**PART1**

Q1)

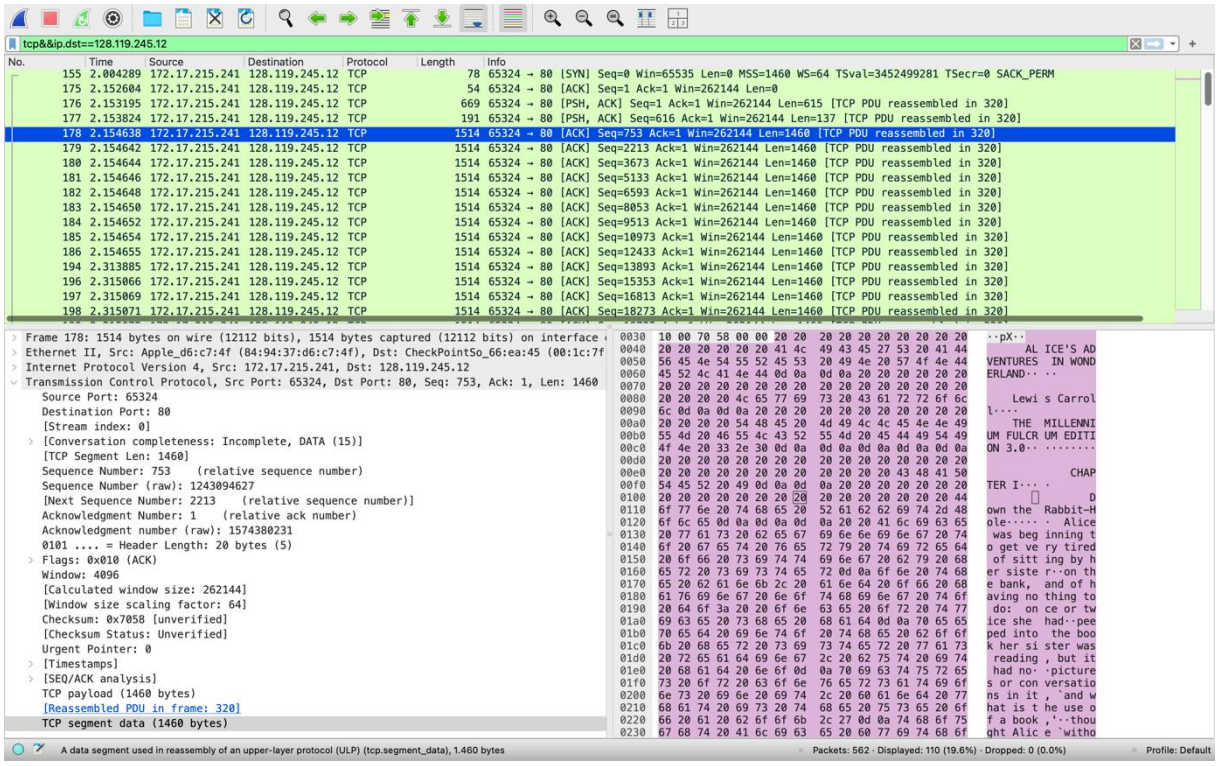


*Figure 1: snapshot of Wireshark filtered by tcp&&ip.dst ==128.119.245.12*

I performed the what you need to, then filter the packets by TCP and I got a packet which has destination= 128.119.245.12**, [tcp&&ip.dst==128.119.245.12]** .

From this packet I look its Transmission control protocol, as you can see in the snapshot *figure 1* **Sequence number = 753** and **Acknowledgement number = 1**. Also next sequence number is 2213.

By sequence number position of the first byte in stream is showed and this ensures the packets will be reassembled in the correct order. Next sequence number =2213 shows where the end of packet ends and also from which byte the other next packet will start. So, by these artifacts we may get information about packet data length 2213-753 = 1460 byte also we could confirm this number from **TCP Segment Len** shown in the snapshot.

By Acknowledgement number we can understand that the last byte of data receiver received. Sometimes segments may lose their order in an unintended way. But even in this case, the Sequence Number allows the receiver to combine these segments in the correct order. The Acknowledgment Number indicates the number of bytes the receiver expects for the next packet exchange. This mechanism also supports the accuracy of the system; that is, it allows the receiver to feedback to the sender whether the data was received or not.

Q2)

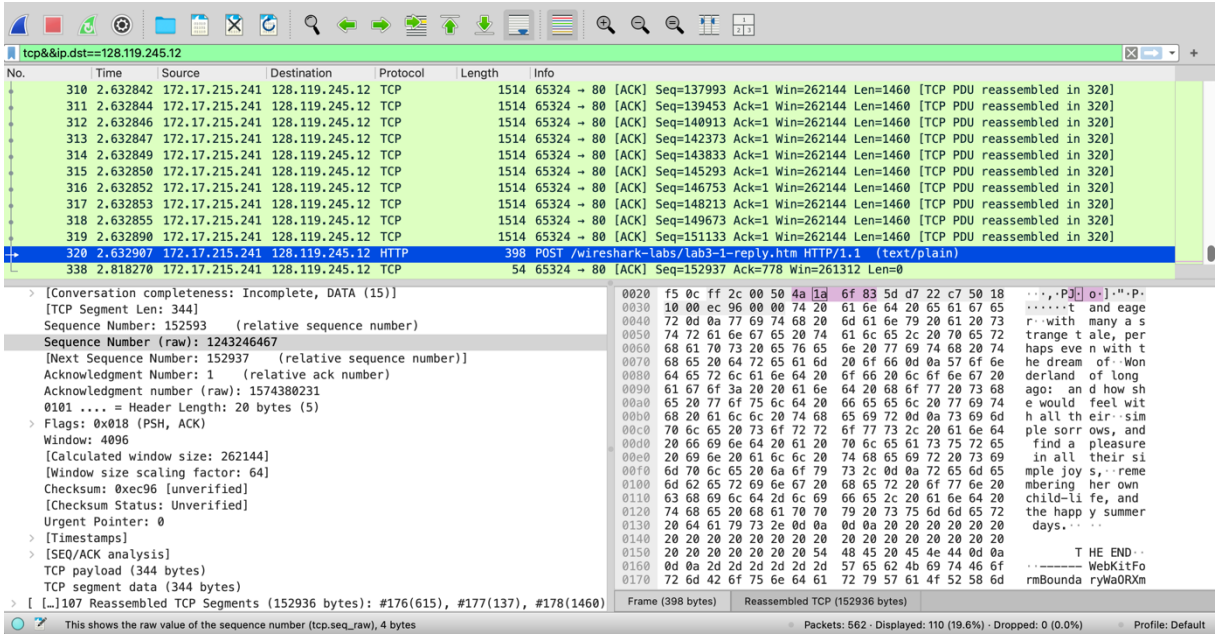


*Figure 2: Evidence for distribution of multiple packets*

Alice.txt is sent distributed to multiple packets, because TCP sends large files in small chunks. As you can see in *figure 2*, the file at HTTP layer was sent by only one POST request but TCP layer split data to segments to carry this request.

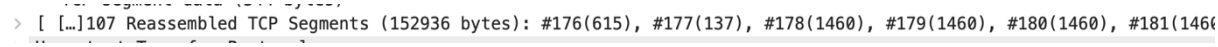


*Figure 3: Reassembled showed*

Also as seen in *figure 3*, by this Reassembled TCP segments, we prove that this transportation is made in multiple segments. The reason for this is that each of the segments separated by Maximum Segment Size is checked separately and missing segments are resent.

Q3)

In *figure 1* our stream index value is 0 , so filter by **tcp.stream eq 0**.

What is RTT?

