Práctica final

Javier Madrid Hijosa (100472291)
Guillermo Villar Sánchez (100472121)

2024-2025

Universidad
Carlos III de Madrid

DESCRIPCIÓN DEL CÓDIGO

Parte 1

REGISTER Y UNREGISTER

Para la parte se debía de implementar el funcionamiento de un sistema peer to peer entre clientes. Para ello primero se desarrollaron las funciones REGISTER y UNREGISTER. En ellas, como en cada función el cliente crea un socket y se conecta con el servidor, el cual ha generado un socket para escuchar nada más se empezó a ejecutar, y al recibir una solicitud de un cliente, genera un thread para atender a la petición, al cual le pasa tanto el socket que ha recibido la petición, como la ip que se ha conectado al socket.
Dentro del thread, recibe tanto el nombre de la petición, como REGISTER o UNREGISTER, y el nombre del cliente que realiza la operación, y los guarda. Después, el servidor muestra el siguiente mensaje, s > OPERATION < operation > FROM < user_name >, donde operation y user_name son las variables guardadas, y procesa el tipo de operación recibido.

Para el caso de REGISTER y UNREGISTER, existe una carpeta llamada users, donde alberga un directorio para cada usuario creado. De esta forma, si un usuario se registra, se crea en users una carpeta con su nombre, y si un usuario se desregistra, si existe la carpeta con su nombre en users, esta se elimina.

CONNECT Y DISCONNECT

Las siguientes funciones fueron primero CONNECT y luego DESCONNECT. En el caso de CONNECT, como antes, se envía al servidor la operación del usuario, “CONNECT”, y el usuario que la realiza. Después, en el cliente, primero se marca en una variable que hay un usuario conectado y cual es, y luego se busca el primer puerto libre que se encuentre y se manda al servidor. Además, en el cliente se genera un thread que estará escuchando peticiones de otros clientes en el puerto encontrado y su ip.

Por otro lado, en el servidor, hay una carpeta llamada “connect”, la cual albergará los usuarios conectados y su información. De esta manera, cuando el servidor procesa la petición “CONNECT”, recibe el puerto del usuario, y crea un archivo .dat en la carpeta connect con el nombre del usuario. En este archivo, escribe tanto la ip como el puerto del usuario correspondiente. De esta manera, para ejecutar “DISCONNECT”, solo se debe de comprobar si hay un archivo llamado como el usuario en la carpeta “connect”, y lo elimina. Además en disconnect en la parte del cliente, se usa la variable usada en connect para guardar un usuario conectado para comprobar si es necesario detener tanto el thread inicializado en connect como el socket que se había lanzado en ese thread.

PUBLISH Y DELETE

Para el caso de PUBLISH, como siempre, el cliente envía al servidor la operación y el usuario que la ejecuta, y este lo recibe, procesa la petición, en este caso “PUBLISH”, y recibe tanto la ruta del archivo como una descripción, además de comprobar ciertos aspectos como si el usuario está conectado y registrado. Si todo ha ido bien, el servidor genera la ruta recibida en la carpeta del usuario que lo ha solicitado, la cual está dentro de la carpeta “users”. De este modo, un usuario llamado “PACO” registrado y conectado, pública un archivo cuya ruta es simplemente “ejemplo/ejemplo.txt”, aparecería la siguiente ruta: users/PACO/ejemplo/ejemplo.txt, aunque el .txt estaría vacío.

En el caso de DELETE, se realiza el intercambio estándar entre cliente y servidor, recibiendo la operación, el usuario y la ruta del archivo, y una vez se hacen las comprobaciones necesarias, como que el archivo existe, se borra el archivo o directorio especificado.
Por ejemplo en el caso anterior, si se pide borrar el directorio ejemplo, se borra este y su contenido, en cambio si se pide borrar ejemplo/ejemplo.txt, se borra el archivo .txt, pero el directorio ejemplo se deja.

LIST_USERS Y LIST_CONTENT.

En el caso de LIST_USERS y LIST_CONTENT, se aprovecha todo lo hecho hasta el momento. Para LIST_USERS, primero se realiza el intercambio y comprobaciones necesarias entre cliente y servidor, y luego, en el servidor, primero se obtiene el número de usuarios conectados o archivos en la carpeta connect. Si esto ha ido bien, el cliente recibe un código 0, como en el resto de funciones, indicando que todo ha ido bien, y recibe el número de usuarios del servidor, por lo que entre en un bucle para recibir uno a uno el usuario, junto a su ip y puerto. Para ello, el servidor va leyendo el contenido de cada archivo de la carpeta connect, cuyos archivos están nombrados con los nombres de los usuarios, y contienen su ip y puerto, por lo que el servidor va enviando uno a uno el usuario, ip y puerto, y el cliente los va mostrando en pantalla según el formato requerido.

