SISTEMAS DISTRIBUIDOS

Práctica Final: Diseño e Implementación de un Sistema *peer-to-peer* (P2P)

ID de grupo de prácticas: 81
Nombre de los participantes:
Liang Ji Zhu
Paolo Michael Webb
Correos electrónicos de los participantes:
100495723@alumnos.uc3m.es
100495955@alumnos.uc3m.es

Madrid, 11 de mayo de 2025

Índice

1. Introducción **2**

2. Parte 1: P2P Básico **2**
2.1. Protocolo de Aplicación 3
2.2. Cliente en Python 3
2.3. Servidor en C 4
2.4. Pruebas automatizadas 6
2.5. Makefile 8

3. Parte 2: Servicio Web **8**
3.1. Diseño 9
3.2. Integración con el cliente 9
3.3. Cambios en el servidor 9
3.4. Pruebas automatizadas 9
3.5. Conclusión 11

4. Parte 3: Sistema P2P con RPC y trazabilidad temporal **11**
4.1. Definición de la Interfaz RPC 11
4.2. Implementación del servidor de logs 11
4.3. Integración en el servidor P2P 12
4.4. Makefile 12
4.5. Pruebas automatizadas 12

5. Batería de pruebas **13**
5.1. Pruebas funcionales (Parte 1) 13
5.2. Pruebas funcionales (Parte 2) 14
5.3. Pruebas concurrentes (Parte 3) 14
5.4. Casos límite y errores controlados 15
5.5. Conclusiones de las pruebas 15

6. Conclusiones Finales **15**

1. Introducción

En el presente proyecto se abordará el diseño e implementación de un sistema *peer-to-peer* (P2P) de distribución de ficheros entre clientes. La práctica se desarrolla de forma incremental en tres partes:

- **Parte 1:** Se sientan las bases del sistema P2P mediante la comunicación con sockets TCP. Los clientes pueden registrarse, darse de baja, conectarse y desconectarse, así como publicar, listar y borrar referencias a ficheros. La transferencia efectiva de archivos se realiza de cliente a cliente, sin que el servidor almacene el contenido.
- **Parte 2:** Se introduce un servicio web REST sencillo, desarrollado en Python, que proporciona la fecha y hora actual en formato `DD/MM/YYYY HH:MM:SS`. Cada operación del cliente utiliza este servicio para obtener un *timestamp* que se envía al servidor junto al código de operación, permitiendo llevar un registro cronológico de la actividad.
- **Parte 3:** Se amplía el servidor con un componente RPC que recibe, para cada petición, el nombre del usuario, la operación y el *timestamp* obtenido del servicio web. Este servidor RPC, basado en `rpcgen`, imprime en pantalla todas las operaciones realizadas, facilitando la monitorización y auditoría de la actividad del sistema.

Con esta estructura, se pretende ofrecer una visión completa del diseño, implementación y validación de un sistema P2P de compartición de ficheros, ilustrando tanto los retos teóricos como las soluciones prácticas adoptadas.

2. Parte 1: P2P Básico

En esta primera aproximación diseñaremos e implementaremos un sistema *peer-to-peer* simple en el que:

- Cada cliente puede **registrarse** o **darse de baja** en el servidor central.
- Un cliente **conectado** abre un socket TCP en un puerto y se anuncia al servidor.
- Un cliente puede **publicar** un fichero (ruta y descripción) o **borrar** una publicación.
- Un cliente puede **listar** los usuarios conectados y el contenido publicado de cualquier usuario.

- Para la obtención de un fichero, `GET_FILE`, el cliente pide al servidor central la IP y puerto remotos, abre conexión directa P2P y descarga el fichero.

2.1. Protocolo de Aplicación

Cada mensaje entre cliente y servidor central va con terminador NUL (`\0`) y el flujo es:

1. El cliente se conecta al servidor mediante TCP.
2. Envía la **operación** en ASCII más un `\0`.
3. Envía parámetros (username, filename, descripción, puerto, etc.) cada uno terminado en `\0`.
4. El servidor responde con un byte de código de retorno:

   0: OK.

   1: Error de usuario (no existe / ya existe / no está conectado).

   2: Error de estado (por ejemplo, usuario no conectado).

