Universidad Carlos III

Sistemas Distribuidos

Curso 2024-25

TRABAJO FINAL

11/05/2025

Inés Fuai Guillén Peña 100495752@alumnos.uc3m.es | g82

Claudia Fernández Menéndez 100495919@alumnos.uc3m.es | g82

Universidad
Carlos III de Madrid

Índice

Introducción

El propósito de este ejercicio es desarrollar e implementar un servicio distribuido utilizando **Sockets TCP**. En este proyecto, crearemos un servidor que escuche peticiones de un cliente y responda adecuadamente a las mismas mediante el uso de la comunicación a través de Sockets TCP. El servidor manejará las solicitudes de registro, conexión, desconexión, publicación de archivos, eliminación de archivos y listado de usuarios conectados, proporcionando así un sistema básico de manejo de usuarios y archivos en una red distribuida.

Además, implementaremos Servicios Web y RPC para que los clientes puedan mandar junto a sus peticiones la fecha y hora de la misma, y para poder visualizar un registro en tiempo real de las operaciones realizadas por cada usuario mostrando la fecha y hora a la que fueron realizadas.

Parte 1

Sockets

Diseño

Servidor

El archivo `server.c` implementa la lógica principal del servidor.Este componente está diseñado para gestionar múltiples clientes de forma concurrente mediante la creación de hilos, de manera que cada cliente que se conecta es atendido en un hilo independiente.

Una de las principales tareas del servidor es el manejo de sockets. Para ello, crea un socket y lo vincula a una dirección IP y puerto específicos. Luego, entra en un bucle de espera donde escucha las conexiones entrantes. Cuando un cliente solicita una conexión, esta es aceptada y se lanza un nuevo hilo para gestionarla.

La comunicación con los clientes se basa en la recepción de comandos enviados a través del socket. El servidor interpreta cada mensaje recibido y ejecuta la acción correspondiente. Entre los comandos disponibles se encuentran `REGISTER`, `CONNECT`, `PUBLISH`, `DELETE`, `LIST_USERS`, `LIST_CONTENT`, `DOWNLOAD`, `DISCONNECT` y `QUIT`.

Para permitir la atención simultánea de múltiples usuarios, el servidor utiliza hilos. Cada vez que un nuevo cliente se conecta, el servidor crea un nuevo hilo que se encarga de gestionar toda la interacción con ese cliente. Esta estrategia de diseño permite escalabilidad y evita que un cliente pueda bloquear el funcionamiento del servidor.

Con el objetivo de proteger los recursos compartidos, especialmente la lista de usuarios y la lista de publicaciones, empleamos un mecanismo de exclusión mutua (`mutex`). Esto garantiza que el acceso concurrente a estos recursos se realice de forma segura, evitando condiciones de carrera y posibles corrupciones de datos cuando varios hilos intentan modificar estas estructuras al mismo tiempo.

Además, el servidor se apoya en funciones auxiliares definidas en el archivo `lines.c`. Este fichero contiene la función `readLine`, que abstrae la lógica de envío y recepción de datos a través del socket. Gracias a estas funciones, se facilita una comunicación más robusta y estructurada entre servidor y cliente.

Cliente

El `client.py` representará al cliente. Está implementado en Python y permite la interacción con el servidor. Este programa representa el punto de entrada para el usuario, permitiéndole conectarse al servidor, enviar comandos y recibir respuestas de forma clara y estructurada.

Una de las funciones fundamentales del cliente es establecer la conexión con el servidor. Para ello, el usuario proporciona la dirección IP y el puerto donde se encuentra activo el servidor. Una vez realizada la conexión, el cliente permanece en espera de introducir comandos que serán procesados por el servidor.

El envío de comandos constituye el núcleo de la interacción del cliente con el sistema. Comandos como `REGISTER`, `CONNECT`, `PUBLISH`, `LIST_USERS`, entre otros, son introducidos por el usuario y enviados al servidor a través del socket abierto. Cada comando puede requerir parámetros adicionales como nombre de usuario, dirección IP, puerto o nombre de fichero.

Finalmente, el cliente también se encarga de recibir las respuestas que devuelve el servidor. Estas respuestas pueden contener información de confirmación, errores, listas de usuarios conectados o contenidos publicados. El cliente interpreta estos mensajes y los muestra por pantalla, proporcionando al usuario información clara sobre el resultado de sus acciones.