RTT corresponds to time between sending a data segment and getting (transmission) of its response. In our context, RTT will represent the time between sending data segment on TCP connection and receiving its ACK. We can use **Statistics -> TCP Stream Graphs -> Round Trip Time** . In *figure 4*, you see the RTTs of packet sended from my computer to http://gaia.cs.umass.edu/wireshark-labs/lab3-1-reply.htm website. In *figure 5*, you see the RTTs of packets came from website to my computer.

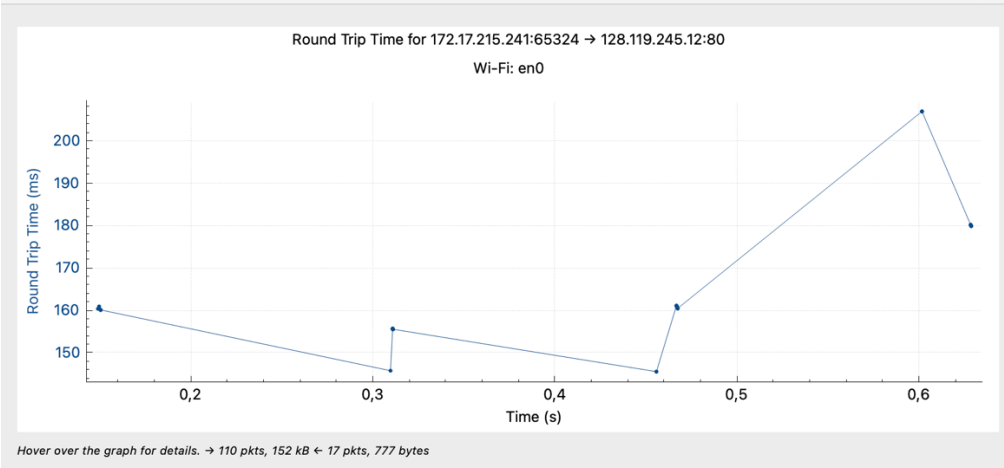


*Figure 4: RTT graph of client to server*

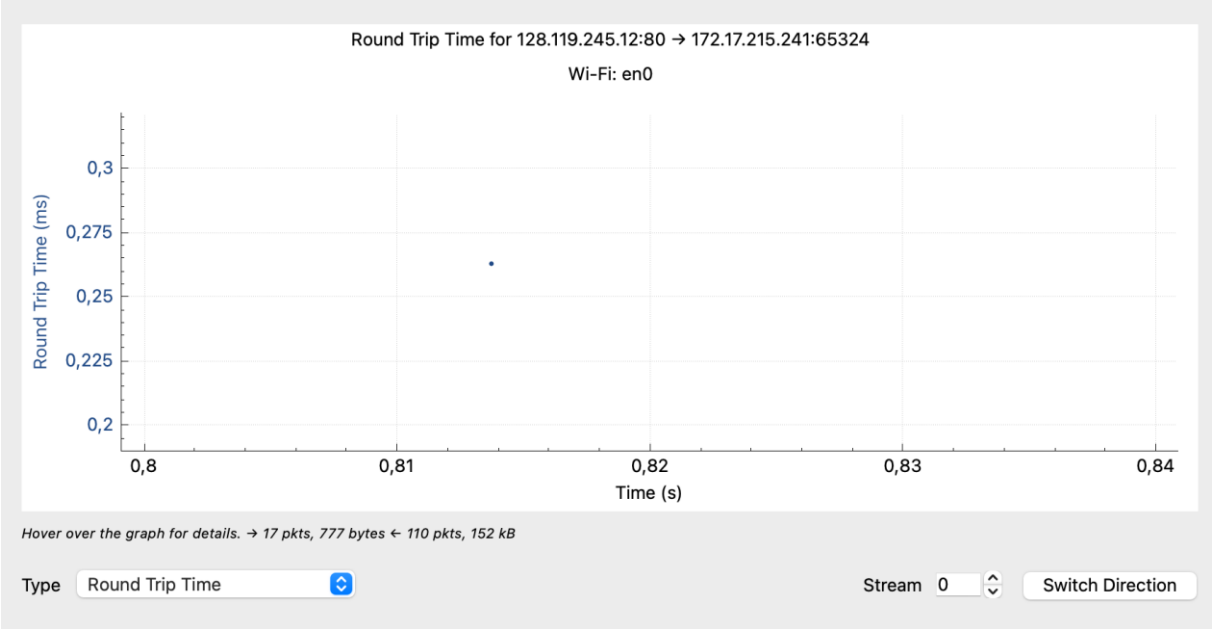


*Figure 5: RTT graph of server to client*

Q4)

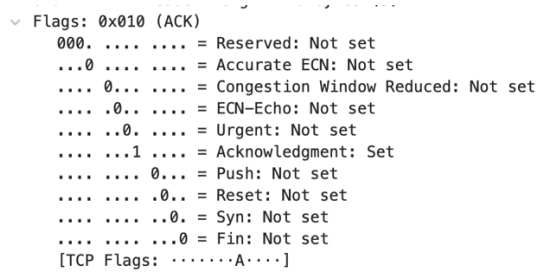


*Figure 6: flag set*

There are 10 flags.

When flag set to 1, that means this segment is that. For example, the packet that I used in Q1's flags are given in *figure 6*, as Acknowledgement is set to 1, this segment is a ACK segment.

**SYN**: It initiates a connection and used in three-way handshaking to establish TCP connection between client and server.

***Example***: Client wants to connect to server, so it sends TCP segment with SYN flag.

**ACK**: It acknowledges that the data is received, also used for acknowledging SYN, FIN, RST flags.

***Example***: Client sends a SYN, then server sends SYN-ACK, then client again sends a segment with SYN and ACK flags set to complete the handshake process.

**FIN**: It is kind a signal to terminate connection. If FIN flag is sent, this states that there will be no more data sending from sender.

***Example***: When data transmisson is completed, client sends a TCP segment which includes FIN flag to close connection.

**RST**: It suddenly terminates the connection.

***Example***: Suppose a TCP segment is sent to a server on a port that is unavailable or closed. A segment with the RST flag is returned to notify the sender of the error.

**PSH**: It is used to state that immediate data processing is required. The data which in segment must be delievered to receiving application right away without waiting for data to fill buffer.

***Example***: Let's say there is a chat program in which a user is typing. To ensure that the input is received and immediately displayed on the other user's screen, each keystroke creates a short TCP segment with the PSH flag.

Q5)

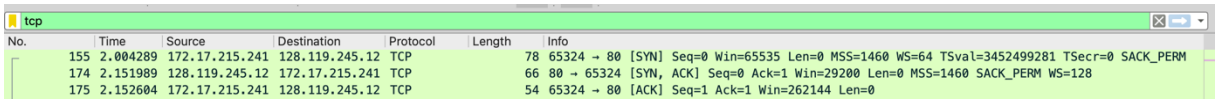


*Figure 7: three-way handshaking of TCP*

Three-way handshake is a process used by TCP connections to create a mutual readiness. This process indicates that both side will communicate that in a way like they are ready to data transmission and exchange. In *figure 7*, you see three packets in the trace.

Firstly, SYN is sent from source (my IP) to destination(the web site that we upload alice.txt). This requests connection start.

Secondly, SYN-ACK is sent from source (the web site) to destination (my IP). This indicates that the server accepts the connection request by sending SYN and ACK flag set.

Thirdly, ACK is sent from source (my IP) to destination (the web site). This indicates that I (client) acknowledge server's SYN&ACK packet.

## PART2

I filtered the given wireshark file with **udp&& !(quic || dns || tcp)** to take only UDP packets.

Q1)

The last two digits of my student id number is "01", but there is no packet has no=1 so I took 2 as selected but fort he first question there are packets that they have same source ip address but they are all sending to same destination as well as 2. I checked with the filter "**udp&& !(quic || dns || tcp)&&ip.src == 172.217.17.234**" you may also see in *figure 8*.

