24.2.2023

IDATT2104 - Datakom – Arbeidskrav 1

Content:

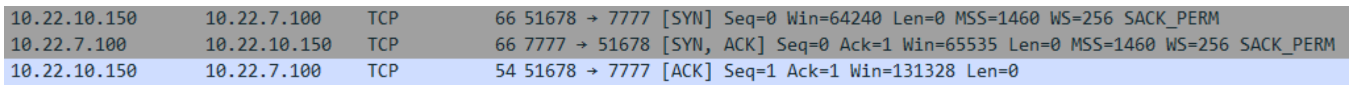
GR.NR 11

Report

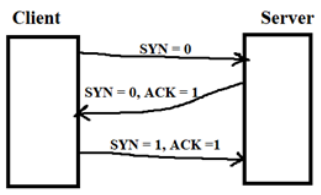
P3: Server – client relationship using TCP/Sockets

In order to create a solution, we use 2 programs - one for client and one for server. Both programs are using the Socket class in Java to communicate with one another. The socket itself is a combination of an ip-address and a port which the client and server use to establish connections and talk to each other. In our case, the server waits for a connection in an infinite while-loop using a specified port (in our case 7777) which it continuously listens to. This is done using the accept() method inside of the server class. When a connection is finally established, another while-loop is used to listen to messages from the client and to handle calculations. In more advanced server configurations, threads are also created and used to handle multiple clients connecting to the same server. The main methods of communication such as receiving and sending messages/notifications are performed using standard IO classes such as InputStreamReader, BufferedReader, PrintWriter. InputStreamReader is responsible for catching the messages sent between a client and a server, while BufferedReader is the one who takes these messages as input and reads them into the program. The PrintWriter class is responsible for handling messages which are sent back to the client either upon a successful calculation or an error.

Here is a demonstration of the 3-way handshake between the client and the server in order to establish connection, note that the client has an IP-address of 10.22.10.150 while the server uses the IP-address of 10.22.7.100. This will be the same for all scenarios which are tested:

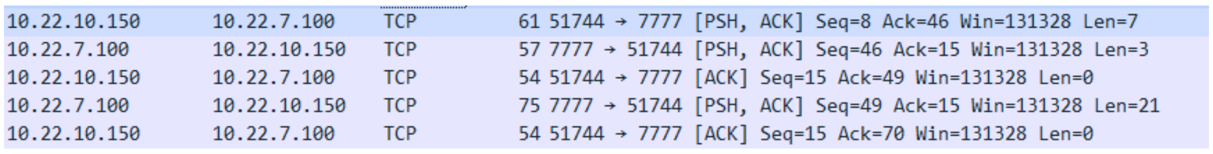


```
10.22.10.150      10.22.7.100       TCP         66 51678 → 7777 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
10.22.7.100       10.22.10.150      TCP         66 7777 → 51678 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
10.22.10.150      10.22.7.100       TCP         54 51678 → 7777 [ACK] Seq=1 Ack=1 Win=131328 Len=0
```



This is what we should see in Wireshark upon successful communication. The connection is using TCP protocol and performs a 3-way handshake. Communication between server and clients goes with the help of TCP protocols.

You can see communication between server and client below. Using Wireshark we can see packets that server sends and recieves from the customer:



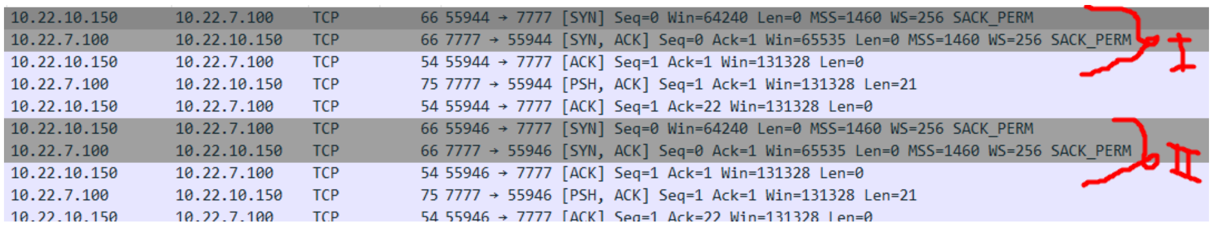
```
10.22.10.150    10.22.7.100     TCP    61 51744 → 7777 [PSH, ACK] Seq=8 Ack=46 Win=131328 Len=7
10.22.7.100     10.22.10.150    TCP    57 7777 → 51744 [PSH, ACK] Seq=46 Ack=15 Win=131328 Len=3
10.22.10.150    10.22.7.100     TCP    54 51744 → 7777 [ACK] Seq=15 Ack=49 Win=131328 Len=0
10.22.7.100     10.22.10.150    TCP    75 7777 → 51744 [PSH, ACK] Seq=49 Ack=15 Win=131328 Len=21
10.22.10.150    10.22.7.100     TCP    54 51744 → 7777 [ACK] Seq=15 Ack=70 Win=131328 Len=0
```

