Práctica Final - Diseño e implementación de un sistema peer-to-peer

Adrián Martínez Cruz 100451213

César López Navarro 100451326

Grupo 86

1. Introducción

En este documento se recoge el desarrollo de la práctica final de Sistemas Distribuidos. Para esta práctica se ha desarrollado una aplicación distribuida que cuenta con un servidor desarrollado 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

Cliente

1. Objetivo general

El cliente, implementado en Python, permite que un usuario se conecte al servidor TCP multihilo implementado en C y realice operaciones como registrarse, conectarse, publicar archivos, consultar usuarios, descargar ficheros, etc. El cliente interpreta comandos introducidos por los usuarios y los envía al servidor para que éste resuelva las peticiones.

2. Arquitectura

El cliente se organiza en una única clase `client`, con métodos estáticos y utiliza **sockets TCP** para comunicarse con el servidor. Además, implementa su propio protocolo de envío y recepción de cadenas de texto terminadas en `\0`. Cuando un usuario se conecta (función `CONNECT`, se comentará en el próximo apartado) se utiliza un **hilo secundario** (`_listener_thread`) para escuchar peticiones entrantes de otros clientes.

3. Comandos implementados

El cliente reconoce los siguientes comandos desde la shell interactiva:

- **`REGISTER <user_name>`**

Envía al servidor una solicitud de registro para el usuario indicado.

- **`UNREGISTER <user_name>`**

Solicita la eliminación del usuario del sistema.

- **`CONNECT <user_ame>`**

Conecta al usuario y abre un **socket de escucha local** en un puerto dinámico, para recibir conexiones entrantes.

- **`DISCONNECT <user_name>`**

Desconecta al usuario, detiene el hilo de escucha y cierra el socket local.

- **`PUBLISH <file_name> <description>`**

Publica un archivo disponible para otros usuarios. La descripción se envía como cadena de texto al servidor.

- **`DELETE <file_name>`**

Elimina el archivo publicado del listado.

- **`LIST_USERS`**

Pide al servidor una lista de todos los usuarios registrados, junto a su IP y puerto, inicialmente a 0 si no estuviesen conectados.

- **`LIST_CONTENT <user_name>`**

Solicita al servidor la lista de archivos publicados por otro usuario.

- **`GET_FILE <user_name> <remote_file_name> <local_file_name>`**

Obtiene un archivo publicado por otro usuario de la siguiente manera:

1. Consulta al servidor los datos del usuario destino.
2. Se conecta directamente a su socket de escucha.
3. Solicita el archivo y lo guarda localmente.

- **`QUIT`**

Cierra la shell y termina el programa.

4. Transferencia de archivos entre clientes

Una de las características más destacadas del cliente es la **transferencia directa de**

archivos P2P entre clientes:

- Cuando un cliente hace `CONNECT`, abre un socket local en un puerto dinámico.
- Alguien que hace `GET_FILE` se conecta directamente a ese puerto y solicita el archivo.
- La función `_handle_incoming_file_request()` gestiona esta solicitud.

Esto simula un sistema de compartición distribuido, aunque siempre mediado por el servidor para obtener los datos IP/puerto del usuario remoto.

5. Finalización del cliente

Como se ha comentado anteriormente, el cliente finaliza cuando el usuario escribe `QUIT` por la línea de mandatos.

Servidor

1. Objetivo general

El servidor implementa un sistema multiusuario en C que permite a múltiples clientes conectarse, registrarse, publicar contenidos, y consultar la información de otros usuarios a través de comandos enviados por socket TCP. Está desarrollado en un modelo **multihilo (con `pthreads`)**, permitiendo que varias peticiones de cliente se gestionen simultáneamente.

2. Estructura

El servidor utiliza **sockets TCP** para la comunicación. Primero crea un socket servidor (`ss`) y lo enlaza al puerto recibido por línea de comandos para después aceptar nuevas conexiones entrantes con la función `accept()`. Cada cliente es atendido por un **nuevo hilo (`pthread`)** que ejecuta la función `tratar_peticion()`

3. Gestión de peticiones

La función `tratar_peticion()` es la encargada de procesar los comandos enviados por el cliente. Cada hilo trabaja de forma aislada y se sincroniza el acceso al socket mediante un `mutex`.

