uc3m | Universidad Carlos III de Madrid

Peer-to-Peer

Sistemas Distribuidos

Fecha 12/05/2024

Curso III

Cuatrimestre 2º

Año 2023-2024

Titulación

Grado en Ingeniería Informática

Profesor

David Expósito Singh – **dexposit@inf.uc3m.es**

Alumno

Álvaro Obies García

100472136 – **100472136@alumnos.uc3m.es**

Santiago Kiril Cenkov Stoyanov

100472051 – **100472051@alumnos.uc3m.es**

Contenidos

Arquitectura

Se utiliza una arquitectura cliente-servidor. Por un lado entre cada proceso cliente y el proceso servidor para la petición de un servicio (Figura 1), y por otro lado entre los clientes para la transmisión de ficheros (Figura 2). En este último caso actuaría como cliente quien solicita un fichero y como servidor quien lo devuelve.

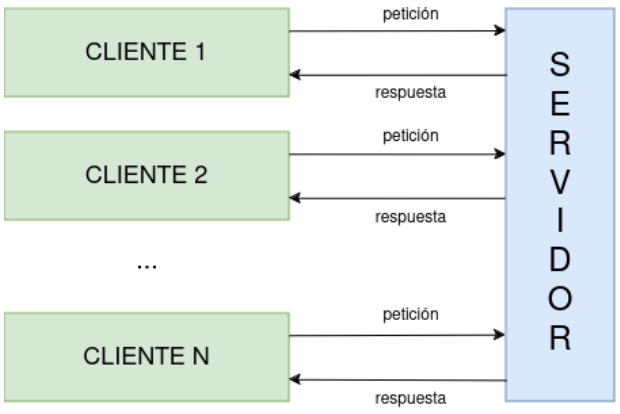

Figura 1: Arquitectura del sistema *peer-to-peer* entre cliente y servidor.

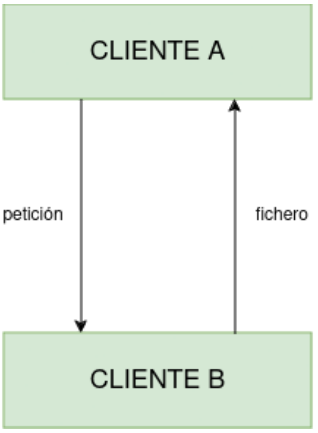

Figura 2: Arquitectura del sistema *peer-to-peer* entre clientes.

Por otro lado, se establece una comunicación entre cada cliente con un servidor de *web services* justo antes de enviar una petición al servidor central. También existe una comunicación entre el servidor central y un servidor de RPC. La arquitectura de ambos es igual que la de la FIgura 1.

Servicios

Los servidores siempre ofrecen servicios y los clientes pueden o no ofrecer servicios.

El servidor *web* ofrece a los clientes:

- **`datetime`**: obtiene la fecha y hora en el momento del procesado de la petición.
    - Se ejecuta cada vez antes de enviar una petición al servidor central.

El servidor central ofrece a los clientes:

- **`register`**: registrar a un usuario en el sistema.
    - Al completarse la operación correctamente, el usuario actuará con el nombre de usuario con el que se ha registrado.
- **`unregister`**: eliminar el registro de un usuario en el sistema.
    - El usuario se desvincula de su antiguo nombre de usuario.
- **`connect`**: conecta a un usuario en el sistema.
    - Para completarse correctamente, el usuario debe estar registrado y no estar conectado.
    - Asigna los recursos necesarios para recibir peticiones de otros usuarios.
- **`disconnect`**: desconecta a un usuario del sistema.
    - Para completarse correctamente, el usuario debe estar registrado y estar conectado.
    - Libera los recursos utilizados para recibir peticiones de otros usuarios.
- **`publish`**: publica un fichero para ser usado en *peer-to-peer*.
    - Para completarse correctamente, el usuario debe estar registrado, estar conectado y el fichero no publicado.
- **`delete`**: elimina un fichero publicado.
    - Para completarse correctamente, el usuario debe estar registrado, estar conectado y el fichero publicado.
- **`listusers`**: lista los usuarios y la información para conectarse a ellos

peer-to-peer.

