

Datakom - Arbeidskrav 2

Del 1 - P3 - Sockets, TCP, HTTP og tråder

Program: `SocketServer.cs` og `SocketClient.cs`, skrevet i C# (C-Sharp).

Selv programmene kjøres lokalt på PC-en, og de dermed omgår det som har med router og lokalnett å gjøre, det blir opprettet en tilsvarende nettverkskommunikasjon lokalt på maskinen. Her brukes `localhost`, dvs loopback-adressen `127.0.0.1`, til PC-en for å "snakke med seg selv". Denne kommunikasjonen skjer via transportlaget og nettverkslaget, hvor transportlaget består av TCP (Transport Control Protocol) og nettverkslaget består av nettverksadressene.

TCP som protokoll fungerer ved at partene som skal kommunisere utfører et såkalt "3-way-handshake", hvor begge sider av kommunikasjonen bekrefter forbindelsen ved å utveksle flaggene `[SYN]`, `[SYN, ACK]` og `[ACK]` og setter sekvensnummer 1 og kvitteringsnummer 1. Ved å sette disse sekvens- og kvitteringsnumrene holder protokollen orden på at pakkene er sendt og mottatt, hvilket bekrefter integriteten av kommunikasjonen.

Enkel klient/tjener (TPC) - Virkemåte og kommunikasjon

Programmet fungerer ved at Server starter opp en lytte-tjeneste, som lytter etter tilkoblingsforsøk på en gitt protokoll, fra en gitt eller vilkårlig IP-adresse og på en spesifisert port. I dette tilfelle ble det brukt en `TcpListener` hvor det meste av administrasjonen rundt tilkoblingen blir abstrahert bort fra programmerer, og blir behandlet av kompilator og programmeringsspråket. IP-adressene som får lov til å koble seg til må spesifiseres, eventuelt settes til å være hvilken som helst (`IPAddress.Any`). Programmet går deretter i en løkke for å akseptere innkommende tilkoblinger.

Programmet er skrevet slik at for hver ny tilkobling blir det opprettet en ny tråd for å behandle denne tilkoblingen. Hva kommunikasjon angår, påvirkes ikke dette av det at man bruker flere tråder. Hver tråd kommuniserer likt. Den eneste forskjellen er at det skjer flere ting parallelt.

Etter en tilkobling, opprettes en `NetworkStream`, som er en kommunikasjonskanal mellom klient og tjener hvor data kan overføres ved å bruke `StreamReader` og `StreamWriter`.

For å forsikre korrekt utveksling av informasjon må det være et 1-til-1-forhold mellom det som sendes (Write) og mottas (Read). Altså, hver linje WriteLine som sendes, må det være nøyaktig én linje ReadLine på mottakersiden. Dette er nødvendig for å holde orden på kommunikasjonen og sørge for god integritet, altså at alt som sendes faktisk kommer frem. I dette tilfellet, hvor programmet er en slags nettverkskalkulator, skrives regnestykket inn på én linje før det sendes fra klient til tjener. Hos tjener blir det "pakket ut" og prosessert ("parse"), der hver enkelt del blir lagt inn i et array. Deretter evalueres oppføringene i henhold til en enkel logisk sjekk som bestemmer om det er operasjonen addisjon eller subtraksjon som skal utføres. Når evalueringen er ferdig, skrives resultatet inn på én linje som sendes tilbake til klient som har en lesefunksjon tilhørende akkurat denne "skriveren" på tjener-siden.

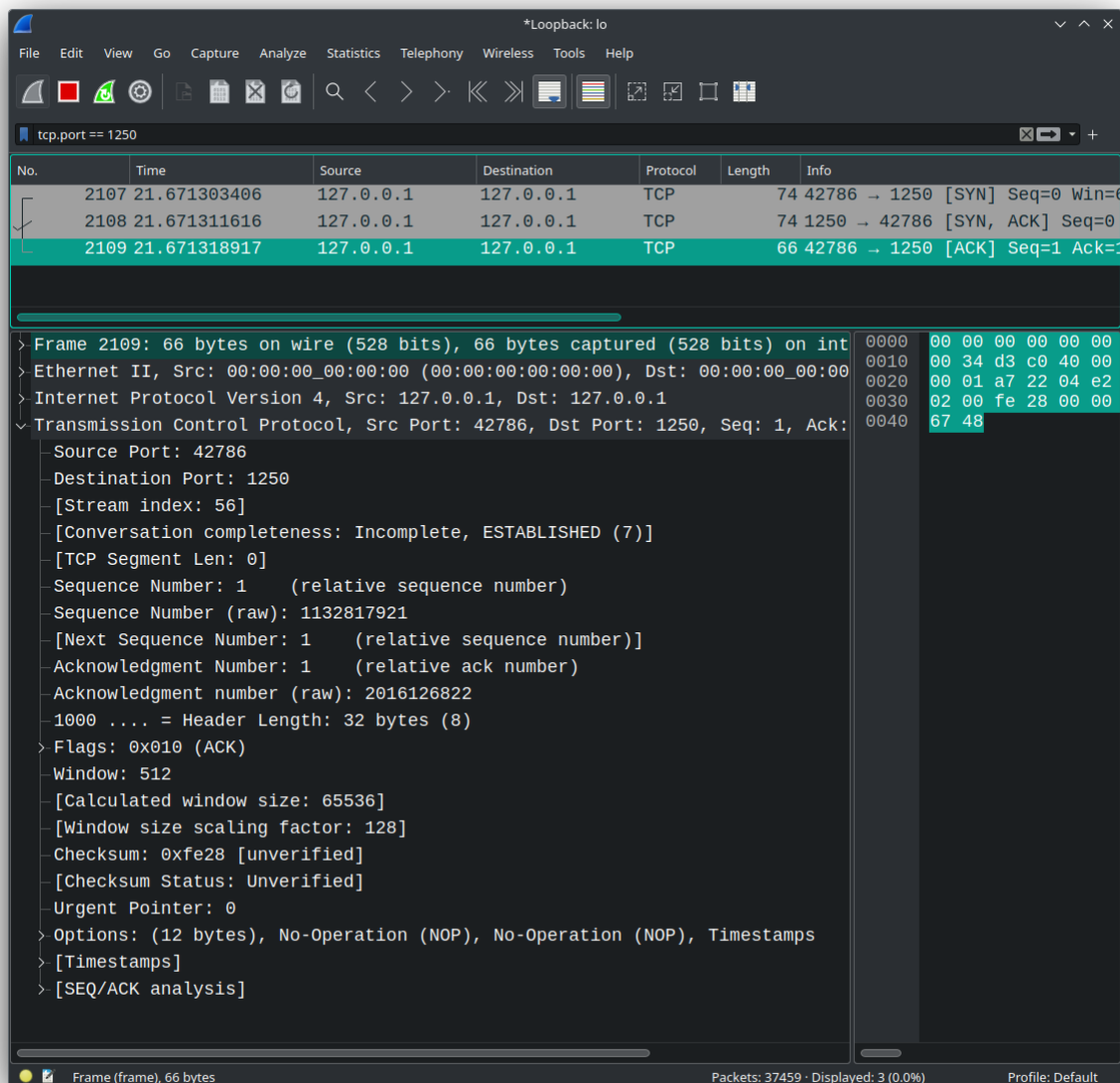
Når programmet avsluttes sørger `using` for at nettverksstrømmen blir terminert riktig og "ryddet opp".

Undersøkelse av kommunikasjon

For overvåkning via Wireshark ble filtreringen `tcp.port == 1250` brukt på Loopback device, altså 127.0.0.1, som er localhost. Ser man under kategorien Internet Protocol v4 for TTL, og Transmission Control Protocol for sekvens og kvittering, finner man følgende:

No	TTL	Sequence number	Next sequence no.	Ack. number	Flags
1	64	0	1	0	SYN
2	64	0	1	1	SYN, ACK
3	64	1	1	1	ACK

Skjerm bilde av svaret tilbake fra tjener til klient:



Neste steg er å sende en kalkulasjonsforespørsel fra klient til tjener. Denne sendes som en String på én linje, og settes sammen på følgende måte: `writer.WriteLine($"{operation},{num1},{num2}");`. For eksempel ble det for testformål sendt tekststrengen `add,1,2`, altså "adder tallene 1 og 2".

```

51
52 // 1:1 Correspondence:
53 // Operation is concatenated into a single line.
54 // Then sent to the server as such, in a single WriteLine
55 // On server-side, there's a single ReadLine that will receive the si
56 writer.WriteLine($"{operation},{num1},{num2}");
57 // Single-line Readline, reading the result/response sent from the
58 string response = reader.ReadLine();
59 Console.WriteLine(response);
60
61 Console.WriteLine("Do you want to perform another calculation? (1: Yes, 2: No)");
62 // Unless you select explicitly "1: Yes", then you will run a "2: No"

```

```

Run
ak_03_csharp_client
Enter the name of the machine where the server program runs: localhost
Connected to server.
Enter first number: 1
Enter second number: 2
Enter operation (add/subtract): add
Result: 3
Do you want to perform another calculation? (1: Yes, 2: No)

```

Her sendes altså 7 tegn. Under TCP > Payload fra klienten, finner vi at TCP Payload er på 8 bytes. Dette stemmer overens med hva som forventes: 7 bytes for 7 tegn, i tillegg til 1 bit for tegnet `newline`, `\n`, som legges til automatisk ved bruk av `WriteLine` og anses som ett tegn i denne sammenhengen.

