COMP416 Project-2

Transport Layer and ARQ Protocols Analysis with Wireshark

Ertan Can Güner 60610

Overview:

This project consists of 4 different parts; SSL Server/Client, Analyzing the SSL Application with Wireshark, Analyzing a UDP connection with Wireshark and implement a Stop-n-Wait ARQ Protocol and analyzing it with Wireshark.

SSL-Client:

SSL-Client has 3 main parts that has to accomplish. First, it should connect to the server with TCP and authenticate. Second, after the verification is done, it should receive a certificate from the server. Finally, after receiving the certificate client should connect to the server with SSL from a different port and it should decode the information from the server accordingly.

Constructor: Takes an address and port number as a parameter and processes the port number if it exceeds the available ports. It then calls three other methods: **ConnectToTCP(), CrtTransfer(), DisconnectFromTCP();**

```
public SSLConnectToServer(String address, int port) throws Exception
{

    this.serverAddress = address;
    if(port > 65535){
        System.out.println("[CLIENT][ERR]:Port number cannot be larger than the maximum number of ports which is 65535.");
        this.serverPort = port % 65535;
    }


    /*
    Loads the keystore's address of client
     */
    System.setProperty("javax.net.ssl.trustStore", KEY_STORE_NAME);

    // Loads the keystore's password of client
    System.setProperty("javax.net.ssl.trustStorePassword", KEY_STORE_PASSWORD);

    ConnectToTCP();
    CrtTransfer();
    DisconnectFromTCP();
    // Load the certificates in the key store
    Create_Key_Store();
}
```

ConnectToTCP: It connects to the server via port ‘5555’.

```
public void ConnectToTCP(){
    try{
        this.client = new Socket(this.serverAddress, 5555);
        System.out.println("[CLIENT][LOG]: Connection estabilised with the TCP Server.");
    }catch (UnknownHostException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

CrtTransfer: It first calls the Verification() method because the server verifies the client before it sends the certificate. The it initializes I/O Streams with the server. It reads from the isTcp (Input Stream) and writes the bytes to the osTcp (OutputStream) which forms the 'server_crt.crt' file.

```
public void CrtTransfer(){
    Verification();
    byte[] contents = new byte[1000];

    try{
        BufferedOutputStream osTcp = new BufferedOutputStream(new FileOutputStream("server_crt.crt"));
        InputStream isTcp = this.client.getInputStream();
        System.out.println("[CLIENT][LOG]: Starting certificate transfer.");
        int readBytes = 0;

        while((readBytes = isTcp.read(contents)) != -1){
            osTcp.write(contents, 0, readBytes);
        }
        osTcp.flush();
        System.out.println("[CLIENT][LOG]: Certificate transfer done.");
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Verification: Initializes I/O Streams from the server and a scanner for user input. The infinite while loop keeps receiving messages from the server and sends the user input to the server. If the message received from the server is '400' or '200' it means the verification failed (400) or succeed (200). If the messages are not that then the message is printed out.

```
public void Verification(){
    String msg = new String();
    String svMsg = new String();
    try{
        this.is=new BufferedReader(new InputStreamReader(this.client.getInputStream()));
        this.os= new PrintWriter(this.client.getOutputStream(), true);
        Scanner sc = new Scanner(System.in);

        while(true){
            svMsg = this.is.readLine();
            if(svMsg.equals("400") || svMsg.equals("200")){
                if(svMsg.equals("400")) {
                    System.out.println("[CLIENT][ERR]: Client verfication failed.");
                    break;
                }else{
                    System.out.println("[CLIENT][SUCC]: Client verfication success.");
                    break;
                }
            }else{
                System.out.println("[Server][RES]: " + svMsg);
            }
            msg = sc.nextLine();
            this.os.println(msg);
            this.os.flush();
        }
    }catch (IOException e) {
        e.printStackTrace();
    }
}
```