We can see all important information using the last column of pictures sent: takes, sequence number, flags and acknowledgement number. So in our situation, it is obvious that the server should get the data from the client before calculating it (first line) and the response will end with an ACK packet which shows us that data was received successfully from the client side after computation.

P3: TCP/Socket connection using multithreaded server

In addition to being able to handle only a single client, a more advanced variation was created in order to handle multiple clients connected to the server simultaneously. This was achieved by extending the server class with a standard thread implementation and allowing each thread to create a connection to the server for each client that requests to communicate. As a result, each client connection is independent from one another and the server can continue to remain operational even when one client chooses to disconnect. This also allows us to handle a significantly larger workload by allowing the possibility of several clients requesting calculator computations simultaneously.

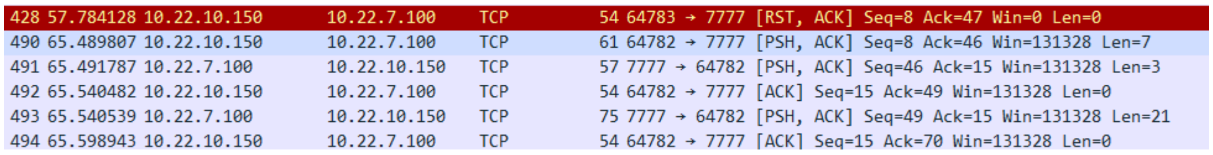
Below is a picture of Wireshark where we can observe two clients simultaneously connecting to the server and performing their 3-way handshake to establish the connection:



```
10.22.10.150    10.22.7.100     TCP    66 55944 → 7777 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
10.22.7.100     10.22.10.150    TCP    66 7777 → 55944 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
10.22.10.150    10.22.7.100     TCP    54 55944 → 7777 [ACK] Seq=1 Ack=1 Win=131328 Len=0
10.22.7.100     10.22.10.150    TCP    75 7777 → 55944 [PSH, ACK] Seq=1 Ack=1 Win=131328 Len=21
10.22.10.150    10.22.7.100     TCP    54 55944 → 7777 [ACK] Seq=1 Ack=22 Win=131328 Len=0
10.22.10.150    10.22.7.100     TCP    66 55946 → 7777 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
10.22.7.100     10.22.10.150    TCP    66 7777 → 55946 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
10.22.10.150    10.22.7.100     TCP    54 55946 → 7777 [ACK] Seq=1 Ack=1 Win=131328 Len=0
10.22.7.100     10.22.10.150    TCP    75 7777 → 55946 [PSH, ACK] Seq=1 Ack=1 Win=131328 Len=21
10.22.10.150    10.22.7.100     TCP    54 55946 → 7777 [ACK] Seq=1 Ack=22 Win=131328 Len=0
```

Once the connection is established, the clients can communicate and request the server to perform any computations they may need at the time. The computation from one client will not affect the computation of another, and the clients do not have to wait for the server to process the first client's request before being able to handle the second client.

In the screenshot below I will show that the server will continue to work even if one client disconnects, while the second client will continue to work with server:



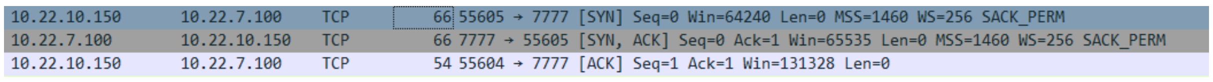
```
428 57.784128 10.22.10.150    10.22.7.100     TCP     54 64783 → 7777 [RST, ACK] Seq=8 Ack=47 Win=0 Len=0
490 65.489807 10.22.10.150    10.22.7.100     TCP     61 64782 → 7777 [PSH, ACK] Seq=8 Ack=46 Win=131328 Len=7
491 65.491787 10.22.7.100     10.22.10.150    TCP     57 7777 → 64782 [PSH, ACK] Seq=46 Ack=15 Win=131328 Len=3
492 65.540482 10.22.10.150    10.22.7.100     TCP     54 64782 → 7777 [ACK] Seq=15 Ack=49 Win=131328 Len=0
493 65.540539 10.22.7.100     10.22.10.150    TCP     75 7777 → 64782 [PSH, ACK] Seq=49 Ack=15 Win=131328 Len=21
494 65.598943 10.22.10.150    10.22.7.100     TCP     54 64782 → 7777 [ACK] Seq=15 Ack=70 Win=131328 Len=0
```

