**File Transfer Protocol (FTP): Project Report**

Class: CS 4390.002

Name: *Edward Hong*

NetID: *exh150330*

There were two implementations of a File Transfer Protocol using TCP and UDP protocols for transportation of the packets. This first section details the design and operation of the TCP implementation of a File Transfer Protocol(FTP).

The TCP implementation of the FTP follows the Client-Server model. In essence, the server is a passive entity which will only serve responses from the client, and the client will initiate any and all requests. As such, the FTP is designed with the client side handling most metadata regarding control and other information such as filename and path. The general procedure of the source code is: on both the client/server side we open our sockets using the specified port number, in this case port 34197 was used. On the server side, we will wait and continually block until a client has connected. Once connection has been established, we utilize the buffers as input and output data streams, reading and writing to them accordingly. A number of control messages are also used in order to handle server request logic, these are: “SEND”, “RECEIVE”, and “FIN”.

While the server is running, it periodically checks the buffer for any data. The first thing that the server should read is one of the three control messages, thus acting in a way as a Start-Of-File message. If the server receives a “SEND” message, this means the client is requesting that the server *send* to it a file. On the other hand, if the server receives a “RECEIVE” message, then the client’s request was for the server to instead *receive* the incoming file. A message of “FIN” means the client wishes to end the connection with the server, and will initiate both the server and client in cleaning up the connection and freeing resources.

On the client side, the application is entirely driven by user input - the user is able to select the functions available from the command line interface’s menu. To handle the client sending a file to the server, the user will input the desired filepath and name. After sending the control message, the client then sends the filename so that the server both knows where to store the file and the file type. A File Input stream is used to read the bytes from this file, which are then written to the buffer. The server will then receive these bytes in it’s own buffer and begin to write these bytes to a new file.

To handle the client requesting a file from the server, the user will again input the desired filepath and name. As before, the client sends the control message and filename metadata to the server, who will then respond immediately with the file data. A generic Data Input stream is used to read from the buffer, while a File Output stream is used to write the bytes to the new file.

The second implementation of the File Transfer Protocol utilized the User Datagram Protocol (UDP) as opposed to TCP. Because of this the application is naturally going to be much more complex as we no longer have the features and guarantees of using TCP.

In this implementation, the server solely acts as the SENDER, whilst the client acts as the RECEIVER. In an actual application of FTP using this implementation, the idea would be to have both users have instances of both a Server and a Client, as opposed to having one individual user be the Server and other individuals be the Clients.

The main concern when implementing FTP using Datagrams, is the limited amount of data which we may put in a UDP packet (64kB). Because of this, large files require multiple packets be sent to the receiver, and the receiver must then reassemble the packets accordingly. In addition, because UDP does not guarantee in-order arrival, there must additionally be some metadata within the packets sent to encode a sequence number for the packets. For these reasons, it is not really advisable to use UDP to implement such a protocol (FTP), however it is nonetheless possible.

To allow for support of arbitrary file size, the following design decisions were made:

* Only one file may be sent from the Server to the Client at a time.
* Each packet being sent by the server would have its first byte of data be a sequence number.
  + A sequence number of 0 will denote a metadata or control packet. This packet contains the file size, the number of packets which will be sent by the server, and the filepath of the file to be sent.
  + All other packets with sequence number > 0 will be data packets. Every byte but the sequence number is data.
* The receiver continually waits and grabs any packets which it receives. It will keep a count of the packet it receives. On every cycle, it will check to see if one of the packets it receives was the metadata packet with seqNum = 0.
* Once the client has the metadata, it will know how many packets it expects to receive from the server. This maximum number of packets is what the client uses to tell when it’s received all data packets. Until the client has received the metadata packet, it assumes the maximum number of packets is infinite.
* Every packet the client receives will be indexed into a 2-Dimensional array, with the rows indexing the sequence number, and the columns being the data itself. All received packets will be stored into the 2D array based on the index of their sequence number - in this way once our 2D array is completely full it will already have all the bytes in order.
* The last data packet in the sequence may be padded with extra 0’s. For instance, if sending a file with 64,001 bytes, the LAST packet will only contain 1 real data byte. The client determines the amount of remaining bytes in the last packet by taking the modulo function of the file’s size and 64000.

Whilst implementing this design of the UDP FTP, I had mistakenly implemented it in the context that the UDP packet’s data resides in the first bytes from 0 up to 64,000. I realized that region of course is the destination/source address and port. Due to time constraints, I decided then instead to scrap the code and instead submit a *working* FTP but with the caveat that it will not support file sizes of greater than 64kB.