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

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 cliente 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;

typedef struct {
    int size;
    int max_size;
    user* users;
} __user_list;

typedef __user_list* user_list;
```

Listing 1: Estructura de datos para el servidor

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 que se cerrará al final de la misma.

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, a través de la librería *zeep*, 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. En el lado cliente se asume que el servidor web se encuentra en la ip *localhost*.

El servicio web se encontrará alojado en un servidor que hará uso de la librería *spyne* para implementar una única función. Esta función obtendrá la fecha actual del dispostivo a través *time* y la devolverá.

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 2: 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 *server.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 necesarios para 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/server.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 correctamente 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 server.c
gcc -g -I/usr/include/tirpc -c src/rpc/print_server.c
```

Listing 3: 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 4: Generación de ejecutables

6. Ejecución del proyecto

Con todo lo anterior, la forma de ejecutar el proyecto pasa por lanzar 3 servidores y, al menos, un cliente. Además, en el caso del primer servidor desarrollado, encargado de la coordinación de clientes en la aplicación distirbuida, se deberá definir una variable de entorno a través (por ejemplo) del comando `env`.

```
# suponiendo que este en la raiz del proyecto

# servidores
env RPC_IP=localhost ./servidor -p 4500 &
./servidor_rpc &
python src/servicio_web/timestamp.py &

# cliente
python src/cliente/client.py -s localhost -p 4500
```

Listing 5: Ejemplo de ejecucion con los servidores en background y en local

7. Descripción de pruebas

Los requisitos de la aplicación nos dificultan el desarrollo de pruebas de manera reproducible, por lo que se ha optado por realizar las pruebas manualmente en local y analizar los resultados. No obstante, en el entregable se incluye un pequeño script en python que permite generar los ficheros necesarios para las pruebas y la entrada de las distintas terminales para cada test para facilitar la ejecución de las mismas.

```
python3 tests/setup_clean.py 0 # generacion de ficheros
python3 tests/setup_clean.py 1 # limpieza de ficheros
```

Listing 6: Generar/limpiar ficheros de prueba y entradas de terminal

Las pruebas se realizarán a través de la ejecución de varios procesos en distintos terminales para la correcta visualización de las distintas impresiones. Al principio de cada prueba se iniciarán el servidor, el servidor RPC y el servidor web; y al final de estás se cerrarán estos servidores.

7.1. Implementación

Hemos desarrollado un pequeño código que prueba la implementación de la estructura desarrollada para almacenar la lista de clientes. Este código realiza inserciones y borrados en dicha lista, imprimiendo su estado por pantalla en cada punto, de forma que podemos analizar el comportamiento de la estructura de datos. Las pruebas están contenidas en el fichero *tests_userList.c* y se pueden compilar y ejecutar a través del comando `make testing`.

Para la ejecución de estas pruebas, se ha modificado el código de la estructura de forma que se inicialice con una capacidad menor a la que usamos en el sistema. Esto nos ha permitido comprobar que la estructura aumenta de capacidad correctamente.

7.2. Un solo cliente

Se realizará una prueba donde no exista concurrencia. Con tan solo un cliente se probará el correcto funcionamiento de la comunicación del cliente con el servidor, del cliente con el servicio web y del servidor con el servicio rpc. El cliente ejecutará las siguientes operaciones

```
REGISTER A
REGISTER B
CONNECT C
CONNECT A
CONNECT B
CONNECT A
PUBLISH HOLA HOLA
LIST_USERS
```

```
LIST_CONTENT B
LIST_CONTENT A
QUIT
```

Listing 7: Operaciones a realizar por el cliente

7.2.1. Resultados

Los resultados obtenidos en cada una de las terminales muestran el correcto comportamiento del sistema.

```
c> REGISTER A
REGISTER OK
c> REGISTER B
REGISTER OK
c> CONNECT C
CONNECT FAIL, USER DOES NOT EXIST
c> CONNECT A
CONNECT OK
c> CONNECT B
CONNECT FAIL
c> CONNECT A
CONNECT FAIL
c> PUBLISH HOLA HOLA
PUBLISH OK
c> LIST_USERS
LIST_USERS OK
c> LIST_CONTENT B
LIST_CONTENT FAIL, USER NOT CONNECTED
c> LIST_CONTENT A
HOLA HOLA
LIST_CONTENT OK
c>QUIT
DISCONNECT OK
+++ FINISHED +++
```