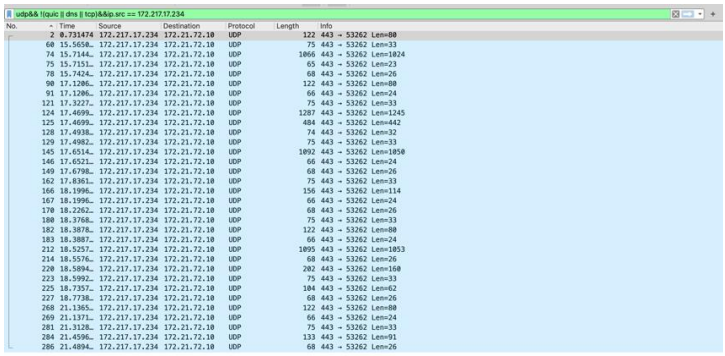


*Figure 8: Only UDP packet filtering*

So, I took packet numbered 3 as my base.

Packet 4 also has same source IP address with packet 3 but different destination IP address.

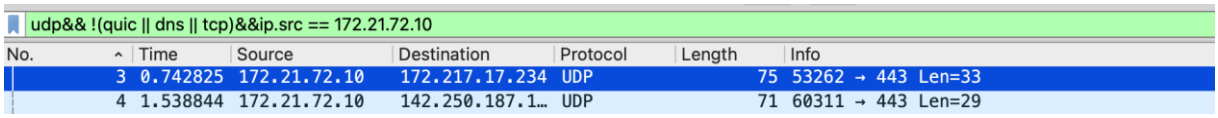


*Figure 9: Looking port numbers of dst and src*

As you may see from *figure 9*, in the info part the Source Port = 53262 and Destination Port=443, it can be also observed from packet 3's User Datagram Protocol side as seen in *figure 10*.

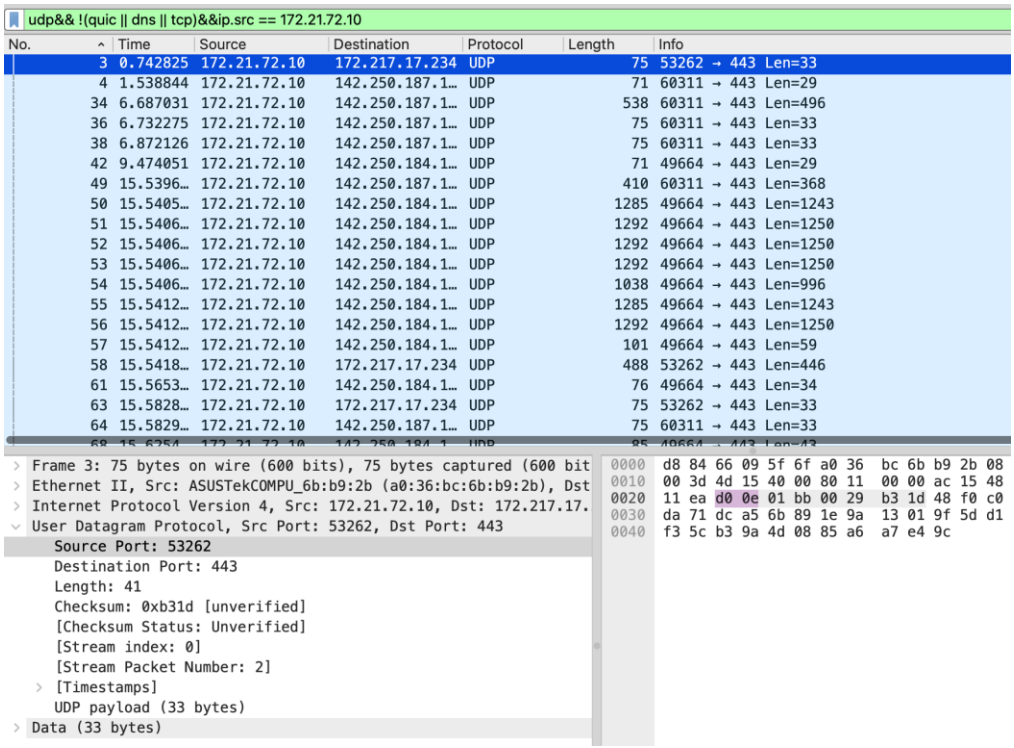


*Figure 10: Looking port numbers of dst and src*

For packet 4, you may see from figure 9, in the info part the Source Port = 60311 and Destination Port=443, again it can be checked from User Datagram Protocol side.

The destination port 443 stands for HTTPS, it is known port for HTTPS.

Source port roles: it identifies sending application on sender's side. It is assigned by clients randomly.

Destination port roles: it identifies receiving application on destination side. This is fixed to listen incoming connections.

Q2)

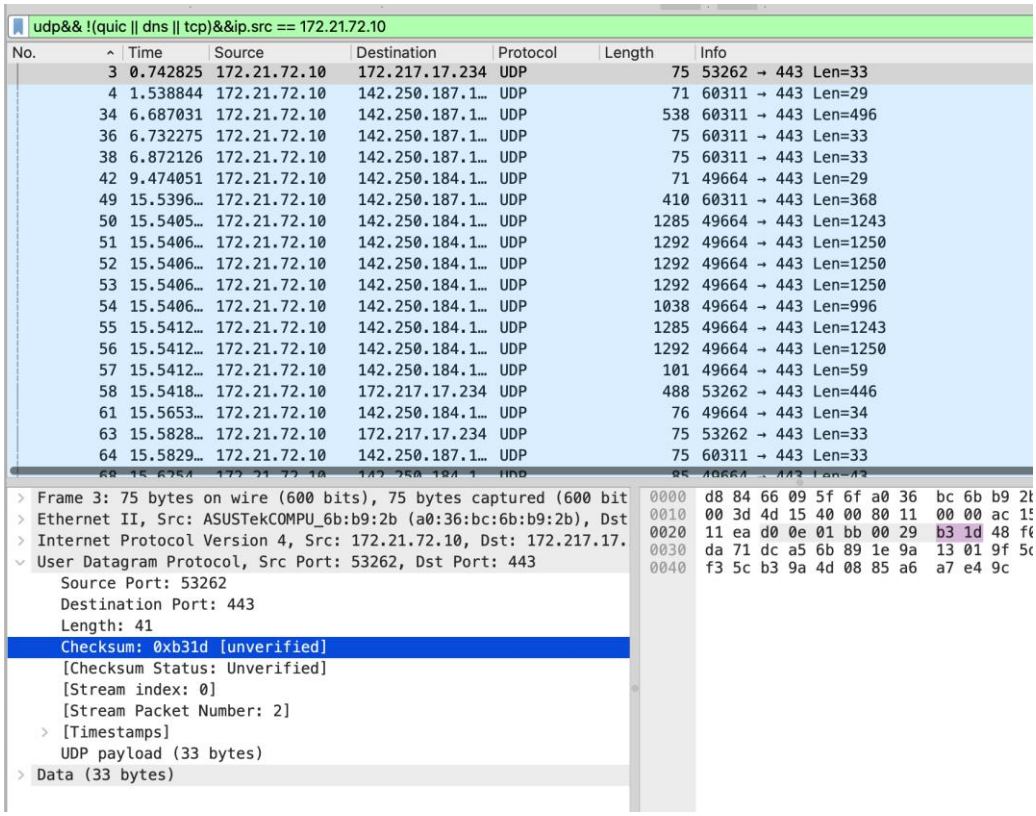


*Figure 11: Examining the checksum field*

In *figure 11*, you may see checksum field of my selected packet 3. UDP checksums are enabled by default on all modern operating systems. In IPv4, it is possible to disable UDP checksums at the socket or operating system level, so this feature is optional. Setting this setting reduces the CPU load that each packet must be processed on both the sender and receiver sides.

