P3 Sistemas Distribuidos

RPC

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 RPC. En este caso, los mensajes serán enviados en forma de estructuras de C. Este sistema, dependiente del archivo origen claves.h, estará formado por un usuario definido en app-cliente.c, tuplas_client.c y tuplas_clnt.c, y un servidor definido por claves.c, tuplas_server.c y tuplas_svc.c. Las funcionalidades del sistema de tuplas están definidas en claves.c. Los archivos tuplas_client.c y tuplas_server.c definen el intercambio de mensajes cliente/servidor con RPC. Por último, app-cliente incluye las diferentes pruebas realizadas por parte del usuario.

2. Funciones claves.c y archivos generados por rpcgen

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 un archivo de especificación RPC (Remote Procedure Call) con extensión .x. En este hemos definido las estructuras y funciones que se emplearán para el intercambio de mensajes entre cliente/servidor. Las estructuras definidas consisten en una estructura CoordAux, similar a Coord pero diferente nombre para que no exista solapamiento con la definida en claves.h; Petición, empleada para la entrada de datos en la mayoría de funciones; Respuesta, estructura contenedor para para la devolución del resultado; y RespuestaGetValue, estructura que devuelve valores de get_value(). Para crear el esqueleto del sistema RPC, usaremos rpcgen. El uso de este comando nos genera los archivos: tuplas.h que contiene las definiciones de estructuras y funciones comunes para cliente y servidor, tuplas_xdr.c que se encarga de serializar y deserializar los datos que se intercambian; tuplas_clnt.c, que contiene los procedimientos de las llamadas RPC por parte del cliente; y tuplas_svc.c, los procedimientos de las llamadas RPC por parte del servidor. Además, se han creado los esqueletos del cliente y servidor, que hemos renombrado como proxy-rpc.c y servidor-rpc.c.

3. Servicio cliente/servidor

Los archivos proxy-rpc.c y servidor-rpc.c simulan un servicio cliente/servidor basado en intercambio de mensajes por RPC. En este, el cliente envía peticiones al servidor, enviando la información a manipular de forma serializada gracias a las funcionalidades de serialización definidas en tuplas_xdr.c. A continuación, el servidor procesa las peticiones según las funciones definidas en claves.c. Por último, el servidor le devuelve el resultado al cliente, serializado nuevamente con tuplas_xdr.c.

El servicio cliente/servidor creado gestiona un sistema de llamadas remotas, el RPC. Gracias a este, el código de cliente accede a las funciones definidas en el servidor. Por la parte del cliente, tuplas_clnt.c se encarga de dar el soporte necesario para tramitar dichas llamadas remotas a estas funciones, enviando a su vez la información a manipular de forma serializada. Por la parte del servidor, tuplas_svc.c proporciona las funciones necesarias para recibir y procesar dichas llamadas a funciones, para posteriormente devolverle de la misma forma los resultados al cliente.

La conexión entre el cliente y el servidor se realiza utilizando el servicio de portmapper, que asocia los programas RPC a un puerto de comunicación. El cliente establece la conexión usando el nombre del host del servidor (localhost), que se define como una variable local tanto en el servidor como en el cliente. El servidor se registra automáticamente en el portmapper al iniciar su ejecución. El cliente crea un manejador de conexión mediante la función rpc_init(), la cual internamente se conecta al servidor por RPC. Esta conexión se

establece al principio de cada función para mantener una mejor organización. Al finalizar cada llamada a una función se invoca rpc_close() para liberar el manejador. En cuanto a la concurrencia del sistema, en rpc_init() se gestiona el acceso a la sección crítica, cuyo tamaño estará comprendido desde la conexión al servidor hasta su posterior desconexión. El resto de clientes esperarán el desbloqueo de la sección crítica con un cond_signal().

Una vez iniciado, el servidor se registra automáticamente en el portmapper, lo que da vía libre a que los clientes lo localicen utilizando el nombre del programa RPC y la versión especificados. Entonces, este queda a la espera de recibir peticiones remotas. Cada vez que llega una solicitud, el servidor deserializa los datos recibidos utilizando las funciones de tuplas_xdr.c, convirtiéndolos en una **`struct Peticion`**, e invoca la función correspondiente definida en el archivo tuplas_svc.c (como set_value_1_1_svc, get_value_1_1_svc, etc.), dependiendo de la operación solicitada.

Cada una de estas funciones toma como entrada los datos de la petición y procesa la solicitud. Una vez que la solicitud es procesada, el servidor prepara la respuesta, la serializa y la envía de vuelta al cliente.

