

Práctica 1:

Servidor y Cliente TFTP

Universidad de Málaga - E.T.S Ingeniería Informática

Desarrollo de Servicios Telemáticos

Antonio J. Galán Herrera

Indicaciones previas	2
Manual de usuario	3
Servidor	3
Cliente	4
Funcionamiento	5
Estructura del proyecto	5
Discusión crítica	7
Bondades	7
Defectos	7

Indicaciones previas

Esta práctica es funcional, pero no cumple todos los requisitos del enunciado.

Todo el proyecto ha sido ampliamente descrito a través de **comentarios** que dividen el código en secciones dentro de los procedimientos, así como la inclusión de Javadocs para describir la función de dichos métodos. Considero por tanto, que la mejor forma de comprender el funcionamiento es leyendo dichas descripciones de los métodos.

El proyecto se ha creado y construido usando el entorno **IntelliJ**.

Los ficheros se crean en tiempo real, pero si se ejecuta el proyecto desde un entorno, puede que este no los muestre hasta pasado un tiempo o hasta terminar el proceso; sin embargo, en el explorador de Windows (o Linux) sí que aparecen en el momento.

Se incluyen externamente a esta memoria y en formato de imagen, un par de **diagramas de flujo** que indican el funcionamiento de las clases Cliente y Servidor.

Manual de usuario

Para más detalles, existe un ejemplo de uso disponible en 'Archivos/Cliente/ejemplo.txt'.

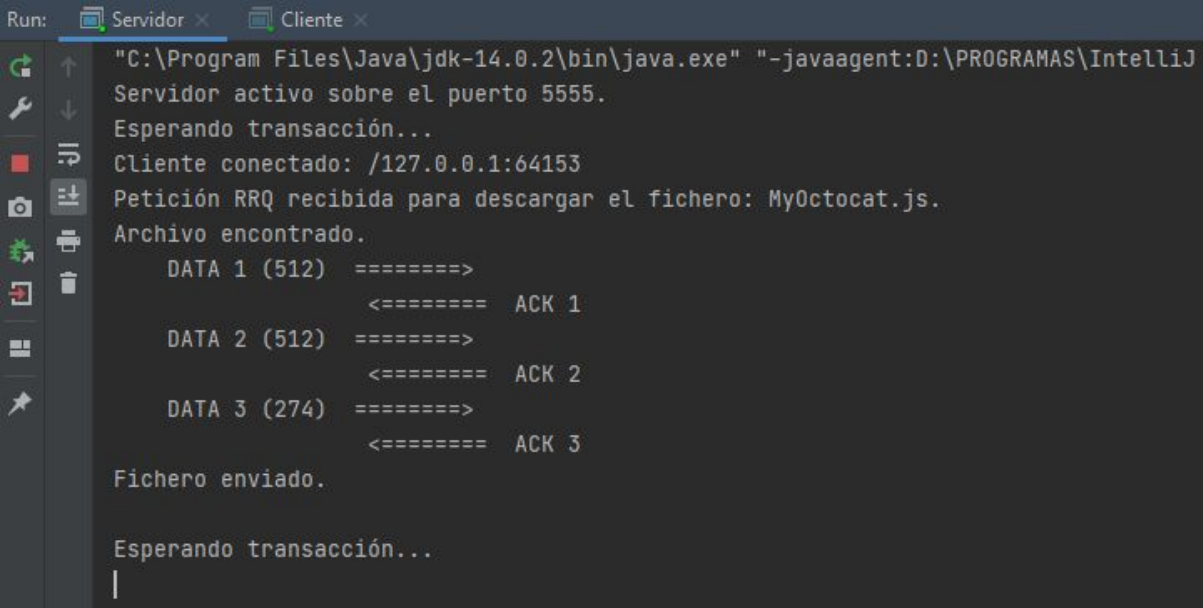
Servidor

Este proceso puede recibir 1 argumento de entrada: **puerto**; en caso de no recibirlo, se solicitará al usuario que ingrese un puerto por consola. El puerto 69 no está disponible, por lo que se pedirá una introducción manual siempre que ese sea el puerto escrito.

El servidor es pasivo, por tanto, esa es la única iteración con el usuario. Una vez se establezca un puerto esperará conexiones de clientes e iniciará los procesos de intercambio de archivos según las peticiones recibidas del cliente (WRQ o RRQ).

Cuando esto ocurra, el servidor mostrará por consola la información de la transferencia: tipo de petición recibida y paquetes DATA/ACK enviados y/o recibidos.

