Universidad Carlos III
Sistemas Distribuidos
2023-24

Práctica final

Fecha: **12/05/24**

NIA: **100472206 / 100472252** Grupo: **81 - 01**

Grado: **Grado en Ingeniería Informática**

Alumnos: **Marcos González Vallejo/Javier Pallarés de Bonrostro**

Correo electrónico: **100472206@alumnos.uc3m.es/100472252@alumnos.uc3m.es**

Tabla de contenido

1. Introducción

En esta práctica, se creará el prototipo de una aplicación que permita la transmisión de ficheros entre usuarios (peer-to-peer) y un servidor central que informe a los usuarios sobre la disponibilidad de estos archivos, los usuarios disponibles…
Además, se desarrollará un servidor RPC que construya un "log" en la terminal de todas las operaciones que recibe el servidor central.
Finalmente, se diseñará un servicio web en Python que permita recuperar la hora actual, de manera que en el "log" anterior, también se imprima la hora de cada operación.

Creemos interesante comentar que hemos creado un **repositorio en GitHub** con el proyecto entero, por si pudiera facilitar la consulta del código y ver la evolución de la aplicación

2. Descripción del código

En cuanto al código, tenemos varios ficheros principales:

- ***client.py***
  En este fichero, encontramos todas las funciones a las que el cliente llama, register, connect, publish… al igual que la creación de un hilo al hacer connect para escuchar peticiones de otros clientes.
  En cuanto al funcionamiento principal, para cada función, lo primero que hacemos es conectarnos al servidor, conexión que se cerrará cuando el servidor le mande al cliente la respuesta de la función.
  Después, mandamos el código de operación (una string con el nombre de la operación). En este momento, es cuando desde el cliente llamamos a una función llamada get_current_timestamp la cual mandará una petición al servidor web que hemos levantado localmente para recibir la fecha y hora actual. Tras obtener la fecha, la mandamos al servidor. A partir de aquí, para cada función, se mandarán los mensajes correspondientes, por ejemplo, para register, además de lo anterior, solo mandaremos el username, para connect, en cambio, además tendremos que mandar al servidor el puerto en el que está escuchando el cliente.
  Por último, tras mandar los mensajes al servidor, el cliente espera por el resultado y ese resultado es evaluado imprimiendo por pantalla el mensaje de error correspondiente dependiendo del código de respuesta que se reciba del servidor.
  Además, al hacer connect, se "comunica" al server que el cliente emisor está escuchando peticiones de otros clientes. Además, debido a que cada cliente lanzado representa una persona distinta, se guarda el username en local.
  Por otro lado, en disconnect, ponemos a falso la variable booleana que mantenía "vivo" al hilo con el que el cliente escuchaba conexiones de otros clientes.
  Al conectarse, crea un hilo que escucha y atiende peticiones de otros clientes para suministrarles los ficheros que necesiten. No se incluye un mutex ya que varios hilos pueden leer los mismos ficheros a la vez (no hay modificaciones). Se ha adaptado la función que se usa para leer cadenas de manera que no codifique los bytes recibidos. De esta forma, se pueden enviar archivos de cualquier tipo y no solo binarios.

- ***servidor.c***
  En cuanto al servidor “inicial”, se inicia creando el socket de escucha de peticiones usando el puerto que pasamos por línea de comandos. Después se mete en un bucle aceptando peticiones en un socket cliente sc. Tras aceptar una petición, en ese sc, obtendrá información sobre el cliente que se ha conectado y creará un hilo para él, que ejecutará la función tratar_petición y le mandará como parámetro ese sc. Dentro de tratar petición, copiará localmente el sc para evitar que otro hilo sobreescriba el mismo espacio de memoria y tras obtener su copia de sc, empezará a tratar los mensajes que reciba del cliente.
  Entre los mensajes generales, encontramos la obtención del código de operación y del string con la fecha y hora que el cliente obtuvo del servidor web. Tras esto, dependiendo de cual sea la operación a realizar, el servidor recibirá el número de mensajes necesarios desde el servidor. Es interesante comentar que, haya o no un error a la hora de manejar los ficheros necesarios de cada operación, se llamará a la función de impresión rpc con una string que contenga usuario, operación, fecha y hora. Aquí, el servidor.c hace de cliente rpc y manda esta string al servidor rpc el cual es el que imprime en consola esta string.
  Por último, tras llevar a cabo la operación recibida, mandará el resultado de la operación y cerrará el socket del cliente. Hay operaciones que tras mandar el resultado, también tendrán que mandar otros datos, como el número de contenidos para cierto cliente ya que el cliente, tendrá que saber cuantos contenidos esperar del servidor. Estas son las operaciones en las que el cliente no solo recibe el resultado, sino que también recibe otros elementos.

