Práctica Final

Práctica Parte 1-2:
Diseño e implementación de un sistema peer-to-peer (P2P)

Miguel Jimeno Casas - 100495932 - 100495932@alumnos.uc3m.es

Alberto Menchén Montero - 100495692 - 100495692@alumnos.uc3m.es

Grupo 82 - Ingeniería Informática

ÍNDICE

Introducción

El presente documento constituye la memoria del trabajo final de la asignatura **Sistemas Distribuidos** del curso 2024/2025. En este proyecto, se propone el diseño, implementación y despliegue de una aplicación distribuida basada en una arquitectura **peer-to-peer (P2P)**, cuya funcionalidad se desarrolla a lo largo de tres partes diferenciadas pero integradas.

La **Parte 1** del proyecto consiste en la implementación de una arquitectura cliente-servidor que permita gestionar el registro de usuarios y el intercambio de información entre ellos. En este componente, se deberán desarrollar tanto el servidor como los clientes, permitiendo operaciones como registro, publicación de contenido, eliminación, listado de usuarios, listado de contenido y desconexión. Este módulo se desarrolla íntegramente en lenguaje C, aplicando los principios de concurrencia y comunicación en red mediante sockets.

La **Parte 2** amplía la funcionalidad del sistema mediante la integración de un **servicio web** desarrollado en Python. Este servicio, desplegado localmente en la máquina de cada cliente, se encarga de proporcionar una cadena con la fecha y hora actual en un formato específico ("dd/mm/yyyy hh:mm:ss"). Esta cadena es utilizada como metadato en cada operación realizada por los clientes, y debe ser enviada junto con el código de operación al servidor central.

Por último, en la **Parte 3** se incluye la implementación de un sistema basado en **llamadas a procedimiento remoto (RPC)**, utilizando el modelo **ONC-RPC** en lenguaje C. Esta parte tiene como objetivo registrar las operaciones que realizan los usuarios del sistema en un servidor dedicado de logging. Cada vez que el servidor central recibe una petición, este debe invocar al servidor RPC para enviar la operación realizada, el nombre del usuario, y la fecha/hora obtenida del servicio web. El servidor RPC, por su parte, imprimirá esta información siguiendo un formato estandarizado.

A lo largo de esta memoria, se describirán con detalle los distintos módulos del proyecto, su implementación, el proceso de compilación y ejecución, así como la batería de pruebas desarrolladas para verificar el correcto funcionamiento del sistema. También se incluirán las conclusiones, dificultades encontradas y opiniones personales sobre el desarrollo del trabajo.

Este proyecto no solo permite consolidar los conocimientos teóricos adquiridos sobre sistemas distribuidos, sino que también exige la aplicación práctica de conceptos como la sincronización de procesos, comunicación en red, diseño modular, integración de tecnologías heterogéneas (C y Python) y despliegue en entornos distribuidos.

Primera Parte - Desarrollo de cliente-servidor

La primera parte del proyecto tiene como objetivo el diseño e implementación de un sistema distribuido basado en una arquitectura **cliente-servidor**, que simula el funcionamiento de una red **peer-to-peer (P2P)** para la distribución e intercambio de ficheros entre clientes.

En este modelo, cada usuario actúa como un nodo que puede ofrecer ficheros a otros usuarios y también descargarlos. La arquitectura del sistema está compuesta por dos programas principales:

- **Servidor multihilo (en lenguaje C):** encargado de coordinar a los distintos clientes del sistema, gestionar el registro de usuarios, mantener información sobre los contenidos publicados y dar soporte a operaciones de consulta y control. Debe ser capaz de atender múltiples conexiones simultáneamente utilizando **sockets TCP** y **concurrencia mediante hilos**.

- **Cliente multihilo (en Python):** proporciona una interfaz de línea de comandos desde la cual los usuarios pueden conectarse, registrarse, publicar ficheros, consultar usuarios conectados, descargar contenido y desconectarse del sistema. Cada cliente, además, actúa como servidor de ficheros para permitir la descarga directa entre usuarios.

La comunicación entre cliente y servidor sigue un protocolo bien definido en el enunciado del proyecto. Cada mensaje enviado por el cliente al servidor incluye un **código de operación** (como REGISTER, CONNECT, PUBLISH, etc.), seguido de los datos necesarios para realizar dicha operación. El servidor responde con un **código de estado** que indica el resultado de la operación (éxito, error, usuario inexistente, etc.).

Las operaciones principales que implementa esta parte del sistema son las siguientes:

- **REGISTER**: Permite a un usuario registrarse en el sistema con un nombre único.

- **UNREGISTER**: Elimina un usuario del sistema.

- **CONNECT**: Activa al cliente como nodo P2P, asignándole un puerto de escucha y permitiendo la descarga de ficheros por parte de otros usuarios.

- **DISCONNECT**: Detiene el servicio del cliente y lo desconecta del sistema.

- **PUBLISH**: Publica un fichero en el sistema, informando al servidor de su nombre y descripción. El fichero no se transfiere al servidor; permanece en la máquina del cliente.

- **DELETE**: Elimina un fichero previamente publicado.
- **LIST USERS**: Permite a un usuario conectado obtener la lista de otros usuarios actualmente conectados, junto con su dirección IP y puerto de escucha.
- **LIST CONTENT**: Devuelve los ficheros publicados por un usuario concreto.
- **GET_FILE**: Realiza la descarga directa de un fichero desde otro cliente conectado.