El servidor tiene asociado un directorio dentro del proyecto, llamado *Archivos/Servidor*, que actúa como «base de datos» del mismo; por tanto, todos los archivos recibidos se almacenan ahí, así como todos los archivos que se envíen deben pertenecer al directorio.

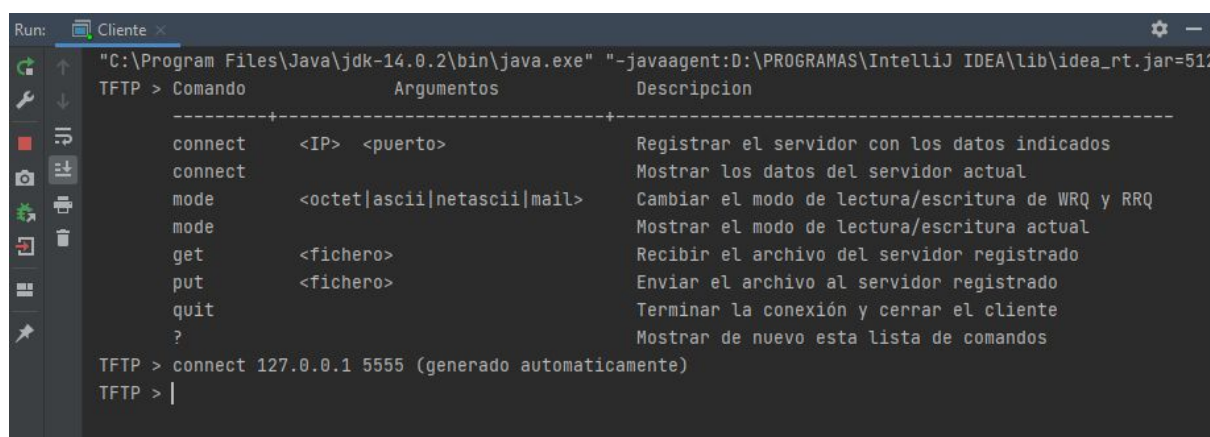


```
Run: Servidor x Cliente x
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" "-javaagent:D:\PROGRAMAS\IntelliJ
Servidor activo sobre el puerto 5555.
Esperando transacción...
Cliente conectado: /127.0.0.1:64153
Petición RRQ recibida para descargar el fichero: My0ctocat.js.
Archivo encontrado.
    DATA 1 (512) =====>
                        <===== ACK 1
    DATA 2 (512) =====>
                        <===== ACK 2
    DATA 3 (274) =====>
                        <===== ACK 3
Fichero enviado.
Esperando transacción...
|
```

Cliente

Este proceso puede recibir 2 argumentos de entrada: **IP** y **puerto**; si se reciben, se establecerá una conexión automáticamente (como se muestra en la imagen); en caso contrario, el cliente arrancará con normalidad sin haber establecido ninguna conexión.

El cliente es activo, solicitando de manera continua una interacción con el usuario a través de su interfaz, que simula una consola de comandos propia. Dicha interfaz detecta cuándo un comando ha recibido los argumentos de forma incorrecta.



```
Run: Cliente x
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" "-javaagent:D:\PROGRAMAS\IntelliJ IDEA\lib\idea_rt.jar=512
TFTP > Comando      Argumentos      Descripcion
-----+-----+-----
connect    <IP> <puerto>      Registrar el servidor con los datos indicados
connect                                Mostrar los datos del servidor actual
mode       <octet|ascii|netascii|mail>  Cambiar el modo de lectura/escritura de WRQ y RRQ
mode                                Mostrar el modo de lectura/escritura actual
get        <fichero>              Recibir el archivo del servidor registrado
put        <fichero>              Enviar el archivo al servidor registrado
quit                                Terminar la conexión y cerrar el cliente
?                                Mostrar de nuevo esta lista de comandos

TFTP > connect 127.0.0.1 5555 (generado automaticamente)
TFTP > |
```

Su funcionamiento viene definido por el propio mensaje informativo que se muestra siempre al inicio del proceso, que consiste en un resumen de los comandos disponibles, sus argumentos y una breve descripción de su funcionalidad.

Al igual que el servidor, el cliente también tiene asociado un directorio dentro del proyecto, llamado *Archivos/Cliente*, que actúa como «base de datos» del mismo; por tanto, todos los archivos recibidos se almacenan ahí. Sin embargo, a diferencia del servidor, el cliente permite enviar cualquier fichero del equipo desde el que se ejecute, empleando una ruta absoluta al fichero.