decodeMessages: This method is called in the Main.java file after the server and client greeted each other over SSL. The messages from the server are sent over 'encrypted'. To decrypt it, for loop scans the server message (svMsg) character by character and odd numbered characters are the first message and even numbered ones are the second message.

```
public void decodeMessages(){
    String[] msg = new String[2];
    String svMsg = new String();
    int index;

    try {
        while((svMsg = is.readLine()) != null){
            for(int i = 0; i < svMsg.length(); i++){
                index = i%2;
                if(i < 2) msg[i] = " ";
                if(svMsg.charAt(i) != ' ') {
                    switch(index){
                        case 0:
                            msg[0] += svMsg.charAt(i);
                            break;
                        case 1:
                            msg[1] += svMsg.charAt(i);
                            break;
                    }
                }
            }
        }
        System.out.println(msg[0] + "\n" + msg[1] + "\n");

    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

DisconnectFromTCP: It closes the TCP socket.

```
public void DisconnectFromTCP(){
    try{
        this.client.close();
        System.out.println("[CLIENT][LOG]Disconneted from TCP server.");
    }catch(IOException e){
        e.printStackTrace();
    }
}
```

Server:

Server should listen to two different ports, one over TCP for certificate transfer and one over SSL. To support multiple clients, connections must be assigned to a thread, which are crtClientThread and SSLServerThread.

Server Constructor: Takes a port number as a parameter and process' the port number if it exceeds a threshold. Creates a crtServerSocket over port '5555' and calls ListenAndAccept() method.

```
public SSLServer(int port)
{
    if(port > 65535){
        System.out.println("[SERVER][ERR]Port number cannot be larger than the maximum number of ports which is 65535");
        this.SSLServerPort = port % 65535;
    }else System.out.println("[SERVER][SUCC]Port number for the SSL Server is: " + this.SSLServerPort);

    try
    {

        //serverControlPanel = new ServerControlPanel("hello server!");


        /*
        Instance of SSL protocol with TLS variance
         */
        SSLContext sc = SSLContext.getInstance("TLS");

        /*
        Key management of the server
         */
        char ksPass[] = KS_PASS.toCharArray();
        KeyStore ks = KeyStore.getInstance("JKS");
        ks.load(new FileInputStream(KS_FILE), ksPass);
        KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
        kmf.init(ks, SK_PASS.toCharArray());
        sc.init(kmf.getKeyManagers(), null, null);


        /*
        SSL socket factory which creates SSLSockets
         */
        sslFactory = sc.getServerSocketFactory();
        sslSocket = (SSLServerSocket) sslFactory.createServerSocket(this.SSLServerPort);

        System.out.println("[SERVER][LOG]: SSL server is up and running on port " + this.SSLServerPort);

        this.crtServerSocket = new ServerSocket(this.CRTServerPort);

        System.out.println("[SERVER][LOG]: CRT server is up and running on port " + this.CRTServerPort);

        while (true)
        {
            ListenAndAccept();
        }
    }
```

ListenAndAccept: Listens two ports; one for certificate transfer and other for SSL connections. After it receives a connection from port '5555' it starts a crtClient thread. After the certificate is transferred the client can connect through SSL.

```
private void ListenAndAccept()
{
    SSLSocket s;
    //For incoming connections
    Socket clientSocket;
    try
    {
        /*Accepts the incoming connection from port number '5555' and sends the
          client to a crtClient thread to start the certificate transfer.
        */
        clientSocket = crtServerSocket.accept();
        System.out.println("[SERVER][LOG]: A TCP connection was established with a client on the address of " + clientSocket.getRemoteSocketAddress());
        crtClientThread crtTransfer = new crtClientThread(clientSocket);
        crtTransfer.start();

        s = (SSLSocket) sslSocket.accept();
        System.out.println("[SERVER][LOG]: An SSL connection was established with a client on the address of " + s.getRemoteSocketAddress());
        SSLServerThread st = new SSLServerThread(s);
        st.start();

    }

    catch (Exception e)
    {
        e.printStackTrace();
        System.out.println("[SERVER][ERR]: Server Class.Connection establishment error inside listen and accept function");
    }
}
```

crtClientThread Constructor: It takes the client socket as parameter and assigns it.