En el caso de LIST_CONTENT, es muy parecido a LIST_USERS, solo que ahora el servidor va leyendo los diferentes archivos en cada carpeta de cada usuario, en el directorio “users”, y se los va enviando al cliente, el cual ha recibido previamente el número de archivos, y los va mostrando en pantalla uno a uno con el formato requerido.

GET_FILE

A diferencia de las otras funciones, la función GET_FILE usa una conexión entre clientes peer-to-peer, por lo que funciona de la siguiente manera. Primero, se ejecuta la función LIST_USERS, para que así, si el usuario del que quieres conseguir un archivo está conectado, se reciba su ip y puerto. Posteriormente, el cliente que ha ejecutado el GET_FILE se conecta al otro cliente a través de su ip y puerto, ya que anteriormente este cliente había creado un socket que estaba escuchando al ejecutar “CONNECT”. Este socket, una vez recibe una conexión, recibe la ruta del archivo solicitado, manda al primer

cliente el tamaño del archivo en bytes, y una vez el primer cliente sabe cuantos bytes debe recibir, el otro cliente manda el contenido del archivo remoto, y el primero lo recibe y lo escribe en la ruta local especificada.

Parte 2

Para la parte del cliente web del servicio web, decidimos crear un servicio SOAP en Python. Para ello hemos hecho uso de Spyne para definir el servicio datetime_service como tal. Existen alternativas como una API REST o Flask, pero siguiendo las instrucciones nos pareció el stack más adecuado.

Hay varias dependencias de Spyne que no están disponibles en Python 3.12, por lo que tuvimos que bajar un par de versiones hasta Python 3.10 para la correcta implementación del cliente web.

Decidimos alojar el servicio en un servidor propio, eso sí, siguiendo las instrucciones de ejecutarlo en la máquina que ejecute el cliente de Python. Se podrían haber combinado ambos en un archivo pero nos pareció la solución más lógica. El servidor funciona hasta que se para manualmente.

Tras el desarrollo de este cliente se introduce en el código de la Parte 1 para actuar de intermediario, de manera que cada vez que el cliente python de la Parte 1 trata de comunicarse con su servidor se envía además del código de operación, el resultado de la llamada al servidor de datetime, que luego utilizaremos más adelante en la impresión de nuestro cliente.

Implementando el RPC como hemos hecho en prácticas anteriores para el envío de información de funciones a través de la red, hemos desarrollado un servidor-rpc que de forma similar al servidor de datetime, actua de “intermediario” recibiendo cada vez que el servidor original lo hace, el nombre de usuario, petición y fecha para imprimirlos.

Donde tuvimos más complicaciones es en la adaptación del Makefile a estos servicios. Evidentemente queríamos mantener la funcionalidad de que el Makefile actuase de “ejecutor” del rpcgen, además de la dificultad añadida que tuvimos al intentar tener el Makefile y el .x del rpcgen en directorios diferentes.

COMPILACIÓN Y EJECUCIÓN

Antes de hacer nada, a lo mejor es necesario instalar algunas librerías con los siguientes comandos, en caso de que no funcione el código tras ejecutar los siguientes comandos:
sudo apt install python3-spyne , **sudo apt install python3-zeep** y **sudo apt install python3-requests** .
Para la compilación de los ejecutables necesarios, primero de todo, se debe de ejecutar **make**. Con ello, se generan dos ejecutables, **servidor** y **servidor-rpc.**

Una vez generados los archivos, hay 4 procesos que ejecutar. Para el servidor, primero hay que ejecutar el comando **export LOG_RPC_IP=ip**, donde ip es la ip que va a usar el servidor rpc. Después, se ejecuta el servidor con ./servidor -p <ip>, donde ip es la ip que va a usar el servidor. Un ejemplo de comando sería **./servidor -p 3000** .
Para ejecuta el servidor-rpc, se ejecuta el comando: **./servidor-rpc** .
Para el servicio web, primero hay que estar en el directorio src, al cual se cambia con **cd src**, y posteriormente se ejecuta con **python3 datetime_service.py** . El servidor web se debe ejecutar junto al cliente, una vez por cliente que esté siendo ejecutado en cada respectivo ordenador.
Para ejecutar el cliente, hay que ejecutar primero un comando para usar la librería dinámica creada con make: **export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.** . Finalmente, para ejecutar el cliente usas python3 ./ cliente . py -s < server > -p < port >, donde server es la server y port son la ip y puerto que está usando el servidor. El puerto no puede ser 8000 ya que está reservado para el servicio web. Un ejemplo de comando es: **python3 client.py -s localhost -p 3000** .

