MEMORIA EJERCICIO EVALUABLE 3

Universidad Carlos III

Grado Ingeniería Informática 2023-24

Procesadores del lenguaje (Grupo 81)

Práctica realizada por:

- **Tomás Mendizábal** (NIA: 100461170, Email: 100461170@alumnos.uc3m.es)
- **Alejandro Díaz Cuéllar** (NIA: 100472173, Email: 100472173@alumnos.uc3m.es)

Diseño elegido

Interfaz de servicio

Lo primero que queremos detallar a al hora de explicar el diseño del programa, ya que estamos usando RPC, es la interfaz de servicio, que hemos detallado en el archivo funciones.x. La interfaz diseñada tiene este aspecto:

```
C/C++
const MAX_SIZE = 256;
const MAX_VECTOR = 32;

struct peticion {
   int op;
   int key;
   opaque valor1[MAX_SIZE];
   int valor2_N;
   double valor2_value[MAX_VECTOR];
};

struct respuesta {
   int status;
   opaque valor1[MAX_SIZE];
   int N_value2;
   double valor2_value[MAX_VECTOR];
};

struct tupla {
   int clave;
   opaque valor1[MAX_SIZE];
   int valor2_N;
   double valor2_value[MAX_VECTOR];
};

program RPC {
   version RPCVER {
      int INIT() = 1;
      int SET_VALUE(struct peticion) = 2;
      struct respuesta GET_VALUE(struct peticion) = 3;
      int MODIFY_VALUE(struct peticion) = 4;
      int DELETE_KEY(int key) = 5;
      int EXIST(int key) = 6;
   } = 1;
} = 99;
```

Como vemos ha sido necesario definir un par de constantes que determinan el tamaño de ciertos elementos, además de un conjunto de estructuras de las que nos serviremos para poder encapsular los datos a transmitir durante la comunicación cliente-servidor, así como

una para poder almacenar los datos de forma eficiente en el almacén. El programa cuenta con 6 funciones, que implementan los servicios especificados en el enunciado.

Lado del cliente

El lado cliente se compone de un archivo generado a partir de la interfaz de servicio, funciones_clnt.c y dos archivos implementados por nosotros que son cliente.c y claves.c. Mientras que cliente.c es el propio cliente, para el que todo lo asociado con RPC es transparente y unicamente realiza llamadas a las funciones con los datos necesarios, claves.c se encarga de implementar la comunicación cliente-servidor mediante RPC realizando las llamadas a las pertinentes funciones de los archivos generados por rpcgen para comunicarse con el servidor.

Cliente.c se compone de una implementación de un cliente que realiza x operaciones al servidor aleatorias usando una semilla y eligiendo una clave al azar. Por su parte, en claves.c se encuentran implementadas las 6 funciones pedidas que realizarán las llamadas al servidor. Para ello en cada función lo que se hace es:

1. Se comprueba que los datos pasados como parámetros desde cliente.c son correctos (solo en algunas funciones).
2. Se recibe el nombre del host de la variable de entorno IP_TUPLAS
3. Se crea el cliente CLIENT *clnt que se conectará con el servidor
4. Se empaquetan, si es necesario, los datos pasados como parámetros para su correcto envío al servidor
5. Se realiza la llamada al servidor con el mensaje y se recibe el resultado en la estructura o variable pertinente, que es a su vez comprobado
6. Se cierra la conexión destruyendo el cliente
7. Se devuelve el resultado a cliente.c

Lado del servidor

El lado del servidor se compone en este caso de un archivo llamado servidor.c y de dos archivos más generados por rpcgen, funciones_svc.c y funciones_server.c, este último está implementado casi en su totalidad por nosotros.

El archivo funciones_server.c es el encargado de recibir los datos enviados desde claves.c y de preparar los datos para poder enviarlos al implementador de las funciones “reales” del servidor, que es servidor.c. Además es el encargado de recibir la operación realizada por el servidor y enviar los datos de vuelta al cliente de manera correcta, utilizando para ello el protocolo de RPC.
Pasemos ahora a explicar la funcionalidad del servidor como tal. No obstante, antes de ello vamos a detallar la forma de almacenamiento elegido y como se usa. Para el almacenamiento de las tuplas hemos decidido usar un archivo de texto binario. Sin embargo este archivo solo se usa para almacenamiento para alojar las tuplas cuando no está

funcionando el servidor, es decir, realmente el almacén está implementado mediante un almacén dinámico que volcará su contenido en el fichero cuando el servidor reciba una señal de cierre. Cuando se inicia el servidor todas las tuplas (si existe alguna) del fichero se cargan en un array dinámico de tuplas.

Para poder iniciar el servidor, el archivo funciones_server.c mantiene una variable global que indica si está o no inicializado, puesto que el cliente no tiene por qué realizar ninguna llamada de inicialización o inicio de la conexión. Por tanto en cada función de funciones_server.c (correspondientes a los 6 servicios del programa) se realiza una comprobación para saber si el servidor está activo y, si no es así, se llama a una función de inicialización.

En esta inicialización se inicializa el array dinámico “almacen” y se guarda el número máximo de elementos. Luego se llama a la función *load* que lee (y crea si es necesario) los elementos del almacen.txt. Adicionalmente, se asigna la señal SIGINT a la función *close_server*.

