Computer Networks – Trivial File Transfer Protocol (TFTP)

Candidate Number: 184514

University of Sussex 2020

Word Count: 1968

Abstract

The purpose of this report is to give a clear and concise description of the protocols, and how/where the specifications for these protocols have been implemented. The client & server applications speak the TFTP Protocol and have been built (as asked) on top of UDP.

Contents

TFTP over UDP

Introduction

The TFTP (Trivial Transfer Protocol) is a simple file transfer protocol that utilises a client-server architecture to transfer files between hosts or local processes. This implementation is a simplified version when compared to the specifications in the RFC 1350. This includes support for octet mode only, so the files should be transferred as a raw sequence of bytes and doesn't read, write or transfer files as characters. In addition no support for error handling when data duplication occurs. The pre-defined maximum size of a data packet is 516 Bytes. This is to the requirements set in the brief. When the projects are both ran, I have attempted to use print statements to help me with testing, along with giving the user a good understanding of the transfer process. The transfer process begins by a client sending a read or write request to the server.

Server

The TFTPUDPServer class handles the actions which will be carried out for every request, and the different packets. By checking the type of the request the client has made, it is able to call the correct class to handle the request, for example a request of type 'Write' will be handled by the write request handler (WRQHandler) class, and for a read request, the read request handler (RRQHandler) class is called.

Client

When the client is created, the default port is passed. In this instance I have used the port 8451. This is also set as the server port which is used to connect to the server. When ran, the client will pass this port value to the run method. This method is used to get the commands inputted by the user consisting of connect, read (1), write (2), timeout value (t) and quit. The port is set to this as the specification stated that it must be greater than 1024, in order to bind a socket to it, so the TFTP's Port of 69 cannot be used. This value was randomly generated and does not bare any significance other than being larger than 1024.

Read Requests

A read request packet (RRQPacket) is created when a read request is made. This is created using the filename (of the file wanting to be read), or can be made using the raw byte data which will be stored in the packet. Figure 1, shows both constructors that can create a read request packet using either of these inputs. The RRQ Packet class contains the following methods available to use:

getPacketType() – Get packet type

getPacketBytes() – Get raw packet data bytes

getBytes() –Get packet data as bytes

getString() - Get packet data as string

getFilename() – Returns the filename

getMode() – Returns the mode (Always Octet for this implementation)

Server

The RRQ packets are used when the server receives an RRQPacket. The filename is then extracted from the packet and helps find the file within the server. Along with this, an Acknowledgment (ACK) packet is sent back to the client if the file, to acknowledge that it has been found & allows the client to read the file from the server. If the file is not found on the server, an error packet is sent back to the client, of type FILE_NOT_FOUND. (Figure 2)

Client

The RRQ packets are used when the user would like to read a file from the server. The client sends an RRQpacket enclosed with the filename within the packet. The server will return a packet and if the packet is not an Acknowledgement (ACK) packet, an error has occur. (Figure 3)

Write Requests

Write requests are made when the user wants to write a file to the server. When these requests are made, a WRQPacket is created. These are created using the filename of the file to be written to, or by the raw byte data stored within the packet. The constructors can be seen in (Figure 4). The WRQPacket class consists of the following methods:

getPacketType() – Get packet type

getPacketBytes() – Get raw packet data bytes

getBytes() – Get packet data as bytes

getString() - Get packet data as string

getFilename() – Returns the filename

getMode() – Returns the mode (Always Octet for this implementation)

Server

On the server side, the WRQ packets are used when the server receives a WRQPacket from the client. The name of the file is extracted from the packet, and used to create a new file on the server. An ACK packet is sent back to the client if no error occurs.

Client

On the client's side, these packets are used when the user wishes to write a file to the server. The client sends the WRQ packet to the server with the filename within the packet. The client will then receive a packet from the server, either an Error packet, if an error occurred, or an ACK packet allowing the file to be written to the server.