```
public crtClientThread(Socket client){
    this.client = client;
}
```

Run: First it calls the verifyClient() method to verify the client. After that if the verification is successful the I/O Streams are initialized to read the certificate file 'server_crt.crt' and sends it in chunks of size variable, which is 1000. After the file is sent to the client it closes the I/O Streams, sockets and closes the connection with the client.

```
public void run(){
    if(!verifyClient()){
        try{
            System.out.println("[SERVER][ERR]: Closing connection with the client: " + this.client.getRemoteSocketAddress());
            client.close();
            return;
        }catch(IOException e){
            e.printStackTrace();
        }
    }else System.out.println("[SERVER][SUCC]: Client verified: ");

    System.out.println("[SERVER][LOG]: Starting certificate transfer to the client: " + this.client.getRemoteSocketAddress());
    File file = new File("server_crt.crt");

    try{
        this.is = new BufferedInputStream(new FileInputStream(file));
        this.os = this.client.getOutputStream();

        byte[] contents;

        long current = 0;
        long fileSize = file.length();

        while (current != fileSize) {
            //Packet size
            int size = 1000;
            if (fileSize - current >= size) {
                current += size;
            } else {
                size = (int) (fileSize - current);
                current = fileSize;
            }
            contents = new byte[size];

            is.read(contents);
            os.write(contents);
        }
    }catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            os.flush();
            client.close();
            os.close();
            is.close();
            System.out.println("[SERVER][LOG]: File transfer done to the client: " + client.getRemoteSocketAddress());

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

verifyClient: It initializes I/O Streams to read from the database that holds the user information. Then it askes the client for their credentials to compare it. The while loop reads from the database line by line and differentiates the username from password with ':' character. The data is stored as [username:password] in the database. If the client fails to match the username or password '400' message is sent to notify the client that their credentials are wrond. '200' message is sent if verification is complete. The method return true or false regarding '200' or '400' message.

```
public boolean verifyClient(){
    boolean check = false;
    String username = new String();
    String password = new String();
    File users = new File("users.txt");

    try{
        this.osV = new PrintWriter(this.client.getOutputStream(), true);
        this.isV = new BufferedReader(new InputStreamReader(this.client.getInputStream()));
        BufferedReader userVer = new BufferedReader(new FileReader(users));
        System.out.println("[SERVER][LOG]: Client waiting to be verified.");
        String line;

        this.osV.println("Username: ");
        username = this.isV.readLine();

        this.osV.println("Password: ");
        password = this.isV.readLine();

        while((line = userVer.readLine()) != null){
            if (username.equals(line.substring(0, line.indexOf(":")))) {
                if (password.equals(line.substring((line.lastIndexOf(":") + 1)))) {
                    System.out.println("[SERVER][LOG]: User Verified!");
                    this.osV.println("200");
                    this.osV.flush();
                    check = true;
                } else {
                    System.out.println("[SERVER][ERR]: Incorrect password! User not verified.");
                    this.osV.println("400");
                    this.osV.flush();
                    check = false;
                }
            }else{
                System.out.println("[SERVER][ERR]: Incorrect username! User not found.");
                this.osV.println("400");
                this.osV.flush();
                check = false;
            }
        }
        userVer.close();
    }catch(IOException e){
        e.printStackTrace();
    }
    return check;
}
```

SSLServerThread: Only two methods are added to this thread. Which are; sendMessages() and getInfo().

sendMessages: It calls the getInfo() method to retrieve the user information in for of a List. The messages are sent with non-persistent behavior meaning that the information is divided into a fixed size and sent over one-by-one.

```
public void sendMessages(){
    try{
        String msg = new String();
        List<String> info = getInfo();
        int index = 0;
        while(!msg.equals("  ")){
            msg = "";
            for(String s: info){
                if(index > (s.length()- 1)) msg += " ";
                else msg += s.charAt(index);
            }

            this.os.write(msg);
            this.os.flush();
            index++;
        }



    }catch(IOException e){
        e.printStackTrace();
    }
}
```

getInfo: Initializes a input stream to read from the database and adds the information to the list and returns it.

```
public List<String> getInfo() {

    List<String> lst = new LinkedList<String>();
    String st = new String();

    try {
        BufferedReader emails = new BufferedReader(new FileReader(new File("users.txt")));
        while ((st = emails.readLine()) != null) {
            lst.add(st.substring(0, st.indexOf(":")));
            lst.add(st.substring((st.indexOf(":") + 1)));
        }
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return lst;
}
```

