. En el set value se empieza haciendo un lock. Se comprueba si ya existe la clave y si no es el caso se inserta en el almacén. Get_value() es sencilla, porque solo requiere de una interacción por el almacén y copia de valores si existe la clave. La función modify es similar a set, pero en vez de realizar una inserción, se modifica un valor del almacén. La penúltima función es delete. Este función busca una clave (si existe) y la borra del almacén. Más adelante tenemos la función exist() que devuelve 0 si no existe la clave y 1 si existe, en otros casos devuelve error.

Cabe destacar que toda la información que se transmite a través de los sockets es independiente del lenguaje ya que todos los campos se envían como cadenas de caracteres.

Pasos para compilar y ejecutar

A continuación se va a detallar una guía con los pasos a seguir para poder compilar el proyecto. El objetivo final será generar los ficheros ejecutables: "servidor" y "cliente". También se destaca que al ejecutar una petición usando cliente y servidor se va a crear un directorio llamado "data_structure". Aquí se generará el archivo del almacén llamado: "almacen.txt". Por último se recomienda abrir dos terminales distintas: una para ejecutar el servidor y otra para ejecutar el cliente.

Los pasos para poder ejecutar con una sola terminal son:

1. Descomprimir el archivo tar.
2. Abrir una terminal.
3. Cambiar el directorio a "Ejercicio2_Distribuidos".
4. Compilar con la sentencia: "make".
5. Para ejecutar el servidor y un cliente secuencial se tiene que correr con el siguiente comando: "./servidor 4200 & ./execute_client.sh cliente"**. Este ejecuta el servidor en segundo plano y el cliente en primer plano.
6. Para terminar el proceso servidor usar el comando ps para encontrar el pid del servidor y luego correr "kill -s INT pid_servidor"

Los pasos para poder ejecutar con dos terminal son:

1. Descomprimir el archivo tar.
2. Abrir dos terminales.
3. Cambiar el directorio a "Ejercicio2_Distribuidos" en ambas terminales.
4. Compilar con la sentencia: "make".
    a. "make" compila el proyecto
    b. "make run_s" en una terminal y "make run_c" en otra.

5. Para ejecutar el servidor y un cliente secuencial se tiene que correr los comandos: “./servidor 4200” y “./execute_client.sh cliente”** cada uno en en las

Los pasos para poder ejecutar las pruebas secuenciales:

1. Descomprimir el archivo tar.
2. Abrir una terminal.
3. Cambiar el directorio a “Ejercicio2_Distribuidos”.
4. Compilar con la sentencia: “make”.
5. Para ejecutar el servidor y un cliente secuencial se tiene que correr con el siguiente comando: “./servidor & ./execute_client.sh secuencial”**. Este ejecuta el servidor en segundo plano y el cliente en primer plano.
6. Para terminar el proceso servidor usar el comando ps para encontrar el pid del servidor y luego correr “kill -s INT pid_servidor”

PRUEBAS

Para las pruebas hemos dividido estas en dos grupos:

- **Pruebas de error** secuenciales, donde se evaluará el funcionamiento de las propias funciones y se verificará si estas funcionan correctamente. Es decir, que get_value coja correctamente los valores para la clave o que devuelva -1 si no existe, que exist devuelva correctamente si existe o no una tupla, que no se permita que la cadena sea mayor a 256 caracteres, etc. Para ello simplemente se ejecutará un archivo con un único cliente que comprobará todos estos casos de prueba
- **Pruebas de concurrencia**. Para probar que el sistema funciona bien cuando se ejecutan muchos clientes a la vez hemos diseñado un pequeño programa al que llamamos “cliente aleatorio” que estará en el archivo cliente.c. Este programa elige de manera aleatoria una operación de entre las disponibles,, realizando la operación. Queríamos simular el hecho de que en el sistema real los clientes van a realizar operaciones distintas y no queríamos estar haciendo un montón de archivos distintos con distintas operaciones. Para probar que funciona con muchos clientes ejecutamos directamente en la terminal un comando del tipo: for i in {1..10}; do ./cliente & done;. Hay que mencionar que este cliente aleatorio tiene dos modos de operación. Si se le llama sin argumentos el cliente utilizará una clave aleatoria, que simula mejor el caso real. Por otro lado, si se le llama con un 1 como argumento (for i in {1..10}; do ./cliente 1 & done;) entonces la clave será estática para todos los clientes y operaciones, lo que sirve para comprobar el correcto funcionamiento de la concurrencia y la atomicidad de las operaciones realizadas en el servidor.

** Se utiliza un archivo .sh para no tener que inicializar las variables de entorno.