P2 Sistemas Distribuidos

Sockets

Sistemas Distribuidos
Alejandro Ladrón de Guevara Jiménez, NIA = 100499541
Fernando Mendizábal Santiago, NIA = 100495773

Índice

1. Introducción

En esta memoria expondremos el trabajo realizado en la creación de un sistema distribuido consistente en un sistema de sockets. En este caso, los mensajes no serán enviados en forma de estructuras de C, ya que deberemos usar tipos de datos serializados compatibles con otros tipos de máquinas. Este sistema, dependiente del archivo origen claves.h, estará formado por un usuario definido en app-cliente.c y proxy-sock.c, y un servidor definido por claves.c y servidor-sock.c. Las funcionalidades del sistema de tuplas están definidas en claves.c. Los archivos servidor-sock.c y proxy-mq.c definen el intercambio de mensajes cliente/servidor por sockets. Por último, app-cliente incluye las diferentes pruebas realizadas por parte del usuario.

2. Funciones claves.c

En este archivo se elabora la forma en la que actúan cada una de las funciones que ofrece el sistema al cliente, para manipular la base de datos. En el caso de nuestro sistema, el sistema de almacenamiento empleado será un archivo de texto nombrado “store.txt”. Las funciones realizan los siguientes pasos:

- *destroy()*: Elimina el fichero que incluye a todas las tuplas, simulando el borrado de estas.

- *set_value(key, value1, N_value2, V_value2, value3)*: Comprueba con la función exist(key) si existe la tupla con dicha clave, además de si el valor de N_value2 se encuentra entre 1 y 32. De darse cualquiera de las dos afirmaciones dará error. Si no existe y N_value2 se encuentra en rango, escribe línea a línea cada una de los valores introducidos siguiendo este patrón:

  Clave: %d
  ‘%s’ *Value1
  %lf %lf %lf…. *Value2: Tantos %lf’s como el valor de “N_value2:”
  Coordenadas: (%d, %d)
  \n

  En la parte final añade un final de línea (\n) que separa de forma visual cada tupla.

- *exist(key)*: Busca en cada línea, con el patrón “Clave: %d” si el número que representa %d es el de la key que se está buscando. Devuelve 1 si lo encuentra y 0 si no lo hace.
- *get_value(key, value1, N_value2, V_value2, value3)*: Busca con el patrón “Clave: %d” si existe en el archivo una tupla con la clave introducida. Si no se encuentra, se devolverá como error. Si se encuentra, siguiendo el patrón explicado en set_value se introducirán línea a línea los valores de la base de datos en las variables debidas pasadas en la función.
- delete_key(key): Se empleará un fichero auxiliar “temp.txt” para escribir el contenido de la base de datos. La función introducirá las líneas de las tuplas de “store.txt” a

“temp.txt”. Mientras, busca si existe la tupla con la key pasada. Si existe, en el proceso de escritura de un fichero a otro, las líneas de dicha tupla se ignorarán y, por último, se eliminará “store.txt”, renombrando con su nombre al fichero auxiliar. Si no existe, se eliminará “temp.txt”.

- modify_value(key, value1, N_value2, V_value2, value3): Esta función realizará una modificación en una tupla con una determinada clave, que consistirá en la eliminación de dicha tupla, con delete_key(key), y en su posterior inserción, con set_value(key, value1, N_value2, V_value2, value3).

Aparte de las funciones de claves.c, hemos creado otro archivo de funciones que son necesarias para el intercambio y lectura de mensajes cliente/servidor. Estas funciones están definidas en el archivo serial.c, y son compartidas gracias al archivo de origen serial.h:

- sendMessage(int sockfd, struct Peticion *p): Esta función se encarga de serializar y enviar los mensajes a través del socket. Convierte cada uno de los valores de un struct Petición, los valores a enviar, a tipo char, tipo de dato en ASCII que no sufre en un intercambio de máquina little-endian a big-endian, y viceversa. Estos son guardados en un buffer que definirá el mensaje a enviar por el socket, una cadena de caracteres con la información de cada valor, delimitados entre sí por el separador “|”. Además, los valores de V_value2 serán delimitados entre sí por comas. Como medida de seguridad para que no se produzcan errores, el value1 no puede contener dicho caracter “|”, por lo que será considerado un carácter especial y se devolverá la función como error en caso de que se encuentre en esta.

- recvMessage(int sockfd, struct Peticion *p): Esta función se encarga de recibir y deserializar el mensaje recibido a través del socket. Al igual que sendMessage(), esta recibe una Petición, que servirá como contenedor de los valores del mensaje recibido. Para ello, se recibe en un buffer la cadena de caracteres enviada por el socket y se pasará a cada parámetro del struct su debido valor. En el caso de “op” y “value1”, que son valores tipo char, se guardará todo caracter que sscanf() lea hasta que se encuentre el caracter “|”.

3. Servicio cliente/servidor

Los archivos proxy-sock.c y servidor-sock.c simulan un servicio cliente/servidor basado en intercambio de mensajes por sockets. En este, el cliente envía peticiones al servidor a través del socket, enviando la información de forma serializada en tipos de datos independientes de C. A continuación, el servidor procesa de la forma que corresponda la petición del cliente, deserializando el mensaje recibido. Por último, el servidor le devuelve el resultado al cliente con el mismo tipo de envío por socket, y posterior deserialización.