Acknowledgements

Acknowledgement packets are created within the AckPacket class. An ACK packet can either be created using the packet's block number, or the raw byte data stored within the packet (Figure 5). These packets are used by the server to let the client know whether the Write/Read request has been acknowledged and is permitted. If the request is rejected, an Error packet is then sent from the server to the client. The methods within the class implement features such as:

getBlockNo() – Gets the packet's block number

getPacketType() – Gets the packet type

getPacketBytes() –Gets raw packet bytes

Errors

Error packets are created within the ErrorPackets class, which also contains an enumeration. The enum contained within the class is ErrorCodes, which identifies the different types of errors and the accompany error messages. Each error type has a code that relates to the type of error (Figure 6). Error packets are created from the error codes/messages, or the raw bytes that are stored within the packet (Figure 7). The class implements the following methods:

getErrorMessage() – Gets the error message from packet

getPacketType() – Gets the packet type

getPacketBytes() – Gets raw packet data bytes

getString() – Get packet data as string

getBytes() – Get packet data as bytes

getErrorCode() – gets the error code from the packet

fromErrorCode() – gets the type of error from the error code

Server

Error packets are used within the read & write request handlers for reasons such as incorrect mode used, (not octet) or when the file is not found (Figure 8). These error packets are created

and sent to the client, if an error occurs. This is used instead of sending the expected packet type with an error, so the client is aware there is an error.

Client

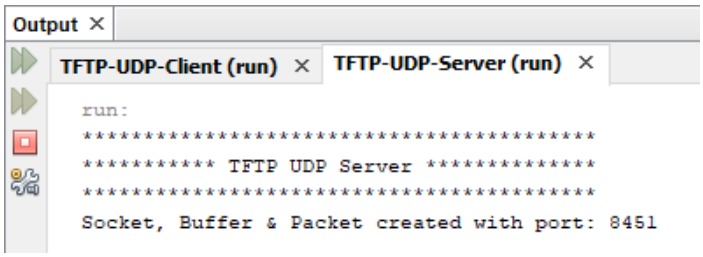
The client will receive error packets from the server instead of the expected packet (for example data packets) if an error has occurred. This is important as then the client is aware of the error. If the client receives an error packet it will print the error message relating to the type of error (Figure 9).

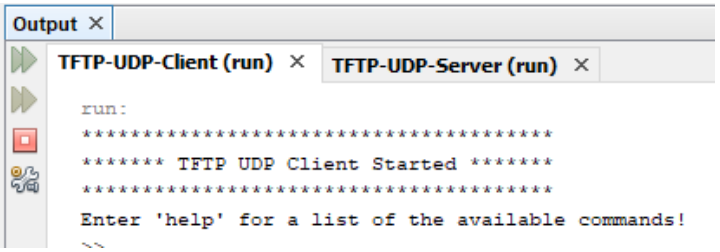
Timeouts

In this implementation, I have set the default timeout value as a constant to be 7000ms, or 7 seconds. This is declared at the top of the class as a constant, which can easily be changed if needed. This is a random value with no meaning behind it. In addition the TFTP-UDP-Client uses the method setSoTimeout. This is used to set the timeout for the socket to the default value. As there is a limit to the number of timeouts that can occur before a transfer stops, I have set this to 15 again at the top of the class as a constant. Again this value is random and has no meaning. Both of these declarations can be seen in (Figure 10).

