Final Team Project

COEN 366

Network Application Development Project A : SImple File Transfer Service

Students:

Anna Bui 40212221

Joseph Regipaultren 40156682

We certify that this submission is my original work and meets the Faculty's Expectations of Originality

Table of content

INTRODUCTION

The main purpose of our client/server program is to give the client the ability to download files from the server's directory to the client directory as well as upload files from the client directory to the server directory. In addition, the transfer should be able to work with any type; such as txt, doc, jpg. The FTP should contain two types of protocol (UDP and TCP). An User Datagram Protocol (UDP) is a communication protocol that doesn't establish a direct connection between the client and server or doesn't ensure that the data arrives intact/in order. Contrary, a Transmission Control Protocol (TCP) is a communication protocol that is connection oriented; which ensures reliability (using acknowledgement mechanism), ordered delivery and flow control. Therefore, in a FTP, the client and the server are expected to interact as follows: First, the Client first connects to the server. If the connection is successful, the client program can start transferring files to the server using the put command and retrieve files from the server using the get command. Note that the client can also change the names of the files in the server's machine.The client program then rests the user command, parses it, and executes it. The server should then reply with the appropriate response message, which the client should be able to handle.When the client quits; the communication with the server closes. However, note that the server should keep listening for new connection requests from other clients.Hence, our client/server program aims to facilitate file transfer between client and server directories with different commands through a UDP or TCP connection.

HIGH LEVEL IMPLEMENTATION OF SERVER AND CLIENT

Our implementation consists of actively letting the server listen for incoming connections from the client; either asking for a TCP connection or a UDP connection. Notice that the client should input the correct ip address and the correct port number indicated in the server side for successful connection.

Server.py

As mentioned, since our server is continuously listening for both types of protocols; 2 functions were created to listen to incoming connections: startingTCPserver(ip_address,port_number) and startingUDPserver(ip_address,port_number). The main function will then start the two server processes in parallel using threads; where one handles TCP connections, and the other UDP connections that will run on the local machine at (127.0.0.1:12000). Once the client's sends a request for connection, depending on the protocol chosen; one of the functions handling the servers responses will be triggered responseToTCPClient(TCPsocket) and respondeToUDPClient(UDPsocket). Both are implemented similarly except for the part where it's receiving requests from the client and sending responses to the client. They are implemented into a while loop as it will keep receiving requests from the client unless the client decides to terminate the connection.

Put response: The put function is used for transferring a client file to the server side; To do so, when receiving the put opcode, it extracts the file name and the size from the request and then reads the file data. If the put was successful, the server constructs a response message with the successful res-code and sends it back to the client using the chosen protocol. If not, the unsuccessful res-code will be sent to the client.

Get response: The get function works similarly to the put; except it's reversed. In other words, it transfers a server's file to the client file. To do so, when it receives the get opcode, it decodes the file name and file size from the requests. If the file exists and is within the limits, the server will read the file and respond back with the file's content. If successful, it will send back a success res-code to the client using the chosen protocol. If not, the unsuccessful res-code will be sent to the client.

Change response: This command allows the user to change the name of an existing file on the server side. When the server receives the opcode for this command, it decodes the current name file and the new file name from the client's request. It will then rename the file. If successful the server sends a successful res-code to the client via the chosen protocol.. If not, the server will send an unsuccessful res-code to the client

Summary response: This command allows the user to obtain the max,min, and average value of a series of numbers on a file of the server. When the server receives the summary op-code, it will decode the file name and size (to verify that it is within the size limit). Knowing the file name, the server will open and iterate through each line to obtain the numbers (The list of numbers should be separated by a space). Then, it will create a new summary file with those values written inside. If it was a success, the server sends a response back to the client indicating a successful res-code via the chosen protocol. If the file does not contain any value, the server will still send a successful code to indicate that the command was received; but will output on its log that the file does not contain any number. But, if the file does not exist it will send an error res-code to the client.

Help response: This command allows the user to have access to the list of existing commands. When the server receives a request opcode, it will prepare a message containing the list of available commands; which will be encoded into bytes combined with the res-code and the length of the message (formatted in 1 byte) via the chosen protocol.

Finally, when an unknown request was sent from the client, it will respond back with the corresponding res-code via the chosen protocol.Please refer to the Format of the Response messages table to see how the responses were converted into bytes and sent depending on the request.