*Es decir, el comando **get** necesita solamente el **nombre del fichero** que se quiere descargar del servidor, mientras que el comando **put** necesita la **ruta del fichero** que se quiera enviar al servidor.*

*Aunque también existe un campo asociado al modo (**mode**), se ha incluido como referencia al estándar de TFTP y ese campo se incluye en los paquetes a la hora de crearlos, pero el hecho de que esté activado un modo u otro no afecta en nada, ya que solo cambia el valor de una variable **String** cuya función es llenar ese campo.*

Funcionamiento

Partiendo del estado en el que ambos procesos están activos, resultaría el estado inicial, donde el servidor estaría esperando a recibir una nueva conexión, mientras que el cliente estaría esperando un comando.

Lo siguiente sería conectarse al servidor a través del cliente, usando **connect**.

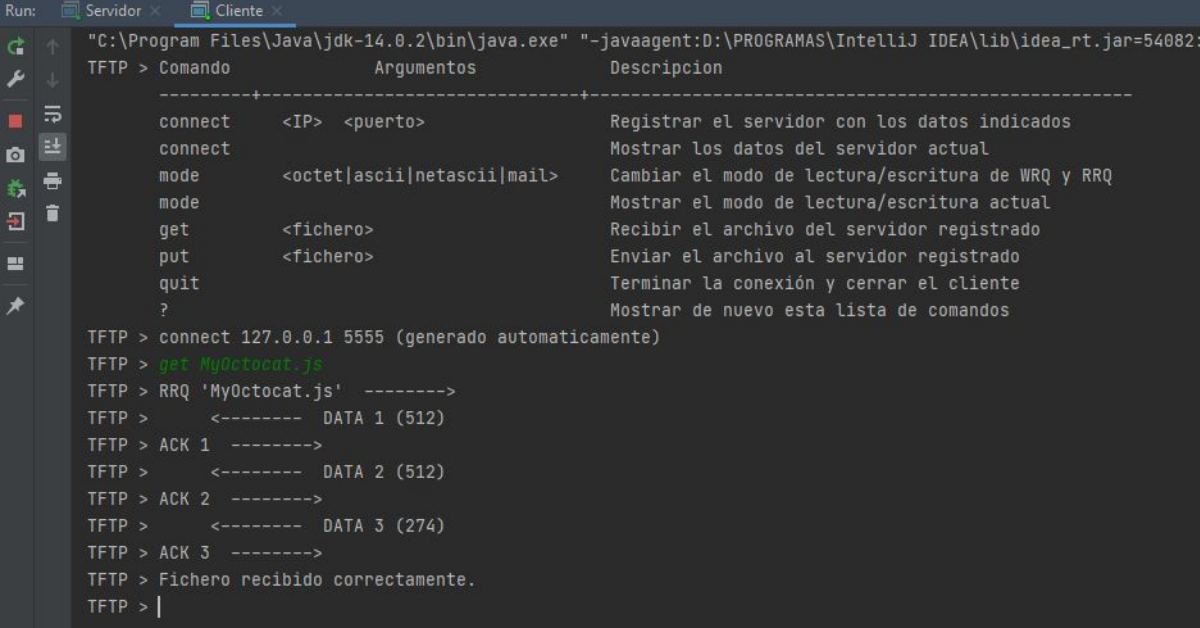
Tras añadir los datos de la conexión, lo siguiente sería **enviar una petición**.

Una vez recibida, el servidor diferencia entre petición RRQ, petición WRQ o cualquier otra cosa, resultando este último caso en un error.

Si es una **petición RRQ**, ejecutará una transacción con el cliente intercambiando paquetes DATA y ACK, iniciando el proceso enviando el DATA 1.

Si es una **petición WRQ**, ejecutará una transacción con el cliente intercambiando paquetes ACK y DATA, iniciando el proceso enviando el ACK 0.