Simple user Guide (with screen captures)

1. Run both TFTPUDPServer & TFTPUDPClient classes. Both are the only runnable classes as they both have a main method. The outputs should look like:

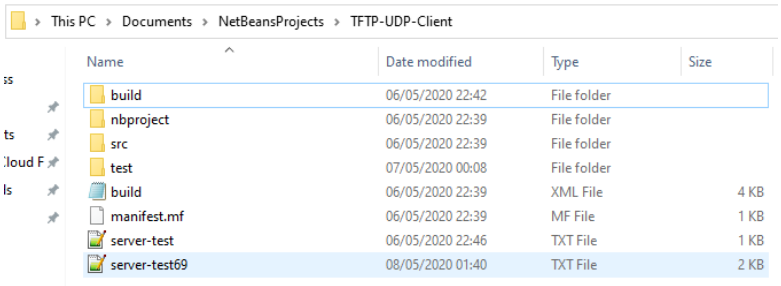




2. Here you can type ‘help’ into the client output to get a list of the commands. We first need to connect to the server. This is done by typing ‘connect’ command followed by the host-name, in this case localhost and the port 8451.

```
help

**************************************
*********** Command List ************
**************************************
connect - connect to server:  [host-name] [port]
1 - Get file from remote-path to local-path!
2 - Put file from local-path to remote-path!
t - Timeout value (ms)
quit - exit
**************************************
>>
connect localhost 8451
>>
```

3. If the connection is successful, no error messages will pop up! Now if we want to get a file from the server to the client, we need to have a file available to read from the server’s root folder. In this case the file ‘server-test69.txt’ is in the root directory. Using the command 1, followed by the name of the file, the file will be copied into the root folder of the client. Ensure to add the extension .txt in the command, or file not found error will pop up. The outputs should look similar to this and now check the root folder for the Client project. The outputs below show what is displayed to the user to break down what is happening step by step. The image on the left is the server’s output and the image on the right is the client’s output. We can see in the image just below the file has been moved into the clients project root folder.



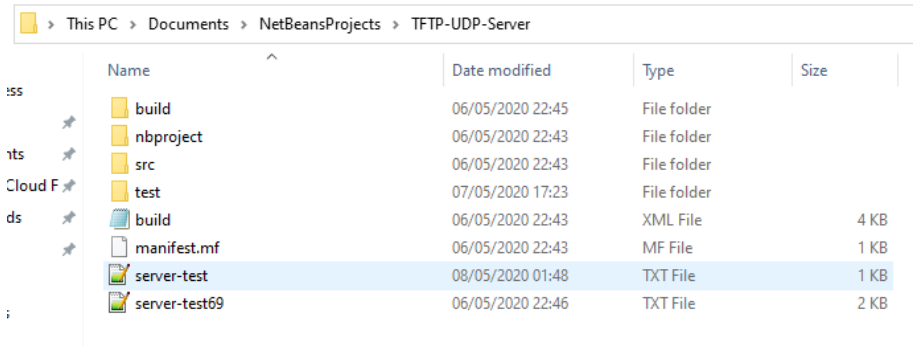
```
Packet has been received!
Calling Read Request Handler (RRQ) & Read request response has been created.
A Read Request (RRQ) has been received: tftp.udp.server.PacketClasses.RRQPacket@24df93dd from /127.0.0.1 : 62771
Socket has been created and timeout set!
File input stream & First buffer created!
No. of Bytes read: 512
Data packet created & sent to the client!
Send to Client Method Invoked!
Created Received Packet & File buffers
First Packet: True!
First Packet is of type DATA, Packet Length Stored.
Datagram packet created and being sent to client with Address: /127.0.0.1and Port: 62771
Send method called for Socket.
Created Received Packet & File buffers
First Packet: False!
File Input Stream Read. Bytes Read: 512

Created packet to be send!
Datagram packet created and being sent to client with Address: /127.0.0.1and Port: 62771
Send method called for Socket.
Created Received Packet & File buffers
First Packet: False!
File Input Stream Read. Bytes Read: 19

Created packet to be send!
Datagram packet created and being sent to client with Address: /127.0.0.1and Port: 62771
Send method called for Socket.
Created Received Packet & File buffers
First Packet: False!
File Input Stream Read. Bytes Read: -1
```

```
1 server-test69.txt
***Receive file***
Buffer created.
Datagram created.
First packet!
Datagram sent by the socket.
Waiting to receive response
Socket Received Datagram!
No Timeout has occurred.
Got the port: 62772
Data Packet has been received!!
Writing to the Data Packet.
Acknowledgment number incremented.
Not the first packet!
Datagram sent by the socket.
Waiting to receive response
Socket Received Datagram!
No Timeout has occurred.
Data Packet has been received!!
Writing to the Data Packet.
Acknowledgment number incremented.
Not the first packet!
Datagram sent by the socket.
Waiting to receive response
Socket Received Datagram!
No Timeout has occurred.
Data Packet has been received!!
Writing to the Data Packet.
Acknowledgment number incremented.
Data Packet is Final packet!
Created a New packet to send.
Send method called! Sending AckPacket.
>>
```

