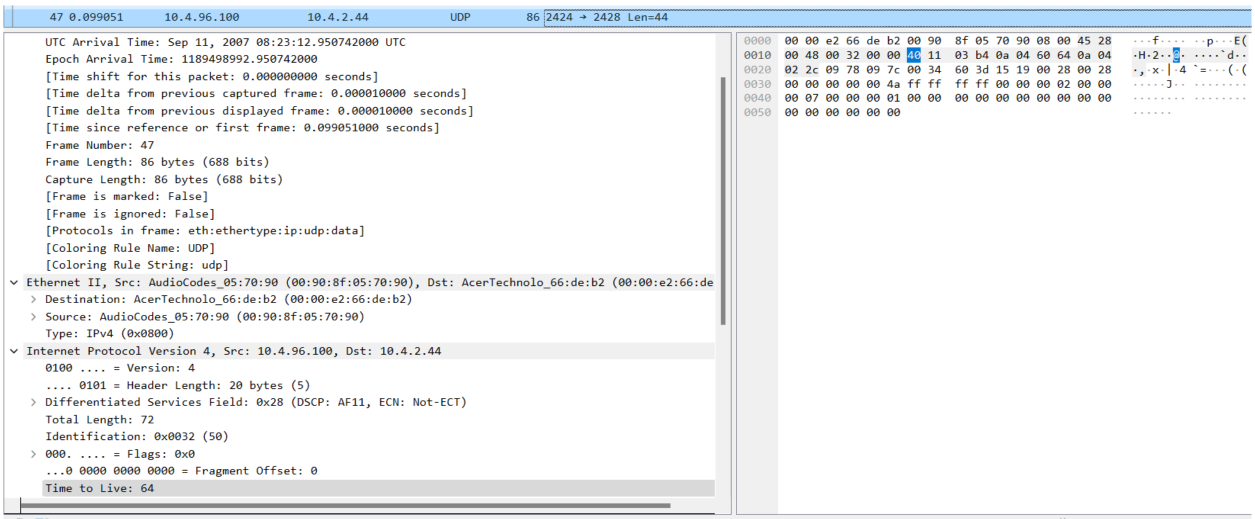
Project Report

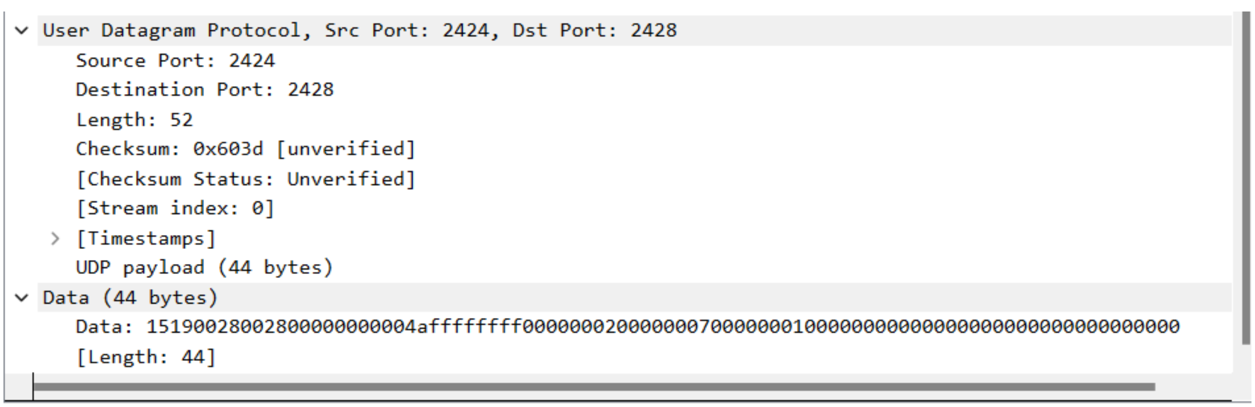
Transport Layer Protocols' Experiments with Wireshark

Ahmet UYAR

Part 1.1. UDP Experiment

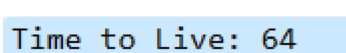
My UDP segment:





[1] UDP Segment

1.1.1. Time to live: 64



[2] TTL

If time to live reaches zero before the packet has reached its destination, the packet gets discarded, packet loss will occur.

1.1.2. Stream index: 0

```
[Stream index: 0]
```

[3] stream index

In Wireshark, the "stream index" for a UDP segment is an identifier used to organize and display related UDP packets in a sequential manner. It helps analyze the flow of UDP communication between specific source and destination IP addresses and ports, simplifying the examination of packet sequences within Wireshark.

```
Checksum: 0x603d
```

[4] checksum

1.1.3. Checksum value of the segment: 0x603d

The checksum value for my UDP segment is 0x603d, serving to ensure the segment's integrity during transmission. If this checksum value were different, it would suggest possible corruption during transit, and the receiving system might reject the segment. TCP, which also uses a checksum, offers error recovery mechanisms. In TCP, if the checksum is invalid, the segment is discarded, and the system may request retransmission. Unlike TCP, UDP lacks built-in error recovery, leaving it to the application layer to handle any needed retransmissions or error management.

1.1.4. Value of my segments reserved bit flag:

```
000. .... = Flags: 0x0
   0... .... = Reserved bit: Not set
   .0.. .... = Don't fragment: Not set
   ..0. .... = More fragments: Not set
...0 0000 0000 0000 = Fragment Offset: 0
```

[5] reserved bit flags

The reserved bit is 0. In the IPv4 header, there's a Reserved bit that's kind of like an empty space reserved for possible future uses—it hasn't been decided what it'll do yet. When you see the Reserved bit as 0, it just means it's not doing anything special right now. It's like a placeholder, kept there in case they want to add new features or changes to the system later on. Right now, everyone agrees to ignore whatever the

Reserved bit might mean because it's not defined yet. It's just there, waiting for potential updates to the protocol.

