Diseño e implementación de un sistema peer-to-peer

Sistemas Distribuidos

Sergio Arroyo Martín 100383378
Alejandro de la Cruz Alvarado 100383497
Daniel de Frutos Díez 100383504

Índice

Índice **1**

Introducción **2**

Diseño **3**

Bases de datos 3

Interacción 3

Funciones 4

Extras 7

Pruebas 7

Información adicional para la compilación y ejecución de los programas 10

Parte 2 **11**

Generación de RPC 11

Pruebas 12

Información adicional para la compilación y ejecución de los programas 15

Parte 3 **15**

Conclusión **15**

Introducción

Esta práctica consiste en el desarrollo de una aplicación peer to peer para compartir archivos, al estilo Napster.

Para ello hemos creado un servidor en C que administra los usuarios con una base de datos hecha en sqlite y a su vez resuelve peticiones de los usuarios como listar los usuarios existentes o publicar archivos.

El cliente, hecho en Java, realiza las funcionalidades principales, como conectarse, publicar archivos, o incluso mandar y recibir archivos de otros usuarios.

La conexión entre ambos agentes es mediante sockets.

El servidor es concurrente creando un thread independiente para atender a cada petición de los clientes, para poder dar servicio a varios clientes distintos al mismo tiempo. Para proteger las operaciones sobre la base de datos utilizamos un mutex.

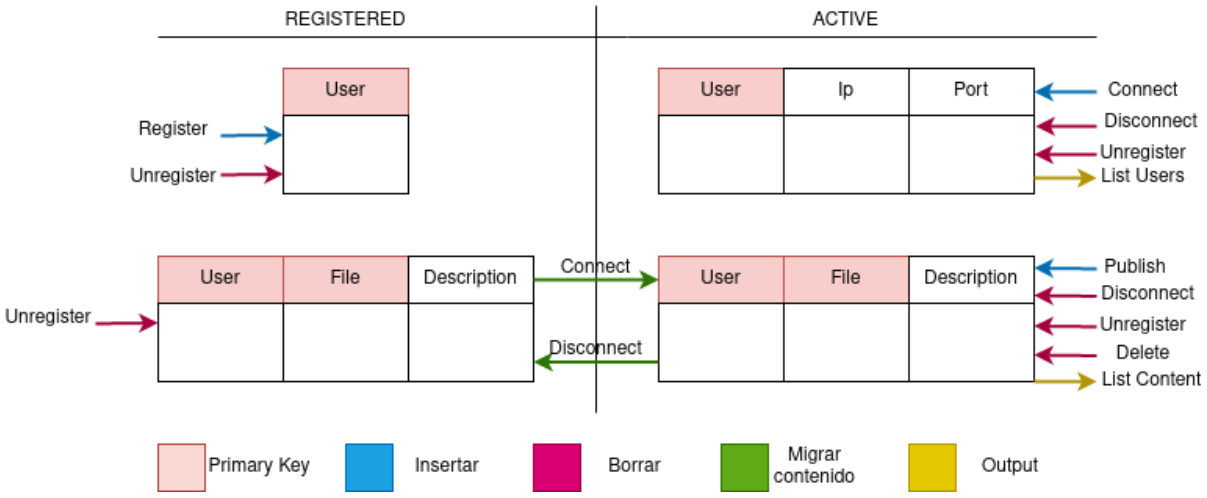
Diseño

Bases de datos

Decidimos utilizar bases de datos antes que almacenamiento en ficheros porque nos parece una solución mucho más formal, elegante y fiable y además nos facilita la realización de las siguientes partes de la práctica.

Las bases de datos están hechas con sqlite, como nos fue recomendado por el profesor, por tener buena integración con un entorno Linux y por ser la que conocíamos.

Nuestro diseño es el siguiente:



Decidimos hacer dos bases de datos en vez de una para tener en una toda la información de los usuarios existentes, a modo de historial, y en otra la información más relevante, la que pueden utilizar los usuarios. Además, este diseño nos da la posibilidad de que cuando un usuario se vuelva a conectar mantenga los mismos archivos publicados que cuando se desconectó.

En el diagrama mostrado anteriormente se puede ver qué operaciones afectan a qué tablas y bases de datos, las claves primarias elegidas y los movimientos entre las tablas.