Hos serveren blir denne strengen tolket, og det regnes ut et svar. Svaret sendes tilbake fra tjener på samme måte som klienten sendte det i første omgang: `writer.WriteLine("Result: " + result);`

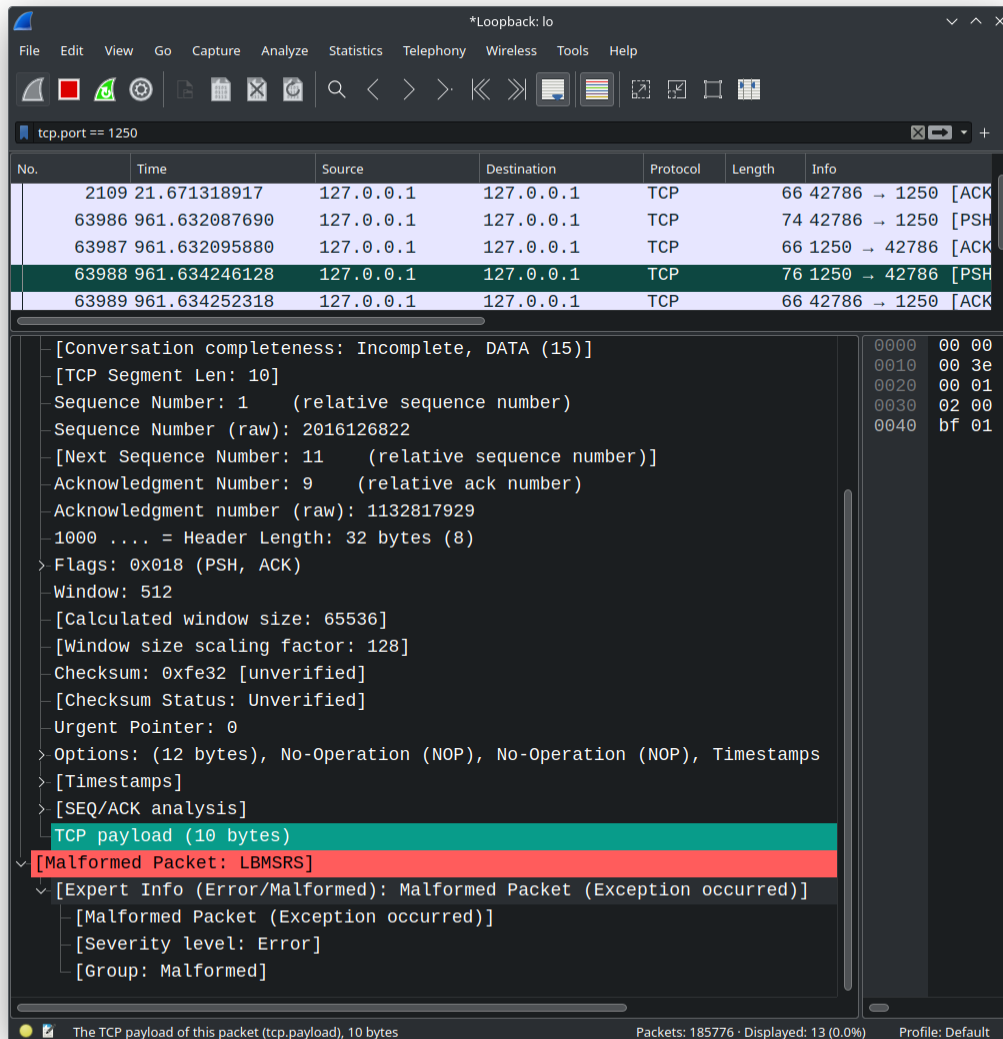
Svaret består først av 8 tegn ("`Result:` ") i tillegg til resultatet av utregningen (her: `3`) og til slutt en `newline`. Til sammen, i dette eksempelet, blir det 10 tegn, hvilket fremkommer av TCP Payload på linje 6 i pakkefangsten.

Tabell med sekvensnummer, kvitteringsnummer, flagg og TCP-nyttelast for operasjonen `add, 1, 2`:

No	Sequence number	Next sequence no.	Ack. number	Flags	TCP Payload
4	1	9	1	PSH, SYN	8
5	1	1	9	ACK	-
6	1	11	9	PSH, ACK	10
7	9	9	11	ACK	-

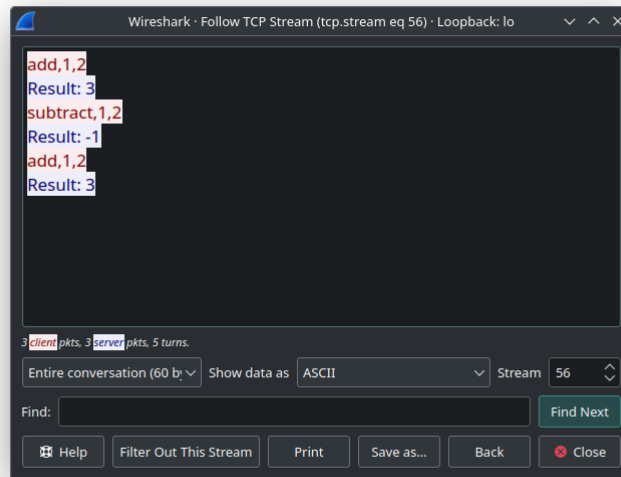
Tilsvarende oppførsel ble observert ved bruk av `subtract`-funksjonen, hvor TCP Payload reflekterte antallet tegn i operasjonen (8), pluss regnestykket (3) og `NewLine` (1), totalt 12 tegn og 12 bytes.

Skjerm bilde fra tjenerens svar tilbake til klienten:



Her dukket det også opp en spennende feilmelding: `Malformed Packet: LBMSRS` ("Lightweight Beacon-based Multicast (LBM) Streaming Reservation Protocol"). Denne ser ut til å indikere at Wireshark ikke greide å lese meldingen riktig, angivelig fordi den "antar at trafikken er LBMSRS, når den egentlig ikke er det". Men siden meldingen kom frem til klienten, er det trygt å anta at denne feilmeldingen enten kom ved en feiltolkning fra Wireshark sin side, eller at den ikke har så mye å si for akkurat dette brukstilfellet.

Skjerm bilde under (Follow > TCP Stream) viser meldingene som ble sendt frem og tilbake mellom klient og tjener:



Nedkobling blir håndtert i koden ved bruk av `using`, som i C# sørger for at variabler blir slettet etter bruk slik at de ikke tar opp ressurser unødvendig. I Wireshark kan man se nedkoblingen som en utveksling av `[FIN, ACK]`, `[FIN, ACK]` og `[ACK]`, som også kan refereres til som et "4-way-handshake".

No.	Time	Source	Destination	Protocol	Length	Info
1885	9.947216910	127.0.0.1	127.0.0.1	TCP	74	37676 → 1250 [SYN] Seq=0 Win=65495 Len=0
1886	9.947224670	127.0.0.1	127.0.0.1	TCP	74	1250 → 37676 [SYN, ACK] Seq=0 Ack=1 Win=65536
1887	9.947231190	127.0.0.1	127.0.0.1	TCP	66	37676 → 1250 [ACK] Seq=1 Ack=1 Win=65536
44271	575.153859393	127.0.0.1	127.0.0.1	TCP	74	37676 → 1250 [PSH, ACK] Seq=1 Ack=1 Win=65536
44272	575.153866433	127.0.0.1	127.0.0.1	TCP	66	1250 → 37676 [ACK] Seq=1 Ack=9 Win=65536
44273	575.155252521	127.0.0.1	127.0.0.1	TCP	76	1250 → 37676 [PSH, ACK] Seq=1 Ack=9 Win=65536
44274	575.155258431	127.0.0.1	127.0.0.1	TCP	66	37676 → 1250 [ACK] Seq=9 Ack=11 Win=65536
44527	579.233406662	127.0.0.1	127.0.0.1	TCP	66	37676 → 1250 [FIN, ACK] Seq=9 Ack=11 Win=65536
44528	579.233503124	127.0.0.1	127.0.0.1	TCP	66	1250 → 37676 [FIN, ACK] Seq=11 Ack=10 Win=65536
44529	579.233512525	127.0.0.1	127.0.0.1	TCP	66	37676 → 1250 [ACK] Seq=10 Ack=12 Win=65536

Flere samtidige klienter (TCP)

Når det kobles flere samtidige klienter opp mot tjeneren, vises dette på samme måte som ved én tilkobling, bare at det naturlig nok dukker opp flere av dem. Under er et skjermbilde fra Wireshark hvor flere klienter kommuniserer samtidig med tjeneren.