1.1.5. Value of my packet length: 86

```
Frame 47: 86 bytes on wire (688 bits), 86 bytes captured (688 bits)
```

[6] packet length

Since 86 bytes have been captured, then the total length of the packet combined with its data, header, and placeholder should be 86

```
.... 0101 = Header Length: 20 bytes (5)
Differentiated Services Field: 0x28 (DSCP: AF11, ECN: Not-ECT)
Total Length: 72
```

[7] total length of packet

```
User Datagram Protocol, Src Port: 2424, Dst Port: 2428
    Source Port: 2424
    Destination Port: 2428
    Length: 52
```

[8] packet length without header

20 bytes (header length) + 52 bytes (data length) + 14 bytes (placeholder length) = 86 bytes

Part 1.2. TCP Experiment

1.2.1. No segment loss or no retransmissions

4269	18.648158	172.21.132.90	128.119.245.12	HTTP	56013	POST /wireshark-labs/lab3-1-reply.htm HTTP/1.1 (text/plain)
4305	18.799662	128.119.245.12	172.21.132.90	HTTP	831	HTTP/1.1 200 OK (text/html)

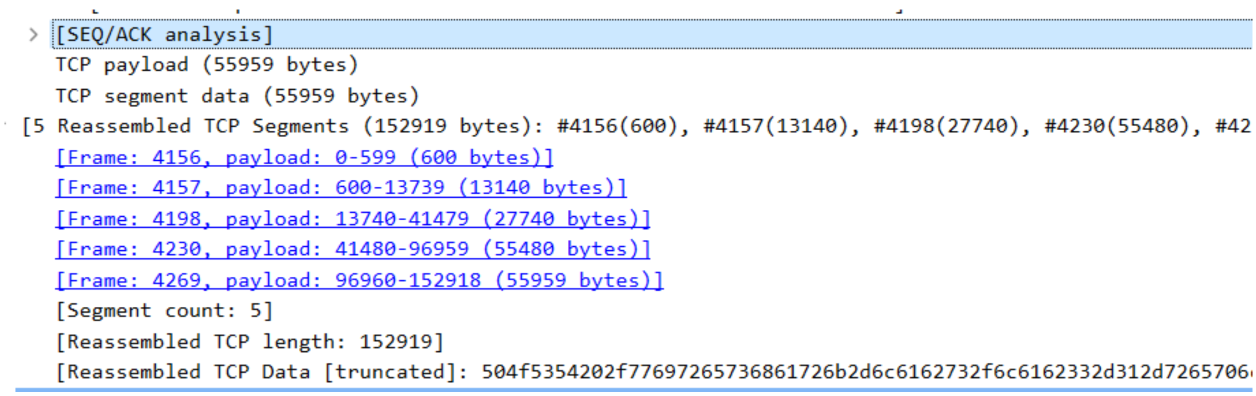
[9] successful post request

As shown in the figure, the connection is between the IPs 172.21.132.90 and 128.119.245.12. Using the postfilter “tcp.analysis.retransmission”, we do not see any retransmissions occur between these IPs

tcp.analysis.retransmission

No.	Time	Source	Destination	Protocol	Length	Info
5390	22.577485	35.186.224.25	172.21.132.90	TCP	54	[TCP Retransmission] 443 → 54418 [FIN, ACK] Seq=1 Ack=2 Win=269 Len=0
5410	22.693761	104.96.151.139	172.21.132.90	TCP	54	[TCP Retransmission] 80 → 54419 [FIN, ACK] Seq=1 Ack=2 Win=501 Len=0