Client.py

The client side will have two functions to start a connection chosen by the user : tcp_connection(ip_address,port_number) and udp_connection(ip_address,port_number)

The user will then be prompted to choose the protocol and input the IP and Port and the debug flag. Based on the user's input (1= TCP, 2 = UDP), the corresponding function to start a connection will start.Both connections are implemented similarly; only the way it sends commands request to the server and receives the responses are different according to the protocol type. They are implemented into a while loop as it will keep sending the client requests unless the client decides to terminate the connection.To facilitate the decoding/process of the response received by the server; a function convertServerResponse(response) was implemented; where the response will return the opcode and the filename length.

Put command: When the user input the “put” command, the system verifies the command by separating the inputs (where the first is the command type, the second part the name of the file) If the file exists in the client side, respects the name length, and respects the size limits, the client will construct a requests the includes the opcode , filename length (5 bits), and the file size (in 4 bytes), as well as the file content. The request will then be sent using the protocol chosen. After sending it, the client waits for a response from the server; which will indicate if it was a success or a failure by decoding the received response.

Get command: The client will send a request: “get” command, which the system verifies and separates accordingly to obtain the request command and the file name and also checks if it is within the required limits. It will then construct a request according to the format (in the request format message table) and send it to the server via the chosen protocol. It will then wait for a response, decodes, and verifies the received res-code; which will then indicate the client if the it was successfully processed or not

Change command: The client will send a request for the “change” command, which the system verifies and separates accordingly to obtain the request command, the file name, and the old filename as well as verifying the limits. If the limits are respected, the request will be sent (formatted) to the server. After sending, the client will wait for a response from the server and decode it; which is either a res-code for successfully processing, or a res-code for unsuccessfully processing it.

Summary command: The client will send a request for the “summary” command, which the system verifies and separates accordingly to obtain the request command, the file name, It will then verify if the file is within the limits, and if so it will reconstruct the summary request opcode and file name to send the requests to the server using the chosen protocol. It will then wait for a response from the server. The client will then receive the response and process it. It will then verify the received res-code to then tell the client if the request was successful or unsuccessful.

Help command: When the system recognizes the help command, the client reconstructs the requests to then send it to the server using the chosen protocol. This request will not include additional data. After sending it, the client waits to receive the response from the server. When receiving the response, it extracts it and decodes the message sent from the server, based on the provided length in the response. If res-code is successful, the client will then see the displayed message sent from the server.

Bye command: When the system recognizes the bye command, it will simply close the client socket; which will then close the connection.

Please refer to the Format of the Request messages table to see how the requests were converted into bytes and sent to the server. A print_debug function was used to print and inform the user whenever each command is executed; if the debug_info is false, the function prints the message regardless the debug_flag value; and if the debug_info is true, it checkers the debug_flag- and if it’s true, it prints the messages.

WIRESHARK

For TCP:

1) When doing the help command, the sequence Number of the TCP SYN segment that is used to initiate the TCP connection between the client and server is 0.The sequence numbers of the two first segments is 1 and 2

