UCD School of Electrical and Electronic Engineering

EEEN20060 Communication Systems

Internet & TCP Report – Option 2

Name: Chukwuebuka Amadi-Obi **Student Number: 21384846**

Name: Mohammed Raian Hossain **Student Number: 21346793**

Brightspace Group: 42

By submitting this report through Brightspace, I certify that ALL of the following are true:

1. We have read the *UCD Plagiarism Policy* (available on Brightspace). We understand the definition of plagiarism and the consequences of plagiarism.
2. We recognise that any work that has been plagiarised (in whole or in part) may be subject to penalties, as outlined in the document above.
3. We have not plagiarised any part of this report. The work described was done by the team, and this report is all our own original work, except where otherwise acknowledged in the report.

Introduction (Ebuka)

The aim of this assignment was to transfer a file by communicating over the Internet using the transport layer protocol TCP. To achieve this, we had to design our own application layer to accept input from the user, our own protocol to communicate over the internet, as well as design both the client and server programs to implement our protocol design.

We designed the protocol together, then divided the work so that Raian worked mostly on the client program, while Ebuka worked on the server side. Raian worked on uploading data from the client to the server then adapted it to send data from the server to the client (downloading data on the client end.) We both then worked to synchronise the client and server's communication expectations, such as the sending of the "welcome" and "goodbye" messages at the start and the end of the program.

TCP (Ebuka)

TCP allows a client to make a connection to a specific port on a server, and then send or receive a stream of bits over that connection (we worked with bytes in implementing our protocol). It guarantees the successful delivery of the files over the network, which means we don't need to implement any form of error detection. The client requires human interaction to send and receive data, but the server needs no human operation; it responds to requests automatically. TCP operates by sending streams of data bytes, it doesn't operate on a Stop & Wait protocol. This means that we need to have some way to control when and how information should be extracted, potentially by using markers, size information or both.

We were given functions to create a socket to listen for incoming connections, a function to bind that socket to a port, and another to listen for client connections on the server side. On the client side we were given a function to connect to the server. We were also given a function to print an error message if an error occurs and functions to send and receive data on our established TCP connections for both the client and server side.

Our Protocol (Raian)

In this protocol we had to design the interaction between the client and server to ensure every case that can occur is covered. To begin with, the server will be active and the client will try to establish a connection with the server. If successful, the server will send a welcome message and the client can then receive that and send its first request. Based on the request, the client and server will go down one of two paths.

If the client requests an upload, the server will check to see if it can accept the file, then send a response, either yes or no. If the client receives yes, then a stream of bytes will be sent to the server for it to accept. Once it's all accepted, or if the transfer fails, a message, stating the outcome, is sent. If the client receives no, then the interaction skips past the byte-sending and the outcome message.

If the client requests a download, it will prepare itself to accept the file before it even sends the request. The server will then respond to the request based on whether it can send the file or not. If it can, then a response will be sent saying so, followed by the stream of bytes of information directly following afterwards. If not, then the data sending and receiving is skipped.

Once either interaction has occurred, the Client will send a message to the Server marking the end of the interaction. The Server then waits for the client to either continue their communication and send a new request or end it.

Client Request (Raian)

Each initial request begins with either an upload or download byte.

If it's an upload byte, then the following information is the file size information and file name, built in the order of upload byte, size information, "#" size end marker, filename then, "{" end marker.

If it's a download byte, then all that follows is the filename.

Server Response (Raian)

The initial response from the server should begin with either a yes or a no byte.

In the context of an upload response, the server only needs to send the yes byte, or the no byte with a reason for saying no in the frame to be sent.

For a download response, the server responds with either the yes or no byte, with the yes byte having size information sent back, while the no byte also has its reason for rejecting the request in the sent frame.

Client Program (Raian)

Client Overview (Raian)

The client programme will ask the user for input at different stages. At the very beginning, it will **ask for information to connect to the server**. Once the connection is established, the client will then **ask for the course of action** to take with the server. Once the action has been completed or attempted, **it will ask if you want to do more with the server,** this is the repeat request, at which point you can say yes and do something else, or say no and then go back to the original start, asking for information to connect to a server.

If the client's connection to the server fails, it'll ask you again to input an IP address and port number. If the requested action between the client and server fails, then you'll be directed to the repeat request. You can also exit the client programme at the IP address stage of initial inputs or at the request-type stage of initial inputs.

