Sistemas distribuidos: ejercicio 3

RPC

Belén Gómez Arnaldo 100472037

Ana Maria Ortega Mateo 100472023

21 de abril de 2024

Descripción de la arquitectura.

La arquitectura de nuestro código se divide en cuatro partes fundamentales; archivo con la definición de la interfaz RPC, archivos del lado del servidor, archivos del lado del cliente y archivos comunes(cabecera y filtros XDR):

- Clave.x : Archivo de definición de la interfaz RPC que se utilizará para realizar llamadas a procedimientos remotos en el programa llamado CLAVE. Contiene, un número de programa, un número de versión del programa y el nombre y número de procedimiento. Los parámetros de los procedimientos vienen definidos en una estructura que en función del procedimiento se utilizará esta misma como parámetro de entrada o de salida.

Los archivos comunes que comparten la parte del cliente y el servidor son:

- Clave.h: Fichero de cabecera (.h) con los tipos y declaración de prototipos de procedimiento, que utiliza el servidor y el cliente para poder comunicarse.Utilizando las definiciones y las funciones establecidas en la interfaz.
- Clave_xdr: En este archivo se encuentran los procedimientos para el empaquetado y desempaquetado. Garantiza que los datos sean transmitidos e interpretados correctamente entre diferentes sistemas.

Los archivos por parte del servidor son los siguientes:

- Clave_svc(stub del servidor): Se encarga de las comunicaciones por parte del servidor. Este archivo se encarga de:
    - Recibir los mensajes de petición.
    - Desempaquetar los parámetros.
    - Invocar al procedimiento que implementa el servicio con los parámetros correspondientes a dicha función.
    - Recoger los resultados de la ejecución del procedimiento local.
    - Empaquetar el resultado en un mensaje de respuesta.
    - Enviar el mensaje al stub del cliente.

- Servidor: El servidor es el encargado de gestionar las peticiones de los clientes y se encarga de almacenar la información que se va añadiendo según las acciones que los clientes realizan. En

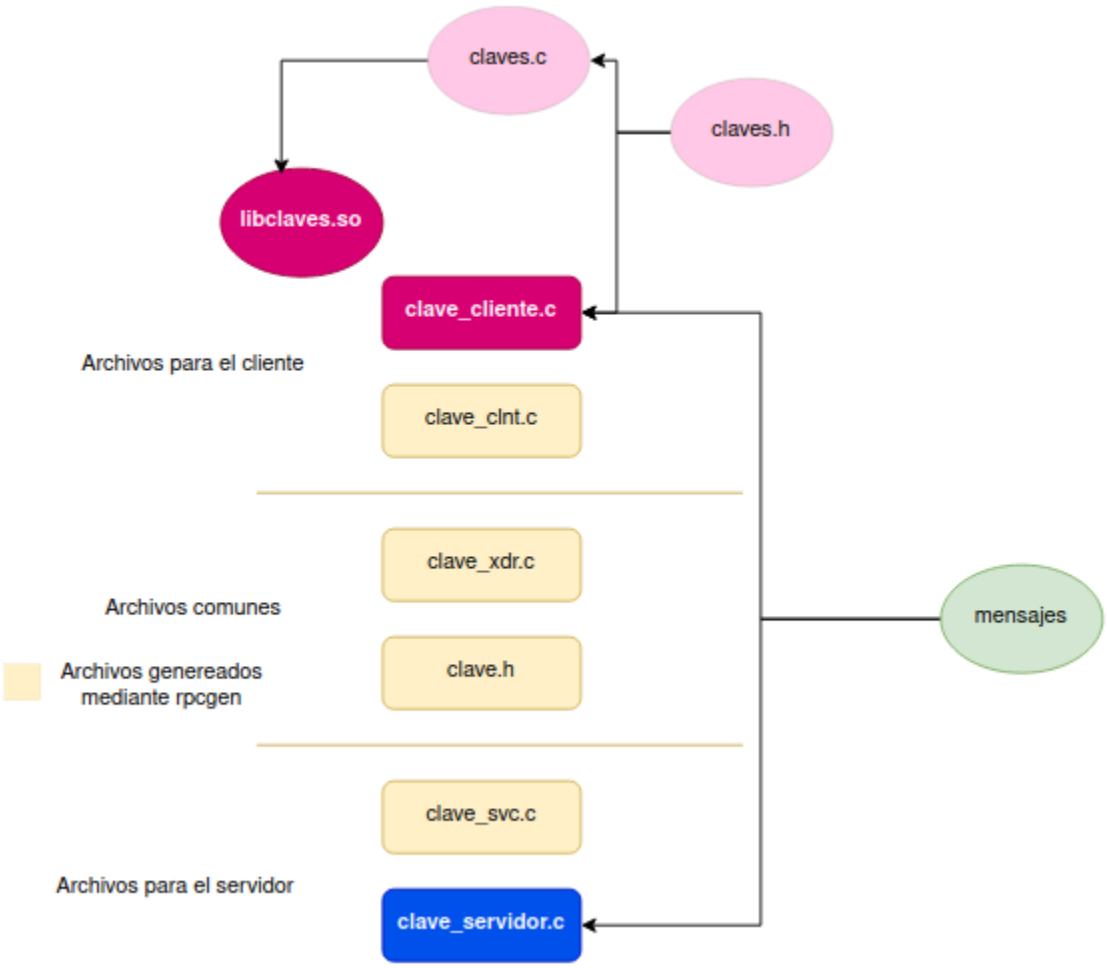
"servidor.c", se encuentra la implementación de todas las funciones. En vez de utilizar el archivo que genera la interfaz(clave_server.c), utilizamos nuestro "servidor.c", como el archivo del servidor con la implementación de los procedimientos, para la comunicación RPC.