Questions:

1. Server: 127.0.0.1:5555 Client: 127.0.0.1:51117
2. The data in the TCP data fields are visible because they are not encrypted.
3. There seems to be 11 different packets that has encrypted Application data.
4. The information exchanged can be seen in the TCP packets but for the SSL packets the information is encrypted.
5. The Flow Graph shows the messages exchanged in an orderly fashion. The first three messages exchanged with the host is the 3-way handshake of the TCP that can be seen in Figure 1.

Figure 1: 3-way handshake of TCP

6. The port for the server is 80 and 51191 is for the client. 3154891349 (Fig.2) is the first sequence number, second is 3154891350 and 2739221609 is the third one.

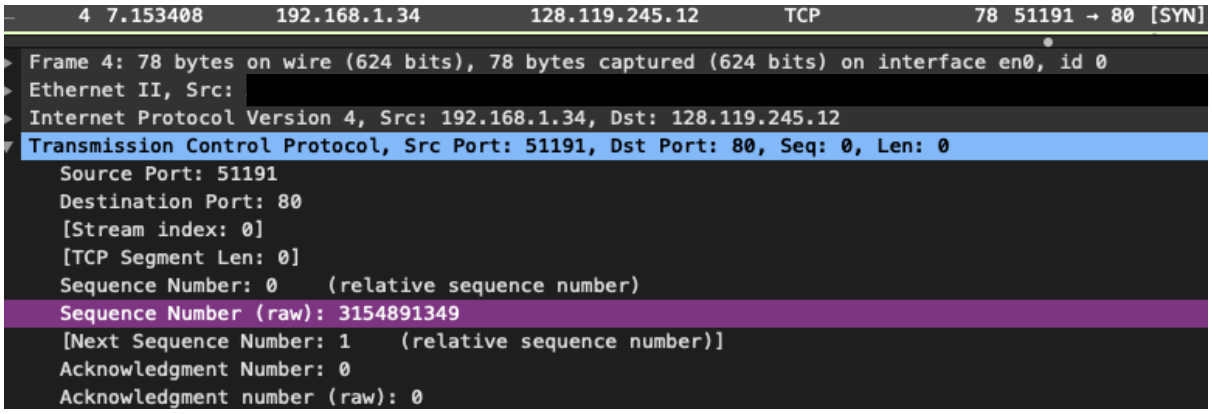

Figure 2: Port numbers for the client and server. First sequence number of the handshake.

7. 3154891350 (Fig.3) is the Sequence number of the SYNACK segment sent by the server in reply to the client's SYN.

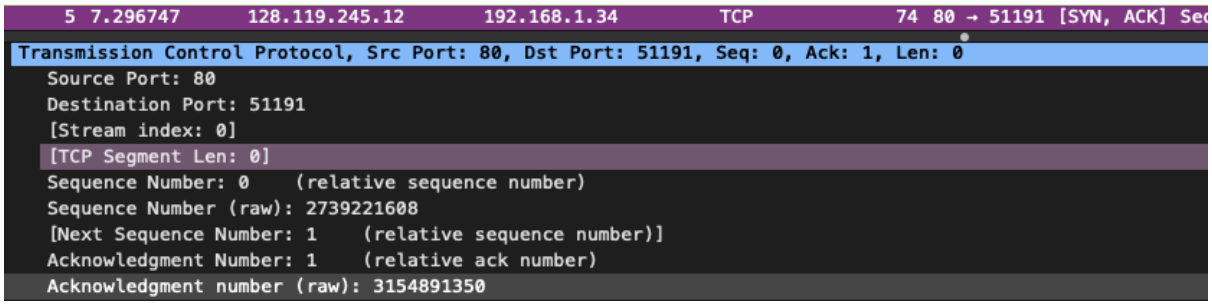

Figure 3: Sequence number of the SYNACK segment sent by the server.