Q3)

All UDP headers have same size which is 8 bytes. This is fixed in UDP. I find it by subtracting data length (33 bytes) from length(which is in User Datagram Protocol side=41)= 41-33=8 bytes. You can see these values from *figure 12*.

UDP uses smaller header size as we observed 8 bytes. This is important because UDP uses small packet sizes, which can reduce network congestion and improve overall network performance. You may see the UDP's header size schema in *figure 13* [1].

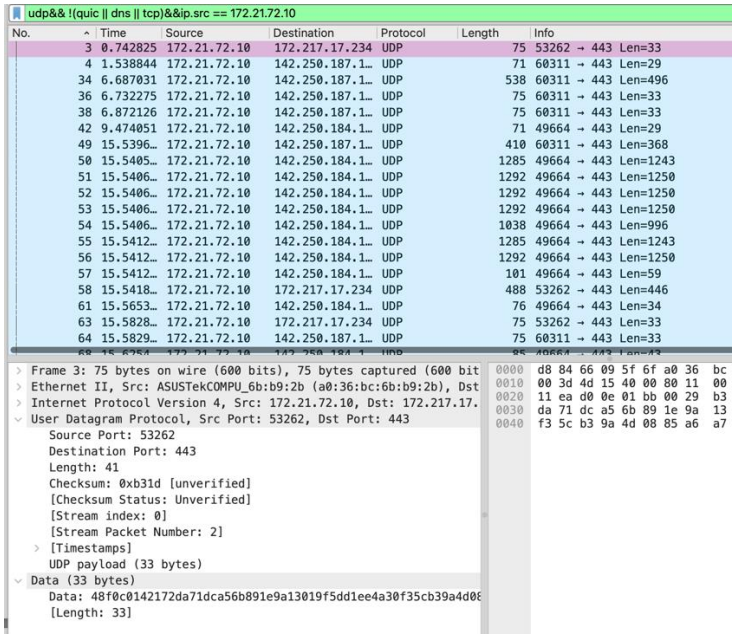


*Figure 12: UDP's header size finding*

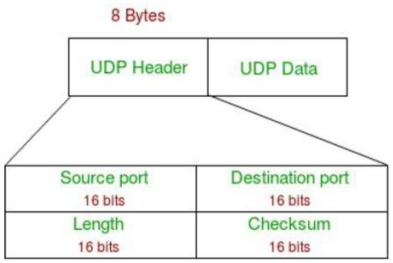


*Figure 13: Schema of UDP header*

Q4)

UDP's stateless nature allows it to scale easily with many clients while using fewer resources because it does not keep track of connection states. However, this can make it difficult to detect and correct errors such as data loss or packet order corruption. The stateless nature of UDP can be disadvantageous, especially in situations where reliability, error control, or data order are important (for example, financial transactions).

Q5)

I filtered the given data by "**tcp**" and look randomly packet 28's and packet 29's header lengths. They are 20 bytes and 32 bytes respectively; you may see in *figure 14* and *figure 15*. So, as we may understand in TCP, header length can vary. As we already examine in question 3 UDP has fixed 8 bytes size for header.

As I researched, TCP's header size can vary in 20 bytes to 60 bytes. A TCP segment that has 20 bytes header size includes source port, destination port, sequence number, acknowledge number, header length, control flags, window size, checksum, urgent pointer, reserved bits. And it can include optional data (0 to 40 bytes), it could be timestamp, maximum segment size etc. As we examine in question 3, figure 13, UDP header has areas like source port, destination port, length and checksum. So the other extra field that TCP has, provides connection following, error checking and helping to handle to TCP. But these lots of extra areas, leads to overhead for TCP when we compare with UDP.

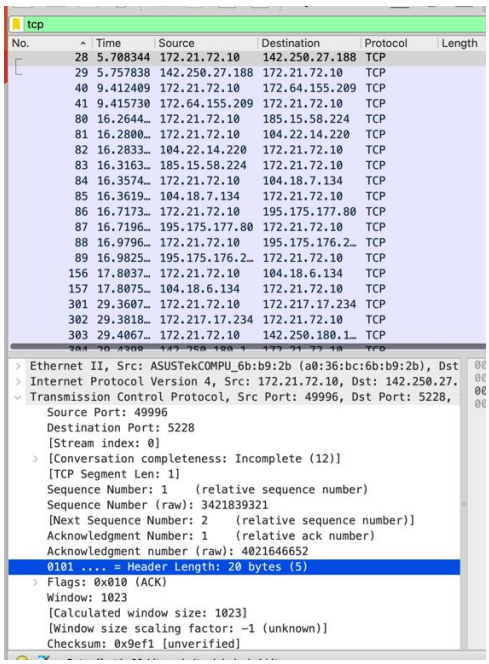


*Figure 14: Header Length of TCP packet 28*

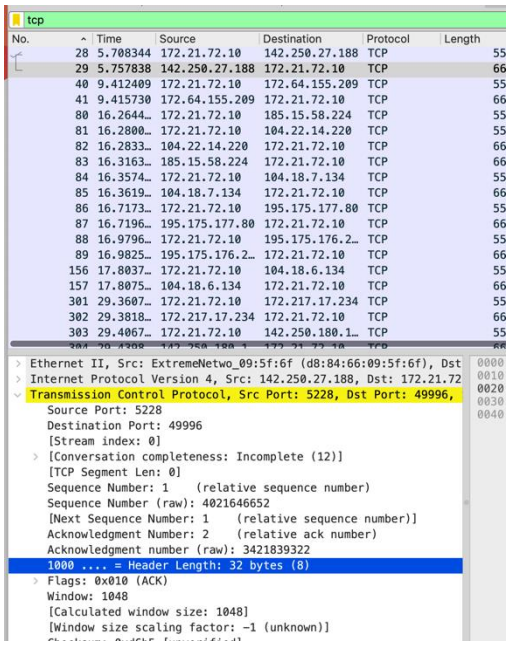


*Figure 15:Header Length of TCP packet 29*

Q6)

First, start with what is QUIC stands for: Quick UDP Internet Connections. It works at the top of UDP, and it gives less latency for connection. It minimizes the overhead and keeps the state of connection. It provides security in communication by correcting the error, encrypting the data (by using TLS). We may see the fields of QUIC in figure 16. There are different areas of QUIC when we compare with UDP such as encryption, security, state of connection.

As you may realize, at the bottom of *figure 16*, there is an area which includes “Client Hello” (Under the QUIC IETF -> CRYPTO). Also, you may see new areas of QUIC in *figure 16* (in Info part) which are DCID, SCID. These are unique IDs for destination and source respectively. There is Token Length (Under the QUIC IETF->QUIC Connection Information) are, this token is for reliable setup.

Side info: As it is new, it is not supported in every browser yet.

