UCD School of Electrical, Electronic and Communications Engineering



EEEN20060 Communication Systems

# Assignment 2 Report

**Author:** Paul Doherty **Student Number:**  10387129

**Working with:** Eimear Nolan, Conor Murray, Lorcan O’Toole

### Declaration of Authorship

I declare that all material in this report is my own work, except where there is clear acknowledgement and appropriate reference to the work of others.

Signed: . . . . . . . . . . . . . . . . . . . . . . . . . Date: . . . . . . . . . . . . . . . . . . . . .

## Introduction

The aim of this assignment was to transfer a file over the Internet. We chose option 2 so we aimed to set up a client/server system to transfer a file from one computer to another. This assignment was carried out in a team of four working in groups of two. The following report describes the protocols used by the client to connect with the server, the algorithms implemented to provide the services required by the assignment and the programming problems we faced when carrying-out the assignment.

## TCP

Transmission Control Protocol (TCP) provides the basic protocol that allows the transfer of files from one program on a computer to another program on another computer. TCP is concerned with the accurate transmission files rather than the speed of transmission. It monitors the individual pieces of data that are being sent and ensures they are received as they were intended to. On receiving a file, TCP works the same; it rearranges any data that may have been “confused” in transmission until it is in the correct order and then receives the information as a single file. There are certain specific services of TCP that make it recommended for this assignment:

* The data being transferred is ordered.
* If a stream of bytes is not acknowledged as being transmitted successfully it is automatically retransmitted.
* The flow of data is controlled to avoid congestion.

### Using TCP

Several functions were used in our code to avail of the TCP’s services. Firstly, after initialising winsock, version 2.2, a handle had to be created for the socket. This was done by the following line of code:

SOCKET clientSocket = INVALID\_SOCKET; // handle called clientSocket

Next in our code, the socket was created and assigned to the handle. This was done using the function socket (), which is found in the library winsock.h. This is used to create an end point for communications. The three arguments that were chosen for this function are as follows: the address family AF\_INET was chosen since we are using the internet protocol IPv4, the type SOCK\_STREAM was chosen and protocol IPPROTO\_TCP was chosen since it was decided we would use a transport control protocol for this assignment. It was implemented in our code as follows:

clientSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

The function inet\_addr() was used to convert the IP address string into a 32-bit number. It was implemented in our code as follows:

service.sin\_addr.s\_addr = inet\_addr(serverIP);

The argument serverIP is an array to hold the IP address of the server.

The function htons() was used to convert a 16-bit integer into network form. This function also returns the value in TCP/IP network byte order and is called from the header file winsock.h. In the following case it was implemented to convert the port number (serverport) into network form.

service.sin\_port = htons(serverPort);

The function connect() was used to establish the connection with the specified socket. It was used in our code as follows:

retVal = connect(clientSocket, (SOCKADDR \*) &service, sizeof(service));

This function takes three arguments; the first argument, clientSocket, identifies the unconnected siocket. The second argument, (SOCKADDR \*) &service, is a pointer to the sockaddr structure which specifies the connection that needs to be made. The third argument, sizeof(service), is length of the structure which was pointed to in the second argument, namely, service.

## Our Protocol

Since a connection was established with the server by TCP, the application layer protocol was designed to best use these services. In our protocol the client sends first. After establishing the link with the server the user is prompted with the options of either getting a file from the server or sending a file to the server. If the option of getting a file from the server is chosen, the user is then prompted to input the file name and file type e.g. “file.txt”. Once this step is complete the user is again prompted and asked if they want to specify a new name to save the file locally. If so, they are prompted to input the desired name. If the user chose the option of transmitting a file to the server, they are prompted with similar requests i.e. input file name to upload, and do they wish to specify a new file name for saving the file on the server. If they do wish to change what the file is to be saved as, what is it? If not, continue. If the user wishes to close the connection at any time the symbol “$” must be inputted. After each user input the function gets() is called to read in the string an convert it to a language the computer will recognise. The code that was described above was implemented as follows:

a:

char choice[10];

char newNameYN[10];

char new\_fName[100];

bool wantsNewName = false;

cout << "\nWhat would you like to do?\n\t1. Get a file from the server (G)\n\t2. Place a file on the server (P)\n\t3. Close the connection (Q)\nEnter your choice: (G/P/Q): ";

gets(choice);