8. The relative value of ACK is 1. When sequence and acknowledgment numbers are equal to 1 the SYNACK segment is made. ACK is calculated by adding one to the sequence field.

9. The sequence number is 3155030442 (Fig.4) for the POST command.

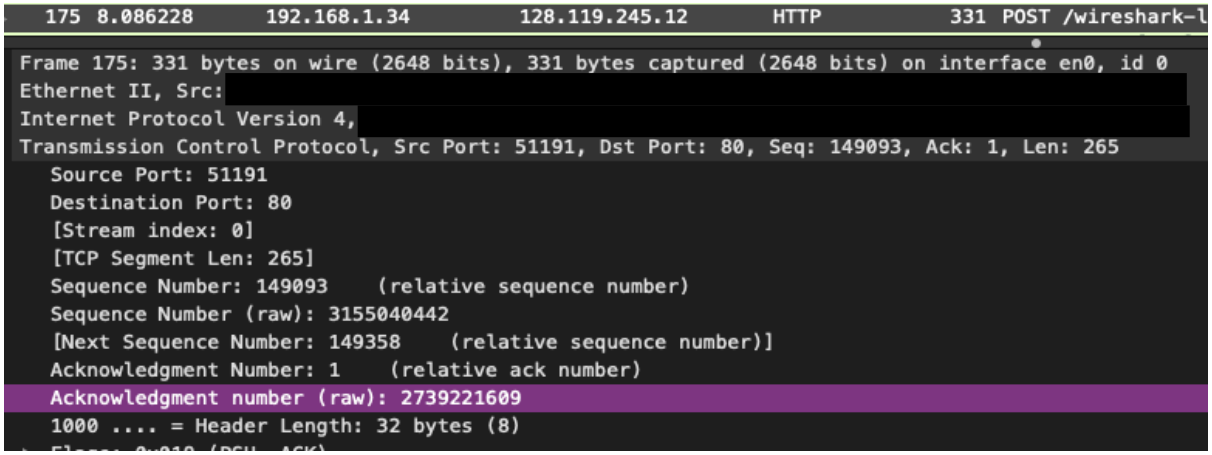

Figure 4: Sequence number of HTTP POST command.

10. The sequence numbers for the first 3 packets (4-5-6) are stated in question 6. 4th- 3154891350, 5th- 3154891958, 6th- 3154892122. The first six segments time stamps can be seen in Fig.5. The ACK for the 5th segment is received as 8th segment (Fig.6). The RTT can be retrieved from Wireshark if we change 'View->Time Display Format' to 'Seconds Since Previous Displayed Packet'. The SYNACK response of the server gives the RTT between the client and the server which is: 0.143ms.

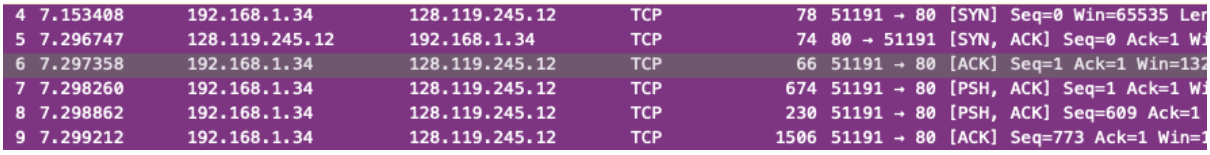

Figure 5: First 6 segments of the connection.

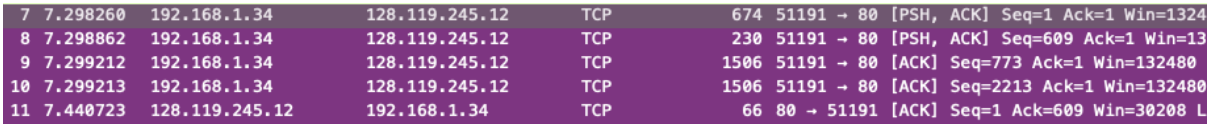

Figure 6: ACK for the 5th segment, received as the 8th segment.