Handling User Input (Raian)

In order to get information from the user, the IP address and filenames are saved in arrays while the port number, request type and repeat request are saved in variables, all through scanf(), while using fgets() to clear the input buffer. The request type and repeat request are put in while loops, which are only broken when a correct response is given from the user, while the other inputs, due to the nature of their uses, have to set off a flag in order for the code to understand that the information they stored was invalid and needs to be refreshed. For example, the IP address can only be identified as being wrong once it's been put to use. So after it's been tried and it fails to connect to the server, instead of having an individual while loop, a flag is set instead to ensure no other code runs while this error is in place, until a suitable part of the code is reached, in this case, the very end of the overarching "while (repeat == 1)" loop, where it'll go back then ask for the IP address and port number again.

Requests and Responses (Raian)

The client can send 4 different types of requests. The main two are Upload and Download, but it can also send ERROR and EXIT. The client also sends the repeat request response to the server as well when the moment arrives.

For Upload, the client will send a frame as described before, with U as the first byte, the file size next followed by '#' to mark the end of the number. Then the file name, with the extension included will come in next then the frame ends with '{'.

```
frametx[0] = up_down;      //Insert U back into the first array spot
frametx[1 + number] = '#';     //Enter end of size information marker
strcat(frametx, request);   //Add filename to frame
strcat(frametx, "{");   //Add end of initial request marker
```

For Download, the client will send a frame as described before, with D as the first byte, then the file name, with the extension afterwards.

After any data transmission attempt has occurred, the client will send a response message to the server "Thanks!" to let the server know it's either received the data or it's sent everything. This signifies the general end to a sending of data bytes.

After this, the client will also send the repeat request response to the server, "RY" to say yes, or "RN" to say no. The R is to signify "Repeat" and is just used for the server to prepare for the next request.

The frame sent for PROBLEM just contains "E" and EXIT just contains "A" in the first array space.

```
// if the user wants to exit, then send a message to the server to
close down the connection
if(stop == 1)
{
response[0] = 'A';
response[1] = '{';
response[2] = 0;
retVal = send(clientSocket, response, MAXRESPONSE, 0);
printf("Exiting client\n");
}
```

File Transfer (Raian)

The client has access to a separate folder called "fileStore". This fileStore is used to store any downloaded files from the server and any uploadable files it could send to the server. This adds a bit of protection to prevent any chances of any part of the client project files to be overwritten from a server download, and helps with general organisation of files.

The below code illustrates that once different checks have been made, the data bytes are simply sent constantly in blocks when being uploaded until the end of the file has been reached. No markers are necessary since we stop once the entire file has been sent. The bolded code shows the important parts, the rest just consists of checks for errors and conditionals to ensure the actions are taken under the right client request type.

```
if(up_down == 'U')  // if the client requests an upload
 {
   if(response[0] == 'Y')  // if the server allows the upload
     {
       ...
       if (retVal != 0)  // if there was a problem
       {
         ...
       }
       else
       {
         while (!feof(fp)) // continue until end of file
         {
          retVal = (int) fread(data, 1, BLOCKSIZE, fp); //readbytes
          if (ferror(fp))  // check for problem
          {
            ...
          }
          else
           {
             totalBytes += retVal;   // add to byte counter
             //Send the block of data that was read
             retVal = send(clientSocket, data, retVal, 0);
```

Server Program (Ebuka)

Server Overview (Ebuka)

At the highest level our server program performs these actions:

1. The server **listens for connection requests** to a certain port, so it is ready for whenever the client tries to establish a connection. When the client connects, the server **accepts the connection** assuming it isn't already connected to another client.
2. The server **receives a request** from the client and reads it. From the information in this request it determines whether the client wants to upload or download data from the server.
3. It **sends a response** to the request, and **then executes on it**, whether this be accepting files from the client or sending it. If the server can't execute the request, it'll notify the client through the response.
4. If the client has no other requests, it'll **close the connection**.

How it performs these actions is outlined in depth below:

Listening for & Accepting Connections (Ebuka)

To set up our server to listen for connections, two TCP socket identifiers were created. One to listen for connections and another one to actually connect to the client. The TCPCreateSocket() function is used to create the socket to be used in other functions.

