Práctica final: sistema peer-to-peer

Universidad Carlos III

Grado Ingeniería Informática 2023-24

Procesadores del lenguaje (Grupo 81)

Práctica realizada por:

- **Tomás Mendizábal** (NIA: 100461170, Email: 100461170@alumnos.uc3m.es)
- **Alejandro Díaz Cuéllar** (NIA: 100472173, Email: 100472173@alumnos.uc3m.es)

TABLA DE CONTENIDO

1. Introducción

El objetivo de este documento es servir como documento explicativo de la práctica final de la asignatura: un sistema peer-to-peer. En esta memoria comentaremos la implementación de los diferentes apartados de la práctica, tanto en el lado cliente como en el lado servidor, así como la forma de compilar el proyecto y la batería de pruebas diseñada para evaluar el mismo.

2. Explicación del funcionamiento del sistema

Cliente Python

La parte del cliente consiste en un solo archivo llamado “client.py". Este archivo contiene una clase *Client* que contiene todas las funciones que puede llamar un cliente. En primer lugar, se encuentra el argument parser. Esta función verifica que se llame al archivo con el siguiente formato: client.py -s *IP* -p *puerto*. El cliente por tanto tendrá que disponer de la IP y del puerto donde se ejecuta el servidor. Si este formato no se cumple el cliente imprimirá por terminal la forma correcta de ejecutar el sistema y terminará.

Terminada la comprobación el cliente ejecutará una función shell. Esta función espera en un bucle infinito hasta recibir los comandos por parte del cliente. Cuando se entra al bucle la función crea el socket que se va a utilizar para contactar con el servidor. A continuación se espera que el usuario teclee las distintas funciones revisando que hayan estado bien escritas. Al recibir una función se hace uso de *Socket.connect()* con la IP y el puerto como argumentos (los que dio el usuario en línea de comandos). Se llama a la función y finalmente se cierra el socket. En la siguiente sección se explicará brevemente el protocolo seguido de cada función del lado del servidor. Antes de hablar de las funciones cabe destacar que todas ellas empiezan usando el servicio web para obtener la fecha y hora.

Función register

El método “register” recibe como parámetro el nombre del usuario a registrar. Usando el socket envía los siguientes campos: código de operación, fecha_hora y nombre de usuario. Finalmente, recibe un entero indicando el estado de la operación.

Función unregister

El método “unregister” recibe como parámetro el nombre del usuario a eliminar. Usando el socket envía los siguientes campos: código de operación, fecha_hora y nombre de usuario. Finalmente, recibe un entero indicando el estado de la operación. Se tiene que resaltar que si un usuario está conectado no podrá ser eliminado ya que puede ser un usuario conectado a otra sesión.

Función connect

El método “connect” recibe como parámetro el nombre del usuario para conectar. Se empieza creando un socket (connect) para el usuario conectado. Para construir la tupla *addr* se utiliza el IP pasado por línea de comandos y el puerto 0. Se usa 0 porque al hacer la función Socket.bind() si el campo de puerto es 0 entonces se escogerá un puerto libre para montar el socket. Después se obtiene el puerto y el IP elegido del socket para enviarlo al servidor. Antes, se crea el thread para atender las peticiones get_file. Más tarde, el socket (cliente) envía los siguientes campos: código de operación, fecha_hora, usuario y puerto de socket (connect). Finalmente, recibe un entero indicando el estado de la operación. Si todo va bien se guarda el nombre del usuario conectado y se da inicio al thread.

Función disconnect

El método “disconnect” recibe como parámetro el nombre del usuario a desconectar. Antes de enviar la información se comprueba si el usuario pasado es el usuario conectado en la sesión. Esto se hace para evitar desconectar a usuarios en sesiones ajenas. Usando el socket envía los siguientes campos: código de operación, fecha_hora y nombre de usuario. Finalmente, recibe un entero indicando el estado de la operación. Si todo va bien usará la función Socket.getsockname() para poder conectarse al socket (connect). Luego, enviará un mensaje especial (la cadena: “end”) para terminar la ejecución del socket (connect). Adicionalmente, hará un Thread.join() y guardará None en la variable del nombre de usuario.

