Protocolo TFTP

Cliente y Servidor (Iterativo)

David Gómez Pérez
Desarrollo de Servicios Telemáticos

1. Manual de usuario

Para la ejecución de este programa, se proporcionan los archivos *Server.jar* y *Client.jar* (el primero para el servidor y el segundo para el cliente). Para la ejecución de cada uno, se deberá de usar los comandos *java -jar Server.jar* y *java -jar Client.jar* respectivamente.

También se proporcionarán tres ficheros de ejemplo para su ejecución. Al finalizar la transacción, se mostrará en el servidor el intercambio de paquetes y en el cliente, un resumen estadístico de paquetes perdidos, enviados y retransmitidos.

1.1. Comandos para el servidor

Para el servidor no hay que introducir ningún comando. Sin embargo, es recomendable ejecutarlo antes que el cliente, pues al ejecutarse este, se mostrará por pantalla el equipo donde se ha lanzado y el puerto de conexión para las comunicaciones.

1.2. Comandos para el cliente

Tenemos seis comandos principales de entrada para el cliente:

"help" o **"HELP"** Muestra la siguiente tabla, donde se ven todos los comandos y sus argumentos, además de una breve explicación.

"quit" Termina la conexión. Se cierra el socket conectado al servidor y se cerrará el cliente.

"**connect** < *host* >**"** Simplemente registra al host al que estamos conectados por el nombre que le pasamos como parámetro.

"mode *octet*||*netascii***".** Cambia el tipo de modo de la transmisión de los datos. Por defecto se encuentra en *octet*. En esta implementación solo se envían

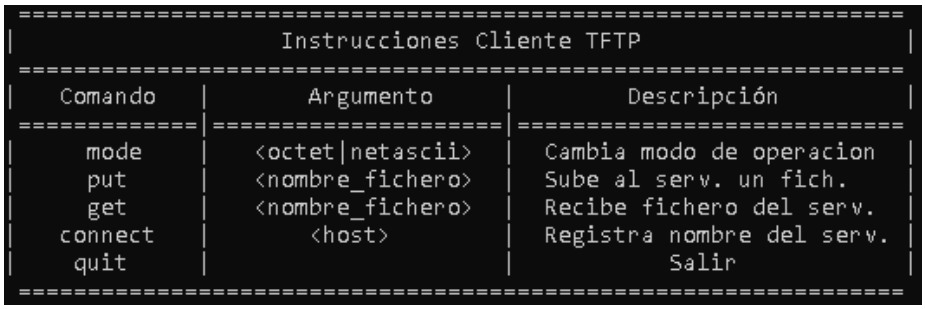


Figura 1: Menú de ayuda

los mensajes por el modo predeterminado.

"get *nombreFichero*"**.** Pedimos al servidor al que estamos conectados un fichero. Este se guardará en el directorio del cliente con el mismo nombre que tenía en el servidor.

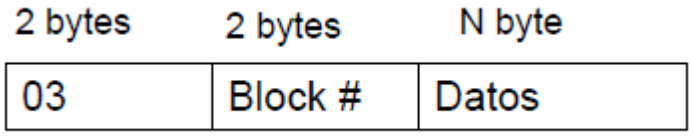
"put *nombreFichero*"**.** Petición para guardar en el servidor un fichero de texto. Este se guardará en el directorio del servidor con el mismo nombre que tenía en el cliente.

2. Clases auxiliares

2.1. Tipos de paquetes

Cada uno de estos paquetes tiene una longitud máxima pre-establecida, en mi caso de 512 bytes.

DataPacket. Paquete que transporta los datos a transmitir. Su longitud máxima de datos a transportar es de 508 bytes, ya que la cabecera ocupa dos bytes y el bloque de datos otros dos.



RRQPacket. Paquete utilizado para la petición de descarga de un fichero. Solamente se envía al inicio de la transacción al utilizar el comando **get**.

