UNIVERSIDAD CARLOS III DE MADRID

Práctica 2: Llamada a procedimientos remotos (RPC)

Sistemas Distribuidos

100077162 Jorge Barata González

100072936 Cristian Galán Galiano

Índice

Diseño del programa

El programa se ha implementado usando el código base ofrecido para el desarrollo de la práctica, que sigue una arquitectura cliente-servidor usando *Remote Proccess Call* (RPC). Al igual que la práctica anterior, el cliente realizará diferentes peticiones al servidor, que le da respuesta.

La aplicación está dividida en tres ficheros distintos: el cliente, el servidor y el XDR. A continuación detallamos el diseño seguido para cada uno de ellos, así como los servicios implementados.

XDR

Aquí se definen los servicios ofrecidos junto con los tipos de los parámetros y los retornos de las funciones: la interfaz del servicio. En la aplicación fichero se llama `text.x` y es usado por el generador `rpcgen` para crear las cabeceras tanto del servidor como del servidor. De éste modo comparten una descripción común de los servicios.

El fichero define un nombre para el programa, un conjunto de versiones y una serie de métodos para cada una de éstas versiones. En nuestro caso sólo tenemos una versión.

Programa: `SERVICIOPROG`

Versión: `SERVICIOVERS`

El objetivo de las versiones es garantizar la compatibilidad entre las antiguas y las nuevas.

A continuación se enumeran las estructuras usadas para los tipos de parámetros y retornos, así como la definición del servicio, versión y métodos:

Método	Parámetros	Retorno
PING	ip, puerto	*ack*
SWAP	ip, puerto, longitud, cadena	letras cambiadas cadena
HASH	ip, puerto, longitud, cadena	*hash*
CHECK	ip, puerto, longitud, cadena, *hash*	correcto
STAT	ip, puerto, ping, swap, hash, check, stat	-
QUIT	-	*Ack*

Cliente

El cliente inicia las peticiones, espera y recibe las respuestas del servidor usando *RPC*.

El programa recibe como parámetros la IP del servidor al que se desea conectar e intenta establecer conexión al principio de su ejecución. A diferencia de la práctica anterior el puerto al que se conecta no es configurable, siendo éste el usado por RPC: 111.

Se ha implementado con una interfaz por consola, en la cual el usuario introduce los mandatos con sus parámetros. Como se puede ver en el diagrama de la Figura 1, una vez conectada la aplicación lee el mandato por la entrada estándar, lo interpreta y realiza la petición, procesando la respuesta y mostrando la información pertinente al usuario a través de la salida de error estándar. Esto se repite hasta que se ejecuta el mandato `quit`.

Comparando el diagrama con el de la práctica anterior podemos ver que RPC nos permite abstraernos de los *sockets* y los *hilos*. En lugar de realizar la petición y esperar la respuesta explícitamente, realizamos una simple llamada a una función pasándole parámetros del tipo de las estructuras descritas en el fichero XDR.

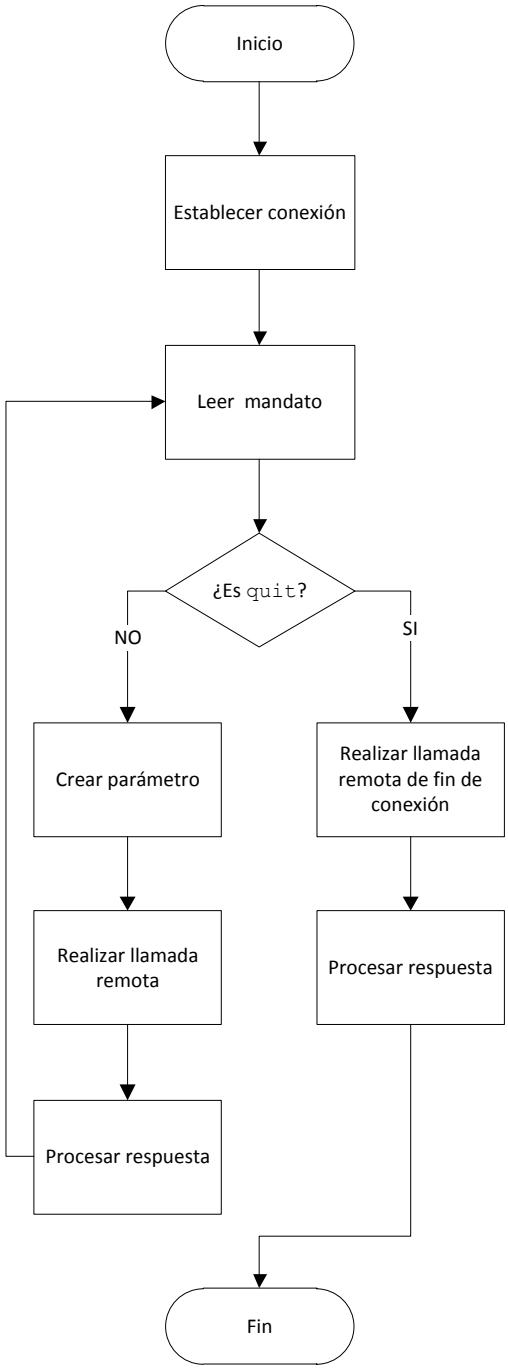

Figura 1: Diagrama de flujo del cliente

Servidor

El servidor es el receptor de las solicitudes del cliente. Su trabajo consiste en esperar y aceptar conexiones, esperar peticiones, procesarlas y devolver las respuestas. Al igual que el cliente usa RPC y el fichero XDR definido anteriormente.