Implementación

Servidor

En primer lugar, implementamos `server.c`, el cual contiene las funciones necesarias para crear el servidor y manejar las conexiones de los clientes. Este escucha en un puerto determinado y acepta conexiones entrantes. Las principales funciones del servidor son:

- **find_user**: Busca un usuario registrado en la lista de usuarios.
- **tratar_registro**: Es la función que atiende la petición `REGISTER` enviada por un cliente. Esta función se encarga de gestionar el registro de nuevos usuarios. Cuando un cliente envía una petición de registro con un nombre de usuario, el servidor verifica si ya existe un usuario con ese nombre. Si no existe y el número de usuarios registrados no ha alcanzado el límite (100), se almacena la información básica del nuevo usuario en la estructura global `users`. Se establece su estado como no conectado, pero registrado. En caso de que el nombre esté ya en uso o se haya alcanzado el máximo de usuarios, se devuelve un código de error correspondiente.
- **eliminar_usuario**: Es la función que atiende la petición `UNREGISTER` enviada por un cliente. Permite dar de baja a un usuario ya registrado. La función busca al usuario por su nombre y, si lo encuentra, lo elimina de la lista de usuarios registrados. Para mantener la integridad del array, el usuario eliminado es sustituido por el último usuario del array y se decrementa el contador global `num_users`. Esta técnica evita huecos y mantiene la eficiencia.
- **tratar_publish**: Es la función que atiende la petición `PUBLISH` enviada por un cliente. Gestiona la publicación de archivos por parte de los usuarios conectados. El servidor primero verifica si el usuario existe y está conectado. Si es así, recibe del cliente el nombre del archivo y su descripción. Se comprueba si el archivo ya ha sido publicado previamente por ese usuario. En caso afirmativo, se devuelve un código de error. Si no, se guarda la

publicación tanto en memoria (en la estructura del usuario) como físicamente en un archivo ubicado en `/tmp`. Esto permite persistencia del contenido y evita duplicados.

- **eliminar_archivo**: Es la función que atiende la petición `DELETE` enviada por un cliente. Se encarga de borrar una publicación específica de un usuario. Comprueba si el usuario existe y está conectado. Si encuentra la publicación con el nombre de archivo indicado, borra el archivo físico del sistema de archivos y elimina la entrada correspondiente de la lista de publicaciones del usuario. También se actualiza el contador de publicaciones.

- **enviar_lista_usuarios_conectados**: Esta función envía al cliente que la solicite una lista de todos los usuarios que están actualmente conectados. Se incluye para cada usuario su nombre, dirección IP y puerto de escucha. Primero se cuenta cuántos usuarios están conectados y se envía ese número, seguido de los datos de cada uno de ellos. Esta función permite a los clientes conocer a otros usuarios activos con los que pueden compartir archivos.

- **tratar_list_users**: Es la función que atiende la petición `LIST_USERS` enviada por un cliente. Comprueba si el usuario solicitante existe y está conectado. Si es así, invoca `enviar_lista_usuarios_conectados` para mandarle la información. Si el usuario no existe o no está conectado, se envía un código de error.

- **enviar_lista_contenido**: Es la función que atiende la petición `LIST_CONTENT` enviada por un cliente. Devuelve al cliente que lo solicita la lista de publicaciones (archivos) de un usuario en concreto. Se envía primero el número de publicaciones y, posteriormente, los nombres de los archivos publicados. Esta información es útil para que los clientes sepan qué contenidos están disponibles.

- **tratar_cliente**: Es la función principal que gestiona toda la comunicación con un cliente individual. Corre en un hilo separado por cada cliente y recibe comandos que interpreta y delega en las funciones correspondientes. Cada mensaje del cliente se analiza línea a línea y, dependiendo del comando recibido, se ejecuta la función que lo gestiona. También se encarga de leer información adicional como la IP y el puerto cuando es necesario.

En cuanto a las conexiones:

- **Conexión de usuarios**: Cuando el servidor recibe un mensaje con este comando, lo analiza dividiéndolo en sus componentes. Se espera que el mensaje contenga también el nombre de usuario, una dirección IP y un puerto. El servidor primero comprueba si el usuario está registrado buscando su nombre en la lista `users`. Si lo encuentra y no está ya conectado, se actualizan sus datos de IP, puerto y se cambia su estado a `conectado = 1`. A continuación, se devuelve al cliente un mensaje con el código `0` para indicar éxito. Si el usuario no existe o ya está conectado, se envía un código de error (1 o 2). Esta lógica se encuentra dentro de un bloque `if` que evalúa si la primera palabra del mensaje recibido es `"CONNECT"`.