4. Now if we wish to write a file to the server from the client we must the command ‘2’ followed by the name of the file in the clients root folder to send to the server’s root folder. We will write the file server-test.txt to the server. Below shows the Servers output, followed below by the client’s output. Next to the client’s output, we can see the servers root folder which now includes the file server-test.txt.

```
Packet has been received!
Calling Write Request Handler (WRQ) & Write Request response has been created.
A Write Request (WRQ) has been received: tftp.udp.server.PacketClasses.WRQPacket@51252877from the address: /127.0.0.1 with the port: 51674
Datagram Socket has been created with a timeout of: 7000ms
File output stream created & receive file method is called!
Datagram Packet sent by the socket! (Send method invoked).
Socket has responded! Datagram Packet received.
Port got: 51674
Packet received & is a Data Packet!
Data Packet of length: 512
Block number of Data packet: 1 with ACK number of: 0
Datagram Packet sent by the socket! (Send method invoked).
Socket has responded! Datagram Packet received.
Packet received & is a Data Packet!
Data Packet of length: 0
Block number of Data packet: 2 with ACK number of: 1
Data Packet is the final packet! New Ack packed creatred to send!
AckPacket has been sent!
```

```
2 server-test.txt
First Packet!
New datagram packet created
Number of timeouts: 0
Method send called!
Waiting for a response.
Response received!
Block Number equal to the first packet's block number!
The port: 51675
A TFTPPacket created using the Datagram Packet.
Packet is of type: ACK
Block number is: 0
Ack Packet block number is: 0
Reading the file buffer.
Bytes read from the file buffer: 512
Block Number: 1 bytes read: 512 Data Packet Created.
New datagram packet created
Number of timeouts: 0
Method send called!
Waiting for a response.
Response received!
A TFTPPacket created using the Datagram Packet.
Packet is of type: ACK
Block number is: 1
Ack Packet block number is: 65536
Reading the file buffer.
Bytes read from the file buffer: -1
Prevoius Packet Length: 512
Data Length: 512
Block Number: 2 bytes read: 0 Data Packet Created.
New datagram packet created
Number of timeouts: 0
Method send called!
Waiting for a response.
Response received!
A TFTPPacket created using the Datagram Packet.
Packet is of type: ACK
Block number is: 2
Ack Packet block number is: 131072
Reading the file buffer.
Bytes read from the file buffer: -1
Prevoius Packet Length: 0
Data Length: 512
>>
```



5. If the user of the client wishes to quit they can use the command ‘quit’ and the output will look like:

```
>>
quit
BUILD SUCCESSFUL (total time: 1 minute 51 seconds)
```

Otherwise if the user wants to know the default timeout value in ms, can use just the command ‘t’ or if the user wishes to update the timeout value use the command ‘t’ followed by an integer amount. Using just ‘t’ however will end the connection.

```
>>
t 1000
Timeout time in ms: 1000
>>
```

Additional Features that could have been implemented

When looking back on my implementation of the TFTP Protocol, there were a few features that I believe I could have further implemented. Firstly, the implementation is tested over a lossless channel and as an additional feature simply introduce a packet loss model at each end of the nodes. In addition, a useful message could be used to describe the directory of the sent/received file. This would have been useful if someone did not read the user guide and didn't know where to look for the file, but I have only just thought of this when writing the report. Finally, the TFTP-UDP-Client file names, error messages and the modes should be in Netascii. As Java doesn't support Netascii natively, I have decided to use Us-ascii. Netascii is a subset of us-ascii so also works.