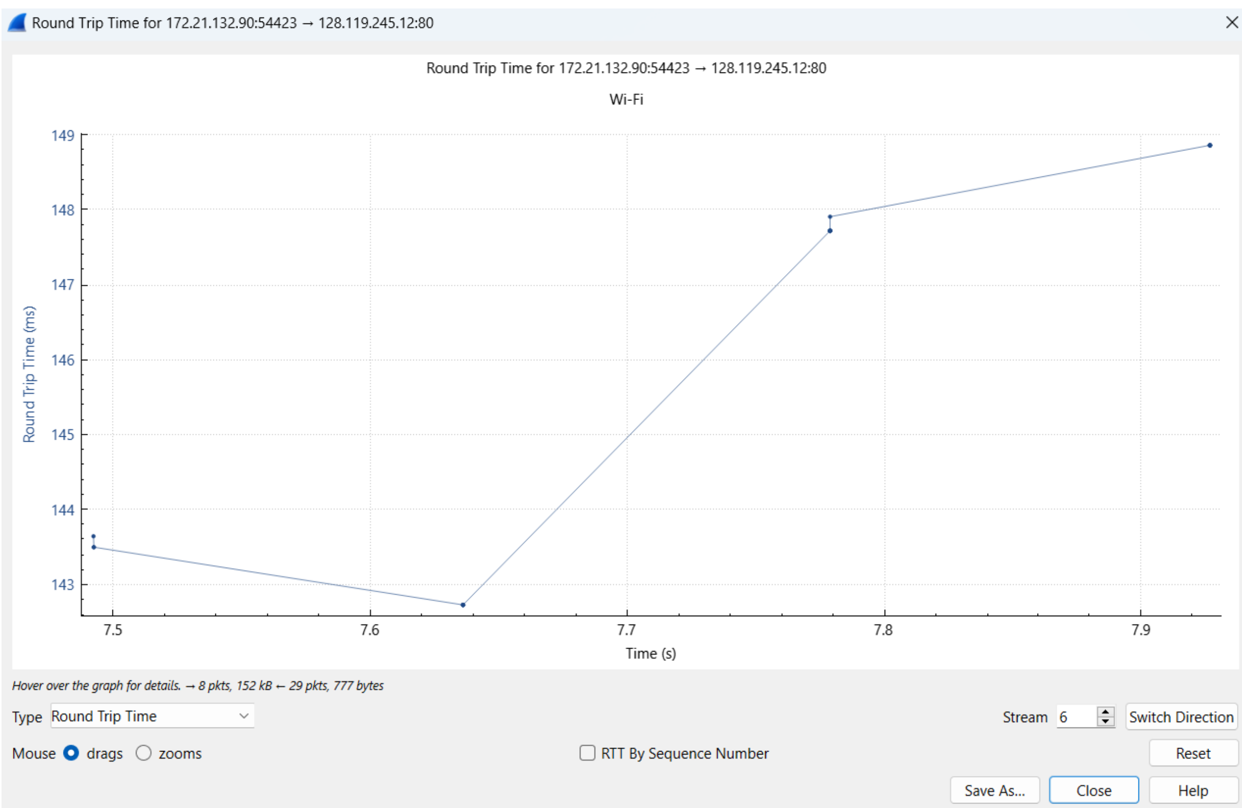
[10] segment loss not in post request



[11] payload

As we can see in the above figure, 55959 bytes were in the TCP payload and by Frame 4269, all of them were successfully transmitted.

1.2.2.



[12] RTT for request

We can infer from the graph that between t=7 and t=7.40, the round trip time has decreased from 143.7 to 140. After t=7.40 the round trip time linearly increased up to t=7.78 and then slowly linearly increased up to t=8.

1.2.3.

TCP-specific information, such as sequence numbers, acknowledgment numbers, window size, and TCP flags are apart from UDP information.

Sequence Numbers:

Explanation: Sequence numbers help organize and rebuild data at the receiving end, making sure everything is in the right order. Each piece of data in TCP has a sequence number, so even if things arrive a bit mixed up, they can be put back together correctly.
Significance: Sequence numbers ensure that data gets put back together accurately, even if it arrives in a jumbled order.

Acknowledgment Numbers:

Explanation: Acknowledgment numbers show what the next expected piece of data is, confirming that everything before it has been received successfully. They let the sender know that all the previous data has made it through.
Significance: Acknowledgment numbers are important for making sure data is intact and controlling how fast the sender sends more data based on what the receiver can handle.

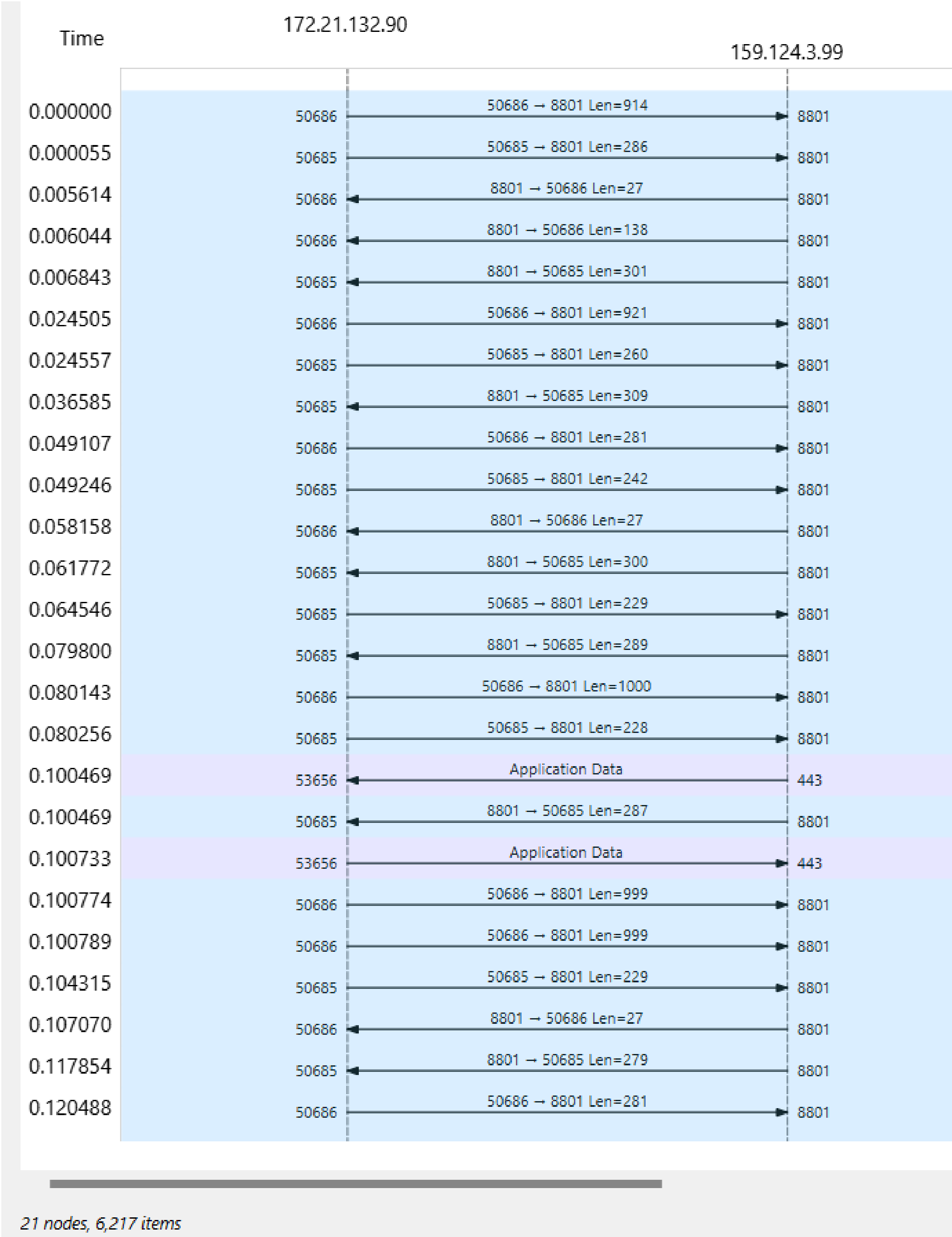
Window Size:

Explanation: Window size is like a traffic cop, deciding how much data the sender can send before it has to wait for a signal from the receiver. It's a way of managing how much space the receiver has to store incoming data.
Significance: Window size helps control the flow of data, preventing the sender from overwhelming the receiver and keeping communication running smoothly.

