uc3m | Universidad Carlos III de Madrid

Proyecto Final
Sistemas Distribuidos

Grupo reducido: 81

Adrian Stefan Giurca
(100495924, 100495924@alumnos.uc3m.es)

María de los Ángeles Vázquez Goncalves
(100495839, 100495839@alumnos.uc3m.es)

Índice

1. Introducción

En este documento, vamos a explicar el diseño y funcionamiento de nuestro sistema del proyecto final, el cual incluye una arquitectura multihilo con sockets TCP, así como un servicio web y un servidor RPC.

2. Descripción del código

2.1. Parte 1 - Sistema peer-to-peer

2.1.1. Descripción general

Nuestro sistema básico está compuesto por un cliente (client.py) y un servidor (servidor.c). El protocolo de comunicación sigue el que se sugiere en el enunciado de la práctica: cada pieza de información independiente se envía en un mensaje separado. En algunas ocasiones se envían strings y en otras números enteros. Sin embargo, en ambos casos debemos transformarlos en strings antes de mandarlos por los sockets, en el caso de C, y a tipo binario en el caso de Python.

La manera que el servidor tiene para almacenar la información sobre qué usuarios están registrados, cuáles están conectados y los archivos publicados por cada uno de ellos es mediante directorios y ficheros. Usamos un directorio `registered_users` para almacenar ficheros con nombre igual a cada nombre de usuario registrado, y el contenido de estos es, por cada línea (`\n`), un nombre de archivo publicado junto con su descripción separados por un espacio. Luego, usamos un directorio `active_users` para almacenar de la misma manera los usuarios conectados, y su contenido esta vez es "`IP/puerto`" del cliente.

El esqueleto del cliente nos fue dado y lo que hicimos fue rellenar los métodos llamados por el método `shell()` con el código de cada servicio ofrecido y crear los métodos y atributos que fuimos necesitando. La estructura general que siguen todos los comandos es la de un cliente TCP, utilizando la librería de python "socket". Cabe destacar además que al principio de todas las funciones verificamos que ninguno de sus parámetros de entrada provenientes de la línea de comandos sea superior a 256 bytes, esto es, 255 caracteres, puesto que el carácter `'\0'` ocupa 1 byte:

1. Crear el socket con la función `socket()`.
2. Establecer la conexión con el servidor con `conect()`.
3. Escribir todos los mensajes necesarios en el socket con unos métodos que añadimos `write_string()` y `write_number()`, dependiendo de si se manda un string o un entero. Internamente usan la función `sendall()`. Más específicamente, `write_string()` utiliza `encode()` para transformar el string a binario y luego lo manda por el socket con `sendall()`, y luego manda el carácter `'\0'` de la misma manera, `write_number()` convierte el número a string y luego hace una llamada a `write_string()`.

4. Leer todos los mensajes necesarios en el socket con unos métodos que añadimos `read_string()` y `read_number()`, dependiendo de si se espera un string o un entero. Internamente usan la función `recv()`. Más específicamente, `read_string()` entra en un bucle while infinito en el que lee byte a byte (caracter a caracter) el contenido del socket con `recv()`, y sale del bucle cuando lee el carácter `'\0'` de la misma manera, momento en el que usa `decode()` para transformar los bytes recibidos a un string. `read_number()` hace una llamada a `read_string()` y luego transforma el string a un número en base decimal.
5. Cerrar la conexión con `close()`.

Para desarrollar el servidor en el lenguaje C nos basamos en el servidor multihilo mediante thread pools con sockets TCP que desarrollamos en el ejercicio evaluable 2, para adaptar el servidor al protocolo necesario redefinimos la `struct peticion` del archivo `comm_sock.h` para guardar todos los distintos strings que el servidor va a recibir del cliente y donde se guardan los argumentos de todos los distintos comandos que el servidor ofrece, esta struct es necesaria no porque se envíe al servidor mediante sockets si no para poder insertar la petición en el buffer circular desde el cual los threads del servidor leen estas peticiones y las atienden para los clientes. También se usaron los archivos `lines.c` y `lines.h` que se nos proporcionaron en uno de los laboratorios de esta asignatura para leer y escribir en sockets para enviar y recibir información, por otro lado, para rellenar las peticiones sea desarrollado una función `read_to_pet()` en el archivo `protocol.c` que se encarga de leer a través de un socket los distintos campos que va a recibir el servidor, cambiando estos de distintas formas dependiendo del comando que se reciba y por último rellenando los campos de una struct peticion. Cabe destacar que la implementación de todas las funciones que implementan los servicios que ofrece el servidor (`REGISTER`, `UNREGISTER`, `CONNECT`, `DISCONNECT`, `PUBLISH`, `DELETE`, `LIST_USERS` y `LIST_CONTENT`) las hemos incluido en los archivos `manage_platform.c` y `manage_platform.h`. El flujo de ejecución del hilo main del servidor se desarrolla de la siguiente manera:

1. Se comprueba que el número de argumentos recibidos por el programa es el correcto, se convierte el puerto sobre el cual se va a ejecutar el servidor a un valor entero con la función `str_to_long()` (implementada en `protocol.c`) y se revisa que el valor pertenezca al rango adecuado de los sockets (entre 0 y 65535).
2. Se obtiene la IP de la máquina sobre la que se ejecuta el servidor y se imprime el mensaje "`init server <ip>:<puerto>`".
3. Se inicializan mutexes, condition variables, atributos de threads, se crean los threads y el socket del servidor.
4. Se hace un `bind()` para asignar la dirección y el puerto que se ha recibido al socket del servidor y se ejecuta `listen()` para escuchar a través de este socket.
5. Se ejecuta el bucle infinito del servidor:
    a. El servidor acepta la conexión de un cliente mediante `accept()`.
    b. Recibe los mensajes del cliente con el mandato que tiene que ejecutar a través de la función `read_to_pet()`.

c. Asigna el socket del cliente a la petición y la inserta dentro del buffer circular para que la petición sea llevada a cabo por uno de los threads del pool de threads.

Por otro lado, el flujo de ejecución de uno de los hilos del pool de threads y que se implementa en la función `servicio()` ocurre en un bucle infinito y es el siguiente:

1. Mientras el número de elementos en el buffer circular es igual a 0 esperará en la condition variable `no_vacio`.
2. Si el thread que está esperando en esta condition variable es despertado por un broadcast que se habrá hecho desde el hilo main al insertar una petición en el buffer circular este procederá a copiar la última petición insertada a una variable local para posteriormente trabajar con ella.
3. Atiende la petición mediante la función `tratar_peticion()`, la cual tiene el siguiente comportamiento:
    a. Comparar el campo `command_str` de la petición recibida con cada uno de los comandos que puede recibir el servidor y ejecutar la función correspondiente de las que hay implementadas en `manage_platform.c`. Si las funciones son `LIST_USERS` o `LIST_CONTENT` se realizan otros chequeos previos mediante funciones que devuelven el número de usuarios o entradas de contenido y revisan casos en los que el byte status no acabará siendo 0 y por tanto no habrá que enviar el resto de la información al cliente.
    b. Enviar el byte que representa el status al cliente.
    c. Si la command_str es `LIST_USERS` o `LIST_CONTENT` y `status` es 0 también se enviarán las entradas de contenido o usuarios que se hayan leído al cliente que ha enviado la orden.
    d. Por último se obtiene la fecha y hora desde el cliente, lo cual se había obtenido mediante un servicio web que se nos pidió desarrollar en la Parte 2 de esta práctica.
    e. Se envía la fecha y hora al servidor rpc encargado de llegar el log de todo el sistema y que tuvimos que implementar en la Parte 3 de esta práctica.

2.1.2. Servicios ofrecidos

REGISTER

En el lado servidor, se verifica si el directorio `registered_users` existe y en caso positivo si el archivo del usuario que se quiere registrar existe. Si alguno de los dos no existe, lo crea. Si el archivo ya existe, es porque el usuario ya está registrado (código 1). Finalmente, se devuelve código 0. Se devuelve código 2 si alguna llamada a una función de C devuelve un valor de error, un patrón que se repetirá en las siguientes funciones.