- **Desconexión de usuarios**: Este comando permite a un usuario conectado cambiar su estado a desconectado sin eliminar su cuenta ni sus publicaciones. En el código, al recibir este comando, el servidor verifica si el usuario indicado existe en la lista de usuarios y si está actualmente conectado. Si ambas condiciones se cumplen, se cambia su estado a `conectado = 0`. En ese caso, se responde con un código `0` al cliente. Si el usuario no existe o no está conectado, se devuelve un código de error apropiado (1 o 2). Esta comprobación también se encuentra directamente dentro de un bloque `if`, sin una función aparte.

Por último, la **eliminación de usuario**: Cuando el servidor recibe este comando, interpreta que el usuario desea eliminar completamente su cuenta del sistema. Primero comprueba si el usuario está registrado y conectado. Si lo está, se recorren las publicaciones en la lista `publicaciones` y se eliminan aquellas asociadas al usuario, incluyendo los ficheros físicos correspondientes en el directorio `/tmp`. Luego, se elimina la entrada del usuario de la lista `users`. Si todo se ejecuta correctamente, se responde con un código `0` al cliente. En caso contrario, si el usuario no existe o no está conectado, se envía un código de error (1 o 2). Esta lógica también está integrada en el flujo principal del servidor, en un bloque `if` que trata específicamente los mensajes que comienzan con `"QUIT"`.

Cliente

El cliente, implementado en Python, permite a los usuarios interactuar con el servidor mediante el envío de comandos. Su principal función es proporcionar una interfaz que facilite el registro, conexión, publicación de archivos y otras operaciones, manteniendo una comunicación constante con el servidor a través de sockets TCP.

Las principales funciones del cliente son:

- **register**: Esta función envía una petición `REGISTER <nombre_usuario>` al servidor para registrar un nuevo usuario. Tras enviar el mensaje, espera una respuesta con un código que indique si el registro fue exitoso (código 0) o si ocurrió un error (usuario ya registrado o límite de usuarios alcanzado).

- **unregister**: Envía una petición `UNREGISTER <nombre_usuario>` al servidor. En esta función, también eliminamos la carpeta del usuario. El cliente interpreta la respuesta para confirmar si la eliminación fue correcta. Si el servidor devuelve un código de error, se informa al usuario.

- **connect**: Se encarga de enviar la orden `CONNECT <nombre_usuario>`. Esta operación permite al servidor registrar que un usuario está activo y disponible para compartir contenidos. La IP y el puerto son necesarios para que otros clientes puedan establecer comunicación directa si fuera necesario. El cliente analiza la respuesta del servidor para comprobar si la conexión ha sido aceptada o rechazada.

- **disconnect**: Permite enviar el comando `DISCONNECT <nombre_usuario>` al servidor. Esta orden informa de que el usuario desea quedar marcado como desconectado sin eliminar ni su cuenta ni sus publicaciones. El cliente muestra la respuesta recibida, que indica si la operación se ha ejecutado correctamente.

- **publish**: Esta función permite a un usuario ya conectado enviar una publicación. Para ello, el cliente envía el comando `PUBLISH <filename> <descripcion>`. Si el usuario está autorizado y el archivo no ha sido publicado antes, el servidor almacenará su información. En caso de error (archivo ya publicado o usuario no conectado), se devuelve un mensaje correspondiente.

- **delete**: Se invoca al enviar `DELETE <filename>`. El cliente solicita así al servidor que elimine una de las publicaciones del usuario. Tras enviar el comando, el cliente espera el código de estado del servidor que indicará si se ha eliminado correctamente o no se ha encontrado el archivo.

- **listusers**: Utiliza el comando `LIST_USERS` para solicitar al servidor la lista de todos los usuarios conectados en ese momento. El cliente procesa la respuesta, que contiene, por cada usuario, su nombre, IP y puerto.

- **listcontent**: Esta función envía `LIST_CONTENT <nombre_usuario_remoto>` y solicita al servidor una lista de todos los archivos publicados por ese usuario. El servidor devuelve los nombres de las publicaciones. Esta información es útil para saber qué contenido se puede solicitar para descargar.

