Sistemas Distribuidos

Grado de Ingeniería en Informática
Curso 2023-2024

Práctica Final

Integrantes del Grupo:
María Isabel Hernández Barrio (100472315, 100472315@alumnos.uc3m.es)
Blanca Ruiz González(100472361, 100472361@alumnos.uc3m.es)

ÍNDICE:

1. Introducción:

Para este proyecto desarrollaremos un sistema peer-to-peer de distribución de archivos entre clientes, utilizando como base los conceptos de comunicación de procesos mediante sockets TCP.

El proyecto se divide en dos partes: la primera parte implica el diseño y desarrollo de un servidor concurrente multihilo en el lenguaje C, mientras que la segunda parte requiere la implementación de un cliente multi hilo en Python.

El servidor es el encargado de coordinar las operaciones del sistema y los clientes interactúan con el servidor y entre sí para compartir e interactuar con los archivos. A través de una serie de comandos definidos, los usuarios pueden registrar sus nombres, publicar archivos, listar usuarios conectados y solicitar la descarga de archivos específicos.

2. Cliente:

El cliente se desarrolla en Python. Tiene los siguientes métodos:

- **register**: Este método se encarga de registrar un nuevo usuario en el sistema. Toma como argumento el nombre de usuario (user) y realiza las siguientes acciones:

  Establece una conexión con el servidor y envía el comando "REGISTER" al servidor. Después, obtiene la fecha y hora del servidor web y la envía al servidor. Envía el nombre de usuario al servidor y recibe y procesa la respuesta del servidor. Si recibe un 0 significa que el proceso ha sido exitoso y se imprime por consola: REGISTER OK. De lo contrario, si se recibe un 1 significa que el nombre de usuario ya está en uso y se imprimirá: USERNAME IN USE. En el caso de que ocurra cualquier otro error se recibirá un 2 y se imprimirá: REGISTER FAIL.

- **unregister**: Es similar al método de registro, pero para anular el registro de un usuario en el sistema. Realiza operaciones similares de comunicación con el servidor y procesa la respuesta correspondiente. Sea la respuesta 0, 1 o 2, tiene el mismo significado que para el método register.

- **connect**: Este método se utiliza para que un usuario se conecte al sistema. Primero comprueba que la longitud del nombre del usuario no exceda los 256 caracteres. Después obtenemos un puerto válido libre para escuchar las solicitudes de otros usuarios y creamos un método que haga de hilo servidor para las conexiones entre clientes. Una vez hecho esto, se realiza la conexión al servidor y se envía el comando "CONNECT", la fecha y hora, el nombre de usuario y el puerto en el que el cliente está escuchando para las solicitudes de descarga. Luego procesa la respuesta del servidor. En caso de éxito recibiremos el código 0 y se imprimirá CONNECT OK. En el caso de que el código sea 1, imprimimos CONNECT FAIL , USER DOES NOT EXIST, si es 2 USER ALREADY CONNECTED y si es 3 es cualquier otro tipo de error y se mostrará CONNECT FAIL.

- **disconnect**: Desconecta a un usuario del sistema. Envía el usuario al servidor y este se encarga de desconectarlo del sistema. Si la respuesta del servidor tiene código 0 es que la desconexión ha sido exitosa y se imprime DISCONNECT OK. Si se recibe un 1 se imprime DISCONNECT FAIL / USER DOES NOT EXIST, si se recibe un 2 DISCONNECT FAIL / USER NOT CONNECTED y con un 3 significa que tenemos otro tipo de error e imprimimos DISCONNECT FAIL.

- **publish**: Permite a un usuario publicar un archivo en el sistema. Toma el nombre del archivo y una descripción como argumentos. Se conecta al servidor, envía el comando "PUBLISH", la fecha y hora, el nombre del archivo y la descripción, y luego procesa la respuesta del servidor. En caso de éxito la respuesta es 0 y se imprime PUBLISH

OK. Por otro lado, si recibimos un 1 imprimimos PUBLISH FAIL , USER DOES NOT EXIST, si es un 2 PUBLISH FAIL , USER NOT CONNECTED, si es un 3 PUBLISH FAIL , CONTENT ALREADY PUBLISHED y en cualquier otro caso imprimimos PUBLISH FAIL.