WRQPacket. Paquete utilizado para la petición de subida de un archivo al servidor conectado. Solamente se envía al inicio de la transacción al utilizar el

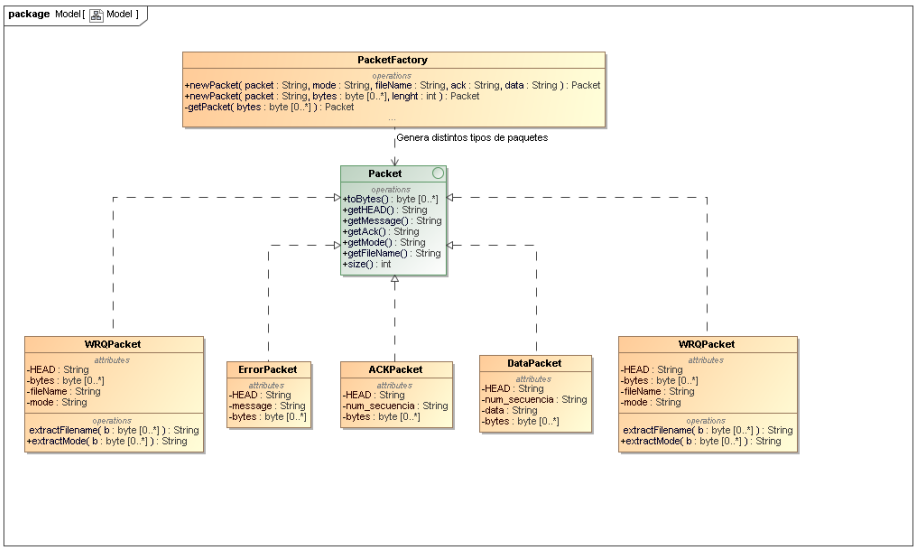
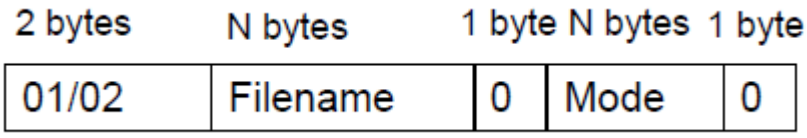


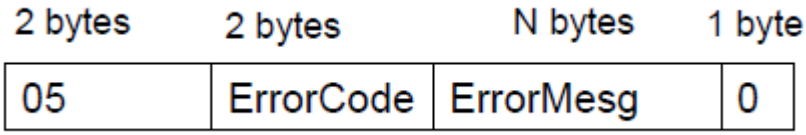
Figura 2: Tipos de paquetes en TFTP

comando **put**.

Estos dos paquetes anteriores tienen el mismo formato de trama, diferenciándose solamente en el número de cabecera, siendo "01"para los paquetes del tipo *RRQPacket* y "02"para los del tipo *WRQPacket*.

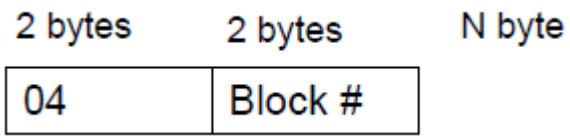


ErrorPacket. Paquete usado para la notificación de cualquier error, como por ejemplo para avisar al cliente de que un fichero no existe. En el campo *ErrorCode*, utilizamos por defecto el valor "00"para simplificar el uso de este tipo de paquetes.



ACKPacket. Paquete utilizado para el envío de confirmaciones de otros pa-

quetes. Ocupa solo 4 bytes y lleva consigo el número de secuencia del paquete a confirmar.



Todos estos paquetes tienen un 5 % de probabilidad de pérdida, a excepción de los paquetes de petición (*RRQ* y *WRQ*), que se envían el 100 % de las veces.