En resumen, esta primera parte establece la infraestructura base del sistema, permitiendo que los usuarios interactúen y compartan ficheros en una red distribuida, bajo un control centralizado parcial gestionado por el servidor. Este componente es esencial, ya que sobre él se construirán las ampliaciones correspondientes a las partes 2 (servicio web) y 3 (RPC) del proyecto.

Procedemos ahora a explicar el código de esta primera parte paso a paso.

Explicación del código del servidor (`server.c`)

1. Estructura general del servidor

El archivo `server.c` implementa el servidor central del sistema P2P, el cual coordina las operaciones entre múltiples clientes. El servidor se ejecuta en C sobre un sistema UNIX/Linux, utilizando **sockets TCP**, y permite la conexión concurrente de múltiples clientes mediante el uso de **hilos (pthreads)**.

Las principales tareas del servidor incluyen:

- Escuchar conexiones en un puerto definido.
- Aceptar múltiples clientes simultáneamente.
- Leer y procesar comandos enviados por los clientes.
- Mantener estructuras internas para gestionar usuarios y archivos publicados.
- Registrar cada operación en un sistema externo RPC de logging (parte 3).

2. Configuración inicial y manejo de sockets

La función `main` realiza los pasos necesarios para inicializar el servidor:

- **Parámetros de entrada**: Se espera que el servidor se ejecute con `./servidor -p <puerto>`. El programa valida que el puerto esté entre 1024 y 65535.

Creación del socket:

```
C/C++
sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

- Se crea un socket TCP usando `AF_INET` (IPv4), tipo `SOCK_STREAM` (conexión), y protocolo TCP.

Configuración del socket:

```
C/C++
setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, ...);
```

Esto permite reutilizar el puerto si el servidor se reinicia rápidamente (evita errores de "puerto en uso").

Bind:

```
C/C++
bind(sd, (struct sockaddr *)&server_addr, sizeof(server_addr));
```

Asocia el socket a una dirección IP y puerto local (servidor).

Listen:

```
C/C++
listen(sd, 10);
```

Pone el socket en modo escucha, permitiendo hasta 10 conexiones pendientes.

- **Accept loop y creación de hilos**:
  El servidor entra en un bucle donde acepta conexiones entrantes con `accept` y lanza un nuevo hilo para cada cliente utilizando `pthread_create`.

3. Estructuras de datos internas

El servidor utiliza arrays fijos para gestionar los usuarios y los ficheros:

- `User users[MAX_USERS]`: almacena nombre, IP, puerto y estado de conexión.

- `FileEntry files[MAX_FILES]`: contiene nombre de archivo, descripción y propietario.

- Se utilizan **mutexes (`pthread_mutex_t`)** para sincronizar el acceso a estas estructuras, garantizando seguridad en entornos multihilo.

4. Funciones `handle_*`

Cada comando del protocolo definido en el enunciado es gestionado por una función específica que sigue este patrón:

- **Validación** del usuario (existencia, conexión, etc.).

- **Bloqueo con mutex** para acceder a la estructura correspondiente.

- **Ejecución de la operación lógica** (añadir/eliminar usuario o archivo).

- **Envío de un byte de respuesta** al cliente indicando el resultado.

Ejemplos:

- `handle_register()`: Registra un nuevo usuario.

- `handle_connect()`: Asocia IP y puerto al usuario y lo marca como conectado.

- `handle_publish()`: Añade un archivo publicado al listado.

- `handle_list_users()`: Devuelve la lista de usuarios conectados.

Estas funciones no gestionan la lectura de datos; simplemente procesan la lógica una vez que los datos ya han sido recibidos.

5. Función `handle_client` – Procesamiento de una conexión

La función `handle_client` es **la más importante** en cuanto a la lógica del servidor. Es llamada en un hilo independiente por cada cliente que se conecta. Sus responsabilidades son:

5.1. Lectura de datos

Utiliza la función `readLine` para leer del socket, siguiendo este orden:

1. **Comando** (ej. REGISTER, CONNECT).

2. **Fecha y hora** (proporcionada por el cliente vía servicio web, parte 2).

3. **Parámetros** adicionales según el comando:
    - 1 parámetro: `username` (ej. REGISTER).
    - 2 parámetros: `username` y `filename` o `port` (ej. CONNECT, DELETE).
    - 3 parámetros: `username`, `filename`, `description` (ej. PUBLISH).

Los datos se almacenan usando `strdup` para facilitar la gestión de memoria.

5.2. Registro de la operación en el servidor RPC

Tras la lectura, la operación se envía al servidor RPC (parte 3), llamando a:

```
C/C++
registrar_log_rpc(username, command, param1, fecha);
```

Esto imprime la operación con su marca de tiempo para propósitos de trazabilidad.

5.3. Ejecución de la operación

Según el comando leído, se llama a la función correspondiente (`handle_register`, `handle_connect`, etc.). La función seleccionada realiza la lógica y responde al cliente con el código de resultado.

5.4. Finalización

Cierra el descriptor del socket y libera memoria dinámica asignada.

6. Gestión de señales y apagado ordenado

El servidor instala un manejador para `SIGINT` (Ctrl+C), que:

- Cambia `server_running` a `false`.
- Llama a `shutdown(sd, SHUT_RDWR)` para salir del `accept`.
- Cierra el socket principal.

Esto permite que el servidor se cierre de manera controlada y sin dejar sockets abiertos.

Explicación del código del cliente (`client.py`)

El archivo `client.py` implementa la lógica del cliente del sistema distribuido. Está desarrollado en Python y sirve como interfaz desde la cual los usuarios pueden interactuar con el servidor central, publicar y gestionar archivos, consultar otros usuarios, e incluso intercambiar archivos directamente entre ellos en un esquema **P2P (peer-to-peer)**.

Este cliente incluye funcionalidades que permiten la conexión con el servidor mediante **sockets TCP**, así como la ejecución de un **hilo de escucha** para recibir peticiones de descarga de otros usuarios. Además, integra una petición HTTP al **servicio web** local para obtener la fecha y hora actual, que se envía con cada operación al servidor como parte del protocolo.

Nota: No se explican en detalle los métodos y estructuras incluidos en la plantilla proporcionada por la asignatura (como `shell()`, `usage()`, `parseArguments()`, etc.), ya que se dan por sabidos y fueron suministrados como esqueleto base del cliente.

1. Estructura general de la clase `client`

La clase `client` está diseñada como una clase estática, conteniendo atributos y métodos de clase. La estructura se divide en tres grandes bloques:

- **Atributos**: información del servidor, puertos, estado de conexión y control del hilo de escucha.
- **Métodos del protocolo**: para enviar comandos al servidor (`REGISTER`, `CONNECT`, etc.).
- **Métodos auxiliares**: envío y lectura de cadenas, obtención de fecha, gestión de archivos, etc.

2. Comunicación cliente-servidor

Cada operación del cliente con el servidor sigue la misma secuencia:

1. Creación de un **socket TCP** con `socket.socket(...)`.
2. Conexión al servidor con `connect(...)`.
3. Envío del comando (por ejemplo, `REGISTER`) como cadena terminada en NULL.
4. Llamada al servicio web local (`get_current_datetime()`) para obtener la fecha y hora.

5. Envío de parámetros según la operación (nombre de usuario, fichero, puerto, etc.).
6. Recepción del **código de respuesta** del servidor (un byte).
7. Interpretación y salida por pantalla.

Para el envío y recepción de cadenas terminadas en NULL se utilizan los métodos:

- `send_string(sock, message)`: codifica y envía una cadena con terminador `\0`.
- `read_string(sock)`: recibe bytes hasta encontrar `\0` y los decodifica.

Estas funciones son reutilizadas en todos los métodos del cliente.

3. Servicio web: obtención de fecha y hora (Parte 2)

El método `get_current_datetime()` realiza una petición HTTP GET al servicio web local desplegado en el puerto 8000:

```
C/C++
response = requests.get("http://localhost:8000/fecha")
```

El resultado se incluye en cada operación del protocolo para que el servidor pueda registrar la operación con su marca temporal. Si hay un error en la conexión al servicio web, se muestra un mensaje en consola, y se cancela la operación.

4. Funcionalidad P2P: transferencia directa entre clientes

Cuando un cliente se conecta con `CONNECT`, se inicializa un **puerto de escucha local** y un **hilo (`_listen_thread`)** que queda a la espera de conexiones entrantes de otros clientes.

4.1 Hilo de escucha (`_listen_thread_function`)

Este hilo:

- Abre un socket en modo escucha (`listen()`).
- Acepta conexiones con `accept()`.
- Recibe una solicitud `GET_FILE` y el nombre del fichero.
- Verifica si el fichero existe.

- Si no existe, envía código de error.
- Si existe, envía:
    1. Código de éxito.
    2. Tamaño del fichero.
    3. Contenido binario del fichero.

Este mecanismo permite que un cliente descargue archivos directamente de otro, sin pasar por el servidor, lo que simula un sistema P2P real.

4.2 Solicitud de fichero (`getfile`)

Para descargar un archivo publicado por otro usuario, el cliente:

1. Verifica que el archivo esté publicado (`LIST_CONTENT`).
2. Obtiene la IP y puerto del usuario remoto (`LIST_USERS`).
3. Se conecta directamente al puerto del otro cliente.
4. Solicita el fichero mediante `GET_FILE` y lo guarda localmente.

Si algo falla en el proceso, se muestran mensajes detallados de error. También se eliminan archivos corruptos parcialmente descargados.

5. Validación de rutas y archivos

En operaciones como `PUBLISH` o `DELETE`, se valida:

- Que se utilicen **rutas absolutas**, como se exige en el enunciado.
- Que el archivo **exista localmente**.

Se utilizan funciones estándar de Python como `os.path.exists()` y `os.path.isabs()` para estas comprobaciones.

6. Manejador de comandos (`shell`)

La interfaz del cliente es interactiva. El método `shell()` actúa como un **intérprete de comandos**, que:

- Lee comandos del usuario por teclado.

- Valida la sintaxis.
- Llama al método correspondiente.
- Muestra el resultado por pantalla.

Permite ejecutar todos los comandos del protocolo: REGISTER, CONNECT, PUBLISH, LIST_USERS, GET_FILE, etc., y salir con `QUIT`.

Segunda Parte - Servicio Web para obtención de fecha y hora

El objetivo de esta segunda parte del proyecto es ampliar el sistema distribuido añadiendo un **servicio web local** que proporcione la **fecha y hora actuales** a los clientes. Esta información debe ser enviada por el cliente al servidor **junto a cada operación**, lo que permite que todas las acciones queden asociadas a una marca temporal exacta.

Los principales requisitos definidos en el enunciado son:

- El servicio debe desarrollarse en **Python**, utilizando la librería `Flask`.
- Debe ofrecer una **única ruta HTTP** (`/fecha`) que devuelva una cadena con el formato:
  `dd/mm/yyyy hh:mm:ss`
- El servicio web se debe desplegar **en la máquina local del cliente**, es decir, cada cliente tiene su propia instancia del servicio.
- Antes de que un cliente envíe una operación al servidor (como REGISTER, CONNECT, etc.), debe solicitar esta fecha y hora al servicio web, y enviarla inmediatamente después del código de operación.

Este componente no tiene lógica de negocio ni persistencia; su único propósito es devolver la fecha actual del sistema en un formato legible.

Implementación del servicio web (`servidor_web.py`)

Para cumplir con este requisito, se ha creado el archivo `servidor_web.py`, el cual implementa un **servidor HTTP sencillo** utilizando Flask. Su funcionalidad consiste en definir una ruta (`/fecha`) que, al recibir una petición, devuelve la fecha y hora actual en el formato requerido.

Código del servidor web:

```python
from flask import Flask
from datetime import datetime

app = Flask(__name__)

@app.route("/fecha")
def obtener_fecha():
    now = datetime.now()
    return now.strftime("%d/%m/%Y %H:%M:%S")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

Detalles importantes:

- **Flask** es un microframework muy utilizado para crear APIs REST en Python de forma sencilla.
- `@app.route("/fecha")`: define la única operación disponible, que se accede mediante una petición GET.
- `datetime.now()` obtiene la fecha y hora actual.
- `strftime(...)` formatea la fecha en el formato exigido.
- El servicio se ejecuta en todas las interfaces locales (`host="0.0.0.0"`) en el **puerto 8000**.

Para que el sistema funcione correctamente, es necesario tener una instancia de este servidor web corriendo **antes de lanzar el cliente**.

Integración del servicio web en el cliente

La integración con el cliente se ha realizado a través del método `get_current_datetime()`, definido dentro de la clase `client` en el archivo `client.py`.

Este método realiza una **petición HTTP GET** a `http://localhost:8000/fecha` antes de cada operación enviada al servidor. La respuesta se procesa y se envía como una cadena al servidor, justo después del comando principal.

Fragmento del método:

```python
@staticmethod
def get_current_datetime():
    try:
        response = requests.get("http://localhost:8000/fecha")
        if response.status_code == 200:
            return response.text.strip()
        else:
            return None
    except Exception as e:
        print("ERROR AL OBTENER FECHA DEL SERVICIO WEB")
        return None
```

Cómo se usa:

Este método se invoca en **todas las operaciones cliente-servidor**:

- `register()`
- `unregister()`
- `connect()`
- `disconnect()`
- `publish()`
- `delete()`
- `listusers()`
- `listcontent()`

- `getfile()`

La fecha obtenida se transmite inmediatamente al servidor para que esta pueda ser registrada o usada en combinación con el sistema RPC (Parte 3).

Tercera Parte - Registro de operaciones mediante RPC

La tercera parte del proyecto tiene como objetivo la implementación de un sistema de **registro de operaciones** mediante **llamadas a procedimientos remotos (RPC)** utilizando el modelo **ONC-RPC** en lenguaje C.

El propósito es permitir que el servidor principal del sistema pueda **delegar la tarea de registrar operaciones** (como REGISTER, PUBLISH, DELETE, etc.) en un servidor remoto especializado. Este registro se realiza a través de una llamada remota que incluye la siguiente información:

- Nombre del usuario que realiza la operación.
- Tipo de operación (REGISTER, CONNECT, DELETE, etc.).
- Parámetro adicional (como el nombre de archivo, si aplica).
- Fecha y hora de la operación (obtenida mediante el servicio web de la Parte 2).

Requisitos técnicos:

- El servidor de la Parte 1 actúa como **cliente RPC**.
- Se debe desarrollar un **servidor ONC-RPC** que reciba las operaciones y las **registre en un fichero `logs.txt`**.
- La interfaz RPC debe definirse en un fichero `.x`.
- La implementación debe garantizar que, por cada operación válida recibida por el servidor principal, se registre su ejecución a través del servicio RPC, siempre que se haya recibido correctamente la fecha desde el servicio web.
- Se deben manejar operaciones con y sin parámetros adicionales.

Este diseño promueve una **separación de responsabilidades** y mejora la trazabilidad de las operaciones del sistema, permitiendo auditar o depurar fácilmente su funcionamiento.

Implementación del sistema RPC

Nuestra solución se compone de 3 archivos principales.

Fichero de interfaz: `claves_rpc.x`

Define la estructura de los datos intercambiados y las funciones remotas disponibles. En este caso:

```
C/C++
struct log_entry {
    string usuario<256>;
    string operacion<256>;
    string param1<>;
    string fecha<256>;
};

program CLAVESRPC_PROG {
    version CLAVESRPC_VERS {
        int log_operation(log_entry) = 1;
    } = 1;
} = 0x31234567;
```

La función `log_operation` acepta una estructura `log_entry` que contiene toda la información necesaria para registrar una operación. Esta estructura soporta operaciones con o sin parámetros adicionales.

Una vez definido este fichero, se utiliza la herramienta `rpcgen` para generar el código fuente necesario (stubs, headers, etc.) que permite compilar tanto el cliente como el servidor RPC.

Servidor RPC: `server_rpc.c`

Este archivo implementa el servicio RPC, que se encarga de recibir llamadas remotas y registrar las operaciones en el fichero `logs.txt`. La función principal es:

```
C/C++
int *log_operation_1_svc(struct log_entry *entry, struct svc_req *req)
```

Su funcionamiento consiste en:

- Abrir el fichero `logs.txt` en modo de **añadir (append)**.
- Formatear la salida según si la operación requiere parámetro adicional (`PUBLISH`, `DELETE`) o no.
- Escribir en el fichero en formato legible:

```C/C++
[17/12/2024 13:45:00] usuario -> OPERACION
[17/12/2024 13:45:02] usuario -> PUBLISH nombre_fichero
```

- Devolver un código de éxito (0) o error (-1).