Interacción

La interacción es básicamente la descrita en el enunciado.
Diferentes usuarios en diferentes máquinas se pueden conectar al sistema y mandar al servidor cadenas de caracteres con las diferentes operaciones disponibles a través de una conexión con sockets. El servidor hará las consultas correspondientes a las bases de datos y devolverá por el socket el resultado de la operación deseada utilizando el mismo formato de cadenas de caracteres.

Una posible interacción entre el cliente y el servidor es la siguiente:
C> PUBLISH memoria.pdf Memoria documentando el funcionamiento de la práctica
Socket: PUBLISH
Socket: user
Socket: memoria.pdf
Socket: Memoria documentando el funcionamiento de la práctica

Servidor:

1. Hacer comprobaciones pertinentes (explicadas en el apartado siguiente)
2. Insertar en la base de datos de activos y tabla archivos el archivo memoria.pdf del usuario user con la descripción proporcionada
3. Devolver resultado de la operación

S> PUBLISH OK
Socket: 0

El servidor guarda información del sistema, de sus usuarios y los archivos que hacen públicos cada uno de ellos pero los ficheros los almacenan los clientes en sus directorios personales. Para poder acceder a los ficheros, los clientes se tienen que conectar entre ellos, de ahí que sea una aplicación peer-to-peer.

Funciones

Para gestionar el funcionamiento del sistema, contamos con una función de comunicación en el servidor, ejecutada en threads independientes, la cual, se encarga de tratar las llamadas de los clientes dependiendo de cual realicen estos. Para las llamadas del cliente, en su fichero client.java contamos con una función para cada llamada, las cuales se encargarán de enviar al servidor la cadena con la respectiva operación y los parámetros necesarios para su tratamiento. Para el caso de la función get_file, hemos implementado un hilo servidor java para los clientes conectados, que se encargará de recibir las llamadas GET_FILE y transferir los ficheros solicitados a dicho cliente. Por último la librería operations se encarga de tratar en la base de datos las diferentes llamadas que realizan los clientes. El sistema cuenta con las siguientes funciones:

OPERACIÓN	CLIENTE	SERVIDOR	OPERATIONS
REGISTER	register()	if(strcmp(buf, “REGISTER”)==0)	registerUser()
UNREGISTER	unregister()	if(strcmp(buf, “UNREGISTER”)==0)	unregisterUser()
CONNECT	connect()	if(strcmp(buf, “CONNECT”)==0)	connectUser()
DISCONNECT	disconnect()	if(strcmp(buf, “DISCONNECT”)==0)	disconnectUser()
PUBLISH	publish()	if(strcmp(buf, “PUBLISH”)==0)	publishFile()
DELETE	delete()	if(strcmp(buf, “DELETE”)==0)	deleteFile()
LIST_USERS	list_users()	if(strcmp(buf, “LIST_USERS”)==0)	list_users()
LIST_CONTENT	list_content()	if(strcmp(buf, “LIST_CONTENT”)==0)	list_content()
GET_FILE	get_file()	NO PARTICIPA	NO PARTICIPA

startServer: Esta función auxiliar es la encargada de cargar las bases de datos en caso de que no existieran previamente, se ejecuta al inicio del main del servidor. Crea las bases de datos descritas anteriormente con las tablas explicadas incluyendo las políticas deseadas sobre los elementos, como ON DELETE CASCADE, que hace que cuando se elimine un usuario se eliminen también sus ficheros asociados, esto reduce ligeramente el número de operaciones que tenemos que hacer sobre las bases de datos.

stopServer: Esta otra función auxiliar protege las bases de datos, se llama en caso de cerrar forzosamente la ejecución del servidor para mantener la consistencia de los datos, ya que de otra forma los usuarios conectados en el momento del cierre se mantendrían cuando el servidor vuelva a estar levantado. Simplemente vuelca todos los datos de la base de datos de usuarios activos en la de usuarios registrados.

REGISTER: Esta es la primera función en implementar la interacción descrita anteriormente. Primero el cliente manda la cadena REGISTER seguida del nombre de usuario que desea registrar. Cuando el servidor detecta la operación REGISTER llama a la función registerUser de la librería operations. Esta función básicamente se conecta a la base de datos de registrados, comprueba si el usuario existía previamente y en caso contrario inserta en la tabla de usuarios con el nombre del nuevo usuario y devuelve cómo ha ido, siendo 0 en caso correcto y 1 en caso de que el usuario ya existiera.