TCP Flags:

Explanation: TCP uses different flags, like SYN (start), ACK (confirm), FIN (finish), RST (reset), PSH (push), and URG (urgent), to send specific messages during communication. These flags help with setting up and ending connections, managing data flow, and handling special situations.
Significance: Flags are like communication signals that help in starting and ending connections, controlling data flow, and dealing with different situations in a network. For example, the SYN flag kicks off a connection during the initial handshake.

1.2.4.
TCP Flowchart:



[13] TCP flowchart

[14] UDP flow chart

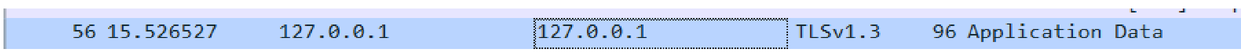
The flow graphs for TCP and UDP segments in Wireshark reveal key differences in how data is transmitted. TCP, being a reliable protocol, starts with a formal connection establishment (three-way handshake), sends data in an ordered sequence, and employs flow control mechanisms for reliable delivery. It often concludes with a formal connection termination. On the other hand, UDP, a connectionless protocol, lacks a formal connection setup, transmits data independently and without guaranteed order, and does not use flow control. It does not have a specific connection termination process. These differences underscore TCP's reliability and structured communication compared to UDP's lightweight, unordered, and connectionless nature.

2.1

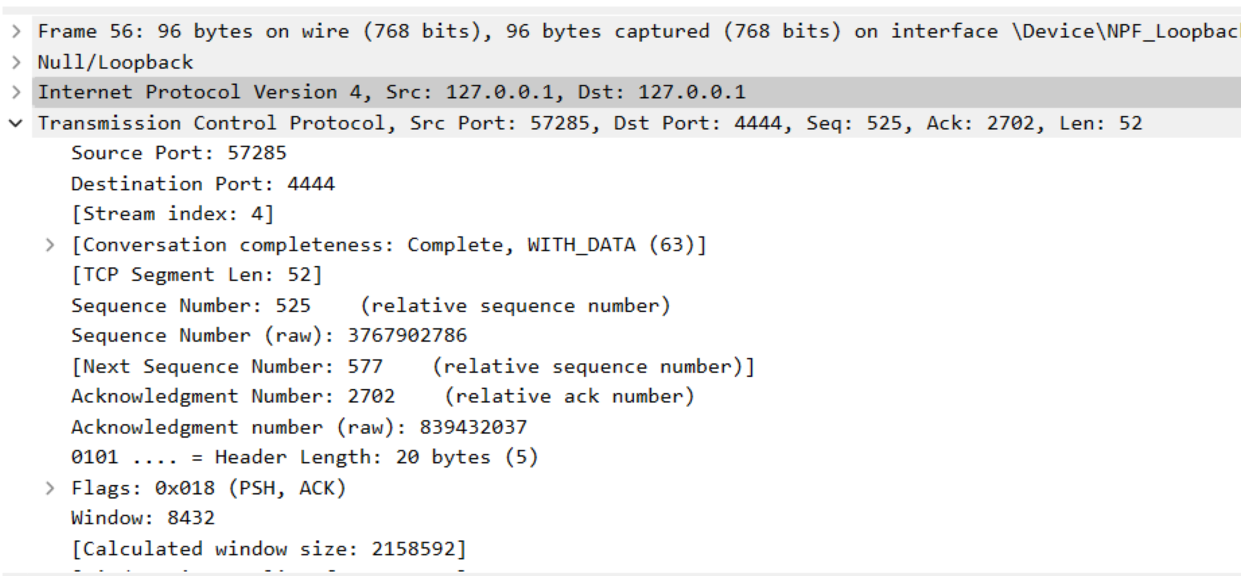
2.1.1

No. 56 - Client -> Server

SSL Protocol is used for server port 4444

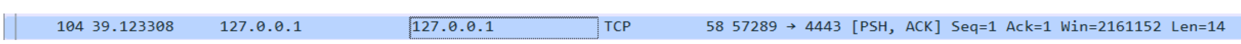


[15] Sent application data (not hello server)



[16] packet information

No. 104 - Client -> Server. Port 4443 for the server was used.