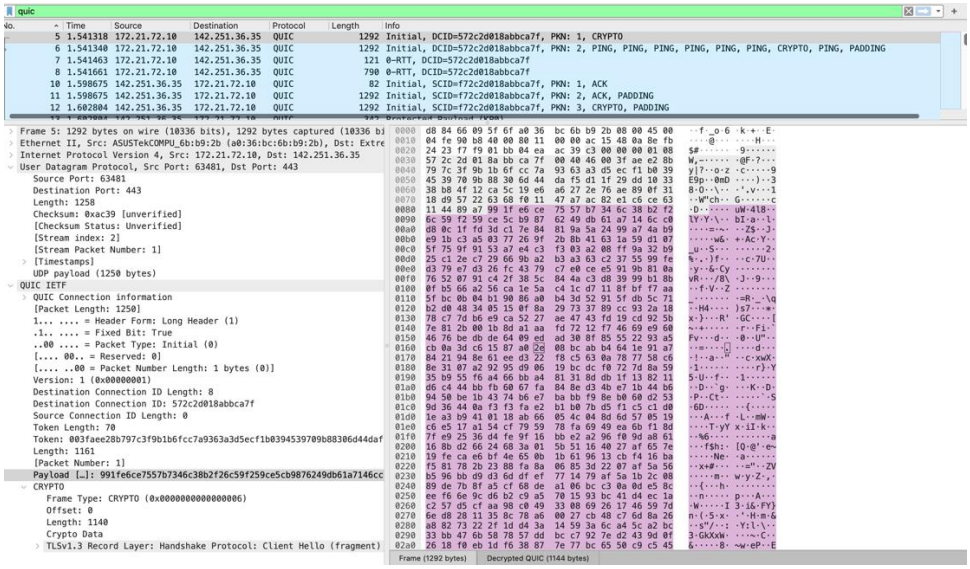


*Figure 16: Examining the QUIC packet 5*

PART3

I generated the key by using these commands in my terminal as you can see in *figure 17,* I inspired with the command that I learn from the website [2  I used these commands to generate keys. My password is “Gulnisa”.

```
gulnisayildirim@Gulnisa-MacBook-Air ~ % keytool -genkeypair -alias sslserver -keyalg RSA -keysize 2048 -validity 365 -keystore server.keystore
Enter keystore password:
Re-enter new password:
Enter the distinguished name. Provide a single dot (.) to leave a sub-component empty or press ENTER to use the default value in braces.
What is your first and last name?
  [Unknown]:  Gulnisa Yildirim
What is the name of your organizational unit?
  [Unknown]:  Student
What is the name of your organization?
  [Unknown]:  Koc University
What is the name of your City or Locality?
  [Unknown]:  Istanbul
What is the name of your State or Province?
  [Unknown]:  Istanbul
What is the two-letter country code for this unit?
  [Unknown]:  TR
Is CN=Gulnisa Yildirim, OU=Student, O=Koc University, L=Istanbul, ST=Istanbul, C=TR correct?
  [no]:  yes

Generating 2,048 bit RSA key pair and self-signed certificate (SHA384withRSA) with a validity of 365 days
        for: CN=Gulnisa Yildirim, OU=Student, O=Koc University, L=Istanbul, ST=Istanbul, C=TR
gulnisayildirim@Gulnisa-MacBook-Air ~ % keytool -exportcert -alias sslserver -file server.crt -keystore server.keystore
Enter keystore password:
Certificate stored in file <server.crt>
gulnisayildirim@Gulnisa-MacBook-Air ~ % keytool -importcert -alias sslserver -file server.crt -keystore client.truststore
Enter keystore password:
Re-enter new password:
Owner: CN=Gulnisa Yildirim, OU=Student, O=Koc University, L=Istanbul, ST=Istanbul, C=TR
Issuer: CN=Gulnisa Yildirim, OU=Student, O=Koc University, L=Istanbul, ST=Istanbul, C=TR
Serial number: 69bf6b57fd1b7938
Valid from: Tue Dec 10 22:29:31 GMT+03:00 2024 until: Wed Dec 10 22:29:31 GMT+03:00 2025
Certificate fingerprints:
         SHA1: 78:AB:69:9D:CF:79:02:0A:29:FB:B2:12:DA:D5:DA:CA:E8:A7:F9:A3
         SHA256: 06:F6:71:DB:D2:D3:2D:33:53:1C:87:96:61:84:93:11:8E:F5:8F:A2:2B:CB:51:83:13:06:C7:6F:F5:72:FE:C1
Signature algorithm name: SHA384withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 01 3F D1 B3 E6 1D B3 2B   6B 3C F8 F1 C7 DF F9 5C  .?.....+k<.....\
0010: 65 44 E2 5D                                        eD.]
]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
gulnisayildirim@Gulnisa-MacBook-Air ~ %
```

*Figure 17: Key generation for SSL*

Q1) First, I run the server then I start the Wireshark by choosing Loopback, then turned to client and run it. I wrote the protocol name “TCP”, also entered file name which is “alice.txt”. Then I stopped the Wiresark. You may see how they look like from *figure 18* and *figure 19*.

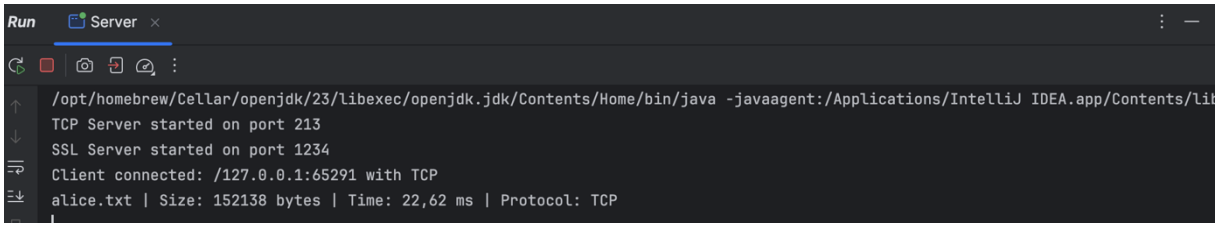


*Figure 18: The apperance of server side, after the steps taken above*

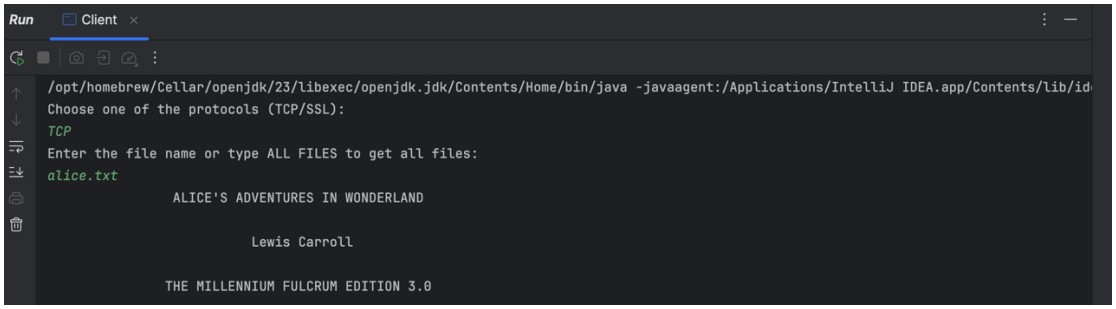


*Figure 19: The apperance of client side, after the steps taken above*

Now, let’s examine the Wireshark for this scenario.

In *figure 20*, we see at the time of server-client communication start. Client (port num=65094) sends a SYN to port=213 which I set TCP port to. Then from 213 port, comes a SYN-ACK. Lastly, we see ACK sent by client. This is three-way handshake which TCP uses as we discussed previous parts. Then, data transmission.