UNREGISTER: Para este caso las funciones del cliente y el servidor son exactas respecto a register. La librería, sin embargo, cambia completamente. Primero abre las bases de datos y activa las claves ajenas (para aprovechar la semántica ON DELETE CASCADE). Comprueba que el usuario existiera y en caso afirmativo borra el usuario de la tabla usuarios en ambas bases de datos, al tener un borrado en cascada también se borrarán los archivos asociados.

CONNECT: Para conectar a un usuario el cliente manda las cadenas de la operación, el usuario que se conecta, su ip y un puerto disponible. El servidor le pasa a la librería los datos recibidos por el socket y esta, tras comprobar que el usuario exista y no esté ya conectado migra los ficheros del usuario a la base de datos de usuarios activos e inserta al usuario en la tabla de usuarios activos junto a la ip y puerto recibidos. Si todo ha ido bien el cliente activa también el hilo de servidor, a la espera de solicitudes de otros clientes.

DISCONNECT: Esta función es opuesta a connect, ya que el cliente cierra el hilo servidor y manda la operación y usuario al servidor, quien migra los archivos a la base de datos de usuarios registrados y borra todos los datos del usuario de la base de datos de usuarios activos. Tal como se pedía en el enunciado, si al ejecutar esta función se produce algún inconveniente, se forzará la desconexión del usuario.

PUBLISH: Mediante esta función, un cliente publica un fichero local junto con una descripción del mismo. Primero en el cliente, se comprueba si el cliente está conectado y si el fichero existe. En caso de que ambos se cumplan, el cliente envía al servidor la cadena PUBLISH seguida de su nombre de usuario, el nombre del fichero y la descripción. Con los datos anteriores, el servidor llama a la función publishFile de operations, que se encargará

de añadir los ficheros asociados al usuario a la base de datos, devolviendo el código de error según como se haya realizado la operación. 0 si todo OK, 1 si el usuario no existe, 2 si el usuario no está conectado y 3 si el contenido ya ha sido publicado anteriormente.

DELETE: Esta función equivale a lo contrario de la anterior. De la misma manera comprueba si el usuario está conectado. En caso afirmativo envía la cadena DELETE seguida del nombre de usuario y del nombre del fichero a eliminar. El servidor recibe lo anterior y se encarga de llamar a la función deleteFile de operations, en la cual se eliminará el fichero solicitado junto con su descripción de la base de datos. Tras ejecutarse la función, se devolverá 0 en caso de que todo se haya realizado OK, 1 en caso de que el usuario no exista, 2 en caso de que el usuario no esté conectado y 3 en caso de que el contenido no se haya publicado.

LIST_USERS: Esta función se utiliza para conocer todos los usuarios conectados al sistema en el momento en el que se realiza la operación. Como resultado, el cliente recibe una lista con los usuarios, ips y puertos que según el enunciado de la práctica, deben ser impresos por pantalla. Además, el programa cliente aprovecha esta información y la almacena en un ArrayList (connectedusers_) que posteriormente, la función getfile empleará para conectarse con otros usuarios. El servidor recibe la petición del cliente y llama a la función list_users() de la librería operations. En operations, primero se abre la base de datos de activos, después se relaciona la base de usuarios activos a la de registrados y se comprueba si el usuario existe. Una vez hecho esto se comprueba si el usuario está conectado y en caso de estarlo se creará una lista con los nombres de los usuarios y sus respectivas ips y puertos. En nuestro caso queríamos que la función retornase un código de error en caso de fallar, y en caso correcto, tanto el número de usuarios como una lista de los mismos, por lo que decidimos implementarla de forma que devolviese el código de error en negativo en caso de error y en el caso de que se haya ejecutado correctamente nos devolviese el número de usuarios conectados. La lista se crea en operations, pero se libera su memoria (free) en el servidor. También es el servidor el que traduce los códigos de error de negativo a positivo y se los pasa al cliente.

LIST_CONTENT: El fin de esta función es mostrar el contenido publicado por otro usuario. Para hacer esto, el cliente envía al servidor el nombre del usuario que realiza la petición y el usuario del que se quiere obtener los contenidos. Como siempre, el servidor recibe estos datos y llama a la función list_content() de la librería operations para que realice la operación. Al igual que en el caso de list_users(), esta función hace las comprobaciones requeridas en la base de datos y posteriormente devuelve un código de error en negativo o, en caso de éxito, nos devolverá el número de ficheros y creará una lista con el nombre de los ficheros y sus descripciones. En este caso, la lista también es destruida en el servidor y los códigos de error son traducidos de negativo a positivo en el mismo, devolviendo al cliente los códigos de error, el número de ficheros y la lista con los mismos tal y como se nos especifica en el enunciado.