En el lado cliente, cabe destacar que creamos un atributo `_registered` que sirve para saber si un usuario se ha registrado, se trata de una lista de strings con todos los nombres de usuario con los que se ha hecho REGISTER y todavía no se ha hecho UNREGISTER. Por tanto, al

recibir el código 0 del servidor, el cliente hace `append()` con el nombre de usuario input. Esta variable nos será útil en siguientes comandos.

UNREGISTER

En el lado servidor, se verifica si el directorio `registered_users` existe y en caso positivo si el archivo del usuario que se quiere registrar existe. Si alguno de los dos no existe, es porque el usuario no está registrado (código 1). En caso de existir el archivo, lo elimina. Nótese como esto hace automáticamente que, al darse de baja un usuario, también se eliminen todos sus archivos subidos al sistema con PUBLISH.

Cabe destacar que en el cliente se hace una comprobación inicial para que, si el usuario que se quiere dar de baja es el usuario actualmente conectado para ese cliente, se haga una llamada a DISCONNECT y se impriman los mensajes en consola necesarios según el código de retorno de DISCONNECT. Esto lo hacemos porque es imposible que un usuario esté conectado pero no registrado, y entre la posibilidad de no permitir al usuario realizar UNREGISTER en este caso y permitirselo, decidimos permitírselo. La manera en la que sabemos qué usuario está conectado es con un atributo que creamos nosotros `_active_user`, el cual vale None cuando no se tiene un usuario conectado y el nombre del usuario cuando el cliente se conecta a él. Por otro lado, en caso de éxito, le quitamos el usuario dado de baja a la lista de `_registered`.

CONNECT

En el lado servidor, se verifica si el directorio `active_users` existe y en caso positivo si el archivo del usuario que se quiere conectar existe. Si alguno de los dos no existe, lo crea, y escribe con `fprintf()` **`"IP/puerto"`** del hilo servidor del usuario que habían sido pasados como parámetros. Si el archivo ya existe, es porque el usuario ya está conectado (código 2). Algo similar se hace para para `registered_users`, si no existe el archivo se devuelve el código 1. Finalmente, se devuelve código 0.

En el lado cliente, cabe destacar que hacemos una comprobación inicial para interrumpir la función al realizar CONNECT cuando ya está conectado a un usuario. Para evitar conflictos con los hilos servidor, debíamos elegir entre permitirlo y hacer DISCONNECT y luego CONNECT con el nuevo usuario, o no permitirlo, y decidimos esto último. Debido a que hacemos esto, en realidad la función lado servidor nunca es capaz de enviar el código 2, pero seguimos imprimiendo el mensaje correspondiente a su casuística. Además, entre medias del primer paso de creación del socket y el segundo paso de conexión con el servidor, tal y como se nos indica en el enunciado, creamos un segundo hilo en el cliente, un hilo que actúa como si fuese un servidor, el cual nos guardamos en `_server_thread` para que luego pueda ser accedido por DISCONNECT para destruirlo. Lo hicimos gracias a la librería "threading".

Este hilo se trata de un "daemon thread", un hilo que no bloquea al hilo principal, es decir, por el que no hay que esperar (hacer `join()`), con el objetivo de correr en segundo plano. A

este hilo le asociamos la función `socket_server()`, la cual realiza los pasos de un servidor TCP y explicaremos con más detalle en GET_FILE. Para poder conocer el puerto del hilo, nos guardamos en `_server_thread_port` el resultado de `sock.getsockname()[1]` después de hacer `bind()` con la IP del cliente, conocida con `socket.gethostbyname(socket.gethostname())`, y 0, lo cual significa que el sistema operativo asigna aleatoriamente un puerto disponible. Para poder destruirlo, creamos un evento `_stop_event` el cual con `clear()` hacemos igual a False. Se trata de la condición para que el hilo se mantenga en un bucle while esperando mensajes de "GET_FILE" o salga de él y termine.

PUBLISH

En el lado servidor, primero verificamos si el archivo del cliente existe en `registered_users` (código 1). Luego, verificamos si el cliente existe buscando su archivo en `active_users` (código 2). En caso positivo, abrimos el archivo y vamos leyéndolo línea a línea para ver si el archivo que se quiere publicar ya ha sido publicado anteriormente. Para ello, extraemos el primer elemento de la línea, correspondiente al nombre del archivo, con `fscanf()` y lo comparamos con el input con `strcmp()`. Solo si llegamos a EOF sin encontrar el archivo, significa que no ha sido publicado aún (en caso contrario, devolvemos el código 3). En ese caso, escribimos en él al final del archivo con el formato adecuado usando `fprint()`.