No.	Time	Source	Destination	Protocol	Length	Info
375	4.443025581	127.0.0.1	127.0.0.1	TCP	74	46028 → 1250 [SYN] Seq=0 Win=65495 Len=0
376	4.443034161	127.0.0.1	127.0.0.1	TCP	74	1250 → 46028 [SYN, ACK] Seq=0 Ack=1 Win=65536
377	4.443041521	127.0.0.1	127.0.0.1	TCP	66	46028 → 1250 [ACK] Seq=1 Ack=1 Win=65536
853	8.992955267	127.0.0.1	127.0.0.1	TCP	74	35262 → 1250 [SYN] Seq=0 Win=65495 Len=0
854	8.992963328	127.0.0.1	127.0.0.1	TCP	74	1250 → 35262 [SYN, ACK] Seq=0 Ack=1 Win=65536
855	8.992970698	127.0.0.1	127.0.0.1	TCP	66	35262 → 1250 [ACK] Seq=1 Ack=1 Win=65536

Her er det tilkoblet to klient på samme server. Det vises tydelig at det utføres to stykk "3-way-handshake", hvilket er som forventet.

Enkel webtjener

Bruker filteret `tcp.port == 8080` for pakkefangst i webtjeneren og kobler til webtjeneren ved å gå til <http://localhost:8080/> i nettleseren.

Webtjenere fungerer i grunn mye på samme måte som klient- og tjener-programmene, ved at det åpnes en lyttetjeneste over TCP, hvor det lyttes på en spesifisert port og man angir hvilke IP-adresser som er tillatt å koble til. Her settes porten til 8080, og IP-adressene er `.Any`, altså alle adresser er tillatt.

```
1 int port = 8080; // 8080 doesn't require elevated privileges to use.
2 TcpListener listener = new TcpListener(IPAddress.Any, port);
```

Lyttetjenesten (`listener`) kjøres i en løkke og venter på at noen kobler seg til den. Når noen gjør det, blir denne lagt inn som en klient.

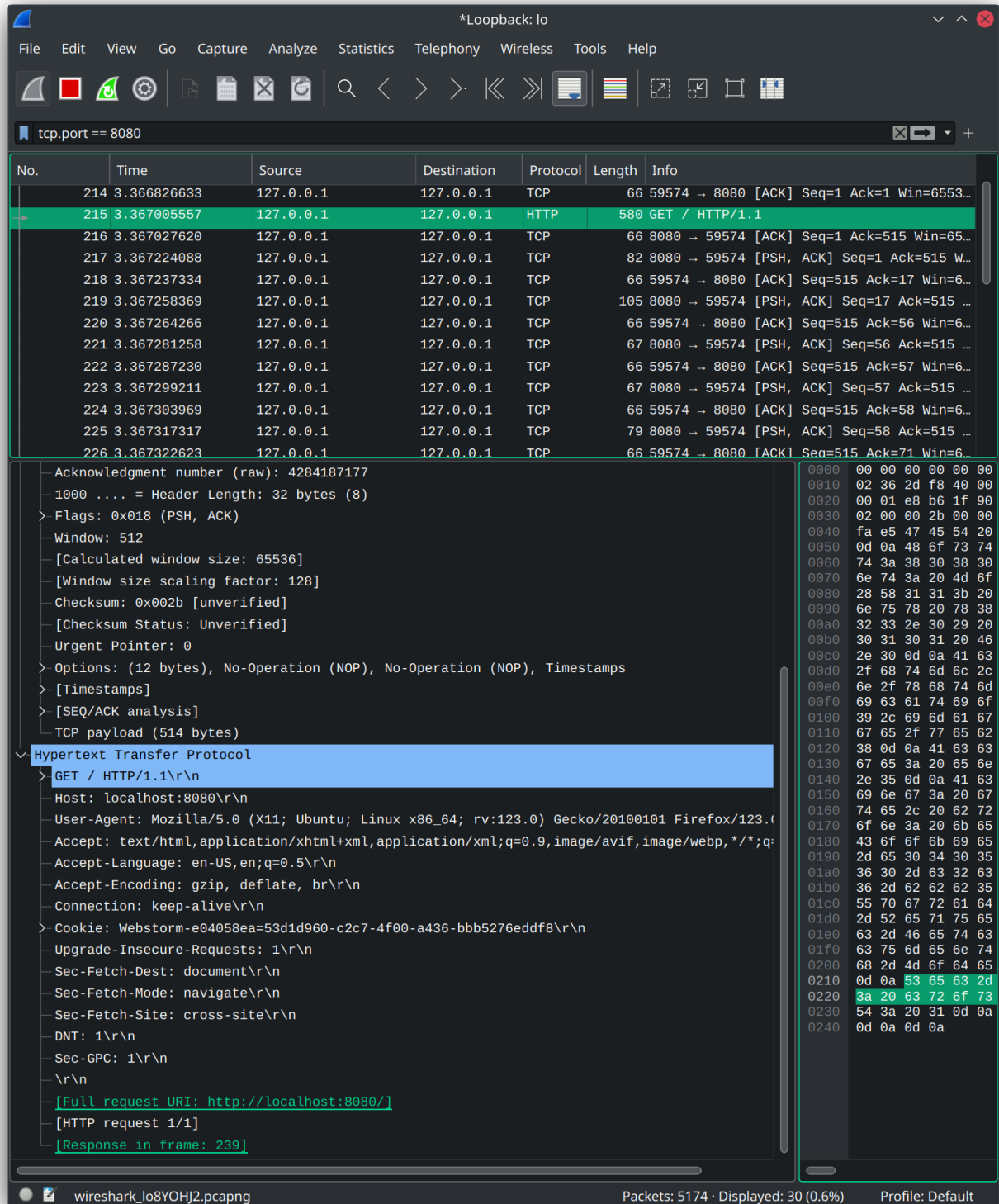
```
1 try {
2     listener.Start();
3     Console.WriteLine($"Server is listening on port {port}...");
4
5     // Keep the listener running in a loop, and accept any connection.
6     while (true) {
7         Console.WriteLine("Waiting for a connection...");
8         TcpClient client = listener.AcceptTcpClient(); // Add/create client
9         Console.WriteLine("Connected!");
10
11     // Rest of the code...
```

Oppkoblingen vises i Wireshark ved at utveksles `SYN`, `SYN`, `ACK` og `ACK` mellom tjener og nettleser.

```
1 212 3.366778635 127.0.0.1 127.0.0.1 TCP 74 59574 → 8080 [SYN] Seq=0 Win=65495
   Len=0 MSS=65495 SACK_PERM TSval=2621700837 TSecr=0 WS=128
2
3 213 3.366802123 127.0.0.1 127.0.0.1 TCP 74 8080 → 59574 [SYN, ACK] Seq=0 Ack=1
   Win=65483 Len=0 MSS=65495 SACK_PERM TSval=2621700837 TSecr=2621700837 WS=128
4
5 214 3.366826633 127.0.0.1 127.0.0.1 TCP 66 59574 → 8080 [ACK] Seq=1 Ack=1
   Win=65536 Len=0 TSval=2621700837 TSecr=2621700837
```

Deretter ber tjeneren om en HTTP-header fra klienten (nettleseren).

```
1 215 3.367005557 127.0.0.1 127.0.0.1 HTTP 580 GET / HTTP/1.1
```



Headeren lagres som et StringBuilder-objekt og sendes tilbake til klienten

```
1  StringBuilder requestHeaders = new StringBuilder();
2  string line;
3  while ((line = reader.ReadLine()) != null && line != "")
4  {
5      requestHeaders.AppendLine($"<LI>{line}</LI>");
6  }
```