GET_FILE: Un cliente solicita un fichero publicado por otro cliente mediante esta función. Para poder hacer get_file el cliente destino debe estar conectado y el cliente que lo solicita debe haber hecho list_users previamente para poder conocer las direcciones y puertos de

los usuarios conectados. Cuando un usuario se conecta, inicia su hilo servidor, que se encargará de recibir las llamadas GET_FILE para enviar el fichero que se le haya solicitado. El hilo servidor comprueba previamente que el fichero exista, cancelando la operación en caso contrario. El cliente origen recibe este fichero y lo almacena en otro fichero propio. Para copiar ficheros a través del socket, hemos empleado la función copyFile que se nos proporcionó para la implementación de la práctica, adaptándola de forma que uno de los extremos corresponda al socket y el otro al fichero. Esta transferencia se realiza entre un cliente y el hilo servidor de otro cliente, por lo que el servidor y la librería operations no se ven involucrados en esta operación del sistema.

Extras

Adicionalmente a la funcionalidad obligatoria hemos decidido añadir algunos detalles que consideramos iban a hacer el proyecto más completo y más pulido.

1. En una ejecución del cliente.java sólo puede existir un usuario conectado a la vez. Esto lo hicimos para poder asegurar que las operaciones hechas se asignen al usuario correcto.
2. Cuando un cliente cierra la ejecución de forma forzosa sin mandar el comando QUIT, al igual que hemos hecho con el servidor, para proteger la base de datos, mandamos primero la operación DISCONNECT
3. Para mayor claridad, en la función list_users se identifica al cliente que está haciendo la llamada.

```
c> LIST_USERS
c> LIST_USERS OK
      Alex      127.0.0.1     46229
    Sergio      127.0.0.1     42101
Dani (you)      127.0.0.1     38387
c>
```

4. La función getFile, como ya ha sido descrito comprueba si un archivo existe antes de mandarlo a otro usuario. Pero para no dar la posibilidad de que un fichero se publique sin tenerlo y poder provocar llamadas que sabemos que no van a funcionar de otros cliente, decidimos añadir la condición de que el archivo existiera previo a hacer un PUBLISH.
5. Aunque no se pide expresamente, decidimos que el servidor también debería imprimir por pantalla las operaciones que se van realizando en el siguiente formato:
   S> Usuario: <usuario> Operación: <operación>.
   Además, también imprime la respuesta que devuelve al cliente, en la forma:
   S> Enviando respuesta al cliente <respuesta>.

Pruebas

Para comprobar el correcto funcionamiento del sistema, realizamos una batería de pruebas, con el objetivo de asegurarnos de que las funciones devuelvan los códigos y mensajes de error que se corresponden a cada caso. Las pruebas que realizamos fueron las siguientes:

1. La operación REGISTER para registrar a un usuario.
   - Resultado obtenido: “REGISTER OK”
   - Código de error: 0
2. La operación REGISTER con un usuario ya registrado.

    - Resultado obtenido: “USERNAME IN USE”
    - Código de error: 1
3. La operación REGISTER con el servidor inactivo.
    - Resultado obtenido: “REGISTER FAIL”
    - Código de error: 2
4. La operación UNREGISTER para dar de baja a un usuario.
    - Resultado obtenido: “UNREGISTER OK”
    - Código de error: 0
5. La operación UNREGISTER con un usuario inexistente.
    - Resultado obtenido: “USER DOES NOT EXIST”
    - Código de error: 1
6. La operación UNREGISTER con el servidor inactivo.
    - Resultado obtenido: “UNREGISTER FAIL”
    - Código de error: 2
7. La operación CONNECT para conectar a un usuario.
    - Resultado obtenido: “CONNECT OK”
    - Código de error: 0
8. La operación CONNECT con un usuario no registrado.
    - Resultado obtenido: “CONNECT FAIL, USER DOES NOT EXIST”
    - Código de error: 1
9. La operación CONNECT para un usuario ya conectado.
    - Resultado obtenido: “CONNECT FAIL, CONNECTION ACTIVE: <user>”
    - Código de error: 2. *
10. La operación CONNECT con el servidor inactivo.
    - Resultado obtenido: “CONNECT FAIL”
    - Código de error: 3
11. La operación DISCONNECT para desconectar a un usuario.
    - Resultado obtenido: “DISCONNECT OK”
    - Código de error: 0
12. La operación DISCONNECT con un usuario no registrado.
    - Resultado obtenido: “DISCONNECT FAIL / USER DOES NOT EXIST”
    - Código de error: 1
13. La operación DISCONNECT para desconectar un usuario no conectado.
    - Resultado obtenido: “DISCONNECT FAIL / USER NOT CONNECTED”
    - Código de error: 2
14. La operación DISCONNECT con el servidor inactivo.
    - Resultado obtenido: “DISCONNECT FAIL”
    - Código de error: 3
15. La operación PUBLISH para publicar un contenido.
    - Resultado obtenido: “PUBLISH OK”
    - Código de error: 0
16. La operación PUBLISH con un usuario no registrado.
    - Resultado obtenido: “PUBLISH FAIL, USER NOT CONNECTED”
    - Código de error: 2
17. La operación PUBLISH para un usuario no conectado. **
    - Resultado obtenido: “PUBLISH FAIL, USER NOT CONNECTED”