Función publish

El método “publish” recibe como parámetro la ruta absoluta del fichero a publicar y la descripción del mismo. Primero se comprueba si el usuario está conectado en la sesión. Si lo está, se usa el socket para enviar los siguientes campos: código de operación, fecha_hora, nombre de usuario, la ruta del archivo y la descripción. Finalmente, recibe un entero indicando el estado de la operación.

Función delete

El método “delete” recibe como parámetro la ruta absoluta del fichero a borrar. Primero se comprueba si el usuario está conectado en la sesión. Si lo está, se usa el socket para enviar los siguientes campos: código de operación, fecha_hora, nombre de usuario y la ruta del archivo. Finalmente, recibe un entero indicando el estado de la operación.

Función list_users

El método “list_users” no recibe parámetros. Primero se comprueba si el usuario está conectado en la sesión. Si lo está, se usa el socket para enviar los siguientes campos: código de operación y fecha_hora. Finalmente, recibe un entero indicando el estado de la operación. Si todo va bien se recibirá además un entero con el número de usuarios. Entonces, se esperará por el nombre, la IP y el puerto de cada uno de estos usuarios. Se mostrará información de cada usuario en una línea.

Función list_content

El método “list_content” recibe como parámetro la ruta absoluta del fichero a borrar. Primero se comprueba si el usuario está conectado en la sesión. Si lo está, se usa el socket para enviar los siguientes campos: código de operación, fecha_hora, nombre del usuario que realiza la operación y nombre del usuario del que se quiere obtener los contenidos. Finalmente, recibe un entero indicando el estado de la operación. Si todo va bien se recibirá además un entero con el número contenidos del usuario. Entonces, se esperará por la ruta y la descripción de los contenidos del usuario. Se mostrará cada publicación en una línea.

Función get_file

El método “get_file” recibe como parámetros el usuario dueño de la publicación, la ruta absoluta de la publicación y la ruta absoluta donde recibir el archivo en la máquina local. Primero se comprueba si el usuario está conectado en la sesión. Si lo está, se usa el socket para enviar los siguientes campos: código de operación, fecha_hora, nombre del usuario y la ruta de la publicación. Finalmente, recibe un entero indicando el estado de la operación. Si todo va bien recibirá la IP y el puerto de usuario dueño de la publicación, con lo que podrá ya realizar el get_file propiamente dicho. Con un socket se conecta al cliente conectado (usando la IP y el puerto recibido del servidor). Después, se manda un mensaje con la ruta del fichero pedido y se espera recibir el contenido del fichero. Usando la ruta pasada (para el archivo local) se abre un nuevo fichero y se escribe el contenido recibido. Si no se pasa ninguna ruta usará el directorio actual. Al final se cierra el fichero y el socket creado.

Función listen

La función listen es la que ejecuta la thread de el cliente conectado. Para empezar usa la sentencia Socket.listen() y entra en un bucle infinito. En este bucle espera a peticiones nuevas usando Socket.accept(). Cada vez que recibe una petición lee el mensaje y comprueba si se trata de la cadena “end”. Si se cumple cerrará los sockets y acabará la ejecución. Si no, abrirá el fichero pedido, mandará los contenidos al cliente, cerrará la conexión y volverá al bucle.

Web Service Python

Para usar el servicio web en Python se utiliza SOAP. Para el transporte de mensajes se utiliza el protocolo HTTP con estilo RPC. Para la codificación de los mensajes se utiliza WSDL. El servidor “web_service.py” crea un servidor con el puerto pasado por línea de comandos y espera a recibir peticiones. Este servidor se crea usando una aplicación que contiene la clase con una función para devolver la fecha y hora actual. En la parte del cliente se creará una dirección web (wsdl) utilizando el mismo número de puerto donde se montó el servidor. Usando el módulo Zeep se creará un objeto soap usando dicha dirección. Finalmente, se usará el objeto para invocar la función de obtener fecha y hora.