The first aborts the connection and the second client sends a response and receives an answer from the server. You can see that there are different clients using the source port. Client with port 64783 aborts while second client 64782 continues to communicate with the server.

P3: Simple web server

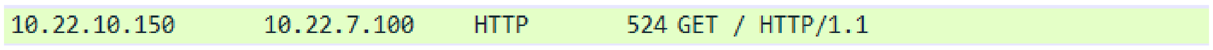
We already described the connection at the top and how it happens using java. The only difference with the previous exercise is that the client sends HTTP response for server and server responds using HTTPS.

Here we can see the same 3-way handshake (server running on port 7777):



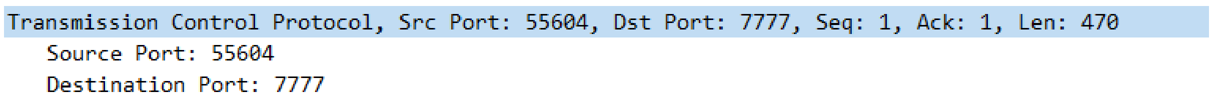
```
10.22.10.150    10.22.7.100     TCP     66 55605 → 7777 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
10.22.7.100     10.22.10.150    TCP     66 7777 → 55605 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
10.22.10.150    10.22.7.100     TCP     54 55604 → 7777 [ACK] Seq=1 Ack=1 Win=131328 Len=0
```

Afterwards, the client sends HTTP requests to the server with a GET message:



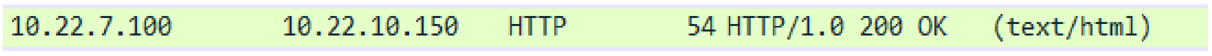
```
10.22.10.150     10.22.7.100     HTTP      524 GET / HTTP/1.1
```

Here we can see source and destination of this HTTP in header of TCP:



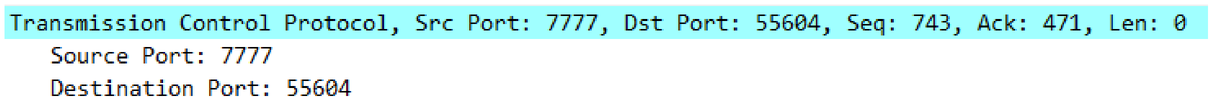
```
Transmission Control Protocol, Src Port: 55604, Dst Port: 7777, Seq: 1, Ack: 1, Len: 470
    Source Port: 55604
    Destination Port: 7777
```

And after this client will respond that HTTP response is OK and returns all text that we want to output on the webpage (text from html file):



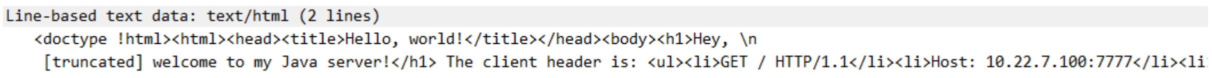
```
10.22.7.100        10.22.10.150    HTTP         54 HTTP/1.0 200 OK    (text/html)
```

We can see that it is response from server from TCP header:



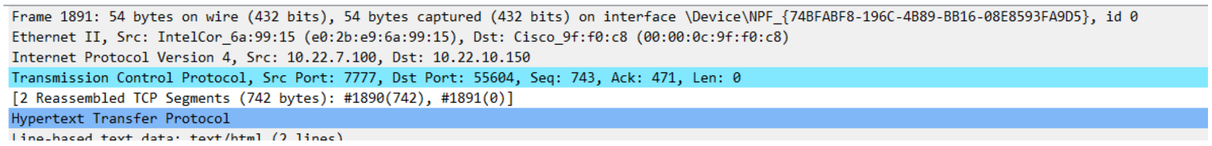
```
Transmission Control Protocol, Src Port: 7777, Dst Port: 55604, Seq: 743, Ack: 471, Len: 0
    Source Port: 7777
    Destination Port: 55604
```