Este servidor puede ejecutarse en segundo plano y se mantiene a la espera de recibir llamadas del servidor principal.

Cliente RPC: `proxy_rpc.c`

Este archivo es incluido y utilizado por el **servidor principal del sistema (Parte 1)** para realizar llamadas al servidor RPC. Su función principal es:

```C/C++
int registrar_log_rpc(const char *usuario, const char *operacion, const char *param1, const char *fecha);
```

Dentro de esta función:

- Se crea un cliente RPC usando `clnt_create(...)`.
- Se prepara la estructura `log_entry` con los datos recibidos.
- Se realiza la llamada remota `log_operation_1(...)`.
- Se gestiona la memoria y se destruye el cliente tras la llamada.

Esta función se invoca desde `server.c`, dentro de `handle_client`, **después de recibir el comando, los parámetros y la fecha del cliente**, garantizando que toda operación importante queda registrada de forma centralizada.

Integración en el servidor principal

El fichero `proxy_rpc.c` se compila junto con el servidor (`server.c`). En la función `handle_client`, tras recibir los datos de la operación, se llama a:

```
C/C++
registrar_log_rpc(username, command, param1, fecha);
```

Esto garantiza que:

- Toda operación enviada desde el cliente al servidor se acompaña de su correspondiente registro en `logs.txt`.
- Las operaciones sin parámetros (como REGISTER) también se registran correctamente.
- Si el servidor RPC no está disponible, se imprime un mensaje de error en el servidor principal, pero **la operación del sistema no se interrumpe**.

Descripción de la forma de compilar

Todo esta descripción está en el archivo README.txt y como ya llevamos muchas páginas de memoria no la copiamos aquí para no saturar demasiado.

Batería de pruebas

Vamos a aportar 3 pruebas de ejecución diferentes. En cada una de estas pruebas haremos diferentes ejemplos de ejecución para poder ver con claridad cómo se ejecutan los diferentes comandos.

Flujo completo de uso con un único cliente

Esta primera prueba tiene como objetivo verificar el correcto funcionamiento del sistema en un **flujo de uso completo** con un solo cliente, incluyendo todas las operaciones principales: `REGISTER`, `CONNECT`, `PUBLISH`, `LIST_USERS`, `LIST_CONTENT`, `GET_FILE`, `DELETE`, `DISCONNECT` y `UNREGISTER`.

El cliente se conecta al servidor, publica un archivo, consulta usuarios y contenidos, descarga el archivo publicado y lo elimina, para finalmente desconectarse y darse de baja del sistema.

Durante la prueba también se realizan algunas acciones incorrectas de forma intencionada, como consultar contenidos de un usuario inexistente o escribir mal el nombre del usuario, para comprobar que el sistema maneja adecuadamente los errores.

Desarrollo de la prueba

Primero el servidor se arranca en el puerto 8080 y comienza a escuchar conexiones entrantes.

```C/C++
alberto@DESKTOP-4G720TB:/mnt/.../PRACTICA_FINAL$ ./server -p 8080
s> init server 0.0.0.0:8080
```

Durante la ejecución del sistema, se almacenan en el servidor los comandos.

```C/C++
s> [FECHA] 11/05/2025 18:49:42
s> OPERATION REGISTER FROM Alberto
s> [FECHA] 11/05/2025 18:49:47
s> OPERATION CONNECT FROM Alberto
s> [FECHA] 11/05/2025 18:50:05
s> OPERATION PUBLISH FROM Alberto
s> [FECHA] 11/05/2025 18:50:19
s> OPERATION LIST_USERS FROM Alberto
s> [FECHA] 11/05/2025 18:50:25
s> OPERATION LIST_CONTENT FROM Alberto
s> OPERATION LIST_CONTENT FROM Alberto
s> [FECHA] 11/05/2025 18:50:36
s> OPERATION LIST_CONTENT FROM Alberto
s> OPERATION LIST_CONTENT FROM Alberto
s> [FECHA] 11/05/2025 18:51:13
s> OPERATION LIST_CONTENT FROM Alberto
s> OPERATION LIST_CONTENT FROM Alberto
s> [FECHA] 11/05/2025 18:51:13
s> OPERATION LIST_USERS FROM Alberto
s> [FECHA] 11/05/2025 18:53:43
s> OPERATION DELETE FROM Alberto
s> [FECHA] 11/05/2025 18:54:05
s> OPERATION LIST_CONTENT FROM Alberto
s> OPERATION LIST_CONTENT FROM Alberto
s> [FECHA] 11/05/2025 18:54:17
s> OPERATION LIST_CONTENT FROM Alberto
s> OPERATION LIST_CONTENT FROM Alberto
s> [FECHA] 11/05/2025 18:54:48
s> OPERATION DISCONNECT FROM Alberto
s> [FECHA] 11/05/2025 18:55:00
```

```
s> OPERATION UNREGISTER FROM Alberto
```

Para que salga todo esto por la terminal del servidor, ejecutamos en la terminal del cliente:

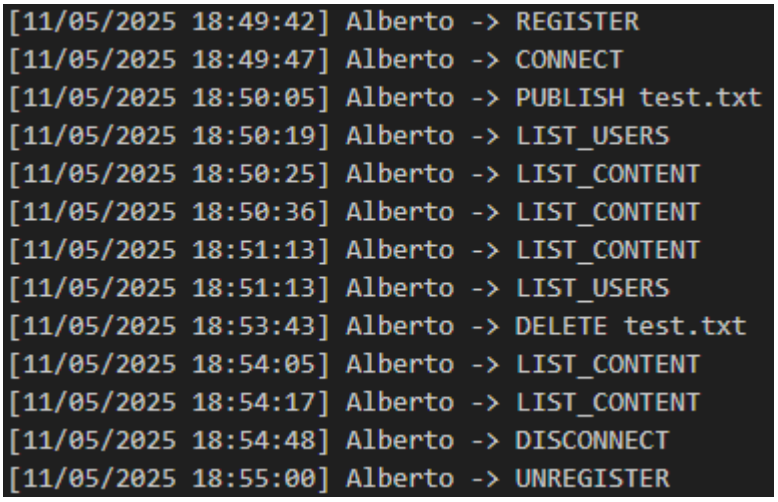
```
C/C++
alberto@DESKTOP-4G720TB:/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/
3-ANO/Github/SISTEMAS_DISTRIBUIDOS/PRACTICA_FINAL$ python3 client.py -s
localhost -p 8080
```

Procedemos a ejecutar diferentes comandos.
En las pruebas realizadas se evalúan múltiples casos representativos del uso real del sistema: operaciones correctas como el registro, conexión, publicación, consulta, descarga, eliminación y baja de un usuario; errores controlados como el uso de nombres de usuario incorrectos (`Albert`, `ALberto`), errores de sintaxis por espacios innecesarios, consultas sin contenido publicado, y validaciones de rutas absolutas necesarias para `PUBLISH` y `DELETE`. También se verifica el correcto funcionamiento del sistema P2P (descarga local mediante `GET_FILE`) y la integración del servicio web de fecha y del sistema RPC para registrar todas las operaciones.

```
C/C++
c> register Alberto
REGISTER OK
c> connect Alberto
CONNECT OK
c> PUBLISH
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/test.txt "Archivo de prueba"
PUBLISH OK
c> list_users
LIST_USERS OK
Alberto 127.0.0.1 48985
c> list_content Albert
LIST_CONTENT FAIL, REMOTE USER DOES NOT EXIST
c> list_content Alberto
LIST_CONTENT OK
test.txt
c> get_file Alberto
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/test.txt copia.txt
```

```
GET_FILE OK
c> Delete
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/test.txt
DELETE OK
c> list_content Alberto  (hay un espacio extra)
Syntax error. Usage: LIST_CONTENT <userName>
c> list_content ALberto
LIST_CONTENT FAIL, REMOTE USER DOES NOT EXIST
c> list_content Alberto
LIST_CONTENT FAIL, USER HAS NO FILES
c> disconnect Alberto
DISCONNECT OK
c> unregister Alberto
UNREGISTER OK
c> quit
+++ FINISHED +++
```

Por último dejamos plasmado en este reporte de las pruebas lo que hay en logs.txt tras realizar la prueba 1.



Flujo concurrente con dos clientes: Miguel y Alberto

Esta segunda prueba tiene como objetivo comprobar el correcto funcionamiento del sistema cuando se utilizan **dos clientes simultáneamente**, simulando un entorno distribuido con interacción entre usuarios. Se validan tanto operaciones individuales como aquellas que afectan a otros clientes, incluyendo la **consulta cruzada de contenidos**, la descarga de archivos desde otro cliente, la publicación concurrente y el control de acceso al eliminar contenidos.

Uno de los objetivos clave es verificar que los archivos solo puedan ser eliminados por su propietario, así como comprobar que el sistema distingue correctamente entre usuarios, incluso si comparten la misma máquina y dirección IP.

Desarrollo de la prueba

Primero se inicia el servidor en el puerto 8080 y comienza a escuchar conexiones entrantes:

```C/C++
alberto@DESKTOP-4G720TB:/mnt/.../PRACTICA_FINAL$ ./server -p 8080
s> init server 0.0.0.0:8080
```

Durante la ejecución del sistema, el servidor va registrando las operaciones en tiempo real gracias al sistema RPC:

```C/C++
s> [FECHA] 11/05/2025 19:09:23
s> OPERATION REGISTER FROM alberto
s> [FECHA] 11/05/2025 19:09:31
s> OPERATION REGISTER FROM Miguel
s> [FECHA] 11/05/2025 19:09:50
s> OPERATION CONNECT FROM alberto
s> [FECHA] 11/05/2025 19:09:58
s> OPERATION CONNECT FROM Miguel
s> [FECHA] 11/05/2025 19:10:24
s> OPERATION LIST_USERS FROM Miguel
s> [FECHA] 11/05/2025 19:11:01
s> OPERATION PUBLISH FROM alberto
s> [FECHA] 11/05/2025 19:11:13
s> OPERATION PUBLISH FROM Miguel
s> [FECHA] 11/05/2025 19:11:23
s> OPERATION LIST_CONTENT FROM Miguel
s> OPERATION LIST_CONTENT FROM Miguel
s> [FECHA] 11/05/2025 19:12:11
s> OPERATION LIST_CONTENT FROM Miguel
s> OPERATION LIST_CONTENT FROM Miguel
s> [FECHA] 11/05/2025 19:12:30
s> OPERATION DELETE FROM Miguel
s> [FECHA] 11/05/2025 19:12:42
s> OPERATION DELETE FROM Miguel
s> [FECHA] 11/05/2025 19:13:25
s> OPERATION LIST_CONTENT FROM Miguel
s> OPERATION LIST_CONTENT FROM Miguel
s> [FECHA] 11/05/2025 19:13:25
s> OPERATION LIST_USERS FROM Miguel
```

