Universidad
Carlos III de Madrid

Universidad Carlos III de Madrid

Curso 2023/2024. Sistemas Distribuidos

Practica Final.

Fecha: 12/05/2024 Entrega: Final

Grupo: 84

Alumno 1:

Nombre y Apellidos: Fernando Consiglieri Alcántara

NIA: 100472111

Alumno 2:

Nombre y Apellidos: David Andrés Yáñez Martínez

NIA: 100451958

Índice

Diseño de los archivos

Los archivos se han organizado en 3 carpetas, detalladas a continuación:

Carpeta "server": Este componente se comunica mediante sockets con los clientes y mediante RPC con el “servidor_rpc”, actuando como cliente en este último caso. Los archivos en esta carpeta son:

- **servidor.c:** Se encarga de recibir las peticiones de los clientes a través del socket, invocar las funciones correspondientes, y devolver las respuestas adecuadas a los clientes.

- **list.c:** Contiene la implementación de las funciones requeridas en la práctica (backend). Además, es donde se desarrolla la lista enlazada, adaptada para este ejercicio y utilizada para almacenar los mensajes proporcionados por los clientes.

- **lines.c:** Incluye tres métodos proporcionados en clase que facilitan la lectura y escritura en el socket.

Carpeta "server_rpc": Este servidor RPC recibe datos del servidor y los imprime al estilo "echo". Los archivos en esta carpeta son:

- **mensaje.x:** A partir de este archivo, se generan los siguientes archivos, que son fundamentales para la comunicación entre el cliente y el servidor:

- **mensaje_clnt.c:** Contiene las funciones de bajo nivel necesarias para invocar las funciones RPC en el servidor_rpc.

- **mensaje_svc.c:** Implementa las funciones del servidor RPC.

- **mensaje_xdr.c:** Contiene las rutinas de codificación y decodificación de datos necesarias para la comunicación entre el cliente y el servidor.

- **Server_rpc.c:** Es el servidor RPC utilizado en prácticas anteriores y se integra con los otros componentes del sistema, actuando como punto de entrada del servidor RPC.

- **Mensaje.h:** Es la cabecera que utilizan tanto el cliente (server) como el servidor (servidor_rpc) para llamar a todas las funcione.

Parte del "cliente":

- **client.py:** Constituye el cliente que se encargará de hacer las llamadas al servidor principal. Desde el cliente ocurren todas las llamadas incluidas en las especificaciones de la práctica.

- **ws-time-service.py:** Incluye el desarrollo del servicio web que se encarga, antes de cada llamada al servidor, de devolver la hora y fecha actual.

- **client_tests_1.py:** Contiene la primera parte de los unitest utilizados para la batería de pruebas.

- **client_tests_2.py:** Contiene la segunda parte de los unitest utilizados para la batería de pruebas.

Protocolo de comunicación

El protocolo de comunicación varía según la parte del proyecto en la que te encuentres. Se dividen en tres partes:

Protocolo de comunicación RPC:

El protocolo de comunicación implementado para la aplicación se basa en Remote Procedure Call (RPC). Esto permite una comunicación más estructurada y simplificada entre el cliente y el servidor. A continuación, se describe el procedimiento que conforma este protocolo:

Servicios de RPC:

En el protocolo de comunicación RPC se definen los servicios que el cliente puede invocar en el servidor, esto se hará de manera remota como si estuviera llamando a funciones locales. Estos servicios se definen dentro de un programa y una versión específica del programa.

En este caso:
- El programa se denomina "LIST_SERVICE".
- Se define una versión del programa llamada "LIST_SERVICE_V1".
- Dentro de esta versión, se declara un servicio llamado "print".

El servicio "print" tiene los siguientes parámetros:
- **username**: Una cadena de texto que representa el nombre de usuario.
- **operation**: Una cadena de texto que describe la operación realizada.
- **date**: Una cadena de texto que indica la fecha en que se realizó la operación.
- **file**: Una cadena de texto que especifica el archivo sobre el cual se realizó la operación. Puede ser opcional.

Este servicio "print" devuelve un entero (**int**), que podría ser utilizado para indicar el éxito o el fracaso de la operación pero que no se usa.

Implementación
Con el uso de RPC, la comunicación entre el cliente y el servidor se vuelve más clara y estructurada. El cliente puede llamar a los servicios definidos en el servidor como si fueran funciones locales, y el servidor puede responder con datos estructurados. Esto

facilita la interoperabilidad y simplifica la implementación del protocolo, permitiendo que el cliente y el servidor se comuniquen de manera más eficiente y con menos errores de interpretación.