- **delete**: Permite a un usuario eliminar un archivo que ha publicado previamente en el sistema. Se conecta al servidor, envía el comando "DELETE", la fecha y hora, el nombre del archivo y procesa la respuesta del servidor. Como siempre en caso de éxito recibe 0 e imprime DELETE OK. De lo contrario, si recibe un 1 imprime DELETE FAIL , USER DOES NOT EXIST, si es un 2 DELETE FAIL , USER NOT CONNECTED, si es un 3 DELETE FAIL , CONTENT NOT PUBLISHED y en cualquier otro caso DELETE FAIL.

- **listusers**: Solicita al servidor la lista de usuarios conectados al sistema. Se conecta al servidor, envía el comando "LIST_USERS", la fecha y hora, y procesa la respuesta del servidor para mostrar la lista de usuarios. Si la respuesta del servidor es 0, se imprimirá LIST_USERS OK y creamos un diccionario en el que vamos a guardar los nombres de los usuarios, su dirección IP y su puerto. Por otro lado si recibimos un 1 imprimimos LIST_USERS FAIL , USER DOES NOT EXIST, si es un 2 LIST_USERS FAIL , USER NOT CONNECTED y en cualquier otro caso LIST_USERS FAIL.

- **listcontent**: Solicita al servidor la lista de archivos publicados por un usuario específico. Se conecta al servidor, envía el comando "LIST_CONTENT", la fecha y hora, el nombre de usuario y procesa la respuesta del servidor para mostrar la lista de archivos. Si la respuesta del servidor es 0, se imprime LIST_CONTENT OK y recibimos el número de archivos del usuario y su información. En cualquier otro caso estaremos ante un error. Si recibimos un 1 imprimimos LIST_CONTENT FAIL , USER DOES NOT EXIST, si es un 2 LIST_CONTENT FAIL , USER NOT CONNECTED, si es un 3 LIST_CONTENT FAIL , REMOTE USER DOES NOT EXIST y en otro caso LIST_CONTENT FAIL.

- **getfile**: Permite a un usuario descargar un archivo de otro usuario. Toma como argumentos el nombre del usuario remoto, el nombre del archivo remoto y el nombre del archivo local donde se guardará. Realiza la conexión con el usuario remoto, envía el comando "GET_FILE", el nombre del archivo remoto y procesa la respuesta del servidor remoto para descargar el archivo.

3. Servidor:

Vamos a tener tres servidores. Uno en C, que es el que se va a encargar de proporcionar al cliente las respuestas para cada método, otro en python que va a establecer la conexión web y va a enviar al cliente la fecha y hora actuales cada vez que se haga una operación.

Primero vamos a empezar explicando el **servidor.c**:

En el main primero comprobamos que tenemos el número de argumentos correctos y una vez comprobado llamamos a las funciones crearDirUsuarios() y crearDirConexiones() que sirven para crear un directorio para guardar los usuarios y sus conexiones. Después creamos y vinculamos los sockets a la dirección del servidor y escuchamos las conexiones entrantes. Creamos un bucle infinito para las conexiones con el cliente. Aquí podemos encontrar la llamada a la función principal del servidor, "handle_client()".

En "handle_client()" primero hemos usado la función "readLine" para poder leer la información hasta el byte \0. Después usamos strncmp() para comparar el mensaje del cliente con diferentes comandos y ejecutar la acción correspondiente. Si el mensaje no coincide con ningún comando conocido, se imprime un mensaje de error y se liberan los recursos antes de finalizar el hilo. Cada operación tiene una función para llevar a cabo su funcionamiento:

- REGISTER: Hemos creado la función "crearDirUser" para poder registrar a los usuarios.

- UNREGISTER: Hemos creado la función "darDeBaja" que nos permite eliminar el usuario registrado.

- CONNECT: Para conectar a los usuarios tenemos "conectarUser". Este conecta el usuario, dentro de un fichero con su nombre se guarda el puerto y la ip. La dirección IP del cliente conectado la obtenemos con getpeername.

- DISCONNECT: Desconectamos a los usuarios mediante "disconnectUser". Para ello construimos el path del archivo del user en Conexiones y eliminamos su archivo, que indica que está conectado.

- PUBLISH: Recibimos en nombre del archivo y la descripción de su contenido con el "readLine". Con "usuarioDeEstaIP" obtenemos el usuario correspondiente del cliente conectado y con "saveFile" guardamos el archivo.