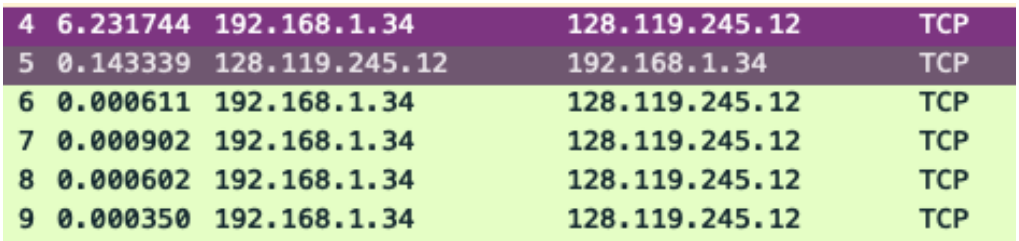

Figure 7: The time difference between recieved packets.

11. DNS filter is used to find the appropriate packets. Because nslookup uses DNS queries to find the information about the desired domain name.
12. nslookup uses DNS as application layer and UDP as transport layer. UDP is faster than TCP as in it doesn't require any handshake to establish connection. It doesn't rely on connection with any endpoint, hence why it is called connectionless.
13. When looking the packet information (Fig.8) of the exchanged messages. The Flag field of the DNS query states to do the query recursively. The advantage of the Iterative lookup is that the burden is put on the client-side rather in Recursive lookup the burden may be put at the top level DNS servers.
14. The header length is 20 bytes. (Fig.9)

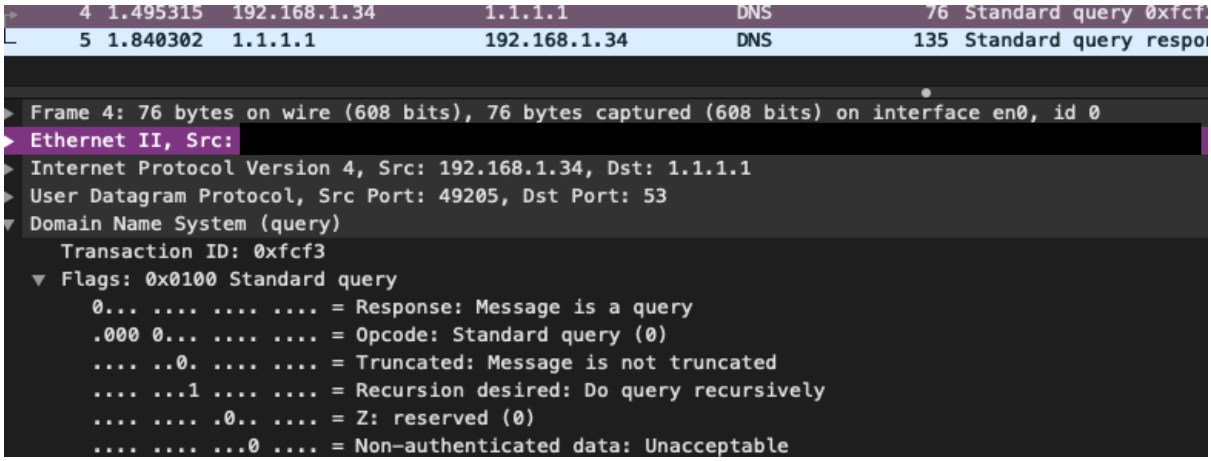

Figure 8: Flag field of the DNS query.