   3: Error de aplicación (contenido duplicado, no publicado, etc.).
5. Para operaciones de `LIST_USERS` o `LIST_CONTENT`, tras el 0 viene un entero en ASCII (`"3\0"`) y a continuación esa cantidad de cadenas NUL-terminated.
6. Para `GET_FILE`, primer byte = existencia (0 / 1), y si existe: tamaño en ASCII terminado en NUL, luego el contenido bruto.

2.2. Cliente en Python

El cliente, implementado en `client.py`, se organiza en torno a una única clase `client` que agrupa todas las operaciones del protocolo P2P como métodos estáticos. A continuación se describen las principales responsabilidades y funciones:

- **Gestión de conexión al servidor central:** Cada operación (`REGISTER`, `UNREGISTER`, `CONNECT`, `DISCONNECT`, `PUBLISH`, `DELETE`, `LIST USERS`, `LIST CONTENT`, `GET USER INFO`) se implementa en un método que abre un socket TCP al servidor, envía primero el nombre de la operación (NUL-terminated) y a continuación sus parámetros (usuario, ruta de fichero,

descripción, puerto, etc.), también NUL-terminated, y finalmente recibe un único byte con el código de retorno. Según el código (0 = OK, 1/2/3 = distintos errores) se informa por pantalla y se actualiza el estado interno (_user) si procede.

- **Servicio P2P de transferencia de archivos:** El método `connect(user)` además de anunciar al servidor que el cliente está disponible, arranca internamente un *thread* de escucha en un puerto efímero. Este hilo acepta conexiones entrantes de otros peers que solicitan `GET_FILE`, comprueba si el fichero existe localmente y, en caso afirmativo, envía su tamaño y el contenido en bloques. De este modo, la descarga se realiza directamente cliente-a-cliente.

- **Descarga de archivos remotos:** El método `getfile(user, remote, local)` primero consulta al servidor central la IP y puerto donde el peer remoto está escuchando (`GET USER INFO`), y luego establece una conexión TCP directamente con ese peer para solicitar el fichero (`GET FILE`). Gestiona la recepción del tamaño, la lectura por bloques y la escritura local, así como la limpieza en caso de error parcial.

- **Listados de usuarios y contenidos:** Los métodos `listusers()` y `listcontent(target)` envían la operación correspondiente y, tras recibir un código 0, leen un entero ASCII con el número de entradas y luego cada cadena NUL-terminated (nombre de usuario, IP, puerto o nombre de fichero) para mostrarla por pantalla.

- **Funciones auxiliares:** `read_string(sock)` lee bytes de un socket hasta encontrar el carácter NUL, devolviendo la cadena Unicode correspondiente. Este helper se reutiliza en casi todas las operaciones para extraer cadenas de parámetros o respuestas.

Con esta arquitectura modular, cada operación del protocolo se encapsula en un método claro y auto-contenido, y la lógica de transferencia P2P se integra de forma transparente para el usuario final.

2.3. Servidor en C

El servidor está implementado en C y hace uso de sockets TCP e hilos POSIX para atender de forma concurrente las peticiones de múltiples clientes. A continuación se describen las funciones más importantes:

`main`
: Se ocupa de:

- Parsear el parámetro de línea de comandos `-p <port>`.
- Crear el socket de escucha, configurar `SO_REUSEADDR`, enlazarlo a la dirección `INADDR_ANY` y al puerto indicado, y ponerlo en modo `listen`.
- Instalar el manejador de señal `SIGINT` para permitir un cierre limpio.
- Entrar en un bucle infinito de `accept()`, donde cada conexión aceptada se delega a un nuevo hilo POSIX (`pthread_create`), que ejecuta la función `client_handler`.

`handle_sigint`
: Función que se invoca al recibir `SIGINT`. Se encarga de:

  - Imprimir un mensaje de apagado.
  - Cerrar el socket principal para liberar el puerto.
  - Terminar el proceso.

`read_string`
: Lector de cadenas NUL-terminated desde un descriptor de socket. Lee byte a byte hasta encontrar un carácter nulo (`\0`), ensambla la cadena en un buffer y devuelve `true` si se leyó al menos un carácter.