Deretter følger en strøm med 23 ACK og PSH. PSH indikerer at datene skal "pushes" direkte til mottaker med en gang det blir tilgjengelig, i stedet for at det blir bufret og sendt som en større pakke. Dette gjør at kommunikasjonen oppleves raskere og mer responsiv for bruker. Til slutt sendes hele meldingen på HTTP, som spesifisert i tjenerkoden.

```
1 writer.WriteLine("HTTP/1.0 200 OK");
2 writer.WriteLine("Content-Type: text/html; charset=utf-8");
3 writer.WriteLine(); // Empty line to separate headers from body
4 writer.WriteLine(); // Empty line to separate headers from body
5 writer.WriteLine("<HTML><BODY>");
6 writer.WriteLine("<H1> Vær hilset! Du har koblet deg opp til min enkle web-
  tjener </H1>");
7 writer.WriteLine("Header fra klient er:");
8 writer.WriteLine("<UL>");
9 writer.Write(requestHeaders.ToString());
10 writer.WriteLine("</UL>");
11 writer.WriteLine("</BODY></HTML>");
```

Resultat i Wireshark:

*Loopback: lo

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

tcp.port == 8080

No.	Time	Source	Destination	Protocol	Length	Info
230	3.367416003	127.0.0.1	127.0.0.1	TCP	66	59574 → 8080 [ACK] Seq=515 Ack=164 Win=...
231	3.367425520	127.0.0.1	127.0.0.1	TCP	71	8080 → 59574 [PSH, ACK] Seq=164 Ack=515...
232	3.367428289	127.0.0.1	127.0.0.1	TCP	66	59574 → 8080 [ACK] Seq=515 Ack=169 Win=...
233	3.367451397	127.0.0.1	127.0.0.1	TCP	690	8080 → 59574 [PSH, ACK] Seq=169 Ack=515...
234	3.367455359	127.0.0.1	127.0.0.1	TCP	66	59574 → 8080 [ACK] Seq=515 Ack=793 Win=...
235	3.367466303	127.0.0.1	127.0.0.1	TCP	72	8080 → 59574 [PSH, ACK] Seq=793 Ack=515...
236	3.367469544	127.0.0.1	127.0.0.1	TCP	66	59574 → 8080 [ACK] Seq=515 Ack=799 Win=...
237	3.367480656	127.0.0.1	127.0.0.1	TCP	81	8080 → 59574 [PSH, ACK] Seq=799 Ack=515...
238	3.367483956	127.0.0.1	127.0.0.1	TCP	66	59574 → 8080 [ACK] Seq=515 Ack=814 Win=...
239	3.367499732	127.0.0.1	127.0.0.1	HTTP	66	HTTP/1.0 200 OK (text/html)
240	3.367632071	127.0.0.1	127.0.0.1	TCP	66	59574 → 8080 [FIN, ACK] Seq=515 Ack=815...
241	3.367645502	127.0.0.1	127.0.0.1	TCP	66	8080 → 59574 [ACK] Seq=815 Ack=516 Win=...

> [12 Reassembled TCP Segments (813 bytes): #217(16), #219(39), #221(1), #223(1), #225(13), #2

✓ Hypertext Transfer Protocol

> HTTP/1.0 200 OK\n

Content-Type: text/html; charset=utf-8\n

\n

[HTTP response 1/1]

[Time since request: 0.000494175 seconds]

[Request in frame: 215]

[Request URI: http://localhost:8080/]

File Data: 757 bytes

✓ Line-based text data: text/html (21 lines)

\n

<HTML><BODY>\n

<H1> Vær hilset! Du har koblet deg opp til min enkle web-tjener </H1>\n

Header fra klient er:\n

\n

GET / HTTP/1.1\n

Host: localhost:8080\n

User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:123.0) Gecko/20100101 Firefox/123.0\n

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8\n

Accept-Language: en-US,en;q=0.5\n

Accept-Encoding: gzip, deflate, br\n

Connection: keep-alive\n

Cookie: Webstorm-e04058ea=53d1d960-c2c7-4f00-a436-bbb5276eddf8\n

Upgrade-Insecure-Requests: 1\n

Sec-Fetch-Dest: document\n

Sec-Fetch-Mode: navigate\n

Sec-Fetch-Site: cross-site\n

DNT: 1\n

Sec-GPC: 1\n

\n

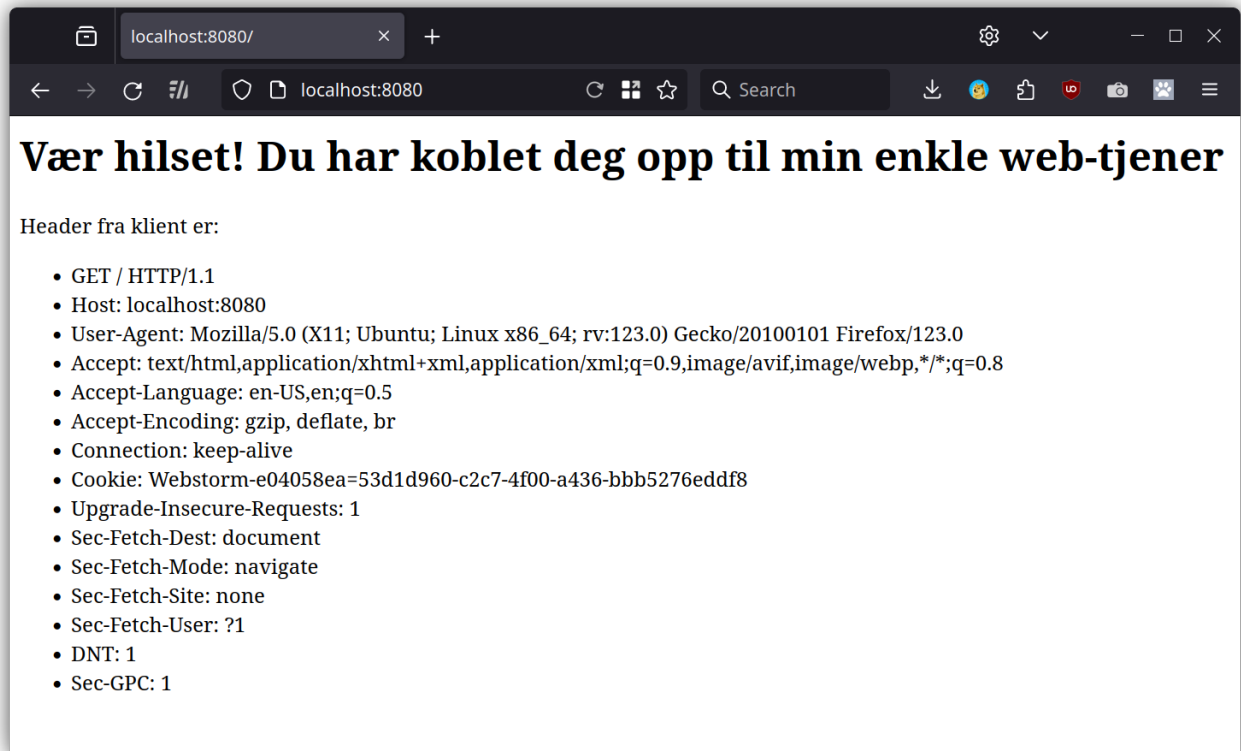
</BODY></HTML>\n

0000 00 00 00 00 00 00 00
0010 00 34 be e8 40 00 00
0020 00 01 1f 90 e8 b6
0030 02 00 fe 28 00 00 00
0040 fa e6