Comunicación P2P (peer-to-peer)

El sistema está diseñado para que, una vez conectados al servidor y publicados los archivos, los clientes puedan compartir directamente sus archivos sin necesitar acceder al servidor. Este enfoque reduce la carga en el servidor y mejora la escalabilidad, ya que las transferencias de archivos ocurren de cliente a cliente. El cliente actúa así de forma dual: como cliente cuando realiza peticiones al servidor y como *servidor P2P (peer-to-peer)* cuando otro usuario le solicita un archivo. Para ello, cada cliente mantiene abierto un hilo secundario con un socket en modo escucha esperando peticiones entrantes. Esta arquitectura multihilo permite atender solicitudes remotas sin interrumpir la ejecución principal del cliente.

Una de las funcionalidades más interesantes del cliente es la descarga de archivos desde otros clientes. Para ellos, primero el cliente debe solicitar con `LIST_USERS` los usuarios conectados y, con `LIST_CONTENT`, los archivos que cada uno ha publicado si lo desea. Cuando el usuario decide descargar un archivo, se utiliza la función **getfile**. Esta función abre una conexión TCP directa con el cliente remoto (es decir, no pasa por el servidor), utilizando la IP y el puerto obtenidos anteriormente mediante `LIST_USERS`. Después, se envía una solicitud `GET_FILE <nombre_usuario_remoto> <nombre_archivo_remoto> <nombre_archivo_copia_local>`. El cliente que actúa como servidor P2P recibe esta solicitud (gracias a un hilo que escucha conexiones entrantes) y, si el archivo existe, lo envía por la red al cliente solicitante. Finalmente, el cliente receptor guarda el archivo en su sistema local.

También hicimos uso de funciones auxiliares que facilitan la interacción con el usuario y el manejo de datos:

- **_start_listening_thread**: Esta función lanza un hilo de escucha en segundo plano. Abre un socket en el puerto definido y acepta conexiones entrantes. Por cada nueva conexión, inicia un nuevo hilo para manejar la solicitud sin bloquear el resto del programa.

- **_handle_client**: Se ejecuta en cada hilo creado por el anterior. Procesa solicitudes de descarga P2P. Si la solicitud es válida (`GET_FILE`), busca el archivo solicitado en el directorio local correspondiente y, si existe, lo envía en bloques al cliente remoto.

- **_get_user_info**: Devuelve la IP y el puerto de un usuario específico almacenados en caché. Es útil para iniciar conexiones directas entre clientes.

- **_update_user_info**: Actualiza la información de IP y puerto de un usuario en la caché local.

- **_process_list_users_response**: Se encarga de interpretar la respuesta del servidor al comando `LIST_USERS`, extrayendo y almacenando la información de todos los usuarios conectados para su uso posterior en conexiones P2P

Observaciones importantes

Para que la descarga de archivos mediante el sistema **P2P** funcione correctamente, es imprescindible seguir un orden específico en la ejecución de los comandos: En primer lugar, es obligatorio ejecutar el comando `LIST_USERS`, ya que proporciona la IP y el puerto de los usuarios conectados. Sin esta información, no es posible establecer una conexión directa con otro cliente. Solo tras obtener esta información se puede proceder con la función `getfile`.

Es fundamental que el cliente remoto al que se desea solicitar un archivo esté conectado y mantenga activo su hilo de escucha. Si el cliente está desconectado o ha cerrado su socket de escucha, la conexión P2P no podrá establecerse y la descarga fallará. Además, aunque un archivo esté publicado, si no se encuentra físicamente en el directorio `/tmp/archivos_<usuario>`, el cliente remoto no podrá servirlo. Esto implica que los archivos compartidos deben existir localmente en el momento de la solicitud.

Parte 2 y 3

Este apartado corresponde a las partes dos y tres de la práctica, estando cada una enfocadas a la implementación de servicios web para obtener la fecha y hora de las operaciones cliente y en la devolución de las mismas junto a la operación realizada y el nombre de usuario, respectivamente.

Servicios Web

Diseño

Hemos utilizado el framework de python *Flask* para implementar el servicio web siguiendo el estilo *REST*. Para ello, hemos creado un nuevo fichero llamado `web_service.py`, que será al que llamemos desde terminal para inicializar el servidor correspondiente a servicio web.

Por otro lado, los clientes se pondrán en contacto con este servidor para poder recibir la fecha y hora exactas del momento de sus peticiones. Esta información será almacenada localmente por los clientes para, posteriormente, incluirla en el mensaje que envíen al servidor principal.