`client_handler`
: Núcleo de la lógica de protocolo. Para cada nueva conexión:

  1. Lee la operación (opcode) y el nombre de usuario, ambos terminados en NUL.
  2. Bloquea el `mutex` global para sincronizar el acceso a la lista de usuarios.
  3. Busca el índice del usuario en la tabla interna.
  4. Según el opcode recibido, ejecuta una de las ramas:

     `REGISTER` Añade un nuevo usuario si no existe y no se ha alcanzado el límite.

     `UNREGISTER` Elimina el usuario de la tabla (intercambiando con el último).

     `CONNECT` Marca al usuario como conectado, registra su dirección IP real (obtenida con `getpeername`) y el puerto desde el que escucha.

     `DISCONNECT` Marca al usuario como desconectado.

`PUBLISH` Registra una referencia (ruta y descripción) en la lista de ficheros del usuario, comprobando duplicados y límite de publicaciones.

`DELETE` Elimina una referencia de fichero publicado, informando si no existía.

`LIST USERS` Devuelve la lista de usuarios conectados (excluyendo al propio).

`LIST CONTENT` Devuelve el listado de ficheros publicados por un usuario remoto.

5. Envía siempre un byte de código de retorno (0=OK,1=error usuario, 2=error estado, 3=error aplicación) y, cuando procede, datos adicionales (conteo y cadenas para listados, IP y puerto para GET USER INFO).
6. Desbloquea el `mutex`, cierra el socket de cliente y termina el hilo.

En conjunto, estas funciones implementan todo el protocolo de la Parte 1, gestionando el registro de usuarios, su estado de conexión, la publicación y eliminación de referencias a ficheros, los listados y la información necesaria para las transferencias P2P directas.

2.4. Pruebas automatizadas

Para garantizar la correcta implementación de todos los casos de uso y de situaciones límite, hemos desarrollado un script de prueba en Bash (`run_test_parte1.sh`) que automatiza la ejecución del servidor, del cliente y la verificación de resultados. A continuación se describen sus componentes y lógica principal:

- **Preparación del entorno:**
  - Se eliminan ficheros y logs previos y se crean los directorios `input/paolo` e `input/liang` con un fichero de prueba en cada uno.
- **Lanzamiento del servidor P2P:**
  - Se arranca el binario `server` en un puerto configurable.
  - Se espera un breve periodo para asegurar que el servidor queda a la escucha.
- **Secuencias de comandos cliente:**

- Para cada usuario (`liang` y `paolo`) se genera un fichero de comandos:
  1. `UNREGISTER` antes de existir, `REGISTER` correcto y duplicado.
  2. `CONNECT`, `PUBLISH` (primer éxito y segundo duplicado).
  3. `LIST_USERS`, `LIST_CONTENT` (usuario propio, inexistente, remoto inexistente).
  4. `GET_FILE`: descarga de fichero válido y comprobación de falla cuando no existe.
  5. `DELETE` (éxito y fallo al borrar dos veces).
  6. `DISCONNECT` y `QUIT`.
- Cada script se redirige a la entrada estándar de `client.py`, capturando su salida en un log específico.

- **Verificación de resultados:**
  - Se define una función `assert()` que busca patrones literales en los logs del cliente para cada paso (`REGISTER OK`, `PUBLISH FAIL`, `CONTENT ALREADY PUBLISHED`, etc.).
  - Para los casos de `GET_FILE OK` se comprueba además, usando `diff`, que el contenido descargado coincide byte a byte con el fichero original.
  - Al final de cada bloque de pruebas, si todas las aserciones pasan, se imprime un mensaje `+++ TODOS LOS TESTS DE LA PARTE 1 SUPERADOS +++`.

Ejemplo de salida esperada:

```
[PASS] REGISTER liang (OK)
[PASS] CONNECT liang (OK)
[PASS] PUBLISH por primera vez (OK)
[PASS] PUBLISH duplicado (FAIL)
[PASS] LIST_USERS (OK)
[PASS] LIST_CONTENT liang (OK)
[PASS] LIST_CONTENT nobody (FAIL)
[PASS] DELETE exist (OK)
[PASS] DELETE duplicado (FAIL)
[PASS] GET_FILE remoto inexistente (FAIL)
[PASS] DISCONNECT liang (OK)

[PASS] REGISTER paolo (OK)
[PASS] CONNECT paolo (OK)
```

```
...
[PASS] GET_FILE liang -> paolo (OK)
[PASS] Contenido de GET_FILE coincide
[PASS] DELETE paolo exist (OK)
[PASS] DELETE paolo duplicado (FAIL)
[PASS] GET_FILE tras delete (FAIL)
[PASS] DISCONNECT paolo (OK)
+++ TODOS LOS TESTS DE LA PARTE 1 SUPERADOS +++
```