Servidor C

La parte del servidor tiene un archivo llamado servidor.c que actuará como servidor para las peticiones de usuario, pero también actuará como cliente para las peticiones RPC. En primer lugar, se declaran varias variables necesarias para manejar el servidor. Se declara dos mutex (uno para copia y otro para el almacén), una variables servidor, dos enteros para los sockets (cliente, y servidor), una puntero para el almacén junto con el número de elementos que tiene y el máximo que puede guardar. Al iniciar el servidor se hace una comprobación de los argumentos dados por el usuario. Se comprueba que el primer argumento sea "-p" y el segundo (puerto) sea un número entre 1024 y 65535. Luego se inicializa el almacén con un malloc y la función de carga de almacén. Dicha función abre un fichero (o lo crea) y copia toda la información de los usuarios almacenada.

Luego se abre el crea el socket del servidor para poder recibir peticiones y se imprime el nombre de la máquina junto con el puerto. Aquí es cuando se entra en un bucle infinito para recibir peticiones. Por cada petición recibida se crea un hilo y se pasa como argumento el socket de cliente que se acaba de crear.

Al iniciar la función de tratar peticiones se utiliza un mutex para copiar el socket a una variable local. A continuación se crea un cliente RPC usando como IP el valor de una variable de entorno llamada "IP_TUPLAS". La variable tendrá que contener la IP del servidor RPC. Posteriormente, se recibe el código de operación y el objeto fecha_hora. Con este código se invocará a las distintas funciones del cliente. Se pasa como parámetro el descriptor del socket y la dirección de una estructura que se usará en RPC. En dicha estructura se almacenará el nombre del usuario y el nombre del fichero si se trata de publish o delete. Después de ejecutar las funciones se pasará el entero de respuesta (menos el caso de list_users, list_conten y get_file) y también se invocará la funcionalidad RPC. Al final se cerrará el socket cliente. Ahora se expondrá el protocolo de cada función en el lado del servidor sin entrar en mucho detalle.

Estructura tupla

Antes de empezar con las funciones propiamente dichas, queremos explicar una de las partes más importantes del servidor, que es la tupla que se almacena en el servidor. Esta estructura de datos es esencial para entender el funcionamiento del servidor.

```C/C++
struct tupla {
   char cliente[MAX_STR];
   struct file files[MAX_STR];
   int file_count;
   int connected;
   char ip[MAX_STR];
   int puerto;};
```

Como vemos esta tupla contiene:

- El nombre del cliente almacenado
- Una lista con sus ficheros publicados
- Un entero que contiene el número de ficheros publicados, lo que ayudará en ciertas operaciones
- Un entero 0 o 1 que codificará si el cliente está conectado
- Una cadena de caracteres para almacenar su IP en caso de estar conectado
- Un entero que almacena su número de puerto en caso de estar conectado

Función register

Al iniciar la función register se obtiene el nombre de usuario usando el socket cliente. Se comprobará si el usuario ya está registrado. Si es así se enviará un código de error. Si no se inserta al usuario en el almacén y se devuelve un 0.

Función unregister

Al iniciar la función unregister se obtiene el nombre de usuario usando el socket cliente.
Se comprobará si el usuario está registrado. Si no lo está se enviará un código de error. Si lo esta se eliminará del almacén y se devuelve un 0.

Función connect

En la función connect se recibe el nombre de usuario y un número de puerto del socket connect. Ahora se hace se comprueba si existe el usuario y si ya está conectado. Si hay error se devuelve 1 o 2 respectivamente. En este momento se obtiene la IP del socket cliente usando getpeername, se conecta al usuario (copiando los valores y cambiando el atributo conectado a 1) y se devuelve 0.