Implementación

El código desarrollado en `web_service.py` es muy sencillo, compuesto principalmente por una función que se encarga de sacar la hora actual y darle el formato especificado, y una función main que pone en marcha la aplicación en *localhost* a través del puerto 5000, el cual es el puerto por defecto de Flask.

Para los clientes, hemos tenido que crear una nueva función llamada `get_timestamp`. Esta función se encarga de llamar a la función `get_time` del servicio web, almacenada en *http://127.0.0.1:5000/get_time*, es decir, en el puerto 5000 del localhost. Ya teniendo esta función definida, simplemente se invoca desde cada uno de los métodos del cliente para obtener la marca temporal (*timestamp*) que se incluirá posteriormente en el mensaje enviado al servidor.

Un ejemplo de uso muy reducido sería el siguiente:

- El cliente llama a la función correspondiente a la operación que quiere realizar
- Lanza la solicitud de fecha y hora actuales mediante una llamada a `get_timestamp()`
- Esta función consulta el servicio web, que devuelve una cadena en el formato XX/XX/XX XX:XX:XX
- El cliente recibe la cadena y se lo guarda en la variable *timestamp*
- FInalmente, el cliente manda el *timestamp*, junto con el resto de los datos necesarios, al servidor

Para más detalles sobre cómo utiliza el servidor esta información, ver la sección correspondiente a RPC.

RPC

Diseño

Para la implementación de esta parte hemos creado un subdirectorio llamado *`server_rpc`*. Dado que para RPC necesitamos generar varios ficheros de forma automática, de esta manera garantizamos tener nuestro directorio raíz más ordenado, teniendo todo lo relacionado con RPC en su propio subdirectorio específico.

El servidor RPC implementado se comunicará con el cliente desarrollado en la parte 1 del proyecto. Este último actuará como cliente en cierto modo, enviando los datos y recibiendo una respuesta. Para cada función que realice el servidor, nuestro servidor RPC devolverá una línea con el cliente que solicitó la operación, la operación realizada y la fecha y hora de la solicitud.

Implementación

Lo primero que hicimos fue crear un fichero `.x` para poder generar los archivos necesarios (`server_rpc.x`). En este declaramos una estructura `data`, que contiene tres variables de tipo string: el nombre de usuario, la operación y el timestamp. A continuación, declaramos la función `rpc_operation`, que es la única que necesitaremos para nuestro servicio RPC.

Una vez generados los archivos, desarrollamos la función correspondiente en `server_rpc_server.c`. Esta función es bastante sencilla, ya que simplemente devuelve los parámetros de entrada en el formato solicitado: `usuario operación fecha hora`.

A continuación, aplicamos los cambios necesarios en `server.c` para integrar el servicio RPC. EL primer paso fue importar la librería RPC y generar un *makefile* para que la compilación de ambos servidores se realice al mismo tiempo. Este paso es crucial, ya que el servidor necesita el fichero de cabecera `.h` generado por RPC y estar bien sincronizado con esa sección, dado que actúa como cliente para el servicio.

Una vez hecho esto, creamos la función `handle_operation` que se encarga de conectar al servidor con el servidor RPC. Dentro de esta función, creamos el cliente, definimos y reservamos memoria para los datos que se van a enviar, y luego hacemos la llamada al servidor. Después de completar la solicitud, destruimos el cliente y liberamos la memoria utilizada, quedando listos para procesar la siguiente solicitud proveniente de `cliente.py`.

El uso de `handle_operation` es sencillo: solo debemos llamarla una vez al inicio de la función `tratar_cliente` y pasarle los argumentos que recibimos de dicha función a través de *sscanf*. La estructura del mensaje es la misma en todos los casos, donde las dos primeras variables que recibe el servidor son la operación y el nombre de usuario. Gracias a la forma en que extraemos los datos, al leer el nombre de usuario asignamos el resto del mensaje del cliente a la variable `timestamp`. De esta forma, si la operación es `publish` o `delete`, la ruta del archivo se incluirá dentro del `timestamp` sin afectar el formato de salida.

Pila de pruebas

Una vez finalizada la implementación del sistema, realizamos una serie de pruebas para verificar su correcto funcionamiento

1. **Registro y eliminación de usuarios (Register y Unregister)**: Probamos a registrar un usuario para, posteriormente, eliminarlo. También intentamos registrar un segundo usuario con el mismo nombre que uno ya registrado, lo cual fue correctamente impedido por el sistema.