2.1.1. Interfaz Packet

Esta es una interfaz utilizada por todos los tipos de paquetes mencionados anteriormente, ya que todos ellos comparten los mismos métodos. Además, en ella se define la variable de capacidad máxima de cada datagrama (*ECHOMAX = 512*) así como las distintas cabeceras de cada uno de los paquetes (para evitar definirlas en cada una de las clases).

2.1.2. Packet Factory

La clase "PacketFactory", facilita en la creación de cada uno de estos paquetes. Tenemos dos constructores:

newPacket (String packet, String mode, String fileName, String ack, String data). Este constructor se encarga de crear un tipo específico de paquete, dependiendo del parámetro *packet*. En caso de que el nombre de este parámetro no se corresponda con ninguno de los paquetes utilizados, devolvemos un paquete *null*. El parámetro *mode* se utiliza para indicar el modo de transmisión, *fileName* para el nombre del fichero, *ack* para el número de bloque o bloque de confirmación de este datagrama y *data* son los datos del fichero que lleva este.

No todos los paquetes utilizan todos los parámetros durante su creación. En caso de que un parámetro no sea necesario para la obtención de éste, entonces utilizaremos la cadena vacía en ese campo.

newPacket (String packet, byte [] bytes, int lenght). Este constructor se utiliza para la recepción de datos, facilitando la obtención de cada fragmento del datagrama recibido. El parámetro ***length*** se utiliza para indicar el tamaño del paquete recibido

En este constructor, se utiliza el método auxiliar *getPacket*, que dado un array

de bytes, se obtiene los dos primeros bytes de este (cabecera del paquete) y devuelve el tipo de datagrama que estamos recibiendo. En caso de que no sea una cabecera registrada, entonces devolvemos un String vacío, luego este constructor nos devolverá un paquete *null*.

```java
/**
 * Devueve un paquete el tipo "packet" con sus argumentos (pueden ser vacíos)
 * El parámetro "data", se utiliza también para crear el mensaje de error
 */
public Packet  newPacket (String packet, String mode, String fileName, String ack, String data){
    Packet p = null;
    if (packet.equals("RRQ")){
        p = new RRQPacket(fileName, mode);
    }else if (packet.equals("WRQ")){
        p =  new WRQPacket(fileName,mode);
    }else if (packet.equals("DATA")){
        p = new DataPacket(ack,data);
    }else if (packet.equals("ACK")){
        p = new AckPacket(ack);
    }else if (packet.equals("ERROR")){
        p =  new ErrorPacket(data);
    }
    return p;
}
```

```java
/**
 * Devuelve un paquete según el tipo, pasando como parámetro un array de bytes
 */
public Packet newPacket (byte [] bytes, int length){
    String packet = getPacket(bytes);
    Packet p = null;
    if (packet.equals("RRQ")){
        p = new RRQPacket(bytes,length);
    }else if (packet.equals("WRQ")){
        p =  new WRQPacket(bytes,length);
    }else if (packet.equals("DATA")){
        p =  new DataPacket(bytes, length);
    }else if (packet.equals("ACK")){
        p = new AckPacket(bytes,length);
    }else if (packet.equals("ERROR")){
        p =  new ErrorPacket(bytes,length);
    }
    return  p;
}
```

Figura 3: Constructores PacketFactory

2.2. Clase TID

Se trata de una clase que implementa una tupla de valores de cualquier tipo. En este caso, utilizaremos los tipos **Integer** para la creación de los tid. Estos se usan para la comprobación de paquetes que el cliente o el servidor reciben. El método boolean equals (TID< $A, B >$ obj) nos permite comparar fácilmente dos TID (el creado con el paquete recibido y otro creado al inicio del intercambio de datos).

```
public class TID<A, B> {

    private A a;
    private B b;

    public TID(A a, B b) {
        this.a = a;
        this.b = b;
    }

    public A _1() {
        return this.a; }

    public B _2() {
        return this.b; }


    public boolean equals(TID<A, B> obj) {
        return obj._1().equals(this._1()) && obj._2().equals(this._2());
    }

    @Override
    public String toString() { return "[" + this._1() + "," + this._2() + "]"; }
}
```

