Universidad Carlos III
Sistemas Distribuidos
Curso 2024-25

Proyecto Final

Tercer curso - Ingeniería Informática

100441823, Martínez López, Noelia (100441823@alumnos.uc3m.es)
100448737, Loor Weir, Iván Sebastián (100448737@alumnos.uc3m.es)

Índice

1. Descripción del código

Parte 1

a) Client.py:

Comunicación con el servidor:

- connect_to_server(): Se establece conexión con el servidor.
- send_string() y receive_string(): Se maneja el envío y recepción de cadenas a través de sockets.

Gestión de usuarios:

- register(): Registra un usuario en el servidor.
- unregister(): Elimina el registro del usuario en el servidor.
- connect(): Conecta un usuario al sistema.
- disconnect(): Desconecta un usuario del sistema.

Gestión de archivos:

- publish(): Publica un archivo con una descripción.
- delete(): Elimina un archivo publicado.
- getfile(): Obtiene un archivo específico.
- handle_file_transfer(): Maneja la transferencia de archivos entre clientes

Funciones de listado:

- listusers(): Muestra la lista de usuarios conectados.
- listcontent(): Muestra el contenido publicado por un usuario.

Funciones de escucha:

- create_listener_socket(): Crea un socket para escuchar peticiones entrantes.
- listener_thread(): Hilo que maneja las conexiones entrantes.

Interfaz de usuario:

- shell(): Intérprete de comandos para interactuar con el programa.
- parseArguments(): Procesa los argumentos de línea de comandos.

b) Server.c:

Gestión del servidor:

- main(): Inicializa el servidor, crea el socket principal y maneja las conexiones entrantes.
- handle_sigint(): Maneja la señal de interrupción (Ctrl+C) para cerrar correctamente el servidor.

Gestión de conexiones:

- handle_client(): Función principal que maneja las peticiones de los clientes (registro, desconexión, etc.).
- read_string(): Lee cadenas terminadas en null desde un socket.

Gestión de usuarios:

- add_user(): Añade un nuevo usuario al sistema.
- remove_user(): Elimina un usuario del sistema.
- find_user(): Busca un usuario por nombre.
- connect_user(): Marca un usuario como conectado y guarda su IP y puerto.
- disconnect_user(): Marca un usuario como desconectado.
- free_user_resources(): Libera recursos asociados a un usuario.

Gestión de archivos:

- add_file(): Añade un archivo a la lista de publicaciones de un usuario.
- remove_file(): Elimina un archivo de la lista de publicaciones de un usuario.

El servidor usa mutexes (pthread_mutex_lock/unlock) para proteger el acceso concurrente a los datos de los usuarios.

Parte 2

a) Web_service.py:

Para configurar la aplicación, se crea una instancia de la aplicación Flask. Se define la ruta '/fecha' para peticiones GET y se controla la API mediante la función:

- obtener_fecha(): Función que maneja las peticiones a la ruta '/fecha', obteniendo la fecha y hora actual, formateando en el formato DD/MM/YYYY HH:MM y convirtiendo un diccionario Python en una respuesta JSON.

Terminado esto, se inicia el servidor web Flask, configurado para escuchar en todas las interfaces (0.0.0.0) en el puerto 5000.

b) Client.py:

En este nuevo cliente, se incluye la biblioteca requests para realizar peticiones HTTP y se añade un atributo para almacenar la URL del servicio web. Además, se implementa un método get_datetime() que consulta el servicio web y devuelve la fecha y hora actual en formato string. Por último, se modifican todos los métodos de comunicación con el servidor para incluir información temporal en cada operación.

c) Server.c:

Se añaden unas pocas líneas con respecto al servidor antiguo. Se lidia con la variable datetime para almacenar la fecha y hora que se recibe del cliente. Después de leer la operación, el servidor lee dicha cadena de fecha y hora, mostrando la marca de tiempo en sus mensajes de log.