Cabe destacar que en el cliente, en esta función, así como en DELETE, LIST_USERS y LIST_CONTENT, se hace una comprobación inicial para ver si `_active_user` es igual a None, es decir, si el usuario no está conectado. Decidimos hacer esto porque `_active_user` debe ser mandado al servidor por los sockets y no hay manera de que el servidor luego sepa si el string enviado es un nombre de usuario válido o si es un string especial para señalizar esto. Debido a que hacemos esto, en realidad la función lado servidor nunca es capaz de enviar el código 2, pero seguimos imprimiendo el mensaje correspondiente a su casuística.

DELETE

En el lado servidor, al igual que en PUBLISH verificamos si el archivo existe en `registered_users` (código 1) y en `active_users` (código 2), y si el archivo ha sido publicado. En caso de no encontrarlo, se devuelve el código 3. En caso contrario, creamos un archivo temporal en el que insertamos todos los registros posteriores al que se pudieran encontrar después del que se quiere borrar, se mueve el puntero del archivo al inicio del registro a borrar, truncamos el archivo en ese punto, y volvemos a escribir los contenidos del archivo temporal.

LIST_USERS

En el lado servidor, LIST_USERS está compuesto de 3 funciones. Primero, en `servidor.c` se llama a `list_users_check()`, el cual verifica si el usuario está registrado (código 1) y conectado (código 2). En ese caso, se llama a `list_users_get_num()` pasando una variable como referencia para poder verificar si el valor de retorno da error y almacenar el número de

usuarios conectados que se tienen que imprimir al mismo tiempo, número el cual se obtiene iterando por el directorio `active_users` y sumando un contador si el archivo encontrado es de tipo `DT_REG`. Si no da error, usamos `malloc()` para crear un array con tantos struct de tipo `ListUserInfo`, creado por nosotros, como número de usuarios conectados haya. Entonces llamamos a `list_users_get_info()` para rellenar esos struct con la info de cada cliente, de nuevo iterando por el directorio y extrayendo el nombre de cada archivo junto con su IP y puerto almacenada en él, separados por un `"/"`. Finalmente, en `servidor.c` mandamos el código correspondiente, y de ser 0 iteramos por este array recién rellenado y por cada uno hacemos 3 `sendMessage()`, uno por cada atributo del struct `ListUserInfo`. Finalmente, realizamos `free()`.

Para poder imprimir la lista de usuarios conectados en el cliente, una vez se obtiene el código 0 y el número de usuarios conectados, se hace un bucle for con este mismo número de iteraciones en cada una de las cuales se reciben los 3 mensajes sockets con el nombre de usuario, IP y puerto. Nos guardamos esta información en un diccionario con estas keys y lo añadimos a un atributo de tipo lista que llamamos `_database_listusers`, ya que nos será necesario acceder a esta información para GET_FILE, y posteriormente imprimimos la información.

LIST_CONTENT

En el lado servidor, realizamos un proceso muy similar al de LIST_USERS, solo que utilizamos las funciones `list_content_get_num()` y `list_content()` para hacer las comprobaciones necesarias, obtener el número de entradas que tiene el archivo del usuario deseado en `registered_users` (el número de `'\n'` que nos encontramos leyendo el archivo), crear un array de la struct `ListContentInfo` con malloc en `servidor.c` siguiendo ese número, rellenarlo con la información extraída de cada líne, enviar el código, y en caso de ser 0, por cada struct, mandamos 2 mensajes con `sendMessage()`: el nombre del archivo publicado y su descripción, finalmente haciendo `free()`.

Para poder imprimir la lista de archivos en el cliente, hacemos algo parecido a LIST_USERS, solo que aquí no guardamos el output en ninguna variable. Simplemente tenemos que hacer un bucle for con número de iteraciones igual al número de archivos mandado por el servidor, leemos el nombre del archivo y su descripción en cada iteración, lo imprimimos, y pasamos a la siguiente iteración.