  - Para completarse correctamente, el usuario debe estar registrado y estar conectado.

- **`listcontent`**: lista los ficheros publicados de usuarios.
  - Para completarse correctamente, el usuario debe estar registrado, estar conectado y los usuarios remotos registrados.

Algunos clientes ofrecen a otros clientes:

- **`getfile`**: transmitir un fichero entre clientes usado *peer-to-peer*.
  - Solo se ofrece por clientes conectados.
  - Para completarse correctamente, el fichero debe existir y ser uno de los ficheros publicados por el usuario remoto.

Datos

Se han establecido límites máximos fijos a algunos de los datos en los mensajes por la simplicidad para codificar el protocolo de comunicación. Estos datos son:

- **`status`**: estado de la ejecución de un servicio por el servidor central.
  - `character (1 byte).`
  - Permite representar desde 10 estados de ejecución (del 0 al 9).
- **`number_users, number_files`**: número de usuarios conectados y número de ficheros publicados respectivamente.
  - `32-bit integer (4 bytes).`
  - Con 4 bytes se pueden representar más de 2 mil millones de usuarios y ficheros.
- **`username, filename, description`**: nombre de usuario, nombre de fichero y descripción de fichero respectivamente.
  - `string[256] (256 bytes).`
  - Con 256 caracteres se puede representar estos 3 datos cómodamente. En esta cadena de caracteres se incluye como último carácter el `\0`.
- **`ip_address`**: dirección IPv4.
  - `string[16] (16 bytes).`
  - Con una cadena de 16 caracteres se pueden representar los 12 dígitos + 3

puntos + `\0`. En 16 bytes también cabe: `localhost\0`.

- **`port`**: número de puerto.
  - `string[6] (6 bytes)`.
  - Con una cadena de 6 caracteres se pueden representar todos los puertos desde el 0 hasta el 65535 + `\0`.
- **`file_chunk`**: tamaño del *buffer* de lectura para ficheros.
  - `stream of bytes (X bytes)`.
  - Se utiliza el tamaño de *buffer* por defecto del sistema operativo. Este valor puede rondar en torno a los 8192 bytes para equipos 64-bit.
- **`datetime`**: fecha y hora.
  - `char[20] (20 bytes)`.
  - Permite representar la fecha y hora siguiendo el formato: `"dd/mm/yyyy hh:mm:ss"` + `\0`.

Otros datos tienen tamaño variable como **`cop`**, el código de operación (`REGISTER`, `UNREGISTER`, `CONNECT`, `...`).

Mensajes

En la siguiente tabla se muestra la secuencia de mensajes enviados entre cliente y servidor central para cada uno de los servicios:

	Cliente =>	**<= Servidor central**
register	1. cop 2. datetime 3. username	4. status
unregister	1. cop 2. datetime 3. username	4. status
connect	1. cop 2. datetime 3. username 4. port	5. status
disconnect	1. cop 2. datetime 3. username	4. status
publish	1. cop	6. status

	2. datetime 3. username 4. filename 5. description	
delete	1. cop 2. datetime 3. username 4. filename	5. status
listusers	1. cop 2. datetime 3. username	4. status if status == 0: 5. number_users 6. username (* number_users) 7. ip_address (* number_users) 8. port (* number_users)
listcontent	1. cop 2. datetime 3. username (client) 4. username (remote client)	5. status if status == 0: 6. number_files 7. filename (* number_files) 8. description (* number_files)

En la siguiente tabla se muestra la secuencia de mensajes enviados entre cliente-cliente para cada uno de los servicios:

	Cliente local =>	**<= Cliente remoto**
getfile	1. cop 2. filename	3. status if status == 0: 4. file_chunk (*local_file_size == remote_file_size*)

Protocolos

Entre los clientes y el servidor central se ha utilizado como protocolo de comunicación *sockets* con TCP. En este problema no es crítico garantizar la entrega de los mensajes pues la pérdida de un mensaje no tiene más consecuencias que retrasos en prestar servicio. Luego se utiliza TCP solamente para mejorar la calidad del servicio y se puede modificar por UDP sin problemas.