Figura 4: Clase TID

2.3. Clase Menu

Esta clase se encarga de inicializar el buffer para la entrada de datos (IP y puerto del servidor) y de comandos por consola. Además, el método void commands(), nos imprime por pantalla el menú con todos los posibles comandos que explicamos al principio.

```
public class Menu {
    private static String instrunctions = " ============================================================ \n" +
                                          "|                  Instrucciones Cliente TFTP                  |\n" +
                                          " ============================================================ \n" +
                                          "|   Comando   |       Argumento      |        Descripción        |\n" +
                                          " =============|======================|========================== \n" +
                                          "|     mode    |   <octet|netascii>   |  Cambia modo de operacion |\n" +
                                          "|     put     |   <nombre_fichero>   |  Sube al serv. un fich.   |\n" +
                                          "|     get     |   <nombre_fichero>   |  Recibe fichero del serv. |\n" +
                                          "|   connect   |       <host>         |  Registra nombre del serv.|\n" +
                                          "|    quit     |                      |           Salir           |\n" +
                                          " ============================================================ \n";

    private static BufferedReader bf;

    public Menu() { bf = new BufferedReader(new InputStreamReader(System.in)); }

    public void commands () { System.out.println(instrunctions); }

    public String nextParameter () throws IOException {
        System.out.print( " > ");
        return bf.readLine();
    }

    public String nextAction () throws IOException {
        System.out.print("tftp > ");
        return bf.readLine();
    }
}
```

Figura 5: Clase Menu

3. Clases TFTPClient y TFTPServer

3.1. Variables más importantes

MAXTRIES. Número de intentos máximos en la recepción de paquetes. En caso de que se cumplan el número máximo de intentos, el paquete correspondiente no se envía.

TIMEOUT. Tiempo máximo de espera en la recepción del paquete. Si ya se ha completado este tiempo, el número de intentos de recepción de este paquete se incrementa en uno.

socket. Variable del tipo *DatagramSocket* utilizado para abrir el canal de conexión entre el cliente y el servidor.

sendPacket. Variable del tipo *DatagramPacket* utilizado para la recepción de datos.

tid. Variable del tipo *TID*, utilizada para la comprobación de paquetes.

factory. Variable del tipo *PacketFactory* utilizada para la creación de paquetes.

receiveMss. Variable del tipo *Map<String,String>*, utilizada para almacenar los mensajes recibidos. La clave corresponde al número de secuencia y el valor a los datos asociados a éste.

Hay muchas más variables, aunque estas son las que he considerado de mayor importancia a la hora de crear estas clases. Además, todas ellas están definidas como privadas y estáticas.

3.2. Método "main"

3.2.1. Cliente

En el cliente, el método principal se encarga de inicializar el menú, la variable *factory* y de llamar al método startCommandLine. Este último método se encarga de mostrar la interfaz del cliente durante su ejecución, además de ser el encargado de enviar las peticiones al servidor. Antes de poder utilizar los comandos en el lado del cliente, deberemos de ingresar tanto la IP del servidor como el puerto de conexión a este, donde ambas deberán de ser válidas. En caso de introducir un puerto no numérico, se lanzará una excepción.

3.2.2. Servidor

En el servidor, el método principal se encarga de iniciar la variable *factory* y de generar de forma aleatoria un puerto de conexión comprendido entre 0 y 65535 y distinto de 69. Luego, se mostrará por pantalla tanto el nombre del equipo como la IP y puertos de conexión.

Una vez hecho esto, se llamará el método privado start, el cuál se encargará de atender las peticiones de los clientes indefinidamente hasta que estos se salgan utilizando el comando **quit**.