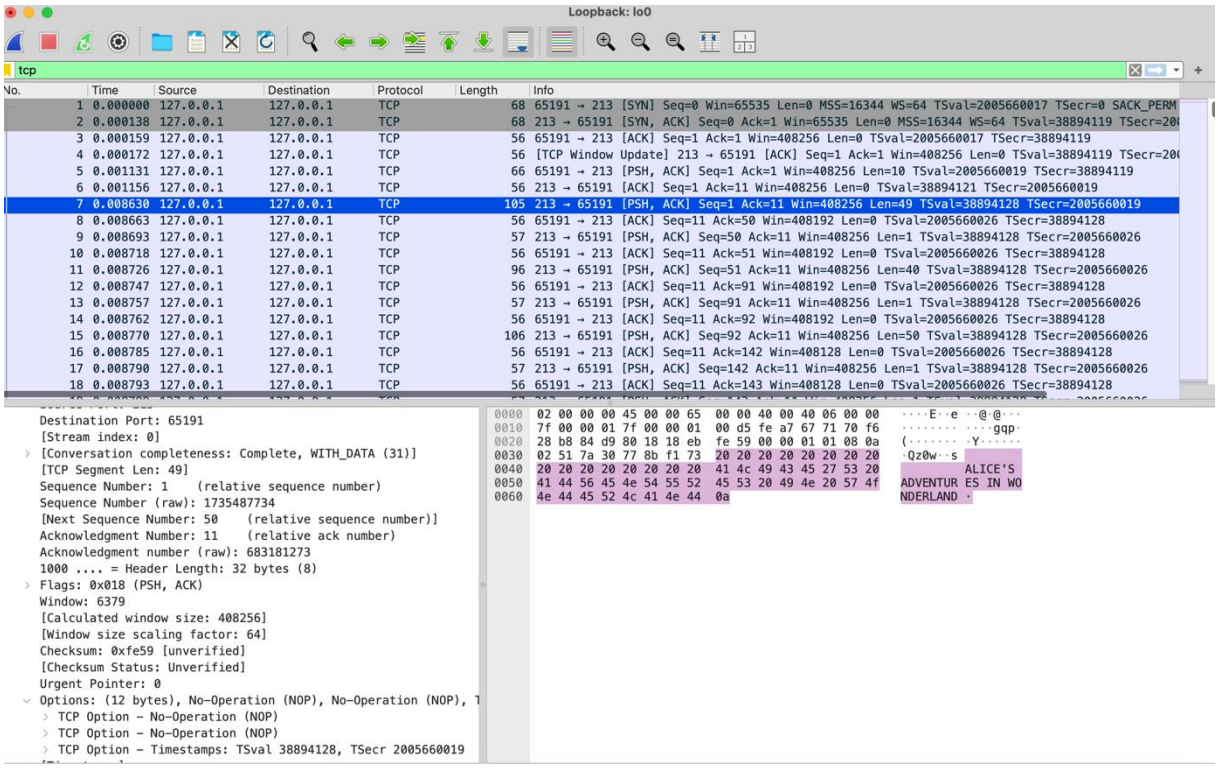


*Figure 20: Examining the TCP connection's three-way handshaking*

As you may see from *figure 20* 's right side, it is sent as clear text so that means the data is not encrypted, this is protocol's structure.

Let's now examine the SSL connection. I do the nearly same things for the TCP part, I just wrote SSL for protocol name again, I want alice.txt from server again.

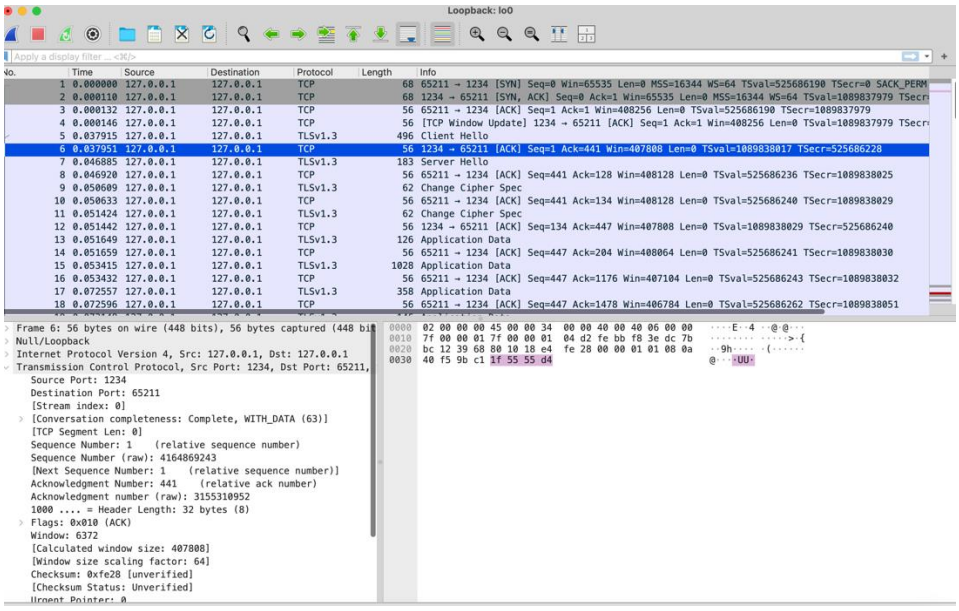


*Figure 21: Unfiltered capture of SSL connection*

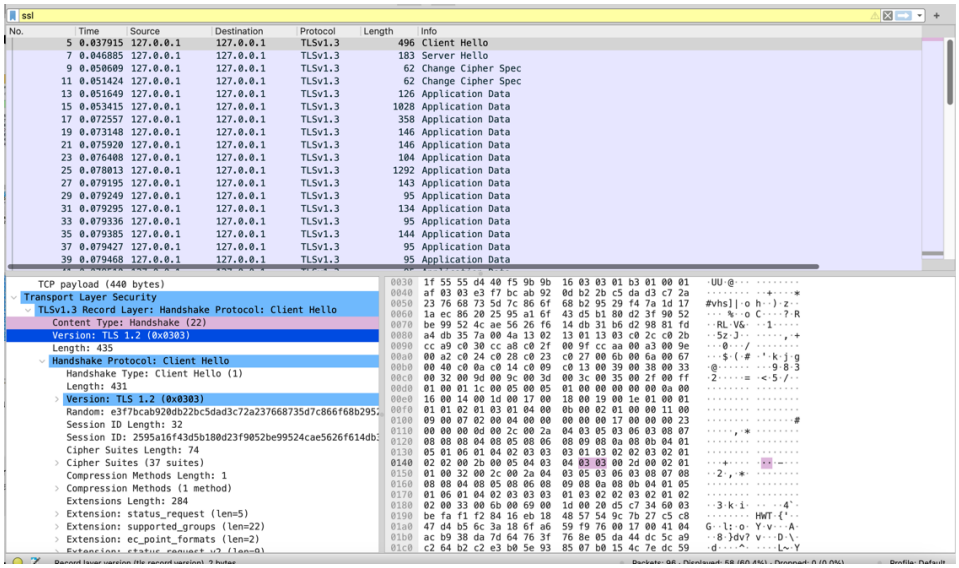


*Figure 22:Filtered capture with “”ssl”*

As we observe in *figure 21* and *figure 22*, SSL is performed on top TCP connection. “Client Hello”, “Server Hello”, “Change Cipher Spec” these messages that you see in *figure 22*, are for encryption. As TCP does not have security, encryption or certificates. But SSL has those features. I created the keys for SSL connection, those keys are exchanged between client and server to maintain data security.

Establishing connection by three-way handshake, they both have.

Q2)

In the above *figure 23* belongs to SSL Wireshark capture, I just follow these step to get flow graph: **Statistics -> Flow Graph.** It shows a diagram of data exchanging and time.

In the above *figure 24* belongs to TCP Wireshark capture. It shows a diagram of data exchanging and time.