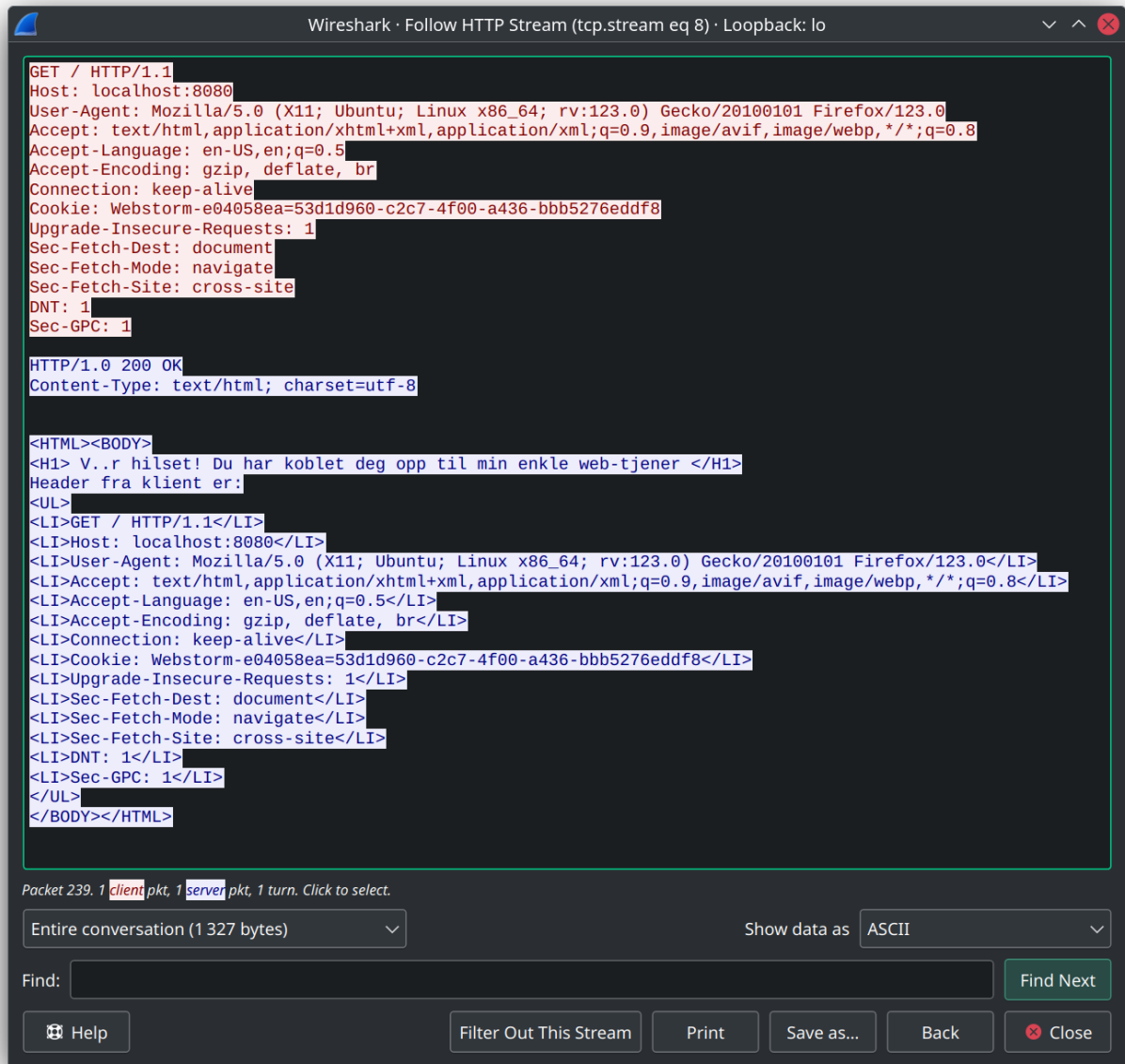
Frame (66 bytes) Reass< >

wireshark_lo8YOHJ2.pcapng Packets: 7210 · Displayed: 30 (0.4%) Profile: Default

Resultatet i nettleseren:



Dette kan også ses ved å følge enten TCP eller HTTP strømmen. Bildet under viser innholdet fra å velge Follow > HTTP Stream i Wireshark, og innholdet er identisk med det som vises ved å velge Follow > TCP Stream.



Nedkobling skjer ved at programkoden `client.Close()` kjøres, og over TCP kan det ses en siste utveksling `[FIN, ACK]` og `[ACK]`. Kommunikasjonen lukkes i tjenerkoden, og det skrives en melding som informerer om dette.

```

1 client.Close();
2 Console.WriteLine("Connection closed.");

```

Sekvens- og kvitteringsnummer, flagg og nyttelast:

No	Sequence number	Ack. number	Flags	TCP Payload
212	0	0	SYN	
213	0	1	SYN, ACK	
214	1	1	ACK	
215	1	1	GET / HTTP/1.1	514 bytes
216	1	515	ACK	

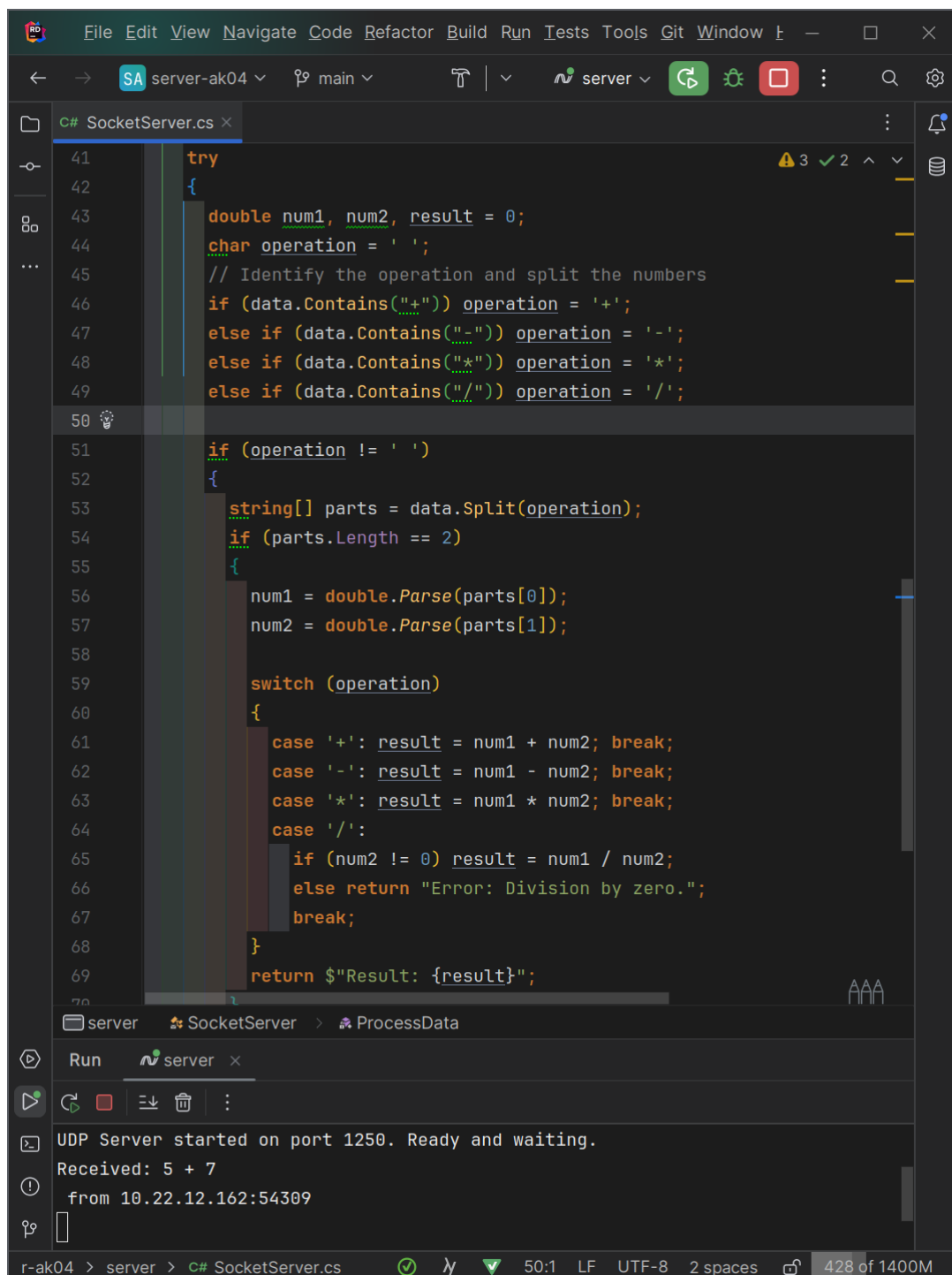
No	Sequence number	Ack. number	Flags	TCP Payload
217	1	515	PSH, ACK	16 bytes
218	515	17	ACK	
219	17	515	PSH	39 bytes
220	515	56	ACK	
221	56	515	PSH	1 bytes
222	515	57	ACK	
223	57	515	PSH	1 bytes
224	515	58	ACK	
225	58	515	PSH	13 bytes
226	515	71		
227	71	515	ACK	71 bytes
228	515	142	ACK	
229	142	515	ACK	22 bytes
230	515	164	ACK	
231	164	515	ACK	5 bytes
232	515	169	ACK	
233	169	515	ACK	624 bytes
234	515	793	ACK	
235	739	515	ACK	6 bytes
236	515	799	ACK	
237	799	515	ACK	15 bytes
238	515	814	ACK	
239	814	515	HTTP/1.0 200 OK	(text/html)
240	515	815	FIN, ACK	
241	815	516	ACK	

Av disse er de viktigste, og mest interessante, nummer 215 og 239, siden disse sender HTTP-innholdet. Detaljene i Wireshark for disse to er avbildet lenger opp, men her ser vi tydelig hvordan kommunikasjon via TCP er delt opp i mindre pakker.

