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 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_CONTETN**: 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

nuestra estructura vector que dobla de capacidad cuando se llena válido por su caracter de prototipo.

Se ha implementado una estructura de datos que consta de una lista de usuarios, donde cada usuario contiene información sobre este; tales como el nombre, los archivos publicados, si está conectado o no, y el ip y puerto. Dado que pueden haber un número indefinido de usuarios registrados se ha optado por utilizar un *vector*, que si se llena doblará su tamaño. Para acceder a esta lista de usuarios se han implementado las siguientes interfaces:

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

2.2.2 Concurrencia del Servidor

Describir como aseguramos la concurrencia de clientes del lado del servidor =¿ acceso a la estructura.

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 es necesario que los mensajes que se envían por los sockets no sean *multi-byte*, por lo que hemos optado por envíar y recibir tiras de caracteres.

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/common.c*. Estas implementan acciones repetidas tanto en el lado cliente y 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.
- `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

Describir aquí como hemos integrado el servidor y las modificaciones necesarias en el código

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.

5.1 Servidor

```
# 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 objeto del servidor
gcc -c servidor.c
```

5.2 RPC

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

5.3 Compilación del ejecutable servidor y el 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
```

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.