[17] tcp connection push

```
> Frame 104: 58 bytes on wire (464 bits), 58 bytes captured (464 bits) on interface \Device\NPF_Loopbac
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
v Transmission Control Protocol, Src Port: 57289, Dst Port: 4443, Seq: 1, Ack: 1, Len: 14
     Source Port: 57289
     Destination Port: 4443
     [Stream index: 7]
   > [Conversation completeness: Complete, WITH_DATA (31)]
     [TCP Segment Len: 14]
     Sequence Number: 1    (relative sequence number)
     Sequence Number (raw): 2106375648
     [Next Sequence Number: 15    (relative sequence number)]
     Acknowledgment Number: 1    (relative ack number)
     Acknowledgment number (raw): 237093383
     0101 .... = Header Length: 20 bytes (5)
   > Flags: 0x018 (PSH, ACK)
     Window: 8442
     [Calculated window size: 2161152]
```

[18] packet information for tcp push

2.1.2

We can see the sent message observing the TCP case:

```
02 00 00 00 45 00 00 36  ea cf 40 00 80 06 00 00   ····E··6 ··@·····
7f 00 00 01 7f 00 00 01  df c9 11 5b 7d 8c bd e0   ········ ···[}···
0e 21 c2 07 50 18 20 fa  0f 9c 00 00 37 32 38 34   ·!··P· · ····7284
37 43 4f 4d 50 34 31 36  0d 0a                     7COMP416 ··
```

[19] inferrable data of TCP push

But when we observe the hello statement of Client:

```
46 7.433431    127.0.0.1    127.0.0.1    TLSv1.3    968 Application Data
```

[20] SSL application data transmission

The body is:

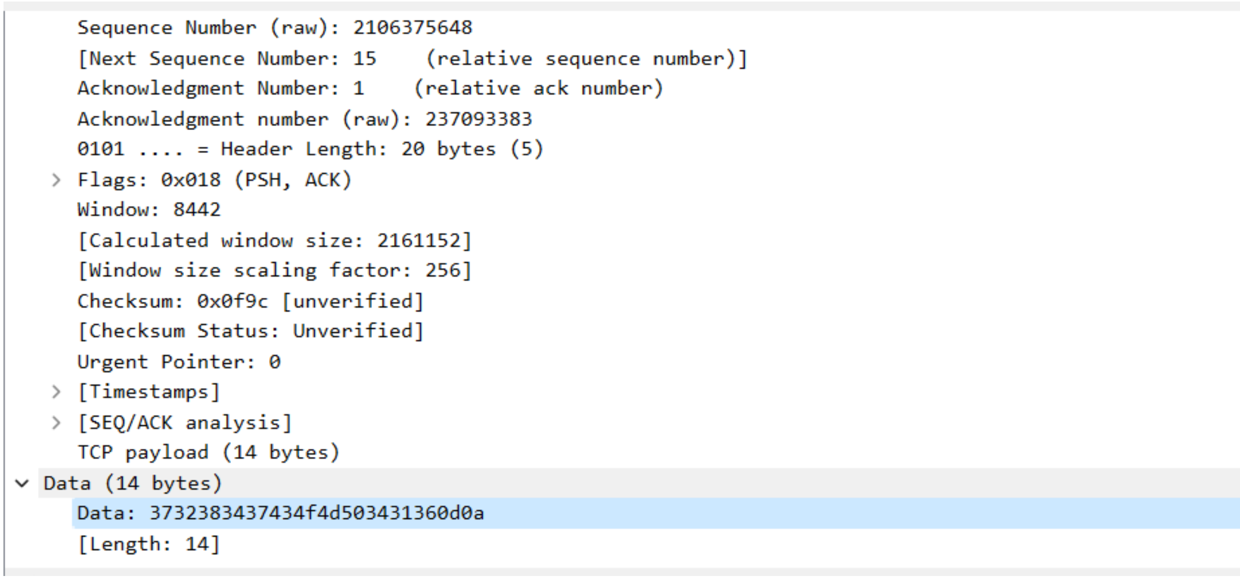
```
97 8f 25 27 45 85 ae f4  3b af d4 a3 72 51 94 83   ··%'E··· ;···rQ··
25 2e cd a3 33 e5 c2 b7  f5 8f ae b6 4c 4b c8 bb   %.··3··· ····LK··
33 20 8d ac 40 55 53 02  47 ff 81 b6 75 b5 a5 90   3 ··@US· G···u···
9b 5a fd 54 1f c8 fe 1e  20 10 70 d6 b5 54 cb 47   ·Z·T····  ·p··T·G
76 86 c2 9c e9 e6 3e 53  38 47 34 fa 35 71 3a 29   v·····>S 8G4·5q:)
0d 88 4b d7 70 12 7d ad  19 d7 68 fe e6 53 db 82   ··K·p·}· ··h··S··
25 a5 93 fd 5d 3e 4b 55  59 c9 f1 d4 06 42 87 8d   %···]>KU Y····B··
8f d6 1d 32 8d d4 36 0c  9c 00 f3 c8 b8 ba e4 e8   ···2··6· ········
ca 02 bd 2b 9f ec ba 5f  21 05 dc 3d 00 eb 8f 88   ···+···_ !··=····
78 8f 13 87 5e 3e 79 a3  41 51 7b 0d 5a 62 2b 70   x···^>y· AQ{·Zb+p
b7 20 e1 fc 82 4d 9d 12  02 8e a0 d7 06 c3 77 2b   · ···M·· ······w+
9b ea 06 90 1d c2 d3 61  02 1b 75 d5 58 1b ae 83   ·······a ··u·X···
d5 0c b1 02 26 5b a6 6a  60 6e e6 fe 86 8d 2b d6   ····&[·j `n····+·
3f 18 e7 57 a3 5f 0b 76  6f 6a 60 ce b0 49 23 94   ?··W·_·v oj`··I#·
14 ec d2 dc bf f9 a9 30  cf a6 0c 3b 89 fe 75 14   ·······0 ···;··u·
20 ab 71 b9 95 62 5c 86  1e f4 d1 54 03 2b e2 50    ·q··b\· ···T·+·P
d4 fd 2d 20 93 19 9c e9  29 18 41 37 52 da 99 da   ··- ···· )·A7R···
dd 8c 69 72 ed b1 b2 d0  ff a4 8f ce 06 72 69 50   ··ir···· ·····riP
f0 2e 15 b7 05 5c 07 ac  eb 7e 08 5b 8d e7 80 dd   ·.···\·· ·~·[····
19 e5 18 6b 0a 0d 74 49  2d 63 92 37 6d 2b 36 9c   ···k··tI -c·7m+6·
92 c8 ae db 71 bf 61 4f  c3 fd 3c 48 51 75 11 f9   ····q·aO ··<HQu··
de 68 93 4e b2 33 82 96  f6 1b 07 4b 63 72 05 5d   ·h·N·3·· ···Kcr·]
1d 67 2f f0 31 70 ae 53  d4 1b 7f f3 f3 a2 14 49   ·g/·1p·S ·······I
```


[21] unattainable information for SSL data transmission

We cannot observe the text in the TLS secure case. The reason is because SSL has an encryption scheme and disallows data observation without the private key.

