uc3m | Universidad Carlos III de Madrid

SISTEMAS DISTRIBUIDOS

PRÁCTICA FINAL

Diseño e implementación de un sistema peer-to-peer

Fecha de entrega: 12/05/2024

Lorenzo Largacha Sanz - 100432129

Sonsoles Molina Abad - 100432073

ÍNDICE

1. Introducción

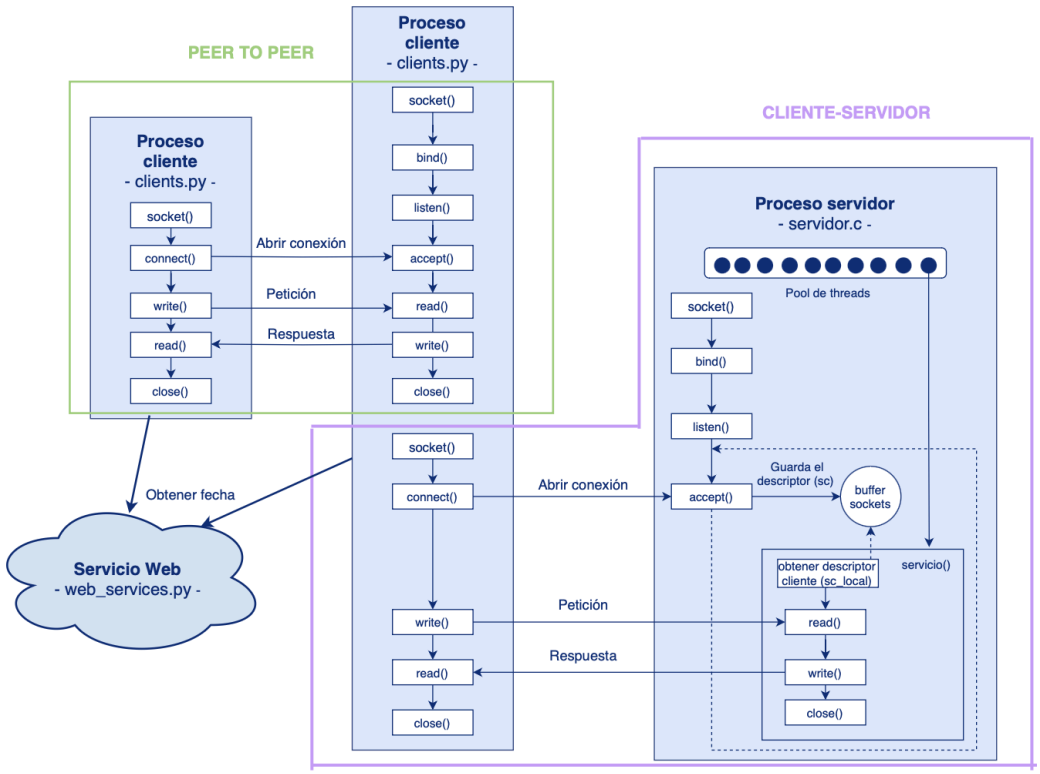
El propósito principal de este proyecto es desarrollar un sistema peer-to-peer comunicado por sockets que permita la distribución eficiente de archivos entre diferentes clientes utilizando una arquitectura de red que no depende de un servidor central para el almacenamiento de archivos. Los archivos se almacenan localmente en las máquinas de los clientes y se transfieren directamente entre ellos bajo demanda.

Adicionalmente, desarrollaremos un servicio web en python que devolverá una cadena de caracteres con la hora y fecha del momento. Ampliaremos también añadiendo un servicio, basado en RPC, que se encargue de imprimir por pantalla las operaciones que realizan los usuarios del sistema.

El sistema estará diseñado para ser robusto, escalable y seguro, proporcionando una plataforma efectiva para compartir recursos de manera distribuida.

2. Diseño de la arquitectura

La arquitectura del sistema queda detallada en el siguiente diagrama:



El servicio se basa en una arquitectura cliente-servidor con comunicación mediante sockets. El cliente realiza una petición al servidor concurrente que será atendida por un proceso ligero (thread) del pool de hilos. Cada thread se comunicará con cada cliente mediante el socket creado por el servidor (buffer de sockets), recibirá la petición del cliente, atenderá la petición, y devolverá la respuesta mediante ese socket. Cada componente de la arquitectura cumple un papel específico en el funcionamiento del sistema, que detallamos a continuación:

2.1. Servidor

Desarrollado en C, coordina las conexiones, mantiene un directorio de archivos disponibles y actúa como intermediario inicial en la conexión entre clientes. Al igual que en los ejercicios anteriores, utilizamos la política de **pool de threads** para la implementación del servidor, creando un conjunto predefinido de hilos (10 en nuestro caso) que estarán disponibles para manejar las solicitudes de los clientes. Los descriptores de los sockets de clientes entrantes se encolan en un buffer de sockets, y se asignan a los hilos disponibles en el pool. Una vez que un hilo ha terminado de manejar una petición, vuelve al pool y está disponible para manejar otra petición, obteniendo un nuevo socket de otro cliente. Este enfoque evita la sobrecarga de recursos del sistema, proporciona un mecanismo de control sobre la concurrencia.

El proceso del programa con sockets TCP funciona de la siguiente forma:

1. El servidor, implementado en el archivo servidor.c, se inicia y espera a que los clientes se conecten. Se crea un socket del lado del servidor que escucha conexiones entrantes. Este socket está en un estado pasivo, esperando conexiones.
2. El servidor entra en un bucle de escucha, donde espera y acepta conexiones entrantes de los clientes. Cuando un cliente intenta conectarse, el servidor acepta la conexión, creando un nuevo socket para esa conexión.
3. Una vez que se establece la conexión, el servidor y el cliente pueden intercambiar datos (serializados en formato de red Big-Endian). El servidor está diseñado para manejar múltiples conexiones simultáneas, es concurrente. Cada socket posee su propio descriptor.
4. En el lado del cliente (clients.py), se crea un socket que se conecta al servidor utilizando la dirección IP y el puerto del servidor. Una vez que se establece la conexión, el cliente puede enviar peticiones al servidor y recibir respuestas.
5. El cliente envía sus peticiones a través de los sockets, estos son recibidos por el servidor y procesados, y posteriormente el servidor envía el resultado obtenido. La comunicación se realiza mediante la lectura y escritura de datos en los sockets.

6. Cuando la comunicación ha terminado, ya sea porque el cliente o el servidor han completado su tarea o por alguna otra razón, se cierra la conexión. Ambos lados del proceso cierran sus sockets asociados con la conexión. En caso de que se termine la ejecución del servidor presionando Ctrl+C, el servidor mandará una señal a todos los threads para que terminen su ejecución y esperará mediante join, además de cerrar todos los sockets que hubiesen podido quedar abiertos en el el buffer de sockets.

El servidor ejecuta las funciones apropiadas basándose en el tipo de operación enviada, además se encarga de mantener la consistencia de los datos y posibles errores de comunicación o de operación. Las funciones utilizadas para enviar y recibir datos mediante sockets han sido implementadas en lines.c.

Al realizar el registro de cliente, se crea un archivo .txt en la carpeta storage para cada usuario en donde se almacenan los contenidos que publican (archivo y descripción).

2.2. Cliente

Desarrollado en python, podrá registrarse, publicar archivos, descargar archivos de otros clientes, y desconectarse del sistema. Esto lo realizará mediante el envío de las siguientes peticiones al servidor:

- Registro de Usuario: REGISTER <userName>
- Baja de Usuario: UNREGISTER <userName>
- Conexión al sistema: CONNECT <userName>
- Publicación de contenidos: PUBLISH <file name> <description>
- Borrado de contenidos: DELETE <file name>
- Listado de usuarios: LIST_USERS
- Listado de contenido de un usuario: LIST_CONTENT <userName>
- Desconexión del sistema: DISCONNECT <userName>
- Transferencia de archivo: GET FILE <user> <remote_file_name> <local_file_name>

Al tratarse de un sistema peer-to-peer, esta última petición se la solicitará al cliente cuyo archivo quiera descargarse. Para ello, los nodos en la red se comunican directamente entre sí sin depender de un servidor centralizado. Aquí, cada instancia del programa actúa como un cliente y un servidor al mismo tiempo.

La clase ClientServer es responsable de escuchar conexiones entrantes y manejar las solicitudes de los clientes. Cada instancia de esta clase se ejecuta en su propio hilo para permitir conexiones múltiples. El cliente que actúa como servidor espera conexiones

entrantes en un bucle y cuando se establece una conexión, se llama al método handle_client() para manejar la solicitud del cliente.

Por otro lado, el método getfile() fuera de la clase ClientServer se encarga de iniciar una solicitud para obtener un archivo de otro cliente en la red P2P. Primero, establece una conexión con el cliente remoto. Luego, envía el comando "GET_FILE" seguido del nombre del archivo solicitado. Espera la respuesta del cliente remoto y maneja los posibles resultados.Si la operación tiene éxito, recibe los datos del archivo y los guarda localmente.