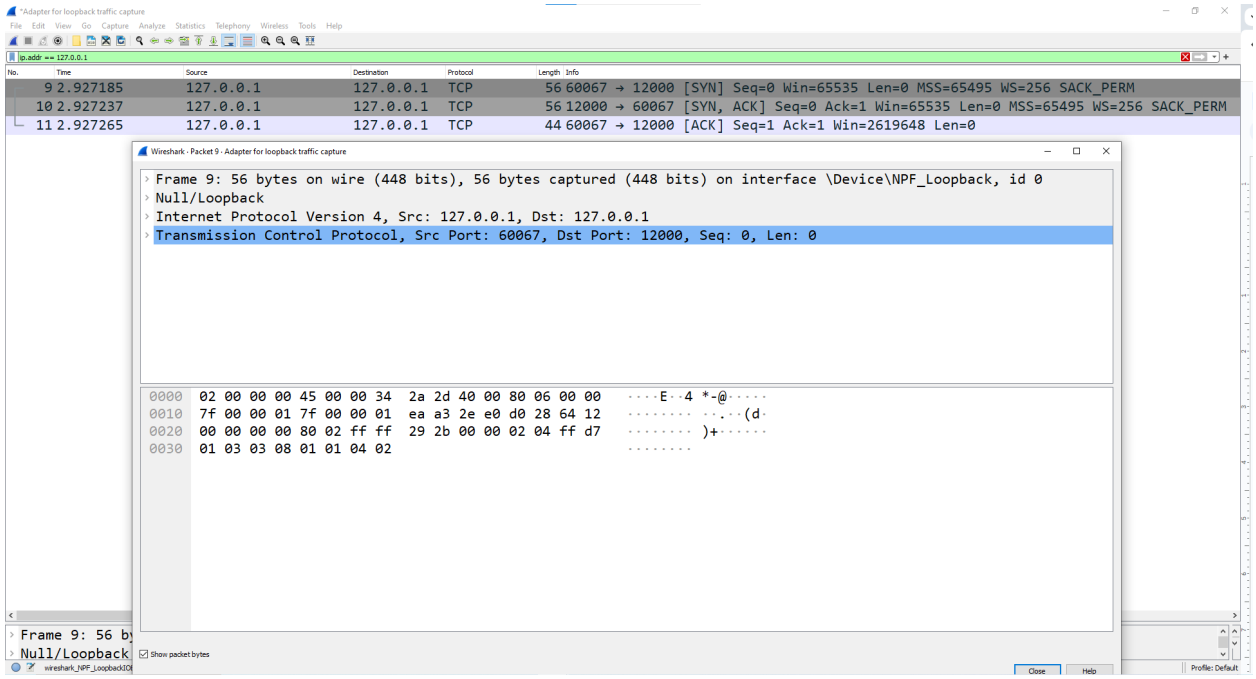

Figure 1

2) The time when each segment was sent was 14.032473 and 14.033448.
The time when ACK was received: 14.032725 and 14.033589.