Once the sockets are created the TCPServerSetup() function is used. This function needs an identifier for a TCP socket (the listening socket) and a port number. We decided that our server will listen for connections on port 32980, we chose not to change the port from the example one given because port values below 2000 are used for standard services.

```
 retVal = TCPserverSetup(listenSocket, SERVER_PORT);
  if (retVal == FAILURE)
    stop = FINAL;
```

After the server is set up, it listens for connections until it finds one with a client. When this happens the TCPServerConnect() function is used, this function takes an identifier for the socket that was listening for requests, and returns an identifier for the socket that has been successfully connected to the client.

```
connectSocket = TCPserverConnect(listenSocket);
    if (connectSocket == FAILURE)  // check for problem
      stop = CLOSE;   // prevent other things happening
```

Requests and Responses (Raian)

The server won't send anything in response to EXIT, ERROR or the repeat request, these only act as informational updates for the server to act accordingly.

In the case of either upload or download, the server response will contain either Y or N in the first array space to signify either yes or no to the request, then the rest of the frame's contents are based on the scenario.

For an accepted upload, 'Y' will be sent and nothing else. The client will then send the stream of data bytes for the server to take in. If the server rejects the request, then an 'N' followed by the reason for the rejection comes in.

```
if (fp == NULL) // check for problem
{
    perror("Problem opening file");
    printf("errno = %d\n", errno);
  // if an error has occurred, then build the negative response frame
    response[0] = 'N';
    // add negative response reason to array
    strcat(response, "Problem opening file             ");
    stop = CLOSE; // don't allow for any receiving
}
else // if there are no issues, send a positive response
{
    response[0] = 'Y';
    response[1] = 0
    stop = CONTIN;
}
// send response built based on situation
retVal = send(connectSocket, response, strlen(response), 0);
```

For an accepted download, the first response frame space will contain ‘Y’, followed by the size information of the file to send. The server will then immediately send the stream of data bytes to the client to try and accept. If the server rejects the download, then the frame response will contain ‘N’ at the start then give the reason as to why it’s rejecting the request as well.

After any attempt has been made at these actions, the server will expect a “Thanks!” response frame to be sent from the client.

File Transfer (Ebuka)

Once the server has determined whether the client wants to upload or download, the server uses a switch statement to execute the right code for uploading or downloading.

```
switch (ReqType){
      case(UPLOAD):
            **Receive file from client**
            break;

      case(DOWNLOAD):
            **Send file to client**
            break;
}
```

In the case of an upload request, the filename from the request is used to open a new file on the server side in the fileStore folder. We used a special folder so there is no chance of the client accidentally overwriting any important files by sending one with the same name as another file, such as sending “server.c” for example.

The file is opened for writing using fopen() and if everything goes smoothly, the server sends an affirmative response which signals the client to send the bytes for the file. They are received using recv() and written to the opened file using fwrite(). This continues in a loop until the full file has been received.

The server handles a download request by opening the file to read (also using the filename and fopen()) and then measuring the size of the file. It uses this info to build the affirmative response to the download request. Once this response has been sent, the server reads the bytes from the start of the file and sends them in blocks of 100. This continues in a loop until the end of the file is reached.

Clean Up and Closing Connections (Ebuka)

The client sends a response after the file has been transmitted, this contains information on whether the user wants to transmit another file or not. Steps 2-4 (receiving requests, sending a response and then executing the requests) occur in a loop until the client has no more files to send. This prompts the server to send the goodbye message as well as call the TCPCloseSocket() function that disconnects and then closes the TCP socket. Then the loop is broken and the server is shut down.

Testing (Ebuka)

Testing our programs was performed at all stages of its development. We both tested the server and the client individually, the client tested to make sure it was handling user input correctly, and the server tested to ensure that it was reading the requests properly.

Testing TCP Connection (Ebuka)

Testing the TCP connection was a simple matter, we put in the IP address of the server at the client and saw the connection notification at the server. We tested this connection with the loopback address on one computer and also tested it on two computers on networks like UCD Wireless or personal hotspots.

