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. Parte 1: P2P básico

En esta primera parte diseñamos e implementamos 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 `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. Cliente conecta TCP al servidor.
2. Envía la **operación** en ASCII más un `\0`.
3. Envía parámetros (username, filename, descripción, puerto, etc.) cada uno terminado en `\0`.
4. 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 Python

La lógica del cliente está en `client.py`. Usamos la librería `socket` y un hilo secundario para el P2P. A continuación se muestra la cabecera de la clase y la implementación de `register` y `getfile`:

Listing 1: Fragmento de `client.py`: register y getfile

```
class client:
    @staticmethod
    def register(user):
        with socket.socket() as s:
            s.connect((client._server, client._port))
            s.sendall(b"REGISTER\0")
            s.sendall(user.encode() + b"\0")
            code = s.recv(1)[0]
            if code == 0:
                print("REGISTER OK"); client._user = user
            elif code == 1:
                print("USERNAME IN USE")
            else:
                print("REGISTER FAIL")

    @staticmethod
    def getfile(user, remote_fn, local_fn):
        # 1) Pido IP/puerto al servidor central
        with socket.socket() as si:
            si.connect((client._server, client._port))
            si.sendall(b"GET USER INFO\0")
            si.sendall(client._user.encode()+b"\0")
            si.sendall(user.encode()+b"\0")
            if si.recv(1)[0]!=0: return
            ip = client.read_string(si)
            port = int(client.read_string(si))
        # 2) Conecto P2P y descargo
        with socket.socket() as s2:
            s2.connect((ip, port))
            s2.sendall(b"GET FILE\0")
            s2.sendall(remote_fn.encode()+b"\0")
            if s2.recv(1)[0]==0:
                size = int(client.read_string(s2))
                with open(local_fn,"wb") as f:
                    remain=size
                    while remain>0:
                        chunk=s2.recv(min(1024,remain))
```

```
                            f.write(chunk); remain-=len(chunk
                                )
                    print("GET_FILE OK")
                else:
                    print("GET_FILE FAIL, FILE NOT EXIST")
```

Listing 1: Fragmento de `client.py`: register y getfile

2.3. Servidor en C

El servidor central está en `server.c`. Usa hilos POSIX para cada conexión entrante. Extracto de la inicialización y la rama de `PUBLISH`:

Listing 2: Fragmento de `server.c`: arranque y PUBLISH

```
int main(...) {
    signal(SIGINT, handle_sigint);
    server_socket = socket(...);
    setsockopt(server_socket, SOL_SOCKET, SO_REUSEADDR
        ,...);
    bind(server_socket,...); listen(server_socket,10);
    printf("s> init server 0.0.0.0:%d\n", port);
    while(1) {
        int *cs = malloc(sizeof(int));
        *cs = accept(server_socket,...);
        pthread_t tid;
        pthread_create(&tid,NULL,client_handler,cs);
        pthread_detach(tid);
    }
}

void *client_handler(void *a) {
    int sock = *(int*)a; free(a);
    char op[MAX_NAME], user[MAX_NAME];
    read_string(sock,op); read_string(sock,user);
    printf("s> OPERATION %s FROM %s\n",op,user);
    // ... logica de REGISTER, CONNECT, etc. ...
    if(strcmp(op,"PUBLISH")==0) {
        char path[MAX_NAME],desc[MAX_DESC];
        read_string(sock,path); read_string(sock,desc);
        // Comprueba duplicados y anade a users[user_idx
            ].files[]
        send(sock,&res,1,0);
    }
```

```
28      close(sock); return NULL;
29  }
```

Listing 2: Fragmento de `server.c`: arranque y PUBLISH

2.4. Pruebas automáticas

Para validar la Parte 1 creamos `run_test.sh` que:

- Registra dos usuarios (`paolo`, `liang`), prueba duplicados.
- Conecta, publica y detecta duplicados.
- Lista usuarios y contenidos, descarga un fichero con `GET_FILE`.
- Borra contenido y comprueba fallo de `GET_FILE`.

Al ejecutarlo obtenemos:

```
[PASS] REGISTER liang (OK)
[PASS] CONNECT liang (OK)
...
[PASS] Contenido de GET_FILE coincide
[PASS] DELETE existing content (OK)
[PASS] GET_FILE tras DELETE (FAIL)
+++ TODOS LOS TESTS SUPERADOS +++
```

Este conjunto de pruebas cubre todos los casos del protocolo de la Parte 1. Así confirmamos que todas las funcionalidades son correctas, y todos los casos extremos se han tenido en cuenta.

2.5. Makefile

Finalmente, el `Makefile` compila cliente y servidor P2P:

Listing 3: Makefile parte 1

```
1  CC = gcc
2  CFLAGS = -Wall -pthread
3  CLIENT_SRCS = client.py
4  SERVER_SRCS = server.c
5
6  all: server
7
```

```
server: server.c
	$(CC) $(CFLAGS) -o server server.c

clean:
	rm -f server
```

Listing 3: Makefile parte 1

Con `make` obtenemos 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 del servicio

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.