Terminal del cliente Alberto

Este cliente realiza las operaciones iniciales para conectarse y publicar un archivo propio

```
C/C++
alberto@DESKTOP-4G720TB:/mnt/.../PRACTICA_FINAL$ python3 client.py -s
localhost -p 8080
c> register alberto
REGISTER OK
c> connect alberto
CONNECT OK
c> PUBLISH
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/test.txt "Archivo de prueba"
PUBLISH OK
```

Terminal del cliente Miguel

Este cliente se registra, se conecta y publica su propio archivo. Luego, interactúa con el contenido de ambos usuarios.

```
C/C++
alberto@DESKTOP-4G720TB:/mnt/.../PRACTICA_FINAL$ python3 client.py -s
localhost -p 8080
c> register Miguel
REGISTER OK
c> connect Miguel
CONNECT OK
c> list_users
LIST_USERS OK
alberto 127.0.0.1 39209
Miguel 127.0.0.1 60813
c> PUBLISH
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/test1.txt "Archivo de prueba"
PUBLISH OK
c> list_content Miguel
LIST_CONTENT OK
test1.txt
c> list_content alberto
LIST_CONTENT OK
test.txt
```

Miguel intenta eliminar el archivo publicado por Alberto, pero el sistema lo rechaza correctamente:

```
C/C++
c> Delete
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/test.txt
DELETE FAIL, CONTENT NOT PUBLISHED
```

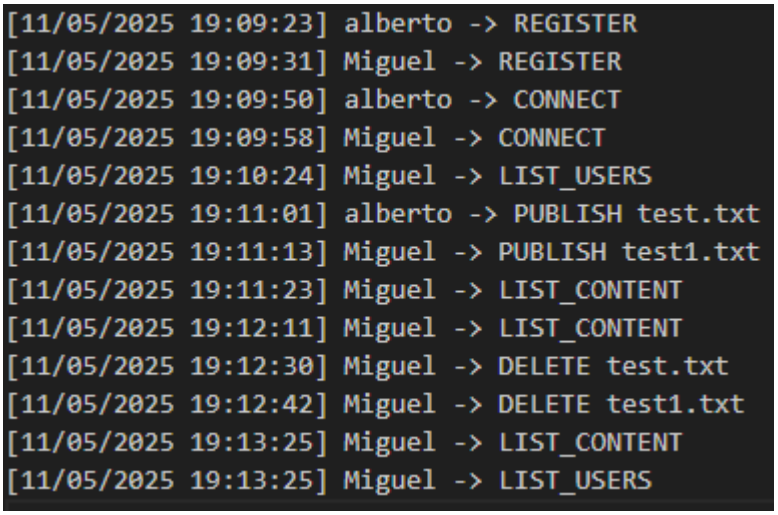
A continuación, elimina **su propio archivo**, lo que sí está permitido:

```
C/C++
c> delete
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/test1.txt
DELETE OK
```

Finalmente, realiza una **descarga P2P** del archivo publicado por Alberto:

```
C/C++
c> get_file alberto
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/test.txt alberto.txt
GET_FILE OK
```

Por último dejamos plasmado en este reporte de las pruebas lo que hay en logs.txt tras realizar la prueba 1.



Casos erróneos y entradas atípicas

Esta prueba tiene como objetivo comprobar el comportamiento del sistema ante **entradas no válidas**, **comandos mal escritos**, **errores comunes del usuario** y situaciones límite,

como nombres de usuario extremadamente largos, símbolos especiales o comandos mal formateados.

El propósito es garantizar que el cliente y servidor sean **robustos** frente a entradas inesperadas y que respondan correctamente, sin producir errores fatales o bloqueos del sistema. También se evalúa el manejo de rutas no válidas en `PUBLISH` y errores al intentar ejecutar operaciones sin conexión válida.

Desarrollo de la prueba

Se arranca el servidor como en pruebas anteriores:

```
C/C++
alberto@DESKTOP-4G720TB:/mnt/.../PRACTICA_FINAL$ ./server -p 8080
s> init server 0.0.0.0:8080
```

Durante la ejecución, se registran las operaciones correctamente gracias a la integración con RPC:

```
C/C++
s> [FECHA] 11/05/2025 19:26:11
s> OPERATION REGISTER FROM
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssss
s> [FECHA] 11/05/2025 19:28:05
s> OPERATION REGISTER FROM |@###~@@#~#€€~¬€¬#@|\ª!"$·ª%!"(%ª·)!"·&
s> [FECHA] 11/05/2025 19:29:45
s> OPERATION REGISTER FROM alberto
s> [FECHA] 11/05/2025 19:29:58
s> OPERATION CONNECT FROM alberto
s> [FECHA] 11/05/2025 19:30:17
s> OPERATION DISCONNECT FROM alberto
s> [FECHA] 11/05/2025 19:30:36
s> OPERATION CONNECT FROM alberto
s> [FECHA] 11/05/2025 19:33:45
s> OPERATION PUBLISH FROM alberto
s> [FECHA] 11/05/2025 19:34:34
s> OPERATION LIST_USERS FROM alberto
s> [FECHA] 11/05/2025 19:35:11
s> OPERATION LIST_CONTENT FROM alberto
s> OPERATION LIST_CONTENT FROM alberto
s> [FECHA] 11/05/2025 19:35:44
s> OPERATION LIST_CONTENT FROM alberto
```

```
s> OPERATION LIST_CONTENT FROM alberto
s> [FECHA] 11/05/2025 19:35:44
s> OPERATION LIST_USERS FROM alberto
```

