uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería Informática
2024-2025

Trabajo Final de Asignatura

“Memoria Proyecto Final”

Esteban Gómez Buitrago - 100485446

100485446@alumnos.uc3m.es

Nicolás Alejandro Cuesta García - 100495966

100495966@alumnos.uc3m.es

Profesor

FELIX GARCIA CARBALLEIRA

Leganés, 11 de Mayo de 2025

Índice

Descripción del código detallando las principales funciones implementadas..... 3
Servidor..... 3
Lógica General..... 3
server.h..... 3
server.c..... 3
socketFunctions.c..... 4
Cliente..... 4
client.py..... 4
protocol.py..... 5
webService.py..... 6
Descripción de la forma de compilar y obtener el ejecutable de todos los procesos involucrados..... 6
Batería de pruebas utilizadas y resultados obtenidos..... 7
Prueba 1 — Registro, conexión y borrado de cuenta concurrente..... 7
Prueba 2 — Usuario ya existente..... 8
Prueba 3 — Borrar a un usuario que no existe..... 8
Prueba 4 — Conectarse a un usuario no existente..... 9
Prueba 5 — Conectarse a un usuario ya conectado..... 9
Prueba 6 — Publicar ficheros y borrarlo. Listar usuarios y contenidos..... 9
Prueba 7 — Disconnect erróneos..... 11
Prueba 8 — Particularidades del disconnect..... 11
Prueba 9 — Get_file con concurrencia..... 12
Conclusiones, problemas encontrados, cómo se han solucionado, y opiniones personales..... 14

Descripción del código detallando las principales funciones implementadas

Servidor

Lógica General

Se creó un fichero principal llamado *server.c* que toma diferentes acciones dependiendo de la acción recibida y los resultados obtenidos de las funciones que realizan los cambios en el servidor. Por el otro lado, se creó un fichero llamado *socketFunctions.c* donde se implementan las funciones que realmente van a manipular las estructuras de datos y dar resultados al cliente. Asimismo, en este otro fichero se encuentran las funciones que se encargan de recibir y enviar mensajes a través de los sockets. Es importante resaltar que estas funciones fueron implementadas de tal forma que pueden mandar cadenas de caracteres. Finalmente, por el lado del servidor también se implementó un fichero de cabecera llamado *server.h.* Más adelante se entrará más en detalle en qué realizaba cada uno de estos ficheros.

server.h

Este fichero de cabecera se encarga de crear los prototipos de las funciones que se van a emplear, definir las estructuras pertinentes, e importar las funciones y bibliotecas necesarias. Es importante resaltar que este será accesible tanto desde *server.c* cómo de *socketFunctions.c.*

server.c

Este fichero implementa un servidor multihilo en C que escucha conexiones TCP de clientes en un puerto especificado. Al inicio, se definen estructuras protegidas por mutexes para manejar listas de usuarios y publicaciones, y se configura el socket del servidor para aceptar conexiones entrantes. Por cada conexión aceptada, se crea un hilo que ejecuta la función SendResponse, la cual maneja la lógica asociada a la petición recibida.

Dentro de SendResponse, se llama a la función parseMessage que se encarga de uno a uno asignar lo que ha recibido. Después de tener cada parte recibida asignada, se realizan diferentes acciones dependiendo del comando encontrado. Para cada acción, se comprueban condiciones necesarias, mediante el uso de funciones implementadas en socketFunctions.c y se responde con un código de estado que indica éxito o error, junto con información adicional en ciertos casos (por ejemplo, lista de usuarios conectados o publicaciones).

En resumen, el servidor está diseñado para ejecutarse continuamente, lo que permite tratar con clientes de forma concurrente. Este se encarga de utilizar las funciones implementadas en socketFunctions.c para encontrar el código, y en algunos casos la información, que debe mandarle al cliente.

socketFunctions.c

Este fichero implementa la lógica de las funciones necesarias para recibir e interpretar los mensajes que llegan al servidor, y realizar las acciones pertinentes dentro del servidor. La primera función implementada es parseMessage, esta recibe los datos enviados por el cliente a través de un socket y los interpreta dividiéndolos en acción, fecha, nombre de usuario y argumentos adicionales. Dependiendo de la acción (como REGISTER, PUBLISH o DELETE), se esperan más o menos líneas de información. También se proporciona una función para liberar la memoria usada por estos mensajes (freeParsedMessage).