Del 2 - P4: UDP, TLS

UDP kalkulator

Kalkulatoren over UDP ble testet både over localhost og via lokalnettet. Testen via lokalnettet ble utført i samarbeid med en annen gruppe, hvor de koblet seg til tjeneren min Pakkefangst fra kjøringvia `netcat`, sendte et regnestykke `5+7`. Tjeneren prosesserer stykket, hvor den separerer ut hvert enkelt tegn og setter det sammen til et regnestykke som kan regnes ut. Koden som tar seg av tolkning og utregningen kan ses i bildet under, i tillegg til et utsnitt av konsollen ved kommunikasjon med en klient.



```
File Edit View Navigate Code Refactor Build Run Tests Tools Git Window
SA server-ak04 main server
C# SocketServer.cs
41 try
42 {
43     double num1, num2, result = 0;
44     char operation = ' ';
45     // Identify the operation and split the numbers
46     if (data.Contains("+")) operation = '+';
47     else if (data.Contains("-")) operation = '-';
48     else if (data.Contains("*")) operation = '*';
49     else if (data.Contains("/")) operation = '/';
50
51     if (operation != ' ')
52     {
53         string[] parts = data.Split(operation);
54         if (parts.Length == 2)
55         {
56             num1 = double.Parse(parts[0]);
57             num2 = double.Parse(parts[1]);
58
59             switch (operation)
60             {
61                 case '+': result = num1 + num2; break;
62                 case '-': result = num1 - num2; break;
63                 case '*': result = num1 * num2; break;
64                 case '/':
65                     if (num2 != 0) result = num1 / num2;
66                     else return "Error: Division by zero.";
67                     break;
68             }
69             return $"Result: {result}";
70         }
71     }
72 }
server SocketServer > ProcessData
Run server
UDP Server started on port 1250. Ready and waiting.
Received: 5 + 7
from 10.22.12.162:54309
r-ak04 > server > C# SocketServer.cs 50:1 LF UTF-8 2 spaces 428 of 1400M
```

Etter at stykket har blitt tolket og regnet ut, sender tjeneren svaret tilbake til klienten som `Result: 12`. I bildet under vises klientens konsoll-vindu hvor det brukes netcat til på koble seg til tjenerens IP-adresse.

```
haakonff@LAPTOP-8S3ANK91:/mnt/c/Users/hfred$ netcat 10.22.212.60 1250 -u
5 + 7
Result: 12^C
haakonff@LAPTOP-8S3ANK91:/mnt/c/Users/hfred$
```

I pakkefangst i Wireshark kan kommunikasjonen ses. Denne er fanget opp på tjener-siden, med filteret `udp.port == 1250`. Den viser lengden på datapakkene. Her med `5 + 7` og NewLine, totalt 6 tegn. Svaret tilbake er `Result: 12`, totalt 10 tegn og inneholder altså ikke en NewLine, som kan ses på `^C` etter tallet `12`, som er keyboard interrupt fra bruker og befinner seg på samme linje som svaret.

No.	Time	Source	Destination	Protocol	Length	Info
147	25.826773499	10.22.12.162	10.22.212.60	UDP	48	54309 → 1250 Len=6
148	25.847827825	10.22.212.60	10.22.12.162	UDP	52	1250 → 54309 Len=10

Siden denne delen ble utført via nettverket er det mulig å finne litt mer spennende informasjon som MAC-adresser for klient og tjener:

- Klient
 - IP: 10.22.12.162
 - Port: 54309
 - MAC: 74:4c:a1:88:29:a7
- Tjener
 - IP: 10.22.212.60
 - Port: 1250
 - MAC: a0:99:9b:00:0d:ed

Fra pakkefangsten kan følgende informasjon om MAC-adresser ses:

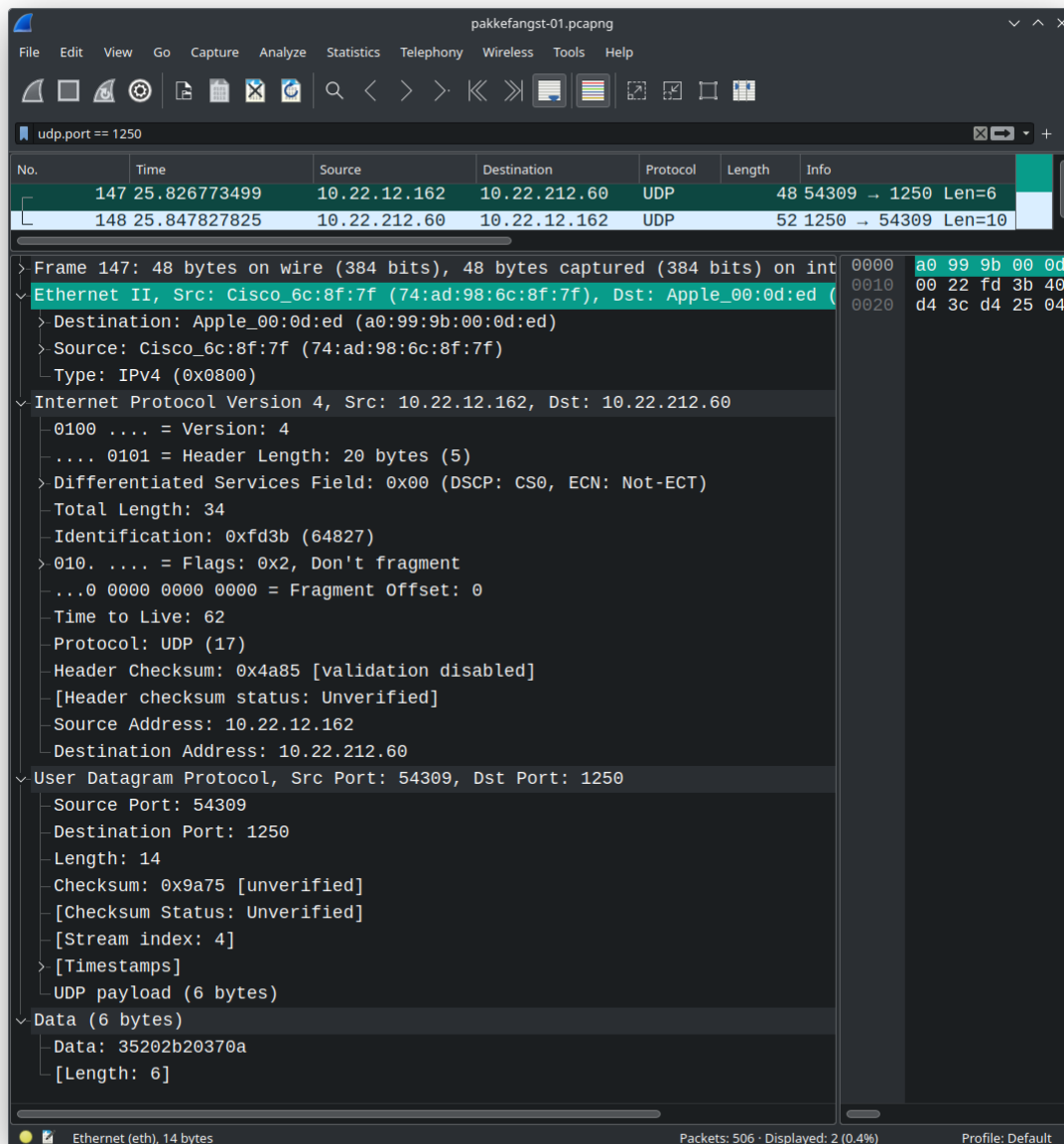
No.	Src-MAC	Dst-MAC	Src-port	Dst-port
147	74:ad:98:6c:8f:7f	a0:99:9b:00:0d:ed	54309	1250
148	a0:99:9b:00:0d:ed	74:ad:98:6c:8f:7f	1250	54309

MAC-adressene kan identifiseres ved oppslag, f.eks på nettstedet [MAClookup.app](#):