Listing 1: Script de prueba automatizada

Este conjunto de pruebas cubre exhaustivamente el protocolo definido para la Parte 1, incluyendo:

- Registro y baja de usuarios (y detección de duplicados).
- Conexión y desconexión al servidor.
- Publicación y borrado de referencias a ficheros (con gestión de duplicados y no publicados).
- Listado de usuarios y contenidos (incluyendo manejos de errores).
- Descarga de ficheros en modo P2P y comprobación de integridad.

Gracias a este script, podemos validar de forma automática y repetible que la implementación en `server.c` y `client.py` satisface todos los requisitos de la práctica.

2.5. Makefile

Finalmente, el `Makefile` compila cliente y servidor P2P. Con `make` obtendremos el binario `server` y luego ejecutamos `python3 client.py -s 127.0.0.1 -p 12345` para interactuar.

3. Parte 2: Servicio Web

Para añadir a la funcionalidad del P2P, en esta parte desarrollamos un servicio web local que devuelve la fecha y hora actual en el formato `DD/MM/YYYY HH:MM:SS`. Este servicio se utiliza para registrar la hora exacta en la que se realizan las operaciones del cliente.

3.1. Diseño

El servicio está implementado en Python usando Flask, en el archivo `datetime_service.py`. El endpoint principal es:

- `GET /datetime`: devuelve una cadena con la fecha y hora actual.

3.2. Integración con el cliente

Se modificó el cliente para que en cada operación (REGISTER, CONNECT, PUBLISH, etc.)

1. Se realice una petición HTTP al servicio web para obtener el timestamp.
2. Se envíe este timestamp al servidor justo después del código de operación.

Este comportamiento está encapsulado en la función `_send_op()`

3.3. Cambios en el servidor

El servidor en C fue modificado para leer y registrar el timestamp recibido tras el opcode. Se añadió un nuevo argumento que se lee al inicio del handler

3.4. Pruebas automatizadas

Para validar esta parte 2, desarrollamos un script de pruebas (`run_test_-parte2.sh`) que automatiza todo el flujo:

1. Arranca el servicio web de timestamps (`datetime_service.py`) y verifica que devuelve un `DD/MM/YYYY HH:MM:SS` válido.
2. Inicia el servidor P2P modificado para incorporar los timestamps en cada operación.
3. Lanza en paralelo dos clientes (`liang` y `paolo`), cada uno ejecutando una secuencia de comandos con retardos (`sleep`) para probar concurrencia.
4. Captura y muestra los logs del servidor y de cada cliente, así como los archivos descargados en `/tmp`.

```
=== Levantando datetime_service ===
=== Levantando servidor P2P ===
=== Ejecutando clientes liang y paolo en paralelo ===
----- server.log -----
s> init server 0.0.0.0:12345
s> OPERATION REGISTER FROM liang AT 11/05/2025 18:02:34
...
s> OPERATION UNREGISTER FROM paolo AT 11/05/2025 18:02:42
----- liang.log -----
c> REGISTER OK
c> CONNECT OK
c> PUBLISH OK
c> PUBLISH FAIL, CONTENT ALREADY PUBLISHED
c> LIST_USERS OK
paolo 127.0.0.1 51145
c> GET_FILE OK
c> DELETE OK
c> GET_FILE OK
c> DISCONNECT OK
c> UNREGISTER OK
----- paolo.log -----
c> REGISTER OK
c> CONNECT OK
c> PUBLISH OK
c> LIST_USERS OK
liang 127.0.0.1 45913
c> GET_FILE OK
c> DELETE OK
c> GET_FILE OK
c> DISCONNECT OK
c> UNREGISTER OK
Contenido de /tmp/liang_copy.txt: Contenido original de
   Liang
Contenido de /tmp/paolo_copy.txt: Contenido original de
   Paolo
+++ FIN de las pruebas de concurrencia entre liang y
   paolo +++
```

Listing 2: Script de prueba automatizada

Este conjunto de pruebas comprueba no solo la funcionalidad básica, sino también:

- La interacción correcta con el servicio de timestamps.