    - Código de error: 2
18. La operación PUBLISH con un contenido ya publicado.
    - Resultado obtenido: “PUBLISH FAIL, CONTENT ALREADY PUBLISHED”
    - Código de error: 3
19. La operación PUBLISH con el servidor inactivo.
    - Resultado obtenido: “PUBLISH FAIL”
    - Código de error: 4
20. La operación DELETE para eliminar un contenido.
    - Resultado obtenido: “DELETE OK”
    - Código de error: 0
21. La operación DELETE con un usuario no registrado. **
    - Resultado obtenido: “DELETE FAIL, USER NOT CONNECTED”
    - Código de error: 2
22. La operación DELETE para un usuario no conectado.
    - Resultado obtenido: “DELETE FAIL, USER NOT CONNECTED”
    - Código de error: 2 *
23. La operación DELETE con un contenido no publicado.
    - Resultado obtenido: “DELETE FAIL, CONTENT NOT PUBLISHED”
    - Código de error: 3
24. La operación DELETE con el servidor inactivo.
    - Resultado obtenido: “DELETE FAIL”
    - Código de error: 4
25. La operación LIST_USERS para listar los usuarios conectados.
    - Resultado obtenido: “LIST_USERS OK”
    - Código de error: 0
26. La operación LIST_USERS con un usuario no registrado. **
    - Resultado obtenido: “LIST_USERS FAIL, USER NOT CONNECTED”
    - Código de error: 2
27. La operación LIST_USERS con un usuario no conectado.
    - Resultado obtenido: “LIST_USERS FAIL, USER NOT CONNECTED”
    - Código de error: 2
28. La operación LIST_USERS con el servidor inactivo.
    - Resultado obtenido: “LIST_USERS FAIL”
    - Código de error: 3
29. La operación LIST_CONTENT para listar los ficheros de un usuario.
    - Resultado obtenido: “LIST_CONTENT OK”
    - Código de error: 0
30. La operación LIST_CONTENT con un usuario no registrado. **
    - Resultado obtenido: “LIST_CONTENT FAIL, USER NOT CONNECTED”
    - Código de error: 2
31. La operación LIST_CONTENT con un usuario no conectado.
    - Resultado obtenido: “LIST_CONTENT FAIL, USER NOT CONNECTED”
    - Código de error: 2
32. La operación LIST_CONTENT con un usuario remoto inexistente.
    - Resultado obtenido: “LIST_CONTENT FAIL, REMOTE USER DOES NOT EXIST”

    - Código de error: 3
33. La operación LIST_CONTENT con el servidor inactivo.
    - Resultado obtenido: “LIST_CONTENT FAIL”
    - Código de error: 4
34. La operación GET_FILE para obtener un fichero de otro usuario.
    - Resultado obtenido: “GET_FILE OK”
    - Código de error: 0
35. La operación GET_FILE con un fichero remoto inexistente.
    - Resultado obtenido: “GET_FILE FAIL, FILE DOES NOT EXIST”
    - Código de error: 1
36. La operación GET_FILE con el server thread del cliente remoto caído.
    - Resultado obtenido: “GET_FILE FAIL”
    - Código de error: 2

Para los casos marcados con ** cabe mencionar, que en nuestro código, para esas funciones, lo primero que comprobamos es que si se están ejecutando desde un usuario no conectado, no haga nada y no se ejecute. Por lo tanto, el programa devuelve el mismo mensaje y el mismo código de error que en el caso de si se llama a la función desde un usuario que no está conectado.