Parte 3

a) Log_service.x:

Se define la interfaz RPC para un servicio de registro de logs. Se declara una estructura llamada log_data que contiene el nombre del usuario, la operación realizada y la marca temporal. Y después, definimos un programa RPC identificado como LOG_PROG con versión LOG_VERS, que ofrece un único procedimiento remoto llamado log_operation, el cual recibe un log_data y devuelve un entero como resultado.

b) Log_client.c:

Se crea la función send_log que sirve como cliente para enviar logs al servidor RPC. Para ello, recibe tres parámetros: el usuario que realizó la operación, el tipo de operación ejecutada y la marca temporal del evento. Para determinar el destino del mensaje, se obtiene la dirección IP del servidor RPC a través de la variable de entorno "LOG_RPC_IP". Una vez obtenida esta dirección, prepara los datos en una estructura adecuada para el envío mediante el protocolo RPC. Posteriormente, establece una conexión TCP, envía los datos mediante el procedimiento remoto log_operation_1, maneja posibles errores y libera los recursos.

c) Log_service_impl.c:

Aquí se implementa el servicio RPC de logging, centrado en la función log_operation_1_svc, que es la función que ejecuta el servidor cuando recibe una solicitud del procedimiento remoto log_operation. La función recibe una estructura log_data con el nombre de usuario, la operación realizada y la marca temporal, y simplemente imprime esta información por la salida estándar. Devuelve un entero

estático con valor 1 para indicar que la operación se completó con éxito. En resumen, log_operation_1_svc actúa como un manejador que registra en consola los logs enviados por clientes remotos.

d) Web_service.py:

Se mantiene igual que en la parte 2

e) Client.py:

Se mantiene igual que en la parte 2

f) Server.c:

Inicialmente, se añade la inclusión del cliente RPC en los archivos de cabecera del servidor. También, se declara la función externa send_log() del mismo para posterior uso. Tal uso se dará para cada operación que procese el servidor, llamando al servicio RPC en cada (register, unregister, connect…), pero con la diferencia de añadir una salida formateada junto al nombre de archivo específico que tengan las operaciones publish y delete.

2. Descripción de la forma de compilar y obtener el ejecutable de todos los procesos involucrados

Parte 1

En este caso, no es necesario hacer uso de un archivo Makefile ya que solo necesitamos compilar el servidor de manera concurrente, con la biblioteca específica de hilos. Solo nos basta con escribir:

```
Unset
gcc -o server server.c -lpthread
```

Parte 2

Así como en la parte 1, realizamos el mismo comando para compilar el proyecto, debido a que solo se añadió el servicio web en lenguaje python:

```
Unset
gcc -o server server.c -lpthread
```

Parte 3

- Prerrequisitos

Antes de compilar y ejecutar el proyecto, se necesitan instalar las bibliotecas de desarrollo de TI-RPC:

```
Unset
sudo apt-get install libtirpc-dev
```

(Se da por obvio que ya está instalado rpcbind)

- Uso del Makefile

Se generan los archivos RPC para el servicio de logs:

- log_service.h: Archivo de cabecera con definiciones
- log_service_clnt.c: Código del cliente RPC
- log_service_svc.c: Código del servidor RPC
- log_service_xdr.c: Código para serialización/deserialización XDR

```
rpcgen -C log_service.x
```

Luego, como el sistema usa la biblioteca TI-RPC (versión más moderna de RPC), se modifican los archivos generados reemplazando las inclusiones de cabeceras estándar de RPC por las versiones de TI-RPC.

```
sed -i 's/<rpc\/rpc.h>/<tirpc\/rpc\/rpc.h>/g' log_service.h
sed -i 's/#include <rpc\/pmap_clnt.h>/#include <rpc\/pmap_clnt.h>\n#include
<tirpc\/rpc\/rpc.h>/g' log_service_svc.c
sed -i 's/#include <rpc\/pmap_clnt.h>/#include <rpc\/pmap_clnt.h>\n#include
<tirpc\/rpc\/rpc.h>/g' log_service_clnt.c
```

