UCD School of Electrical, Electronic and Communications Engineering

EEEN20060 Communication Systems

TCP Report

Author: Fergal Lonergan **Student Number:** 13456938

Working with: Conor Burke on Server program

Collaborating with : Darren Coughlan & Sam Luby who designed the Client Program

Declaration of Authorship

I declare that the work described in this report was done by the people named above, and that the description and comments in this report are my own work, except where otherwise acknowledged. I have read and understand the consequences of plagiarism as discussed in the EECE School Policy on Plagiarism, the UCD Plagiarism Policy and the UCD Briefing Document on Academic Integrity and Plagiarism. I also understand the definition of plagiarism.

Signed: Fergal Lonergan **Date:** 28/04/15

Introduction

This assignment was a four week assignment as part of the UCD module EEEN20060 (Communications Systems). This assignment was then subdivided into two separate options:
Option 1: In a group of two you are expected to use the http protocol and design a client program that can then be used to download files from any webserver over the internet.
Option 2: In this option you were required to design a server and client program where the client was able to download and upload files to the server, along a joint network using an Ethernet cable, or over the internet using IP addresses, who would then handle them accordingly. You were also tasked with writing your own application layer protocol so that the server and client could communicate with each other and perform the desired tasks whilst only receiving user input from the client side. For this option you were allowed work in two groups of two, one group designing the server program and the other designing the client program.
We decided to choose Option 2 where Conor and I would design the server whilst Darren and Sam designed the client.
Our transfer-layer software and any lower level software needed to establish a connection and transfer the bytes were provided, including our transmission control protocol (TCP) functions and Winsock functions.

Contents

TCP

TCP is widely used over the internet as a transfer layer protocol. It is a core protocol of the Internet Protocol Suite and was originally designed to complement the internet

protocol (IP). Therefore as a whole the entire suite is often referred to as TCP/IP. TCP is designed to provide reliable, ordered and error checked data from one computer to another along either an Ethernet connected network of computers of over the internet which may not necessarily be reliable.
TCP achieves this by opening a connection between two computers that wish to communicate before error checking the data and setting it in octets along the connection. This is similar to what a link-layer protocol would do along a single link, the implementation of which we discussed in our last assignment.
The TCP takes care of all of the necessary processes discreetly away from the view of the user, apart from establishing and closing the connection.
TCP deals with a stream of bytes not blocks of any particular size. As a result of this the server, during download for example, may send a file of 5000 bytes in blocks of 100 bytes as we did in our program, and the client receive them in blocks of 50 and no error in transmission will occur. TCP has been designed to handle each byte individually and as a result is able to ensure that the bytes will be received in the correct sequence and un-corrupted. The TCP varies the size of the blocks in order to ensure that they do not exceed the maximum block size of the part of the network for which they are travelling on, but also to improve the efficiency of transmission.

Winsock

The Winsock or Windows Sockets API (WSA) defines a standard interface between a Windows TCP/IP client application and the underlying client application. Including the library in our program provides our program with functions which allow us to use the TCP, with the basic interface between them being referred to as a socket.
For our program the code needed to open and close the sockets needed to transfer the data correctly was provided however we needed to call the socket anytime we wished to transfer data to the other computer.

Using Winsock

There are many functions in winsock I will give brief explanations of them here.

This is the code needed to initialise winsock in your program:

```
    WSADATA wsaData;  // create structure to hold winsock data
    ret = WSAStartup(MAKEWORD(2,2), &wsaData);  // initialise Winsock
```

This used to declare the socket:

```
    SOCKET clientSocket = INVALID_SOCKET;  //create handle called
clientSocket

    clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); //
create the socket
    if (clientSocket == INVALID_SOCKET) - - - // deal with error
    else printf("Socket created\n" );  // otherwise success
```

We then have the clean up function which should be put before the end of your program:

```
  ret = WSACleanup();
```

Server socket

```
    SOCKET serverSocket = INVALID_SOCKET;
    serverSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (serverSocket == INVALID_SOCKET) - - - // deal with error

    ret = bind(serverSocket, (SOCKADDR *) &service, sizeof(service));
    if( ret == SOCKET_ERROR) - - -  // deal with error

    ret = listen(serverSocket, 2);  // allow
    if( ret == SOCKET_ERROR)  - - -  // deal with error
```