We send simple text over socket here and this text is also in this server's response body and will be shown on the web page that we are opening:



```
Line-based text data: text/html (2 lines)
    <doctype !html><html><head><title>Hello, world!</title></head><body><h1>Hey, \n
     [truncated] welcome to my Java server!</h1> The client header is: <ul><li>GET / HTTP/1.1</li><li>Host: 10.22.7.100:7777</li><li
```

Here is the whole response, which also includes the MAC addresses of the source and destination (IntelCor_6a:99:15 and Cisco_9f:f0:c8 respectively):

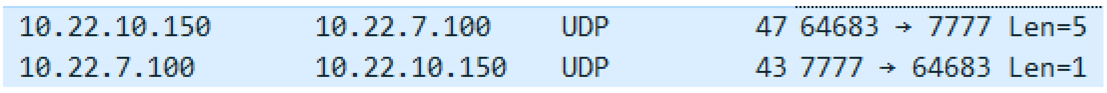


```
Frame 1891: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface \Device\NPF_{74BFABF8-196C-4B89-BB16-08E8593FA9D5}, id 0
Ethernet II, Src: IntelCor_6a:99:15 (e0:2b:e9:6a:99:15), Dst: Cisco_9f:f0:c8 (00:00:0c:9f:f0:c8)
Internet Protocol Version 4, Src: 10.22.7.100, Dst: 10.22.10.150
Transmission Control Protocol, Src Port: 7777, Dst Port: 55604, Seq: 743, Ack: 471, Len: 0
[2 Reassembled TCP Segments (742 bytes): #1890(742), #1891(0)]
Hypertext Transfer Protocol
Line-based text data: text/html (2 lines)
```

P4: Server – client connections using UDP

In this case, the approach is similar to creating the calculator using TCP. However, we are now utilizing Java classes which use UDP connection. The main difference between the connection types is that TCP is connection-oriented, whereas UDP is connectionless. In simpler terms, TCP prioritizes safer data transfer with retransmission of lost data packets in case of a communication issue and UDP prioritizes speed and transferring over as much data as possible. The most noticeable difference between the two is the lack of a 3-way handshake in the UDP protocol which serves as a means to confirm that the connection is successfully established and operational. This can be observed using Wireshark to see how a server and client communicate via the UDP protocol.

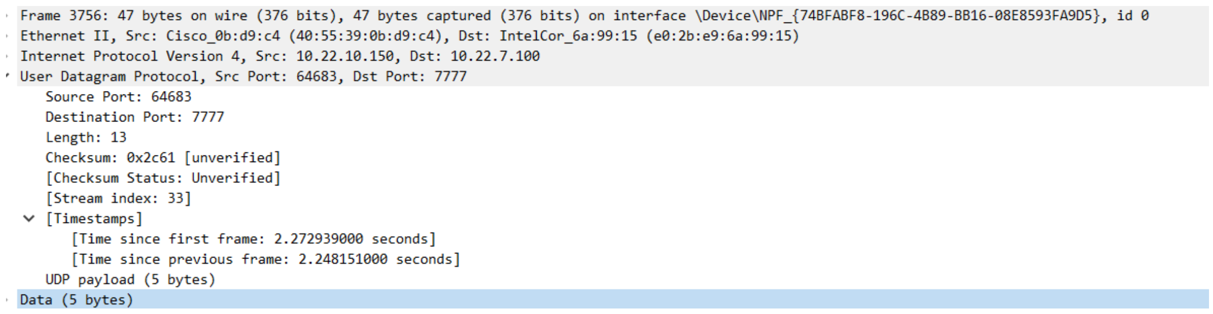
Here below you can see the transfering of data between server and client. The client sends a request for computation to the server and then receives a result and can use it for various purposes:



```
10.22.10.150       10.22.7.100     UDP         47 64683 → 7777 Len=5
10.22.7.100        10.22.10.150    UDP         43 7777 → 64683 Len=1
```