- DELETE: Recibimos el nombre del fichero con "readLine". Con "usuarioDeEstaIP" obtenemos el usuario correspondiente del cliente conectado. "deleteFile" para eliminar el archivo que construye la ruta completa del archivo para el username proporcionado y lo elimina.

- LIST_USERS: Con “listarUsers” se leen los archivos en un directorio específico, se interpreta cada archivo como un usuario conectado y se recopila información sobre ellos en una lista, que luego se devuelve para su uso posterior. Después se itera sobre cada usuario en la lista lista_usuarios y se imprime en la consola los datos del usuario. Por último, se envían los datos del usuario al cliente a través del socket utilizando la función sendLine(), que envía una línea completa de datos.

- LIST_CONTENT: Se obtiene la dirección IP del cliente conectado utilizando la función getpeername(). Se comprueba si el usuario que hace la operación está registrado y conectado con “isRegistered” y “isConnected”. Se llama a la función “listarContenido” para obtener la lista de archivos del usuario. Después se itera sobre cada archivo en la lista lista_archivos y se envía cada archivo al cliente a través del socket.

Respecto al **rpcgen**, hemos creado el archivo “operaciones.x” para enviar todos los datos. El usuario se guarda en la clase client de python.

En el servidor.c, al final de la función “handle_client” gestionamos la conexión con el **servidor RPC:**
Se envía la respuesta al cliente, excepto para LIST_USERS y LIST_CONTENT, que la envian por su cuenta. Con “get_operation_1” enviamos todos los datos (el nombre del usuario, la operación que realiza y la fecha que se recibe con la petición) al **servidor RPC.**
Realmente hemos dejado el protocolo UDP, que es el que rpcgen utiliza por defecto.

En segundo lugar, hemos creado el **servidorweb.py**. Consta de un servicio web que proporciona la fecha y la hora actual cuando se realiza una llamada al método “fecha_hora”. Se define una clase FechaHora que hereda de ServiceBase. Dentro de esta clase, se define un método llamado fecha_hora. Este método no tiene parámetros y devuelve la fecha y la hora actuales formateadas como una cadena. Después, se imprime un mensaje indicando que el servidor está escuchando en la dirección http://127.0.0.1:8000 y que el WSDL está disponible en http://localhost:8000/?wsdl. Creamos un servidor utilizando “make_server” en el puerto 8000 y se inicia el servidor para que esté siempre disponible y maneje las solicitudes entrantes.
En el cliente para cada método tenemos que añadir:

```
wsdl = "http://localhost:8000/?wsdl"
    cliente_fh = zeep.Client(wsdl=wsdl)
    fecha_y_hora = cliente_fh.service.fecha_hora()
    cliente_fh.transport.session.close() # Desconectarse del serverweb
    msg_fh = (fecha_y_hora + '\0').encode()
    print(msg_fh)
```

De esta manera se conecta al Servidor Web y obtenemos la fecha y hora proporcionadas y las usamos para imprimirlas cada vez que hagamos una operación.

4. Descripción del proceso de compilación:

Para compilar tenemos nuestro Makefile. En este Makefile solo hemos incluido el servidor y el servidor RPC. Estos son los pasos que seguimos para ejecutar el proyecto.

1. Hacer Make: generará todos los archivos del servidor RPC y su cliente, que es nuestro servidor principal (el de la primera parte, ahora llamado operaciones_client.c).
2. Poner Servidor RPC en marcha, servirá desde el localhost:

   ```
   ./operaciones_server
   ```
3. Poner Servidor Principal en marcha, servirá desde el host y puerto que le especificamos, en nuestro caso el 8085 y el "192.168.1.62", que es uno que tenía disponible nuestro dispositivo:

   ```
   ./operaciones_client -p 8085 -r 192.168.1.62
   ```
4. Poner el Servidor Web en marcha, en el puerto 8000:

   ```
   python3 serverweb.py &
   python3 -mzeep http://localhost:8000/?wsdl
   ```

5. Lanzar nuestros clientes, uno en cada terminal (usamos una terminal o cliente por usuario):

   ```
   python3 client.py -s 192.168.1.62 -p 8085
   ```

5. Batería de pruebas:

Para realizar la batería de pruebas hemos hecho lo siguiente:

- REGISTER: registrar un usuario, registrar un usuario ya registrado.

```
c> REGISTER Isabel
REGISTER OK

c> REGISTER Isabel
USERNAME IN USE
```