3.3. Métodos para el envío y la recepción de paquetes

private static void send (Packet packet) Este método recibe como parámetro un paquete de cualquier tipo y lo envía utilizando el buffer del socket creado.

```
private static void send(Packet packet) throws IOException {
    sendPacket = new DatagramPacket(packet.toBytes(), packet.size(), clientAddress, clientPort);
    socket.send(sendPacket);
}
```

Figura 6: Método *send* en cliente y servidor

private static Packet receive () Este método se encarga de recibir un paquete y devolverlo. Además, se comprueba que el paquete recibido pertenece a la entidad con la que estamos conectados.

Entre el cliente y el servidor, existe una pequeña diferencia en este método. Esta diferencia es la creación del identificador o TID. En el cliente, este se crea antes de realizar una petición; mientras que en el servidor se crea en este método, pues necesitamos saber el origen de la petición para poder crear el TID.

```
private static Packet receive() throws IOException {
    receivePacket = new DatagramPacket(new byte[Packet.ECHOMAX], Packet.ECHOMAX);
    socket.receive(receivePacket);
    //creamos un tid auxiliar
    int server_port = receivePacket.getPort();
    TID<Integer, Integer> tid_aux = new TID<~>(socket.getLocalPort(), server_port);
    if (tid.equals(tid_aux)) {
        return factory.newPacket(receivePacket.getData(), receivePacket.getLength());
    } else {
        return null;
    }

}
```

Figura 7: Método *receive* en el cliente

```
private static Packet receive() throws IOException {
    receivePacket = new DatagramPacket(new byte[Packet.ECHOMAX], Packet.ECHOMAX);
    socket.receive(receivePacket);
    //si es el primer paquete que recibimos, obtenemos el puerto por el que este se comunica
    if (!requestReceived) {
        clientPort = receivePacket.getPort();
        clientAddress = receivePacket.getAddress();
        requestReceived = true;
        //creamos tid
        tid = new TID<>(clientPort, serverPort);
        System.out.println("TID para las transacciones: " + tid);
    }
    //creamos un tid auxiliar
    TID<Integer, Integer> tid_aux = new TID<~>(clientPort, serverPort);
    if (tid.equals(tid_aux)) {
        return factory.newPacket(receivePacket.getData(), receivePacket.getLength());
    } else {
        return null;
    }
}
```

Figura 8: Método *receive* en el servidor

3.4. Otros métodos comunes

private static String str (int ack) Este método se encarga de convertir un número de secuencia de entero a cadena de caracteres. Su uso es simplemente para que en los paquetes que utilicen bloques de confirmación, este ocupe dos bytes.

private static List <String> openAndRead (FileReader fR) Este método es utilizado para leer un fichero de 508 en 508 bytes de datos, ya que los 4 bytes restantes corresponden a la cabecera y al número de secuencia correspondiente. Todas las cadenas extraídas las guardamos en una lista de "String"por orden según se van extrayendo del fichero.

private static void saveMessage (String fileName) Este método se encarga de guardar en un fichero el texto extraído del fichero. En caso de que este documento de texto no exista, se creará un fichero vacío en el directorio del cliente/servidor.

3.5. Método para envío de ficheros

private static void sendFile(String fileName) Este método se encarga de enviar un fichero. Como argumento, le pasamos el nombre del fichero que queremos leer.

Primero, tratamos de abrir este fichero y, en caso de que no exista, mandaremos un mensaje de error. Luego, usaremos el método *openAndRead* para leer el fichero. Después de esto, recorreremos la lista de datos a enviar. Durante su recorrido, lo primero enviamos (o no) el paquete creado con la cabecera *DATA* y luego, esperaremos respuesta (un ack). Realizaremos esto hasta que todos los mensajes se hayan enviado.

3.6. Método para la recepción de ficheros

private static void receiveFile() Este método se encarga de ir recibiendo los paquetes que enviamos en el método anterior, y vamos almacenando estos datos en la variable *receiveMss*, para luego mostrarlos (se guarda solo en el caso de que hallamos recibido algún dato).

