uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería Informática
2022-2023

Sistemas Distribuidos

Sockets

Grupo B
Ángel José Mancha Núñez
100451302

Ruth Navarro Carrasco
100451032

Abril 2023

Índice

Diseño **2**
Implementación de sockets 2
Compilación **4**
Casos de prueba **5**

Diseño

En este ejercicio se ha realizado la implementación de un servicio distribuido utilizando sockets con el fin de almacenar las peticiones que los clientes realicen. Con respecto al diseño, se ha seguido con la base que se realizó para el ejercicio de colas de mensajes POSIX pero se han realizado algunas modificaciones. Entre ellas, utilizar sockets en lugar de colas de mensajes. La estructura del código la podemos encontrar en la siguiente figura:

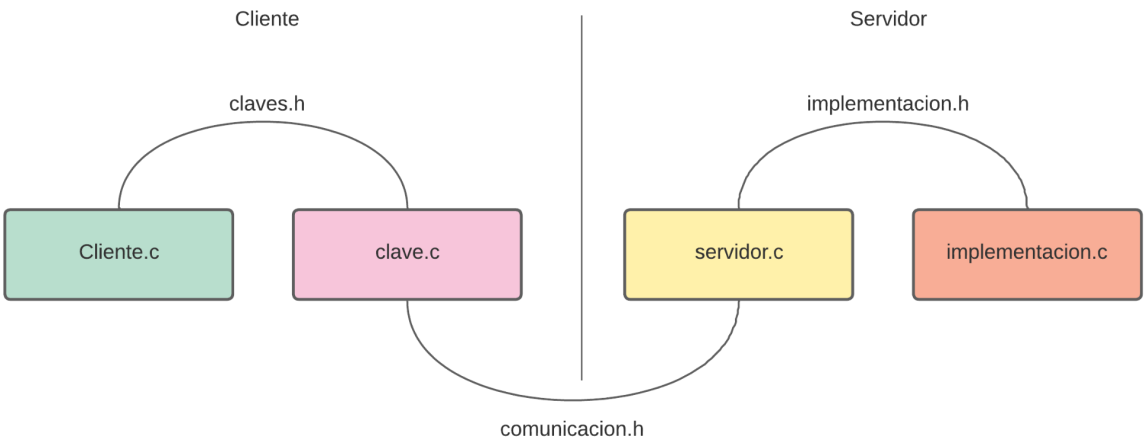

Figura 1. Diseño de la estructura del servicio distribuido
Fuente: Elaboración propia

En primer lugar, cabe recordar que en el fichero *comunicacion.h* es donde se ha definido una estructura *petición* y una de *respuesta* para poder realizar el paso de mensajes de manera satisfactoria. Dentro de la estructura de petición tenemos los siguientes campos: *clave (int), valor1 (char), valor2 (int), valor3 (double), código de operación (int)*. Por otro lado, dentro de la estructura de respuesta tenemos los siguientes campos: *clave, valor1, valor2, valor3, código de error (int).*

Implementación de sockets

Para poder utilizar sockets de manera satisfactoria, se ha tenido en cuenta la propia arquitectura de la máquina y de la red. Existen dos formas de ordenar los bytes en la memoria de una máquina o en una red: Little Endian o Big Endian. Como ya sabemos, el orden de bytes de la red está estandarizado y se define como Big Endian. Teniendo en cuenta que la arquitectura de la máquina puede ser diferente a la de la red (Little Endian), es necesario que el programa convierta correctamente los datos entre el orden de bytes del *host* y el orden de bytes de la red. Para ello, se han utilizado las funciones *htonl()* y *ntohl()* con el fin de garantizar una comunicación correcta y fiable entre la máquina y la red.

Además, ha sido necesario establecer la red y el puerto que se va a utilizar para poder comunicar el cliente y el servidor. Para ello, se ha hecho uso de las variables de entorno, tal

y como se especifica en el enunciado. Para ello se define en el cliente la función *obtener_var_entorno* que obtiene el número de puerto del servidor mediante la variable de entorno *PORT_TUPLAS*. Esta función es llamada posteriormente para establecer el puerto del servidor en el que se va a conectar el cliente.

Por otro lado, con el fin de tener un diseño independiente del lenguaje de programación, a la hora de tener que enviar la información a través de los sockets, no se han enviado directamente estructuras de C. Para poder mandar de forma correcta la información entre el servidor y el cliente, es necesario mandar cada campo de la tupla de forma independiente. Además, teniendo en cuenta la arquitectura de la red, hay que ser cuidadoso a la hora de mandar una variable de tipo *double*, como lo es en este caso valor3, ya que ocupa 64 bits en memoria y la capacidad máxima de la red es de 32 bits. Para ello, hemos pasado el double a tipo *string*.

En definitiva, el cliente se encarga de establecer la conexión con el servidor, enviar la petición y recibir la respuesta y de definir la dirección del servidor y conectarse con él a través de un socket. El servidor se encarga de realizar la conexión entre el socket del servidor y el socket del cliente (utilizando bind()) y quedando a la espera de que un cliente se conecte a él. Una vez haya recibido las peticiones, las leerá, procesará y enviará una respuesta al cliente.