Después se implementan funciones para registrar/dar de baja usuarios (register_user, unregister_user), manejar conexiones (register_connection, unregister_connection), comprobar estados (is_user_registered, is_user_connected), y gestionar archivos publicados (register_publication, delete_publication, get_publications). Finalmente, también incluye utilidades para enviar y recibir mensajes desde los sockets (sendByte, sendMessage, readLine). En conclusión, este fichero actúa como el núcleo de gestión de estado y comunicación del servidor.

Cliente

client.py

El archivo client.py implementa la lógica del cliente en un sistema distribuido de intercambio de archivos peer to peer. Ha sido modificado partiendo del esqueleto proveído inicialmente. Este archivo gestiona tanto la interacción con el servidor como la comunicación directa entre clientes, incluyendo control de errores, gestión de hilos, y verificación de parámetros, todo ello desde una interfaz de línea de comandos.

Está organizado en una única clase estática llamada client, que contiene todos los métodos y atributos necesarios para interactuar tanto con el servidor central como con otros clientes mediante conexiones punto a punto (P2P). No se crean instancias de esta clase; todos los métodos y variables son de clase y se utilizan de forma estática.

El cliente se comunica con el servidor central por medio de una serie de funciones que han sido implementadas en protocol.py. Es capaz de registrar y borrar usuarios, conectarse y desconectarse, publicar o eliminar archivos, y obtener listados de usuarios y contenidos. Cada comando del usuario es procesado dentro de un bucle de interpretación de comandos (shell), que lee las entradas desde la consola, verifica su sintaxis y ejecuta la acción correspondiente.

El archivo también implementa una funcionalidad P2P, que permite a los clientes conectarse entre sí para transferirse archivos directamente. Para ello, el cliente levanta un socket de

escucha en un puerto local y arranca un hilo que atiende conexiones entrantes. Cuando otro cliente solicita un archivo mediante GET_FILE, este hilo se encarga de recibir la solicitud y crear otro hilo no bloqueante para que lo transmita en bloques.

Las transferencias de archivos entre clientes están gestionadas con cuidado. El cliente que realiza la descarga establece una conexión con el cliente remoto, verifica la respuesta y guarda los datos recibidos en un archivo local. Si ocurre algún error durante la transferencia, el cliente imprime un mensaje de fallo y elimina el archivo incompleto.

El cliente también incorpora validaciones en todos los comandos, incluyendo la longitud y formato de las cadenas, y la obligatoriedad de rutas absolutas para los archivos locales. Además, utiliza un servicio web externo (mediante la librería zeep) para obtener la fecha y hora actuales, que se incluyen en las peticiones al servidor.

protocol.py

El archivo protocol.py es, básicamente, una capa intermedia entre la lógica de alto nivel del cliente y los sockets TCP. Encapsula toda la lógica de comunicación de red con el servidor, permitiendo que el cliente maneje los comandos a un nivel más alto sin preocuparse por los detalles del protocolo ni por el control de errores en los sockets. Se encarga de construir los mensajes según el protocolo, enviarlos, recibir respuestas y traducir los códigos numéricos recibidos en mensajes comprensibles.

En primer lugar, define dos constantes: MAX_LEN, que es el tamaño máximo permitido para campos como nombres de usuario o rutas, y MAX_FILE_SIZE, que representa el tamaño máximo de un bloque de archivo que puede ser leído o enviado en cada ronda de transferencia.

El archivo contiene un diccionario llamado SETTINGS, que mapea cada operación del protocolo (como register, connect, publish, etc.) a los códigos de respuesta posibles y sus mensajes correspondientes. Cada subdiccionario tiene también una clave 'default' que define el código de error por defecto si ocurre un fallo en el cliente que no sea uno de los ya contemplados.