Información adicional para la compilación y ejecución de los programas

Para la compilación de nuestros programas es requerida la instalación de sqlite3. Para instalar esta herramienta, se han de introducir los siguientes comandos en la terminal de debian:

```
$ sudo apt-get install sqlite3
```

```
$ sudo apt-get install libsqlite3-dev
```

Para la compilación del cliente, se necesita tener instalado javac. De la misma manera, para la ejecución del cliente se necesita tener instalado java.

Hemos añadido al Makefile la compilación de los archivos java, por lo que con el simple comando make estará todo listo para ejecutar.

Parte 2

En esta parte de la práctica crearemos un servidor **RPC** que nos proporcionará procedimientos remotos para ejecutar las funciones del sistema. Nuestro servidor se conectará con el nuevo servidor RPC y la respuestas que obtenga se las redirigirá a los clientes.

Las funciones que gestionan la base de datos son las mismas que utilizamos en la primera parte, solo que en este caso, en lugar de ejecutarse en el servidor, se ejecutarán en el servidor RPC (storage_server).

El programa cliente en java permanece igual que antes, pues no se ve involucrado en los intercambios entre el server y el server RPC.

El servidor, sin embargo, ha sido modificado y adaptado a esta nueva situación para poder conectarse al servidor RPC y emplear los procedimientos remotos.

Generación de RPC

Para generar el servidor RPC el primer paso es crear un script de generación, a partir del cual, el software de generación RPCGEN nos genera los programas requeridos para la creación del servidor RPC.

La implementación de este script ha resultado ser la tarea más ardua de esta parte de la práctica, ya que cada vez que intentábamos meter una modificación para mejorar o corregir algún fallo, había que generar de nuevo todos los programas y adaptarlos a nuestro trabajo. Un ejemplo fue en una ocasión en la que nombramos al fichero de generación “rpc.x”, lo que provocó que el software generase un fichero “rpch” que interfería con el “rpc.h” de la librería de linux. Finalmente conseguimos establecer el tipo de los argumentos y de los datos devueltos por las funciones RPC que son los reflejados en nuestro fichero storage.x.

A partir de aquí, el software nos genera seis ficheros: storage.h, storage_server.c, storage_svc.c, storage_clnt.c, storage_client.c y Makefile.storage.

En storage.h no modificamos apenas nada, simplemente añadimos algún include y constantes necesarias.

storage_client.c no lo utilizamos para nada, pues nuestro cliente corresponde a nuestro servidor: server.c. Simplemente nos sirvió como ejemplo para utilizar las funciones de procedimientos remotos.

storage_clnt.c no lo modificamos, lo dejamos como nos venía generado.

Aunque **storage_svc.c** no debería tocarse, nosotros realizamos unas pequeñas modificaciones, consistentes en introducir en el main la llamada a nuestra función startServer(), encargada de inicializar las bases de datos, y un signal que capture la señal Ctrl+C para ejecutar nuestra función stopServer y así poder salir del servidor RPC.

En **storage_server.c** se van a llevar a cabo todas las operaciones sobre la base de datos. El programa generado por rpcgen nos proporciona un esqueleto con las funciones que debemos completar para realizar cada una de las operaciones. En cada una de las funciones de este fichero llamamos a su respectiva función de la librería operations y asignamos el valor true a retval. Los valores de retorno de estas funciones deben pasarse por el puntero result, para que el software de rpc sea capaz de devolverlo a su cliente, que en nuestro caso es server, de forma correcta. Mención aparte merecen las funciones list_users() y list_content(), que además de devolver un entero con el código de error, tienen que devolver una lista de strings con la información requerida. Para hacer esto aprovechamos que en el estándar XDR se pueden definir vectores de tipo string e hicimos que estas dos funciones devolviesen un elemento de este tipo. Para incluir la información de los códigos de error y del número de elementos (usuarios o ficheros según el caso) se nos ocurrió ampliar la lista devuelta por la función de operations añadiéndole dos strings más con esta información, de forma que al cliente RPC (en nuestro caso server) le llega una única lista de strings con toda la información.