DISCONNECT

En el lado servidor, se verifica si el directorio `active_users` existe y en caso positivo si el archivo del usuario que se quiere desconectar existe. Si alguno de los dos no existe, es porque el usuario no está conectado (código 2). En caso de existir el archivo, lo elimina. Anterior a ello también se verifica si el usuario está registrado (código 1).

En el lado cliente, cabe destacar que paramos la ejecución del hilo servidor creado en CONNECT, independientemente del código mandado por el servidor o las posibles

excepciones que puedan saltar, así como hacer el atributo `_active_user` igual a None. Para parar el hilo servidor, hacemos `_stop_event.set()`, y para tener al cliente actualizado sobre el estado del hilo, hacemos igual a None los atributos `_server_thread` y `_server_thread_port`.

GET_FILE

En el cliente, primero se verifica, además de los inputs, que el usuario remoto no sea el _active_user, que el archivo local no sea un directorio y el camino hacia él por los directorios existe. Sí que permitimos que el archivo en sí no exista. A continuación, hacemos una llamada a LIST_USERS para actualizar el atributo `_database_listusers` y buscamos el usuario remoto iterando por este array de diccionarios. Cabe destacar que colocamos un parámetro a la función de LIST_USERS para poder no imprimir nada en pantalla al llamarlo desde GET_FILE. Solo si lo encontramos, realizamos los pasos para conectarnos como cliente TCP a la IP y puerto especificados por LIST_USERS, montando así un sistema peer-to-peer. Mandamos un mensaje "GET_FILE" y otro mensaje con el archivo pedido. Al obtener el código 0, se abre el archivo local con la flag "`w+b`", el cual abre el archivo para escribir y leer en binario, el archivo se crea si no existe y se trunca si existe. Finalmente, escribimos en el archivo local el contenido recibido por sockets.

Por su parte, el hilo servidor, al escuchar un mensaje "GET_FILE", abre el archivo remoto deseado con la flag "`rb`" para leer en modo binario, y a menos que este no exista u ocurran otras excepciones, envía su tamaño con `write_number()`, y su contenido con `read()` y `sendall()`.

2.2. Parte 2 - Servicio web

Para implementar el servicio web que se nos pide hemos utilizado las librerías de python **spyne** y **zeep**, las cuales sirven para crear servicios web y clientes basándose en el modelo de APIs SOAP. Hemos usado como base un ejemplo que usa estas librerías y que aparecen en las diapositivas del tema 8 y lo hemos modificado para que el servicio presente un único método llamado `send_date_hour()` en el cual se obtiene un datetime actual con `datetime.datetime.now()`, se convierte a un string con `strftime()` y se devuelve al cliente python. En el cliente python, hemos implementado un método `get_date_hour()` utilizando el wsdl que se genera por parte del servicio web, se crea un cliente con zeep y a continuación se ejecuta el servicio `send_date_hour()`. Esta función se llama al final de todas las operaciones en las que el servidor en C está implicado y el cliente le envía el string que recibe del servicio web al servidor. Cabe destacar que a la hora de ejecutar este servicio web hemos tenido un problema referente a la versión de Python que estábamos usando, por eso hemos tenido que bajar nuestra versión de python a la 3.10.17 para que las librerías spyne y zeep funcionen correctamente, hemos probado esta parte en los servidores guernika y estas librerías funcionan correctamente en la versión 3.11.2 de python.

2.3. Parte 3 - RPC

Para implementar el servicio RPC que se nos pide en esta parte lo primero que hicimos fue definir la interfaz del servicio en un archivo .x, en donde definimos un único método `send_log_rpc()` que recibe como argumento una struct de tipo `send_log_params`, esta struct define 4 campos con los distintos campos que el servidor de nuestro sistema le tiene que enviar al servidor RPC para el log de las operaciones. Los campos que hemos incluido han sido 3 strings `username`, `operation` y `filename` de longitud variable máxima de 256 caracteres y un último string `date_time` con una longitud variable máxima de 22 caracteres debido a que este string siempre estará acotado por un número menor a 22 caracteres. Tras esto utilizamos el comando "`rpcgen -NMa rpc_logger.x`" para generar todo el código plantilla para implementar el resto del servicio.