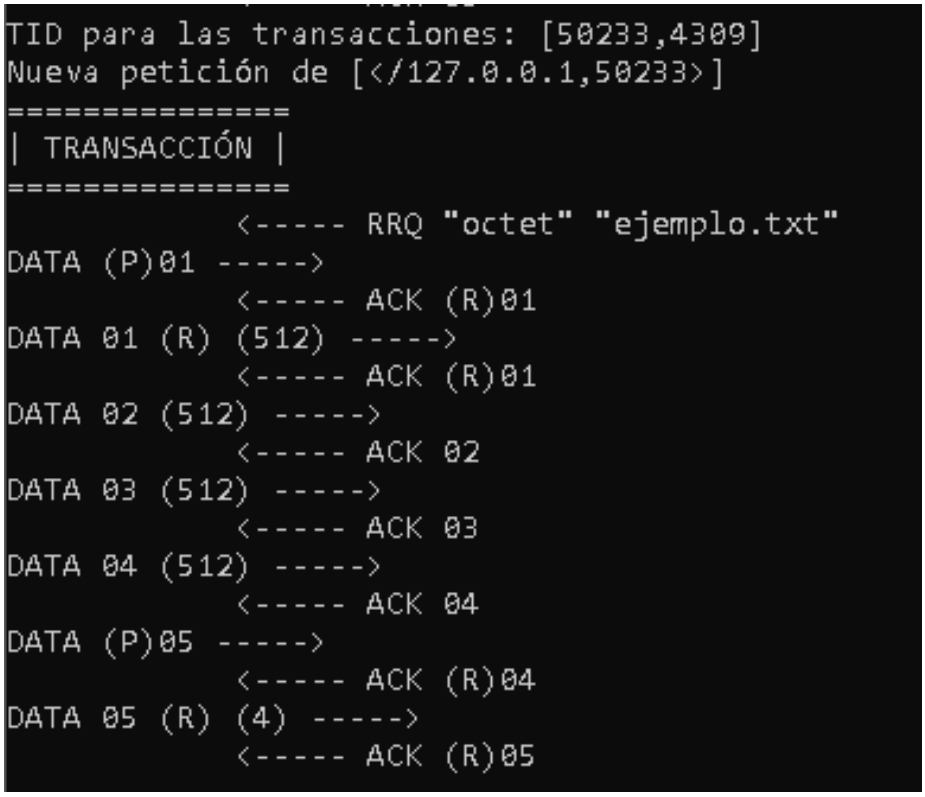
En ambos casos, lo que hacemos es esperar primero al dato del fichero y luego, enviamos un bloque de confirmación. Como no sabemos cuando vamos a parar de recibir mensajes, utilizamos una variable llamada *EOF*, la cual nos indica si hemos recibido el último paquete de datos cuando éste tiene un tamaño menor a *ECHOMAX* (512 bytes).

En caso de que el fichero tenga un tamaño que sea múltiplo de 512 bytes, entonces enviamos un paquete de más sin datos (solo la cabecera), que tendrá longitud de 4 bytes.