Se reconocen los siguientes comandos:

- `REGISTER <user_name>`: registra un nuevo usuario.
- `UNREGISTER <user_name>`: elimina un usuario registrado.
- `CONNECT <user_name>`: conecta un usuario y actualiza su IP/puerto.
- `DISCONNECT <user_name>`: desconecta al usuario.
- `PUBLISH <file_name> <desc>`: publica un archivo con descripción.
- `DELETE <file_name>`: elimina un contenido publicado.
- `LIST_USERS`: envía la lista de usuarios.
- `LIST_CONTENT <user_name>`: envía una lista de los archivos publicados por otro usuario.
- `GET_FILE <user_name> <remote_file_name> <local_file_name>`: copia de manera local un archivo publicado por otro usuario.

4. Multihilo y sincronización

Cada cliente se atiende con un **hilo independiente**, lo que permite gestionar múltiples conexiones concurrentes. Todo este funcionamiento ha sido desarrollado anteriormente en el ejercicio evaluable 2.

5. Finalización del servidor

El servidor se detiene correctamente al recibir la señal `SIGINT` (Ctrl+C), liberando los recursos mediante la función `stop_server()` que destruye el mutex y cierra el socket.

Gestión de Errores

Se comprueban errores al abrir sockets, aceptar conexiones o acceder a ficheros con el objetivo de maximizar la robustez del resultado final. El servidor responde con códigos numéricos a los clientes según el resultado de las operaciones y se usan buffers seguros (`MAXSTR, MAXCHUNK`) y funciones de lectura protegida (`readLine`). Además, se considera necesario aclarar que hay errores que es imposible que se den por la estructura del código y, aún así, se ha decidido incluir código de depuración para ellos. Un posible caso es la ejecución de las funciones PUBLISH y DELETE cuando un usuario aún no está registrado, cuyo error se muestra

en cuanto el cliente no encuentra un usuario que publique o borre contenido pero no se llega a enviar al servidor ningún código numérico.

3. Servicio Web

Como parte de la segunda parte del proyecto, se ha desarrollado un servicio web SOAP con el objetivo de proporcionar una marca temporal a cada operación enviada desde el cliente al servidor. Este servicio actúa como un reloj remoto al que accede el cliente antes de ejecutar cualquier operación.

El servicio ha sido implementado en Python utilizando la librería Spyne, que facilita la creación de servicios web con soporte para el protocolo SOAP. La operación principal que ofrece es get_datetime, que devuelve la fecha y hora actuales en el formato dd/mm/yyyy hh:mm:ss.

Este servicio se ejecuta de forma local en la misma máquina donde se lanza el cliente, en el puerto 8090. Desde el cliente, se utiliza la librería Zeep para consumir el servicio, enviando la respuesta como parte de cada petición al servidor.

En el servidor, esta marca temporal se imprime en la salida estándar junto al nombre del usuario y la operación realizada, facilitando la trazabilidad y el análisis de las peticiones.

El servicio se ejecuta en segundo plano junto al servidor principal, garantizando disponibilidad constante durante la ejecución de las pruebas.

4. Compilar y generar los ejecutables

Para la compilación y generación de ejecutables se ha utilizado un Makefile. Será necesario ejecutar en terminal en el directorio *Práctica_Final* dos comandos en dos terminales.

En la primera:

```
./run-server.sh
```

Y en la segunda:

```
./run-client.sh
```

El primer script corresponde a la compilación del programa y los ejecutables del servidor y el servicio web.

El segundo script ejecuta el archivo de cliente en Python. Si se quieren ejecutar varios

clientes, se debe abrir una terminal diferente para cada cliente ya que si dos o más usuarios realizan peticiones desde la misma terminal puede dar lugar a errores no previstos. Cada usuario debe ejecutar el archivo *client.py* en una terminal diferente.

