Universidad Carlos III

Sistemas Distribuidos

Curso 2023-24

Práctica Final

Diseño e implementación de un sistema peer-to-peer

Ingeniería Informática, Tercer curso

Adrián Fernández Galán (NIA: 100472182, e-mail: 100472182@alumnos.uc3m.es)
César López Mantecón (NIA: 100472092, e-mail: 100472092@alumnos.uc3m.es)

Prof . Félix García Caballeira y Alejandro Calderón Mateos
Grupo: 81

Índice

1. Introducción **2**

2. Diseño original **2**
2.1. Cliente 2
2.2. Servidor 2
2.2.1. Implementación en el servidor 2
2.2.2. Concurrencia del Servidor 3
2.3. Comunicación 3
2.3.1. Funciones auxiliares para la comunicación 4

3. Servicio web **4**

4. Integración del servidor RPC **4**

5. Compilación **5**

6. Descripción de pruebas **5**

7. Conclusiones **6**

1. Introducción

En este documento se recoge el desarrollo de la práctica final de Sistemas Distribuidos. Para esta práctica hemos desarrollado una aplicación distribuida que cuenta con 2 servidores desarrollados en lenguaje C, un código clinete desarrollado en python y un servicio web desarrollado igualmente en Python. A continuación describiremos el diseño e implementación de cada una de las partes del sistema.

2. Diseño original

La aplicación constará de dos partes diferenciadas: los clientes y el servidor.

2.1. Cliente

Los servicios proporcionados por la aplicación se podrán utilizar a través de los clientes programados en *Python*. Para ello los clientes tendrán una interfaz de comandos con la que podrán acceder a las distintas funcionalidades:

- **REGISTER**: Este mandato permitirá al usuario crearse una nueva cuenta en nuestra aplicación
- **UNREGISTER**: Este mandato permitirá al usuario borrar su cuenta de nuestra aplicación
- **CONNECT**: Este mandato permitirá al usuario conectarse a la aplicación, por lo que dispondrá del resto de funcionalidades
- **DISCONNECT**: Este mandato permitirá al usuario desconectarse de la aplicación, perdiendo el acceso al resto de funcionalidades
- **PUBLISH**: Este mandato permitirá al usuario publicar un archivo para que sea visible al resto de clientes. Para ello proporcionará un *path* absoluto del fichero
- **DELETE**: Este mandato permitirá al usuario borrar un archivo publicado anteriormente
- **LIST_USERS**: Este mandato permitirá al usuario conocer cuales son los usuarios que están conectados
- **LIST_CONTENT**: Este mandato permitirá al usuario conocer los archivos publicados por el usuario proporcionado
- **GET_FILE**: Este mandato permitirá al usuario obtener un archivo publicado por otro usuario

Cabe destacar que al realizar el comando **CONNECT** se creará un nuevo hilo por parte del cliente para atender peticiones de otros clientes sobre los ficheros publicados. De esta manera el cliente tendrá un hilo que siga utilizando la interfaz de comandos mientras que otro hilo dará soporte a las peticiones de otros clientes.

2.2. Servidor

El servidor implementa los servicios necesarios para la coordinación de clientes. Para esto se apoya en una estructura de implementación propia especialmente diseñada para las particularidades de la práctica.

2.2.1. Implementación en el servidor

Se ha implementado una estructura de datos para representar la lista de usuarios registrados en el sistema. Para cada usuario se almacena su nombre, los archivos que tiene publicados, su ip, su puerto y si está o no conectado. La estructura empleada es la siguiente:

```
#define CHARSIZE 512

typedef struct {
    char name[CHARSIZE];
    char description[CHARSIZE];

} file;

typedef struct {
    char name[CHARSIZE];
    file* contents;
    int contentsLen;
    int contentsMaxLen;
    int conected;
    int port;
    char ip[32];

} user;
```

Dado que pueden haber un número indefinido de usuarios registrados, y que cada usuario puede tener publicados un número cualquiera de ficheros, se ha optado porque la estructura doble su tamaño cada vez que sea necesario relocalizar la memoria.