Función disconnect

La función disconnect recibe un nombre de usuario del socket cliente. Primero, comprueba si existe el usuario y si no está conectado. Si hay error devuelve 1 o 2 respectivamente. Finalmente, se establece al usuario como no conectado y se devuelve un 0.

Función publish

La función publish recibe un usuario, ruta de archivo y una descripción. Aquí se comprueba que el usuario exista, esté conectado y que el archivo no esté ya publicado. Si hay error se devuelve 1, 2 o 3 respectivamente. Finalmente, se le atribuye la ruta y la descripción al usuario y se devuelve un 0.

Función delete

La función publish recibe un usuario y una ruta de archivo. Aquí se comprueba que el usuario exista, esté conectado y que el archivo esté publicado bajo ese nombre. Si hay error se devuelve

1, 2 o 3 respectivamente. En otro caso, se borra el archivo del usuario y se devuelve un 0. Se asume que se pueden borrar archivos de usuarios que no sea el usuario conectado.

Función list_users

La función list_users no recibe nada. Primero se comprueba si hay clientes y si hay clientes conectados. Si hay error se envía un "1" o "2" con el socket, respectivamente. Si no se envía un "0". Luego se envía el número de clientes conectados y por cada cliente también se envía el nombre, IP y puerto.

Función list_content

La función list_content recibe el nombre del cliente que opera la función y el nombre del usuario a extraer los datos. Se verifica que el cliente esté registrado, que el cliente esté conectado y que el usuario esté registrado. Si hay error se envía "1", "2" o "3" con el socket, respectivamente. Si no se envía un "0". Luego se envía el número de archivos y por cada archivo también se envía el nombre y descripción.

Función get_file

La función get_file recibe el nombre del cliente y la ruta del archivo. Se verifica que el cliente existe, que el cliente esté conectado y que el archivo existe. Si hay error se envía "2", "2" o "1", respectivamente. Posteriormente, se envía un "0" y una cadena con el IP y el puerto del cliente. Como no está dentro del protocolo el nombre del usuario no se envía al servidor aunque si se comprueba en el lado cliente que sea el usuario conectado.

Servidor RPC

Para poder usar este tipo de llamadas, se creó un servicio RPC usando un archivo llamado "operaciones.x". En él hay una función llamada send_op_log que recibe como parámetro una estructura. Al usar la llamada RPCgen se crean varios archivos. En el archivo titulado "operaciones_sever.c" se implementa la función. Cabe destacar que la implementación es sencilla ya que solo se debe convertir la operación a mayúsculas e imprimirla por pantalla. El fichero que actuaba de cliente se borró y toda su funcionalidad se trasladó a "servidor.c". De esta manera nuestro servidor actúa como cliente de RPC. Al comienzo de la función para tratar la petición se declaran todas las variables RPC. Luego a medida que avanza la función se llena la estructura con los correspondientes datos y al final se hace la llamada RPC.

3. Compilación y ejecución del sistema

Pasos previos en Python

Antes de ejecutar se debe seguir estos pasos para que la parte relativa a python funcione:

1. Hacer *$ cd ssdd_p2_100461170_100472173*
2. Crear un "virtual environment" llamado venv usando el comando: *$ python3 -m venv venv*
3. Activar el venv usando el comando: *$ source venv/bin/activate*
4. Instalar todos los requisitos del sistema usando: *$ pip install -r requirements.txt*

Nota: Los pasos 1 y 2 (crear un virtual environment) son opcionales, pero recomendables

Guía de compilación y ejecución del sistema

Una vez que se hayan realizado los pasos previos se puede seguir con estos pasos para ejecutar una sesión. Esta es la forma más cómoda de ejecutar el servicio.