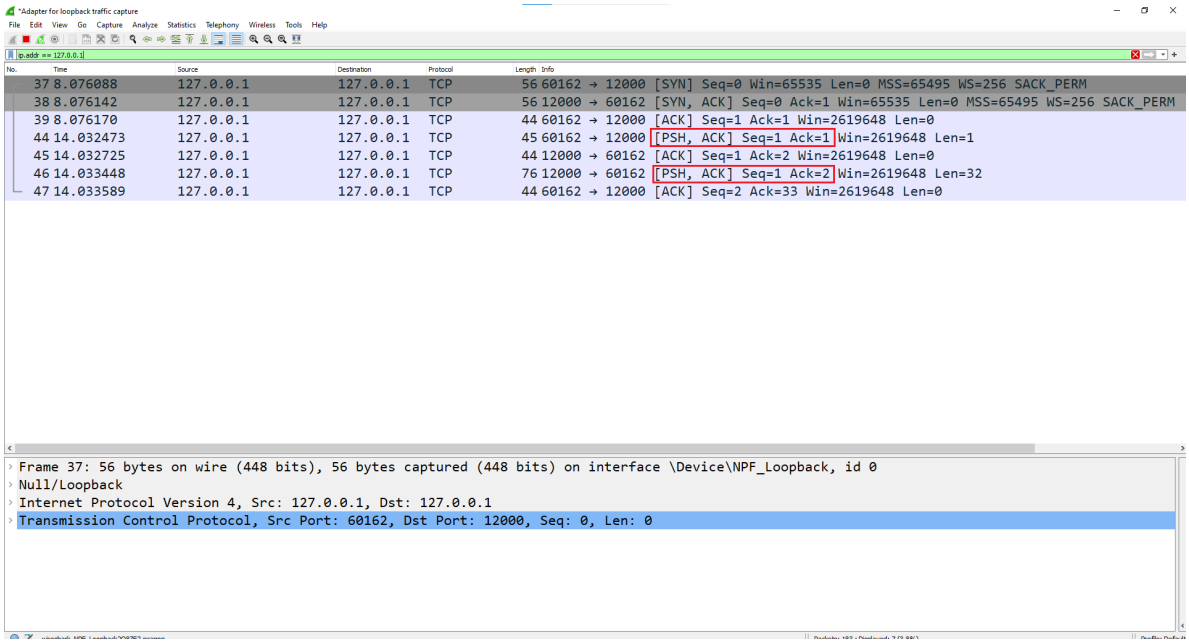

Figure 2

The RTT value for each of the two segment: 0.000252 and 0.000141

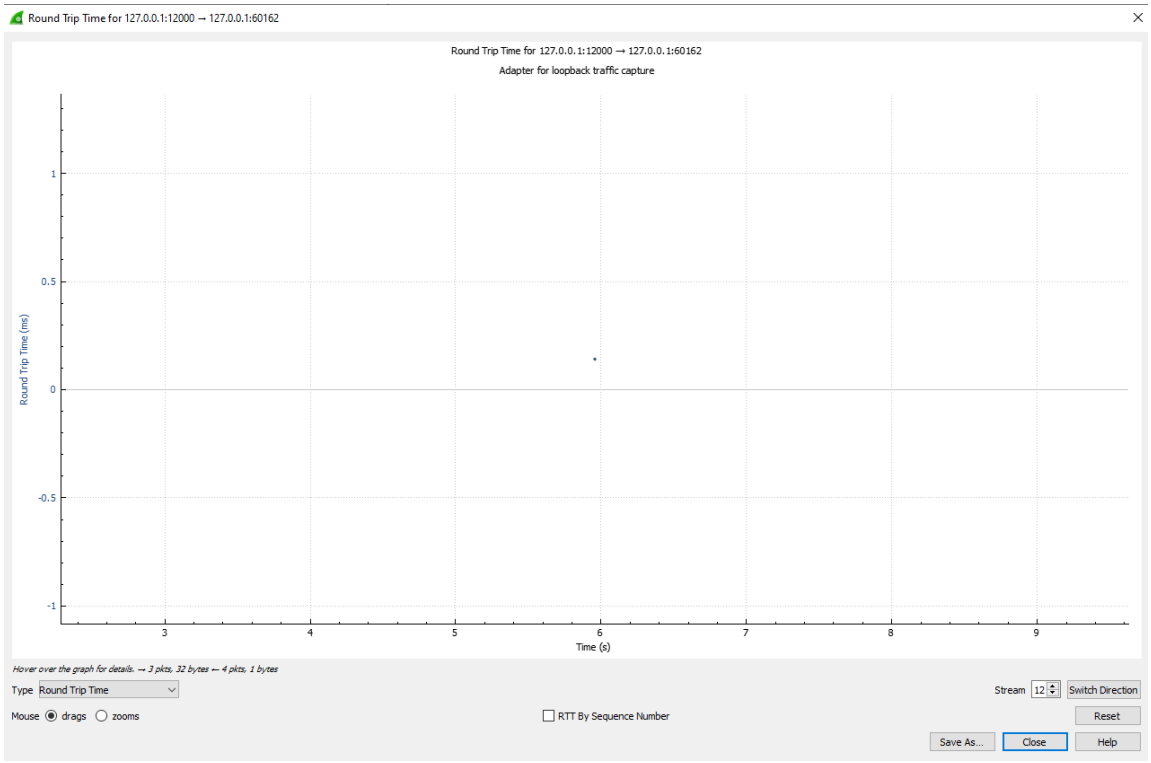

Figure 3

3) The length of each of the first six TCP segments when doing help and summary command

    1. Packet 32, Length = 1
    2. Packet 34, Length= 32
    3. Packet 216, Length = 11
    4. Packet 218, Length =15
    5. Packet 531, Length = 19
    6. Packet 533, Length = 60