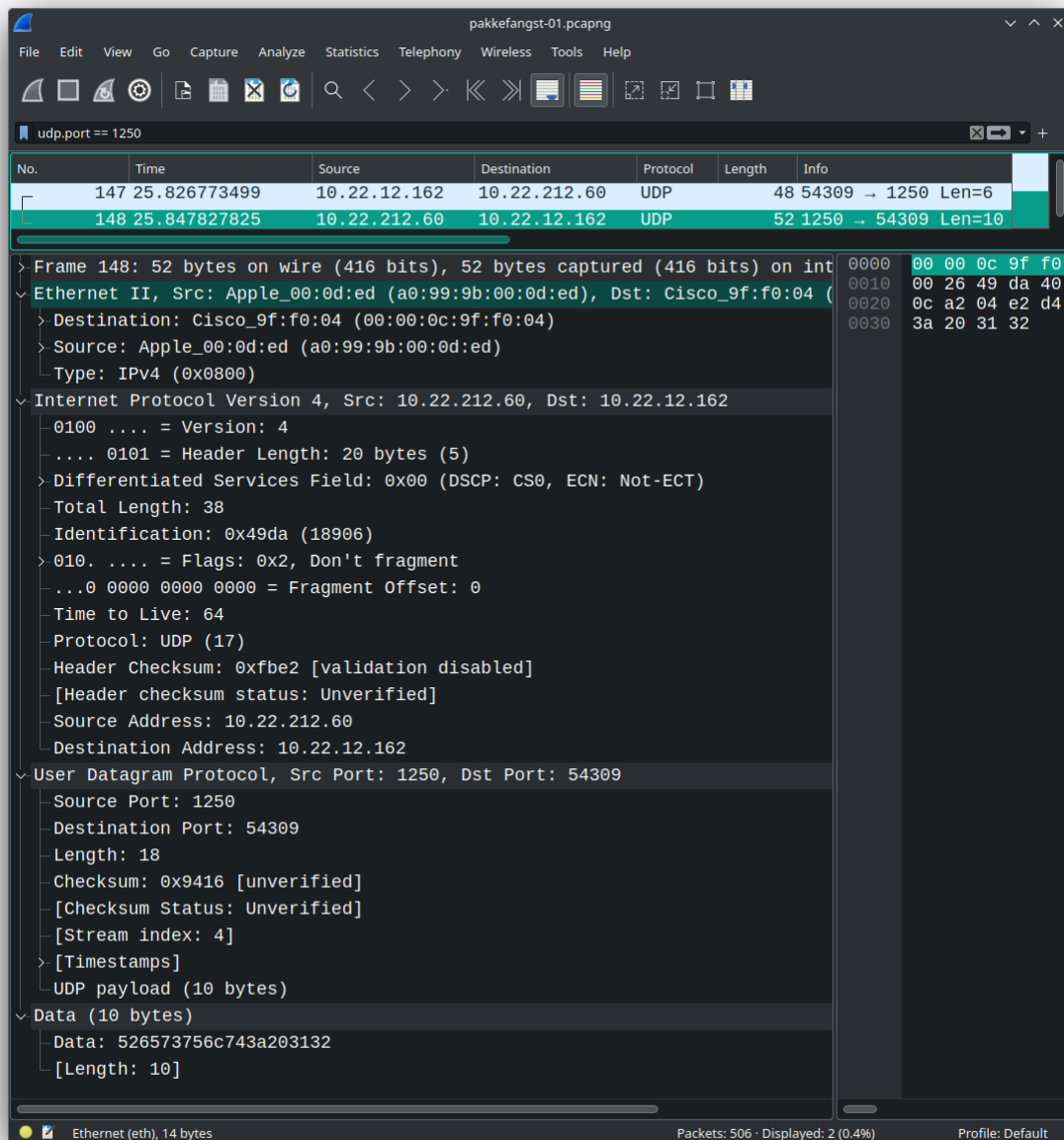
MAC	Vendor	OUI
74:ad:98:6c:8f:7f	Cisco Systems, Inc	74:AD:98
a0:99:9b:00:0d:ed	Apple, Inc.	A0:99:9B

Dermed er den ene parten i kommunikasjonen en Cisco-enhet. Siden tjener kjører på en Apple-maskin med Linux, er det trygt å anta at det er denne som er identifisert med Apple sin MAC-adresse. Videre kan man dermed anta at Cisco-enheten er en router eller wifi-utvider på NTNU campus. Samme informasjon kan vi enkelt se på **Ethernet II, Src:** fra hvor, til hvor, kommunikasjonen gikk siden den ene maskinen kommuniserer via et grensesnitt navngitt **Cisco**, mens den andre identifiseres som **Apple_00**.

Fra klient til tjener:



Svar fra tjener til klient:



Programkoden som tar seg av kommunikasjonen kan ses under. Linje 12 viser hvordan tjeneren mottar pakker fra en klient, som er det som sendes i linje 147 på pakkefangsten (bildene over her). Videre viser linje 16 at den mottatte dataen sendes til ProcessData via et funksjonsskall. Denne funksjonen er vist i bildet lengre opp, der et er et utklipp av IDE og konsoll. Linje 19 hvordan koden sender tilbake svar til klienten. Dette svaret fangs opp i linje 148 i Wireshark.

```

1 public static void Main()
2 {
3     const int PortNumber = 1250;
4     UdpClient udpServer = new UdpClient(PortNumber);
5     Console.WriteLine("UDP Server started on port " + PortNumber + ". Ready and
waiting.");
6
7     try
8     {
9         while (true)
10        {
11            IPEndPoint remoteEndPoint = new IPEndPoint(IPAddress.Any, 0);
12            byte[] receivedBytes = udpServer.Receive(ref remoteEndPoint);

```



```

13     string receivedData = Encoding.ASCII.GetString(receivedBytes);
14     Console.WriteLine($"Received: {receivedData} from {remoteEndPoint}");
15
16     string response = ProcessData(receivedData);
17
18     byte[] responseBytes = Encoding.ASCII.GetBytes(response);
19     udpServer.Send(responseBytes, responseBytes.Length, remoteEndPoint);
20 }
21 }
22 catch (Exception exception)
23 {
24     Console.WriteLine("An error occurred: " + exception.Message);
25 }
26 finally
27 {
28     udpServer.Close();
29 }
30 }

```

TLS

TLS (Transport Layer Security) er en protokoll laget for å sikre datakommunikasjon over nettverket, f.eks mellom en nettleser og en nettside (tjener). I praksis betyr dette at TLS krypterer datatrafikken som sendes mellom klient og tjener. Denne prosessen kalles ofte et "TLS handshake", hvor begge parter utveksler krypteringsnøkler og sertifikater for å bekrefte identitet og autentisitet, i tillegg til at det avtales en krypteringsmetode som skal benyttes for økten. Tjener sender et sertifikat til klienten for å bekrefte sin identitet, hvorpå klienten, f.eks en nettleser, verifiserer sertifikatets gyldighet ved å sjekke det opp mot en kjente autoriserte og pålitelige sertifikatinstanser. Når identiteten er bekreftet, blir det generert symmetriske sessjonsnøkler, som brukes til å kryptere datastrømmen