- UNREGISTER: desregistramos usuario, desregistramos que no está registrado.

```
c> UNREGISTER Isabel
UNREGISTER OK
```

```
c> UNREGISTER Paula
USER DOES NOT EXIST
```

(Volvemos a registrar a “Isabel” después de esto, para usarlo en las demás pruebas)

- CONNECT: Conectamos un usuario, conectamos un usuario no registrado, conectamos un usuario ya conectado.

```
c> REGISTER Isabel
REGISTER OK

c> CONNECT Isabel
CONNECT OK

c> CONNECT Paula
CONNECT FAIL, USER DOES NOT EXIST

c> CONNECT Isabel
USER ALREADY CONNECTED
```

- CONNECT: desconectamos usuario conectado, desconectamos usuario no registrado, desconectamos usuario no conectado.

```
c> DISCONNECT Isabel
DISCONNECT OK

c> DISCONNECT Tarzan
DISCONNECT FAIL / USER DOES NOT EXIST

c> DISCONNECT Isabel
DISCONNECT FAIL / USER NOT CONNECTED
```

Para estas pruebas hemos usado una misma terminal para ver cómo va funcionando, de ahora en adelante usaremos una terminal por usuario, para dejarlo todo separado y claro.

- PUBLISH: Usuario no registrado publica un archivo, usuario no conectado publica un archivo, usuario registrado y conectado publica un archivo. Publicar un archivo ya publicado.

```
c> PUBLISH archivo_desconocido.txt Archivo publicado por un usuario no registrado
PUBLISH FAIL, USER DOES NOT EXIST

c> REGISTER Cristian
REGISTER OK

c> PUBLISH archivoC.txt Descripcion de un archivo de Cristian, que no se ha conectado
PUBLISH FAIL, USER NOT CONNECTED

c> CONNECT Cristian
CONNECT OK

c> PUBLISH archivoC.txt Descripcion de un archivo guay de Cristian
PUBLISH OK
```

```
c> PUBLISH archivoC.txt Este archivo ya habia sido publicado..
PUBLISH FAIL, CONTENT ALREADY PUBLISHED
```

- DELETE: Usuario válido borra un archivo (justo el de antes), usuario no registrado borra un archivo (nueva terminal sin registrar a nadie), usuario no conectado borra un archivo (registramos a alguien y operamos) y usuario borra contenido inexistente.

```
c> DELETE archivoC.txt
DELETE OK
```

Tenemos algún fallo en el delete, porque nos deja eliminar archivos que no existen o con usuarios no autorizados.

- LIST_USERS: pedir lista de usuarios sin estar registrado, pedir lista de usuarios sin estar conectado, pedir lista de usuarios estando registrado y conectado. Vemos que

hay usuarios con la misma dirección ip porque lo estoy ejecutando en un mismo portatil.

```
c> LIST_USERS
LIST_USERS FAIL, USER DOES NOT EXIST
c> REGISTER Usuario_Listo
REGISTER OK

c> LIST_USERS
LIST_USERS FAIL, USER NOT CONNECTED
c> CONNECT Usuario_Listo
CONNECT OK

c> LIST_USERS
LIST_USERS OK
        Usuario_Listo 34095 0.0.0.0
        Isabel 36537 192.168.1.62
        Cristian 43711 192.168.1.62
c>
```

- LIST_CONTENT: lo mismo, más pedir la lista de contenido de un usuario que no existe. También, pedir una lista de contenido de un usuario que no tiene nada publicado.

```
c> LIST_CONTENT Cristian
LIST_CONTENT FAIL , USER DOES NOT EXIST
c> REGISTER Nuevo
REGISTER OK

c> LIST_CONTENT Cristian
LIST_CONTENT FAIL , USER NOT CONNECTED
c> CONNECT Nuevo
CONNECT OK

c> LIST_CONTENT Cristian
LIST_CONTENT OK
        test3.txt "Un archivo no tan interesante"
        test2.txt "Un archivo interesante"
c> LIST_CONTENT User_Inexistente
LIST_CONTENT FAIL , REMOTE USER DOES NOT EXIST
```