Terminal del cliente – Ejecución de comandos no válidos

PRUEBAS RARAS:

```C/C++
alberto@DESKTOP-4G720TB:/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/
3-ANO/Github/SISTEMAS_DISTRIBUIDOS/PRACTICA_FINAL$ python3 client.py -s
localhost -p 8080
c> register
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
REGISTER OK
c> register |@###~@@#~#€€~¬€¬#@|\ª!"$·ª%!"(%ª·)!"·&
REGISTER OK
c> register H       KJASNCJKNAS     SA CJ SDA CJHSA=)·!I$
Syntax error. Usage: REGISTER <userName>
c> register Alberto Menchén Montero
Syntax error. Usage: REGISTER <userName>
c> register
Syntax error. Usage: REGISTER <userName>
c> alberto
Error: command ALBERTO not valid.
c> register alberto
REGISTER OK
c> connect alberto
```

```
CONNECT OK
c> disconect alberto
Error: command DISCONECT not valid.
c> disconnect alberto
DISCONNECT OK
c> list_users
LIST_USERS FAIL2
c> connect alberto
CONNECT OK
c> PUBLISH
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/test1.txt
PUBLISH FAIL
c> PUBLISH
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/ "Archivo de prueba"
PUBLISH FAIL
c> PUBLISH
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/noexister.txt "Archivo de prueba"
PUBLISH FAIL, FILE NOT FOUND
Exception: local variable 's' referenced before assignment
c> PUBLISH
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/test1.txt "Archivo de prueba"
PUBLISH OK
c> get_file alberto
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/noexister.txt noexisteeeee.txt
GET_FILE FAIL, FILE NOT PUBLISHED
c> get_file alberto
/mnt/c/Users/Alberto/Desktop/ingenieria_informatica/3-ANO/Github/SISTEMAS_DI
STRIBUIDOS/PRACTICA_FINAL/test1.txt "·$·(/Y/(·T&%$·/·"(%$·"/$//.txt
GET_FILE FAIL
c>
```

Por último dejamos plasmado en este reporte de las pruebas lo que hay en logs.txt tras realizar la prueba 1.

```
[11/05/2025 19:26:11] ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
[11/05/2025 19:28:05] |@###~@@#~#€€~¬€¬#@|\ª!"$·ª%!"(%ª·)!"·& -> REGISTER
[11/05/2025 19:29:45] alberto -> REGISTER
[11/05/2025 19:29:58] alberto -> CONNECT
[11/05/2025 19:30:17] alberto -> DISCONNECT
[11/05/2025 19:30:36] alberto -> CONNECT
[11/05/2025 19:33:45] alberto -> PUBLISH test1.txt
[11/05/2025 19:34:34] alberto -> LIST_USERS
[11/05/2025 19:35:11] alberto -> LIST_CONTENT
[11/05/2025 19:35:44] alberto -> LIST_CONTENT
[11/05/2025 19:35:44] alberto -> LIST_USERS
```

Conclusiones y Problemas encontrados

El desarrollo de este proyecto de sistemas distribuidos ha supuesto una experiencia muy enriquecedora a nivel técnico y formativo. A lo largo de su implementación, hemos tenido la oportunidad de aplicar y consolidar múltiples conceptos clave de la asignatura, como la comunicación mediante sockets TCP, el diseño de sistemas multihilo, la utilización de servicios web y la integración de procedimientos remotos (RPC).

Consideramos que ha sido un proyecto muy interesante, tanto por la complejidad progresiva de sus tres partes como por la necesidad de integrar tecnologías heterogéneas y coordinar componentes en distintos lenguajes de programación (C y Python). Le hemos dedicado una cantidad considerable de tiempo, especialmente en la depuración y la comprensión detallada del funcionamiento de cada módulo, lo que nos ha permitido no solo completar la práctica, sino también **entender en profundidad cada una de sus partes**.

Durante el desarrollo, hemos encontrado algunos desafíos que nos han obligado a investigar, repensar soluciones y mejorar la calidad del código:

- Uno de los aspectos más delicados fue **el paso del parámetro adicional (`param1`) al servidor RPC**, necesario en operaciones como `PUBLISH` y `DELETE`. Al principio, omitíamos esta información o la enviábamos incorrectamente, lo que impedía que se registrase de forma completa en `logs.txt`. Finalmente, conseguimos estructurar correctamente el envío y registro del parámetro cuando era necesario.

- Tuvimos también ciertos **problemas en la operación `DELETE`**, especialmente en la validación del nombre del archivo y la correcta eliminación del mismo de la estructura interna. Fue necesario revisar la lógica que gestionaba los índices del array de archivos, así como asegurarse de que el cliente proporcionaba el nombre en el formato esperado.

- Otro obstáculo significativo fue **la gestión de rutas absolutas** al publicar archivos. El enunciado exige el uso de rutas completas, pero en el listado de contenidos se debía mostrar sólo el nombre del archivo. Esto nos obligó a extraer el nombre base del archivo (`os.path.basename`) antes de enviarlo al servidor, sin alterar la ruta local del fichero, lo cual implicó ajustar tanto el cliente como el servidor para mantener la coherencia del sistema.

A pesar de estas dificultades, creemos que el resultado final ha sido muy satisfactorio. Hemos conseguido implementar todas las funcionalidades exigidas, garantizando la robustez, el correcto funcionamiento en red, la concurrencia de procesos y la trazabilidad de operaciones. Además, este proyecto nos ha proporcionado una visión práctica y realista del diseño e implementación de sistemas distribuidos, algo fundamental para nuestra formación como ingenieros informáticos.