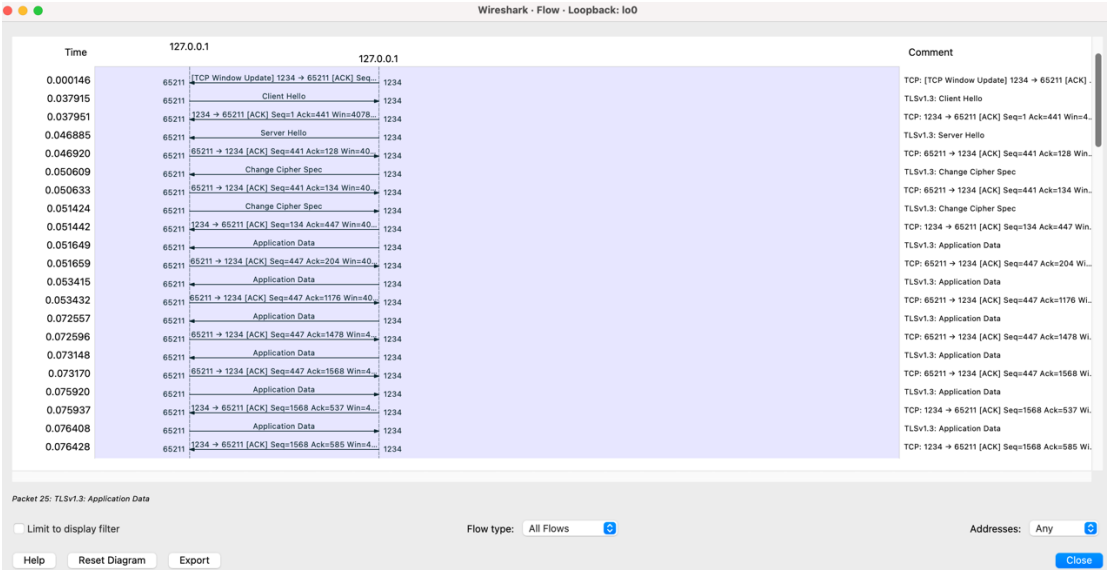


*Figure 23: Flow graph of SSL*

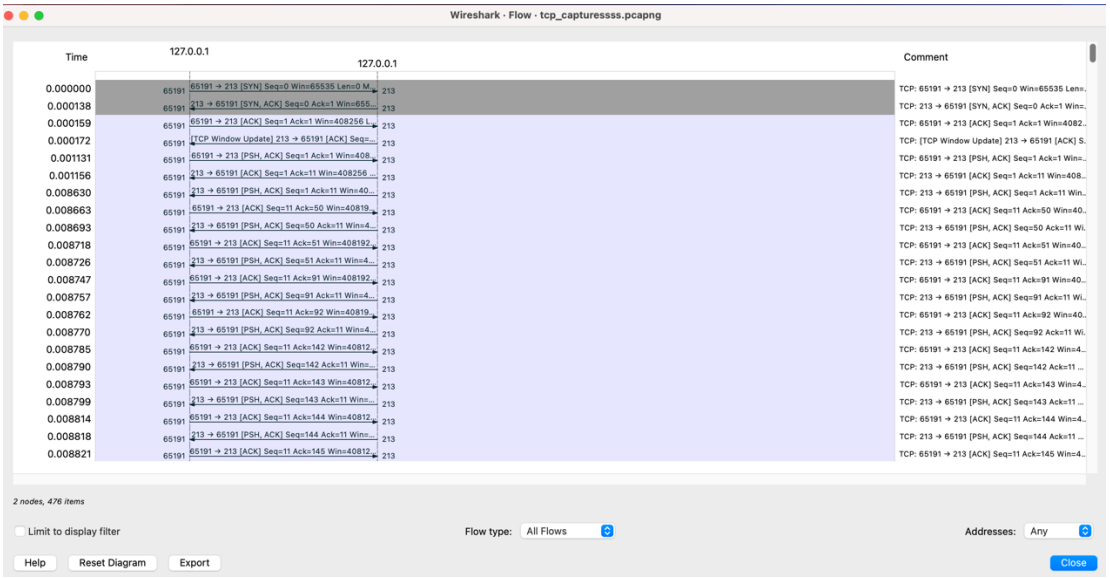


*Figure 24: Flow graph of TCP*

Why I use these graphs, I want to show the time slots of both connections, as you may realize TCP is better in terms of latency when we compare with SSL since there are key exchanging and certification confirmation in SSL, these additional areas want more time. In addition, SSL has additional encryption and decryption operations, this creates more processor load.

Q3) SSL addresses Authentication (by key exchanging certificates) and data confidentiality (by encrypting) concerns. As we have discussed before in question 1 by using its figures. TCP alone cannot address those things.

Q4)

4A)

I adapted the whole file sending from server to client by “ALL FILES” command. Let try this commend for both connections with 2 clients at the same time. You may see these in *figure 25* and *figure 26*.

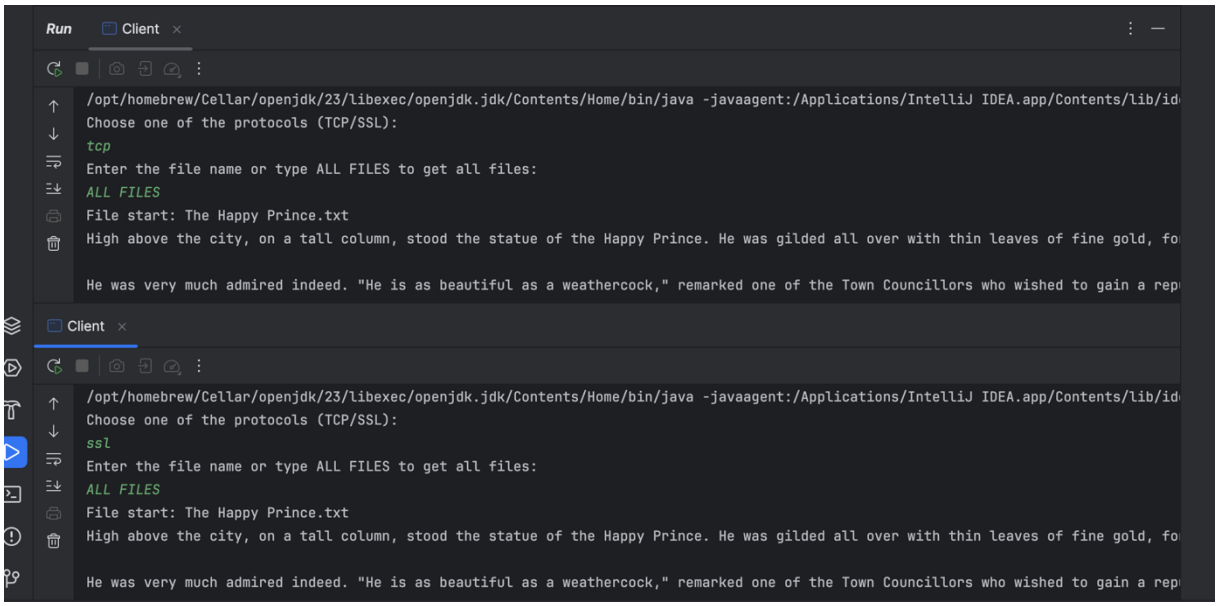


*Figure 25:Two clients with different connections tests the ALL FILES command*

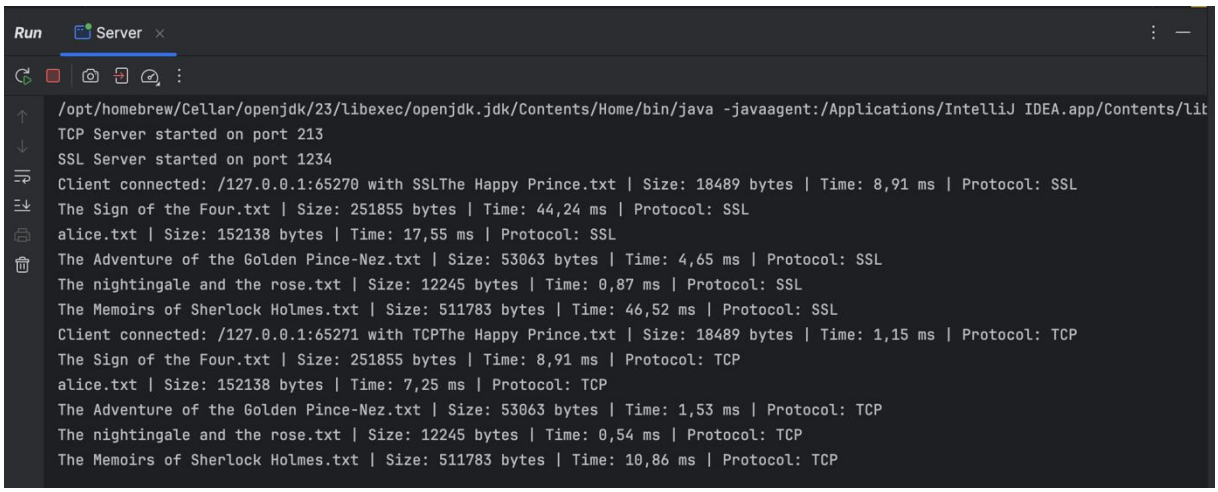


*Figure 26:Server side of demonstration of ALL FILES command*

4B) I could not plot the graphs using Java. As I guess python is prohibited. I will just analyze result by manually. I will made the same things as I done in 4a but I will just show server side separately for clarity.

