Sistemas distribuidos

**Ejercicio evaluable 2: sockets TCP**

Logotipo

Descripción generada automáticamente

**Grupo 82, grupo N**

**Elisa Utrilla Arroyo (100451242)**

**Luis González Perea (100451116)**

**Diseño y funcionamiento**

En general, sigue el mismo diseño que el ejercicio 1. Sin embargo, la forma de enviar datos con socket es diferente respecto a la de las colas. Aquí se ilustra gráficamente:

|  |
| --- |
| Diagrama, Esquemático  Descripción generada automáticamente |

1. **El cliente manda una petición a claves**, a través de la llamada a una función (tal y como pasaba en el anterior ejercicio).
2. Claves crea un socket cliente y **busca establecer conexión con el servidor**. Lo logra gracias a las variables de entorno que introducimos en tiempo de ejecución em el ejecutable del cliente y el del servidor (la ip del servidor es el localhost y el puerto de conexión 8080). Obtiene el “host” mediante **gethostbyname(IP\_TUPLAS)**, y, una vez establecido, configuramos el socket\_in server\_addr al puerto introducido anteriormente
3. Transformamos la petición de tipo **struct a char** utilizando la función “peticion\_to\_char” que hemos incluido en “separar\_mensaje.c” (módulo que implementa la lógica de paso de struct a char y viceversa, pues no debemos enviar structs, como hacíamos en el ejercicio evaluable 1). Una vez transformada, **mandamos el mensaje al socket servidor** usando “sendMessage”.
4. En el servidor, indicamos que su dirección pueda usarse incluso después de desconectarse. Inicializamos el servidor. Cuando se ha conectado con el cliente, creamos **hilos detached bajo demanda para atender las peticiones**. Los hilos accederán al tratamiento de la petición mediante mutex. Las peticiones, recibidas como char, se leen empleando la función “readLines” del módulo “lines.c”, y se transforman a struct mediante “char\_to\_peticion” de “separar\_mensaje.c”
5. El tratamiento de peticiones es exactamente igual que en el proyecto anterior, usando **un sistema de ficheros** para almacenar las tuplas.
6. Después del tratamiento de la petición, el struct respuesta se **transforma a char y es enviado al socket cliente** mediante la función “sendMessage”.

El código de **“cliente.c”** utiliza la biblioteca compartida “claves” para realizar peticiones al servidor. Desde el main llama a todas las funciones a modo de prueba.

El código de **“claves.c”** se comunica con un servidor a través de sockets. Las peticiones que se pueden realizar:

1. init(): Inicializa el servidor.
2. set\_value(): Establece un valor asociado a una clave determinada en el servidor.
3. get\_value(): Obtiene el valor asociado a una clave determinada del servidor.
4. modify\_value(): Modifica el valor asociado a una clave determinada en el servidor.
5. exist(): Verifica si una clave determinada existe en el servidor.
6. delete\_key(): Elimina una clave determinada del servidor.
7. copy\_key(): Copia el valor asociado a una clave a otra clave en el servidor.

**Forma de compilar**

Para compilar nuestro proyecto y por comodidad, hemos utilizado el mismo makefile que en el trabajo anterior, aunque hemos añadido las funciones de **separar\_mensaje.c** (pasa los struct a char y viceversa para el envío de peticiones) y **lines.c** (código proporcionado en Aula Global para enviar y recibir los mensajes del socket con control de errores):

Antes de nada, posicionamos qué elementos irán enlazados para qué ejecutables y cuáles objetos irán dentro de la librería dinámica, de esta forma:

Texto

Descripción generada automáticamente

1. Creamos la biblioteca dinámica libclaves.soque será compartida con cliente.c y, en este caso, añadimos lines.c a esta biblioteca dinámica (para el envío por sockets). Primero compilamos claves.c como una librería dinámica.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

1. Al crear el ejecutable de clientes, enlazamos el archivo .so a la compilación.

Texto

Descripción generada automáticamente

1. Compilamos el servidor. Utilizamos 3 programas: servidor.c, tratamiento\_serv.c (tiene las funcionalidades que haría el servidor tras recibir el mensaje del cliente por su cola) y lines.c (que contiene las funciones de enviar y recibir mensajes por el socket).

Texto

Descripción generada automáticamente

1. No hace falta indicar mensaje.h en el makefile porque ya está incluido en todos los ficheros.

Y asi se nos generan los ficheros necesarios que se pueden ejecutar en dos terminales distintas usando ./cliente y ./servidor. En este caso, cómo usamos variables de entorno, ejecutamos los ejecutables de la siguiente manera: **PORT\_TUPLAS=8080 ./servidor** y **IP\_TUPLAS=localhost PORT\_TUPLAS=8080 ./cliente**.

Hay que destacar que en este caso hay que exportar la ruta de la biblioteca dinámica, así que previamente usaremos este comando en la terminal: **export LD\_LIBRARY\_PATH=/home/elisa/Escritorio/distribuidos-ejercicio-2**.

El ejecutable clientes.c es solo un archivo de pruebas que ejecuta todas las posibles funciones del servidor en el que además comprobamos la concurrencia de este .En nuestro caso, los hilos se crean **bajo demanda en el servidor,** es decir, no se crea otro hilo hasta que no se envía la petición.

**Pruebas**

Hemos realizado una serie de pruebas para comprobar el correcto funcionamiento del programa:

* **Prueba 1**: se encuentra en cliente.c, consiste en ejecutar la función init.
* **Prueba 2**: se encuentra en cliente.c, consiste en usar set\_value con valores válidos.
* **Prueba 3**: se encuentra en cliente.c, consiste en insertar valores para la clave de la prueba anterior.
* **Prueba 4**: se encuentra en cliente.c, consiste en comprobar que existe la tupla asociada a la clave de la prueba 2.
* **Prueba 2.1**: se encuentra en cliente.c, consiste en comprobar que intentar insertar valores para una clave ya existente devuelve error.
* **Prueba 5**: se encuentra en cliente-modify.c, consiste en modificar los valores asociados a una clave insertada con anterioridad.
* **Prueba 6**: se encuentra en cliente-modify.c, consiste en comprobar que da get\_value devuelve error al tratar de recuperar los valores de una clave que no tiene asignada valores previamente.
* **Prueba 7**: se encuentra en cliente-existfail.c, consiste en comprobar que la función exist devuelve 0 cuando la clave no existe.
* **Prueba 8**: se encuentra en cliente-copy.c, consiste en copiar los valores de una tupla a otra cuya clave existe.
* **Prueba 9**: se encuentra en cliente-copy.c, consiste en copiar los valores de una tupla a otra cuya clave no existe, por lo que se creará una nueva tupla con la nueva clave y los valores pasados por parámetro.
* **Prueba 10**: se encuentra en cliente-copy.c, consiste en copiar los valores de una tupla no existente a otra cuya clave si existe, devolverá error.
* **Prueba 11**: se encuentra en cliente-delete.c, consiste en crear una clave y borrarla.
* **Prueba 12**: se encuentra en cliente-delete.c, consiste en intentar borrar una clave que no existe, devolverá -1.