En **server.c**, para que pueda funcionar como cliente del servidor rpc, tuvimos que hacer las siguientes modificaciones: incluimos storage.h, el cual declara las nuevas funciones del servidor, declaramos las variables globales clnt, retval_, result_ y host, que nos harán falta a la hora de utilizar las funciones de rpc. En el main creamos el cliente de rpc con una conexión tipo “tcp” y comprobamos que el cliente no sea nulo. Después, en cada una de las operaciones llamamos a las respectivas funciones rpc guardando el resultado en result_ para posteriormente enviarlo al cliente. En el caso de las funciones list_users() y list_content() debemos decodificar la matriz de strings recibida del server RPC, identificando código de error, número de elementos y la lista en sí. Para que funcione correctamente la comunicación entre server y storage_server tuvimos que crear una matriz de strings de un tamaño fijo que establecimos a 256, para que nos la rellenase el procedimiento remoto.

Por último en el fichero Makefile.storage tuvimos que hacer las modificaciones necesarias para poder compilar todos los programas mientras desarrollamos la aplicación y efectuamos todas las pruebas pertinentes. En este makefile no se encuentra la compilación del cliente.java, ya que no se modificó en absoluto con respecto a la primera parte de la práctica.

Pruebas

Para comprobar el funcionamiento del sistema RPC, realizamos una batería de pruebas, con el objetivo de asegurarnos de que las funciones devuelvan los códigos y mensajes de error que se corresponden a cada caso. Las pruebas que realizamos fueron las siguientes:

1. La operación REGISTER para registrar a un usuario.
    - Resultado obtenido: “REGISTER OK”
    - Código de error: 0
2. La operación REGISTER con un usuario ya registrado.
    - Resultado obtenido: “USERNAME IN USE”
    - Código de error: 1

3. La operación REGISTER con el servidor inactivo.
    - Resultado obtenido: “REGISTER FAIL”
    - Código de error: 2
4. La operación UNREGISTER para dar de baja a un usuario.
    - Resultado obtenido: “UNREGISTER OK”
    - Código de error: 0
5. La operación UNREGISTER con un usuario inexistente.
    - Resultado obtenido: “USER DOES NOT EXIST”
    - Código de error: 1
6. La operación UNREGISTER con el servidor inactivo.
    - Resultado obtenido: “UNREGISTER FAIL”
    - Código de error: 2
7. La operación CONNECT para conectar a un usuario.
    - Resultado obtenido: “CONNECT OK”
    - Código de error: 0
8. La operación CONNECT con un usuario no registrado.
    - Resultado obtenido: “CONNECT FAIL, USER DOES NOT EXIST”
    - Código de error: 1
9. La operación CONNECT para un usuario ya conectado.
    - Resultado obtenido: “CONNECT FAIL, CONNECTION ACTIVE: <user>”
    - Código de error: 2. *
10. La operación CONNECT con el servidor inactivo.
    - Resultado obtenido: “CONNECT FAIL”
    - Código de error: 3
11. La operación DISCONNECT para desconectar a un usuario.
    - Resultado obtenido: “DISCONNECT OK”
    - Código de error: 0
12. La operación DISCONNECT con un usuario no registrado.
    - Resultado obtenido: “DISCONNECT FAIL / USER DOES NOT EXIST”
    - Código de error: 1
13. La operación DISCONNECT para desconectar un usuario no conectado.
    - Resultado obtenido: “DISCONNECT FAIL / USER NOT CONNECTED”
    - Código de error: 2
14. La operación DISCONNECT con el servidor inactivo.
    - Resultado obtenido: “DISCONNECT FAIL”
    - Código de error: 3
15. La operación PUBLISH para publicar un contenido.
    - Resultado obtenido: “PUBLISH OK”
    - Código de error: 0
16. La operación PUBLISH con un usuario no registrado. **
    - Resultado obtenido: “PUBLISH FAIL, USER NOT CONNECTED”
    - Código de error: 2
17. La operación PUBLISH para un usuario no conectado.
    - Resultado obtenido: “PUBLISH FAIL, USER NOT CONNECTED”
    - Código de error: 2 *
18. La operación PUBLISH con un contenido ya publicado.