Tras cada transacción, el servidor volverá a quedarse esperando una nueva petición, mientras que el cliente volverá a esperar un nuevo comando por consola.



```
Run: Servidor x Cliente x
"C:\Program Files\Java\jdk-14.0.2\bin\java.exe" "-javaagent:D:\PROGRAMAS\IntelliJ IDEA\lib\idea_rt.jar=54082:
TFTP > Comando      Argumentos      Descripcion
-----+-----
connect  <IP> <puerto>      Registrar el servidor con los datos indicados
connect                                     Mostrar los datos del servidor actual
mode     <octet|ascii|netascii|mail>  Cambiar el modo de lectura/escritura de WRQ y RRQ
mode                                         Mostrar el modo de lectura/escritura actual
get      <fichero>              Recibir el archivo del servidor registrado
put      <fichero>              Enviar el archivo al servidor registrado
quit                                           Terminar la conexión y cerrar el cliente
?                                                Mostrar de nuevo esta lista de comandos

TFTP > connect 127.0.0.1 5555 (generado automaticamente)
TFTP > get MyOctocat.js
TFTP > RRQ 'MyOctocat.js' ----->
TFTP > <----- DATA 1 (512)
TFTP > ACK 1 ----->
TFTP > <----- DATA 2 (512)
TFTP > ACK 2 ----->
TFTP > <----- DATA 3 (274)
TFTP > ACK 3 ----->
TFTP > Fichero recibido correctamente.
TFTP > |
```

Estructura del proyecto

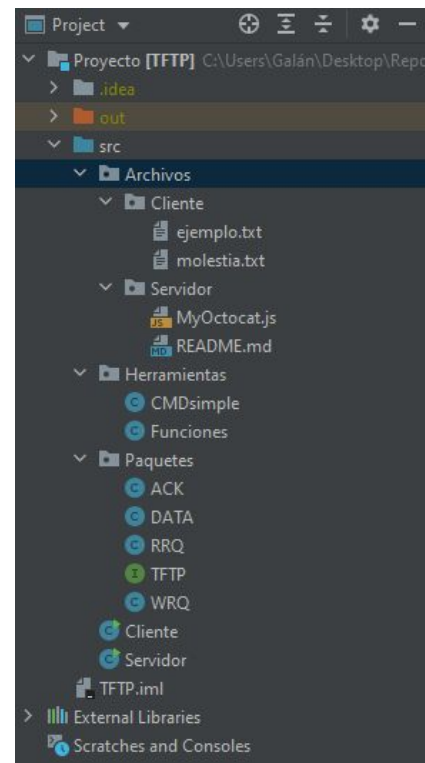
El proyecto se divide en 2 módulos, sin contar la carpeta donde se almacenan los archivos.

- El **paquete «Herramientas»** define las clase:
 - **CMDSimple**, que es la que construye y administra la interfaz del cliente, simulando una consola de comandos.
 - **Funciones**, que describe una serie de métodos comunes empleados por el resto de clases.
- El **paquete «Paquetes»**, contiene una clase para cada tipo de paquete definido por TFTP, todos asociados a una interfaz que les otorga una misma estructura a todas las clases.

Todos los paquetes cuentan con:

- **int opcode**, que almacena el identificador de ese paquete.
- **byte[] buffer**, que almacena la estructura del paquete.
- **void montar()** que crea la estructura del paquete con los datos indicados.
- **void desmontar()** que dada la estructura de un paquete, extrae los datos que contiene.

Como no todos los paquetes son iguales, las clases varían entre sí como en la implementación de **montar()** y **desmontar()**, así como el número de variables.



Discusión crítica

Bondades

Considero que tal y como he planteado la práctica, y gracias al uso de GitHub, he podido trabajar de forma más cómoda, lo que me ha permitido modularizar y documentar el código de manera que cualquiera que lo lea sepa qué ocurre en cada momento si tiene conocimiento del funcionamiento del protocolo TFTP.

Pienso que la estructuración del proyecto ha sido adecuada e intuitiva, tratando de respetar lo máximo posible una estructura parecida entre los paquetes, y tratando de reflejar una simetría en las clases del cliente y el servidor.

La práctica cuenta con la mayoría de requisitos pedidos en el enunciado.

Defectos

No obstante, algunos requisitos no han sido cumplidos, como: controlar los errores mediante paquetes ERROR -que no fueron implementados por ese motivo-, ni verificar las transacciones mediante el uso de un TID. Por tanto, no se manejan errores como enviar/recibir ficheros que ya están guardados o no existen, ni se muestran las estadísticas de paquetes perdidos.

Por otra parte, el servidor se ejecuta permanentemente una vez se inicia, por lo que para finalizarlo es necesario matar el proceso desde el IDE o la consola de comandos; además, las transacciones pienso que se podrían haber optimizado mejor, aplicando una modularización genérica referente a los intercambios de paquetes ACK y DATA.