4. Ejemplos de ejecución

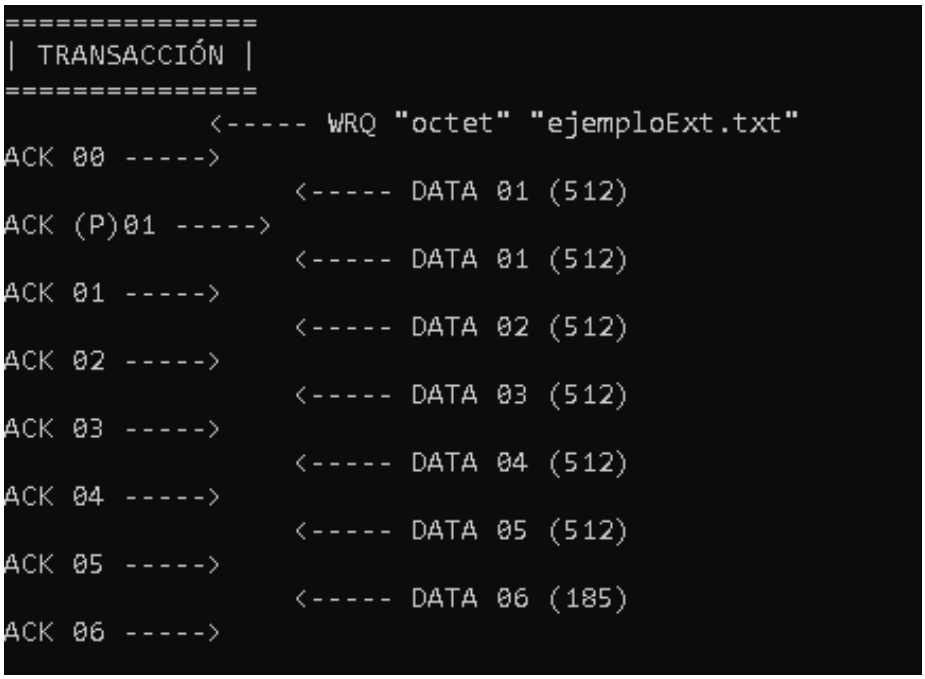
4.1. Obtener fichero *ejemplo.txt* del servidor

El fichero *ejemplo.txt* tiene una longitud múltiplo de *ECHOMAX* bytes, por lo que al final, se enviará un paquete de 4 bytes (solo la cabecera) sin datos para indicar el final de fichero. En el lado del servidor se muestra el intercambio de tramas, mientras que en el lado del cliente, se ven los paquetes enviados, perdidos y retransmitidos sólo por el cliente.



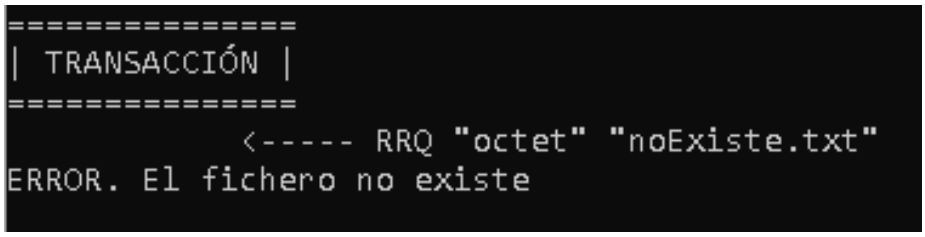
4.2. Subir fichero *ejemploExt.txt* al servidor

El fichero *ejemploExt.txt* es parecido al anterior, solo que tiene una trama más.



4.3. Obtener o subir un fichero inexistente

Cuando tratamos de obtener un fichero o de subirlo, y este no existe, entonces mostramos por pantalla (tanto cliente como servidor), que este fichero no existe.



4.4. Formato de fichero no válido

Los nombres de los ficheros tienen un formato, y este es una cadena alfanumérica, seguida de un punto y una extensión (también alfanumérica). En caso de que este formato no se cumpla, se mostrará un mensaje en el lado del cliente.

5. Crítica

5.1. Puntos en contra

1. Las tramas y el resumen estadístico no coinciden siempre.

2. Hay veces que las tramas no se muestran correctamente.

3. En el comando "put", hay veces que se lanza un error de *nullException* (casi nunca, pero el error está ahí).

5.2. Puntos a favor

1. La instrucción "help"facilita un menú con los diferentes comandos disponibles.

2. Los ficheros descargados o subidos se guardan en otros ficheros.

3. Se comprueba que los paquetes provengan del servidor al que estamos conectados, evitando así recibir paquetes no deseados.

4. Se ha modularización de los distintos tipos de paquetes permite añadir otros nuevos paquetes (en caso de que sea necesario) a este protocolo de manera muy sencilla.