Los archivos por parte del cliente:

- Clave_clnt(stub cliente): Se encarga de las comunicaciones por parte del cliente. Este archivo se encarga de:
  - Suplantar al procedimiento a ejecutar.
  - Localizar al servidor.
  - Empaquetar los parámetros y construir los mensajes.
  - Enviar el mensaje al servidor.
  - Recibir el mensaje de respuesta.
  - Desempaquetar los resultados del mensaje de respuesta.
  - Devolver los resultados a la función llamante.

- Cliente: El cliente está compuesto por una biblioteca ("libclaves.so"), por dos archivos de código ("clienteX.c" y "claves.c") y una cabecera de "claves.h" donde se definen todas las funciones que el cliente puede realizar. En "clienteX.c" no se implementa nada de código de las funciones, sirve solamente para que el cliente pueda escribir las funciones que quiere realizar, junto con los parámetros necesarios para realizar la función. En claves.c es donde se realiza la lógica de las funciones por parte del cliente y para que se pueda establecer la comunicación con RPC.Al igual que el servidor, no se utiliza el archivo del cliente generado por rpcgen, la lógica del código para la comunicación con RPC se encuentra en "claves.c". La biblioteca sirve para generar el ejecutable y así poder enlazar claves.c con el cliente.

El cliente y el servidor acceden a un archivo de "mensajes.c", donde se implementa la función para poder obtener la dirección IP del servidor. En la cabecera "mensajes.h" se incluye la definición de las funciones. Este es el diagrama de la arquitectura:



Descripción de los componentes.

Para el desarrollo de esta práctica, hemos partido del ejercicio evaluable 2 y hemos modificado la comunicación mediante sockets por RPC. Al sustituir los archivos del cliente y servidor generados por RPC por servidor.c y claves.c , utilizamos ese código generado en nuestros archivos. Nuestro código parte de tres componentes principales:

1. ClienteX.c : El cliente podrá introducir cualquier acción de las que están definidas y se ejecutará las funciones correspondientes a dicha acción.
2. Claves.c : Se encarga de manejar las acciones que realiza el cliente y mandarle la petición al servidor. Está compuesta por seis funciones diferentes; init, set_value, get_value, modify, delete y exist.

- Se definen las variables necesarias para cada función y en caso necesario, se realizan comprobaciones(del valor de N). También se declaran variables necesarias para la comunicación RPC por parte del cliente, definiendo por ejemplo, el estado de retorno, la entrada de los parámetros de la función o la respuesta.
- Se obtienen las variables de entorno con la función ObtenerVariablesEntorno.Esta función, obtiene la variable de entorno IP_TUPLAS y las guarda en la variable global definidas en mensajes.h (componente que comparten tanto cliente como servidor).
- Obtenemos la dirección IP con nuestra función ObtenerIP(), que simplemente devuelve la IP anteriormente almacenada en la variable global y la asignamos a la variable host, para definir la dirección del servidor.
- Creamos el cliente RPC, pasando, la dirección del servidor, el nombre del programa y de la versión(definidas en la interfaz) y utilizamos el protocolo “tcp”. Se ha decidido utilizar este protocolo en RCP porque proporciona fiabilidad y evita la saturación de red con un mecanismo de congestión y control de flujo.
- Se asignan los datos de entrada correspondientes a cada función.
- Se llama a la función remota correspondiente. Los resultados o el resultado que devuelve la función se guardan en la variable res. En caso de error se almacenará -1 y en caso de éxito 0.
- Se destruye el cliente RPC y se liberan los recursos asignados a dicha comunicación.
- Finalmente, se devuelve la respuesta al cliente.

3. Servidor.c : En servidor.c se realiza la implementación de funciones para gestionar las peticiones y almacenar las tuplas con memoria dinámica. Las funciones del servidor se ejecutan en caso de que haya habido una solicitud por parte del cliente a ese procedimiento. Dependiendo de la función que se ejecute se realiza una acción sobre la lista enlazada, se guarda en la variable de respuesta el contenido esperado por el cliente y se devuelve el estado de retorno(retval).