A diferencia de la práctica anterior el puerto no es configurable, siendo éste el usado por RPC: 111.

Una vez corriendo, el servidor sólo usa la salida de error estándar para registrar las peticiones y no lee nada por la entrada estándar. Al iniciarse se queda escuchando esperando peticiones. Sin embargo, a diferencia de la práctica anterior nos abstraemos de los de los *sockets* y de la concurrencia. El fichero `server.c` simplemente define las funciones correspondientes a cada servicio definido en el fichero XDR, siguiendo un formato concreto que incluye la versión del programa utilizado.

El servidor tan sólo finaliza si sufre algún error, recibe una señal de interrupción o es matado. Entonces finaliza la conexión (Figura 2).

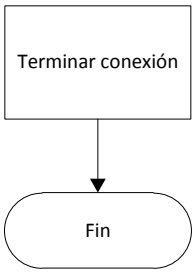

Figura 2: Fin de la ejecución del servidor

Servicios

Como en el enunciado ya se especifica la interfaz de los servicios ofrecidos tanto para el cliente como para el servidor, en éste apartado nos vamos a limitar a describir brevemente cómo se produce la comunicación desde que el usuario introduce un mandato hasta que recibe la respuesta desde el punto de vista de abstracción que nos ofrece **RPC**. Asumimos que el cliente ya ha establecido conexión.

1. El cliente recibe el mandato
2. El cliente comprueba que el mandato y sus argumentos son correctos
3. El cliente crea los parámetros
4. El cliente llama al método remoto con los parámetros creados
5. El servidor procesa la petición
6. El servidor envía la respuesta del método remoto al cliente
7. El cliente procesa la respuesta del método remoto

En la Figura 3 se puede ver el diagrama de secuencia:

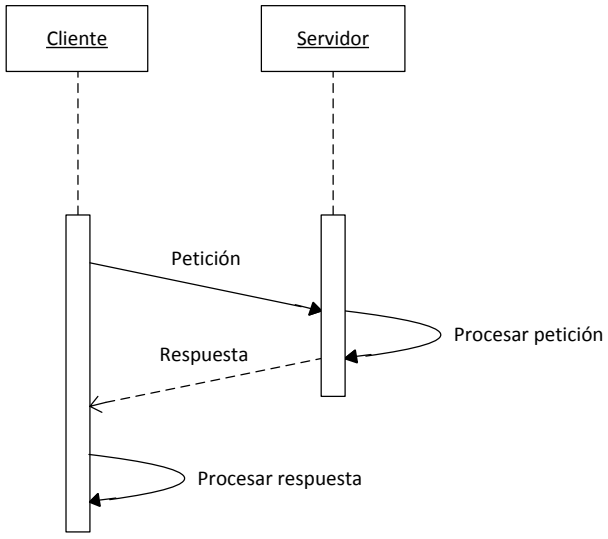

Figura 3: Diagrama de secuencia de la comunicación entre el cliente y el servidor

Esto se aplica para todos los mandatos.

Pruebas realizadas

Las pruebas realizadas son exactamente las mismas que en la práctica anterior. Aquí volvemos a resumir el algoritmo y mostramos el *script* anterior adaptado a la nueva práctica, pero nos hemos abstenido de volver a comentar la salida de ejemplo dado que es la misma.

Para la realización de las pruebas se ha usado el *script* de la Figura 3. Realiza lo siguiente:

1. Usa un artículo de la Wikipedia como fichero de pruebas.
2. Lanza el servidor en *background*, redirigiendo todas las salidas al fichero `server.out`.
3. Lanza cinco clientes en *background*, redirigiendo todas las salidas a ficheros diferentes para cada uno de ellos, llamados `client.out`. Para comunicarse con ellos posteriormente utiliza *named pipes*, comúnmente conocidas como *fifo*: son ficheros que mandan los datos recibidos de la entrada estándar directamente a la salida estándar. Imprimimos continuamente su contenido a la entrada estándar del cliente a través de *pipes*. Para enviar los mandatos basta con escribir en la *fifo* correspondiente al cliente con el que nos queramos comunicar.
4. Pedimos al primer cliente que calcule el *hash* del fichero usado para las pruebas para poder comprobar posteriormente que el *hash* es los servicios `hash` y `check` son correctos.
5. Ejecutamos una serie de mandatos para comprobar que todos funcionan correctamente. Lanzamos cada mandato como un proceso `echo` en *background*. De ésta forma provocamos una carrera entre los mandatos con el objetivo de simular un entorno real, con peticiones simultáneas en la medida de lo posible. En el *log* del servidor deberíamos apreciar como las peticiones han llegado en un orden diferente al que se lanzaron, así como peticiones que se han iniciado seguidamente antes de acabar las anteriores.