Las funciones send_str, recv_str, recv_byte y recv_bytes manejan el envío y recepción de datos básicos por el socket. Estas funciones validan tamaños, detectan desconexiones prematuras, y tienen en cuenta la terminación de \0 para cadenas, tanto a la de hora de enviar como de recibir. En particular, recv_str lee carácter a carácter hasta detectar el byte nulo que marca el final de la cadena.

La función communicate_with_server abstrae el envío de una lista de cadenas a través del socket y la recepción del código de resultado. Devuelve una tupla que contiene el código recibido y el socket abierto, que puede ser reutilizado si es necesario recibir más datos. En caso de error, devuelve el código de error por defecto y None como socket.

A partir de ahí, se definen una serie de funciones que representan cada comando que el cliente puede enviar al servidor: register, unregister, connect, disconnect, publish, delete, list_users y list_content. Cada una prepara los parámetros del mensaje, llama a

communicate_with_server, cierra el socket si no necesita reutilizarlo y devuelve el mensaje asociado al código recibido.

Las funciones list_users y list_content requieren una segunda fase de lectura desde el socket, después de recibir el código de resultado. Si el código es 0, estas funciones leen un número entero enviado como cadena que indica cuántos elementos deben recibirse, y luego reciben una línea de información por cada elemento. En caso de error en cualquiera de estas etapas, devuelven el mensaje asociado al código de error por defecto.

webService.py

El archivo webService.py define e implementa un servicio web SOAP en python que provee la fecha y hora actual. Para lograr esto, se utiliza la biblioteca spyne para definir el servicio y el protocolo SOAP, y se emplea wsgiref para levantar un servidor web que atienda peticiones en la dirección http://127.0.0.1:5000.

Dentro del archivo se define una clase llamada DateTimeService que hereda de ServiceBase. Esta clase contiene un único método llamado get_datetime,que simplemente obtiene la fecha y hora actuales usando datetime.now() y las formatea en una cadena con el formato día/mes/año horas:minutos:segundos.

Finalmente, en la parte inferior del archivo, dentro del bloque if __name__ == '__main__':, se configura el servidor WSGI usando make_server de wsgiref.simple_server. Se habilita el registro de logs con nivel INFO y se imprime en consola la dirección del servicio y la URL del WSDL. Luego se inicia el servidor con serve_forever(), que mantiene el servicio activo a la espera de peticiones SOAP entrantes. Para que el servicio web funcione fue necesario en el cliente utilizar la biblioteca zeep que permite al cliente consumir el servicio, y poder recibir la fecha.

Descripción de la forma de compilar y obtener el ejecutable de todos los procesos involucrados.

Es importante resaltar que para que se logre ejecutar el código será necesario tener las bibliotecas *zeep* y *spyne* instaladas ya que estas son necesarias para el servidor web.

Primero debemos establecer el servidor.

Para ello, ejecute en una misma sesión de la terminal los siguientes comandos:

1. make

2. ./server -p <puerto del servidor>

Por otro lado, ya sea en la misma máquina que el servidor u otra, podemos crear

tantos clientes como queramos.

Con ese fin, primero debemos ejecutar el servicio web en una sesión de la terminal:

3. python3 webService.py

En otra sesión de la terminal dentro de la misma máquina que el servidor web,

creamos un cliente de la siguiente manera:

4. python3 client.py -s <ip del servidor> -p <puerto del servidor>

Batería de pruebas utilizadas y resultados obtenidos.

Prueba 1 — Registro, conexión y borrado de cuenta concurrente

Debe ser probada alternando la ejecución de cada comando por cliente. De este manera, podemos comprobar el correcto funcionamiento del registro y conexión más básicos y, además, comprobar que el servidor puede manejar a ambos a la vez.