- El registro de tiempo de cada operación en el servidor.
- El comportamiento esperado ante duplicados y condiciones de error.
- La transferencia P2P concurrente de ficheros entre clientes.

3.5. Conclusión

La parte 2 ha permitido mejorar el sistema añadiendo el tiempo a cada operación. Esta mejora facilita el análisis del comportamiento del sistema y pone las bases para el registro remoto con RPC en la parte 3.

4. Parte 3: Sistema P2P con RPC y trazabilidad temporal

En la tercera parte cambiamos la funcionalidad de registro para extraerla a un servicio RPC, generado automáticamente con `rpcgen`, y preparamos todo el proyecto con un nuevo `Makefile`. Además, incluimos un script de pruebas que lanza clientes en paralelo y verifica el correcto registro y transmisión de ficheros.

4.1. Definición de la Interfaz RPC

La especificación del servicio de registro se definió en un archivo `log.x`, siguiendo el estilo de `rpcgen`. Esta interfaz permite enviar una entrada de log estructurada (usuario, operación, timestamp) al servidor de logs remoto. Se utiliza una única función remota, `SENDLOG`, que solo devuelve la confirmación de recepción. A partir de este archivo se generaron automáticamente los ficheros necesarios mediante `rpcgen`, incluyendo stubs cliente y servidor, interfaz `log.h` y el esqueleto de la función de servicio.

4.2. Implementación del servidor de logs

El servidor de logs (`log_server`) hace uso la operación `SENDLOG`, que recibe cada entrada de log y la imprime por la salida estándar. Esto permite poner la salida en un fichero, `(rpc.log)`, y registrar todas las acciones relevantes del sistema, con los datos del usuario, la operación y la marca de tiempo.

Para asegurar la consistencia, cada línea es registrada de forma atómica, evitando interferencias entre los hilos concurrentes.

4.3. Integración en el servidor P2P

El servidor de la práctica fue modificado para delegar el registro de operaciones al servidor RPC. Para cada operación recibida desde un cliente, el servidor:

- Consulta la fecha y hora actuales a través de un servicio REST externo en Python.
- Construye una entrada de log con los datos del usuario, la operación y el timestamp.
- Envía esta entrada al servidor de logs utilizando la interfaz RPC.

4.4. Makefile

El nuevo `Makefile` realiza:

- Generación de los stubs RPC:
- Compilación de `log_server`, `server` y `datetime_service.py`.
- Targets para limpieza (`make clean`) y para lanzar pruebas.

4.5. Pruebas automatizadas

Para validar el correcto funcionamiento del sistema completo (servidor central, servidor RPC, y cliente P2P), se desarrolló un script (`run_test_-parte3.sh`) que simula una sesión con múltiples clientes actuando en paralelo.

Este script realiza las siguientes acciones:

1. Lanza en segundo plano los tres servicios principales: el servidor de logs RPC, el servicio REST de fecha y hora, y el servidor P2P.
2. Prepara dos secuencias de comandos: una para Alice y otra para Bob. Cada una realiza operaciones como `REGISTER`, `CONNECT`, `PUBLISH`, `LIST`, y `GET_FILE`.
3. Ejecuta los dos clientes de forma concurrente, simulando accesos simultáneos al servidor central y transferencias directas entre ellos.
4. Verifica automáticamente la salida:

- Comprueba que todas las operaciones se registraron en `rpc.log` correctamente y con timestamp.
- Verifica que los logs del servidor central muestran las operaciones en orden y sin conflictos.
- Confirma que las transferencias de fichero P2P han funcionado mediante comparación exacta del contenido.

5. Batería de pruebas

Con el objetivo de garantizar los casos extremos y el correcto funcionamiento de nuestro proyecto para todas las partes, se ha desarrollado una batería de pruebas automatizadas y manuales que cubren tanto el funcionamiento básico como situaciones límite. A continuación describimos esas pruebas.