Figures

```
/**
 * Creates a Read Request (RRQ) Packet using the filename.
 *
 * @param filename - Name of the file to be written to.
 */
public RRQPacket(String filename) {
    this.fname = filename;
    byte[] fnameBytes = getBytes(filename);
    byte[] modBytes = getBytes(mode);
    this.bytes = new byte[fnameBytes.length + modBytes.length + 2];
    ByteBuffer buffer = ByteBuffer.wrap(this.bytes);
    buffer.putShort((short) getPacketType().getOpcode());
    buffer.put(fnameBytes);
    buffer.put(modBytes);
}

/**
 * Creates a Read Request (RRQ) Packet using the raw byte data.
 *
 * @param dataInPacket  - Array (Byte) of the data to be stored in packet
 * @param len length of the packet
 */
public RRQPacket(byte[] dataInPacket, int len) {
    this.fname = getString(dataInPacket, 2);
    int offset = 2;
    while (dataInPacket[offset] != 0 && offset < len) {
        offset++;
    }
    offset++;
    this.bytes = new byte[len];
    System.arraycopy(dataInPacket, 0, this.bytes, 0, len);

}
```

Figure 1

```
} catch (FileNotFoundException e) {
    System.out.println("Error Packet");
    //Error packet created and sent to the client.
    ErrorPacket errorPacket = new ErrorPacket(ErrorPacket.ErrorCodes.FILE_NOT_FOUND, "file not found: " + rrqPKT.getFilename());
    DatagramPacket packetToSend = toDatagramPacket(errorPacket, addressOfClient, portOfClient);
```

Figure 2

```
} catch (FileNotFoundException ex) {
    ErrorPacket errorPacket = new ErrorPacket(ErrorPacket.ErrorCodes.FILE_NOT_FOUND, "Can't write to: " + localFile);
    //.send called sending error packet
    sckt.send(toDatagramPacket(errorPacket, serversAddress, serversPort));
```

Figure 3

```java
/**
 * Constructor creates a Write Request (WRQ) Packet using the filename.
 *
 * @param fname - The name of the file to be written to.
 */
public WRQPacket(String fname) {
    this.fname = fname;
    byte[] fnBytes = getBytes(fname);
    byte[] modeBytes = getBytes(mode);
    //Size of bytes array needs to be larger than (fnamelen + len of mode)
    this.bytes = new byte[fnBytes.length + modeBytes.length + 2];
    ByteBuffer buffer = ByteBuffer.wrap(this.bytes);
    buffer.putShort((short) getPacketType().getOpcode());
    //Adds to buffer bytes of mode and fname
    buffer.put(fnBytes);
    buffer.put(modeBytes);
}

/**
 * Alternate Constructor creates a Write Request (WRQ) Packet from
 * the Raw Byte Data.
 *
 * @param dataInPacket - Data to be stored in the packet
 * @param len - Length of the packet
 */
public WRQPacket(byte[] dataInPacket, int len) {
    /*Filename is the set to a string value of dataInPacket.
    Done by converting scalar value  to a string.*/
    this.fname = getString(dataInPacket, 2);
    //Start Position of Mode.
    int modeOffset = 2;
    while (dataInPacket[modeOffset] != 0 && modeOffset < len) {
        modeOffset++;
    }
    modeOffset++;
    //Bytes stores the len of packet.
    this.bytes = new byte[len];
    /*Copies dataInPacket array from start pos 0 to bytes array start pos 0.
    Len number of elements copied.*/
    System.arraycopy(dataInPacket, 0, this.bytes, 0, len);

}
```