BATERÍA DE PRUEBAS

Para las pruebas de la práctica final decidimos centrarnos en las distintas operaciones que podemos realizar, y los requisitos previos que conllevan cada una de ellas. Se han de ejecutar en un orden concreto, y nos debemos cerciorar de que cualquier error en esa ordenación supone el lanzamiento del error adecuado.

Ejecución del cliente sin alguno de los servidores preparados

Register - Unregister el creado - Unregister uno sin crear y ver q falle

Connect con Register - Connect sin register

Prueba	**Resultado esperado**	**Resultado obtenido**
Ejecutar todo menos servidor	<operation> FAIL	<operation> FAIL
Ejecutar todo menos el servidor web	El cliente no funciona	Salta un error del servidor web, aunque el resto de funciones van perfectamente
Ejecutar todo menos servidor-rpc	El cliente no funciona	Se bloquea el cliente
Borrar las carpetas users y connect y ejecutar el servidor	Las carpetas se crean	Las carpetas se crean
Probar clientes y servidor entre diferentes ordenadores	Todo funciona correctamente	Todo funciona correctamente si clientes y servidor están en la misma red, y, cuando se hace una conexión peer to peer entre dos clientes, uno de los clientes y servidor no se ejecutan en el mismo ordenador
Registrar usuario	Aparece una carpeta con el nombre del usuario en users	Aparece una carpeta con el nombre del usuario en users
Registrar o desregistrar usuario cuyo nombre es demasiado largo (más de 255 carácteres)	REGISTER FAIL	REGISTER FAIL
Registrar usuario ya registrado	USERNAME IN USE	USERNAME IN USE
UNREGISTER usuario sin registrar	USER DOES NOT EXIST	USER DOES NOT EXIST
UNREGISTER usuario registrado	La carpeta del usuario se elimina de la carpeta users	La carpeta del usuario se elimina de la carpeta users
UNREGISTER de un usuario con archivos publicados	Se borra tanto el usuario como sus archivos	Se borra tanto el usuario como sus archivos

Conectar usuario	El usuario se conecta correctamente y puede recibir peticiones	El usuario se conecta correctamente y puede recibir peticiones
Conectar usuario ya conectado	USER ALREADY CONNECTED	USER ALREADY CONNECTED
Conectar usuario inexistente	USER DOES NOT EXIST	USER DOES NOT EXIST
Conectar usuario habiendo ya otro conectado	Se desconecta al anterior usuario, y se conecta el nuevo	Se desconecta al anterior usuario, y se conecta el nuevo
Desconectar usuario conectado	El usuario se desconecta correctamente (se borra el archivo correspondiente de connect), y se detienen el thread lanzado en connect	El usuario se desconecta correctamente (se borra el archivo correspondiente de connect), y se detienen el thread lanzado en connect
Publicar archivo	Se crea la ruta especificada en la carpeta del usuario en users	Se crea la ruta especificada en la carpeta del usuario en users
Publicar archivo con descripción demasiado larga	c > PUBLISH FAIL	c > PUBLISH FAIL
Publicar archivo de un usuario no registrado	PUBLISH FAIL , USER DOES NOT EXIST	PUBLISH FAIL , USER DOES NOT EXIST
Publicar archivo de un usuario no conectado	PUBLISH FAIL , USER NOT CONNECTED	PUBLISH FAIL , USER NOT CONNECTED
Publicar un archivo ya subido	PUBLISH FAIL , CONTENT ALREADY PUBLISHED	PUBLISH FAIL , CONTENT ALREADY PUBLISHED
Eliminar archivo publicado	Se borra la ruta especificada de la carpeta del usuario en users	Se borra la ruta especificada de la carpeta del usuario en users
Eliminar archivo no publicado	DELETE FAIL , CONTENT NOT PUBLISHED	DELETE FAIL , CONTENT NOT PUBLISHED

