Candidate No. -149111

Networks Coursework 2

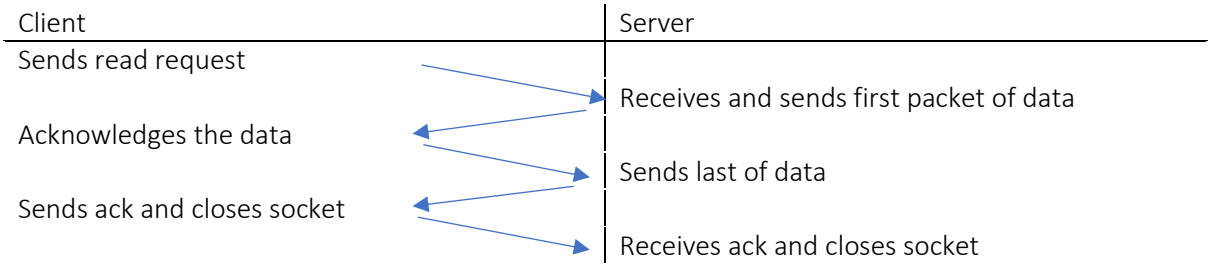
TFTP is a protocol that copies files from a client to a server (write) or to copy files from the server on to the client (read). TFTP is a simplified version of FTP however, this comes at the cost of it being less capable. It is implemented on top of UDP and TCP.

In this report, I will be describing the way I have coded TFTP for both UDP and TCP. The design choices I have made will be discussed. I will also be referencing my code throughout to make the descriptions clearer.

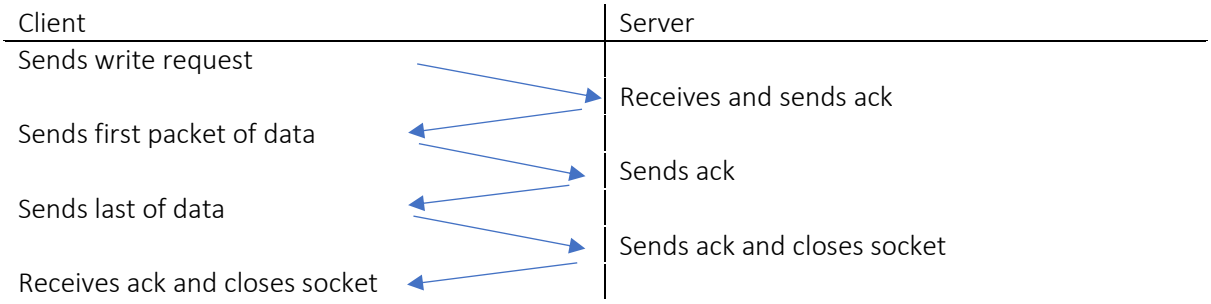
TFTP – UDP

UDP is unreliable and connectionless. When implementing TFTP using UDP, acknowledgements need to be used throughout to ensure that packets have reached their destination and let the other side know that more data can be sent.

For UDP the read request transactions go like so:



And the write requests like so:



Client

The client side requires 3 arguments to run, an IP address, request type and a file name. To start, the IP address (args[0]) is taken and a DatagramSocket is made with it (line 93). The type of request is then found using an if statement (Line 100-117), a 1 means it's read request and a 2 means it's a write request.

A DatagramPacket is made based on the request type. This is done in the createRequest method (line 150). This method takes in an opcode and a file name and will change them with the mode (octet) into bytes, then inserting them into an array. Once this is made it is sent to the server (line 126).

A write request means a file needs to be read in (lines 327). This is done using the Files class which, has a method that can read in a file in bytes and puts it into a byte array.

The socket has a timeout set (line 128) which will throw a SocketTimeoutException if it doesn't receive a reply in an amount of time. This time is set to 2000 this is because if we set it too high then there will be a slow reaction to packet loss or corruption. Whereas, if it's set too low then we have premature timeouts which mean unnecessary retransmission.

Packets are collected from the socket, which is done in the method collect (line 188). An empty DatagramPacket is used to store the collected packets content. When we receive the packet its buffer is put into the empty packets (inPacket) buffer (line 198).

To continuously receive packets a receive statement is put inside a while loop (line 195). This will execute until lastPacket becomes true, which happens when a data packet's size is less than 516, an error packet is received or we finish sending all the data to the server.

The receive() method is surrounded in a try catch so that if a timeout occurs then it's caught and the last packet can be resent again. If the server never received the expected packet, the client will resend its last packet. However, if the server did receive the packet but the servers reply was lost, then the client will still fire a timeout even though it's not necessary. This packet will be ignored and the server will then timeout and resend the packet we need.

When we receive a packet, there is an if statement to find out a packets type. An error packet will cause the socket to close. If it's an ack, it starts by checking whether we have sent all our data (line 214). If so then the socket can be closed.