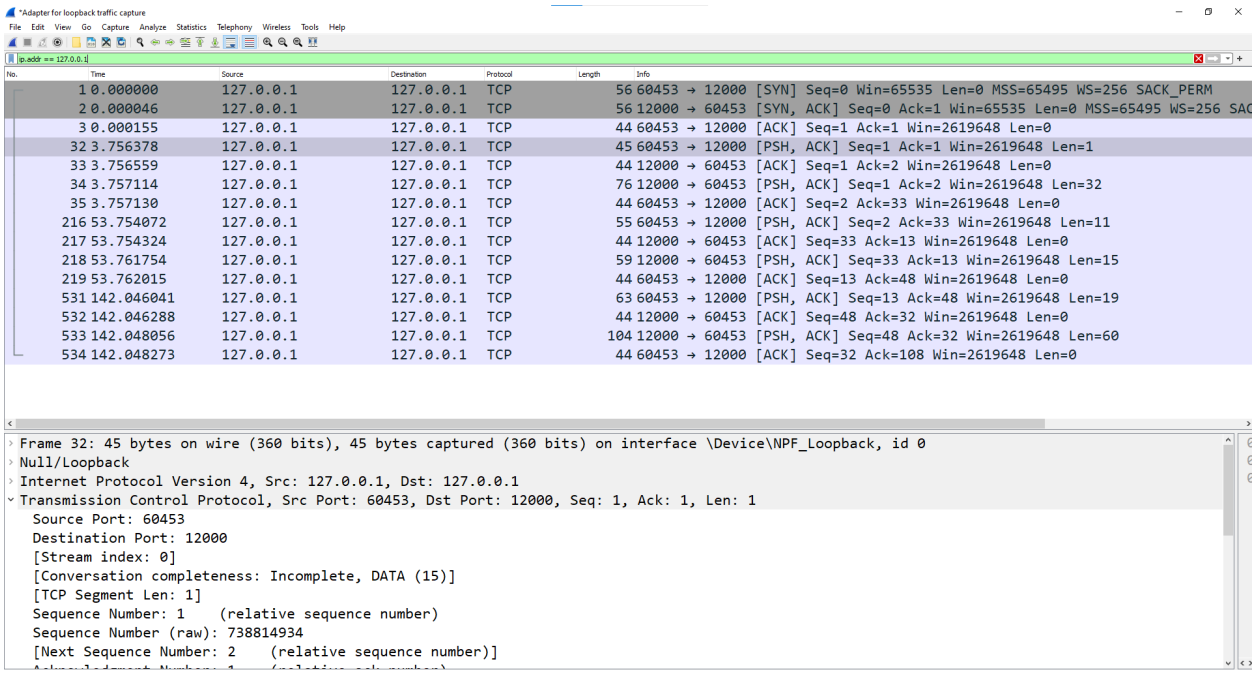

Figure 4

4) The minimum available buffer space advertised at the received is 2619648 bytes shown in the figure above. The sender is continuously sending data without being slowed down by the receiver's buffer capacity, leading to an increase in the buffer size.

5) There are no retransmitted segments in the trace file. It can be verified in the time sequence graph for all sequences shown below:

6) There are no retransmitted segments in the trace file. It can be verified in the time sequence graph for all sequences shown below (stevens)

7) The receiver typically acknowledges 13 bytes data in an ACK when doing the summary command.

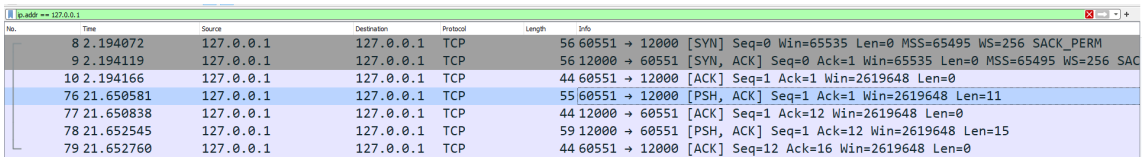

Figure 5

Additionally, The receiver typically acknowledges 32 bytes data in an ACK when doing the help command.

472	123.187437	127.0.0.1	127.0.0.1	TCP	45	60551 → 12000 [PSH, ACK] Seq=12 Ack=16 Win=2619648 Len=1
473	123.187712	127.0.0.1	127.0.0.1	TCP	44	12000 → 60551 [ACK] Seq=16 Ack=13 Win=2619648 Len=0
474	123.188409	127.0.0.1	127.0.0.1	TCP	76	12000 → 60551 [PSH, ACK] Seq=16 Ack=13 Win=2619648 Len=32
475	123.188423	127.0.0.1	127.0.0.1	TCP	44	60551 → 12000 [ACK] Seq=13 Ack=48 Win=2619648 Len=0

Figure 6

8) The throughput for the TCP connections when doing summary, help and put command