Por un lado, en el archivo `rpc_logger_client.c` implementamos una función `send_log()` el cual recibe como argumentos punteros `char*` a los 4 campos que hemos descrito en el archivo .x y que rellena la struct `send_log_params` que describimos anteriormente, recibe la IP del servidor RPC a través de la variable de entorno `LOG_RPC_IP` como se había especificado en el enunciado, crea un cliente con tcp, envía la struct `send_log_params` al servidor RPC y por último libera toda la memoria que se ha alocado en el proceso.

Por otro lado, en el archivo `rpc_logger_server.c` rellenamos el código de la función `send_log_rpc_1_svc` de forma adecuada para imprimir en pantalla todos los campos que se reciben del cliente como se especifica en el enunciado. Por último, desde el servidor de nuestro sistema tras recibir la fecha y hora del cliente python se envía usando la función `send_log()` los campos de la struct peticion `client_user_name`, `client_user_name`, `client_user_name` (solo si la operación es `PUBLISH` o `DELETE`) y `date_hour`.

Cabe destacar que hemos tenido un problema a la hora de probar el servicio RPC en los servidores guernika debido a que este no tenía instalado el programa `rpcbind`, que es necesario para que el servicio funcione correctamente, aún así, nuestro proyecto pasa todas las pruebas que especificamos más adelante en el apartado 4 en nuestros ordenadores teniendo el servicio RPC.

3. Compilación y obtención de los ejecutables

Para compilar los archivos de código C de este proyecto hemos escrito un archivo Makefile con el cual al hacer el comando "make all" se puede compilar todo el código del proyecto incluyendo la Parte 3 de RPC. En este archivo encontramos varias reglas:

- **make rpc_logger_xdr**: compila el archivo rpc_logger_xdr.c y deja en el directorio raíz del proyecto el archivo rpc_logger_xdr.o sin linkear debido a que es necesario para compilar tanto el servidor de nuestro sistema como el servidor rpc.
- **make proxy-rpc**: compila los archivos rpc_logger_clnt.c y rpc_logger_client.c dentro del directorio servidor y la output de este comando son los archivos rpc_logger_clnt.o y rpc_logger_clnt.o.

- **make servidor-rpc**: compila dentro del directorio rpc los archivos rpc_logger_svc.c y rpc_logger_server.c y a partir de los archivos rpc_logger_svc.o, rpc_logger_server.o y rpc_logger_xdr.o genera el ejecutable del servidor rpc el cual se llama servidor-rpc.
- **make servidor**: compila dentro del directorio servidor el archivo servidor y a partir de todas las dependencias de este programa se genera el ejecutable servidor.
- **make clean**: elimina los archivos .o, ejecutables y los directorios active_users y registered_users que son generados por la compilación y ejecución del proyecto.

A continuación detallamos todos los comandos que usamos para la ejecución del proyecto:

- **`python3 cliente/client.py -s <IP> -p <PORT>`**: ejecuta nuestro cliente python y en los campos de IP y PORT se tiene que proporcionar la IP y puerto del servidor del sistema. En las pruebas hemos probado a ejecutar varios clientes al mismo tiempo, funcionando todo de manera correcta tal y como describimos en el apartado 4.
- **`./servidor/servidor -p <PORT>`**: ejecuta el servidor del sistema y le asigna el puerto que se le designa en el campo PORT, al ejecutar se indica la IP y puerto del proceso para poder ejecutar el cliente de forma adecuada.
- **`python3 servicio-web/servidor-web.py`**: ejecuta el servicio web que hemos desarrollado en la Parte 2, este se debe ejecutar obligatoriamente en la misma máquina en la que se ejecute un cliente de python.
- **`./rpc/servidor-rpc`**: ejecuta el servidor RPC que hemos desarrollado en la Parte 3, es importante ejecutar previamente **`rpcbind`** con permisos de administrador para iniciar el servicio en la máquina.

4. Batería de pruebas

El objetivo de estas pruebas es ver si cada comando muestra en la pantalla de la terminal del cliente los mensajes intencionados para cada posible escenario.