Entre clientes también se utilizan *sockets* con TCP. En este caso sí que se desea garantizar la entrega puesto que los accesos a disco, que es donde se hallan los ficheros, es lento. Por tanto, la pérdida de paquetes implicaría leer repetidamente en un cliente un fichero alargando así aún más la entrega del fichero completo a otro cliente.

Tanto para clientes como para el servidor central se utiliza el dominio de comunicación `AF_INET` especificado por POSIX el cual permite la comunicación mediante TCP/IPv4 entre distintas máquinas.

Entre el cliente y el servidor *web* se usan *web services* empleando el protocolo de intercambio de información estructurada SOAP, el cual usa XML como formato de mensajes. Se ha utilizado SOAP frente a otros protocolos como REST debido a su flexibilidad para representar la estructura de los mensajes. Esto permite escalar el sistema en el futuro a pesar de que actualmente solo se utilizan *web services* para obtener la hora y fecha.

Para la comunicación entre el servidor central y el servidor de RPC se utiliza RPC, el cual internamente utiliza *sockets*.

Codificación

Servidor *web*

El servidor *web* se encuentra implementado en un único fichero `./ws-server.py`. Este utiliza como *target namespace*: `http://tests.python-zeep.org/`. Esta URI es proporcionada por `zeep`, la librería utilizada para implementar los servicios *web*, para pruebas.

Este se ejecuta siempre utilizando el puerto 8000. Se ha decidido utilizar este puerto constante en lugar de pedir al usuario que introduzca el puerto por simplicidad, puesto que este tendría que proporcionarlo tanto para el servidor *web* como para cada cliente.

Servidor central

El servidor central se encuentra implementado en un único fichero `./server.c`. La principal funcionalidad del servidor central es centralizar la gestión de la información para la comunicación *peer-to-peer* entre clientes. El servidor no se encarga de enviar ni recibir ficheros.

El servidor central se ha implementado de forma que sea concurrente. Al aceptar una petición de un cliente se crea un nuevo *socket* y un hilo. Así el hilo se encargará de enviar y recibir mensajes usando el nuevo *socket* para una petición del cliente que se está atendiendo. Tras ello el hilo, sus recursos, el *socket* y sus recursos de red asociados se

destruyen. Luego un hilo en el servidor se encuentra asociado en un momento dado a un servicio y no a un cliente.

Por otro lado, se utiliza un modelo de concurrencia bajo demanda, es decir, se crea un hilo y se asigna su servicio por cada petición que se procesa en el servidor. Este modelo permite atender a un mayor y variable número de clientes, simplificar la gestión de hilos pero aumenta la complejidad del sistema cuando existen numerosos hilos trabajando al mismo tiempo.

Para la ejecución de los servicios se utilizan ficheros CSV auxiliares. El fichero `./users.csv` almacena los clientes registrados y el fichero `./connected.csv` almacena los clientes conectados. Para almacenar los ficheros publicados se utiliza un fichero CSV por cada cliente denominado `./<username>.csv`, donde `<username>` es el nombre de usuario del cliente. Estos ficheros se almacenan en el directorio `./files`, el cual debe existir previamente.

El servidor central tiene acceso a los tamaños en bytes de los datos mediante variables globales y constantes definidas al comienzo del programa.

Cliente

El cliente se encuentra implementado en un único fichero `./client.py` como la clase `client`. Los clientes tienen como objetivo final comunicarse *peer-to-peer* con otros clientes. Para ello necesitan antes gestionar la información para la comunicación cliente-cliente con el servidor.

El cliente al ejecutarse hará accesible una línea de comandos para introducir servicios. Al introducir correctamente el nombre y argumentos de cada servicio, se ejecutará un método de la clase para ese servicio. Estos métodos se encargan de validar los parámetros pasados para cumplir con las restricciones de los datos, establecer la conexión entre el cliente y el servidor central y/o entre clientes, enviar el mensaje de la petición, recibir el mensaje de la respuesta y gestionar la respuesta y errores producidos durante la comunicación.