En caso de que se termine la ejecución del servidor presionando escribiendo QUIT por terminal, se cerrará el thread creado para el cliente.

3. Servicio WEB

Este componente proporciona la hora y fecha actual cada vez que se invoca. Este servicio es consultado por el cliente antes de enviar cualquier operación al servidor para anexar la marca de tiempo a las operaciones realizadas. Creamos un nuevo fichero, web_services.py, para abordar este servicio, y desde client.py lo invocamos para conseguir la fecha actual. Utilizamos la librería Zeep para el cliente y Spyne para el servicio.

4. RPC

Este servicio, implementado en C usando el modelo ONC-RPC, recibe información sobre las operaciones de los usuarios desde el servidor principal e imprime esta información, incluyendo la marca de tiempo obtenida del servicio web. Esto ayuda en el monitoreo y registro de actividades en tiempo real.

Para definir la Interfaz de Servicio hemos creado un archivo con extensión “operations.x” donde se especifican las estructuras de datos y los prototipos de las funciones que serán accesibles a través de RPC. Este archivo es utilizado por “rpcgen” para generar código que facilita la implementación de RPC.

No hemos podido completar esta parte del proyecto, pero hemos creado el archivo server_operations.c con la implementación de la función que usaríamos en el servidor RPC.

5. Secuencia de compilación y ejecución

Para realizar la compilación del programa empleamos un archivo Makefile, el cual hemos modificado según los ajustes de sockets y RPC. Escribiendo el comando **./setup.sh** en la terminal, se compilaran los archivos C y se instalarán las dependencias necesarias de python (si aparece que el script no tiene permisos de ejecución, escribir primero **chmod +x setup.sh**).

Para ejecutarlo, abriremos tres terminales, en una llamaremos al web service mediane **python3 web_services.py**, en otra llamaremos al servidor mediante **./server -p <port>** y después, en otra terminal, llamaremos al cliente escribiendo **python3 ./client.py -s <server> -p <port>.**

6. Batería de pruebas

Para probar el correcto funcionamiento del sistema implementado, se analizarán diferentes peticiones mediante la introducción de sus respectivos comandos por la shell y se comparará lo que se muestre por ella con el mensaje esperado.

6.1. Pruebas de Funcionamiento

REGISTER

Test	Comandos de prueba	Resultado esperado	Resultado obtenido
Test #1	`REGISTER user1`	`REGISTER OK`	`REGISTER OK`
Test #2	`REGISTER user1`	`USERNAME IN USE`	`USERNAME IN USE`
Test #3	`REGISTER user1` (probar con el servidor cerrado)	`REGISTER FAIL`	`REGISTER FAIL`

UNREGISTER

Test	Comandos de prueba	Resultado esperado	Resultado obtenido
Test #4	`REGISTER user1` `UNREGISTER user1`	`UNREGISTER OK`	`UNREGISTER OK`
Test #5	`UNREGISTER user2`	`USER DOES NOT EXIST`	`USER DOES NOT EXIST`
Test #6	`UNREGISTER user1` (probar con	`UNREGISTER FAIL`	`UNREGISTER FAIL`

	el servidor cerrado)		

CONNECT

Test	Comandos de prueba	Resultado esperado	Resultado obtenido
Test #7	REGISTER user1 CONNECT user1	CONNECT OK	CONNECT OK
Test #8	CONNECT user2	CONNECT FAIL, USER DOES NOT EXIST	CONNECT FAIL, USER DOES NOT EXIST
Test #9	CONNECT user1 CONNECT user1	USER ALREADY CONNECTED	USER ALREADY CONNECTED
Test #10	CONNECT user1 (probar con el servidor cerrado)	CONNECT FAIL	CONNECT FAIL

PUBLISH

Test	Comandos de prueba	Resultado esperado	Resultado obtenido
Test #11	REGISTER user1 CONNECT user1 PUBLISH file description	PUBLISH OK	PUBLISH OK
Test #12	PUBLISH file description (sin haber hecho antes register)	PUBLISH FAIL, USER DOES NOT EXIST	PUBLISH FAIL, USER DOES NOT EXIST
Test #13	REGISTER user1 PUBLISH file description (sin haber hecho antes connect)	PUBLISH FAIL, USER NOT CONNECTED	PUBLISH FAIL, USER NOT CONNECTED
Test #14	REGISTER user1 CONNECT user1 PUBLISH file description PUBLISH file description	PUBLISH FAIL, CONTENT ALREADY PUBLISHED	PUBLISH FAIL, CONTENT ALREADY PUBLISHED
Test #15	PUBLISH file description (sin estar abierto el servidor)	PUBLISH FAIL	PUBLISH FAIL