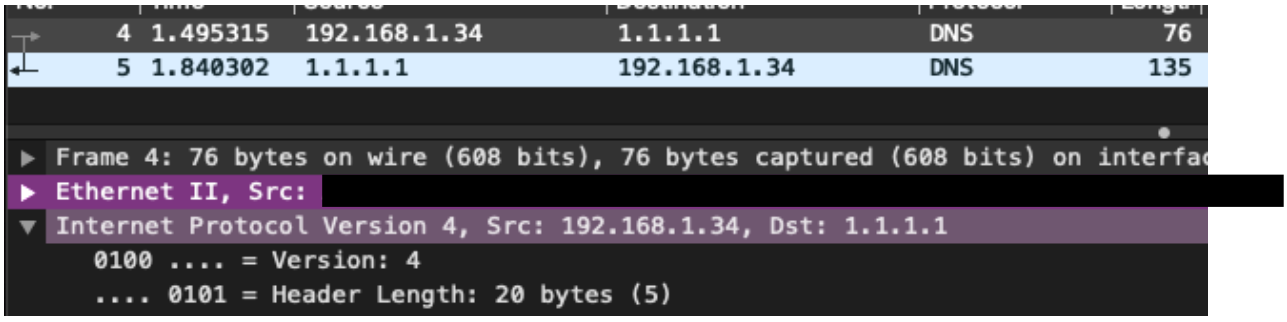

Figure 9: Header length of the request.

15. It has two checksums. One for the header and one for the Datagram itself.

16. There are total of 16 retransmissions happened from sender to the receiver. When we look at WireShark's interface, if there were no packet losses the communication between the sender and receiver should look like in Fig.10. Constant back and forth communication where sender send the packets and receiver acknowledges them as it receives them. But unfortunately, there is packet loss, so it looks more like Fig.11. If we see more than one transmission from the sender sequentially that means that retransmission happened.

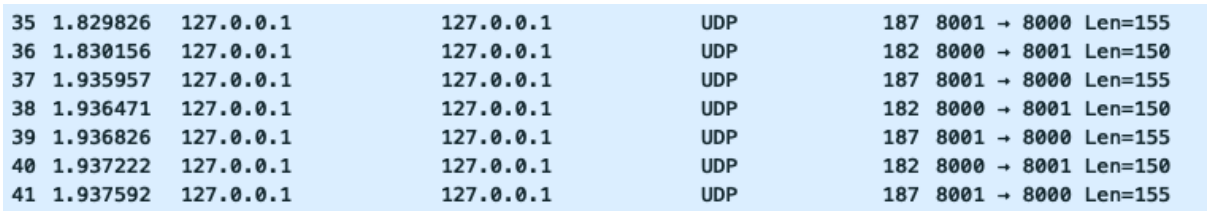

Figure 10: Transmission without packet loss

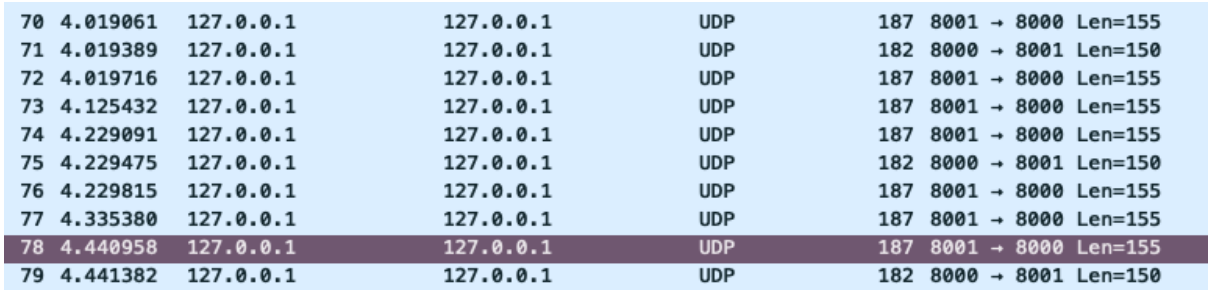

Figure 11: Transmission with packet loss

17. The timeout was set to 100ms. The transmission took 5 seconds to finish. There were 16 retransmission and a total of 50 transmissions from the sender. If retransmissions didn't happen the process would take 3.4 seconds to finish, roughly. It makes sense because if we were using TCP the numbers would drastically change but because of UDP the transmission times are negligible, so the only bottleneck is the timeout period of the sender. Shorter the timeout faster the transmission, but may use more bandwidth than expected.

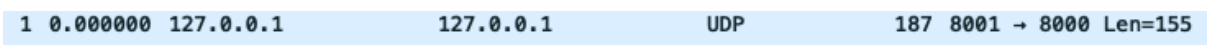

Figure 12: First packet.

Figure 13: Last packet.