Universidad
Carlos III de Madrid

Universidad Carlos III

Sistemas Distribuidos

Curso 2024-25

EJERCICIO 3

27/04/2025

Inés Fuai Guillén Peña 100495752@alumnos.uc3m.es | g82

Claudia Fernández Menéndez 100495919@alumnos.uc3m.es | g82

Universidad
Carlos III de Madrid

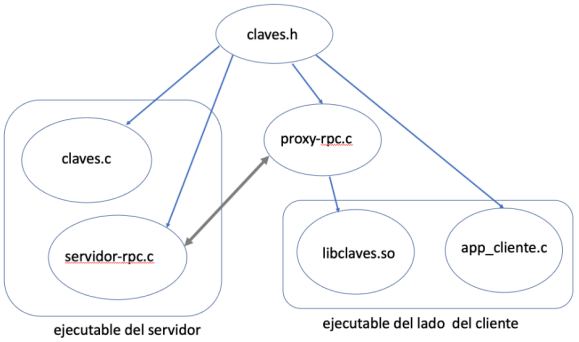
Introducción

El propósito de este ejercicio es desarrollar e implementar un servicio distribuido utilizando **RPC**. Este servicio permitirá el almacenamiento de tuplas de datos compuestas por una clave y múltiples valores asociados a ella.

Para lograrlo, se ha diseñado una aplicación que incluye un servidor responsable de gestionar y almacenar las claves, junto con una API que facilitará a los procesos cliente el acceso a los servicios proporcionados por el sistema.

Diseño

La aplicación debe seguir la siguiente estructura:



El archivo `claves.x` es el fichero principal que define la interfaz del sistema RPC. En él se declaran los tipos de datos utilizados en la comunicación y los procedimientos remotos que estarán disponibles para los clientes. Este archivo sirve como contrato entre cliente y servidor, y a partir de él se generan automáticamente varios archivos clave mediante la herramienta *rpcgen*.

Los principales ficheros generados son:

- `claves_clnt.c`: contiene las funciones que permiten al cliente invocar los procedimientos remotos definidos en `claves.x`
- `claves_svc.c`: implementa el esqueleto del servidor RPC. En él se encuentra el bucle principal del servidor y se registran los procedimientos definidos en el `.x`
- `claves_xdr.c`: contiene las funciones necesarias para la serialización y deserialización de los datos (marshalling y unmarshalling), basadas en *XDR* (External Data Representation). Esto permite que tanto cliente como servidor puedan intercambiar estructuras complejas de forma segura y consistente.

Proxy

El `proxy-rpc.c` será la API que permitirá al cliente comunicarse con el servidor. Es el responsable de ofrecer la interfaz a los clientes y se encarga de implementar los servicios (funciones) contactando con el servidor anterior. Aquí se implementarán las funciones que enviarán los mensajes correspondientes al servidor.

Servidor

En el archivo `claves.c`, se implementarán las funciones que se nos piden.

El `servidor-rpc.c` implementa los procedimientos del servidor que responden a las peticiones enviadas por el proxy (cliente). Este código se basa en los stubs generados automáticamente por *rpcgen*, pero también incorpora la lógica necesaria para invocar las funciones reales implementadas en *claves.c.*

Los archivos de proxy y servidor están relacionados mediante la cabecera `claves.h`, donde se definirán los prototipos de las funciones llamadas por el cliente.

Cliente

El `app_cliente.c` representará a un cliente, que solo llamará la función, o funciones, que quiera.

El archivo `libclaves.so` será la biblioteca que utilizarán las aplicaciones de usuario para usar el servicio.

Todos los ficheros de los clientes tienen en su cabecera `proxy-rpc.h`, donde se definen las funciones implementadas en `proxy-rpc.c`

Implementación

En primer lugar, desarrollamos el código de `claves.c`. Aquí creamos las funciones *set_value*, *get_value*, *modify_value*, *delete_key*, *exist* y *destroy* siguiendo los requisitos del enunciado. Considerando que las tuplas se almacenan en una lista enlazada, es importante señalar que la función *set_value* inserta la tupla recibida como parámetro al inicio de la lista. Esta decisión se tomó porque, no solo el orden de las tuplas del cliente no es relevante, sino también porque de esta forma es más eficiente.