Listing 4: Fragmento de datetime_service.py

```
@app.route('/datetime')
def current_datetime():
ts = datetime.now().strftime('%d/%m/%Y␣%H:%M:%S')
return Response(ts, mimetype='text/plain')
```

Listing 4: Fragmento de datetime_service.py

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()`:

Listing 5: Cliente: envío de operación con timestamp

```
@staticmethod
def _send_op(s, op_str):
s.sendall(op_str.encode('utf-8') + b'\0')
ts = requests.get('http://127.0.0.1:5000/datetime').text
s.sendall(ts.encode('utf-8') + b'\0')
```

Listing 5: Cliente: envío de operación con timestamp

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:

Listing 6: Servidor: recepción del timestamp

```
char op[MAX_NAME];
char ts[MAX_NAME];
char user[MAX_NAME];
read_string(client_sock, op);
read_string(client_sock, ts);
read_string(client_sock, user);
printf("s> OPERATION %s FROM %s AT %s\n", op, user, ts);
```

Listing 6: Servidor: recepción del timestamp

3.4. Pruebas de validación

Creamos un script `run_test.sh` que arranca el servicio web, lanza el servidor y realiza operaciones del cliente. This script checks that the timestamp is correct and that it appears in the log of the service:

Listing 7: Script de prueba automatizada

```
TS=$(curl -s http://127.0.0.1:5000/datetime)

...

grep -qE "^s> OPERATION REGISTER FROM liang AT
    [0-9]{2}/[0-9]{2}/[0-9]{4}" server.log
```

Listing 7: Script de prueba automatizada

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 marcas temporales y concurrencia

En esta tercera parte se amplía el sistema P2P añadiendo dos mejoras principales:

- Incorporación de **marcas temporales** en las operaciones mediante un servicio web externo.
- Registro **concurrente de operaciones** en el servidor, asegurando consistencia.

4.1. Servicio Web de Fecha y Hora

Para registrar todas las operaciones en el servidor junto con su fecha y hora, se implementa un microservicio en Python usando `Flask`, que ofrece la fecha y hora actuales en formato `dd/mm/yyyy HH:MM:SS`:

Listing 8: datetime_service.py

```
@app.route('/datetime')
def current_datetime():
ts = datetime.now().strftime('%d/%m/%Y␣%H:%M:%S')
return Response(ts, mimetype='text/plain')
```

Listing 8: datetime_service.py

El cliente realiza una petición HTTP al servicio antes de enviar cualquier operación al servidor central.

4.2. Cliente con Timestamp

El cliente Python se ha adaptado para:

- Obtener un timestamp del servicio web antes de cada operación.
- Enviar dicho timestamp al servidor tras el nombre de la operación, ambos terminados en NUL (`\0`).

Esto se encapsula en el método auxiliar `_send_op`. Ejemplo:

Listing 9: Envío de operación con timestamp

```
@staticmethod
def _send_op(s, op_str):
s.sendall(op_str.encode('utf-8') + b'\0')
try:
ts = requests.get('http://127.0.0.1:5000/datetime').text
except Exception:
ts = ""
s.sendall(ts.encode('utf-8') + b'\0')
```

Listing 9: Envío de operación con timestamp

4.3. Servidor Concurrente con Registro de Operaciones

El servidor central se mantiene en C y se ha extendido para registrar cada operación en un archivo `server.log`. Cada entrada incluye:

- El tipo de operación.
- El nombre del usuario.
- El timestamp recibido desde el cliente.

Ejemplo de entrada de log:

```
s> OPERATION REGISTER FROM alice AT 10/05/2025 23:08:02
```

Este log se genera usando `fprintf` dentro de secciones protegidas por un `mutex`, para garantizar la concurrencia segura entre hilos.

4.4. Prueba de concurrencia

El script `run_concurrencia_test.sh` lanza dos clientes en paralelo que realizan operaciones completas: `REGISTER`, `CONNECT`, `PUBLISH`, `LIST`, `GET FILE`, etc.

Este test verifica que:

- Todas las operaciones aparecen correctamente en `server.log`.
- El fichero `rpc.log` recoge un resumen de las acciones.
- Las transferencias P2P (extttGET FILE) funcionan correctamente incluso bajo concurrencia.

Fragmento de salida:

```
alice REGISTER 10/05/2025 23:08:02
bob REGISTER 10/05/2025 23:08:03
...
alice_copy.txt => Hola desde Bob
bob_copy.txt   => Hola desde Alice
+++ TEST DE CONCURRENCIA COMPLETADO +++
```

4.5. Makefile y scripts de prueba

El Makefile se ha mantenido consistente, compilando el servidor y permitiendo limpieza rápida del entorno.

Los scripts `run_test.sh` y `run_concurrencia_test.sh` garantizan que el sistema funciona correctamente de forma secuencial y concurrente.

4.6. Conclusión de la Parte 3

Se ha implementado un sistema robusto que permite trazabilidad temporal y ejecución concurrente segura de operaciones. La integración con un servicio REST externo ilustra la interoperabilidad entre componentes distribuidos en diferentes lenguajes (Python y C), y prepara el sistema para futuras extensiones (auditoría, replicación, etc.).

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.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 concurrentes (Parte 3)

En la Parte 3 se desarrolló `run_concurrencia_test.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.

5.3. 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.

5.4. 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.