Protocolo de comunicación SOCKET:

Este protocolo de comunicación se aplica entre el servidor y el cliente de python en cada solicitud. Aquí está la descripción del protocolo:

Parte del cliente:
El cliente funciona a través de una función que hace de shell y recibe los comandos escritos por consola. En primer lugar, el cliente recibe como parámetros el puerto y la ip a la que está asociado el servidor.
Todas las llamadas a funciones siguen un protocolo similar:

- El cliente se conecta al socket del servidor a través del método “connect_socket”.
- Genera un mensaje según los requerimientos especificados en el enunciado.
- Envía el mensaje al servidor.
- Según el resultado, imprimirá y devolverá un mensaje diferente.
- Independientemente del resultado o de si ha habido algún error, cierra el socket.

A continuación, explicaré las funciones que siguen un protocolo diferente:

connect: antes de enviar el mensaje necesita escoger un puerto para enviarlo en el mensaje, por lo que tendrá que crearlo y hacer un bind. Una vez el servidor ha aceptado la conexión, se lanza un hilo al método “listen_requests” con el socket creado como argumento. En ese momento, el cliente también pasa a ser servidor.

listen_requests: este método tras hacer un listen del socket que ha recibido, empieza un bucle que se detiene a través de una flag. Cuando reciba un mensaje comprobará primero si la operación es “GET_FILE” y el nombre del archivo. En caso de poder leerlo, envía su contenido de vuelta por el socket, tras enviar el mensaje cerrará el socket creado y cuando se cierre el hilo cerrará el socket que recibió como parámetro. El hilo se cerrará con ayuda de la flag, esta se llamará cuando hagas disconnect, cuando se haga un quit o cuando se cierre el programa por interrupción del teclado.

list_users: Además de tratar el mensaje con el resultado de la operación, recibe la lista de usuarios que tiene que encargarse de imprimir.

list_content: Funciona de forma parecida al anterior, imprimiendo esta vez el contenido del usuario que has pedido.

get_file: Tiene que hacer una primera llamada al servidor para recibir, con la lista de usuarios, la ip y puerto del cliente del que quiere recibir el archivo, ahora se necesita un nuevo socket para el otro cliente. Le envía el nombre del archivo que quiere copiar y si no hay ningún problema escribirá en el archivo pasado por parámetros el contenido recibido. Finalmente, cerrará los dos sockets que ha abierto.

Parte del servidor:
El servidor crea un socket, lo bindea al puerto que se le especifica cuando se llama al binario y mas tarde se queda en un bucle while infinita hasta que alguien le manda una petición y con cada petición crea un hilo. Dentro de cada hilo se realizan los siguientes pasos:

- Se lee el mensaje enviado por el cliente a través del socket.
- Se verifica el tipo de operación solicitada por el cliente.
- Dependiendo del tipo de operación, se realiza un conjunto específico de acciones.
- Para cada tipo de operación, se espera recibir ciertos parámetros del cliente, como el nombre de usuario, la fecha, el nombre del archivo, etc.
- Se realiza una llamada a funciones de backend para ejecutar la operación solicitada (registro de usuario, eliminación de archivo, etc.).
- Se llama a un procedimiento remoto (RPC) para informar sobre la operación realizada.
- Se envía un código de error de vuelta al cliente a través del socket para informar sobre el resultado de la operación.
- Si se produce un error durante el procesamiento de la operación, se cierra la conexión con el cliente y se finaliza el hilo.
- Se libera la memoria utilizada para almacenar los parámetros de la operación.

En resumen, el protocolo de comunicación garantiza la consistencia en el formato de los datos enviados y recibidos, lo que facilita su interpretación tanto en el cliente como en el servidor. Todos los envíos de información se realizan con sockets, mediante el uso de las funciones readLine, sendMessage y recvMessage.

Protocolo de comunicación WEB SERVICE:

El archivo proporciona un servicio web simple que utiliza el protocolo SOAP (Simple Object Access Protocol) para ofrecer la hora actual a través de la web. Este servicio se implementa en Python utilizando la biblioteca Spyne. La hora actual se obtiene utilizando el método get_time, que devuelve la hora en un formato de cadena.

- Se define un servicio llamado TimeGetter que proporciona un único método llamado get_time. Este método devuelve la hora actual en formato de cadena.
- Se configura el servicio especificando el tipo de datos que recibe y devuelve, así como el protocolo de comunicación que utilizará.
- Se crea un servidor WSGI (Web Server Gateway Interface) utilizando la biblioteca wsgiref en Python. Este servidor actúa como punto de entrada para las solicitudes SOAP entrantes.
- El servidor se inicia y comienza a escuchar las solicitudes entrantes en la dirección http://127.0.0.1:8000.
- Para consumir el servicio desde otro método o aplicación, se genera una URL WSDL (http://localhost:8000/?wsdl). Esta URL proporciona una descripción detallada del servicio, incluidos los métodos disponibles y los tipos de datos esperados.
- Se utiliza la biblioteca zeep para crear un cliente SOAP que se conecta al servicio web utilizando la URL del WSDL generada anteriormente.
- Se llama al método get_time() del cliente SOAP creado para obtener la hora actual del servicio web.

