P1 Sistemas Distribuidos

Colas de Mensajes

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 colas. En este, existen cuatro archivos dependientes de un archivo de origen llamado claves.h. Estos archivos son los siguientes: los referentes al cliente (app-cliente.c y proxy-mq.c) y al servidor (claves.c y servidor-mq.c). Las funciones de las diferentes funcionalidades del sistema se encuentran en claves.c. Los archivos servidor-mq.c y proxy-mq.c realizan el intercambio de mensajes cliente-servidor por colas. Y por último los app-cliente suponen las diferentes peticiones por parte de los clientes.

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
  Valor 1: ‘%s’
  Valor 2: %d
  %lf %lf %lf…. *Tantos %lf’s como el valor de “Valor 2:”
  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): En el sistema, la modificación de una tupla será considerada como eliminar la tupla con dicha clave y añadirla como una tupla nueva con la nueva información. Por ello, esta función consistirá en un delete_key(key) y en un posterior set_value(key, value1, N_value2, V_value2, value3).

3. Servicio cliente/servidor

Los archivos proxy-mq.c y servidor-mq.c simulan un servicio cliente/servidor basado en cola de mensajes. En este, el cliente envía peticiones al servidor a través de la cola de mensajes del servidor, de nombre "/100495773", NIA de uno de los integrantes del grupo. A continuación, el servidor procesa de la forma que corresponda la petición del cliente. Por último, el servidor le devuelve el resultado al cliente a través de la cola de mensajes del cliente. Cada cliente posee una cola diferente siguiendo este patrón: "/cliente_%d_%ld", donde %d es el ID del proceso y %ld el ID del hilo del cliente.

El archivo servidor-mq.c se encarga de simular el lado servidor, encargado de recibir y tramitar las peticiones. Al igual que proxy-mq.c, tiene definido el struct Petición, contenedor de los diferentes valores de cada petición: key de la tupla, variables de la tupla, tipo de operación, nombre de la cola del cliente y resultado.

Lo primero que realiza es la inicialización de los mutex, cond signals y threads correspondientes. A continuación, crea y abre la cola de servidor, entrando en un while infinito, a la espera de peticiones de cliente que recibirá por dicha cola. Al recibir una petición, creará un thread que ejecutará tratar_mensaje(void *mess), función encargada de procesar las peticiones que recibe como valor la petición. En tratar_mensaje(), se lee la operación a realizar, y dependiendo del valor definido en la Petición se llamará a una u otra función definida en claves.c. Por último, los valores devueltos por la función se introducen en esta Petición y se reenvían al cliente a través de su cola individual. En dicho thread, para no generar condición de carrera entre threads, se bloquea un mutex al principio de tratar_mensaje(), haciendo que no existan otros threads que interfieran en dicha operación. Tras terminar su función, el thread que está operando avisa al resto de que ha finalizado y desbloquea el mutex. Mientras no haya dado dicho aviso, el servidor espera con pthread_cond_wait() antes de crear nuevos threads.

El archivo proxy-mq.c se encarga de contener las funciones que le permiten a los clientes mandar las peticiones al servidor y comunicarse con él. Al igual que el servidor, lo primero que hace es crear la estructura Petición seguido de la inicialización de una variable "pet" de ese tipo de estructura. Luego, define para cada función los valores correspondientes de la estructura a enviar como petición. Finalmente, al final de cada operación se llama a la función cliente(), la cual se encarga de la comunicación con el servidor. Esta función, crea y abre la cola de cliente, abre la de servidor y envía el mensaje al servidor para justo después

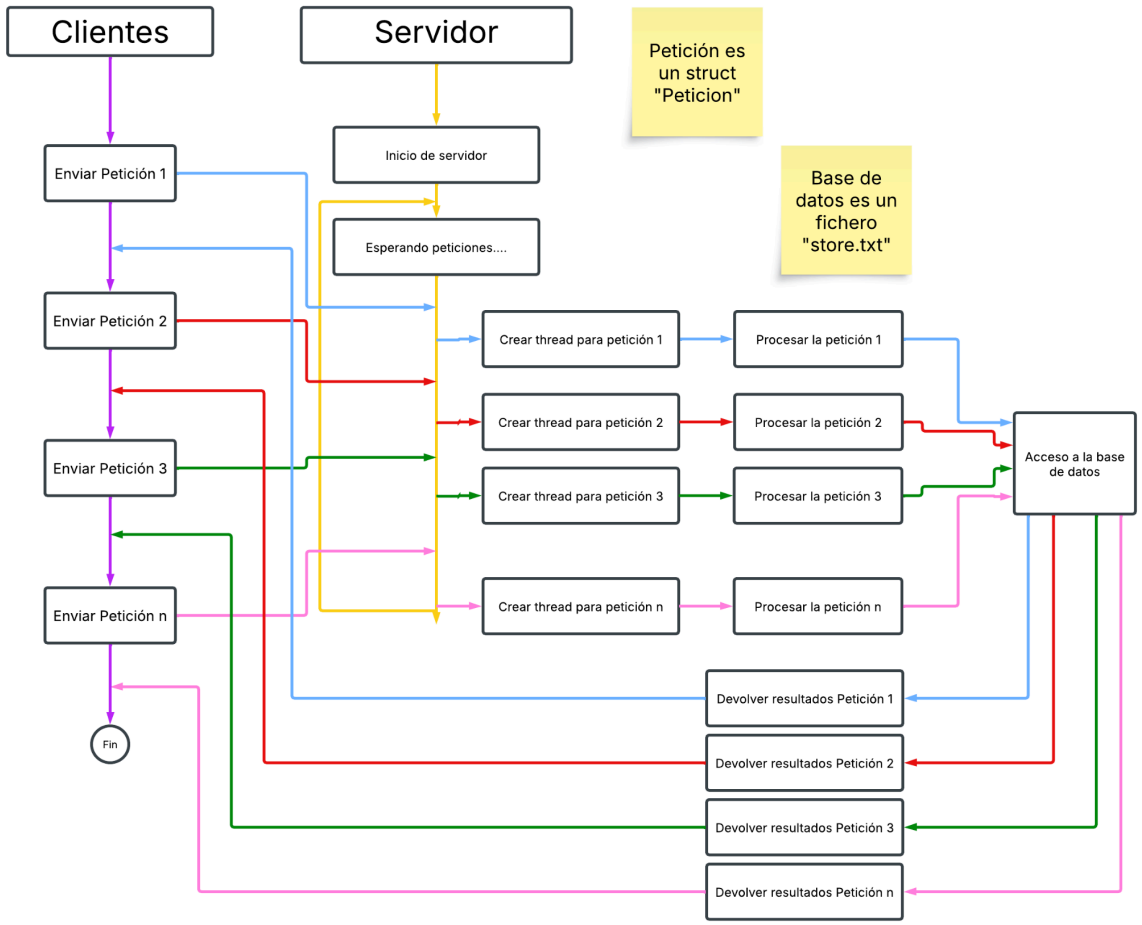
ponerse en modo de escucha esperando su respuesta. Una vez recibida la respuesta, muestra el error por pantalla en el caso de que lo hubiera habido y cierra las colas, borrando la del cliente y terminando el programa.