El archivo servidor-sock.c se encarga de simular el lado servidor, recibiendo y tramitando las peticiones. Al igual que proxy-sock.c, tiene acceso a las funciones de intercambio de mensajes por sockets, que son sendMessage() para serializar y enviar las peticiones y recvMessage() para recibir las peticiones serializadas, deserializarlas y guardarlas de nuevo

en la petición. Estas funciones se encuentran en el archivo serial.c y ambos pueden acceder gracias al archivo serial.h.

La conexión entre el servidor y el cliente se realiza a través de localhost con un puerto, definido en forma de variable de entorno para el cliente y como argv[1] para el servidor (./servidor <port>; argv[1]=<port>). El cliente recibe el valor del puerto de dicha variable y el servidor del argumento del mandato. Ambos lo pasan a int para poder emplearlo. En ambos se inicializa el socket con esta cifra como puerto.

Posteriormente, se hacen las comprobaciones debidas para el correcto envío de mensajes: comprobar que el socket está vacío, asociarle la IP (localhost) y el puerto debidos (<port>). Finalmente, permanece en escucha a la espera de peticiones y entra en un bucle infinito para atenderlas. Al recibir una petición, usa la función recvMessage() para deserializar el mensaje recibido y convertirlo a forma struct Petición, para que pueda ser procesado. Por cada petición recibida por socket, se creará un thread que procese esta petición. Dependiendo de la función definida se llamará a una u otra función de claves.c. Por último, los valores devueltos por la función son serializados y enviados de vuelta con la función sendMessage(). Para evitar condición de carrera entre threads, un thread bloquea el mutex al principio del tratamiento de un mensaje, y lo desbloquea justo al acabar de manipular el fichero que contiene las tuplas guardadas. Se ha considerado que ese momento es nada más ser devueltas las funciones del servicio de tuplas.

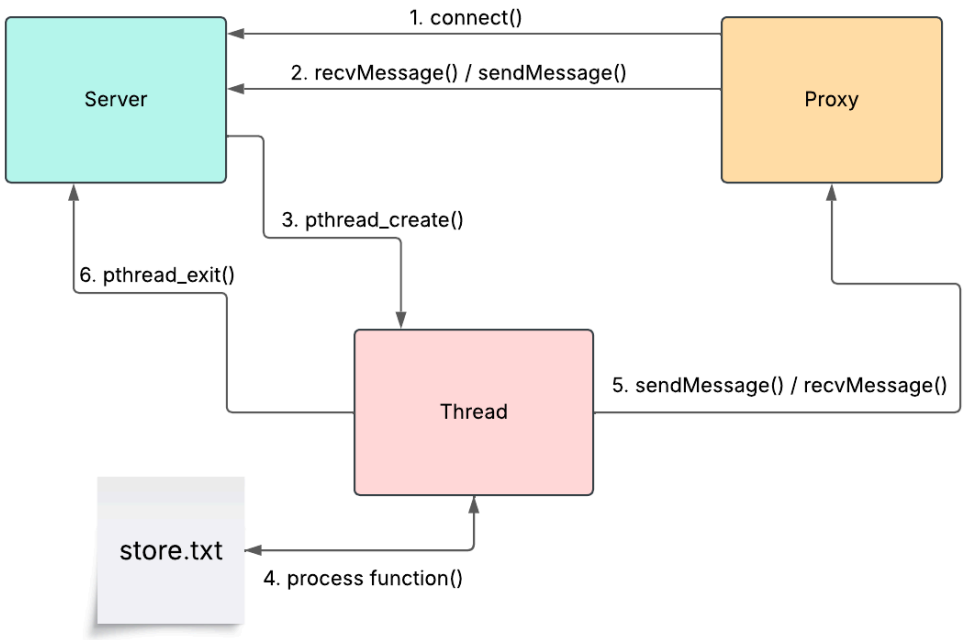
Por la parte del cliente, al ser llamada una función, se guardan los debidos datos en un struct, que gracias a sendMessage() serán serializados y enviados a través del socket. Después, esperará a recibir los datos devueltos por el servidor.

4. Makefile y ejecutables

Al igual que en la práctica anterior, se ha creado un archivo Makefile en el que se recogen mandatos encargados de crear los ejecutables del sistema. En él se especifica tanto la creación de la biblioteca dinámica libclaves.so como los diferentes permisos para que el sistema pueda ser ejecutado, añadiendo en cada ejecutable el archivo serial.c. Su uso consiste en la generación de dos ejecutables: el del servidor formado por claves.c, serial.c y servidor-sock.c; y el del cliente formado por app-cliente.c, serial.c y libclaves.so. Además, se han añadido los diferentes mandatos que activan los permisos debidos para cada funcionalidad del sistema.