En segundo lugar, implementamos `servidor-rpc.c`. Aquí, cada función *xxx_svc* (por ejemplo, *set_value_1_svc*, *get_value_1_svc*, etc.) recibe una solicitud procedente del cliente, interpreta los argumentos y llama a la función correspondiente de `claves.c`. El resultado se almacena en la variable de salida (**result*), que luego es devuelta al cliente. En el caso de funciones que devuelven estructuras más complejas, como *get_value*, se realiza la asignación dinámica de memoria y copia de los datos necesarios para que el cliente pueda recibirlos correctamente.

En cuanto al código del archivo `proxy-rpc.c`, aquí se implementan las funciones que servirán de API para el cliente para enviar las peticiones al servidor. Cada función del proxy se encarga de: primero, Inicializar el cliente RPC a través de *claves_prog_1*, obteniendo la IP del servidor desde la variable de entorno *IP_TUPLAS* para después construir la estructura de petición, con los argumentos proporcionados por el cliente y realizar la llamada remota correspondiente (****_1*, como *set_value_1*, *get_value_1*, etc.). Después, procesa la respuesta y devuelve el resultado al cliente y, finalmente, cierra el cliente RPC mediante *close_rpc_client()* para liberar recursos. Es importante destacar que en el caso de *get_value*, debido a que el cliente no necesita los valores *value1* y *value2* para seguir funcionando, se omite su uso en el proxy. Solo se copia el valor de *value3* si es necesario.

Finalmente, en los archivos `app-clientes-X.c` definimos los valores de la petición y llamamos a la función que queremos.

Compilación

Para la compilación del programa utilizamos un archivo *Makefile.claves*. Ejecutamos el comando *make -f Makefile.claves* en la terminal y se generan dos ejecutables: el ejecutable del servidor que implementa el servicio y el ejecutable de cliente obtenido a partir del archivo *app-cliente-X.c* (X haciendo referencia al número del cliente). También se genera la biblioteca dinámica *libclaves.so*.

Para ejecutar al cliente tenemos primero que exportar IP_TUPLAS usando el comando *export IP_TUPLAS=localhost*. Después, ejecutamos el servidor (*./servidor-rpc*) en una terminal, y el cliente (*IP_TUPLAS=localhost ./app-cliente-X*) en otra.

Casos de prueba

Hemos desarrollado varios clientes para realizar las pruebas de funcionamiento del sistema. A continuación, explicaremos con más en detalle en qué consiste cada cliente, el comportamiento esperado y los resultados obtenidos. Ninguno de los casos de prueba funcionará si el servidor no está activo en otra terminal.

Nota: Para ir revisando el correcto funcionamiento del código, hemos añadido mensajes de depuración tanto en la terminal del servidor como en la del cliente para saber dónde estamos en todo momento (CLAVES y SERVER para `servidor-rpc.c` *y PROXY para* `proxy-rpc.c`*).*

Cliente 1: Prueba básica de inserción y obtención (set_value, get_value)

Este cliente trata de insertar una tupla mediante *set_value,* para la que establecerá una *key* identificadora. Posteriormente, hará un *get_value* con dicho key para recuperar ese valor y consultar la tupla recién insertada.

Resultados: Podemos observar los dos accesos al servidor, que envía las respuestas que recibirá el cliente. Ambas operaciones se realizan correctamente.

Cliente 2: Prueba básica de modificación de valores

Este cliente realizará tres funciones: primero llamará a *set_value* para crear e insertar una tupla. A continuación, realizará cambios en la tupla recién creada mediante *modify_value*. Finalmente, realiza un *get_value* para recuperar la tupla y verificar que efectivamente se haya modificado el valor.

Resultados: Se observan los tres accesos al servidor y tres diferentes respuestas que devuelve el proxy. No hay ningún error inesperado, las operaciones se realizan correctamente.