Para el servidor RPC se compilan:

- log_service_impl.c: Servicio de logs
- log_service_svc.c: Código del servidor RPC
- log_service_xdr.c: Código para serialización/deserialización XDR

dejando como resultado el ejecutable log_server

```
gcc -Wall -g -I/usr/include/tirpc -Wno-unused-variable $(pkg-config --cflags
libtirpc) -o log_server log_service_impl.c log_service_svc.c
log_service_xdr.c $(pkg-config --libs libtirpc)
```

Para compilar el servidor principal lo mismo se hace con:

- server.c: Servidor principal
- log_client.c: Cliente RPC
- log_service_clnt.c: Código del cliente RPC
- log_service_xdr.c: Código para serialización/deserialización XDR

dejando como resultado el ejecutable server.

```
gcc -Wall -g -I/usr/include/tirpc -Wno-unused-variable $(pkg-config --cflags
```

```
libtirpc) -o server server.c log_client.c log_service_clnt.c log_service_xdr.c
$(pkg-config --libs libtirpc)
```

(Se utilizó "..usr/include/tirpc -Wno-unused-variable.." para ignorar un warning que aparecía en los ficheros generados automáticamente con rpcgen)

- Compilación

Para compilar todo el proyecto se escribe:

```
Unset
make all
```

Para eliminar los archivos generados durante la compilación:

```
Unset
make clean
```

3. Batería de pruebas

Ejecución Parte 1

En la terminal 1 ejecutamos el servidor:

```
Unset
./server -p 8888
```

Después, en la terminal 2 se corre el cliente:

```
Unset
python3 client.py -s localhost -p 8888
```

Se pueden abrir más terminales para probar condiciones de hilos cuando se ejecutan varios clientes.

Ejecución Parte 2

En la primera terminal iniciamos el servicio web:

```
Unset
python3 web_service.py
```

Abrimos otra terminal, pero ahora para ejecutar el servidor del sistema:

```
Unset
./server -p 8888
```

Se ejecutan múltiples instancias de cliente si se desea abriendo muchas terminales más con el siguiente comando:

```
Unset
python3 client.py -s localhost -p 8888
```

Ejecución Parte 3

Para realizar las pruebas correspondientes, primero hay que compilar el proyecto como se indicó anteriormente y ya luego ir añadiendo los siguientes comandos.

En una terminal se inicia el servicio web de esta manera:

```
Unset
python3 web_service.py
```

En otra terminal, se inicia el servidor RPC:

```
Unset
./log_server
```

En la tercera terminal se define la variable de entorno LOG_RPC_IP y ejecuta el servidor principal:

```
Unset
export LOG_RPC_IP=localhost
./server -p 8888
```

Por último, se pueden crear tantas terminales como se desee para ejecutar los clientes simulando concurrencia y verificando que el sistema responde correctamente a múltiples clientes a la vez:

```
Unset
python3 client.py -s localhost -p 8888
```