Para la comunicación *peer-to-peer*, el cliente hace uso de ficheros JSON auxiliares:

Cuando se ejecuta el servicio **`listusers`**, se crea o actualiza el fichero `./listusers-<username>`, donde `<username>` es el nombre del cliente llamando al

servicio. Este fichero se usa durante el servicio **`getfile`** para obtener la IP y puerto del cliente remoto. De esta forma no se contacta con el servidor y se evita un *single point of failure*, a cambio es necesario actualizar esa lista antes **`getfile`** de con **`listusers`** cada vez.

Cuando se ejecuta el servicio **`publish`**, se crea o actualiza el fichero `./published-<username>`, donde `<username>` es el nombre del cliente llamando al servicio. De forma similar, el servicio **`delete`** elimina un registro de esta lista. Este fichero se utiliza durante el servicio **`getfile`** en el método privado **`__sharefile`** para conocer si el fichero solicitado por el cliente remoto está anunciado en el servidor por el cliente local.

Los clientes tienen acceso a los tamaños en bytes de los datos mediante variables globales y constantes definidas al comienzo del programa.

Ejecución

Se debe ejecutar primero los servidores y luego los clientes.

Servidor *web*

Para ejecutar un proceso servidor *web* utilizar:

```
python3 ./ws-server
```

Servidor RPC

Para su compilación ejecutar el comando: `make` el cual generará el ejecutable: `rpc_server`. Para ejecutar un proceso servidor central utilizar:

```
./rpc_server
```

Servidor central

Para su compilación ejecutar el comando: `make` el cual generará el ejecutable: `server`. Para ejecutar un proceso servidor central utilizar:

```
./server -p <port>
```

- `<port>`: puerto estático en el que el servidor central atenderá peticiones. Debe ser un puerto no privilegiado, es decir: `1024 <= <port> <= 65535`. Y distinto al

puerto del servidor *web*.

Cliente

Para ejecutar un proceso cliente utilizar:

```
python3 ./client -s <server> -p <port>
```

- `<server>`: dirección IPv4 del servidor central o *localhost*. Para su correcto funcionamiento, la dirección IP debe ser la misma que la dirección IP del servidor central.
- `<port>`: puerto estático del servidor central. Debe ser un puerto no privilegiado, es decir: `1024 <= <port> <= 65535`. Para su correcto funcionamiento, el puerto debe ser el mismo que el puerto especificado para el servidor central.

Pruebas

Debido a la complejidad de probar los servicios con el intérprete de servicios en los clientes, no se proporcionan ficheros de prueba. A cambio se describen a continuación los distintos estados en los que se puede hallar el sistema. Para probarlos, escribir manualmente la entrada.

Descripción	Salida `client`	Salida `server`

`register`:
- Cliente registrado.
	`c> register <username>`
	`dd/mm/yyyy hh:mm:ss`
	`USERNAME IN USE`
		`s> OPERATION FROM <username>`

`register`:
- Errores.
	`c> register <username>`
	`dd/mm/yyyy hh:mm:ss`
	`REGISTER FAIL`
		`s> OPERATION FROM <username>`

`register`:
- Cliente no registrado.
- 1 cliente - 1 servidor.
	`c> register <username>`
	`dd/mm/yyyy hh:mm:ss`
	`REGISTER OK`
		`s> OPERATION FROM <username>`
		`./users.csv`:
		`...`
		`<username>`

`unregister`:
- Cliente no registrado.
	`c> unregister <username>`
	`dd/mm/yyyy hh:mm:ss`
	`USERNAME DOES NOT EXIST`
		`s> OPERATION FROM <username>`

`unregister`: - Errores.	`c> unregister <username>` `UNREGISTER FAIL`	`s> OPERATION FROM <username>`
`unregister`:	`c> unregister <username>`	`s> OPERATION FROM <username>`