1. Abrir cuatro terminales
2. En todas las terminales hacer *$ cd ssdd_p2_100461170_100472173*
3. La primera terminal tendrá el servidor del cliente. Se deber usar:
    a. *$ make*
    b. *$ ./execute_server.sh*
        i. Esto es idéntico a utilizar: *$ export IP_TUPLAS=localhost; ./servidor_cliente -p 3000*
4. En la segunda terminal estará el servidor RPC. El comando es:
    a. *$ ./servidor_rpc*
5. En la tercera terminal estará el servicio web. Los comandos son:
    a. *$ source venv/bin/activate*
    b. *$ python3 web_service.py -p 8000*
6. En la cuarta terminal estará el cliente en python. Los comandos son:
    a. *$ source venv/bin/activate*
    b. $ python3 client.py -s localhost -p 3000 -ws 8000

4. Batería de pruebas para la evaluación del sistema

Para evaluar las pruebas se aportarán comandos en texto para pasar como input al cliente. Para ejecutar las pruebas se pueden guardar en un fichero de texto (ej: test1.txt) y ejecutar usando el comando *$ cat test1.txt. | python3 client.py -s localhost -p 3000 -ws 8000*

**** Si ocurren errores al ejecutar las pruebas pueden deberse a que hay contenido insertado en el almacén (usuarios y/o archivos). Esto puede dar errores, especialmente en el connect. Para borrar el almacén simplemente eliminar (o borrar el contenido) el archivo almacen.txt después de cerrar el servidor. Asumimos que para cada prueba se parte de un almacén vacío**

Pruebas de 1 sesión

Función register y unregister

```
Unset
register user1
register user2
register user3
register user1
unregister user1
unregister user2
unregister user4
unregister user3
quit
```

```
c> register user1
register user2
register user3
register user1
unregister user1
unregister user2
unregister user4
unregister user3
quit
c> REGISTER OK
c> c> REGISTER OK
c> c> REGISTER OK
c> c> USERNAME IN USE
c> c> UNREGISTER OK
c> c> UNREGISTER OK
c> c> USER DOES NOT EXIST
c> c> UNREGISTER OK
c> +++ FINISHED +++
```

Función connect y disconnect

```
Unset
register user1
register user2
connect user3
connect user1
connect user2
disconnect user3
disconnect user2
disconnect user1
unregister user1
unregister user2
quit
```

```
c> register user1
register user2
connect user3
connect user1
connect user2
disconnect user3
disconnect user2
disconnect user1
unregister user1
unregister user2
quit
c> REGISTER OK
c> c> REGISTER OK
c> c> CONNECT FAIL, USER DOES NOT EXIST
c> c> CONNECT OK
c> c> CLIENT ALREADY CONNECTED
c> c> DISCONNECT FAIL / USER NOT CONNECTED
c> c> DISCONNECT FAIL / USER NOT CONNECTED
c> c> DISCONNECT OK
c> c> UNREGISTER OK
c> c> UNREGISTER OK
c> +++ FINISHED +++
```

Funciones publish y delete

Unset

```
publish file1 descr1
register user1
publish file1 descr1
connect user1
publish file1 descr1
publish file1 descr2
disconnect user1
delete file1
connect user1
delete file2
delete file1
disconnect user1
delete file1
unregister user1
quit
```

```
c> publish file1 descr1
c> PUBLISH FAIL, CLIENT NOT CONNECTED
c> register user1
c> REGISTER OK
c> publish file1 descr1
c> PUBLISH FAIL, CLIENT NOT CONNECTED
c> connect user1
c> CONNECT OK
c> publish file1 descr1
c> PUBLISH OK
c> publish file1 descr2
c> PUBLISH FAIL, CONTENT ALREADY PUBLISHED
c> disconnect user1
c> DISCONNECT OK
c> delete file1
c> DELETE FAIL, CLIENT NOT CONNECTED
c> connect user1
c> CONNECT OK
c> delete file2
c> DELETE FAIL, CONTENT NOT PUBLISHED
c> delete file1
c> DELETE OK
c> disconnect user1
c> DISCONNECT OK
c> delete file1
c> DELETE FAIL, CLIENT NOT CONNECTED
c> unregister user1
c> UNREGISTER OK
c> quit
+++ FINISHED +++
```