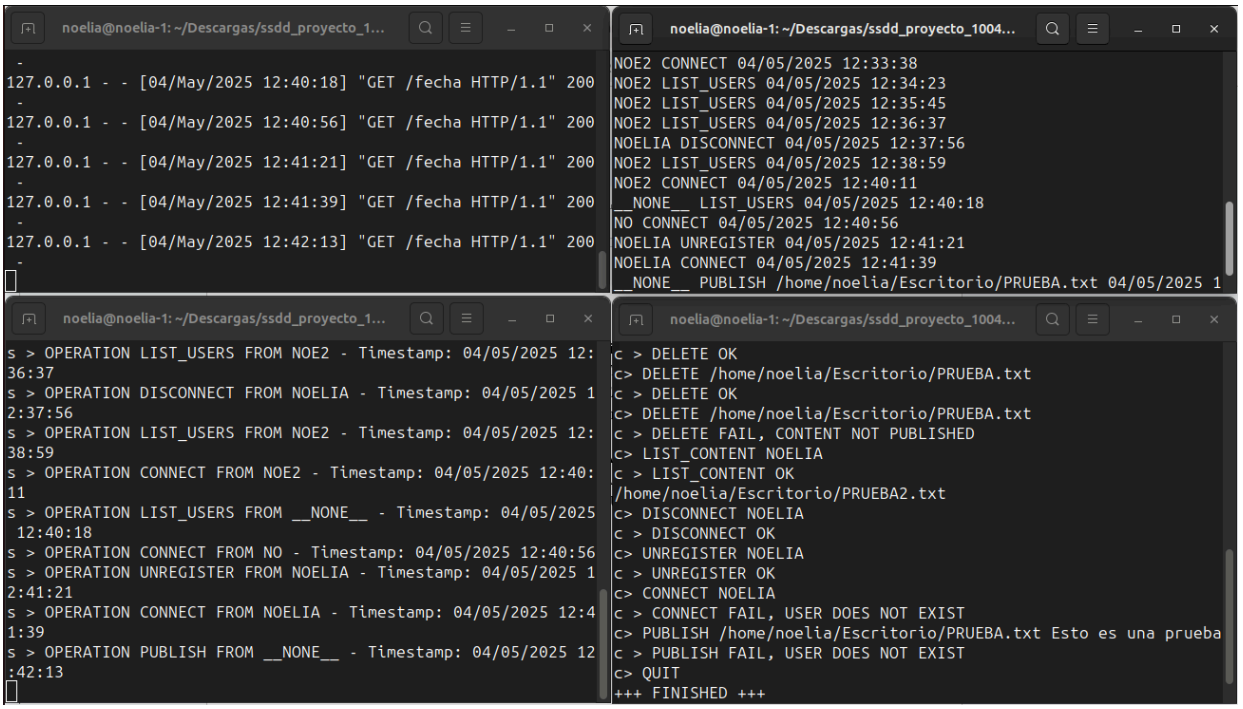
Pruebas Realizadas

Para comprobar que la práctica funciona adecuadamente hemos diseñado este conjunto de pruebas que se han probado para cada una de las tres partes de este proyecto. Las pruebas son las siguientes:

- Creación de un nuevo usuario **REGISTER < user_name >**
    - Resultado esperado:REGISTER OK
    - Resultado obtenido:REGISTER OK
- Creación del mismo usuario por segunda vez.
    - Resultado esperado:USERNAME IN USE
    - Resultado obtenido:USERNAME IN USE
- Conectarse al sistema con el usuario creado **CONNECT < user_name >**
    - Resultado esperado:CONNECT OK
    - Resultado obtenido:CONNECT OK
- Conectarse de nuevo con el mismo usuario
    - Resultado esperado:USER ALREADY CONNECTED
    - Resultado obtenido:USER ALREADY CONNECTED
- Publicación de contenido con ese usuario **PUBLISH < file_name > < description >**
    - Resultado esperado:PUBLISH OK
    - Resultado obtenido:PUBLISH OK
- Publicación del mismo contenido.
    - Resultado esperado: PUBLISH FAIL , CONTENT ALREADY PUBLISHED
    - Resultado obtenido:PUBLISH FAIL , CONTENT ALREADY PUBLISHED
- Publicación de un segundo archivo.
    - Resultado esperado:PUBLISH OK
    - Resultado obtenido:PUBLISH OK
- Listado del contenido del usuario **LIST_CONTENT < user_name >**
    - Resultado esperado: LIST_CONTENT OK
      file_name_1
      file_name_2
    - Resultado obtenido: LIST_CONTENT OK
      /home/noelia/Escritorio/PRUEBA.txt
      /home/noelia/Escritorio/PRUEBA.txt
      /home/noelia/Escritorio/PRUEBA2.txt
- Borrado del primer archivo publicado **DELETE < file_name >**
    - Resultado esperado:DELETE OK
    - Resultado obtenido:DELETE OK