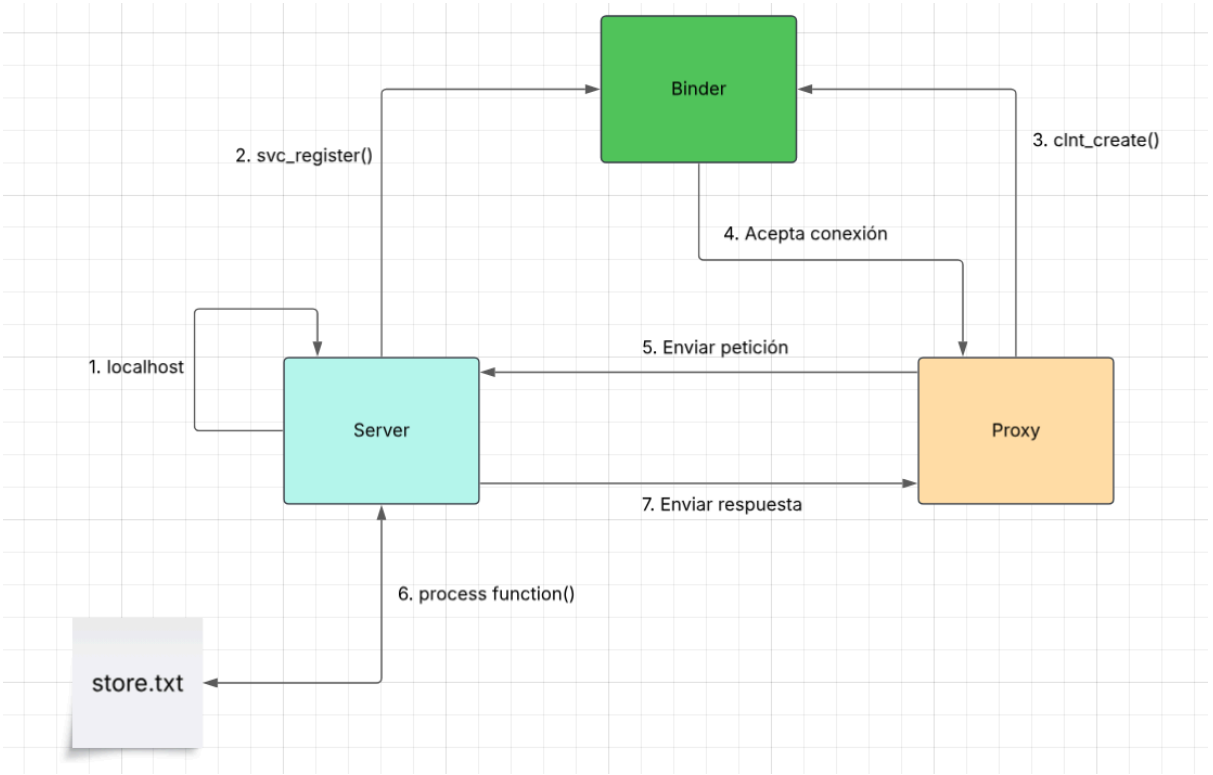
Por la parte del cliente, este inicia la conexión como se explicó anteriormente y después llama a la función correspondiente para la petición deseada. Las funciones inicializan una petición y rellenan los datos necesarios para ese tipo de petición en el **`struct Petición.`** Se manda la petición al servidor como se explicó previamente, este la procesa y se la devuelve al cliente.

4. Makefile y ejecutables

El archivo con extensión .x también ha generado un Makefile para la correcta compilación del sistema. En él se especifica tanto la creación de la biblioteca dinámica libclaves.so, como todos los archivos creados por rpcgen y los diferentes permisos para que el sistema pueda ser ejecutado. Su uso consiste en la generación de dos ejecutables: el del servidor formado por claves.c, servidor-rpc.c, tuplas_svc.c y tuplas_xdr.c y el del cliente formado por app-cliente.c y libclaves.so. La librería dinámica contiene tanto proxy-rpc.c como tuplas_clnt.c y tuplas_xdr.c. Además, se han añadido los diferentes mandatos que activan los permisos debidos para cada funcionalidad del sistema. Para compilar el sistema habrá que llamar al comando “make -f Makefile.tuplas”.

El ejecutable del servidor tiene por nombre “servidor” y el del cliente, “cliente”. A la hora de probar el sistema, primero será inicializada en el terminal del cliente la variable de entorno IP_TUPLAS=localhost. Posteriormente, se llamará en otro terminal al servidor con ‘./servidor. 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. En él, al iniciarse el servidor, este se registra al binder para que los clientes puedan conectarse a él. Al iniciarse el cliente, por medio del binder se intenta conectar con el cliente. Tras conectarse, se llama a la función remota y se le envía la información a manipular al servidor. Este procesa dicha información, manipulando los datos guardados en store.txt, y le devuelve el resultado al cliente.

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, consiste en la prueba del correcto funcionamiento de exist() y delete_key(), sin definir en las pruebas anteriores. También se ha dejado la prueba extra consistente en la inserción de 1000 tuplas a través de 1000 threads.

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 último escenario, simplemente se comprueba la funcionalidad de las funciones exist() y delete_key(). Primero se usa la función exist para comprobar la existencia de la key. Si existe, utiliza delete_key() y la borra, finalizando la prueba. Si no existe, lanza un error y finaliza.

7. Extras y conclusiones.

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.