El ejecutable del servidor tiene por nombre “servidor” y el del cliente, “cliente”. A la hora de probar el sistema, primero serán inicializadas en el terminal del cliente las variables de entorno IP_TUPLAS=localhost y PORT_TUPLAS=<port>, siendo <port> el número de puerto a emplear. Posteriormente, se llamará en otro terminal al servidor con ‘./servidor <port>’, siendo <port> el valor de PORT_TUPLAS. Finalmente, en el terminal del cliente nuevamente se ejecutará “./cliente”. Tras este paso final, se mostrarán en pantalla los resultados de las pruebas ejecutadas, recogidas en app-cliente.c. En el terminal del servidor se mostrará feedback sobre el correcto funcionamiento del sistema de tuplas, mientras que en el del cliente se mostrará el resultado devuelto por el servidor.

5. Diagrama de flujo



En este diagrama de flujo, exponemos el funcionamiento principal de nuestro sistema cliente servidor. Este consiste en el envío por parte de los clientes, cualesquiera, de peticiones al servidor. Por cada petición realizada, el servidor crea un thread que tramita dicha petición interactuando con la base de datos para finalmente enviar los resultados al cliente. Para cada petición realizada, cada cliente espera para recibir su resultado antes de realizar otra petición.

6. Batería de pruebas

Para la batería de pruebas hemos ideado cinco pruebas que demuestran el correcto funcionamiento de nuestro sistema. Dichas pruebas están definidas en app-cliente.c, y podrán ser probadas con el ejecutable del cliente. Los dos primeros escenarios consisten en la inserción, con set_value(), y posterior modificación, con modify_value(), de una tupla con una key específica. Los dos segundos escenarios consisten en las mismas pruebas pero realizadas con el uso de múltiples hilos, en concreto 12, que prueban la correcta concurrencia de nuestro sistema. El último escenario, llamado prueba extra, consiste en la inserción de 1000 tuplas a través de 1000 threads. Dicha prueba extra ha sido realizada tanto con el envío de estructuras de C por los sockets como de datos serializados por el mismo medio. Esto último se ha realizado para probar las diferencias de coste computacional entre ambos métodos.

En los dos primeros escenarios, se reinicia la base de datos borrándola con destroy(). para probar su correcto funcionamiento, se le ha añadido a cada uno que reciba con get_value() la tupla insertada, para posteriormente imprimirla por pantalla. Como podemos observar en

la pantalla del terminal al ejecutar el sistema ambos escenarios han obtenido resultados satisfactorios.

En los segundos escenarios, al igual que en el primero, se reinicia la base de datos con destroy(). Para enviar peticiones de manera simultánea, se definen threads que llamen a las funciones debidas. En este caso, en el primer escenario a set_value() y en el segundo a modify_value(). Primero, los thread introducirán tuplas con valores diversos pero keys diferentes. En la prueba posterior, para dichas keys se modificarán los valores de las tuplas. Como podemos observar por el terminal se han realizado las pruebas correctamente. Al abrirse el fichero “store.txt” podemos comprobar la correcta introducción y modificación de estas tuplas.

En el tercer escenario, al igual que en los dos anteriores, se reinicia la base de datos y se llaman a diversos threads para procesar peticiones. En este caso emplearemos 1000 threads. Al igual que en la prueba 3, cada thread insertará una tupla.

7. Extras y conclusiones

Como extra para esta práctica, hemos realizado una prueba para comparar las diferencias en cuanto a la eficiencia de nuestro sistema dependiendo de la estructura empleada para enviar los datos a través del socket. Compararemos la eficiencia entre emplear la estructura que implementamos en la práctica anterior, un struct de C, y el empleado en esta nueva, un buffer serializado.

Para ello hemos añadido una función el app-cliente al final del todo del código como comentario, llamada prueba_extra, en la que se crean 1000 threads para que envíen 1000 peticiones de set_value() al servidor, para posteriormente medir el tiempo que tarda cada tipo de estructura enviada en ser procesada por el servidor.

En el equipo en el que hemos realizado las pruebas, el tiempo ha sido de 0,17s para el método con struct y 0,20s para el del buffer serializado. Los tiempos pueden variar dependiendo de la máquina en la que se realice la prueba, pero podemos concluir que ambos tiempos son bastante similares. Esto nos indica que, aunque este factor no es determinante a la hora de decidir cuál de los métodos es mejor, se ve un mejor resultado en el método con struts que con el de serialización. Sin embargo, podemos concluir que dicho aumento del coste computacional es admisible dado las mejoras que ofrece este último método al sistema.

Para finalizar, nuestro sistema distribuido tiene características que lo convierten en un modelo eficiente en cuanto a su coste computacional. Esto se debe al uso de un fichero de texto como almacenamiento de las tuplas. El acceso y manipulación de este tipo de almacenamiento de datos es bastante simple, lo que supone una ventaja respecto a otros sistemas de bases de datos. Sin embargo, su acceso simple produce que pueda ser modificado de forma externa al sistema, corrompiendo o borrando los datos que contenga. Por ello, para un sistema pequeño como el que hemos creado esta forma de almacenamiento es idónea. En el caso de querer escalar este sistema, deberemos cambiar esta base de datos por otra más sofisticada.