Función list_users

```
Unset
list_users
register user1
register user2
list_users
connect user1
list_users
disconnect user1
unregister user1
unregister user2
quit
```

```
c> list_users
c> LIST_USERS FAIL , CLIENT NOT CONNECTED
c> register user1
c> REGISTER OK
c> register user2
c> REGISTER OK
c> list_users
c> LIST_USERS FAIL , CLIENT NOT CONNECTED
c> connect user1
c> CONNECT OK
c> list_users
c> LIST_USERS OK
user1 127.0.0.1 48759
c> disconnect user1
c> DISCONNECT OK
c> unregister user1
c> UNREGISTER OK
c> unregister user2
c> UNREGISTER OK
c> quit
+++ FINISHED +++
```

Función list_content

```
Unset
list_content user1
register user1
list_content user1
connect user1
list_content user2
publish file1 descr1
publish file2 descr2
list_content user1
disconnect user1
register user2
connect user2
list_content user1
disconnect user2
unregister user1
unregister user2
quit
```

```
c> list_content user1
register user1
list_content user1
connect user1
list_content user2
publish file1 descr1
publish file2 descr2
list_content user1
disconnect user1
register user2
connect user2
list_content user1
disconnect user2
unregister user1
unregister user2
quit
c> LIST_CONTENT FAIL , CLIENT NOT CONNECTED
c> c> REGISTER OK
c> c> LIST_CONTENT FAIL , CLIENT NOT CONNECTED
c> c> CONNECT OK
c> c> LIST_CONTENT FAIL , REMOTE USER DOES NOT EXIST
c> c> PUBLISH OK
c> c> PUBLISH OK
c> c> LIST_CONTENT OK
file1 "descr1"
file2 "descr2"
c> c> DISCONNECT OK
c> c> REGISTER OK
c> c> CONNECT OK
c> c> LIST_CONTENT OK
file1 "descr1"
file2 "descr2"
c> c> DISCONNECT OK
c> c> UNREGISTER OK
c> c> UNREGISTER OK
c> +++ FINISHED +++
```

Función get_file

En estas pruebas se usará el archivo de ejecución como directorio absoluto. Antes de ejecutar las pruebas se creará un archivo de prueba.

```
Unset
echo "Esto es un archivo prueba" > prueba.txt
```

```
Unset
get_file user1 prueba.txt copy_prueba.txt
register user1
get_file user1 prueba.txt copy_prueba.txt
connect user1
publish prueba.txt descr1
get_file user1 prueba.txt copy_prueba.txt
publish /dir/prueba.txt descr1
get_file user1 /dir/prueba.txt file2
```

```
get_file user1 prueba.txt /dir/copy_prueba.txt
disconnect user1
unregister user1
quit
```

```
c> get_file user1 prueba.txt copy_prueba.txt
register user1
get_file user1 prueba.txt copy_prueba.txt
connect user1
publish prueba.txt descr1
get_file user1 prueba.txt copy_prueba.txt
publish /dir/prueba.txt descr1
get_file user1 /dir/prueba.txt file2
get_file user1 prueba.txt /dir/copy_prueba.txt
disconnect user1
unregister user1
quit
c> GET_FILE , USER NOT CONNECTED
c> c> REGISTER OK
c> c> GET_FILE , USER NOT CONNECTED
c> c> CONNECT OK
c> c> PUBLISH OK
c> GET_FILE OK
c> c> PUBLISH OK
c> c> GET_FILE FAIL / FILE NOT EXIST
c> c> GET_FILE FAIL
c> c> DISCONNECT OK
c> c> UNREGISTER OK
c> +++ FINISHED +++
```

