
# Sistemas distribuidos

**Grado en Ingeniería Informática**

# Práctica Final:

## Dise˜no e implementación de un sistema peer-to-peer

Darío Caballero Polo

(100451112, grupo 81, 100451112@alumnos.uc3m.es)

Raúl Miguel Carrero Martín

(100451286, grupo 81, 100451286@alumnos.uc3m.es)

# Índice de contenidos

# 1. Descripción del código

## 1.1. Estructura de archivos

El proyecto se estructura de la siguiente manera:

```
.
├── client.py
├── requirements.txt
├── server.c
├── server_functions
│   ├── server_functions.c
│   └── server_functions.h
├── web_service
│   └── time_ws.py
└── rpc_server
    ├── rpc_server.c
    ├── rpc_server.x
    └── rpc_server_functions
        ├── rpc_server_functions.c
        └── rpc_server_functions.h
```

## 1.2. Cliente

Se han añadido los siguientes atributos a la clase `client`:

Python

```python
    _connected_user = None
    _listening = False
    _server_thread = None
    _server_socket = None
    _users_list = []
    _ws_url = "http://localhost:5000/get_time"
    _force_disconnect = False
```

**`_connected_user`** se utiliza para identificar al usuario que está conectado en ese momento con el cliente.
**`_listening`** se utiliza para conocer si en este momento se está ejecutando el hilo de escucha a peticiones de otros clientes para transferir archivos peer-to-peer.
**`_server_thread`** es el hilo de escucha a peticiones de otros clientes. Se llama *_server_thread* porque es el hilo que lleva a cabo la función de servidor dentro del cliente.
**`_server_socket`** es el socket por el cual se escuchan las peticiones de otros clientes. Se llama *_server_socket* porque es el socket por el que el cliente se comunica con otros clientes actuando como servidor. El nombre no significa que se conecte con el servidor a través de este socket.
**`_users_list`** es la lista de usuarios conectados que se obtiene del servidor a través del comando LIST_USERS.
**`_ws_url`** es la URL del servicio web de obtención de fecha y hora.
**`_force_disconnect`** es una flag que indica que se ha de forzar la desconexión del usuario y el cierre del hilo de escucha debido a la ejecución de QUIT o a la recepción de la señal SIGINT.

La estructura general de las operaciones es la siguiente:

- Detección de errores en el formato de los valores de entrada
- Creación del socket de conexión con el servidor.
- Llamada al servicio web para obtener la fecha y hora.
- Envío del código de operación.
- Envío de la fecha y hora.
- Envío de otros campos necesarios.
- Recepción del código de error.
- Recepción de otros campos necesarios.
- Cierre de la conexión.
- Devolución del código de error.

Cada operación tiene sus peculiaridades, pero todas siguen esa estructura general. Las más diferentes e interesantes son CONNECT y GET_FILE.

- **CONNECT**:
  Crea tanto el `_server_thread` como el `_server_socket` antes de la creación del socket de `conexión con el servidor.`
- **GET_FILE**:
  El socket que se crea se utiliza para conectarse a otro cliente, no al servidor. La IP y el puerto del otro cliente se obtienen de `_users_list`. El archivo remoto se recibe en *chunks* de 1024 Bytes, hasta que se lee un byte vacío, lo que indica que el otro cliente ha cerrado la conexión y la transmisión ha finalizado.

## 1.3. Servidor

La función *main* del servidor se encarga de escuchar las peticiones de los clientes. Cuando se recibe una, se crea un hilo para procesar dicha petición. Se bloquea la ejecución del hilo principal hasta que se haya copiado el descriptor del socket del cliente en el hilo de procesamiento de la petición. Esa copia es lo primero que se ejecuta en el hilo, tras lo cual se desbloquea el hilo principal y se permite la conexión con más clientes. De esta forma, se permite el procesamiento paralelo de las peticiones. En el caso de las operaciones que acceden a archivos del servidor, ya sea el de *users.txt*, el de *connected_user.txt* o el de *published_files.txt*, el hilo que desee acceder a alguno de ellos debe bloquear el respectivo *mutex* para garantizar lecturas y escrituras limpias y evitar corromper el archivo u obtener un *segmentation fault* en el caso de que un hilo haya borrado un archivo y otro lo intente leer antes de que lo haya vuelto a crear. Cada archivo tiene su respectivo *mutex* para bloquear la ejecución tan solo si se intenta acceder al mismo archivo.

## 1.4. Apuntes sobre las operaciones

A continuación se tratarán aspectos sobre la implementación de las operaciones que no se especficaban en el enunciado, que han sido ligeramente modificados.

### 1.4.1. REGISTER

Se ha implementado toda la funcionalidad especificada, siguiendo cada requisito al detalle y sin agregar ninguna funcionalidad.

### 1.4.2. UNREGISTER

En este caso, tampoco se ha implementado ningún detalle relevante más allá de lo especificado.

### 1.4.3. CONNECT

En el enunciado se indica que, ante un error debido a que el cliente ya estuviese conectado, se debe devolver USER ALREADY CONNECTED. Como no se especifica claramente, asumimos que el se refiere a que el cliente estuviese ya conectado con el mismo usuario. En caso de que el cliente intente conectarse a un usuario distinto, se realiza la desconexión del anterior usuario y se conecta el nuevo, devolviendo CONNECT OK.