Cliente1:

```
Unset

c> REGISTER esteban

c> REGISTER OK

c> CONNECT esteban

c> CONNECT OK

c> DISCONNECT esteban

c> DISCONNECT OK

c> UNREGISTER esteban

c> UNREGISTER OK
```

Cliente2:

```
Unset

c> REGISTER alex

c> REGISTER OK

c> CONNECT alex

c> CONNECT OK

c> DISCONNECT alex

c> DISCONNECT OK

c> UNREGISTER alex

c> UNREGISTER OK
```

Prueba 2 — Usuario ya existente

Primero registramos a un usuario y luego tratamos de registrar al mismo. Obtendremos un error de “USERNAME IN USE”.

Cliente1:

```
Unset

c> REGISTER ana

c> REGISTER OK
```

Cliente2:

```
Unset
c> REGISTER ana

c> USERNAME IN USE
```

Prueba 3 — Borrar a un usuario que no existe

Continuando con la anterior, también cabe la posibilidad de que tratemos de borrar un usuario que no existe.

Cliente:

```
Unset

c> UNREGISTER ana

c> USER DOES NOT EXIST
```

Prueba 4 — Conectarse a un usuario no existente

Cliente:

```
Unset

c> CONNECT ana

c> CONNECT FAIL , USER DOES NOT EXIST
```

Prueba 5 — Conectarse a un usuario ya conectado

Cliente:

```
Unset

c> REGISTER ana

c> REGISTER OK

c> CONNECT ana

c> CONNECT OK

c> CONNECT ana

c> USER ALREADY CONNECTED
```

Prueba 6 — Publicar ficheros y borrarlo. Listar usuarios y contenidos

Cliente1:

```
Unset
c> REGISTER ana

c> REGISTER OK

c> CONNECT ana
```

```
c> CONNECT OK

c> PUBLISH /archivo1 archivo1

c> PUBLISH OK

c> PUBLISH /archivo2 archivo2

c> PUBLISH OK

c> PUBLISH /archivo3 archivo3

c> PUBLISH OK
```

Cliente2:

```
Unset
c> REGISTER esteban

c> REGISTER OK

c> CONNECT esteban

c> CONNECT OK

c> LIST_USERS

c> LIST_USERS OK

        esteban 127.0.0.1 37469

        ana 127.0.0.1 42809

c> LIST_CONTENT OK

        /archivo3

        /archivo2

        /archivo1
```

Cliente1:

```
Unset
c> DELETE /archivo1
```

```
c> DELETE OK
```

Cliente2:

```
Unset

c> LIST_CONTENT ana

c> LIST_CONTENT OK

        /archivo3

        /archivo2
```

Prueba 7 — Disconnect erróneos

```
Unset

c> DISCONNECT ana

c> DISCONNECT FAIL , USER DOES NOT EXIST

c> REGISTER ana

c> REGISTER OK

c> DISCONNECT ana

c> DISCONNECT FAIL , USER NOT CONNECTED
```

Prueba 8 — Particularidades del disconnect

Si un usuario está conectado y realiza la conexión con otro nombre de usuario distinto entonces también se realiza la operación de desconexión primero del usuario conectado antes de conectarse con el otro nombre de usuario

```
Unset

c> REGISTER alex

c> REGISTER OK

c> CONNECT alex
```

```
c> REGISTER esteban

c> REGISTER OK

c> CONNECT esteban

c> DISCONNECT OK

c> CONNECT OK
```

Aparte, la operación de desconexión debería realizarse siempre que el usuario introduzca por consola el comando QUIT.

```
Unset

c> REGISTER alex

c> REGISTER OK

c> CONNECT alex

c> QUIT

c> DISCONNECT OK

+++ FINISHED +++
```

Prueba 9 — Get_file con concurrencia

Cliente1:

```
Unset

c> REGISTER sara

c> REGISTER OK

c> CONNECT sara

c> CONNECT OK

c> PUBLISH /tmp/video.mp4 desc
```

```
c> PUBLISH OK
```

Cliente2:

```
Unset

c> REGISTER tom

c> REGISTER OK

c> CONNECT tom

c> CONNECT OK
```

Cliente3:

```
Unset

c> REGISTER OLI

c> REGISTER OK

c> CONNECT OLI

c> CONNECT OK
```

Teniendo en cuenta que estos vídeos son archivos pesados y que, por tanto, su descarga tarda cierto tiempo, ejecutamos de manera simultánea en el Cliente2 y Cliente3 lo siguiente:

```
Unset
GET_FILE sara /tmp/video.mp4 /tmp/tom_video.mp4
```

En ambos obtendremos el archivo en el sistema y, además, feedback en el shell:

```
Unset
> GET_FILE OK
```

Conclusiones, problemas encontrados, cómo se han solucionado, y opiniones personales.

