1. Introducción

El segundo entregable se compone de dos ejercicios no relacionados entre sí.

El primero consiste en retomar la idea del "servidor de ficheros" del entregable anterior, pero en este caso implementando un servidor FTP que utiliza colas de mensajes y RMI para la comunicación entre el cliente y el servidor.

El segundo es una extensión del chat p2p que se ha realizado durante los laboratorios.

---

2. Ejercicio 1. Servidor FTP usando RMI y colas de mensajes

La primera práctica de la entrega 2 mantiene la temática de la entrega 1 pero en este caso la implementación se realiza utilizando RMI y colas de mensaje RabbitMQ, para lo cual se utilizará Java como lenguaje de programación. Necesitarás implementar dos procesos (clases): el cliente y el servidor FTP. El cliente envía las peticiones al servidor utilizando una cola de mensajes RabbitMQ y el servidor FTP responde a las peticiones utilizando RMI, invocando métodos que el cliente debe exponer (en este sentido el cliente actúa como un servidor RMI).

El cliente leerá de un `fichero_peticiones` la lista de ficheros a solicitar al servidor. Cuando reciba estos ficheros a través de llamadas RMI, el cliente los irá dejando en una carpeta `directorio_descargas` que se le pasa como parámetro. Se pueden lanzar varios clientes, pero cada uno debe tener su propio directorio de descargas, y posiblemente también su propio fichero de peticiones.

El servidor buscará los ficheros solicitados en una carpeta `document_root` que se le pasa como parámetro, e irá escribiendo en un fichero `fichero_log` todas las peticiones que reciba, cada una con el *timestamp* en que se recibió.

La siguiente figura ilustra la arquitectura general del sistema descrito:

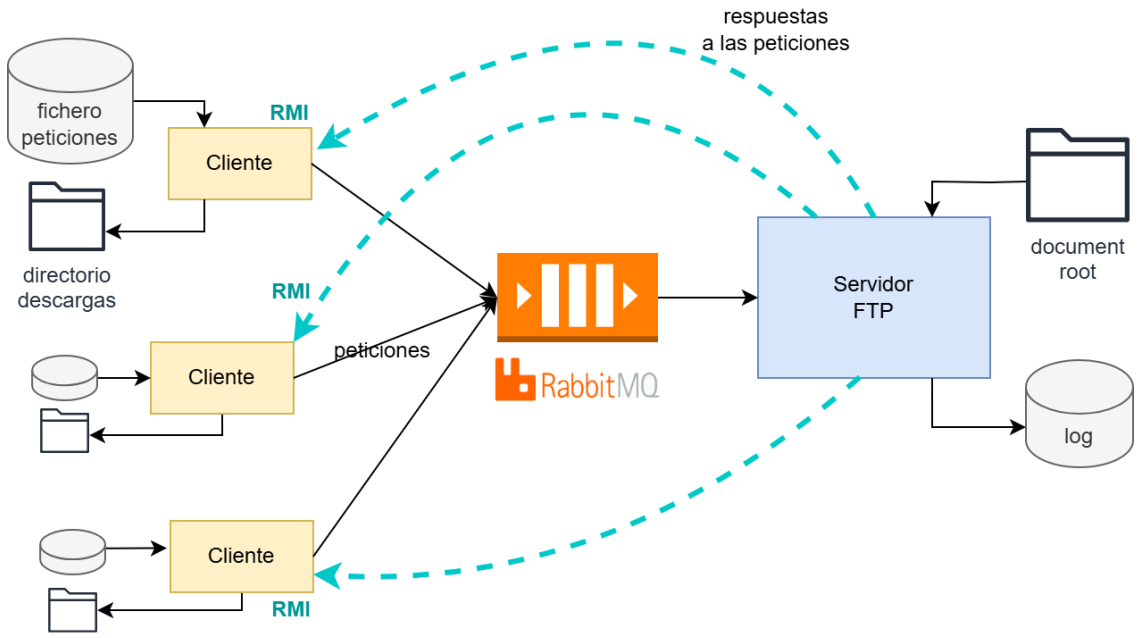

Figura 1. Comunicaciones entre los clientes y el servidor

2.1. Protocolo

Para enviar la petición, el cliente encolará un mensaje en RabbitMQ. El mensaje es una cadena de texto con el siguiente formato:

- Nombre del cliente (por ejemplo `Cliente_1`). Este es el nombre con que ese cliente es localizable via RMI (el cliente se registró en el registro RMI con este nombre).
- Espacio en blanco.
- Comando solicitado. Puede ser uno de los siguientes:
  - `ISFILE <nombre_fichero>`: pregunta si el fichero existe.
  - `ISREADABLE <nombre_fichero>`: pregunta si el fichero es legible.
  - `GETNUMLINES <nombre_fichero>`: pregunta cuántas líneas tiene el fichero.
  - `GETLINE <nombre_fichero> <n_linea>`: pregunta por la línea número `n_linea` del fichero.

Para enviar la respuesta el servidor invocará vía RMI un método del cliente. Para ello primero debe localizar al cliente en el registro RMI, usando el nombre que viene en el comando, y después invocar uno de los siguientes métodos del cliente, según qué respuesta quiere enviar:

- `setIsFile(boolean)`: el servidor invoca este método para responder a la pregunta `ISFILE`.
- `setIsReadable(boolean)`: el servidor invoca este método para responder a la pregunta `ISREADABLE`.
- `setNumLines(int)`: el servidor invoca este método para responder a la pregunta `GETNUMLINES`.

- `setLine(String)`: el servidor invoca este método para responder a la pregunta `GETLINE`.

2.2. Proceso Servidor

Este proceso implementa el servidor FTP. Se trata de un servidor multihilo en el que habrá varios *Workers* encargados de procesar las consultas, acceder a los ficheros solicitados dentro de `document_root`, y una vez tengan la respuesta realizarán una invocación remota en el cliente que realizó la consulta para enviarle la respuesta y escribir en el log.

Para coordinar a estos hilos *Worker* se utilizará una cola bloqueante similar a la que tuviste que implementar en el primer ejercicio de la entrega anterior, sólo que en esta ocasión no será necesario implementarla porque Java ya proporciona una clase para ello (`ArrayBlockingQueue`).

2.2.1. Ejecución del servidor

Ya que la ejecución del servidor implica el uso adecuado del *classpath*, para simplificar su lanzamiento usará un script llamado `lanzar_srvftp`, el cual necesitará los siguientes parámetros:

Forma de uso del lanzador

```
./lanzar_srvftp <tam_cola> <num_workers> <document_root> <fichero_log>
```

Se recuerda que se debe lanzar el registro rmi antes de lanzar el servidor.

2.2.2. Implementación

Esta es la arquitectura interna del servidor:

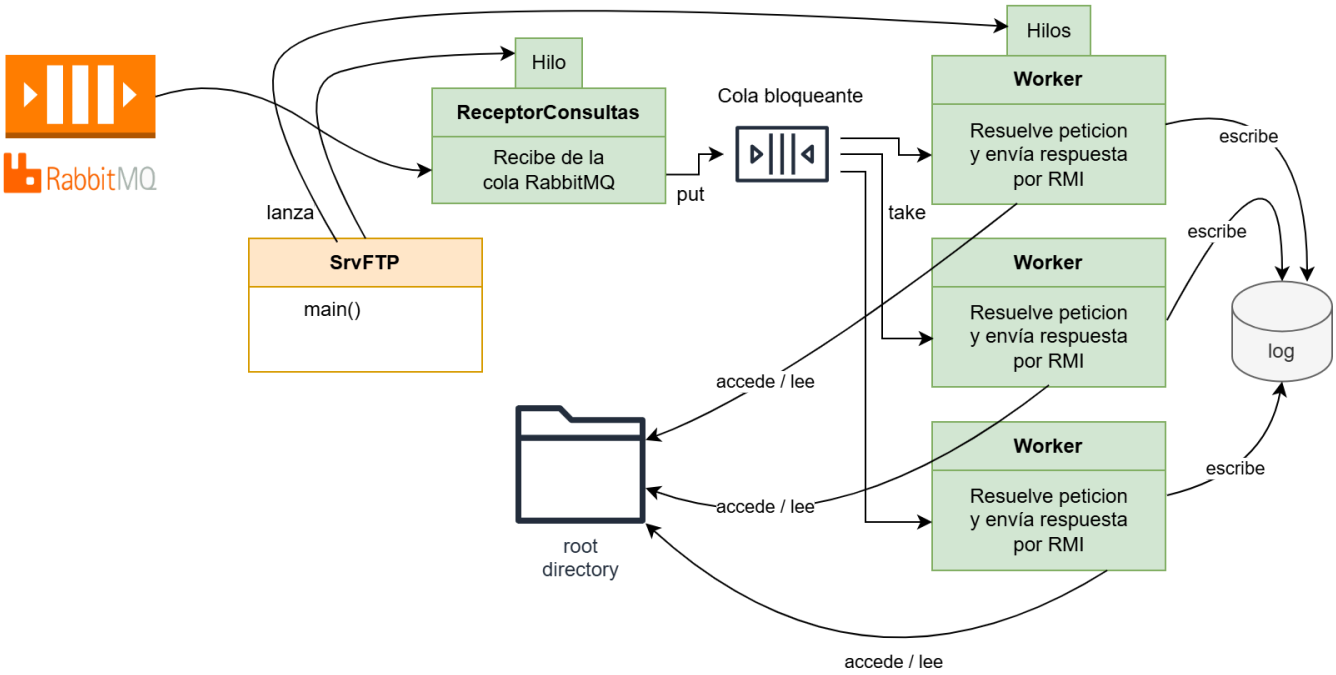

Figura 2. Arquitectura interna del servidor

En el método `main()` de la clase `SrvFTP` se realizan las siguientes tareas:

- Comprobar los argumentos recibidos desde la línea de comandos.
- Instanciar e iniciar los hilos de la clase `Worker`. El método `run()` de la clase `Worker` es un bucle infinito donde se lee de la cola bloqueante la siguiente petición, se tokeniza, se lleva a cabo la acción expresada en el mensaje recibido (lo que implicará acceder a la carpeta `document_root` se localiza el objeto RMI que el cliente ha registrado para recibir las respuestas, se invoca al método adecuado del objeto remoto del cliente y finalmente se escribe en el fichero de log del servidor la petición recibida.
- Instanciar e iniciar un hilo de la clase `ReceptorConsultas`. El método `run()` de la clase `ReceptorConsultas` se encarga de esperar la llegada de mensajes en la cola Rabbit y meterlos en la `ArrayBlockingQueue` a través de la cual se comunica con los hilos de la clase `Worker`, los cuales son los verdaderos encargados en dar respuesta a los mensajes recibidos por el servidor.
- Quedar a la espera de que el hilo `ReceptorConsultas` termine su ejecución (ya que éste nunca termina, el servidor queda siempre en ejecución)

Para implementar esta funcionalidad, se escriben dos ficheros `.java`:

- `SrvFTP.java`: contiene la clase `SrvFTP` con su método `main()` por el que todo arranca, así como las clases `ReceptorConsultas` y `Worker`, que son las encargadas de recibir las peticiones y de procesarlas, respectivamente.
- `FileSystemActions.java`: es un fichero auxiliar con funciones para llevar a cabo sobre el sistema de ficheros en `root_dir` las diferentes operaciones que puede solicitar un cliente (mirar si el fichero existe, si es legible, cuántas líneas tiene, y leer una línea dada)

2.2.3. Formato del fichero de log

El fichero de log contendrá una línea por cada petición recibida, y cada línea tendrá en primer lugar la fecha y hora de recepción del mensaje (en el formato generado por Java), después, separado por una coma, la petición tal como viene del cliente (esa petición contiene en primer lugar el nombre del cliente y luego el comando solicitado). Ejemplo:

Ejemplo de fichero de log

```
2025-04-12 12:00:00.000,Cliente_1 ISFILE fichero.txt
2025-04-12 12:00:01.000,Cliente_1 ISREADABLE fichero.txt
2025-04-12 12:00:02.000,Cliente_1 GETNUMLINES fichero.txt
2025-04-12 12:00:03.000,Cliente_1 GETLINE fichero.txt 1
2025-04-12 12:00:04.000,Cliente_1 GETLINE fichero.txt 2
```

2.3. Proceso Cliente

2.3.1. Ejecución

Se invocaría pasándole tres argumentos, pero al igual que con el servidor usaremos un *script* para simplificar su lanzamiento. Este *script* se llama `lanzar_cliente` y necesita los siguientes argumentos:

Forma de uso

```
./lanzar_cliente <id_cliente> <fichero_peticiones> <directorio_descargas>
```

El primer argumento es un identificador de cliente, el segundo es el nombre del fichero donde se encuentran los ficheros que se quieren descargar del servidor FTP y el tercero es el nombre de la carpeta donde se guardarán los ficheros descargados.

El identificador de cliente es crucial, porque será usado por el cliente como parte del nombre que registrará en `rmiregistry`, para la RMI que el servidor usará para enviarle la respuesta a la consulta. Así, si se le pasa como identificador "X", el cliente se registrará en `rmiregistry` como "Cliente_X".

A diferencia del cliente del entregable anterior que internamente creaba varios hilos para simular varios clientes, en este caso este proceso simula un solo cliente. Para lanzar varios clientes simultáneos podemos utilizar el script `lanzar_clientes` el cual necesita como primer argumento el número de clientes a lanzar y como segundo argumento el fichero con las consultas. Este script simplemente llama varias veces al script `lanzar_cliente` con un identificador de cliente diferente cada vez, pasando a todos el mismo fichero de peticiones pero creando un nombre de carpeta de descargas distinto en cada uno, usando el id de cliente para generar ese nombre, de modo que el cliente con id "X" tendrá su carpeta de descargas "ClienteX".

De todas formas, este cliente tendrá dos hilos, puesto que uno de ellos será el creado por RMI para atender las invocaciones remotas del servidor, mientras que el otro será el hilo principal que envía las peticiones a través de Rabbit. Es necesario por tanto comunicar ambos hilos. El método usado será de nuevo una cola bloqueante, como se describe seguidamente.

2.3.2. Implementación

El método `main()` del cliente se encuentra en el fichero `Cliente.java`. En dicho método se realizan las siguientes tareas:

- Comprobar los argumentos de línea de comandos.

- Crear una instancia de la clase `ClienteImpl`. Esta clase implementa la clase `ClienteInterface` la cual expone 4 métodos remotos: `setIsFile()`, `setIsReadable()`, `setNumLines()` y `setLine()`, los cuales como hemos visto serán utilizados por el servidor para responder a los mensajes enviados por el cliente a través de la cola rabbit.
- Conectar con una cola rabbit que llamaremos de momento “cola_ftp” (aunque en la práctica final debe ser modificada por el nombre y las iniciales de los apellidos del alumno) que es la que creó el servidor ftp. El cliente envía las peticiones al servidor FTP a través de dicha cola Rabbit.
- Invocar al método `procesarDescargas()`` .
  - Este método procesa las líneas del fichero de entrada (`fich_peticiones`) y, por cada línea leída de dicho fichero, tiene que realizar su descarga. Para ello tiene que componer una secuencia de mensajes que serán enviados a través de la cola Rabbit, siguiendo el protocolo antes explicado. Primero debes averiguar si el fichero existe, después si es legible, después cuántas líneas tiene y finalmente iterar pidiendo cada línea de una en una.
  - Observa que tras enviar cada mensaje, es necesario esperar la respuesta enviada por el servidor antes de poder enviar el siguiente mensaje, pero la respuesta no se recibe inmediatamente sino de forma asíncrona cuando el servidor invoque el método RMI correspondiente (el cual es en el fondo un *callback*)
  - Por tanto, para forzar a que el método `procesarDescargas()` espere a que la respuesta del servidor llegue vía RMI, necesitamos sincronizar el hilo principal con el que atiende las peticiones RMI.
  - Esto lo lograremos implementando métodos adicionales en la clase `ClienteImpl` pensados no para que el servidor los invoque remotamente, sino para que los invoque el cliente, dejándole bloqueado hasta recibir la respuesta. Estos métodos se denominan `getIsFile()`, `getIsReadable()`, `getNumLines()` y `getLine()`. Observa cómo por cada método RMI `setXXXXX()` existe la contrapartida `getXXXXX()` que es la que el cliente invoca para esperar la respuesta del servidor. La sincronización interna dentro de la clase ClienteImpl entre los métodos `setXXXXXXXX()` (invocados mediante RMI desde el servidor) y los métodos `getXXXXXXXX()` se realiza a través de una `ArrayBlockingQueue` de Object de tamaño 1.