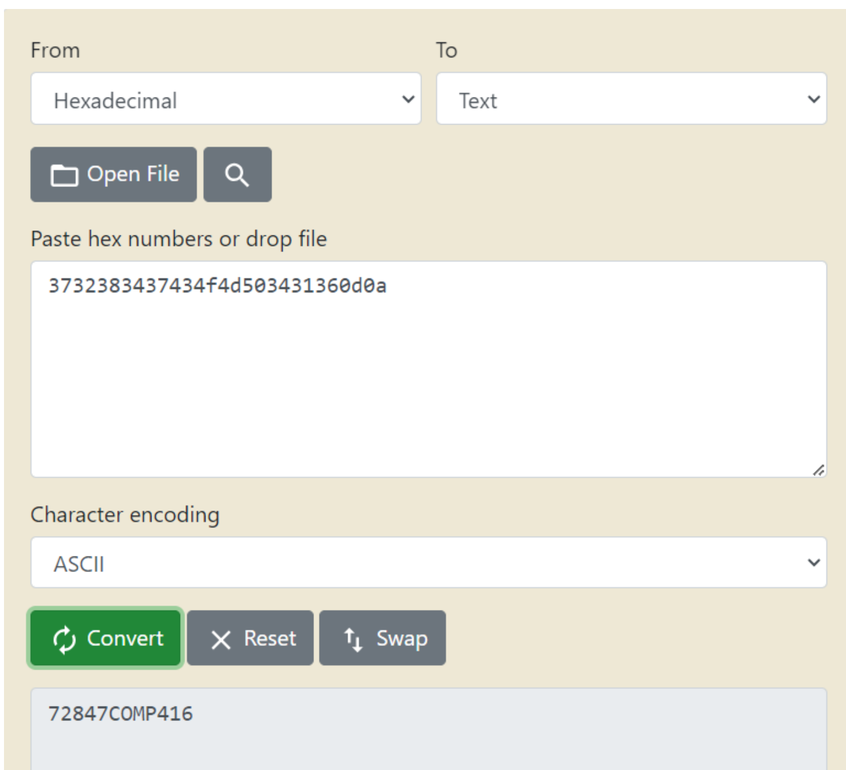
2.1.3

TCP Case:



[22] TCP connection data

Indeed, the data “3732383437434f4d503431360d0a” results in the string “72847COMP416” in ***ASCII***



[23] TCP Data converted to ASCII

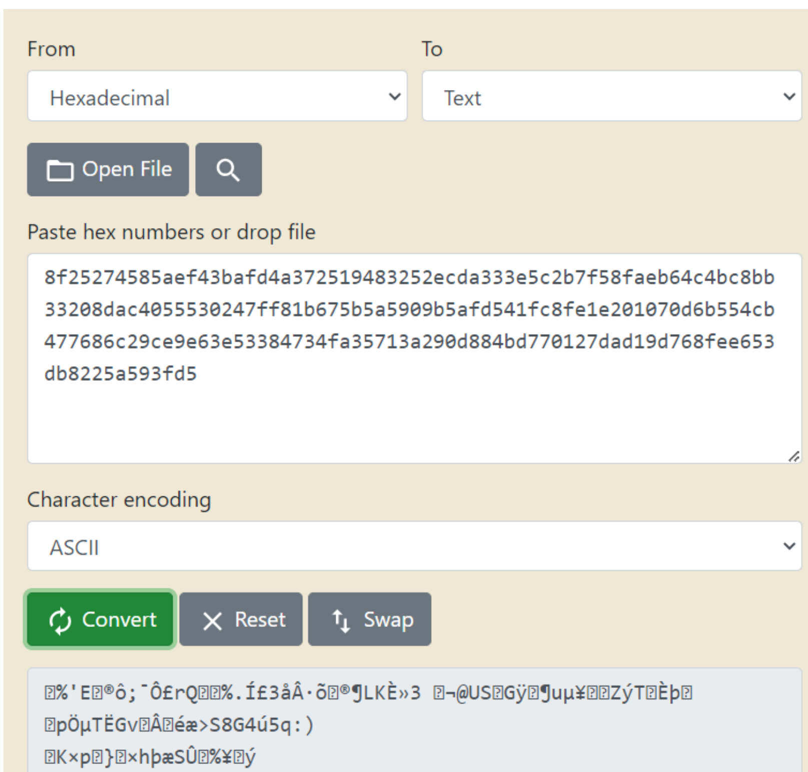
SSL Case:

```
      Acknowledgment number (raw): 3767902696
      0101 .... = Header Length: 20 bytes (5)
    > Flags: 0x018 (PSH, ACK)
      Window: 8441
      [Calculated window size: 2160896]
      [Window size scaling factor: 256]
      Checksum: 0x57d9 [unverified]
      [Checksum Status: Unverified]
      Urgent Pointer: 0
    > [Timestamps]
    > [SEQ/ACK analysis]
      TCP payload (924 bytes)
v Transport Layer Security
    v TLSv1.3 Record Layer: Application Data Protocol: Application Data
        Opaque Type: Application Data (23)
        Version: TLS 1.2 (0x0303)
        Length: 919
        Encrypted Application Data [truncated]: 8f25274585aef43bafd4a372519483252ecda333e5c2b7f58faeb64c
```

[24] SSL connection data

As we can observe, the application data is encrypted. “Change Cipher Spec” operation is also repeated so each time a possible adversary encounters a different scheme. Observing or interpreting the resulting encrypted data is meaningless.