### 1.4.4. PUBLISH

Si se recibe PUBLISH FAIL / USER NOT CONNECTED del servidor, hay una inconsistencia entre el registro de usuarios conectados del servidor y el propio cliente. El cliente sí está conectado para sí mismo, pero no de cara al servidor. Para solucionar la inconsistencia entre ambas partes sin modificar el protocolo, se ha decidido que se realice la desconexión del cliente al recibir este error desde el servidor, incluyendo el cierre del hilo de escucha. Este error es difícil de encontrar, pero podría ocurrir ante, por ejemplo, un reinicio en el servidor mientras el usuario se encontraba conectado.

### 1.4.5. DELETE

De nuevo, si se recibe un DELETE FAIL / USER NOT CONNECTED del servidor, se procede a desconectar al usuario para garantizar la consistencia de los datos.

### 1.4.6. LIST_USERS

Al igual que en las anteriores operaciones, si se recibe un LIST_USERS FAIL / USER NOT CONNECTED del servidor, se desconecta al usuario para garantizar la consistencia de los datos.

### 1.4.7. LIST_CONTENT

De igual forma, si se recibe un LIST_CONTENT FAIL / USER NOT CONNECTED del servidor, se realiza la desconexión del usuario en el cliente para garantizar la consistencia de los datos.

### 1.4.8. DISCONNECT

En caso de que se intente desconectar a un usuario diferente al guardado en `_connected_user`, se emitirá un DISCONNECT FAIL / USER NOT CONNECTED. Sin embargo, si el usuario que se intenta desconectar es el mismo que se indica en `_connected_user`, pero aún así el servidor devuelve un DISCONNECT FAIL / USER NOT CONNECTED, eso significa que se ha generado una inconsistencia entre el registro de servidor y el cliente. En el servidor no aparecía como conectado, pero el cliente sí lo estaba para sí mismo. Por ello, en este caso se desconecta al usuario en el lado del cliente y se cierra su hilo de escucha, logrando consistencia entre el servidor y el cliente.

Cada vez que se cierra el servidor y se vuelve a lanzar, su registro de usuarios conectados (*connected_user.txt*) se borra. Esta decisión se debe a que tras reiniciar el servidor, no se puede garantizar que los usuarios que había conectados lo sigan estando. Debido a este motivo, en caso de que el cliente realice DISCONNECT mientras el servidor esté caído, en lugar de devolver DISCONNECT FAIL, se devuelve un DISCONNECT OK y se realizan todos los procesos de desconexión en el lado del cliente, incluyendo el cierre del hilo de escucha. No tiene sentido bloquear la desconexión hasta que el servidor se haya relanzado, pues una vez vuelva a estar disponible, no va a tener registro de que el usuario esté conectado, por lo que devolvería un DISCONNECT FAIL / USER NOT CONNECTED. Por otro lado, a pesar de que se indica que cuando se devuelve un DISCONNECT FAIL se ha de cerrar el hilo de escucha, hemos decidido no hacerlo, ya que causaría una discrepancia entre los datos del servidor y del cliente. Por ello, y para mantener la consistencia, tan solo se fuerza el cierre del hilo a pesar de un fallo en la desconexión si se realiza mediante el comando QUIT o la señal SIGINT. En dichos casos, la flag `_force_disconnect` toma valor True y se cierra el hilo de escucha a pesar de que haya un error en la desconexión. Por lo tanto, hemos modificado ligeramente el funcionamiento del cliente con respecto a estos casos ya que lo encontramos adecuado. No obstante, estas modificaciones en el cliente no afectan al protocolo, por lo que tanto el cliente como el servidor pueden interactuar con el servidor o el cliente, respectivamente, desarrollado por otros grupos.

### 1.4.9. GET_FILE

Al enviarse el contenido de los archivos como bytes, se logra que cualquier tipo de archivo pueda ser enviado. No solo funciona con archivos de texto, sino que otros formatos como imágenes también pueden ser transferidos.

# 2. Compilación y ejecución

Para realizar la compilación del proyecto, se ha de correr el comando `make` en el directorio raíz del proyecto. El Makefile se encarga tanto de realizar la generación de las RPC como de compilarlo todo. Si se desea eliminar los archivos generados, así como todos los registros del servidor: *users.txt*, *connected_users.txt* y *published_files.txt*, se ha de ejecutar `make clean`.

Los detalles sobre cómo ejecutar cada parte del proyecto se encuentran en el archivo *README.md*. En él hay una guía detallada que indica dónde y cómo ejecutar cada comando, pudiendo copiarlos y pegarlos directamente en terminal.

# 3. Batería de pruebas

Se ha realizado una extensa batería de pruebas para comprobar el funcionamiento esperado del sistema ante cualquier situación, incluyendo casos límite y caminos inusuales de ejecución. Las especificaciones y resultados se encuentran en el archivo *bateria_pruebas.pdf*.

# 4. Conclusiones

Esta práctica nos ha ayudado a afianzar nuestros conocimientos sobre sockets y RPC. Trabajar con distintos lenguajes de programación en el cliente y en el servidor nos ha enseñado la gran utilidad de la computación distribuida. Fue realmente gratificante comprobar que la transferencia funcionaba con cualquier tipo de archivo y que todo se realizaba correctamente. La mayor dificultad fue establecer la base del servidor. Una vez se hizo esto, la implementación de cada una de las operaciones fue muy similar.La otra gran complejidad se dio en la toma de decisiones con casos inusuales que no se describen en el enunciado. Todas esas peculiaridades son las que se han relatado en esta memoria, y en nuestra opinión mejoran el funcionamiento del sistema sin alterar el protocolo.

El desarrollo de la batería de pruebas nos permitió solucionar pequeños errores que no afectaban al funcionamiento habitual del sistema, pero causaban fallos al realizar acciones o combinaciones de acciones muy concretas.