4. Makefile, libclaves.so y ejecutables

Para la correcta compilación de los ejecutables del servidor y de los clientes, se ha creado un archivo Makefile en el que se recogen los diferentes mandatos encargados de este trabajo. En él se especifica la creación de la biblioteca dinámica libclaves.so, que sirve como contenedor de las funcionalidades de proxy-mq.c. El uso de esta biblioteca es debido a que son independientes del ejecutable y sus funciones son empleadas por todos los archivos app-cliente.c. Posteriormente se crean tres ejecutables: el del servidor formado por claves.c y servidor-mq.c; el del cliente1 formado por app-cliente-1.c y libclaves.so; y el del cliente2 formado por app-cliente-2.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”. El del cliente1 tiene “cliente1” y el del cliente2 tiene “cliente2”. A la hora de probar el sistema, se llamará en una terminal al servidor con ./servidor. Posteriormente, en otra terminal se llamará a cualquiera de los dos ejecutables del cliente, tanto ./cliente1 como ./cliente2. En ambos casos se recibirá feedback sobre el resultado de las operaciones realizadas tanto en el servidor como en el cliente. Estos resultados se explicarán en la batería de pruebas.

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 dos escenarios en los que se prueban las diferentes funcionalidades del sistema. El primer escenario, definido en app-cliente-1.c y compilado en el ejecutable cliente1, consiste en el envío por parte de un cliente de peticiones al servidor que empleen todas las funciones definidas en claves.c, con el fin de comprobar que todas ellas funcionan correctamente. El segundo escenario, definido en app-ciente-2.c y compilado en el ejecutable cliente2, consiste en la generación de múltiples hilos, en concreto 12, que empleen la función set_value() para introducir una tupla cada uno, con el fin de probar la concurrencia del servidor.

En el primer escenario, cliente1, se reinicia la base de datos borrándola con destroy(). A continuación, se introduce una tupla con número de clave 4 y determinados valores para el resto de parámetros, gracias al uso de set_value(). Posteriormente, se modifican los valores de la tupla con dicha clave gracias a modify_value(). Por último, se emplea get_value() para acceder a los valores de la tupla modificada y se imprimen por la pantalla del cliente. Gracias a esta prueba podemos comprobar el funcionamiento de todas las funciones: destroy(); set_value(), en la que también se emplea exist(); modify_value(), en la que también se emplea delete_key(); y get_value().

El resultado que se obtiene al ejecutar el programa es el esperado. En el terminal de cliente se imprimen los valores obtenidos por get_value() de la tupla modificada. El servidor nos indica si el resultado de las operaciones ha sido satisfactorio, con un 0, o si ha dado error, con un -1. Además, imprime los valores pasados al cliente con get_value(). En la base de datos “store.txt” aparece la tupla con la clave indicada y los valores modificados.

En el segundo escenario, cliente2, al igual que en cliente1 se reinicia la base de datos con destroy() y se define la estructura ThreadData ya que este es el escenario concurrente. Después, se definen todos los valores de la estructura Petición que se introducirán en los distintos threads y se van introduciendo con un bucle for, para posteriormente crear los threads de la función thread_function(). Esta función se encargará de llamar a set_value() y realizar la petición para finalmente hacer join del thread y, una vez acabados todos, acabar el programa.

El resultado que se obtiene al ejecutar el programa es el esperado. En el terminal del cliente se indica si el resultado ha sido satisfactorio con un mensaje para cada thread y en el servidor se indica con un 0.

Aclaraciones:

*En ciertos casos se puede producir un error al usar destroy(), debido a que en nuestros app-cliente lo empleamos al principio para resetear la base de datos. Esto provoca que si no existía en un principio dicha base de datos, se imprima dicho error por pantalla.

*El servidor es concurrente y no genera condición de carrera porque las peticiones de los hilos se tramitan de forma individual sin solaparse, pero la introducción en cola de dichas peticiones es de carácter azaroso. Esto puede provocar que haya hilos que vean tramitada su petición antes que otros a pesar de haber sido creados posteriormente.

*El sistema ha sido compilado y ejecutado correctamente en el Aula Virtual Debian (Escritorio), también llamado “Guernika”.