- Cliente registrado. - 1 cliente - 1 servidor.	dd/mm/yyyy hh:mm:ss UNREGISTER OK	./users.csv: ... ~~<username>~~ ...
connect: - Cliente no registrado.	c> connect <username> dd/mm/yyyy hh:mm:ss CONNECT FAIL, USER DOES NOT EXIST	s> OPERATION FROM <username>
connect: - Cliente conectado.	c> connect <username> dd/mm/yyyy hh:mm:ss CONNECT FAIL, USER ALREADY CONNECTED	s> OPERATION FROM <username>
connect: - Errores.	c> connect <username> dd/mm/yyyy hh:mm:ss CONNECT FAIL	s> OPERATION FROM <username>
connect: - Cliente registrado y no conectado. - 1 cliente - 1 servidor.	c> connect <username> dd/mm/yyyy hh:mm:ss CONNECT OK	s> OPERATION FROM <username> ./connected.csv: ... <username>;<ip>;<port> ./files/<username>.csv: *empty file*
disconnect: - Cliente no registrado.	c> disconnect <username> dd/mm/yyyy hh:mm:ss DISCONNECT FAIL, USER DOES NOT EXIST	s> OPERATION FROM <username>
disconnect: - Cliente no conectado.	c> disconnect <username> dd/mm/yyyy hh:mm:ss DISCONNECT FAIL, USER NOT CONNECTED	s> OPERATION FROM <username>
disconnect: - Errores.	c> disconnect <username> dd/mm/yyyy hh:mm:ss DISCONNECT FAIL	s> OPERATION FROM <username>
disconnect: - Cliente registrado y conectado. - 1 cliente - 1 servidor.	c> disconnect <username> dd/mm/yyyy hh:mm:ss DISCONNECT OK	s> OPERATION FROM <username> ./connected.csv: ... ~~<username>;<ip>;<port>~~ ... ~~./files/<username>.csv~~
publish: - Cliente no registrado.	c> publish <filename> <description> dd/mm/yyyy hh:mm:ss PUBLISH FAIL, USER DOES NOT EXIST	s> OPERATION FROM <username>

publish: - Cliente no conectado.	c> publish <filename> <description> dd/mm/yyyy hh:mm:ss PUBLISH FAIL, USER NOT CONNECTED	s> OPERATION FROM <username>
publish: - Fichero publicado.	c> publish <filename> <description> dd/mm/yyyy hh:mm:ss PUBLISH FAIL, CONTENT ALREADY PUBLISHED	s> OPERATION FROM <username>
publish: - Errores.	c> publish <filename> <description> dd/mm/yyyy hh:mm:ss PUBLISH FAIL	s> OPERATION FROM <username>
publish: - Cliente registrado, conectado y fichero no publicado. - 1 cliente - 1 servidor.	c> publish <filename> <description> dd/mm/yyyy hh:mm:ss PUBLISH OK ./published-<username>.json: [... {"Filename":<filename>, "Description":<description>}]	s> OPERATION FROM <username> ./files/<username>.csv: ... <filename>;<description>
delete: - Cliente no registrado.	c> delete <filename> dd/mm/yyyy hh:mm:ss DELETE FAIL, USER DOES NOT EXIST	s> OPERATION FROM <username>
delete: - Cliente no conectado.	c> delete <filename> dd/mm/yyyy hh:mm:ss DELETE FAIL, USER NOT CONNECTED	s> OPERATION FROM <username>
delete: - Fichero no publicado.	c> delete <filename> dd/mm/yyyy hh:mm:ss DELETE FAIL, CONTENT NOT PUBLISHED	s> OPERATION FROM <username>
delete: - Errores.	c> delete <filename> dd/mm/yyyy hh:mm:ss DELETE FAIL	s> OPERATION FROM <username>
delete: - Cliente registrado, conectado y fichero publicado. - 1 cliente - 1 servidor.	c> delete <filename> dd/mm/yyyy hh:mm:ss DELETE OK ./published-<username>.json: [... ~~{"Filename":<filename>,~~ ~~"Description":<description>}~~. ..]	s> OPERATION FROM <username> ./files/<username>.csv: ... ~~<filename>;<description>~~ ...