En general, pensamos que este proyecto fue un trabajo que nos permitió entender más en detalle, y mediante la práctica, todo lo cubierto en la asignatura. Sin embargo, también se presentó como un trabajo de mucha exigencia y bastantes horas de dedicación. Tanto así, que al final no logramos implementar un proceso remoto que funcionase correctamente con el resto del código y se decidió no entregar esa parte del proyecto. De la misma manera, en las primeras dos partes del proyecto también nos encontramos múltiples problemas, no obstante estos lograron ser solucionados mediante largas sesiones de depuración.

Antes de hablar de los problemas que encontramos durante el proceso, es importante resaltar que ciertas explicaciones de errores descritas en el enunciado no fueron implementadas ya que sentimos que son redundantes, o sencillamente no alcanzables. Esto ocurrió principalmente con los *printf* de errores del LIST_CONTENT, PUBLISH y DELETE. Un ejemplo de ello, es el error causado cuando un usuario intenta hacer un LIST_CONTENT sin estar registrado, aquí ni siquiera se le ha asignado un nombre a este usuario.

El primer gran problema que nos encontramos fue que en múltiples ocasiones nuestro servidor entró en loops infinitos y no sabíamos en qué parte del código ocurría esto. Fue necesario agregar impresiones a lo largo de todo el código para lograr identificar dónde estaban ocurriendo estos loops y lograr solucionar el problema. Más adelante, por algún motivo que aún desconocemos, descubrimos que en algunas ocasiones, existían *printfs* que no mostraban nada en la pantalla pese a estar siendo ejecutados. Esto fue realmente problemático ya que hizo más difícil la tarea de depurar el código. Para solucionar esto, utilizamos el comando *fflush(stdout)* para forzar la impresión del standard output.

Nos topamos con múltiples otro problemas, sin embargo, el qué más problemas causó fue cuando se estaba probando el código en *guernika.* Por alguna razón, muchas veces que se intentaba abrir la aplicación VisualStudio esta no se abría, y era necesario cerrar la pestaña y volver a intentarlo. Esto nos hizo perder bastante tiempo ya que no era posible saber cuándo iba a funcionar y cuándo no. Asimismo, al intentar utilizar GitHub aquí, los ficheros aparecían siempre en la sección de “nuevos cambios” por lo que nunca era posible realizar un *pull* del repositorio, siempre decía que había que gestionar estos cambios antes. Cada vez que queríamos probar un código, era necesario clonar el repositorio de Git en la misma carpeta, ya que este *pull* no funcionaba (y realizar constantemente *merge* era complicado).

Finalmente, durante la prueba de la función GET_FILE tuvimos un inconveniente. Al realizar la operación el cliente recibía que se había completado correctamente, pero en el directorio no aparecía el fichero. Más extraño aún, una docena de minutos después, aparecía de la nada. Estuvimos un par de horas depurando para entender qué ocurría, hasta que descubrimos (mediante comandos de linux) que el fichero realmente si se estaba copiando inmediatamente, pero por alguna razón aparecía copiado en la interfaz después de varios minutos.

En definitiva, aunque fue un proyecto que nos puso a prueba en muchos sentidos y nos generó varios dolores de cabeza, también consideramos que fue una experiencia valiosa de aprendizaje. Sentimos que, a pesar de las dificultades y del hecho de que no logramos entregar completamente la funcionalidad remota, aprendimos muchísimo tanto en términos técnicos como en habilidades de trabajo en equipo y gestión del tiempo. Creemos que estos desafíos, aunque frustrantes en el momento, terminan dejando las lecciones más duraderas.