Se tiene en cuenta todos los errores posibles que se realizan durante todo este proceso(tanto en cliente como en servidor). Si sucede un error en el servidor, se devuelve en la respuesta al cliente un -1.

Ejecución.

Para compilar el código primero usamos el comando “rpcgen -NMa clave.x”. Esto generó los archivos necesarios para usar RPC con el servidor y el cliente. Sin embargo, no hemos usado el archivo de cliente que se genera automáticamente, sino que tenemos 3 clientes distintos. Antes de ejecutar el código hay que hacer el comando “make -f Makefile.clave”.

Para ejecutar el código del servidor solo hace falta poner por terminal el comando "./servidor". Para ejecutar los clientes, tras hacer el make, solo hace falta poner esta línea en la terminal: "env IP_TUPLAS=direccion_ip ./clienteX", siendo direccion_ip la dirección del servidor y X el número del cliente que se quiere ejecutar. Los códigos de los clientes se pueden cambiar para hacer diferentes funciones, nosotras hemos creado el código que consideramos adecuado para hacer una buena batería de pruebas. Si se quiere cambiar el código de los clientes hay que ejecutar después el comando "make -f Makefile.clave". Para ejecutar varios clientes de forma concurrente basta con usar el comando "env IP_TUPLAS=direccion_ip clienteX & env IP_TUPLAS=direccion_ip clienteX".

Pruebas.

Comportamiento secuencial

Para comprobar el funcionamiento secuencial del programa usaremos los archivos de *cliente1.c y cliente2.c* . Cuando se llama a cualquiera de las funciones se comprueba el valor que devuelven. En caso de que sea -1 se imprimirá un mensaje de error. Para comprobar el estado del servidor después de cada función se han incluído mensajes de error y mensajes para imprimir la lista cuando se modifica. De esta forma es fácil comprobar el estado del servidor y la lista. Se han hecho las siguientes pruebas, en cliente1 y cliente2:

- En cliente1 se comprueba todas las funciones(init, set_value, modify, get_value, delete y exist), comprobando que se realizan las funciones correctamente y se muestra por la terminal los valores esperados. Realizamos en primer lugar un init, después dos set values, comprobamos con get value la clave añadida anteriormente y comprobamos los valores devueltos. Hacemos un modify para comprobar que se cambian los valores en la lista enlazada y un exist para comprobar que la clave existe. Finalmente eliminamos una de las tuplas y comprobamos con otro exist que ya no se encuentra almacenada.
- En cliente2 realizamos pruebas para la depuración de errores y poder comprobar que son detectados correctamente.

Comportamiento concurrente

Para comprobar el comportamiento concurrente, lanzamos el primer cliente (cliente1) y hemos creado otro, en este caso cliente3, que se ejecutará de manera concurrente. Ambos realizan distintas acciones, si un cliente realiza el set de una clave, el otro puede acceder a ella mediante un get. De la misma manera, si un cliente elimina una clave, el otro cliente no podrá acceder a ella. En el enunciado no se especifica el orden exacto en el que se tienen que hacer las funciones de los clientes. No se especifica que se tenga que

intercalar o que se tenga que hacer primero todo un cliente y después el siguiente. Por ello, no hemos definido un orden concreto. Al compilar el código usamos el flag "-M" se genera un servidor multihilo, lo que significa que el servidor es capaz de gestionar múltiples clientes sin necesidad de utilizar mutex o variables condición. Tras evaluar el funcionamiento concurrente con este flag hemos decidido no utilizar mutex en las funciones ya que el servidor funciona correctamente de forma concurrente con este flag. Agregar mutex además del flad -M es una carga extra al servidor que no consideremos necesario porque con este flag el servidor ya puede manejar varios clientes de forma concurrente.

Conexión

Además de las pruebas para las funciones, hemos realizado las siguientes pruebas para comprobar errores de conexión o errores en las variables de entorno.

- No especificar la variable de entorno. Si ejecutamos el cliente con este comando "./cliente1.c", como no estamos definiendo la dirección ip del servidor, se produce un error en las variables de entorno y no se ejecuta ninguna de las funciones.
- Funcionamiento concurrente: si al ejecutar el código lo hacemos con este comando en los clientes "env IP_TUPLAS= localhost ./cliente1 & ./cliente3" también se obtendrá un funcionamiento incorrecto porque no se ha definido la dirección ip del servidor para el proceso del cliente3.

Al probar el código el nuestro ordenador funciona poniendo tanto la dirección ip como localhost. Sin embargo, hemos tenido problemas al probarlo en guernika. Hemos ejecutado el cliente y el servidor desde la misma dirección ip, por lo que poniendo como dirección localhost funcionan correctamente todos los clientes. Sin embargo, si ponemos la dirección ip directamente obtenemos errores de conectividad. Esto nos parece raro porque en nuestro ordenador sí funciona bien el código poniendo la dirección ip de la máquina.