4.1. Pruebas que aplican a todos los comandos

Comando se hace en un estado inválido

Nos dimos cuenta de que un usuario tiene 3 estados excluyentes:

(1) No registrado y no conectado
(2) Registrado y no conectado
(3) Registrado y conectado

Si siempre se utiliza el mismo nombre de usuario para REGISTER, UNREGISTER, CONNECT y DISCONNECT, y en GET_FILE se solicita el archivo de un nombre de usuario diferente al propio (LIST_CONTENT puede ser para cualquier nombre de usuario), en la siguiente tabla vemos cuáles serían los comandos válidos en cada estado.

ESTADO	COMANDOS VÁLIDOS
1	- REGISTER
2	- UNREGISTER - CONNECT
3	- PUBLISH - DELETE - LIST_USERS - LIST_CONTENT - GET_FILE - DISCONNECT

Por tanto, la primera prueba que hicimos fue intentar realizar, en cada estado, operaciones pertenecientes a los otros estados. A continuación mostramos una tabla con los mensajes que deben aparecer y nos aparecen en la terminal del cliente según el comando y el estado actual, haciendo lo necesario para que no ocurran otro tipo de errores:

	ESTADO ACTUAL		
COMANDO	**1**	**2**	**3**
REGISTER	REGISTER OK	USERNAME IN USE	USERNAME IN USE
UNREGISTER	USER DOES NOT EXIST	UNREGISTER OK	UNREGISTER OK
CONNECT	CONNECT FAIL, USER DOES NOT EXIST	CONNECT OK	USER ALREADY CONNECTED
PUBLISH DELETE LIST_USERS LIST_CONTENT GET_FILE DISCONNECT	[COMANDO] FAIL, USER DOES NOT EXIST	[COMANDO] FAIL, USER NOT CONNECTED	[COMANDO] OK

Parámetros de los comandos tienen más de 256 bytes

Se nos indica esta condición para todos los parámetros de los comandos, como podría ser `<user_name>`. Significa que el parámetro tenga 255 caracteres (el caracter `'\0'` ocupa 1 byte adicional) En estos casos, hicimos pruebas para asegurarnos de que en estos casos se imprima: `[COMANDO] FAIL` ya que entraría dentro de la categoría de "otros errores".

Comando se realiza con el servidor caído (sin compilar)

De forma similar al anterior caso, hicimos pruebas ejecutando todos los archivos excepto el del servidor.c y simplemente escribiendo cada comando, asegurándonos de que siempre se

imprime `[COMANDO] FAIL`, excepto para PUBLISH, DELETE, LIST_USERS y LIST_CONTENT, pues por el tema que ya habíamos explicado sobre no poder mandar _active_user = None por el socket, aquí pueden imprimirse diferentes cosas dependiendo de si se ha podido hacer anteriormente CONNECT: `[COMANDO] FAIL` si se ha realizado cuando el servidor no estaba caído y `LIST_CONTENT FAIL, USER DOES NOT EXIST` en caso contrario.

4.2. Pruebas para grupos de comandos según directorio o archivos usados

Para los comandos REGISTER, UNREGISTER valoramos el directorio `registered_users` y para los comandos CONNECT, DISCONNECT y LIST_USERS valoramos el directorio `active_users`. Consideramos los siguientes casos para cada comando:

<table>
<tr><th>DESCRIPCIÓN</th><th>REGISTER y CONNECT</th><th>UNREGISTER, DISCONNECT y LIST_USERS</th></tr>
<tr><td>El directorio no existe.</td><td rowspan="2"><code>[COMANDO] OK</code></td><td rowspan="2">Para UNREGISTER:
<code>USER DOES NOT EXIST</code>

Para los demás:
<code>[COMANDO] FAIL, USER NOT CONNECTED</code></td></tr>
<tr><td>El directorio existe y está vacío.</td></tr>
<tr><td>El directorio existe y contiene 1 único archivo.</td><td rowspan="2">Si el archivo del usuario pedido no existe:
<code>[COMANDO] OK</code>

Si el archivo del usuario pedido existe:

Para REGISTER:
<code>USERNAME IN USE</code>

Para CONNECT:
<code>USER ALREADY CONNECTED</code></td><td rowspan="2">Si el archivo del usuario pedido no existe:

Para UNREGISTER:
<code>USER DOES NOT EXIST</code>

Para los demás:
<code>[COMANDO] FAIL, USER NOT CONNECTED</code>

Si el archivo del usuario pedido existe:
<code>[COMANDO] OK</code></td></tr>
<tr><td>El directorio existe y contiene 3 archivos.</td></tr>
</table>

Los comandos PUBLISH y LIST_CONTENT utilizan los contenidos individuales de un archivo de `registered_users`, por lo que hicimos pruebas para valorar estos casos y verificar que se devuelve [COMANDO] OK, en PUBLISH se añade el contenido correcto, y en LIST_CONTENT se muestra el contenido correcto.

- El archivo del usuario está vacío.
- El archivo del usuario tiene 1 entrada.
- El archivo del usuario tiene 3 entradas.

4.3. Otras pruebas específicas de comando

DESCRIPCIÓN	RESULTADO
Hacemos UNREGISTER del usuario que además de registrado, está conectado en la terminal actual.	`UNREGISTER OK` Además, el archivo del usuario se elimina de active_users por una llamada a DISCONNECT.
Hacemos PUBLISH 2 veces con el mismo nombre de archivo.	`PUBLISH FAIL, CONTENT ALREADY PUBLISHED`
Hacemos DELETE para un usuario cuyo archivo bajo el directorio `registered_users` existe pero está vacío.	`DELETE FAIL, CONTENT NOT PUBLISHED`
Hacemos DELETE para un usuario cuyo archivo bajo el directorio `registered_users` existe y tiene 3 entradas pero ninguna de ellas es el contenido a eliminar.	`DELETE FAIL, CONTENT NOT PUBLISHED`
Hacemos DELETE para un usuario cuyo archivo bajo el directorio `registered_users` existe, tiene 3 entradas y eliminamos la primera.	`DELETE OK` Además, el contenido deseado se borra del archivo correspondiente en `registered_users`.
Hacemos DELETE para un usuario cuyo archivo bajo el directorio `registered_users` existe, tiene 3 entradas y eliminamos la segunda.	`DELETE OK` Además, el contenido deseado se borra del archivo correspondiente en `registered_users`.
Hacemos DELETE para un usuario cuyo archivo bajo el directorio `registered_users` existe, tiene 3 entradas y eliminamos la tercera.	`DELETE OK` Además, el contenido deseado se borra del archivo correspondiente en `registered_users`.
Hacemos LIST_CONTENT con un nombre de usuario que no existe bajo el directorio `registered_users`.	`LIST_CONTENT FAIL, REMOTE USER DOES NOT EXIST`
Hacemos LIST_CONTENT con un nombre de usuario cuyo archivo existe pero está vacío	`LIST_CONTENT OK` No se imprime nada más.
Hacemos GET_FILE para pedirse un archivo a sí mismo.	`GET_FILE FAIL`
Hacemos GET_FILE para pedir un archivo remoto de un nombre de usuario que no existe bajo el directorio `active_users`.	`GET_FILE FAIL`

Hacemos GET_FILE para pedir un archivo remoto que no ha sido publico por el usuario remoto.	`GET_FILE FAIL, FILE NOT EXIST`
Hacemos GET_FILE para pedir un archivo remoto que no existe.	`GET_FILE FAIL, FILE NOT EXIST`
Hacemos GET_FILE pasando como archivo local un archivo que no existe en un directorio que sí existe.	`GET_FILE OK`
Hacemos GET_FILE pasando como archivo local un directorio.	`GET_FILE FAIL`
Hacemos GET_FILE pasando como archivo local un archivo que no existe en un directorio que no existe.	`GET_FILE FAIL`

5. Conclusiones

En este proyecto hemos aunado el conocimiento de todo el curso en un solo sistema. No tuvimos ningún problema destacable durante su desarrollo gracias a los ejercicios evaluables que habíamos hecho anteriormente. Aunque tuvimos que aprender varias cosas por primera vez, como por ejemplo usar hilos en Python, el proceso en general fue relativamente sencillo. Lo que nos resultó más complejo fue construir la batería de tests y asegurarnos de que contemplábamos todos los casos posibles, y lo solucionamos intentando extraer patrones como el de los 3 estados del usuario.