The code which allows the server to accept a connection:

```
   SOCKET cSocket = INVALID_SOCKET;  // create another socket for the
client connection

    cSocket = accept(serverSocket, NULL, NULL );  // wait for
connection
    if( cSocket == INVALID_SOCKET)  - - - // deal with error
```

The function used for sending the bytes along the network using the socket:

```
    ret = send(clientSocket, data, nByte, 0); // send nByte bytes from
data array
    if( ret == SOCKET_ERROR) - - - // deal with error
```

The function to receive the bytes from in our case the client:

```
   ret = recv(clientSocket, data, 100, 0); // get 100 bytes, put in
data array
    if( ret < 0) - - - // deal with error
    else if (ret == 0) - - - // connection has been closed
    else - - - // process received bytes
```

And finally the code needed to shut down the connection:

```
    ret = shutdown(clientSocket, SD_SEND);  // close sending side of
connection first
    ret = closesocket(clientSocket);  // then close connection
completely
```

All this code was given to us beforehand and deals with the TCP allowing us to focus on designing our application_layer protocol.

Our Protocol

We decided as a group to initially make a really simple, fool-proof protocol that involved a lot of error checking to ensure that our design worked correctly. Once we got the design working to an extent that we were happy with that we would then add in extra details and functionality to reduce errors etc.
The protocol began with the client sending us a request in the form of a character array which started with either a ”D”, “U” or “E”, depending on whether they wished to download, upload or exit the program respectively. Also in this request would be the name of the file that they wished to download or upload, the end of which was marked by an “@” and, in the case of upload the number of bytes, the end of which was also marked by an “@”, which would later be converted to an integer.
In the case of download we would first check that the file was in our directory and if it was we would send a small acknowledgement stating that we found the file and were going to start sending. If it wasn’t we would send a negative acknowledgement.
For upload we would create a file of the specified filename and ask then send a small acknowledgement, which would be a “y” if we were ready to receive and an “n” if we were not willing to receive the file as an error has occurred. If we were willing to receive the file the client would then begin to send it to us while we printed it to the array.
To exit the program we changed our boundary conditions and shut down the connection.
Later on we decided to introduce a loop that allowed us to repeat the process and a progress bar on which showed the user how far along the transmission was.
If the connection was to be closed we would first close the transmission socket before closing the connection socket and then exiting the program.

Our Program

For our programs we decided to first implement a fool proof design first before expanding to include additional features such as the progress bar and the repeat process. It initially establishes a secure connection with the client before by using the Winsock functions described earlier and then proceeds to enter our do loop where the majority of our program lies.
We also have three different functions a receive_File used to manage most of the operation of the upload mode of the program, a send_File function which is used to manage most of the operation of our download mode of the program and final a printError function used to print meaningful errors to the screen in case an error has occurred in transmission.
There are also three modes to our program the download mode, used for when the client wishes to download a file from the server, the upload mode, used for when a client wants to upload a file to the server, and finally the exit mode, used for when the client wishes to close the connection and exit the progam carefully.

This program is a built to act as a server where it takes in information from the client and manipulates it in order to upload or download files over the internet or on an Ethernet cable via a computer network. Our protocol was for us to first establish a secure connection with the client. Then we wait for a request from the client in the form of a character array and break it up into the relevant components we need for our report.

Download

If we are doing download we expected to first receive a D, then the file name, i.e.(filename.txt) and then an @ symbol which marked the end of the filename. We then extracted the mode, i.e. first letter, the file name i.e. filename.txt, and attributed them to a character mode and character array file_name respectively. We then declared a file pointer of type FILE called fpi for input file and attempt to open the file for binary reading and checking for errors to ensure the file is there, if it is not or a transmission error occurs when receiving the request these are printed to the screen.

send_File

Assuming it is we call the send_File function which takes the following arguments: the socket which the data will be sent on cSocket, a file pointer to the input file fpi and a character file_found which will equal to "y" if the file has been found or "n" if it has not. It prints to tell the user where it is in the program and then calculates the size of the file to be sent, or downloaded, using the fseek and ftell functions and attributing the value to nBytes, the number of bytes in the file. Whilst it does this it checks for errors and prints them to the screen should one occur.
It then sends a small acknowledgement to the client telling them that we have found the file, that it is nBytes long and that we are about to send the data. it then sends the data checking to make sure an error hasn't occurred whilst printing a progress report to the screen so the user can see how much of the file has been transferred to the client and if the file transmits correctly it closes the file and then prints that we have had a successful transmission and the file is sent and exits the function.