Para el manejo de la estructura se ha implementado la siguiente interfaz de funciones:

- `createUserList()`: Esta función crea una lista de usuarios vacía.
- `searchUser()`: Esta función busca un usuario en la lista.
- `addUser()`: Esta función añade un nuevo usuario en el caso de que no exista ya.
- `removeUser()`: Esta función borra al usuario si existe en la lista.
- `addContent()`: Esta función añade nuevo contenido a un usuario existente.
- `removeContent()`: Esta función borra contenido a un usuario existente.
- `destroyList()`: Esta función borra la lista de usuarios por completo, liberando la memoria empleada para almacenar la misma.

2.2.2. Concurrencia del Servidor

El servidor atiende clientes de manera concurrente. Para esto hemos optado por una aproximación de hilos bajo demanda. Es decir, cada vez que el servidor recibe una petición creará y lanzará un hilo *detached* para el tratamiento de la misma. Con esto, son necesarios dos requisitos: detener al hilo principal para que no reciba una nueva petición hasta que el hilo haya copiado la anterior y asegurar el acceso exclusivo a la estructura definida en el apartado anterior.

Para lo primero, al igual que en la práctica dos, hacemos uso de un mutex y una condición que bloqueal al hilo principal hasta que el hilo haya copiado el descriptor, ip y puerto del cliente conectado.

Para asegurar el acceso exclusivo a la estructura de datos y, así, evitar condiciones de carrera, hacemos uso de un segundo mutex que protege cualquier sección de código que acceda a la lista de usuarios.

2.3. Comunicación

La comunicación entre los distintos clientes y el servidor ocurre de manera independiente a las distintas máquinas que se conectan a nuestro servicio a través de sockets. Para conseguir esto hemos codificado todas las comunicaciones en cadeanas de caracterres, a excepción del valor de retorno de las operaciones. Además, para evitar problemas ocasionados por el ordenamiento de bytes enviamos un único carácter a la vez.

La comunicación será una comunicación por petición, por lo que se abrirá la conexión con el servidor para realizar la petición correspondiente y, si todo ha funcionado correctamente, se cerrará la conexión.

2.3.1. Funciones auxiliares para la comunicación

Para facilitar la implementación de la comunicación mediante *sockets* se han implementado una serie de funciones auxiliares recogidas en el fichero *src/servidor/common.c*. Estas implementan acciones repetidas en el lado servidor, así como gestión de errores y envío y recepción de mensajes adaptados a las necesidades del sistema.

- `serverSocket()`: crea y devuelve un *socket* para el servidor en un número de puerto dado. Además, se ejecuta la llamada `listen()` para permitir al servidor aceptar conexiones más adelante.
- `serverAccept()`: esta función permite al servidor aceptar una conexión con un cliente.
- `clientSocket()`: crea y devuelve un socket para un cliente en el sistema. Además, este *socket* estará conectado al servidor.
- `sendMessage()`: envía un mensaje contenido en un *buffer* a través de un *socket*.
- `recvMessage()`: recibe un mensaje y lo almacena en n *buffer* dado.
- `writeline()`: se apoya de la función `sendMessage()` para enviar un mensaje hasta el final de cadena.
- `readLine()`: se apoya de `recvMessage()` para recibir un menesaje hasta el final de la cadena de texto.

3. Servicio web

Se ha desarrollado un servcio web en *Python* que proporciona la fecha y la hora en la que se realizó la petición al servicio web. Este servicio web se llamará desde el lado del cliente cada vez que se quiera realizar un llamada a algún servicio de la aplicación, y se mandarán al servidor junto al resto de datos necesarios.

4. Integración del servidor RPC

En esta sección describimos cómo hemos abordado la implementación de un servidor (desde este momento referido como *servidor rpc*) que recibe una cadena de carácteres y la imprime por pantalla. El proceso que actuará de cliente en esta comunicación es el servidor desarrollado para la aplicación distribuída de compartición de ficheros.