2. **Conexión y desconexión (Connect y Disconnect)**: Registramos un usuario, lo conectamos al sistema y luego lo desconectamos. El funcionamiento fue correcto en todos los casos. Además, verificamos que se creara un subdirectorio en `/tmp/` para almacenar los archivos del cliente conectado.

3. **Publicación de archivos (Publish)**: Tras registrar y conectar un cliente, se procedió a publicar un archivo. El sistema respondió de forma adecuada y el archivo quedó correctamente almacenado.

4. **Listado de usuarios conectados (List_users)**: Abrimos tres terminales con tres clientes diferentes conectados al sistema y ejecutamos esta operación desde uno de ellos. El listado mostró correctamente los tres usuarios activos, junto con sus respectivas direcciones IP (idénticas por estar en la misma máquina) y puertos. Al desconectar alguno de ellos, el listado se actualizaba adecuadamente.

5. **Listado de contenidos publicados (List_content)**: Publicamos varios archivos desde dos usuarios distintos (a y b) y ejecutamos esta operación desde las terminales de ambos, tanto para visualizar sus propios contenidos como los del otro usuario. En todos los casos, los listados fueron precisos y coherentes.

6. **Descarga de archivos (Get_file)**: Registramos y conectamos dos clientes, a y b. El cliente a publicó dos archivos. El cliente b ejecutó `list_users` y `list_content` para visualizar los datos del cliente a, y finalmente descargó uno de los archivos mediante `get_file`. Al comprobar el sistema de archivos, pudimos ver que el subdirectorio de *a* contenía ambos archivos y el de *b* había recibido una copia del archivo descargado, con el nombre especificado.

7. **Eliminación de archivos (Delete)**: Intentamos borrar archivos previamente publicados por un cliente. Si el archivo no existía, el sistema mostraba un mensaje de error adecuado. Solo se permitió eliminar archivos pertenecientes al usuario activo y listados en su propio `list_content`.

8. **Finalización del cliente (Quit)**: Al escribir `quit` en la terminal del cliente, el programa finalizaba correctamente su ejecución. Además, comprobamos que se eliminaba automáticamente el subdirectorio correspondiente al cliente en `/tmp/`, tal como se había implementado.

Compilación

Para el correcto funcionamiento del sistema, se requieren al menos cuatro terminales independientes, cada una destinada a ejecutar uno de los siguientes componentes:

- Servidor principal
- Servidor web
- Cliente(s)
- Servidor RPC

Importante: El servidor RPC se encuentra en un subdirectorio distinto (`server_rpc`), por lo que su terminal debe estar ubicada en dicha carpeta para su ejecución.

Desde el directorio raíz del proyecto ejecutamos los siguientes comandos:

```
export LOG_RPC_IP=localhost

make
```

Esto compila todos los archivos `.c` necesarios, incluyendo aquellos que hacen uso del servicio RPC.

Ahora, en cada terminal ejecutamos lo siguiente:

- **Servidor principal:** `./server -p <num_puerto>`
- **Servidor web:** `python3 web_service.py`
- **Cliente(s):** `python3 client.py -s localhost -p <num_puerto>`
- **Servidor RPC**:
    - Primero accedemos al subdirectorio mediante `cd server_rpc`
    - Después, ejecutamos `./server_rpc_server`

Podemos tener varios clientes abiertos en distintas terminales.

Una vez todos los servidores estén activos, el sistema estará listo para recibir y procesar comandos desde las terminales de los clientes, permitiendo realizar todas las operaciones disponibles.

Conclusión

Gracias a esta práctica, hemos podido comprender el funcionamiento de los servidores y los sistemas distribuidos, que constituyen la base de la gran mayoría de los servicios utilizados día a día. Hemos comprobado la importancia de una arquitectura bien diseñada y cómo es posible sincronizar diferentes programas, incluso cuando están desarrollados en distintos lenguajes de programación. Esta capacidad de integración resulta fundamental, ya que permite la conectividad entre distintos sistemas, los cuales pueden funcionar de forma independiente y coordinada dentro de un entorno común.

Ha sido una experiencia especialmente enriquecedora, ya que no solo hemos trabajado aspectos técnicos como la programación o la gestión de procesos concurrentes, sino que también hemos aprendido a resolver problemas reales relacionados con la comunicación entre sistemas.

Como punto a destacar, nos ha llamado la atención un comportamiento inesperado: durante la ejecución del proyecto salta un mensaje de DEBUG, a pesar de que la línea correspondiente en el código está comentada. No hemos logrado entender por qué ocurre esto.