//scanf("%c",&choice);

switch(choice[0])

{

case '1':

case 'g':

case 'G':

{

cout << "You have chosen GET" << endl;

cout << "Enter filename, with extension, which you wish to request: ";

gets(fName); // read in the string

cout << "Do you want to specify a new name for saving the file locally? (Y/N): ";

gets(newNameYN);

switch(newNameYN[0])

{

case 'Y':

case 'y':

{

cout << "Enter new local filename: ";

gets(new\_fName);

wantsNewName = true;

break;

}

default:

{

break;

}

}

reqLen=sprintf(request,"G %s",fName);

break;

}

case '2':

case 'p':

case 'P':

{

//cout << "Operation not yet supported!\n\tClosing connection..." << endl;

//reqLen=sprintf(request,"$");

//break;

bool debug=true;

cout << "You have chosen PUT" << endl;

cout << "Enter filename, with extension, which you wish to upload: ";

gets(fName); // read in the string

cout << "Do you want to specify a new name for saving the file on the server? (Y/N): ";

gets(newNameYN);

switch(newNameYN[0])

{

case 'Y':

case 'y':

{

cout << "Enter new name for upload: ";

gets(new\_fName);

reqLen=sprintf(request,"P %s\n",new\_fName);

break;

}

default:

{

reqLen=sprintf(request,"P %s\n",fName);

break;

}

}

}

case '3':

case 'q':

case 'Q':

{

cout << "You have requested to close the connection.\n\tClosing connection..." << endl;

reqLen=sprintf(request,"$");

break;

}

default:

The header sent by the server contains an opening message, a byte count of the number of bytes to be received; this is so client will know when the whole file has been sent. In our code the header was received using the function recv(). It was implemented as follows:

nRx = recv(clientSocket, reply, BLK\_SIZE, 0);

The argument BLK\_SIZE is the number of bytes to be received. nRx will be the number of bytes received. The header is marked by the bytes we received before the “ \* ” was detected. As the header will contain some of the file, the following loop marks the header as separate from the file:

for (k=0; k<nRx; k++)

{

if (reply[k] == '\*')

{

reply[k] = 0;

break;

}

}

The above loop sets the number of bytes wrote to file to zero. After this the amount of bytes being wrote to file is kept track of by byteCount = nWrite;. Once the number of bytes received is equal to the number of bytes sent we know that the file has ended. This was done by the following piece of code:

if (byteCount == filesize)//compares what the server told us was in the file to the recieved bytes

{

printf("Correct number of bytes has been received\n");

printf("%d bytes have been written to file\n", byteCount);

}

//closing file

fclose(fpo);

## Our Program

The software we wrote was designed to allow small files to be transferred between two computers via the internet. Our software is supposed to use an established TCP connection to do this. While our software did contain some functions the majority of the code was written in main(){}. It was designed this way as it was the first time we wrote code like this and it would have been an added complication to use functions. The functions we did use were for a standard error message - void printError(void), a function to send a reply after the client has uploaded a file - void sendReply(SOCKET cSocket, char\* response), and a function to send a file - int sendFile(SOCKET cSocket, char \*fName, bool debug).

### Function printError()

This function is called regularly throughout. It is the standard error that is displayed on the command prompt window when there is a problem with establishing TCP with the server.