Figure 4

```java
/**
 * Creates a new AckPacket (Acknowledgement Packet) from block number.
 * @param blockNo block number of packet
 */
public AckPacket(int blockNo) {
    this.blockNo = blockNo;
    this.packetBytes = new byte[PACKET_LEN];
    ByteBuffer buffer = ByteBuffer.wrap(packetBytes);
    buffer.putShort((short) getPacketType().getOpcode());
    buffer.putShort((short) blockNo);
}

/**
 * Constructor creates a new AckPacket from the byte data.
 * @param dataInPacket byte array of data to be stored in packet
 * @param len length of byte data
 */
public AckPacket(byte[] dataInPacket, int len){
    ByteBuffer buffer = ByteBuffer.wrap(dataInPacket);
    buffer.position(2);
    this.blockNo = buffer.getShort();
    this.packetBytes = new byte[len];
    System.arraycopy(dataInPacket, 0, packetBytes, 0, len);
}
```

Figure 5

```
UNDEFINED(0, "Undefined error, check error message!"),
FILE_NOT_FOUND(1, "File not found!"),
ACCESS_VIOLATION(2, "Access violation!"),
DISK_FULL(3, " Disk full / allocation exceeded!"),
ILLEGAL_TFTP_OPERATION(4, "Illegal TFTP operation!"),
UNKNOWN_TID(5, "Unknown transfer ID!"),
FILE_EXISTS(6, "File exists already!"),
NO_USER(7, "No such user!");
```

Figure 6

```
/**
 * Constructor that creates an Error packet from the Error code
 * and Error message.
 *
 * @param eCode  - code of error to be stored in packet
 * @param eMessage - message to be stored in error packet
 */
public ErrorPacket(ErrorCodes eCode, String eMessage) {
    this.errorCode = eCode;
    this.errorMessage = eMessage;
    byte[] msgBytes = getBytes(eMessage);
    this.bytes = new byte[msgBytes.length + 4];
    ByteBuffer buffer = ByteBuffer.wrap(bytes);
    buffer.putShort((short) getPacketType().getOpcode());
    buffer.putShort((short) eCode.getErrorCode());
    buffer.put(msgBytes);
}

/**
 * Constructor that creates an Error packet from the raw Byte data.
 *
 * @param bytes  - Byte array of data to be stored in packet
 * @param len - Length of data to be stored in packet &Used as size of array
 */
public ErrorPacket(byte[] bytes, int len) {
    ByteBuffer buffer = ByteBuffer.wrap(bytes);
    buffer.position(2);
    this.errorCode = ErrorCodes.fromErrorCode(buffer.getShort());
    this.errorMessage = getString(bytes, 4);
    this.bytes = new byte[len];
    System.arraycopy(bytes, 0, this.bytes, 0, len);
}
```

Figure 7

```
} catch (FileNotFoundException e) {
    System.out.println("Error Packet");
    //Error packet created and sent to the client.
    ErrorPacket errorPacket = new ErrorPacket(ErrorPacket.ErrorCodes.FILE_NOT_FOUND, "file not found: " + rrqPKT.getFilename());
    DatagramPacket packetToSend = toDatagramPacket(errorPacket, addressOfClient, portOfClient);
```

Figure 8

```
if (pkt instanceof ErrorPacket) {
    System.out.println(((ErrorPacket) pkt).getErrorMessage());
    return;
} else if (pkt instanceof DataPacket) {
    //Else if the packet received is a data packet (What we want.)
    System.out.println("Data Packet has been received!!");
```

Figure 9

```
private final int PCKT_LEN = 516, DATA_LEN = 512, MAX_AMOUNT_TIMEOUTS = 15;
//Random number again chosen for timeout length
int TIMEOUT = 7000;
```

Figure 10

(All visible red lines are my NetBeans 80 character line) – Used for formatting when copying code.