- Borrado del mismo archivo publicado
    - Resultado esperado:DELETE FAIL, CONTENT NOT PUBLISHED
    - Resultado obtenido:DELETE FAIL, CONTENT NOT PUBLISHED
- Listado del contenido del usuario después del borrado de uno de los archivos.
    - Resultado esperado:LIST_CONTENT OK
      file_name_1
    - Resultado obtenido:LIST_CONTENT OK
      /home/noelia/Escritorio/PRUEBA2.txt
- Creación y conexión de un nuevo usuario para probar el listado de usuarios **LIST_USERS**
    - Resultado esperado: REGISTER OK & CONNECT OK
      LIST_USERS OK
      USER1 IP1 PORT1
      USER2 IP2 PORT2
    - Resultado obtenido: REGISTER OK & CONNECT OK
      LIST_USERS OK
      NOELIA 127.0.0.1 39653
      NOE2 127.0.0.1 55665
- Transferencia de archivo desde el segundo usuario conectado con el archivo del primero **GET_FILE < user_name > < remote_file_name > < local_file_name >**
    - Resultado esperado:GET_FILE OK
    - Resultado obtenido:GET_FILE OK
- Transferencia de un archivo que no exista
    - Resultado esperado:GET_FILE FAIL , FILE NOT EXIST
    - Resultado obtenido:GET_FILE FAIL , FILE NOT EXIST
- Desconexión del primer usuario **DISCONNECT < user_name >**
    - Resultado esperado:DISCONNECT OK
    - Resultado obtenido:DISCONNECT OK
- listado de usuarios después de desconectar uno de los usuarios **LIST_USERS**
    - Resultado esperado: LIST_USERS OK
      USER1 IP1 PORT1
    - Resultado obtenido: LIST_USERS OK
      NOE2 127.0.0.1 55665
- Conectarse al sistema con un usuario que no existe
    - Resultado esperado:CONNECT FAIL , USER DOES NOT EXIST
    - Resultado obtenido:CONNECT FAIL , USER DOES NOT EXIST
- Dar de baja a uno de los usuarios **UNREGISTER < user_name >**
    - Resultado esperado:UNREGISTER OK
    - Resultado obtenido:UNREGISTER OK

- Conectarse con ese usuario que ya no existe
    - Resultado esperado:CONNECT FAIL , USER DOES NOT EXIST
    - Resultado obtenido:CONNECT FAIL , USER DOES NOT EXIST
- Publicación de contenido sin estar conectado **PUBLISH < file_name > < description >**
    - Resultado esperado:PUBLISH FAIL , USER DOES NOT EXIST
    - Resultado obtenido:PUBLISH FAIL , USER DOES NOT EXIST
- Finalizar el intérprete con el comando **QUIT**



4. Conclusiones

La práctica ha sido bastante completa e intuitiva a diferencia de los ejercicios evaluables. Tal vez influye el hecho de haber tenido más tiempo para dedicarle, pero creemos que la arquitectura de conectar usuarios y que realicen operaciones entre ellos sin pedir operaciones complicadas de entender, han hecho más amena la práctica.

En principio, encontramos problemas con los métodos “get_file” y “delete” ya que no estábamos seguros de si se debía crear un fichero en local o que sólo apareciese en el servidor. También fue de bastante dificultad encontrar errores mínimos, por operar con archivos que se iban volviendo más grandes con el paso del tiempo (más de 600 líneas).

Pasado eso, las siguientes partes fueron mejores ya que la cantidad de líneas era menor, el sistema rpc ya lo habíamos practicado recientemente en los ejercicios evaluables y el servicio web no fue muy difícil de implementar a comparación de los archivos iniciales de la parte 1.