Hay que señalar que las solicitudes de establecimiento de conexión se efectúan secuencialmente. No las hemos incluido en la carrera de procesos del punto cinco dado que podrían mandarse mandatos antes de que los clientes se conectasen, llevando a error. Por la misma razón no hemos incluido el mandato `quit`, ya que los clientes podrían desconectarse antes de acabar todos los mandatos. Tuvimos que comprobar manualmente que los clientes podían conectarse y desconectarse en cualquier situación.

Al finalizar el script, tenemos los siguientes ficheros que nos sirven para comprobar que la aplicación funciona correctamente:

- Salidas de los clientes
  - `client.out.1`

    - client.out.2
    - client.out.3
    - client.out.4
    - client.out.5
- Fichero de prueba y fichero tras haber pasado por el servicio *swap*
    - file
    - file.swap
- Salida del servidor
    - server.out

```
#/bin/bash

SERVER=./server
CLIENT=./client
WORKSPACE="../test"
CLIENT_FILE="$WORKSPACE/file"
CLIENT_FILE_SWAP="$WORKSPACE/file.swap"
CLIENT_FILE_URL="http://en.wikipedia.org/wiki/Distributed_computing"
CLIENT_INPUT="$WORKSPACE/client.in"
CLIENT_OUTPUT="$WORKSPACE/client.out"
SERVER_OUTPUT="$WORKSPACE/server.out"

#Preparamos el workspace
mkdir -p $WORKSPACE
rm $WORKSPACE/*

#Descargamos un fichero de ejemplo
curl $CLIENT_FILE_URL -o $CLIENT_FILE

#Iniciamos el servidor
$SERVER &> $SERVER_OUTPUT &

# Usamos una fifo para cada cliente
for n in {1..5}
do
      mkfifo $CLIENT_INPUT.$n
      tail -f $CLIENT_INPUT.$n | $CLIENT -s 0.0.0.0 &>\
$CLIENT_OUTPUT.$n &
done
```

```
# Obtenemos el hash del documento con el primer cliente para\
comprobar las ejecuciones de check
echo "hash $CLIENT_FILE" > $CLIENT_INPUT.1 &
hash=`tail -n 1 $CLIENT_OUTPUT.1`

# Probamos los servicios.
commands=("ping" "swap $CLIENT_FILE $CLIENT_FILE_SWAP" "hash\
$CLIENT_FILE" "check $CLIENT_FILE 222" "check $CLIENT_FILE $hash"\
"stat")
for command in "${commands[@]}"
do
      for n in {1..5}
      do
            echo "$command" > $CLIENT_INPUT.$n &
      done
done

# Limpiamos las fifos
rm $CLIENT_INPUT*
exit
```

Figura 3: *Script* de pruebas

Conclusiones

Las conclusiones obtenidas consisten en las diferencias del uso de RPC frente a los *sockets*, librería que utilizamos en la práctica anterior. El servidor ofrece los mismos servicios en ambas prácticas, pero con implementaciones diferentes.

La capa de abstracción que ofrece RPC nos ha ahorrado muchísimo trabajo comparado con la implementación con *sockets*. No nos hemos tenido que preocupar de las peticiones, hilos y *sockets*.

Además, nos permite centrarnos en el protocolo del sistema, definiendo la interfaz en el XDR que además va a ser compartido tanto por el cliente como por el servidor.

También hemos podido apreciar que hay que seguir algunos convenios para usar RPC, como por ejemplo nombrar las funciones de una forma muy concreta o el puerto empleado. Los *sockets* ofrecen más flexibilidad, pero para éste tipo de programas ésta diferencia es prácticamente irrelevante frente a la cantidad de ventajas que ofrece RPC.

Tareas realizadas

Tareas	Horas dedicadas
Documentación sobre *c* y *rpcgen*	4
Creación de conexión inicial	4
Implementar el servicio *ping*	2
Comunicar el cliente con el servidor usando el servicio *ping*	2
Implementar el servicio *intercambiar letras*	2
Comunicar el cliente con el servidor usando el servicio *intercambiar letras*	2
Implementar el servicio *calcular función resumen*	2
Comunicar el cliente con el servidor usando el servicio *calcular función resumen*	2
Implementar el servicio comprobar *función resumen*	1
Comunicar el cliente con el servidor usando el servicio *comprobar función resumen*	2
Implementar el servicio *obtener estadísticas*	1
Comunicar el cliente con el servidor usando el servicio *obtener estadísticas*	2
Pruebas	4
Memoria	6