Función quit

Esta prueba es conjunta, se prueba la primera parte y luego la segunda.

```
Unset
register user1
connect user1
quit
```

```
Unset
connect user1
disconnect user1
unregister user1
quit
```

```
c> register user1
connect user1
quit
c> REGISTER OK
c> c> CONNECT OK
c> c> DISCONNECT OK
+++ FINISHED +++
(venv) ~/Desktop/sistemas_distribuidos/Pra
3000 -ws 8000
c> connect user1
disconnect user1
unregister user1
quit
c> CONNECT OK
c> c> DISCONNECT OK
c> c> UNREGISTER OK
c> +++ FINISHED +++
```

Pruebas de 2 sesiones

Estas pruebas están pensadas para probar las funcionalidad de más de una sesión a la vez. Cada cuadro de código representará una sesión Para probarlas seguir este orden:

- Intercalar las sentencias de la primera sesión con las de la segunda. Es decir ejecutar una línea de la primera, seguida de una línea de la segunda, así sucesivamente.

Se ejecutan con: *$ python3 client.py -s localhost -p 3000 -ws 8000*

Funciones register y unregister

```
Unset
register user1
unregister user2
quit
```

```
Unset
register user2
unregister user1
quit
```

```
c> register user1
c> REGISTER OK
c> unregister user2
c> UNREGISTER OK
c> quit
+++ FINISHED +++
```

```
c> register user2
c> REGISTER OK
c> unregister user1
c> UNREGISTER OK
c> quit
+++ FINISHED +++
```

Funciones connect y disconnect

```
Unset
register user1
connect user1
disconnect user1
unregister user1
quit
```

```
Unset
register user2
connect user2
disconnect user2
unregister user2
quit
```

```
c> register user1
c> REGISTER OK
c> connect user1
disconnect user1
unregister user1
quit
c> CONNECT OK
c> c> DISCONNECT OK
c> c> UNREGISTER OK
c> +++ FINISHED +++
```

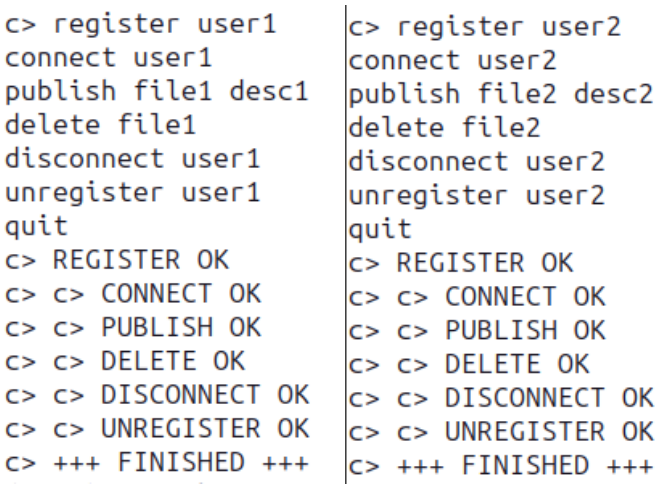
```
c> register user2
c> REGISTER OK
c> connect user2
disconnect user2
unregister user2
quit
c> CONNECT OK
c> c> DISCONNECT OK
c> c> UNREGISTER OK
c> +++ FINISHED +++
```

Funciones publish y delete

```
Unset
register user1
connect user1
```

```
publish file1 desc1
delete file1
disconnect user1
unregister user1
quit
```

```
Unset
register user2
connect user2
publish file2 desc2
delete file2
disconnect user2
unregister user2
quit
```