Cliente 3: Prueba de concurrencia

Este cliente trata de poner a prueba el funcionamiento de nuestro sistema en términos de concurrencia. Simula la ejecución de dos clientes diferentes, que tratan de hacer una operación *set_value* y *modify_value* al mismo tiempo. Las operaciones irán dirigidas a valores de *key* diferentes, por lo que antes de lanzar los hilos de los clientes se hará un *set_value* con un valor de *key* 10, que es el que se usará por el cliente que quiere modificar.

Resultados: Primero se observa la ejecución del primer *set_value.* A continuación, si seguimos el orden de lo impreso por terminal, vemos cómo comienza la ejecución de *cliente1,* que aparentemente para y empieza *cliente2,* que vemos que llama al proxy para hacer el *modify_value. D*espués, parece ser que volvemos a *cliente1,* que finaliza su ejecución y ya después *cliente2* hace lo mismo. Tras varias ejecuciones, siempre obtenemos el mismo resultado.

Cliente 4: Prueba de eliminación de tupla

Este cliente creará una tupla con *set_value,* que posteriormente eliminará mediante el uso de *delete_key.* Finalmente, llama a *get_value* para comprobar si la eliminación de la tupla ha sido exitosa, en cuyo caso saltará un error con respecto a que la tupla a consultar no existe.

Resultados: En el lado servidor vemos dos intentos veces un error en *get_value_1_svc(),* siendo lo que sería la tercera iteración donde ya aparece el error en el cliente (-1). Esto indica el correcto funcionamiento de nuestro cliente, que ha eliminado una tupla que posteriormente está intentando

recuperar mediante el *get_value.* Como el cliente 7, su ejecución es más lenta que la de los demaś clientes debido al error forzado.

Cliente 5: Prueba de consulta de tupla inexistente

Este cliente trata de confirmar si una tupla que no existe existe mediante *exist.* Buscamos forzar que el resultado sea 0, lo cual significa que, en efecto, no existe dicha tupla.

Resultados: Dado que recibimos un valor distinto de 1 (que indicaría que la tupla existe), entramos en el condicional que nos devuelve *"\nLa tupla con la clave %d no existe\n", key.* Esto indica que la operación se ha realizado correctamente, saltando el error.

Cliente 6: Prueba de operaciones en claves múltiples

Este cliente hará dos *set_value* seguidos. Con él buscamos comprobar que dicha función funciona correctamente y que permite la inserción de varias tuplas.

Resultados: En efecto, vemos que se llama a *set_value* dos veces. Para la segunda, en el lado servidor, se puede observar que ambos se han realizado correctamente, pues en la lista final vemos el *[key]* de ambos.

Cliente 7: Prueba de error en el servidor (respuestas del servidor)

Este cliente trata de insertar una tupla mediante *set_value,* en la que el valor de *v2* es inválido para forzar un error. Con él, pretendemos depurar la detección de errores del propio *set_value.*

Resultados: Parecido al cliente 4, vemos dos intentos de realizar el *set_value,* y ya en tercero sería cuando se nos devuelve el error. Como el cliente 4, su ejecución es más lenta que la de los demaś clientes debido al error forzado.

Cliente 8: Prueba de eliminación de todas las tuplas

Usamos este cliente como un *hard-reset*, en el que simplemente llamamos a *destroy* para eliminar todas las tuplas creadas por los distintos clientes.

Resultados: En efecto, se destruyen todas las tuplas, por lo que podemos volver a generar tuplas nuevas con las keys de las antiguas y no saltaría ningún error.

Observaciones

Al ejecutar un cliente que crea una tupla, esta se mantiene siempre que el servidor permanece encendido. Esto implica que, por ejemplo, si ejecutamos *app-cliente-1* varias veces, la primera se hará correctamente pero las siguientes darán error, pues se intenta insertar una tupla que ya ha sido insertada.

Si llamamos a *app-cliente-8,* podemos volver a ejecutar a los clientes que insertan tuplas sin darnos error. Lo mismo ocurre si cerramos el servidor y lo volvemos a abrir, todas las tuplas que hubiera habrán sido eliminadas y nos dejará volver a insertar.