5.1. Pruebas funcionales (Parte 1)

En la Parte 1 se diseñó el script `run_test_parte1.sh`, que automatiza las siguientes pruebas:

- Registro de usuarios y detección de duplicados.
- Conexión de usuarios y publicación de contenidos.
- Verificación de errores por contenido duplicado.
- Listado de usuarios y de ficheros publicados.
- Descarga de ficheros mediante conexión P2P.
- Borrado de contenidos y verificación del fallo controlado en GET_FILE tras la eliminación.

Todas las pruebas son validadas mediante aserciones (`assert`) que comprueban tanto el contenido como el flujo esperado. El script imprime resultados tipo `[PASS]` y finaliza exitosamente sólo si todas las condiciones son correctas.

5.2. Pruebas funcionales (Parte 2)

Para validar esta Parte 2, desarrollamos un script de pruebas (`run_test_parte2.sh`) que automatiza todo el flujo de funcionamiento:

1. Arranca el servicio web de timestamps (`datetime_service.py`) y comprueba que devuelve una cadena con formato `DD/MM/YYYY HH:MM:SS`.
2. Inicia el servidor P2P modificado, el cual ahora integra los timestamps en cada operación atendida.
3. Lanza en paralelo dos clientes (`liang` y `paolo`), cada uno ejecutando una secuencia de comandos previamente definidos. Se incluyen retardos controlados (`sleep`) entre comandos para probar condiciones de carrera.
4. Captura y muestra los logs generados por el servidor y los clientes, así como los archivos descargados en el directorio temporal `/tmp`.

Esta prueba permite observar la integración del servicio REST en tiempo real, la correcta incorporación de marcas temporales y la validación de que las operaciones concurrentes se manejan correctamente.

5.3. Pruebas concurrentes (Parte 3)

En la Parte 3 se desarrolló `run_test_parte3.sh`, que simula dos clientes en paralelo (Alice y Bob), ejecutando comandos de forma escalonada y concurrente. Se validan los siguientes aspectos:

- Registro simultáneo de múltiples usuarios.
- Publicación de contenidos concurrente.
- Acceso a contenidos del otro usuario durante la sesión activa.
- Transferencia de ficheros entre clientes mediante P2P.
- Desconexión ordenada y acceso controlado.

Los resultados del test muestran en el log del servidor que las operaciones se intercalan temporalmente, lo que demuestra que el servidor es concurrente y no secuencial. Además, el contenido transferido con `GET_FILE` se compara con el original usando `diff`, garantizando integridad.

5.4. Casos límite y errores controlados

Además de los casos correctos, se incluyen pruebas para validar el comportamiento ante:

- Intento de listar usuarios sin estar conectado.
- Publicación de ficheros sin conexión previa.
- Descarga de ficheros no existentes.
- Solicitud de información de un usuario desconectado.

Estas pruebas permiten verificar que todos los errores previstos por el protocolo están correctamente codificados y gestionados tanto por el servidor como por los clientes.

5.5. Conclusiones de las pruebas

Las pruebas realizadas confirman:

- La implementación del protocolo cumple con los requisitos.
- Las conexiones y transferencias P2P funcionan correctamente.
- El servidor maneja concurrencia de múltiples clientes.
- Todos los errores definidos se detectan y reportan de forma adecuada.

Este enfoque exhaustivo asegura la fiabilidad y estabilidad del sistema ante cualquier entrada esperada o errónea.

6. Conclusiones Finales

A lo largo de las tres partes de este proyecto hemos diseñado, implementado y testeado un sistema distribuido, basado en una arquitectura P2P con múltiples componentes: sockets TCP, servicios REST y RPC.

En la primera parte construimos la base funcional: un servidor que registra a los usuarios y gestiona referencias a ficheros, junto a un cliente que permite conectarse, publicar, listar y descargar contenidos de forma directa entre dos clientes. En la segunda parte mejoramos la trazabilidad, añadiendo timestamps a todas las operaciones mediante un servicio web propio. Por

último, en la tercera parte completamos el sistema distribuido del proyecto, delegando el registro de logs a un servidor independiente con RPC. Las pruebas automatizadas con múltiples clientes simultáneos demostraron que el sistema funciona correctamente incluso con concurrencia y casos límites.

En conclusión, hemos construido un sistema P2P funcional, que probamos a fondo. Hemos conseguido implementar los conocimientos que hemos aprendido a lo largo del cuatrimestre a un proyecto final, en el que conseguimos implementar un sistema distribuido completo y funcional.