    - Resultado obtenido: “PUBLISH FAIL, CONTENT ALREADY PUBLISHED”
    - Código de error: 3
19. La operación PUBLISH con el servidor inactivo.
    - Resultado obtenido: “PUBLISH FAIL”
    - Código de error: 4
20. La operación DELETE para eliminar un contenido.
    - Resultado obtenido: “DELETE OK”
    - Código de error: 0
21. La operación DELETE con un usuario no registrado. **
    - Resultado obtenido: “DELETE FAIL, USER NOT CONNECTED”
    - Código de error: 2
22. La operación DELETE para un usuario no conectado.
    - Resultado obtenido: “DELETE FAIL, USER NOT CONNECTED”
    - Código de error: 2
23. La operación DELETE con un contenido no publicado.
    - Resultado obtenido: “DELETE FAIL, CONTENT NOT PUBLISHED”
    - Código de error: 3
24. La operación DELETE con el servidor inactivo.
    - Resultado obtenido: “DELETE FAIL”
    - Código de error: 4
25. La operación LIST_USERS para listar los usuarios conectados.
    - Resultado obtenido: “LIST_USERS OK”
    - Código de error: 0
26. La operación LIST_USERS con un usuario no registrado. **
    - Resultado obtenido: “LIST_USERS FAIL, USER NOT CONNECTED”
    - Código de error: 2
27. La operación LIST_USERS con un usuario no conectado.
    - Resultado obtenido: “LIST_USERS FAIL, USER NOT CONNECTED”
    - Código de error: 2
28. La operación LIST_USERS con el servidor inactivo.
    - Resultado obtenido: “LIST_USERS FAIL”
    - Código de error: 3
29. La operación LIST_CONTENT para listar los ficheros de un usuario.
    - Resultado obtenido: “LIST_CONTENT OK”
    - Código de error: 0
30. La operación LIST_CONTENT con un usuario no registrado. **
    - Resultado obtenido: “LIST_CONTENT FAIL, USER NOT CONNECTED”
    - Código de error: 2
31. La operación LIST_CONTENT con un usuario no conectado.
    - Resultado obtenido: “LIST_CONTENT FAIL, USER NOT CONNECTED”
    - Código de error: 2
32. La operación LIST_CONTENT con un usuario remoto inexistente.
    - Resultado obtenido: “LIST_CONTENT FAIL, REMOTE USER DOES NOT EXIST”
    - Código de error: 3
33. La operación LIST_CONTENT con el servidor inactivo.

    - Resultado obtenido: “LIST_CONTENT FAIL”
    - Código de error: 4
34. La operación GET_FILE para obtener un fichero de otro usuario.
    - Resultado obtenido: “GET_FILE OK”
    - Código de error: 0
35. La operación GET_FILE con un fichero remoto inexistente.
    - Resultado obtenido: “GET_FILE FAIL, FILE DOES NOT EXIST”
    - Código de error: 1
36. La operación GET_FILE con el server thread del cliente remoto caído.
    - Resultado obtenido: “GET_FILE FAIL”
    - Código de error: 2

Información adicional para la compilación y ejecución de los programas

Para generar el servidor RPC, se requiere la instalación de rpcgen y para la ejecución de los programas, se requiere la herramienta rpcbind.

De igual manera, para compilar el servidor de la parte 1, es necesario instalar la herramienta sqlite3. Para el cliente, en su compilación se necesita javac y en su ejecución java.

Parte 3

En esta parte de la práctica, desarrollamos un servicio web utilizando **JAX-WS**. Este servicio tendrá como objetivo transformar a mayúsculas un texto enviado por el cliente Java. Para ello hemos implementado el servicio web en Java importando la librería javax.jws donde hemos programado el método para la conversión y en el main hemos introducido el Endpoint.
Hemos tenido un inconveniente al darnos cuenta de que al compilar el servicio, los imports que habíamos realizado no funcionaban teniendo la librería en el directorio.

Para terminar la parte sólo tendríamos que generar el wdsl y hacer la llamada desde el cliente antes del publish y delete, de tal forma que todas las acciones sobre los ficheros se hagan en mayúsculas gracias al servidor web creado.

Conclusión

Para la parte 1, los métodos que nos han resultado más problemáticos han sido List_Users y List_Content ya que había que enviar y recibir multitud de elementos, al tener que transferir una lista entera. Al hacer el RPC, al igual que en el caso anterior, las funciones que han sido más difíciles de adaptar al nuevo sistema han sido List_Users y List_Content, debido a que se realizaban de manera diferente al resto de funciones.

En conclusión, nos ha parecido una práctica muy entretenida y útil, ya que hemos podido crear y comprender un sistema de comunicación tipo peer-to-peer para la distribución de

ficheros, además del uso del servicio RPC. En el caso de Web Service, no hemos podido realizarlo debido a problemas de importación, aunque hemos realizado el código.