listusers: - Cliente no registrado.	c> list_users dd/mm/yyyy hh:mm:ss LIST_USERS FAIL, USER DOES NOT EXIST	s> OPERATION FROM <username>
listusers: - Cliente no conectado.	c> list_users dd/mm/yyyy hh:mm:ss LIST_USERS FAIL, USER NOT CONNECTED	s> OPERATION FROM <username>
listusers: - Errores.	c> list_users dd/mm/yyyy hh:mm:ss LIST_USERS FAIL	s> OPERATION FROM <username>
listusers: - Cliente registrado y conectado. - 1 cliente - 1 servidor.	c> list_users dd/mm/yyyy hh:mm:ss LIST_USERS OK <username1> <ip1> <port1> <username2> <ip2> <port2> ... <usernameN> <ipN> <portN> ./listusers-<username>.json: [{"Username":<username1>, "IP address":<ip1>, "Port":<port1>}, {"Username":<username2>, "IP address":<ip2>, "Port":<port2>}, ... {"Username":<usernameN>, "IP address":<ipN>, "Port":<portN>}]	s> OPERATION FROM <username>
listcontent: - Cliente no registrado.	c> list_content <username> dd/mm/yyyy hh:mm:ss LIST_CONTENT FAIL, USER DOES NOT EXIST	s> OPERATION FROM <username>
listcontent: - Cliente no conectado.	c> list_content <username> dd/mm/yyyy hh:mm:ss LIST_CONTENT FAIL, USER NOT CONNECTED	s> OPERATION FROM <username>
listcontent: - Cliente remoto no registrado.	c> list_content <username> dd/mm/yyyy hh:mm:ss LIST_CONTENT FAIL, REMOTE USER DOES NOT EXIST	s> OPERATION FROM <username>
listcontent: - Errores.	c> list_content <username> dd/mm/yyyy hh:mm:ss LIST_CONTENT FAIL	s> OPERATION FROM <username>
listcontent:	c> list_content <username>	s> OPERATION FROM <username>

- Cliente registrado, conectado y cliente remoto registrado. - 1 cliente - 1 servidor.	dd/mm/yyyy hh:mm:ss LIST_CONTENT OK <filename1> "<description1>" <filename2> "<description2>" ... <filenameN> "<descriptionN>"	
getfile: - Fichero no existe o no publicado.	c> get_file <username> <remote_filename> <local_filename> dd/mm/yyyy hh:mm:ss GET_FILE FAIL, FILE DOES NOT EXIST	s> OPERATION FROM <username>
getfile: - Errores.	c> get_file <username> <remote_filename> <local_filename> dd/mm/yyyy hh:mm:ss GET_FILE FAIL	s> OPERATION FROM <username>
getfile: - Fichero existe y publicado. - 1 cliente - 1 servidor.	c> get_file <username> <remote_filename> <local_filename> dd/mm/yyyy hh:mm:ss GET_FILE OK ./<local_filename>: (local_filename content == remote_filename content)	s> OPERATION FROM <username>
Cualquier servicio: - N clientes - 1 servidor.	Proceso <username1>: *servicio1* Proceso <username2>: *servicio2* ... Proceso <usernameN>: *servicioN*	s> OPERATION FROM <username1> s> OPERATION FROM <username2> ... s> OPERATION FROM <usernameN>

Consideraciones adicionales

Se puede mejorar el código empleando un fichero CSV compartido entre clientes python y servidor central en C que almacene los tamaños de los datos enviados en los mensajes en lugar de replicarlos como variables globales y constantes en cada programa.

Se puede introducir una *flag* para **`getfile`** en el intérprete de servicios del cliente que actualice la lista de usuarios del cliente de forma automática evitando tener que ejecutar manualmente **`listusers`**.

Conclusiones

La práctica final de Sistemas Distribuidos es una de las más interesantes hasta el momento en la carrera de Ingeniería Informática. En esta hemos diseñado, codificado y probado cómo distintas máquinas de distinta naturaleza e implementación se comunican para conseguir objetivos comunes.