- ***servidor_handle.c***
  Este fichero, contiene todo el código relacionado con el manejo de ficheros e información que llegue del cliente.
- ***servidor_rpc_server.c***
  Este fichero se encarga de imprimir, en la terminal en la que esté levantado, la string compuesta por username, operación, fecha y hora que se manda desde el servidor.c.
- ***web_server.py***
  Finalmente, este fichero es el encargado de levantar localmente un servidor web usando el método POST.

2. Guía de uso

Para poder ejecutar la aplicación sin problemas debemos seguir los siguientes sencillos pasos:

1. Para compilar debemos ejecutar ambos Makefile, es decir, en la terminal se ejecutará:
    a. make (para compilar el cliente y el servidor normal)
    b. make -f Makefile.servidor_rpc (para compilar lo relativo al servidor RPC)
2. Después, para que el servicio RPC funcione correctamente debemos ejecutar:
    a. sudo rpcbind start
3. Para que el servicio web funcione correctamente, es necesario instalar los paquetes:
    a. spyne (para el servidor web)
    b. zeep (para el cliente)
4. Además, es **requisito obligatorio,** que el cliente disponga de un directorio en el mismo nivel que el ejecutable "client.py" con el nombre **./local_storage/<nombre_usuario>**. Es en esta carpeta donde deben existir los ficheros que quiera publicar.
5. Finalmente debemos levantar todos los servidores necesarios:
    a. ./server -p <puerto_server_central>
    b. ./server_rpc
    c. python3 web_server.py
    d. python3 client.py -s <address> -p <puerto_server_central>
6. Creemos importante destacar que los archivos deben crearse de forma manual. Es decir, si "usuario1" publica "fichero1", éste debe ser credo manualmente en ./local_storage/usuario1 ya que si no devolverá error la operación GET_FILE de ese archivo.

3. Batería de pruebas

Hemos realizado una extensa batería de pruebas que verifican el correcto funcionamiento tanto de casos específicos como de volumen de conexiones. Para ejecutar todas las pruebas se emplea un shell script que las ejecuta una tras otra “all_tests.sh”. No se comprueba que cada operación reciba el número de parámetros correctos ya que esa función la realiza la shell del cliente por defecto.

1. Caso correcto (Volumen

Se prueba la conexión y envío de ficheros entre 300 clientes y se verifica que la copia de archivos y las operaciones relacionadas con el servidor central, servicio web y RPC son correctas.

2. Argumentos incorrectos

Al recibir el cliente los argumentos incorrectos devuelve error sin iniciar la aplicación

```
Running tests in ./tests/t1-wrong_args.py...
usage: client.py [-h] -s S -p P
client.py: error: the following arguments are required: -s, -p
```

3. Dirección de conexión errónea en el cliente

Es similar a un servidor caído, cuando ejecuta una operación, ésta falla.

```
Running tests in ./tests/t2-wrong_address.py...
REGISTER usuario1
c> REGISTER FAIL
```

4. Operación Register

a. Usuario ya registrado

```
Running tests in ./tests/t3-register1-already_registered.py...
REGISTER usuario1
c> REGISTER OK
REGISTER usuario1
c> USERNAME IN USE
```

b. Servidor caído

Devuelve *REGISTER FAIL* al no poder ejecutar la operación.

5. Operación Unregister

a. Usuario no registrado

```
Running tests in ./tests/t5-unregister1-user_does_not_exist.py...
UNREGISTER usuario1
c> USER DOES NOT EXIST
```

b. Servidor caído

Devuelve *UNREGISTER FAIL* al no poder ejecutar la operación.

6. Operación Connect

a. Usuario no registrado

```
Running tests in ./tests/t7-connect1-user_not_registered.py...
CONNECT usuario1
c> CONNECT FAIL, USER DOES NOT EXIST
```

b. Usuario ya conectado

```
Running tests in ./tests/t8-connect2-already_connected.py...
REGISTER usuario1
c> REGISTER OK
CONNECT usuario1
c> CONNECT OK
CONNECT usuario1
c> USER ALREADY CONNECTED
```

c. Servidor caído

Devuelve *CONNECT FAIL* al no poder ejecutar la operación.

7. Operación Publish

a. Usuario no registrado

```
Running tests in ./tests/t10-publish1-user_not_registered.py...
PUBLISH title1 descripcion muy muy interesante
c> PUBLISH FAIL USER, DOES NOT EXIST
```

b. Usuario no conectado

```
REGISTER usuario1
c> REGISTER OK
CONNECT usuario1
c> CONNECT OK
DISCONNECT usuario1
c> DISCONNECT OK
PUBLISH title1 una descripcion muy interesante
c> PUBLISH FAIL, USER NOT CONNECTED
```

c. Contenido ya publicado

```
Running tests in ./tests/t12-publish3-already_published.py...
REGISTER usuario1
c> REGISTER OK
CONNECT usuario1
c> CONNECT OK
PUBLISH title1 una descripcion muy interesante
c> PUBLISH OK
PUBLISH title1 de nuevo una descripcion muy muy interesante
c> PUBLISH FAIL, CONTENT ALREADY PUBLISHED
```

d. Servidor caído

Devuelve *PUBLISH FAIL* al no poder ejecutar la operación.