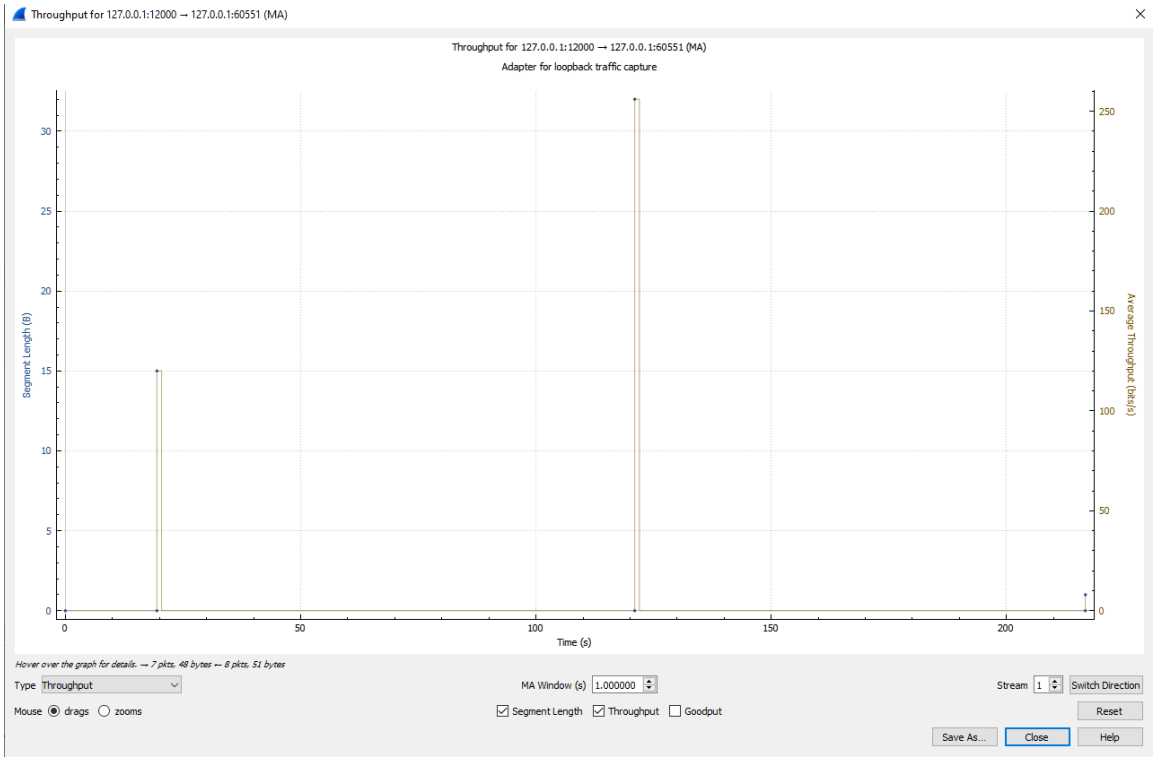

Figure 7

9) Time Sequence Graph (Stevens) plotting when doing summary, help and put command

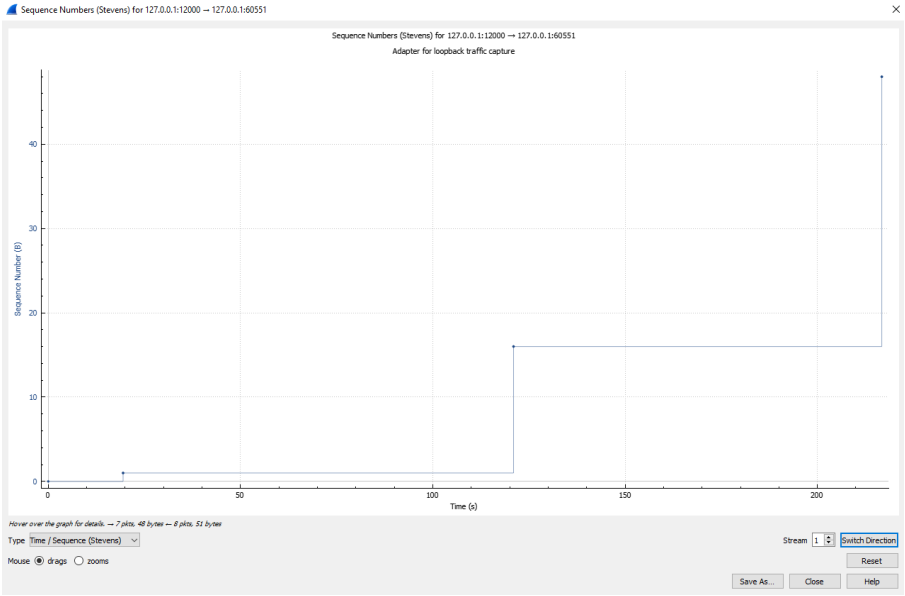

Figure 8

for UDP:

1) There are 4 fields in the UDP header: Source port, Destination Port, Length, and Checksum

2) The length of each of the UDP header fields are 8, because Source port (2 bytes), Destination Port (2 bytes), Length(2 bytes), and Checksum (2 bytes)