As shown here;



[25] unattainable data from SSL transmission

2.1.4 - Using the below results in our java project we have created the following graph

SSL Case:

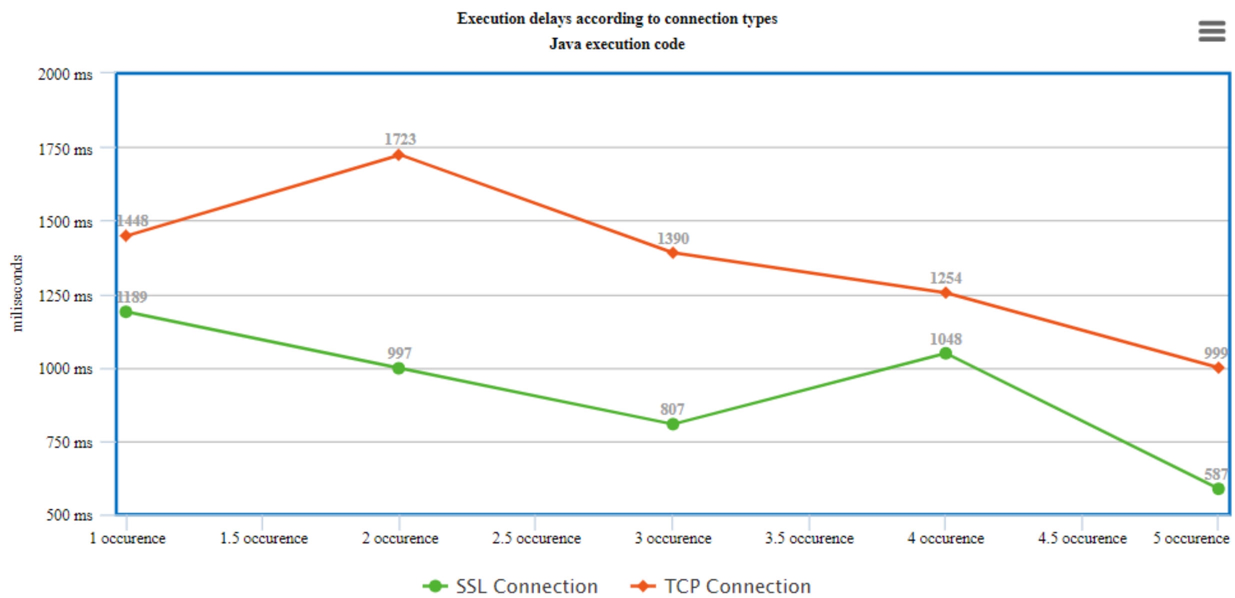
```
C:\Users\AHMET\.jdks\openjdk-21\bin\jav
Choose a service: 1 for SSL, 2 for TCP
1
Enter a message for echo:
72487COMP416
Server response: server_ack
Execution 1 Delay: 1189 ms
Enter a message for echo:
72487COMP416
Server response: server_ack
Execution 2 Delay: 997 ms
Enter a message for echo:
72487COMP416
Server response: server_ack
Execution 3 Delay: 807 ms
Enter a message for echo:
72487COMP416
Server response: server_ack
Execution 4 Delay: 1048 ms
Enter a message for echo:
72487COMP416
Server response: server_ack
Execution 5 Delay: 587 ms

Process finished with exit code 0
```

[26] 5 recurrence of connection for SSL

TCP Case:

```
Choose a service: 1 for SSL, 2 for TCP
2
Enter a message for echo:
72847COMP416
Server response: server_ack
Execution 1 Delay: 1448 ms
Enter a message for echo:
72847COMP416
Server response: server_ack
Execution 2 Delay: 1723 ms
Enter a message for echo:
72847COMP416
Server response: server_ack
Execution 3 Delay: 1390 ms
Enter a message for echo:
72847COMP416
Server response: server_ack
Execution 4 Delay: 1254 ms
Enter a message for echo:
72847COMP416
Server response: server_ack
Execution 5 Delay: 999 ms
```

[27] 5 recurrence of connection for TCP



[28] Plot for data received in [27] and [26]

The reason SSL cases showed quicker execution times than TCP might be because SSL has an initial setup (handshake) but then becomes more efficient for ongoing data transfer. Unlike TCP, SSL only performs the handshake once during connection setup, and if connection reuse methods are used, this overhead is spread across multiple requests. TCP has a slower start, causing higher delays at the beginning of a connection. Network factors like latency, bandwidth, and packet loss also impact SSL and TCP performance. Additionally, the specific implementation and optimizations in Java's SSL handling can influence the observed differences in delays. Analyzing the captured packets in Wireshark can give more details on SSL handshake, data transfer, and network traffic.

2.1.5

The "Client Hello" and "Server Hello" packets are part of the SSL/TLS handshake process when a client wants to securely connect to a server.

- Client Hello Packet: When a client initiates a connection, it sends a "Client Hello" packet to the server. This packet includes information like the supported SSL/TLS versions, supported cipher suites (encryption algorithms), and other details.

- Server Hello Packet: In response to the "Client Hello," the server sends a "Server Hello" packet. This packet contains information like the chosen SSL/TLS version and cipher suite. It's the server's way of saying, "I acknowledge your request, and here's how we'll communicate securely."

Observations:

- Client Hello: This is like the client saying "hello" and presenting a list of options for secure communication.