Forma de compilación

La compilación de nuestro proyecto es sencilla, en caso del código del servidor se realiza mediante make. Explicado a continuación:

- **Variables:** Se establecen variables para el compilador (CC) y los flags de compilación (CFLAGS).
- **SERVIDOR y CLIENTE DE RPC:** Se definen las fuentes y objetos del servidor y el cliente de RPC.
- **SERVIDOR RPC:** Se definen las fuentes y objetos del servidor RPC.
- **all:** Es la regla principal que compila tanto el servidor como el servidor RPC.
- **Compilación de objetos del cliente de RPC:** Se define una regla para compilar los objetos del cliente de RPC.
- **Compilación del servidor:** Se define una regla para compilar el servidor, utilizando los objetos del servidor y del cliente de RPC.
- **Compilación de objetos del servidor RPC:** Se define una regla para compilar los objetos del servidor RPC.
- **Compilación del servidor RPC:** Se define una regla para compilar el servidor RPC, utilizando los objetos del servidor RPC.
- **runs y runrpc:** Son comandos para ejecutar el servidor y el servidor RPC, respectivamente.
- **clean y fclean:** Son reglas para limpiar los archivos generados durante la compilación, con fclean eliminando también los binarios.
- **re:** Es una regla para reconstruir todo el proyecto desde cero.

Todos los archivos .c tienen flags comunes como -Wall -Werror para garantizar un código limpio. Además, se añaden las flags -lpthread o -lrt respectivamente.

Ejecución

La ejecución del programa se divide en cuatro partes. Primero, se abren cuatro terminales.

En la primera terminal, se ejecuta el servidor RPC con el comando "./servidor_rpc". Aquí se imprimirán todas las peticiones que lleguen al servidor.

En la siguiente terminal, se ejecuta el servicio web, por ejemplo, "python3 ws-time-service.py". Los mensajes impresos en esta terminal son irrelevantes para el funcionamiento del programa.
Luego, se abre una tercera terminal y se inicia el servidor, especificando el puerto deseado, por ejemplo, "-p 4242". La sintaxis sería "./servidor -p 4242". En esta terminal se imprimirá la dirección IP del servidor, la cual se necesitará para la configuración del cliente.
Finalmente, en la cuarta terminal, se ejecuta el cliente con el comando "python3 client.py -s 172.22.219.110 -p 4242" para este ejemplo. Una vez que todo esté configurado en el cliente, se pueden comenzar a enviar comandos.

Decisiones de Diseño

Guiados por Elías, hemos decidido incluir este apartado debido a las dudas surgidas durante la ejecución del proyecto, las cuales no fueron contempladas en el PDF inicial de la práctica. Esto nos enfrenta a nuevos desafíos, y aquí discutiremos cómo los hemos abordado:

1. En el caso de que dos usuarios intenten registrarse y conectarse simultáneamente, el resultado en la consola sería el siguiente:

    c> register fer REGISTER OK

    c> register carlos REGISTER OK

    c> connect fer CONNECT OK

    c> connect carlos USER ALREADY CONNECTED

La indicación de "USER ALREADY CONNECTED" no se debe a que Carlos ya esté conectado, sino porque Fer ya lo está, por lo que debería desconectarse para permitir que Carlos se conecte.

2. Si un usuario se registra y se conecta, pero luego intenta conectar a un usuario que no existe:

    c> register fer REGISTER OK

    c> connect fer CONNECT OK

    c> connect david USER ALREADY CONNECTED

El resultado es similar al caso anterior.

3. Intentar realizar un listado de contenido (list_content) sin estar conectado:

    c> list_content fer LIST_CONTENT FAIL, USER NOT CONNECTED

Este caso no tiene sentido, ya que no hay ningún usuario conectado en este cliente, independientemente de si Fer está conectado en otra terminal o no.

4. Intentar realizar un listado de usuarios (list_users) sin estar conectado:

   c> list_users fer LIST_USERS FAIL, USER NOT CONNECTED

De manera similar al caso anterior, en este cliente no hay ningún usuario conectado, por lo que no se puede hacer uso de la función list_users.

5. Intentar desconectar a un usuario desde otro diferente. No hay ninguna especificación clara respecto a esto ya que no podemos cerrar el hilo abierto por el connect desde un cliente diferente por lo que hemos mantenido el posible comportamiento anómalo que puede tener el programa si se intenta hacer eso.