**Function sendReply()**

This function is called after the user has inputted their desired requests on uploading a file. It tells the user how many bytes have been sent in the transmission.

retVal = send(cSocket, response, nIn, 0);

printf("Sent %d bytes, waiting for reply...\n", retVal);

This is provided that retVal is not == SOCKET\_ERROR

if( retVal == SOCKET\_ERROR) // check for error

{

printf("\*\*\* Error sending response\n");

printError();

}

### Function sendFile()

This function is called directly after the sendReply() function. It prepares the chosen file for sending and informs the receiving computer of the size of the incoming file:

FILE \*fpi; // file handle for input file

char data[BLK\_SIZE+1]; // array of bytes

int nRead; // number of bytes read

int retCode; // return code from functions

long byteCount = 0; // total number of bytes read

fpi = fopen(fName, "rb"); // open for binary read

int filesize;

fseek(fpi,0,SEEK\_END);

filesize = ftell(fpi);

fseek(fpi,0,SEEK\_SET);

cout << endl << "Filesize = " << filesize << endl;

After doing this it converts the file into blocks and starts sending one block at a time. After file has been successfully sent it closes the file.

**Important sections of main()**

After the declarations the first part of our code is initialising winsock and establishing the TCP connection. This was done by calling functions from the winsock2 header file. Firstly a handle had to be created for the socket and the socket in turn assigned to the handle. This is described in detail in the TCP section of the report. Once this was accomplished an IP address has to be declared for the server which the client wants to connect to. We implemented this by prompting the user to choose their desired server:

printf("Enter IP address of server: ");

scanf("%20s", serverIP); // get IP address as string

The same was done for the port number:

printf("Enter port number: ");

scanf("%d", &serverPort); // get port number as integer

A structure was then built to identify the service required:

sockaddr\_in service;

service.sin\_family = AF\_INET;

service.sin\_addr.s\_addr = inet\_addr(serverIP);

Once this was done, we could try to connect to the service:

retVal = connect(clientSocket, (SOCKADDR \*) &service, sizeof(service));

This again is described in better detail in the TCP section.

When the connection was established, we could start working on the application layer for the program. This section involves the input of the user. The user decides if they want to send or receive a file. It was decided that this section would be best implemented using “switch” decisions. We chose this method because it was the simplest way forward as it involved several different pathways the user could have chosen when using the program. Loops would have been too complicated, confusing and slow.

The code then had to be implemented so that it could actually send and receive the wanted file. This involved language that would be able to open the wanted files, have them ready for sending or receiving, be able distinguish between the header file and actual wanted file and use algorithms to detect when said files were transmitted successfully.

## Testing

This assignment involved a lot of testing. Since there was so much code to be compiled we uncovered some very serious errors in the beginning, in particular the byte counts that kept track of the transmitted files. The testing was made all the more difficult since both the client and server had to be in sync with each other. It was confusing to know at times where the errors were coming from. To combat this precaution was taken at nearly every step of the code to write individual error messages describing which part of the code was causing the errors. For example, when establishing the TCP error checks had to be made regularly:

clientSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

if (clientSocket == INVALID\_SOCKET) // check for error

{

printf("\*\*\* Failed to create socket\n");

printError();

stop = true;

}

else printf("Socket created\n" );

A big question was when to end the while loop ( while (!stop){} ) in which the majority of our program is wrote. Major errors arose when we began testing because we had the functions to close winsock and the TCP connection inside the while loop. This led to an error in the running of the program. Data could only be sent once when the loop existed like this.

Once these problems were solved our program ran as desired. A successfully uploaded file was recognised with the screen output:

cout << "File is availible on server" << endl;

A received file was recognised by the screen output:

printf("Correct number of bytes has been received\n");

printf("%d bytes have been written to file\n", byteCount);

## Conclusion

We have succeeded in implementing our client program. This was not an easy assignment; it involved a lot research into new material and to deeper into programming algorithms. However, it was extremely interesting and exciting. The internet is such an important part of modern and its complexity isn’t even acknowledged by the common user. This assignment required us to learn how computers interact and respond with each other. I feel this assignment has helped me understand the workings of basic file sharing protocols and I am looking forward to increase this understanding.