- Server Hello: The server responds by picking the best options from the client's list and establishing the specifics of the secure connection.

2.1.6

tls.handshake.type == 1

No.	Time	Source	Destination	Protocol	Length	Info
36	7.390527	127.0.0.1	127.0.0.1	TLSv1.3	472	Client Hello

[29] Client Hello TLS handshake

tls.handshake.type == 2

No.	Time	Source	Destination	Protocol	Length	Info
38	7.414787	127.0.0.1	127.0.0.1	TLSv1.3	171	Server Hello

[30] Server Hello TLS handshake

tls.handshake.type

No.	Time	Source	Destination	Protocol	Length	Info
36	7.390527	127.0.0.1	127.0.0.1	TLSv1.3	472	Client Hello
38	7.414787	127.0.0.1	127.0.0.1	TLSv1.3	171	Server Hello

[31] Server and Client TLS handshakes

As we can observe in the figures above, handshake types are different only in Server Hello and Client Hello. It shows the sequence of the handshake process, where each type has a specific role in creating a secure communication channel.

The protocol avoids sending the certificate for each connection by using a process called session resumption. In the initial handshake between the client and server, the server sends its certificate to the client. After this, the server and client can agree to resume the session in subsequent connections. During session resumption, the server doesn't need to send the certificate again; instead, both parties reuse the previously exchanged cryptographic parameters. This helps save bandwidth and speeds up the establishment of secure connections by skipping the redundant transmission of certificates.

```
v TLSv1.3 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 423
  v Handshake Protocol: Client Hello
      Handshake Type: Client Hello (1)
      Length: 419
      Version: TLS 1.2 (0x0303)
```

[32] Client Hello

```
∨ Transport Layer Security
  ∨ TLSv1.3 Record Layer: Handshake Protocol: Server Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 122
    ∨ Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 118
        Version: TLS 1.2 (0x0303)
```

[33] Server Hello

Moreover, we previously observed packets that had content "Application Data" and "Change Cipher". This means the handshake is still being used.

2.2

After setting the path of my Java JRE, I followed the steps in the oracle link. Password used to create: **AHMET3994**

```
Windows PowerShell
PS C:\Users\AHMET\Desktop\keystore> ls
PS C:\Users\AHMET\Desktop\keystore> keytool -genkey -alias mykey -keyalg RSA -keypass changeit -keystore keystore.jks
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  AHMET UYAR
What is the name of your organizational unit?
  [Unknown]:  Koc University
What is the name of your organization?
  [Unknown]:  Koc
What is the name of your City or Locality?
  [Unknown]:  Istanbul
What is the name of your State or Province?
  [Unknown]:  Sariyer
What is the two-letter country code for this unit?
  [Unknown]:  TR
Is CN=AHMET UYAR, OU=Koc University, O=Koc, L=Istanbul, ST=Sariyer, C=TR correct?
  [no]:  yes
```

[34] Keystore generation using keytool

Performing a cat, we have:

```
PS C:\Users\AHMET\Desktop\keystore> cat .\keystore.jks
```

[35] performing cat

```
          *†H†÷

0n1
0       UTR10Sariyer10Istanbul1
0
U
Koc10U
Koc University10U
AHMET UYAR0
231208125809Z
240307125809Z0n1
0       UTR10Sariyer10Istanbul1
0
U
Koc10U
Koc University10U
AHMET UYAR0,"0
          *†H†÷
```

[36] data successfully entered in keystore

After we have exported and gained the certificate, we then import to create the truststore cacerts.jks

```
PS C:\Users\AHMET\Desktop\keystore> keytool -import -v -trustcacerts -alias mykey -keypass AHMET3994 -file mykey.cer -keystore cacerts.jks -storepass AHMET3
994
Owner: CN=AHMET UYAR, OU=Computer Engineering Faculty, O=Faculty of Engineering, L=Istanbul, ST=Sariyer, C=TR
Issuer: CN=AHMET UYAR, OU=Computer Engineering Faculty, O=Faculty of Engineering, L=Istanbul, ST=Sariyer, C=TR
Serial number: 199a44fe
Valid from: Fri Dec 08 16:07:07 EET 2023 until: Thu Mar 07 16:07:07 EET 2024
Certificate fingerprints:
         SHA1: 9E:C3:D5:B8:AB:3A:50:B1:05:18:0D:B9:35:58:6F:F3:A5:7E:E2:FB
         SHA256: 51:C1:0C:6E:3D:C4:96:D8:D1:A8:AA:B1:E1:CF:FF:15:E0:C9:E4:34:C5:75:3A:64:BE:FE:AF:EF:14:52:D4:45
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 4A 8A A6 FB 40 87 96 AF   55 80 91 D3 88 F8 B6 B5  J...@...U.......
0010: 5E 10 27 ED                                        ^.'.
]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
[Storing cacerts.jks]
PS C:\Users\AHMET\Desktop\keystore>
```

[37] creating truststore using the created certificate and keystore

As a result, the truststore is created. We have now three files that will be added to the project deliverables.

```
    Directory: C:\Users\AHMET\Desktop\keystore


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         8.12.2023     16:26           1025 cacerts.jks
-a----         8.12.2023     16:07           2316 keystore.jks
-a----         8.12.2023     16:23            963 mykey.cer


PS C:\Users\AHMET\Desktop\keystore> |
```

[38] keystore, certificate, and truststore

REMINDER: Password used to create: **AHMET3994**

Deliverables
Keystore.jks: keystore file
Mykey.cer: certificate
Cacerts.jks: truststore

IMPORTANT TO NOTE: The pcapng file provided in the deliverables contains both the SSL and TCP packets

REFERENCES

[23], [25] - https://www.rapidtables.com/convert/number/hex-to-ascii.html