6. Intentar hacer unregister de un cliente conectado. De la misma manera que el anterior puede tener algún comportamiento extraño.

7. El programa provocará un shutdown del socket del hilo cuando ocurre:

   - Quit del programa
   - Disconnect
   - Interrupción de teclado

Batería de pruebas

La batería de pruebas funciona a través de dos archivos, client_tests_1.py y client_tests_2.py.

Hay un total de 25 pruebas y es necesario modificar en las primeras líneas el client._port y el client._server con los valores adecuados al servidor para que funciones. Esto es debido a que los tests están hechos para ejecutarlos directamente por lo que no hay una forma sencilla de pasar argumentos.

Las pruebas constituyen la mayoría de los casos de uso tantos correctos como incorrectos para probar casi todas las salidas posibles (por lo menos las que se pueden forzar).

CLIENT_TESTS_1.PY		
NOMBRE DE LA PRUEBA	**DESCRIPCIÓN**	**SALIDA ESPERADA**
test_a_register_success	Test básico que prueba a registrar a un usuario	client.RC.OK
test_b_register_repeated_failure	Test para comprobar que si el usuario está registrado da error	client.RC.ERROR
test_c_register_exceed_bytes_failure	Test para comprobar que si el nombre de usuario es demasiado largo da error	client.RC.USER_ERROR

test_d_unregister_success	Test para comprobar que funciona el darse de baja	client.RC.OK
test_e_unregister_ nonexistent_user_failure	Test para comprobar que no se puede dar de baja un usuario que no existe	client.RC.ERROR
test_f_connect_success	Test para comprobar que funciona la conexion	client.RC.OK
test_g_connect_ nonexistent_user_failure	Test para comprobar que da error cuando intentas conectar a un usuario no registrado	client.RC.ERROR
test_h_connect_already_ connected_user_failure	Test para comprobar que da error cuando intentas conectar a un usuario ya conectado	client.RC.USER_ERROR
test_i_publish_success	Test para comprobar que funciona la publicación de archivos	client.RC.OK
test_j_publish_exceed_ bytes_failure	Test para comprobar que da error cuando intentas publicar un archivo con un nombre muy grande	client.RC. ANOTHER_CASES
test_k_publish_not _register_failure	Test para comprobar que da error cuando intentas publicar un archivo estando el usuario sin registrar (Esta prueba está comentada porque, aunque funcione, genera muchos problemas mantener el cliente sin desconectar)	client.RC.ERROR
test_l_publish_not _connected_failure	Test para comprobar que da error cuando intentas publicar un archivo estando el usuario sin conectar	client.RC.USER_ERROR

CLIENT_TESTS_2.PY		
NOMBRE DE LA PRUEBA	**DESCRIPCIÓN**	**SALIDA ESPERADA**
test_m_already _published_failure	Test para comprobar que da error cuando intentas publicar un archivo ya publicado	client.RC.OTHER_CASES

test_n_delete_success	Test para comprobar que funciona la eliminación de archivos publicados	client.RC.OK
test_o_delete_not _register_failure	Test para comprobar que da error cuando intentas eliminar un archivo publicado estando el usuario sin registrar (Esta prueba está comentada porque, aunque funcione, genera muchos problemas mantener el cliente sin desconectar)	client.RC.ERROR
test_p_delete_not _connected_failure	Test para comprobar que da error cuando intentas eliminar un archivo publicado estando el usuario sin conectar	client.RC.USER_ERROR
test_q_delete_not _exists_failure	Test para comprobar que de error cuando intentas eliminar un archivo que no ha sido publicado	client.RC.OTHER_CASES
test_r_list_users _success	Test para comprobar que funciona la lista de usuarios conectados	client.RC.OK
test_s_list_users_not _connected_failure	Test para comprobar que da error cuando intentas ver la lista de usuarios sin estar conectado	client.RC.USER_ERROR
test_t_list_content _success	Test para comprobar que funciona la lista de contenido de un usuario	client.RC.OK
test_u_list_content_not _connected_failure	Test para comprobar que da error la lista de contenido cuando el usuario no está conectado	client.RC.USER_ERROR
test_v_list_content_not _exists_failure	Test para comprobar que de error la lista de contenido cuando el usuario remoto no existe	client.RC.OTHER_CASES
test_w_get_file_success	Test para comprobar que funciona get file	client.RC.OK
test_x_get_file_not _connected_failure	Test para comprobar que da error el get file cuando el usuario no está conectado	client.RC.USER_ERROR
test_y_get_file_not _exists_failure	Test para comprobar que funciona get file	client.RC.ERROR