Lo primero, hemos descrito la interfaz necesaria utilizando el lenguaje XDR. En este caso necesitaremos una única función que reciba una cadena de texto.

```
program PRINT {
    version PRINTVER {
        int rpc_print(string impresion) = 0;
    } = 1;
} = 99;
```

Listing 1: Interfaz XDR

Esta interfaz nos permite generar 4 ficheros de código que implementan la funcionalidad deseada en una función `rpc_print_1`. Esto lo hacemos a través del comando `rpcgen`.

Para que el proceso cliente conozca la dirección IP del servidor rpc hemos decidido utilizar una variable de entorno *RPC_IP*. Para su consulta hemos desarrollado una función `get_ip`, implementada en el fichero *servidor.c*. Por último, hemos implementado una función que encapsula la construcción y envío de la cadena de texto exigida por el enunciado al servidor rpc.

5. Compilación

En esta sección nos centraremos en la forma de compilar los servidores, ya que son la única parte del código escrita en un lenguaje compilado. La compilación del proyecto está contenida en un fichero *Makefile* situado en la raíz del proyecto, de forma que a través del comando `make` se generen todos los ejecutables necesariospara levantar el sistema.

El código necesario para el servidor está distribuido en varios ficheros. Primero, el código encargado de la gestión de la comunicación con los clientes python y la ejecución de los servicios se encuentra contenido en el fichero *src/servidor/servidor.c*; segundo, el código que encapsula la funcionalidad de la estructura de datos se encuentra en el fichero *src/servidor/server_storage.c*; y, por último, el código para la invocación de los procesos remotos se encuentra en el fichero *print_clnt.c* generado a través del comando `rpcgen`.Para generar el ejecutable correcatmente hemos compilado cada uno de estos ficheros por separado con las opciones de compilación adecuadas para crear varios ficheros objeto; después hemos enlazado estos ficheros en la generación del ejecutable *servidor*.

El código del servidor RPC está contenido exclusivamente en los ficheros *print_svc.c* y *print_server.c*. De nuevo, compilaremos estos dos ficheros de manera separada para luego enlazarlos.

A continuación se muestran los comandos necesarios para generar los ficheros objeto:

```
# generacion de objeto de la implementacion
gcc -c src/servidor/server_storage.c

# generacion de objeto de funciones auxiliares
gcc -c src/servidor/common.c

# generacion de objetos de archivos rpc
gcc -g -I/usr/include/tirpc -D_REENTRANT -o print_clnt.o -c src/rpc
/print_clnt.c
gcc -g -I/usr/include/tirpc -D_REENTRANT -o print_svc.o -c src/rpc/
print_svc.c

# generacion de objeto de servidores
gcc -c servidor.c
gcc -g -I/usr/include/tirpc -c src/rpc/print_server.c
```

Listing 2: Compilación de ficheros en ficheros objetos

Y con el siguiente fragmento del fichero *Makefile* se enlazan los ficheros y se generan ambos ejecutables con los nombres *servidor* y *servidor_rpc*:

```
# generacion de ejecutable del servidor
gcc -g -Wall -lrt -o servidor servidor.o server_storage.o common.o
print_clnt.o -lnsl -lpthread -ldl -ltirpc

# generacion de ejecutable del servidor rpc
gcc -g -Wall -lrt -o servidor_rpc print_server.o print_svc.o -lnsl
-lpthread -ldl -ltirpc
```

Listing 3: Generación de ejecutables

6. Descripción de pruebas

Descripción de pruebas

7. Conclusiones

Este ejercicio combina casi todas las tecnologías que se nos han presentado durante el curso en un sistema completo que pretende aproximarse a una aplicación distribuída. Esto nos ha permitido afianzar los conocimientos adquiridos en la asignatura y desarrollar nuestras competencias para la programación de servicios distribuidos.

Además, esta práctica nos presenta por primera vez la necesidad de integrar nuevos servicios basados en otra tecnología sobre un sistema ya funcional. Esta es una aptitud verdaderamente interesante y valiosa de cara a nuestro desarrollo como informáticos.