Eliminar archivo de un usuario no registrado	DELETE FAIL , USER DOES NOT EXIST	DELETE FAIL , USER DOES NOT EXIST
Eliminar archivo de un usuario no conectado	DELETE FAIL , USER NOT CONNECTED	DELETE FAIL , USER NOT CONNECTED
LIST_USERS correcto	Se muestran los usuarios con sus ips y puertos en el formato correspondiente	Se muestran los usuarios con sus ips y puertos en el formato correspondiente
LIST_USERS de usuario no registrado	LIST_USERS FAIL , USER DOES NOT EXIST	LIST_USERS FAIL , USER DOES NOT EXIST
LIST_USERS de usuario no conectado	LIST_USERS FAIL , USER NOT CONNECTED	LIST_USERS FAIL , USER NOT CONNECTED
LIST_CONTENT correcto	Se muestran los archivos y sus rutas correctamente, incluso si hay varios en distintas carpetas	Se muestran los archivos y sus rutas correctamente, incluso si hay varios en distintas carpetas
LIST_CONTENT de usuario no conectado	LIST_CONTENT FAIL , USER NOT CONNECTED"	LIST_CONTENT FAIL , USER NOT CONNECTED"
list_content mostrar archivos de un usuario no registrado	c> LIST_CONTENT FAIL, REMOTE USER DOES NOT EXIST	c> LIST_CONTENT FAIL, REMOTE USER DOES NOT EXIST
get_file correcto	Se establece una conexión entre dos usuarios, se copia el contenido remoto en la ruta local especificada correctamente	Se establece una conexión entre dos usuarios, se copia el contenido remoto en la ruta local especificada correctamente
get_file de un archivo que no existe	c> GET_FILE FAIL , FILE NOT EXIST	c> GET_FILE FAIL , FILE NOT EXIST
Ejecutar Disconnect si hay un usuario ejecutado al terminar el cliente	El usuario se desconecta y el cliente termina de ejecutar	El usuario se desconecta y el cliente termina de ejecutar

CONCLUSIONES

En este proyecto hemos aprendido cómo funcionan las conexiones peer-to-peer para así poder saber que sucede al descargar un torrent. Aunque hemos tenido alguna que otra complicación, hemos sabido solucionarlas con tiempo y esfuerzo. Algunos de los problemas a destacar fueron el manejo de archivos en los directorios, como al registrar o conectar un cliente, o fallo con la creación del socket y thread de un cliente cuando se conectaba.

También hubo otros fallos a destacar como la combinación del makefile que ya teníamos con el makefile del servicio RPC, o que se guardara la ip correcta de un cliente al conectarse, para así poder ejecutar correctamente la función GET_FILE.

Aun así, todos estos problemas se pudieron solucionar parando a entender que estaba pasando y como se estaba comportando el programa, y usando correctamente el código y funciones desarrolladas.

Si que hubo un problema que como tal no se pudo arreglar. Si se prueba el caso extremo de que haya un cliente y un servidor en la misma máquina, entonces si un cliente de una maquina distinta se conecta al otro cliente, se da error, debido a que el servidor registra localhost como la ip del primer cliente, por lo que cuando el segundo cliente se intenta conectar al primero, se conecta al localhost, dando error. Para arreglar esto, se optó por guardar la ip pública del cliente, pero al hacer eso, si se intentaba hacer las pruebas con los clientes y servidor en el mismo ordenador, ya no funcionaban, por lo que se optó por dejarlo como estaba, debido a que esta práctica seguramente se evalúe usando solo una máquina, y funciona correctamente usando clientes y servidor en diferentes ordenadores, siempre y cuando estén en la misma red, y un cliente no se ejecute en el mismo ordenador que el servidor.

También tuvimos un problema con la estructura del proyecto, ya que intentamos dejar el código en una carpeta src, y el makefile fuera, pero al generar los archivos rpc, esto generó problemas, por los que tuvimos que sacarlos. También sacamos el código del cliente, yq eu al subir u obtener un archivo, la ruta absoluta contenía la carpeta src, lo cual daba problemas, por lo que optamos por sacar client.py de src.

Al final, este trabajo ha sido bastante interesante, y nos ha ayudado a entender tanto el funcionamiento entre servidor-clientes, como a desarrollarlos, lo cual tendrá grandes beneficios para nuestra vida profesional.