Here is the result for “ALL FILES” command in TCP.

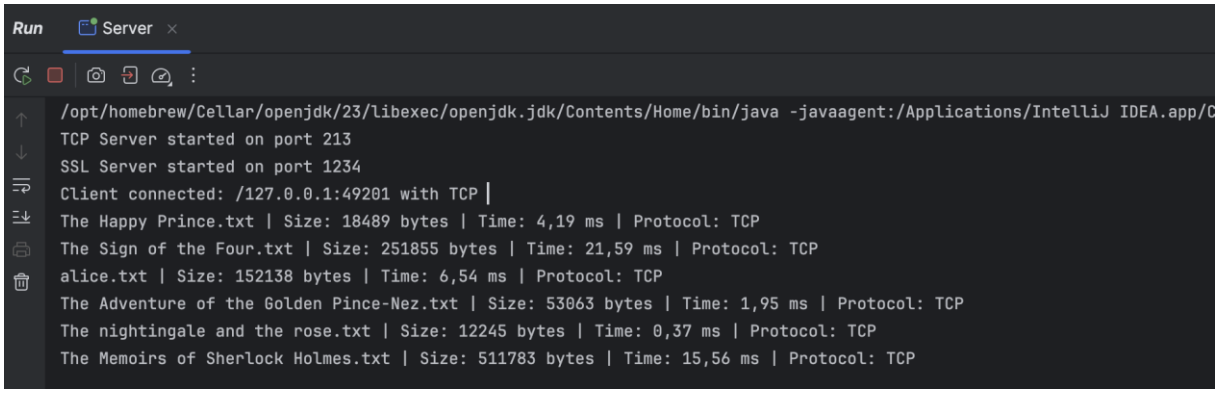


*Figure 27: Results of use of ALL FILES command in TCP*

Here is the result for “ALL FILES” command in SSL.

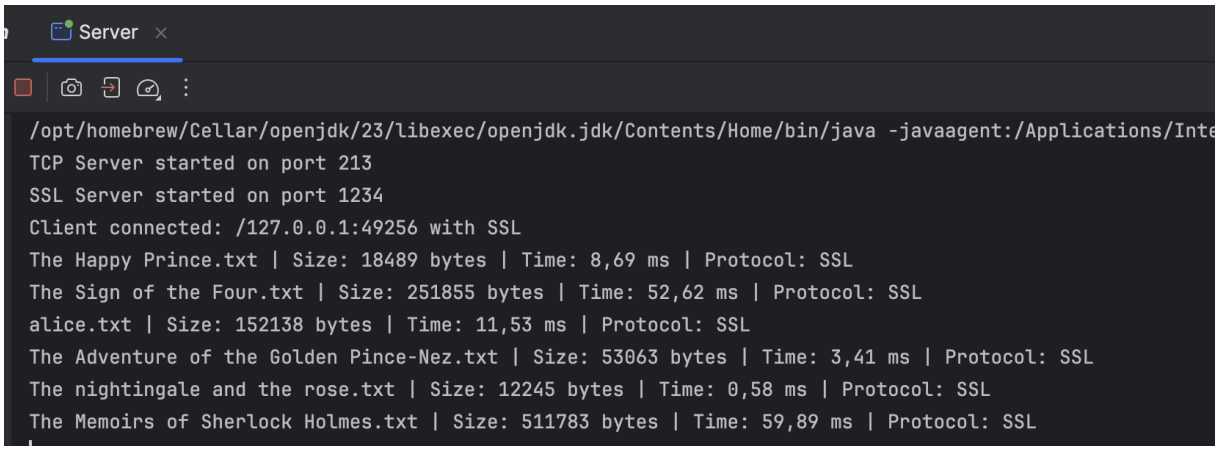


*Figure 28:Results of use of ALL FILES command in SSL*

As you may see from *figure 27* and *figure 28*, SSL has approximately 3 times lower data transmission speed compared to TCP.

Additionally, client can leave from server by typing “QUIT”. You can see the demonstration of it in *figure 29* and *figure 30* .

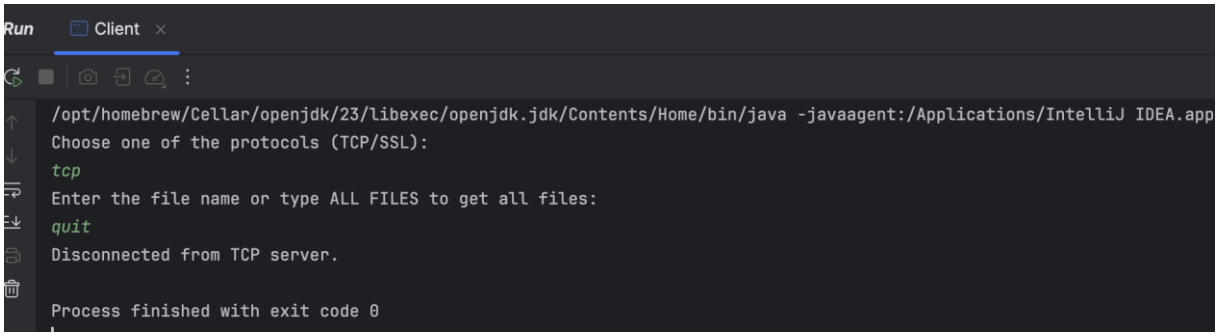


*Figure 29: Client side of QUIT demonstration*

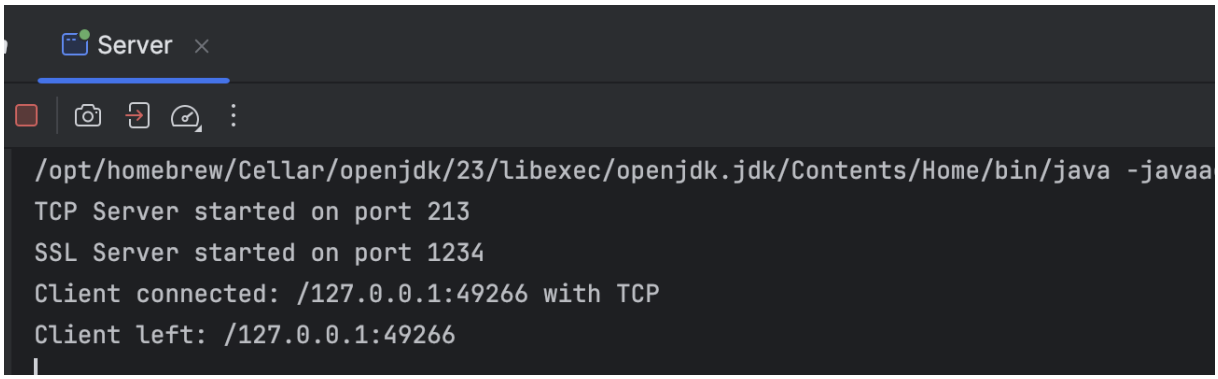


*Figure 30: Server side of QUIT demonstration*

Note: generated certificate, keystore, and trust store  are in server and client project files.

REFERENCES

[1] *Figure 13* (https://www.geeksforgeeks.org/user-datagram-protocol-udp/)

[2] *Figure 17* (https://www.fico.com/fico-xpress-optimization/docs/latest/insight5/install/GUID-884D3D7F-6B22-49A3-B941-2453F989F565.html)