El servidor funciona mediante una función llamada tratar_petición, que recibe los datos procedentes de funciones_server.c y traslada los datos a una u otra función dependiendo de la operación que le indique la llamada proveniente de funciones_server.c. Todas estas funciones, al hacer uso del almacén, están correctamente sincronizadas gracias al uso de un mutex que protege el acceso y modificación de cualquier elemento del almacén.

En cuanto a las propias funciones, en init() se hace un lock, se libera el almacén y también se vacía el archivo de texto “almacen.txt”. Al final se hace un unlock del mutex, como en el resto de funciones. En el set value se empieza haciendo un lock. Se comprueba si ya existe la clave y si no es el caso se inserta en el almacén. Get_value() es sencilla, porque solo requiere de una interacción por el almacén y copia de valores si existe la clave. La función modify es similar a set, pero en vez de realizar una inserción, se modifica un valor del almacén. La penúltima función es delete. Esta función busca una clave (si existe) y la borra del almacén. Más adelante tenemos la función exist() que devuelve 0 si no existe la clave y 1 si existe, en otros casos devuelve error.

Tanto el cliente como el servidor se valen para su funcionamiento de otros dos archivos “nexo” generados por rpcgen a partir de la interfaz de usuario, que son funciones.h y funciones_xdr.c.

Pasos para compilar y ejecutar

A continuación se va a detallar una guía con los pasos a seguir para poder compilar el proyecto. El objetivo final será generar los ficheros ejecutables: “servidor”, “cliente” y “prueba_sec”. También se destaca que al ejecutar una petición usando cliente y servidor se va a crear un directorio llamado “data_structure”. Aquí se generará el archivo del almacén

llamado: “almacen.txt”. Por último se recomienda abrir dos terminales distintas: una para ejecutar el servidor y otra para ejecutar el cliente.

Los pasos para poder ejecutar con una sola terminal son:

1. Descomprimir el archivo tar.
2. Abrir una terminal.
3. Cambiar el directorio a “Ejercicio3_Distribuidos”.
4. Compilar con la sentencia: “make”.
5. Para ejecutar el servidor y un cliente secuencial se tiene que correr con el siguiente comando: “./servidor & ./execute_client.sh cliente”**. Este ejecuta el servidor en segundo plano y el cliente en primer plano.
6. Para terminar el proceso servidor usar el comando ps para encontrar el pid del servidor y luego correr “kill -s INT pid_servidor”

Los pasos para poder ejecutar con dos terminal son:

1. Descomprimir el archivo tar.
2. Abrir dos terminales.
3. Cambiar el directorio a “Ejercicio3_Distribuidos” en ambas terminales.
4. Compilar con la sentencia: “make”.
    a. “make” compila el proyecto
    b. “./servidor” en una terminal y “./execute_cliente.sh cliente” en otra.
5. Para ejecutar el servidor y un cliente secuencial se tiene que correr los comandos: “./servidor” y “./execute_client.sh cliente”** cada uno en en las

Los pasos para poder ejecutar las pruebas secuenciales:

1. Descomprimir el archivo tar.
2. Abrir una terminal.
3. Cambiar el directorio a “Ejercicio3_Distribuidos”.
4. Compilar con la sentencia: “make”.
5. Para ejecutar el servidor y un cliente secuencial se tiene que correr con el siguiente comando: “./servidor & ./execute_client.sh secuencial”**. Este ejecuta el servidor en segundo plano y el cliente en primer plano.
6. Para terminar el proceso servidor usar el comando ps para encontrar el pid del servidor y luego correr “kill -s INT pid_servidor”

PRUEBAS

Para las pruebas hemos dividido estas en dos grupos:

- **Pruebas de error** secuenciales, donde se evaluará el funcionamiento de las propias funciones y se verificará si estas funcionan correctamente. Es decir, que get_value coja correctamente los valores para la clave o que devuelva -1 si no existe, que exist devuelva correctamente si existe o no una tupla, que no se permita que la cadena

sea mayor a 256 caracteres, etc. Para ello simplemente se ejecutará un archivo con un único cliente que comprobará todos estos casos de prueba

- **Pruebas de concurrencia**. Para probar que el sistema funciona bien cuando se ejecutan muchos clientes a la vez hemos diseñado un pequeño programa al que llamamos “cliente aleatorio” que estará en el archivo cliente.c. Este programa elige de manera aleatoria una operación de entre las disponibles, realizando la operación. Queríamos simular el hecho de que en el sistema real los clientes van a realizar operaciones distintas y no queríamos estar haciendo un montón de archivos distintos con distintas operaciones. Para probar que funciona con muchos clientes ejecutamos directamente en la terminal un comando del tipo: for i in {1..10}; do ./cliente & done;. Hay que mencionar que este cliente aleatorio tiene dos modos de operación. Si se le llama sin argumentos el cliente utilizará una clave aleatoria, que simula mejor el caso real. Por otro lado, si se le llama con un 1 como argumento (for i in {1..10}; do ./cliente 1 & done;) entonces la clave será estática para todos los clientes y operaciones, lo que sirve para comprobar el correcto funcionamiento de la concurrencia y la atomicidad de las operaciones realizadas en el servidor.

** Se utiliza un archivo .sh para no tener que inicializar las variables de entorno manualmente.