Listing 8: Resultados obtenidos en el cliente

```
A REGISTER  11-05-2024 08:32:41
B REGISTER  11-05-2024 08:32:46
C CONNECT     11-05-2024 08:32:51
A CONNECT     11-05-2024 08:32:55
A PUBLISH HOLA  11-05-2024 08:33:18
A LIST_USERS  11-05-2024 08:33:22
A LIST_CONTENT  11-05-2024 08:35:56
A LIST_CONTENT  11-05-2024 08:36:25
A DISCONNECT  11-05-2024 08:36:44
```

Listing 9: Impresiones del servidor RPC

7.3. Comunicación entre dos clientes

Para probar el correcto funcionamiento de la comunicación entre clientes se tendrán dos clientes que realizarán distintas operaciones: el primero publicará un archivo y el segundo se lo pedirá. De esta manera ambos clientes tendrán su propio hilo para atender peticiones y el servidor podrá aprovechar la concurrencia para atender a los dos clientes.

Los clientes ejecutarán las siguientes operaciones

```
REGISTER A
CONNECT A
PUBLISH /tmp/ficheroA.txt HOLA MUNDO
```

```
QUIT
```

Listing 10: Operaciones a realizar por el cliente A

```
REGISTER B
CONNECT B
GET_FILE A /tmp/ficheroA.txt /tmp/fileA.txt
QUIT
```

Listing 11: Operaciones a realizar por el cliente B

7.3.1. Resultados

El resultado obtenido coincide con el resultado esperado. Ambos clientes son capaces de acceder a la estructura concurrentemente sin condiciones de carrera y conectarse entre ellos para la transferencia de un archivo. El resultado obtenido en las terminales de ambos clientes y del servidor rpc es parecido al del test anterior, por lo que no será incluído en la memoria.

```
# antes de la ejecucion de get_file
[@localhost]$ ls /tmp/ | grep .txt
ficheroA.txt
[@localhost]$ cat /tmp/ficheroA.txt
HOLA MUNDO DESDE CLIENTE A
# despues de la ejecucion de get_file
[@localhost]$ ls /tmp/ | grep .txt
ficheroA.txt
fileA.txt
[@localhost]$ cat /tmp/fileA.txt
HOLA MUNDO DESDE CLIENTE A
```

Listing 12: Resultado de la transferencia del archivo

7.4. Prueba de estrés para el servidor

Con el objetivo de probar la concurrencia del servidor, y su capacidad para atender a muchos clientes, se utilizarán 5 clientes (con los nombres de la A a la E) que se conectarán al servicio, publicarán un fichero, realizarán un listado de contenido y obtendrán el fichero publicado por el siguiente cliente. Para ello todos los clientes ejecutarán un código similar al siguiente

```
REGISTER i
CONNECT i
PUBLISH /tmp/ficheroPrueba_i
LIST_CONTENT i+1
GET_FILE i+1 /tmp/ficheroPrueba_i+1 <dir_actual>/obtenido_i
```

Listing 13: Operaciones a realizar por cada cliente i

7.4.1. Resultados

Para la ejecución de este test es necesario ejecutar de manera concurrente 5 clientes. Para ello nos hemos apoyado de un pequeño script de bash que se adjunta a continuación. Dado el funcionamiento de los mandatos en *background*, cada cliente se ejecutará en una subterminal propia, asegurándonos la ejecución simultánea de todos los procesos.

```
#!/bin/bash

function set_server () {
    printf "\033[0;33mSETUP_SERVERS\033[0m\n"
    env RPC_IP=localhost ./servidor -p 4500 > svc_$1.log &
    ./servidor_rpc > rpc_$1.log &
    python3 src/servicio_web/timestamp.py > tmstmp_$1.log &
```

```
}

function kill_server () {
    killall -s INT servidor
    echo "kill server"
    killall -s KILL servidor_rpc
    echo "kill python3"
    killall -s KILL python3
}


set_server "concurrencia"
python3 src/cliente/client.py -s localhost -p 4500 < test_stressA.in &
python3 src/cliente/client.py -s localhost -p 4500 < test_stressB.in &
python3 src/cliente/client.py -s localhost -p 4500 < test_stressC.in &
python3 src/cliente/client.py -s localhost -p 4500 < test_stressD.in &
python3 src/cliente/client.py -s localhost -p 4500 < test_stressE.in &
sleep 5
kill_server
```

Listing 14: Script para la ejecucion concurrente de 5 clientes

El resultado de este test es el esperado. Cada cliente ha podido ejecutarse sin problemas, obteniendo en el directorio actual el fichero solicitado al siguiente cliente. Además, la estructura de datos no ha presentado ningún tipo de condición de carrera.

8. 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.