This snippet of code is how we calculate the value of nBytes, the number of bytes in the file. At this stage we have ensured that the file has been found and opened, so we seek to the end of the fila and attribute this position to nBytes as an integer value. We then return to the start of the file in order to transmit the data correctly.

```
retVal = fseek(fpi, 0, SEEK_END);  // set current position to end of file
    if (retVal != 0)  // there was an error print it and close file
    {
        printf("\n*********************************************************\n");
        perror("Error in fseek");
        printf("errno = %d\n", errno);
        fclose (fpi);
        printf("\n*********************************************************\n");
        return 2;
    }
    nBytes = ftell(fpi);  // find out what current position is which is size of
file and set it to nBytes
    printf("File size is %ld bytes", nBytes);  // print it

    retVal = fseek(fpi, 0, SEEK_SET);   // set current position to start of file
    if (retVal != 0)  // if there was an error print it and close input file
    {
        printf("\n---------------------------------------------------------\n");
        perror("Error in fseek");
        printf("errno = %d\n", errno);
        fclose (fpi);
        printf("\n---------------------------------------------------------\n");
        return 3;
    }
```

The following snippet of code describes how we transfer the data and also print our progress bar explaining how much of the file has been transmitted. We continue to send the data as long as the end of the file is not reached and we haven't sent more data than the file size. Whilst doing this we calculate the percentage of the file transmitted and if this is above the next multiple of 10 below 100 the progress bar is updated to include illustrate how far along in the transmission stage we are.

```
while(!feof(fpi)&&SentBytes<nBytes)
    {
        BytesSending = (int) fread(data, 1, BLK_SIZE, fpi); // bytes left to be
sent

        retVal = send(cSocket, data, BytesSending, 0);  // send bytes to TCP
        SentBytes+=BytesSending; // increase bytes sent to date
        printf("----------------------------------------------------------------
");
        printf("\n\t\t       PROGRESS BAR \n");
        printf("[");

        while(SentBytes<nBytes)   //while the number of bytes sent is less than
the total number of bytes in the file
        {
            //fread attempts to read in up to 100 bytes from the file . retSend
is how many bytes are read.
            ret = (int) fread(data, 1, BLK_SIZE, fpi);
            if (ferror(fpi))  // check for error
            {
                printError();
                fclose(fpi);   // close input file
                return 3;
            }
            else
            {
                retVal = send(cSocket, data, ret, 0); //sends retVal amount of
bytes to the client each time.
                SentBytes+=retVal;    //incrementing the amount of bytes sent to
client to date

                percentage = (SentBytes/(double)nBytes)*100; // calculating the
percentage
                if(percentage>threshold) // if percentage increase by 10%
another section of progress bar printed
                {
                    printf("===%d%%",threshold);
                    threshold+=10;
                }
            }
        }
}
        printf("===100%%]\n"); // end of progress bar
        printf("Download has worked correctly\n");
```

Upload

If the client wishes to upload we expect to receive a request in the form Ufilename.txt@1000000@ where U denotes the mode to upload, filename.txt is the

name of the file, and 1000000 is the size of the file. The "@ "symbols are again used as end markers for the different character arrays in our request array. Similarly to download we retrieve the mode, file name, and size of bytes and attribute them to a char mode, char array file_name and a char array number, which is then converted to an int using the atoi function, respectively. Meanwhile we have created a file pointer fpo, denoting output file, and opened it checking for errors before sending an acknowledgement to the client saying that we are ready to receive their upload. We then call the function receive file that will take care of the main file handling and receiving of bytes for the upload mode.