Compilación

En cuanto a la compilación del proyecto hay algunas diferencias con respecto a la compilación de la anterior aplicación distribuida con la implementación de colas, ya que con la implementación de sockets es necesario definir una dirección IP y un número de puerto en el que estará escuchando el servidor.

Para evitar tener que definir unas direcciones IP y de puerto ya preestablecidas dentro del código del proyecto, se ha optado por darle la opción al usuario de definirlas en línea de comando y pasarlas como argumento a los ejecutables de *servidor.c* y *cliente.c.* Para ello deberemos compilar primero el proyecto con la herramienta *make (*más adelante se detalla el contenido del fichero *makefile)* y a continuación lanzar el servidor con la siguiente línea de comandos:

```C/C++
make
./servidor <port>
```

Donde `<port>` representa el número de puerto en el cual estará escuchando el servidor y al cual el cliente enviará sus peticiones. Cabe destacar que el servidor funciona correctamente en el puerto **4200.**

Para lanzar el cliente será necesario abrir una terminal nueva y antes de llamar al ejecutable del cliente deberemos introducir la siguiente línea de comando para determinar la fuente de la biblioteca:

```
C/C++
export LD_LIBRARY_PATH = $LD_LIBRARY_PATH: .
```

A continuación, para lanzar el cliente, será necesario especificar la dirección IP y el número de puerto al cual se enviarán las peticiones. Para ello se definirán las variables de entorno *IP_TUPLAS* y *PORT_TUPLAS* en la consola y se le pasarán al programa cliente de la siguiente manera con el comando *env var var1=value var2=value ./cliente*:

```
C/C++
env IP_TUPLAS=localhost PORT_TUPLAS=<port> ./cliente
```

Donde *IP_TUPLAS* será la dirección IP local, es decir *localhost* y el puerto será el mismo que hayamos definido en el servidor **(4200)** para que así se pueda establecer la comunicación

En cuanto al contenido del fichero *makefile,* aquí es donde está la generación de los ejecutables *./cliente y ./servidor,* los cuales se llevan a cabo compilando cada archivo de origen del cliente y servidor con el compilador y la biblioteca compartida libclaves.so.

En cuanto a la generación de la biblioteca dinámica libclaves.so, primero se compila cada archivo de origen de la biblioteca compartida, es decir el archivo *claves.c* con el compilador y la bandera “PIC”, y después se genera una biblioteca compartida con el compilador y las banderas de enlace.

Una librería dinámica son módulos que se cargan en tiempo de ejecución en lugar de estar vinculados en tiempo de compilación. Uno de los beneficios del uso de las librerías dinámicas es que hacen que el ejecutable sea más ligero. Además, ayuda a la mejora del rendimiento ya que como son cargadas en la memoria del sistema, permite una reducción en el tiempo de carga inicial.

Casos de prueba

Para comprobar el correcto funcionamiento del programa hemos decidido crear varios clientes y que cada uno tenga un comportamiento distinto. En la siguiente tabla describimos brevemente que hace cada cliente y los valores que reciben.

Caso de prueba	Descripción	Valor esperado	Valor recibido
Cliente 1	Este primer cliente inserta un elemento (***set_value(10, "mensaje", 2, 3.3)***), posteriormente lo modifica (***modify_value*(10, "modificación", 4, 4.4)**) e imprime el contenido de sus clave actualizada (***get_value*(10)**).	Se espera que la función get_value imprima los valores modificados. *set:* 0 *modify:* 0 *get:* 0 (valores: "modificación", 4 y 4.4	*set:* 0 *set:* 0 *modify:* 0 *get:* 0 (valores: "modificación", 4 y 4.4
Cliente 2	Este cliente se encarga de insertar un elemento ***set_value*(10, "mensaje", 2, 3.3)**), modifica su contenido ((***modify_value*(10, "modificación", 4, 4.4)**)) y lo copia a una nueva clave ***(copy_key(10, 15)***). Por último hace una llamada a (***get_value*(15)**) para imprimir el contenido de la clave la nueva clave a la que se han copiado los datos.	Se espera que el contenido de get_value de la clave 15 sea el mismo que el de la clave 10. *set:* 0 *modify:* 0 copy: 0 *get:* 0 (valores: "modificación", 4 y 4.4	*set:* 0 *modify:* 0 copy: 0 *get:* 0 (valores: "modificación", 4 y 4.4
Cliente 3	Inserta un elemento (***set_value(10, "mensaje", 2, 3.3)***), borra un elemento ya existente (***delete_key(10)***) y borra todos los ficheros dentro del directorio de peticiones (***init()***).	*set:* 0 *delete:* 0 *init:* 0	*set:* 0 *delete:* 0 *init:* 0
Cliente 4	Intenta crear un elemento ya existente (***set_value(10, "mensaje", 2, 3.3)***), intenta modificar un elemento que no existe (***modify_value*(20,**	*set:* -1 *modify:* -1	*set:* -1 *modify:* -1

	“modificación”, 4, 4.4)).		
Cliente 5	Se lanzan cliente y servidor en puertos que no coinciden	*connect to server: : Connection refused* *set_value(): código de error -1* *connect to server: : Connection refused* *get_value(): código de error -1* *[...]*	*connect to server: : Connection refused* *set_value(): código de error -1* *connect to server: : Connection refused* *get_value(): código de error -1* *[...]*