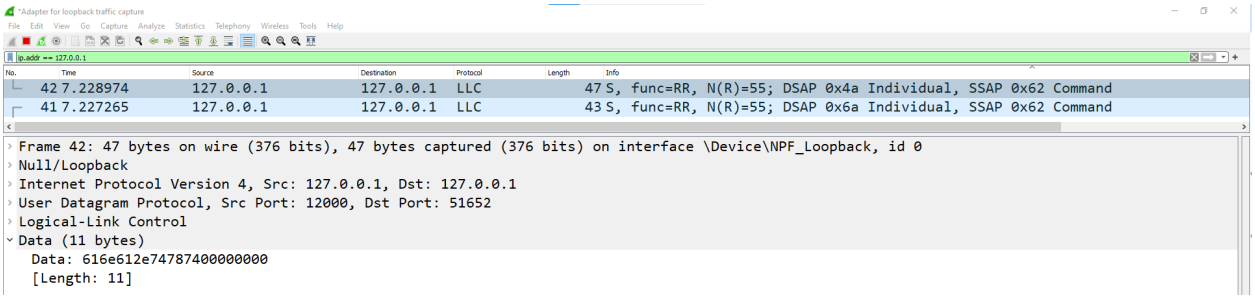

Figure 9

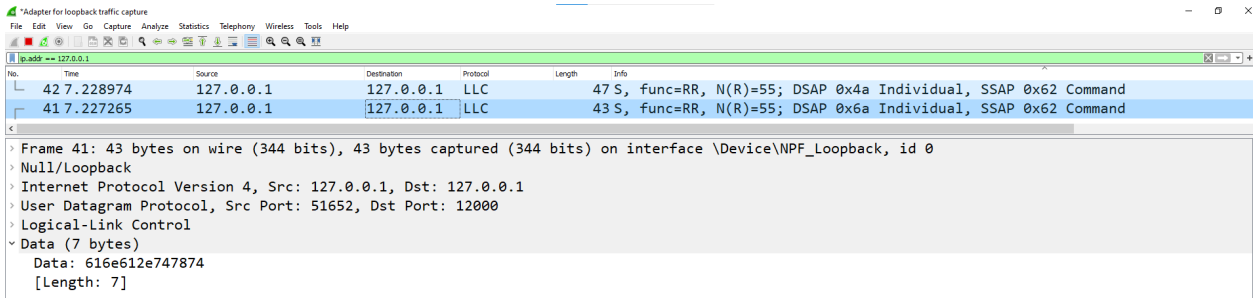

Figure 10

3) The length field is the number of bytes in the UDP segment.
4) The maximum number of bytes that can be included in a UDP payload is (2^16-1) + header
5) The protocol number of UDP is 0x11 hex, which is 17 in decimal