Esto se representa en la siguiente figura en la que los colores naranja y verde indican los dos hilos que ejecutan las diferentes partes.

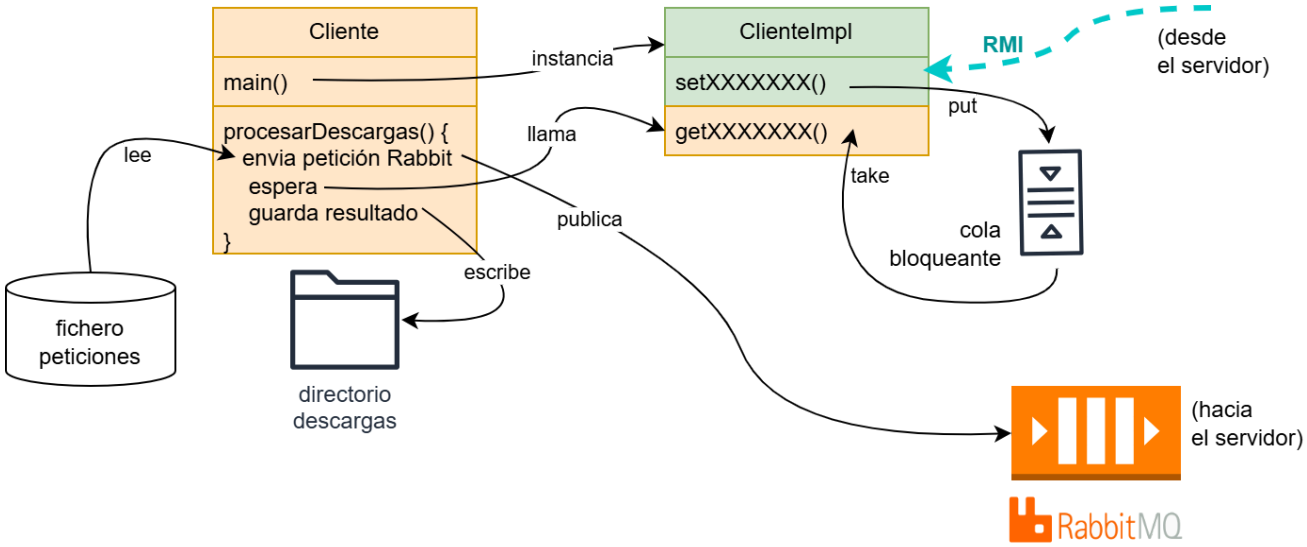

Figura 3. Arquitectura interna del cliente

La implementación se realiza en los siguientes ficheros:

- `ClienteInterface.java` contiene la declaración de la interfaz remota.
- `ClienteImpl.java` contiene la implementación de la interfaz remota, con los métodos `setXXXXX()` y `getXXXXX()` antes descritos.
- `Cliente.java` contiene la clase `Cliente` con sus métodos método `main()` y `procesarDescargas()` antes descritos, junto con un par de métodos auxiliares relacionados con la carpeta de descargas.