DELETE

Test	Comandos de prueba	Resultado esperado	Resultado obtenido
Test #16	DELETE file	DELETE OK	DELETE OK

Test #17	DELETE file (sin haber hecho antes register)	DELETE FAIL, USER DOES NOT EXIST	DELETE FAIL, USER DOES NOT EXIST
Test #18	DELETE file (sin haber hecho antes connect)	DELETE FAIL, USER NOT CONNECTED	DELETE FAIL, USER NOT CONNECTED
Test #19	DELETE file (sin haber hecho antes publish)	DELETE FAIL, CONTENT NOT PUBLISHED	DELETE FAIL, CONTENT NOT PUBLISHED
Test #20	DELETE file (sin estar abierto el servidor)	DELETE FAIL	DELETE FAIL

LIST_USERS

Test	Comandos de prueba	Resultado esperado	Resultado obtenido
Test #21	CONNECT USER1 CONNECT USER2 LIST_USERS	LIST_USERS OK USER1 IP1 PORT1 USER2 IP2 PORT2 ...	LIST_USERS OK USER1 IP1 PORT1 USER2 IP2 PORT2 ...
Test #22	LIST_USERS(sin hacer register antes)	LIST_USERS FAIL, USER DOES NOT EXIST	LIST_USERS FAIL, USER DOES NOT EXIST
Test #23	LIST_USERS(sin hacer connect antes)	LIST_USERS FAIL , USER NOT CONNECTED	LIST_USERS FAIL , USER NOT CONNECTED
Test #24	LIST_USERS(sin estar abierto el servidor)	LIST_USERS FAIL	LIST_USERS FAIL

LIST_CONTENT

Test	Comandos de prueba	Resultado esperado	Resultado obtenido
Test #25	PUBLISH file1 description2 PUBLISH file1 description2 LIST_CONTENT USER1	LIST_CONTENT OK file1 description1 file2 description2	LIST_CONTENT OK file1 description1 file2 description2
Test #26	LIST_CONTENT USER1(sin hacer register antes)	LIST_CONTENT FAIL, USER DOES NOT EXIST	LIST_CONTENT FAIL, USER DOES NOT EXIST
Test #27	LIST_CONTENT USER1(sin hacer connect antes)	LIST_CONTENT FAIL , USER NOT CONNECTED	LIST_CONTENT FAIL , USER NOT CONNECTED
Test #28	LIST_CONTENT USER1(sin hacer register antes de USER1)	LIST_CONTENT FAIL, REMOTE USER DOES NOT EXIST	LIST_CONTENT FAIL, REMOTE USER DOES NOT EXIST
Test #29	LIST_CONTENT USER (sin estar abierto el servidor)	LIST_CONTENT FAIL	LIST_CONTENT FAIL

DISCONNECT

Test	Comandos de prueba	Resultado esperado	Resultado obtenido
Test #30	REGISTER USER1 CONNECT USER1 DISCONNECT USER1	DISCONNECT OK	DISCONNECT OK
Test #31	DISCONNECT USER1 (sin hacer register antes)	DISCONNECT FAIL/ USER DOES NOT EXIST	DISCONNECT FAIL/ USER DOES NOT EXIST
Test #32	DISCONNECT USER1 (sin hacer connect antes)	DISCONNECT FAIL/ USER NOT CONNECTED	DISCONNECT FAIL/ USER NOT CONNECTED
Test #33	DISCONNECT USER1 (sin estar abierto el servidor)	DISCONNECT FAIL	DISCONNECT FAIL

GET_FILE

Abrir dos terminales de cliente.

Test	Comandos de prueba	Resultado esperado	Resultado obtenido
Test #34	REGISTER USER1 CONNECT USER 1 REGISTER USER 2 CONNECT USER2 LIST_USERS GET_FILE USER1 autores.txt aut.txt	GET_FILE OK	GET_FILE OK
Test #35	GET_FILE USER1 d.txt aut.txt (con fichero que no hay en el directorio)	GET_FILE FAIL / FILE NOT EXIST	GET_FILE FAIL / FILE NOT EXIST
Test #36	GET_FILE USER1 autores.txt aut.txt(sin estar abierto el servidor)	GET_FILE FAIL	GET_FILE FAIL