If all our data hasn't been sent then we enter the else statement where it starts by setting up the block number. Block numbers are used to see if we are receiving the packets in the correct order. The length of the content is used to initialise amountLeft which represents the amount we have left to send for this packet. This is either set as 512, if we have more than 512 bytes to send (line 229) or to the actual amount left to send (line 231).

A for loop is used to write the bytes we are going to send into a ByteArrayOutputStream which, stops when i is equal to amountLeft. Then if all the files data has been sent then allSent is set to true (line 243) and we need to wait for a final ack before closing the connection.

The packet is formed with all the header information and the data. It is assigned to outPacket (line 244) which is a global variable so it can be resent if needed. There is a final if statement after the send which will only execute if the numbers of bytes sent in the last packet is 512 and if so, then an empty packet is sent.

The final kind of packet that can be received is data. It starts by checking the block number is correct (line 277), if it isn't then a timeout will happen. Otherwise, the data is copied into a ByteArrayOutputStream (line 287).

Once the data is extracted an ack packet is sent back and we exit the method. We then check if it's a read request (line 131) and if it is we write the data we stored in the ByteArrayOutputStream to a file. This is done in the method writeFile (line 313).

Server

The code for the server is the same as the client. There are however, a few differences. Firstly, the server is made multithreaded. This means that when the main server receives a packet it makes a new thread and sets up a new socket for it as one socket cannot deal with multiple connections, a new port number must be chosen.

The if statement is extended so that it can pick up the two types of requests. A read request (line 82 UDPResponder) will result in the same thing as what the client's data condition does (line 272-298 UDPClient), it will just send a data packet. A write request (line 206 UDPResponder) will send an ack with a block number of zero to indicate to the client that it's ok to start sending data.

The server will also send an error packet (line 322 UDPResponder) if it's trying to read a file from the server that doesn't exist. This will cause both the sockets to close.

The server has to be made multiple threaded so that it can deal with multiple clients requesting it at the same time. This is done by separating it into two classes. One class is UDPServer, this will be the main server with a hard-coded port that the client will send to, to start a connection. This will then receive the packet and make a new UDPResponder which extends Runnable class (line 35 UDPServer).

When this is called and a new thread is created, a random number between 1024 and 65535 is used to make a new socket. This is because you cannot have multiple clients using the same port only one can use a port. Making threads and giving them each a new socket with a random port fixes this and allows multiple clients requesting the server.

TFTP – TCP

To implement the TFTP with TCP is simpler than UDP. This is because TCP is a reliable protocol that guarantees reliability and that the packets are in-order. Due to this a lot of the code from UDP can be erased as we no longer have to deal with Acks.

Client

Firstly, a server is setup the same as UDP. Once we have the port we need to get the input and output streams (line 100-101). The input stream is where the socket can read the data that the server sends and the output stream is where the data is written, to send to the server.

Once the socket is set, the type of request needs to be established. This is done the same as in UDP with an if statement (lines 104-115). Once determined the request packet is made, this again is the same as in UDPClient class. Once the packet is created it is written to the output stream.

If the request type is a write, then the data is sent straight after the request. You no longer have to wait for a reply from the server to send, as this is TCP and it's reliable. The packet containing the data is sent to the server. Block numbers are still sent, even though there is a guarantee that they arrive in-order. This is kept because we want to keep the packet structure the same.

The structure and creation of the data packets to be sent on a write are the same as in UDP which is explained above. The differences here are that they are no longer put into DatagramPackets and sent, everything is kept in a byte array and put into the output stream when you want to send.

A request for read means data is just going to be read in, nothing needs to be returned to the sender. A byte array is setup which will keep hold of all the data being read in (line 200). A while loop is then entered (line 203), which will iterate through until an error or the last packet is received. We know it's the last packet when the amount of data that can be found in the input stream is less than 516. Once all the data has been read in the data is written out to a file (line 230).

Data could be written to the output stream faster than it is being read in by the other side. This can result in the code reading too many bites at once and never ending. To fix this an if statement is used to only take 516 bytes, or less, out of the input stream (line 210-216).

Once all the data has been sent or received then the socket and input streams are closed and the client has finished.

Server

The server is made multithreaded so that it can deal with multiple clients at the same time. This is done by making a server socket which is going to be the main server that all requests are sent to (line 36 TCPServer). This server socket will accept any requests that are passed to it and then put into a new thread. These threads are of class TCPResponder and where the transaction deals with a particular client.

Again, like the client, the input and output streams are setup to send and receive data from the server (line 57-58 TCPResponder). When the thread is started, a while loop is entered that essentially blocks the program until there is data in the input stream (line 64). Once the input stream has data then the program can move forward.

The data from the input stream is then put into a buffer (line 69), and the op code and file name is extracted (line 71-72). If it's a write request then the server will just wait for data and write it to a file, this is the exact same as if the client was dealing with a read request. If the request is a read request then the same would happen as if the client was doing a write request.

Once all the data has been sent or received then the socket and input streams are closed and the thread is finished.