receive_File

The receive_File function takes in the arguments cSocket of type socket, the file pointer fpo, which denotes the output file, of type FILE, a character array called request which received_data which holds the received data from the client and an integer value nByte, which is the length of the file in bytes. It prints to let the user know where in the program it is and also that it is intending to receive bytes from the client. It then receives the data from the client. It looks for these in blocks of 100 (BLK_SIZE) however we know from our discussion earlier that it may not receive the bytes like this, however the recv functions sorts the incoming bytes into this block size.
It then checks to see that there have been no errors with the connection. Once it has received the bytes and there was no error it then prints the received bytes into the desired file location before getting more bytes. it also prints a progress bar so that the user knows how far along the download is if the upload has worked correctly it prints to the screen to let the user know it has succeeded and then exits the function after closing the file. Once a successful transmission has occurred we exit the loop and return to main where the program loops and asks if the client would like to download upload or exit again. As well as the download or upload option there is an exit option that lets you exit from the program safely.

This code snippet I from our receive_File function and explains how we receive bytes, write them to the file, check for errors and print our progress bar.

```
nRx = recv(cSocket, received_data, BLK_SIZE,0); // setting nRx to the number of
total bytes received

        if( nRx == SOCKET_ERROR)  // check for error
        {

printf("\n*******************************************************\n");
            printf("Problem receiving\n");
            printError();

printf("\n*******************************************************\n");
            stop = 1;  // exit the loop if problem
        }
        else if  (nRx == 0)  // connection closed
        {

printf("\n*******************************************************\n");
            printf("nRx has reached 0\n"); //test
            printf("Connection closed by server");

printf("\n*******************************************************\n");
            stop = 1;
```

```
        }
        else
        {
            // keeping track of amount of bytes written to output file
            data_written = fwrite(received_data, 1, nRx, fpo);
            bytes_rec += data_written; // increasing total bytes received by the
bytes written

            percentage = (bytes_rec/(double)nByte)*100; // calculating
percentage
            if(percentage>threshold) // if precentage goes up by another 10%
progress bar shows this
            {
                printf("===%d%%",threshold);
                threshold+=10;
            }


        }// to exit the while loop when the entire file has been read/sent


        while(bytes_rec<nRx)   //while the number of bytes sent is less than the
total number of bytes in the file
        {
            //fread attempts to read in up to 100 bytes from the file . retSend
is how many bytes are read.
            ret = (int) fread(received_data, 1, 100,fpo);

            if (ferror(fpo))  // check for error
            {
                printError();
                fclose(fpo);   // close input file
                return 3;
            }

        }
    }

    printf("===100%%]\n"); // end of progress bar
```

Exit

The final mode is exit. The program will exit our main loop once and “E” is read from the user and close the connection correctly using the TCP functions. Otherwise if it again receives a “D” or a “U” it will continues to loop around until an error occurs or the exit mode has been called.

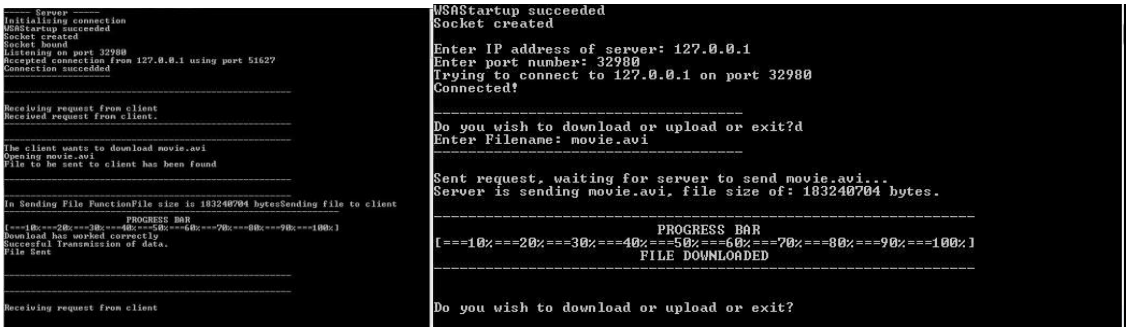
printError

Our final function is used for error detection. Print error receives no arguments nor does it return any, it simply prints the last error that occurred to the screen using the functions provided by the winsock library.