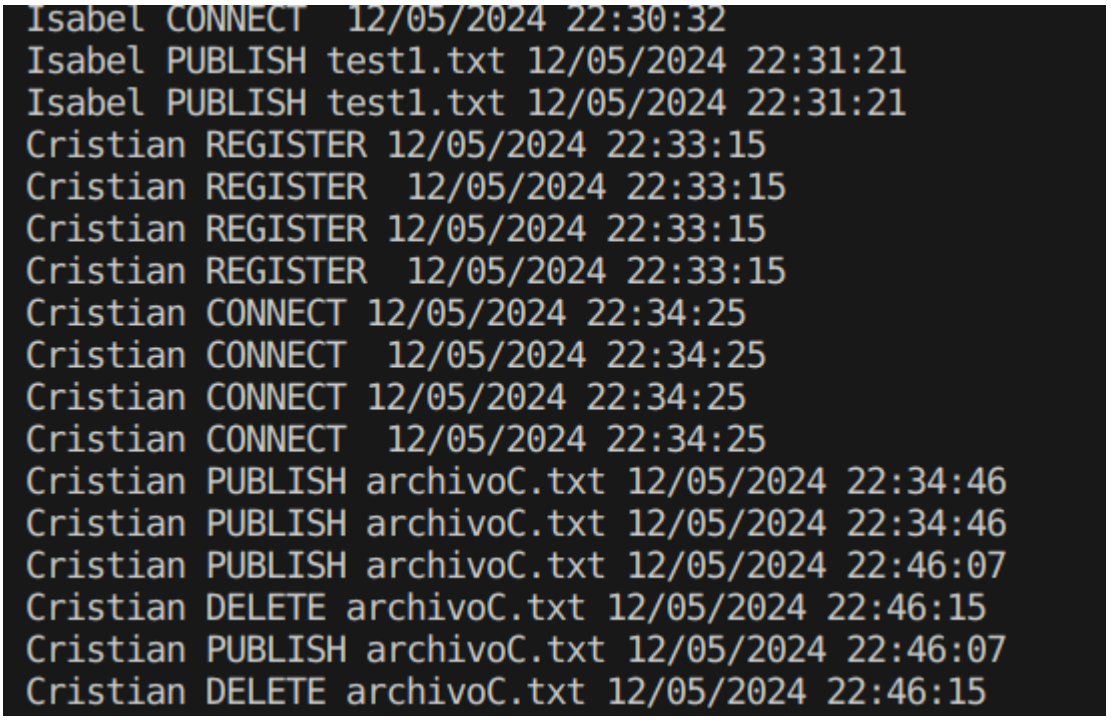
```
c> LIST_CONTENT Pepe
LIST_CONTENT OK
```

- GET_FILE:

No nos ha dado tiempo de implementarlo funcionalmente. Dejamos el código en el cliente.py, pero no hemos sido capaces de hacerlo funcionar a tiempo.

- Servidor RPC:

Dejamos aquí una captura de lo que obtiene nuestro servidor RPC, ha habido un problema que no hemos solucionado por falta de tiempo, que es que parece que el servidor rpc no cierra bien sus conexiones y recibe varias veces una operación que ya ha sido procesada y cerrada, y cuyo hilo ya ha sido terminado. Se ve aquí reflejado:



- Servidor Web: el servidor web sí que funciona correctamente, ya que obtiene la fecha y la hora como se solicita y esta se envía al Servidor General y de ahí al Servidor RPC.

6. Conclusiones:

Hemos enfrentado problemas para la transferencia de bytes de un servidor c a un cliente python y viceversa, sobretodo al hacer varios sends seguidos de diferentes datos que queríamos mantener por separado o enviar y recibir sólo los bytes utilizados y no los máximos, tomó bastante tiempo hasta que lo pudimos conseguir.

También hemos tenido problemas de última hora al testear el delete. Pensábamos que funcionaba correctamente, y así parecía en pruebas realizadas con anterioridad, pero algún fallo ha habido que ha provocado que no funcione como se espera.

Además hay un fallo con el Servidor RPC, que recibe varias veces una misma operación en ocasiones, si no se realiza otra inmediatamente.

El problema principal que ha impedido el desarrollo adecuado del get_file ha sido la limitación de probarlo con una sola máquina, ya que creo que daba problemas al tener todos nuestros clientes la misma dirección ip, ya que los servidores de los usuarios no aceptaban conexiones de otros.

A pesar de todo, estamos satisfechos con el resultado final, ya que hemos puesto muchas horas de trabajo y aprendido mucho sobre cómo construir un sistema distribuido, el envío de datos y la relación cliente-servidor, además de adquirir mucha familiarización con el lenguaje de C, que desde el principio de la asignatura suena un poco intimidante.