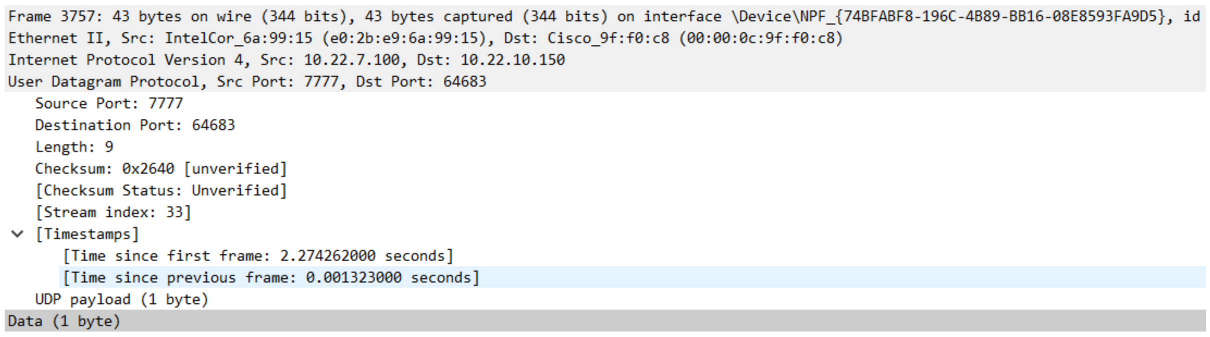
In Java we send packages using the DatagramPacket class, which takes a constructor buffer with data, buffer's size, destination address and the port of the server. Here is the main difference between a UDP and TCP implementation in java. For UDP, we need to know where to send response-packages back using request data packet's properties, but in TCP everything works with the help of the established 3-way handshake connection. Additionally, we should take care of the buffer size in Java's UDP implementation. If it is too small, data would not be fully transferred. Let's use a theoretical example: the buffer size was established by the buffer of the received package in Server. We get the equation "5+2" consisting of 3 bytes, but we need to send the string "Result is 7" which has 11 bytes. So the client will get a result like "Res". And if we send from the client equation of "5+22", the client will get the response "Resu". So it is really important to make the buffer of the server independent and to handle it correctly.
Here is a code example which demonstrates how to correctly handle the buffer:
//Constructor of data packet
DatagramPacket outputPacket = new DatagramPacket(b2, b2.length, datagramPacket.getAddress(), datagramPacket.getPort());
//Send this packet to destination that is already in properties using socket
datagramSocket.send(outputPacket);

Request from server (Data Packet header from example):



```
Frame 3756: 47 bytes on wire (376 bits), 47 bytes captured (376 bits) on interface \Device\NPF_{74BFABF8-196C-4B89-BB16-08E8593FA9D5}, id 0
Ethernet II, Src: Cisco_0b:d9:c4 (40:55:39:0b:d9:c4), Dst: IntelCor_6a:99:15 (e0:2b:e9:6a:99:15)
Internet Protocol Version 4, Src: 10.22.10.150, Dst: 10.22.7.100
User Datagram Protocol, Src Port: 64683, Dst Port: 7777
    Source Port: 64683
    Destination Port: 7777
    Length: 13
    Checksum: 0x2c61 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 33]
  ⌄ [Timestamps]
        [Time since first frame: 2.272939000 seconds]
        [Time since previous frame: 2.248151000 seconds]
    UDP payload (5 bytes)
Data (5 bytes)
```

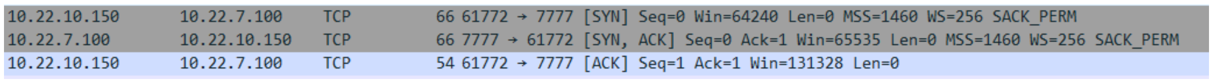
Response (Data Packet header from example):



```
Frame 3757: 43 bytes on wire (344 bits), 43 bytes captured (344 bits) on interface \Device\NPF_{74BFABF8-196C-4B89-BB16-08E8593FA9D5}, id
Ethernet II, Src: IntelCor_6a:99:15 (e0:2b:e9:6a:99:15), Dst: Cisco_9f:f0:c8 (00:00:0c:9f:f0:c8)
Internet Protocol Version 4, Src: 10.22.7.100, Dst: 10.22.10.150
User Datagram Protocol, Src Port: 7777, Dst Port: 64683
    Source Port: 7777
    Destination Port: 64683
    Length: 9
    Checksum: 0x2640 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 33]
  ⌄ [Timestamps]
        [Time since first frame: 2.274262000 seconds]
        [Time since previous frame: 0.001323000 seconds]
    UDP payload (1 byte)
Data (1 byte)
```