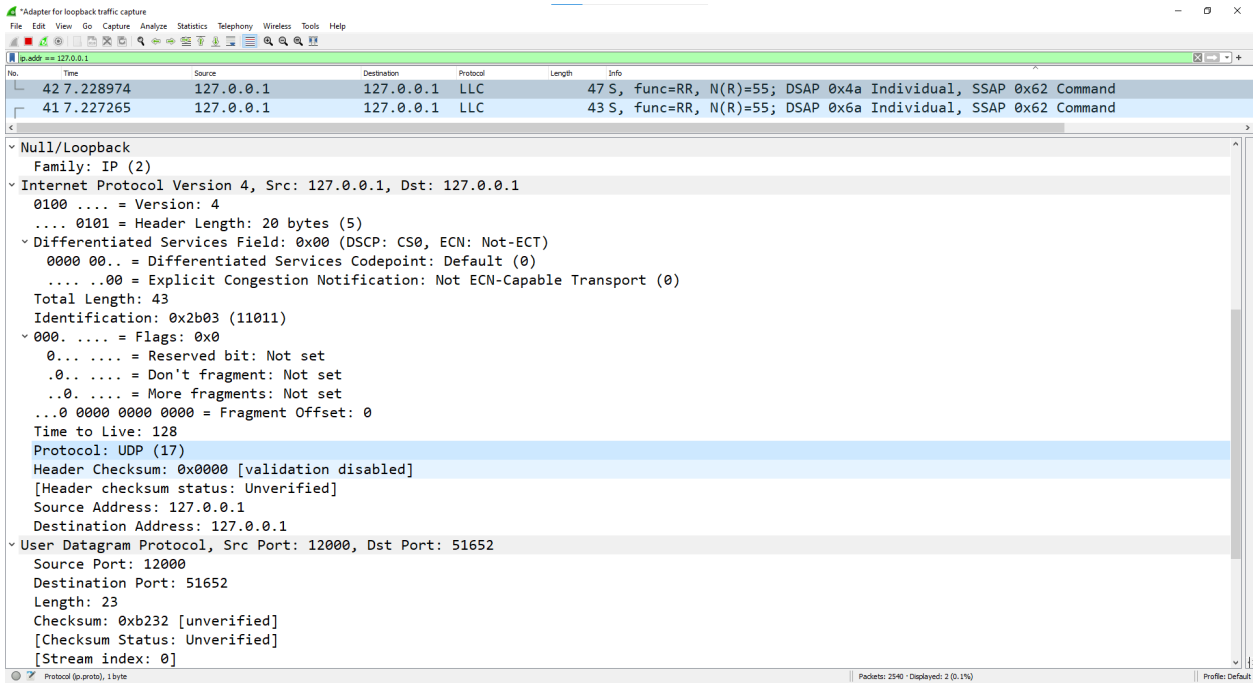

Figure 11

6) The source port of the UDP packet that is sent by the host is the same destination port as the reply packet; and similarly, the destination port of the UDP packet that is sent by the host received is the same as the source port of the reply packet.

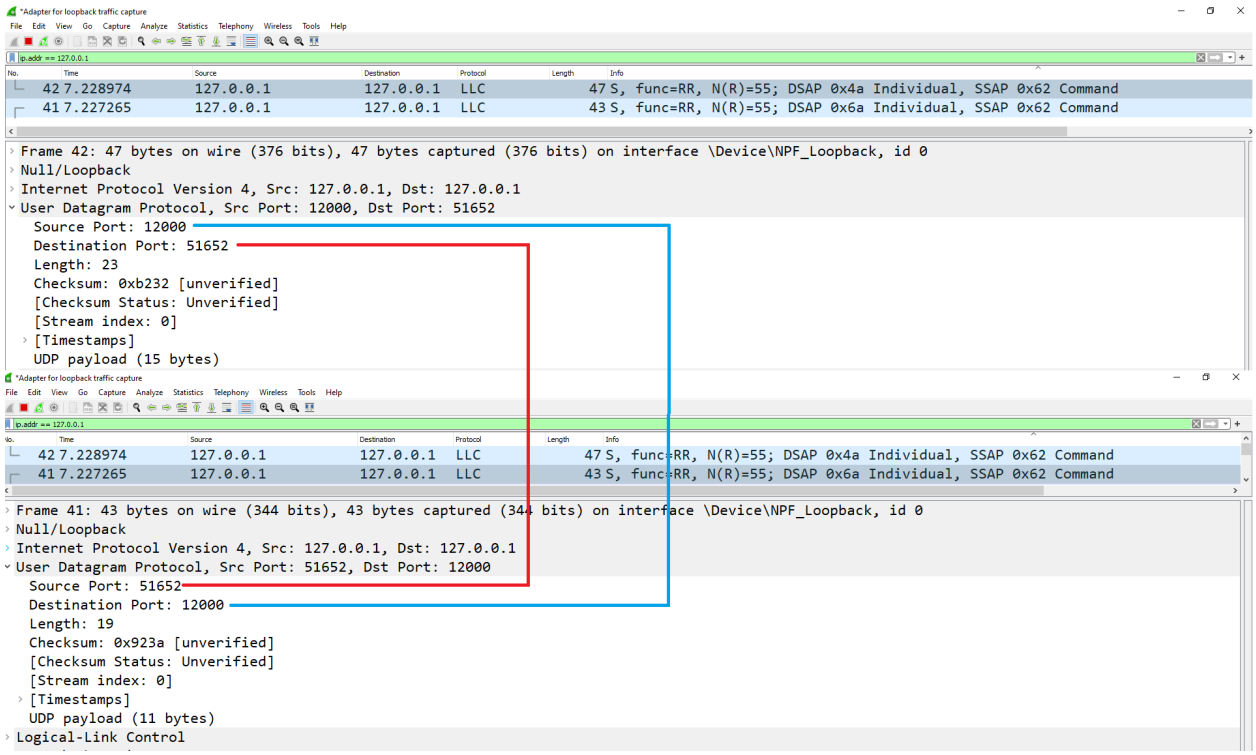

Figure 12

CONCLUSION

The project successfully fulfilled all its requirements, delivering a functional FTP system that accommodates different file types and utilizes both UDP and TCP protocols.