5. Batería de Pruebas

A continuación se muestran una serie de pruebas que han sido realizadas por los desarrolladores para comprobar el correcto funcionamiento de esta práctica.

Para su correcta interpretación se debe asumir que estas pruebas han sido realizadas teniendo en cuenta los requisitos necesarios para ello, por ejemplo, en la prueba de la función connect, se requiere que un usuario esté registrado para conectarse, por lo que, aunque esto no se ve en la siguiente tabla, se entiende que el registro se ha realizado anteriormente.

Además, para la limpieza de las pruebas de la función get_file, se ha utilizado un directorio con ficheros de prueba, así como un directorio dentro de éste para las copias que hacen los usuarios.

INPUT	EXPECTED OUTPUT	ACTUAL OUTPUT
REGISTER pepe	c> REGISTER OK	c> REGISTER OK
REGISTER pepe (usuario ya existe)	c> USERNAME IN USE	c> USERNAME IN USE
UNREGISTER pepe	c> UNREGISTER OK	c> UNREGISTER OK
UNREGISTER pepe (no existe)	c> USER DOES NOT EXIST	c> USER DOES NOT EXIST
CONNECT pepe	c> CONNECT OK	c> CONNECT OK
CONNECT pepe (ya conectado)	c> USER ALREADY CONNECTED	c> USER ALREADY CONNECTED
DISCONNECT pepe	c> DISCONNECT OK	c> DISCONNECT OK
DISCONNECT pepe (no conectado)	c> DISCONNECT FAIL, USER NOT CONNECTED	c> DISCONNECT FAIL, USER NOT CONNECTED
PUBLISH file.txt descripción	c> PUBLISH OK	c> PUBLISH OK
PUBLISH file.txt Descripción (usuario no conectado)	c> PUBLISH FAIL, USER NOT CONNECTED	c> PUBLISH FAIL, USER NOT CONNECTED

DELETE file.txt	c> DELETE OK	c> DELETE OK
DELETE file.txt (archivo no existe)	c> DELETE FAIL, CONTENT NOT PUBLISHED	c> DELETE FAIL, CONTENT NOT PUBLISHED
LIST_USERS	c> LIST_USERS OK + lista de usuarios	c> LIST_USERS OK + nombres IPs puertos
LIST_USERS (no conectado)	c> LIST_USERS FAIL, USER NOT CONNECTED	c> LIST_USERS FAIL, USER NOT CONNECTED
LIST_CONTENT juan	c> LIST_CONTENT OK + lista de archivos de juan	c> LIST_CONTENT OK + nombres de ficheros
LIST_CONTENT juan (sin contenidos)	c> LIST_CONTENT FAIL, USER HAS NO CONTENT	c> LIST_CONTENT FAIL, USER HAS NO CONTENT
GET_FILE juan tmp/prueba1.txt tmp/copias/copia1.txt	c> GET_FILE OK	c> GET_FILE OK
GET_FILE juan tmp/prueba2.txt tmp/copias/copia2.txt (archivo no existe)	c> GET_FILE FAIL, FILE NOT EXIST	c> GET_FILE FAIL, FILE NOT EXIST
QUIT	(Finaliza la ejecución)	(Finaliza la ejecución)

Por motivos de simplicidad y extensión de esta memoria, las pruebas de fallo por motivos ajenos al cliente no se han considerado en esta batería, pero se han probado con resultados exitosos.

6. Conclusiones

Gracias a este proyecto hemos aprendido a utilizar sockets TCP para establecer comunicaciones entre procesos, tanto desde C como desde Python. Hemos entendido cómo diseñar un protocolo de intercambio de mensajes entre cliente y servidor, y cómo gestionar múltiples conexiones simultáneas mediante programación multihilo. Además, hemos interiorizado el funcionamiento de una arquitectura peer-to-peer, en la que los propios clientes se comunican entre sí para intercambiar archivos, manteniendo al servidor como elemento de coordinación. Esta práctica ha sido clave para afianzar conceptos fundamentales de los sistemas distribuidos y aplicarlos de forma práctica en un entorno realista.