P4: Server – client communication with TLS

Because we are establishing a connection through TCP, the server and client would perform their typical 3-way handshake first in order to communicate:



```
10.22.10.150    10.22.7.100     TCP     66 61772 → 7777 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
10.22.7.100     10.22.10.150    TCP     66 7777 → 61772 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
10.22.10.150    10.22.7.100     TCP     54 61772 → 7777 [ACK] Seq=1 Ack=1 Win=131328 Len=0
```

Additionally because we are utilizing TLS, we would also be able to observe a TLS handshake between the two which consists of two main components:

- “Client hello” - The client sends to the server all important information about itself such as the supported TLS version, the supported cipher suites, and a string of random bytes known as the "client random."
- “Server Hello” - This is essentially the server’s response to a client’s greeting with the necessary information which will be used to secure the connection.

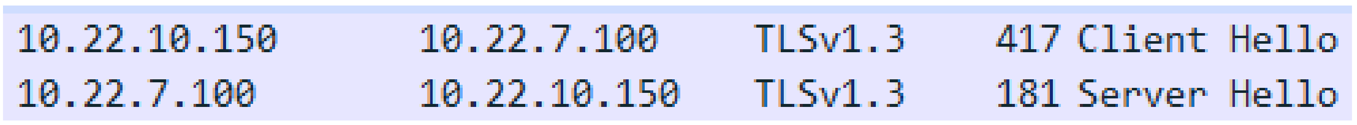
In order to have the client and server connect with TLS, we used our custom certificate that was created with the command keytool and which uses RSA encryption. We used the same certificate on both machines to create and establish secure connections.

Here is an example of a command for creating certificate which we used: keytool -genkey -alias signFiles -keyalg RSA -keystore examplestore. This command auto generates the keyStore and asks us simple questions to create a customized self-signed certificate.

We pass the certificate to server using system properties below :
-Djavax.net.ssl.keyStore=examplestore
-Djavax.net.ssl.keyStorePassword=YOUR_PASSWORD

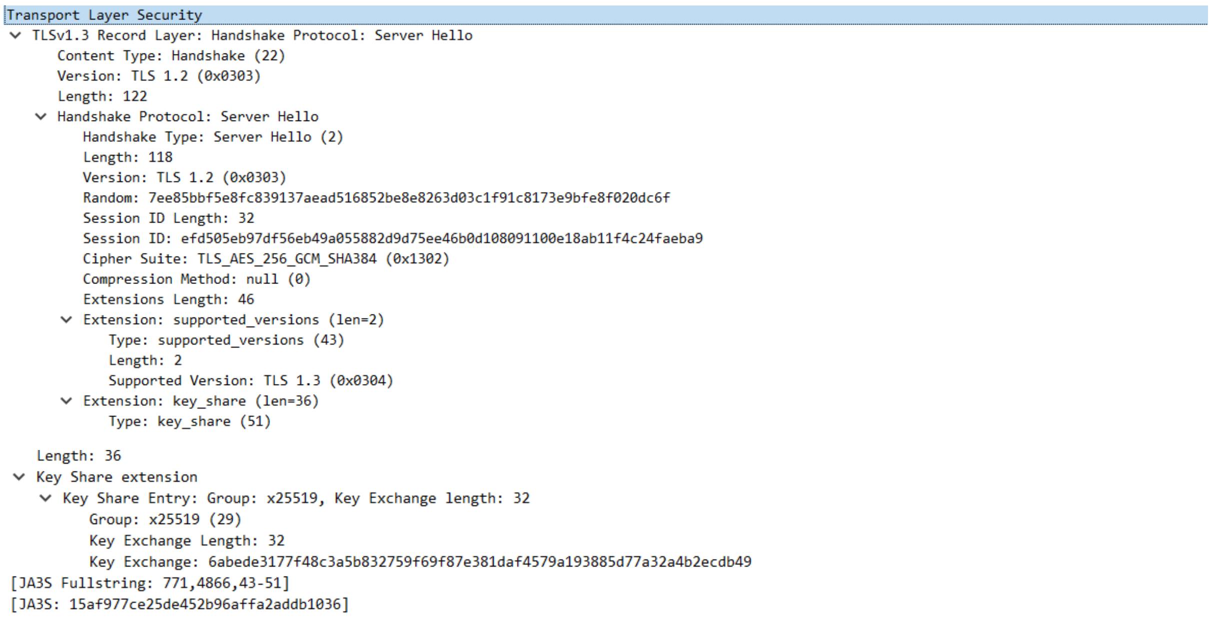
It is important to have a certificate in the same program or to show the absolute path to it, because Java system property for keystore does not support relative paths.
We should also force client to trust to our certificate using properties:
-Djavax.net.ssl.trustStore=examplestore
-Djavax.net.ssl.trustStorePassword=YOUR_PASSWORD