8. Operación Delete

a. Usuario no registrado

```
Running tests in ./tests/t14-delete1-user_not_registered.py...
DELETE title1
c> DELETE FAIL, USER DOES NOT EXITS
```

b. Usuario no conectado

```
Running tests in ./tests/t15-delete2-user_not_connected.py...
REGISTER usuario1
c> REGISTER OK
CONNECT usuario1
c> CONNECT OK
DISCONNECT usuario1
c> DISCONNECT OK
DELETE title1
c> DELETE FAIL, USER NOT CONNECTED
```

c. Contenido no publicado

```
Running tests in ./tests/t16-delete3-not_published.py..
REGISTER usuario1
c> REGISTER OK
CONNECT usuario1
c> CONNECT OK
DELETE title1
c> DELETE FAIL, CONTENT NOT PUBLISHED
```

D. Servidor caído

Devuelve *CONNECT FAIL* al no poder ejecutar la operación.

9. Operación List Users

A. Usuario no registrado

```
Running tests in ./tests/t18-list_users1-user_not_registered.py...
LIST_USERS
c> LIST_USERS FAIL, USER DOES NOT EXIST
```

B. Usuario no conectado

```
Running tests in ./tests/t19-list_users2-user_not_connected.py...
REGISTER usuario1
c> REGISTER OK
CONNECT usuario1
c> CONNECT OK
DISCONNECT usuario1
c> DISCONNECT OK
LIST_USERS
c> LIST_USERS FAIL, USER NOT CONNECTED
```

C. Servidor caído

Devuelve *LIST_USERS FAIL* al no poder ejecutar la operación.

10. Operación List Content

A. Usuario no registrado

```
Running tests in ./tests/t20-list_users3-server_down.py...
LIST_USERS
c> LIST_USERS FAIL, USER DOES NOT EXIST
```

B. Usuario no conectado

```
Running tests in ./tests/t22-list_content2-user_not_connected.py...
REGISTER usuario1
c> REGISTER OK
CONNECT usuario1
c> CONNECT OK
DISCONNECT usuario1
c> DISCONNECT OK
LIST_CONTENT usuario1
c> LIST_CONTENT FAIL, USER NOT CONNECTED
```

c. Propietario del contenido no registrado

```
REGISTER usuario1
c> REGISTER OK
CONNECT usuario1
c> CONNECT OK
LIST_CONTENT usuario2
c> LIST_CONTENT FAIL, REMOTE USER DOES NOT EXIST
```

d. Servidor caído

Devuelve *LIST_CONTENT FAIL* al no poder ejecutar la operación.

e. Propietario del contenido sin contenido

```
Running tests in ./tests/t25-list_content5-user_without_content.py...
REGISTER usuario1
c> REGISTER OK
CONNECT usuario1
c> CONNECT OK
LIST_CONTENT usuario1
c> LIST_CONTENT OK
--------------
n_files: 0
```

f. Propietario del contenido con múltiples ficheros

```
LIST_CONTENT usuario1
c> LIST_CONTENT OK
n_files: 6
title6 descr6

title3 descr3

title4 descr4

title5 descr5

title2 descr2

title1 descr1
```

11. Operación Disconnect

a. Usuario no registrado

```
Running tests in ./tests/t27-disconnect1-user_not_registered.py...
DISCONNECT usuario2
c> DISCONNECT FAIL / USER DOES NOT EXIST
```

b. Usuario no conectado

```
REGISTER usuario1
c> REGISTER OK
CONNECT usuario1
c> CONNECT OK
DISCONNECT usuario1
c> DISCONNECT OK
DISCONNECT usuario1
c> DISCONNECT FAIL / USER NOT CONNECTED
QUIT
```

c. Servidor caído

Devuelve *DISCONNECT FAIL* al no poder ejecutar la operación.

12. Operación Get File

a. Propietario del contenido desconectado

```
DISCONNECT usuario1
c> DISCONNECT OK
GET_FILE usuario1 title1 title1copia
c> GET_FILE FAIL
```

b. Fichero inexistente

```
GET_FILE usuario1 title300 title1copia
c> GET_FILE FAIL / FILE NOT EXIST
```

c. Ejecutado tras List Users

```
LIST_USERS
c> LIST_USERS OK
n_connections: 2
usuario2 127.0.0.1 49579
usuario1 127.0.0.1 58467
PUBLISH title2 description1
c> PUBLISH OK
GET_FILE usuario1 title1 title1copia
c> GET_FILE OK
```

d. Ejecutado antes de List Users

```
GET_FILE usuario1 title1 title1copia
c> GET_FILE OK
```

Sin haber ejecutado antes List Users, le solicita al servidor central la información que necesita del propietario del fichero.