Función list_users

```
Unset
register user1
connect user1
list_users
disconnect user1
unregister user1
quit
```

```
Unset
register user2
connect user2
list_users
disconnect user2
unregister user2
quit
```

```
c> register user1
c> REGISTER OK
c> connect user1
c> CONNECT OK
c> list_users
c> LIST_USERS OK
user1 127.0.0.1 53737
user2 127.0.0.1 59117
c> disconnect user1
unregister user1
quit
c> DISCONNECT OK
c> c> UNREGISTER OK
c> +++ FINISHED +++
```

```
c> register user2
c> REGISTER OK
c> connect user2
c> CONNECT OK
c> list_users
c> LIST_USERS OK
user1 127.0.0.1 53737
user2 127.0.0.1 59117
c> disconnect user2
unregister user2
quit
c> DISCONNECT OK
c> c> UNREGISTER OK
c> +++ FINISHED +++
```

Función list_content

```
Unset
register user1
connect user1
publish file1 desc1
list_content user2
delete file1
disconnect user1
unregister user1
quit
```

```
Unset
register user2
connect user2
publish file2 desc2
list_content user1
list_content user1
```

```
disconnect user2
unregister user2
quit
```

```
c> register user1
c> REGISTER OK
c> connect user1
c> CONNECT OK
c> publish file1 desc1
c> PUBLISH OK
c> list_content user2
c> LIST_CONTENT OK
file2 "desc2"
c> delete file1
c> DELETE OK
c> disconnect user1
c> DISCONNECT OK
c> unregister user1
c> UNREGISTER OK
c> quit
+++ FINISHED +++
```

```
c> register user2
c> REGISTER OK
c> connect user2
c> CONNECT OK
c> publish file2 desc2
c> PUBLISH OK
c> list_content user1
c> LIST_CONTENT OK
file1 "desc1"
c> list_content user1
c> LIST_CONTENT OK
c> disconnect user2
c> DISCONNECT OK
c> unregister user2
c> UNREGISTER OK
c> quit
+++ FINISHED +++
```

Función get_file

Unset

```
echo "Esto es un archivo prueba" > prueba.txt
```

Unset

```
register user1
connect user1
publish prueba.txt fichero de prueba
disconnect user1
unregister user1
quit
```

Unset

```
register user2
connect user2
get_file user1 prueba.txt copy_prueba.txt
disconnect user2
```

```
unregister user2
quit
```

```
c> register user1
c> REGISTER OK
c> connect user1
c> CONNECT OK
c> publish prueba.txt fichero de prueba
c> PUBLISH OK
c> disconnect user1
c> DISCONNECT OK
c> unregister user1
c> UNREGISTER OK
c> quit
+++ FINISHED +++
```

```
c> register user2
c> REGISTER OK
c> connect user2
c> CONNECT OK
c> get_file user1 prueba.txt copy_prueba.txt
GET_FILE OK
c> disconnect user2
c> DISCONNECT OK
c> unregister user2
c> UNREGISTER OK
c> quit
+++ FINISHED +++
```

5. Conclusiones

Para completar esta práctica hemos tenido que resolver varios problemas y errores. Uno de los problemas que más dificultades nos dio fue la comunicación usando sockets entre python y c. Al comienzo utilizamos la función de Sokcet.recv(1024) para recibir 1024 caracteres. Sin embargo esta implementación no es correcta, ya que es mejor recibir los mensajes byte a byte. También se destaca que para los envíos de mensajes teníamos mucho código repetido que hacía lo mismo. Para resolver esto se implementó una función que codifica los datos y los envía a través del socket. Finalmente, un problema complicado fue la interpretación de los protocolos. Para cada función se tuvo que implementar para que no haya errores y se siga el protocolo a rajatabla.

En cuanto a qué nos ha parecido la práctica, hemos de agradecer el formato de entrega de trabajos seguido durante el curso. Nos parece que haber podido ir entregando ejercicios que eran fundamentales para luego desarrollar esta práctica nos ha ayudado a entender mejor todos los conceptos y a poder implementar esta práctica rápidamente, creemos que es todo un acierto. Además sentimos que a lo largo de la asignatura hemos aprendido mucho y estamos muy contentos con el desarrollo de la asignatura.