Client hello/Server hello between our pc:



```
10.22.10.150        10.22.7.100      TLSv1.3     417 Client Hello
10.22.7.100         10.22.10.150     TLSv1.3     181 Server Hello
```

Here is our “Server Hello” handshake which was observed using Wireshark:

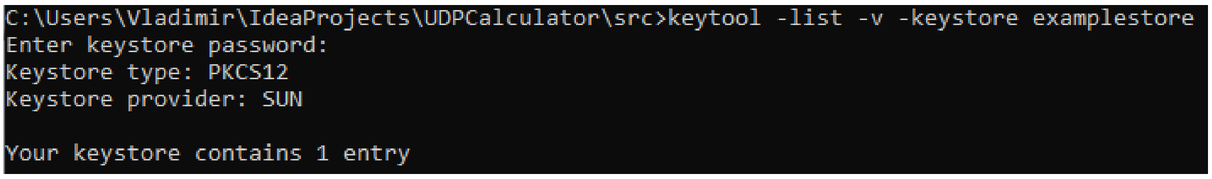


```
Transport Layer Security
v TLSv1.3 Record Layer: Handshake Protocol: Server Hello
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 122
  v Handshake Protocol: Server Hello
      Handshake Type: Server Hello (2)
      Length: 118
      Version: TLS 1.2 (0x0303)
      Random: 7ee85bbf5e8fc839137aead516852be8e8263d03c1f91c8173e9bfe8f020dc6f
      Session ID Length: 32
      Session ID: efd505eb97df56eb49a055882d9d75ee46b0d108091100e18ab11f4c24faeba9
      Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
      Compression Method: null (0)
      Extensions Length: 46
    v Extension: supported_versions (len=2)
        Type: supported_versions (43)
        Length: 2
        Supported Version: TLS 1.3 (0x0304)
    v Extension: key_share (len=36)
        Type: key_share (51)

  Length: 36
v Key Share extension
  v Key Share Entry: Group: x25519, Key Exchange length: 32
      Group: x25519 (29)
      Key Exchange Length: 32
      Key Exchange: 6abede3177f48c3a5b832759f69f87e381daf4579a193885d77a32a4b2ecdb49
[JA3S Fullstring: 771,4866,43-51]
[JA3S: 15af977ce25de452b96affa2addb1036]
```

We can see from the Server Hello data packet body that the TLS version which is being used here is 1.3. Due to this, we cannot see the certificate of this connection because the handshake messages containing the certificates are encrypted. However, we must also keep in mind that the certificates are being created by ourselves locally, which means that the server is not even sending out the certificate in our situation. The certificates which we have created can be found inside of the keystore, and are viewed using the following command from the command prompt:

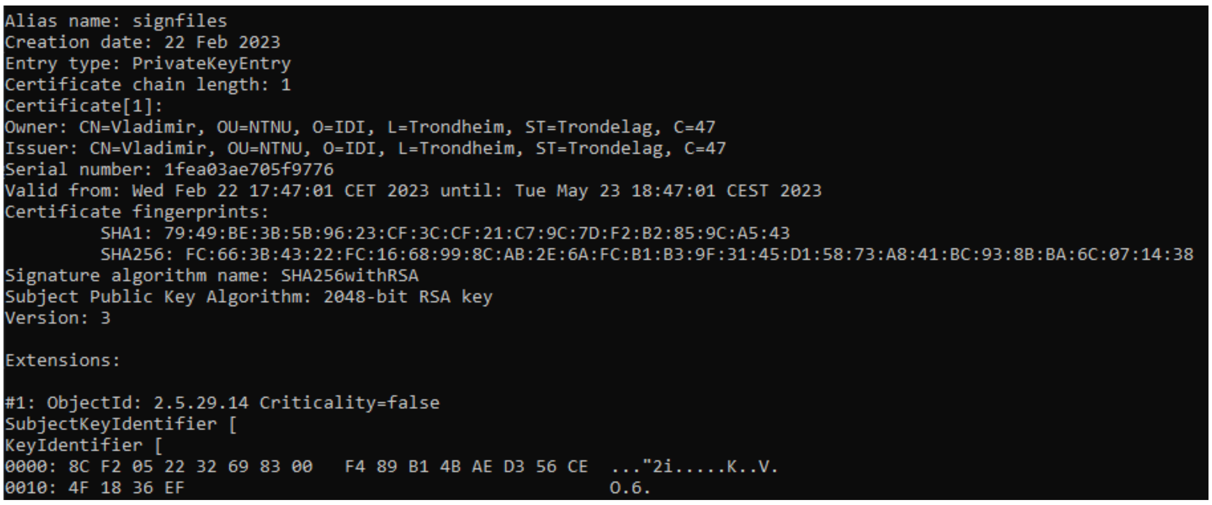
keytool -list -v -keystore examplestore



```
C:\Users\Vladimir\IdeaProjects\UDPCalculator\src>keytool -list -v -keystore examplestore
Enter keystore password:
Keystore type: PKCS12
Keystore provider: SUN

Your keystore contains 1 entry
```

With the following command, we can see the certificate which the connection makes use of that we have created ourselves:



```
Alias name: signfiles
Creation date: 22 Feb 2023
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Vladimir, OU=NTNU, O=IDI, L=Trondheim, ST=Trondelag, C=47
Issuer: CN=Vladimir, OU=NTNU, O=IDI, L=Trondheim, ST=Trondelag, C=47
Serial number: 1fea03ae705f9776
Valid from: Wed Feb 22 17:47:01 CET 2023 until: Tue May 23 18:47:01 CEST 2023
Certificate fingerprints:
         SHA1: 79:49:BE:3B:5B:96:23:CF:3C:CF:21:C7:9C:7D:F2:B2:85:9C:A5:43
         SHA256: FC:66:3B:43:22:FC:16:68:99:8C:AB:2E:6A:FC:B1:B3:9F:31:45:D1:58:73:A8:41:BC:93:8B:BA:6C:07:14:38
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: 8C F2 05 22 32 69 83 00   F4 89 B1 4B AE D3 56 CE  ..."2i.....K..V.
0010: 4F 18 36 EF                                        O.6.
```

From the picture above, we can find the certificate’s serial number, fingerprints and the algorithms used by the key to achieve encryption. This key is what keeps our connection secure from potential threats that may attempt to catch and read the transferred data.

In order to determine the protocol, encryption and hash algorithms of our connection, we are required to analyze the Cipher Suite which is sent in the “Server Hello” data package. The Cipher Suite contains a single line which tells us about the algorithms it uses and what kind of protocol the connection runs on. Inside of our package, we find the following Cipher Suite which contains the following sentence: TLS_AES_256_GCM_SHA384 (0x1302)

We can break down the sentence to determine the following information about the connection:

- TLS is the protocol that we are using to secure the connection, which essentially means that the connection contains some forms of cryptography in order to keep it secure from potentially unwanted threats. In our case, Wireshark specifies to us that the version of TLS in use by the connection is TLSv1.3.
- AES_256_GCM is the encryption algorithm, which is short for Advanced Encryption Standard and is the longest and the strongest level of symmetric encryption which the standard offers.
- SHA384 is the hashing algorithm which the connection utilizes, and is a hashing algorithm that belongs to the SHA-2 family of cryptographic

hashes. It produces a 384 bit digest of a message in order to protect the data by making it unreadable even if an attacker gains access to it.

During the TLS handshake, the two communicating parties also generate a session key at the start of the session. This key is used temporarily only for that specific session in order to perform symmetric encryption on the data sent between the two parties, and is discarded once the session ends. In order to create this key, the TLS handshake contains crucial information from both the client and the server which contains the components used to create a secure session key. Inside of the two main parts which we talked about (Server Hello and Client Hello), there are strings of random bytes attached with the data package. These are known respectively as the “client random” and “server random”. Additionally, the client sends another random string of bytes after the authentication of the SSL certificate. This last string of random bytes is known as the “premaster secret” and can only be decrypted using the private key located inside of the server. Finally, both the server and the client generate their session keys using the server random, the client random and the premaster secret and should arrive at the same results if everything up until this point has been performed correctly.