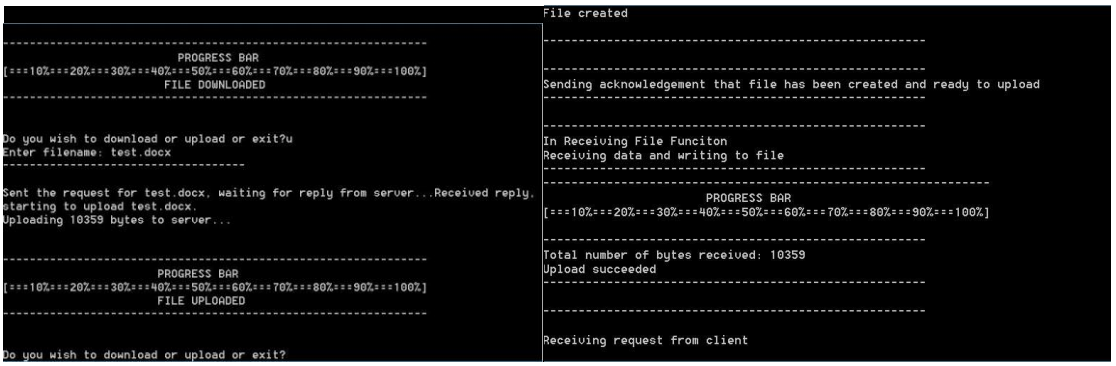
```
    // Initialise winsock, version 2.2, giving pointer to data structure
    retVal = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (retVal != 0)  // check for error
    {
        printf("*** WSAStartup failed: %d\n", retVal);
        printError();
        return 1;
```

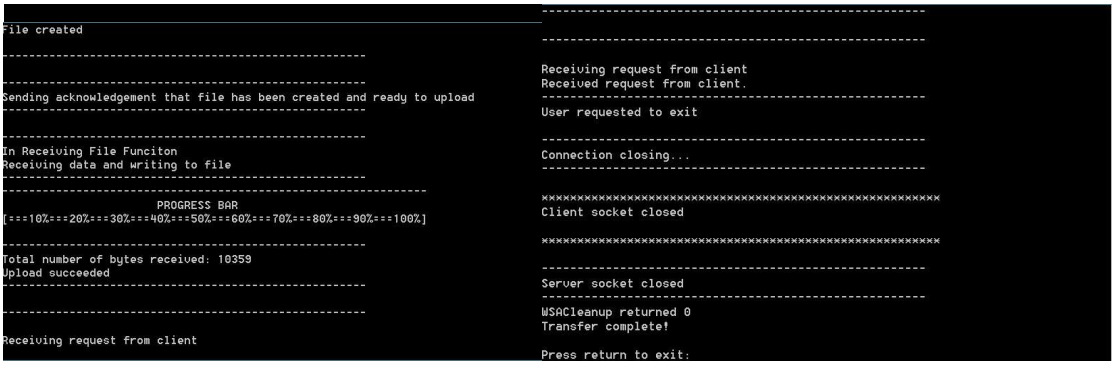
```
}
printf("WSAStartup succeeded\n" );
```

Testing



.





Terminal outputs of .avi file from Sam Luby

Repeat test from Darren Coughlan

We tested a few files transferring between our own laptops and everything worked perfectly, irrespective of the file type etc. You also have the option to upload and download continually until you exit the program or run into a transmission or connection error during the same running. Our code worked exactly as desired when completing these tasks. We tried to run it over the internet as well to test that it worked also but our laptops actively refused transmission and we were not sure how to open the sockets on our firewall and disable our sockets correctly so we ceased trying to test it this way.

Conclusion

We have succeeded in designing a simple server that can be used by a client, using a program that adheres to our protocol, to download files from or upload files to our server on a connected network of computers or the internet.
Had we had more time we would have liked to include the CreateDirectory tool to create a separate file location for the client, to ensure that if the client uploads a file to our server with the same name as a file already in our directory that it would not overwrite it. Using this method the client would be able to upload and download as they pleased without affecting our program whatsoever.