The error detection was also tested, if a bad IP was entered the following message is displayed:

```
Enter the IP address of the server (e.g. 137.43.168.123): 12345

Enter the port number on which the server is listening, or enter E to
exit: 32980
ClientConnect: Trying to connect to 12345 on port 32980
ClientConnect: Problem connecting
PrintProblem: WSA Code 10051 = A socket operation was attempted to an
unreachable network.
```

Testing Request Handling (Ebuka)

Below is an example of some debug text printed on the server side:

```
Received total of 16 bytes: |U5190#crest.jpg{|

****************************
Received upload request!
****************************
DEBUG REQUEST INT:85
DEBUG FILENAME - crest.jpg, FILESIZE - 5190
received request, sending response
Opening fileStore\Zcrest.jpg for binary write
```

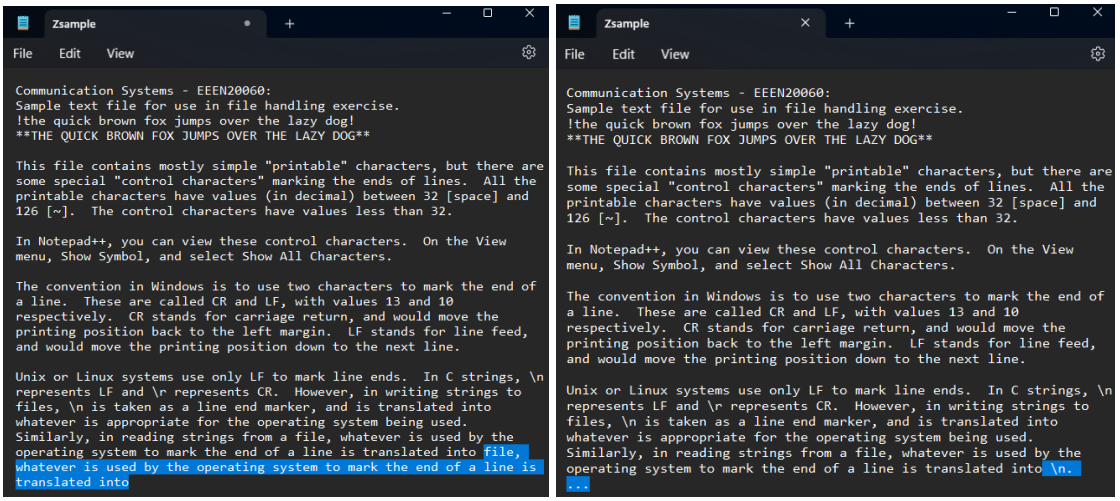
- By printing the full request as well as the type of request received, we could see that it was being received properly and that it read the request type properly.
- The 85 printed is the ASCII code for the letter ‘U’, the first value in the sequence of characters sent as the request.
- Both the filename and the file size were printed to make sure they were being read properly from the request
- Finally the file path is printed to make sure the data being received will be written to the correct file.

By printing this debug information, we could test the sending and receiving of the requests easily, we tested it with different characters as well as numbers and other symbols in the file name. There is also an appropriate error message if the file name is invalid.

Testing File Transfer (Ebuka)

File transfer was tested using different file types such as .txt, .jpg and even .docx files. These files were sent and received with no issues over the different networks which allowed this transfer. Sample.txt was useful to make sure the send and receive loops were properly implemented, to ensure there either weren't any files received with any excess bytes, or that receiving wasn't cut short.

Below you can see how too much was printed on the left for Zsample.txt while the right shows what was expected. The highlighted parts show the difference in the texts. This is an example of when too much was being written to the output file and even overwriting some data at the end of the file.



Conclusion (Raian)

We have succeeded in implementing the application layer protocol to transfer many types of files using TCP. The application allows for transfer both ways and is robust enough to allow for user error in inputs at different stages.

The protocol we designed is simple and efficient, separately sending request and response frames before sending any form of data, meaning we don't have to differentiate between the data bytes and header bytes. The only area of inefficiency would be the use of the "Thanks!" frame, sent from the client to the server, to mark the end of a successful attempt of data transfer in either direction, as it may be possible to implement the protocol in C without it.

This final implementation was designed in such a way that if the same server code was on multiple devices, then each server could have custom welcome and goodbye messages to differentiate from each other and have access to different "fileStores". The same client code would also be possible to have on different devices.

The only main improvement that could have been implemented is the client printing what files it has access to and/or the server sending a list of what files it has available in its fileStore if the client requests it to help the user avoid trying to upload or download a file which either doesn't exist or the programmes don't have access to.