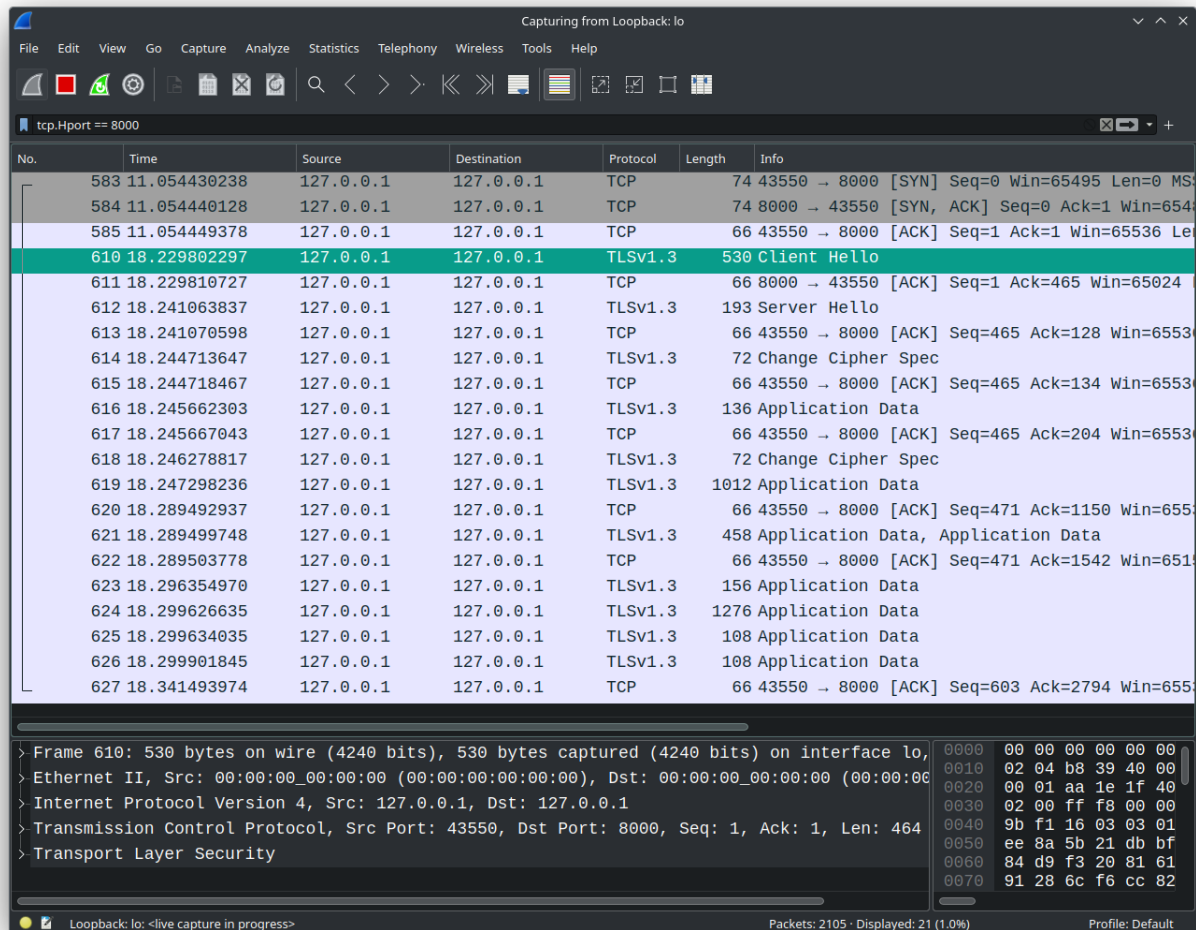
Kommandoer brukt for å generere keystore, compilere koden og starte hhv. tjener og klient:

```

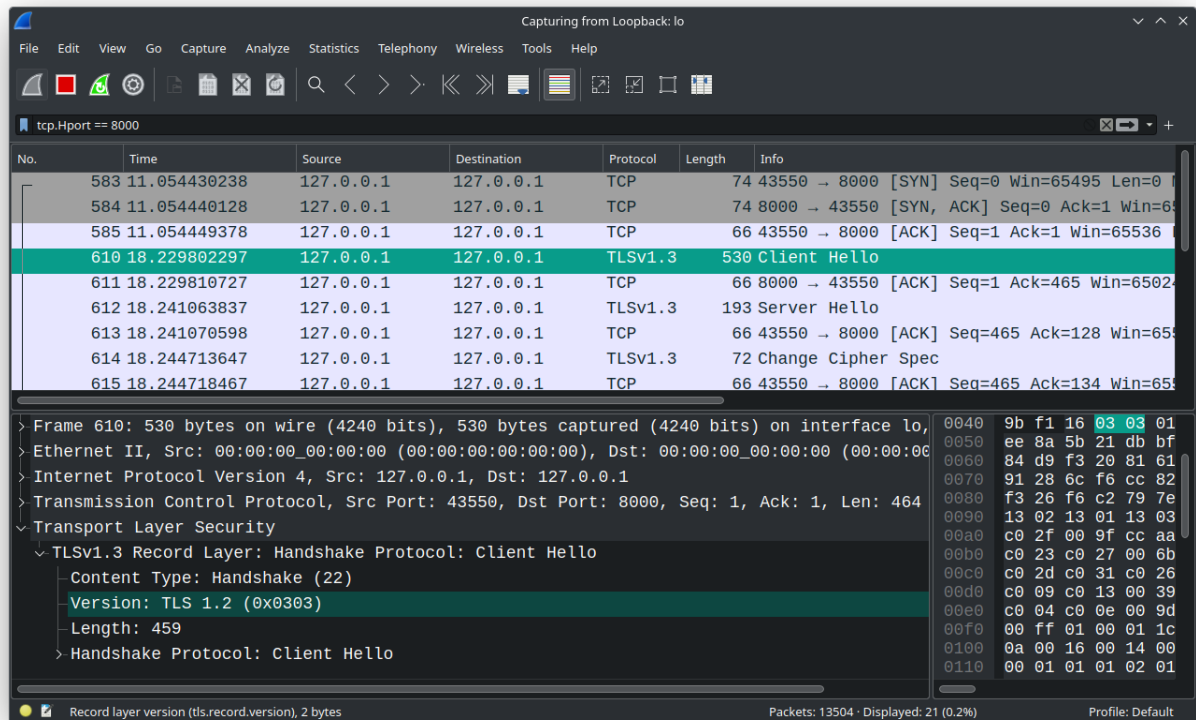
1 keytool -genkey -alias signFiles -keyalg RSA -keystore examplestore
2 javac JavaSSLServer.java
3 javac JavaSSLClient.java
4
5 java -Djavax.net.ssl.keyStore=examplestore -
   Djavax.net.ssl.keyStorePassword=password JavaSSLServer
6 java -Djavax.net.ssl.trustStore=examplestore -
   Djavax.net.ssl.trustStorePassword=password JavaSSLClient

```

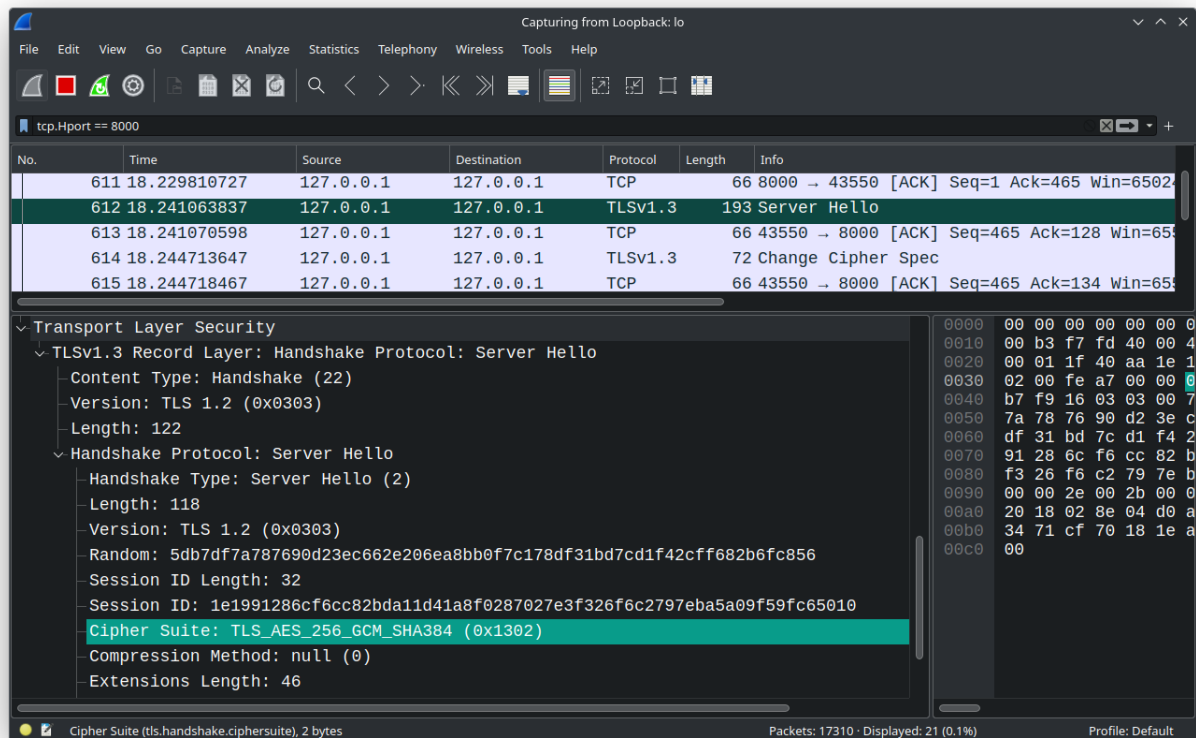
Ved å sende meldingen 123 fra klient til tjener fanger Wireshark opp følgende trafikk:



Fra kolonnen "Protocol" vises det tydelig at det er **TLSv1.3** som benyttes, altså versjon 1.3. Likevel, ved nøyere inspeksjon i detaljene, under Transport Layer Security > TLSv1.3 at det er **Version: TLS 1.2 (0x0303)** som benyttes. Dette kan ha å gjøre med at selv om både klient og tjener egentlig støtter v1.3, finner de ut at de ikke kunne kommunisere over v1.3 likevel og benytter v1.2 som en "fallback". Alternativt er det Wireshark som ikke helt greier å fange opp riktige detaljer fra økten.



Kryptografisuiten som ble valgt er **Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)**



Sertifikatet som benyttes er **OCSP Multi (2)**

```
Extension: status_request_v2 (len=9)
  Type: status_request_v2 (17)
  Length: 9
  Certificate Status List Length: 7
  Certificate Status Type: OCSP Multi (2)
  Certificate Status Length: 4
  Responder ID list Length: 0
  Request Extensions Length: 0
```

Sesjonsnøkler

Opprettelse av sessjonsnøkler begynner ved utveksling av `ClientHello` fra klient, som inneholder en liste med støttede kryptografisuiter og en streng med tilfeldige bytes. Denne tilfeldig genererte strengen, også kjent som "`client random`", brukes både som en unik ID for økten, og som en del av nøkkelen som genereres senere. Tilsvarende med `ServerHello`, fra tjener, som svarer med hvilken kryptografisuite som støttes av begge og skal benyttes, i tillegg til en "`server random`", som senere kombineres med "`client random`" til å fungere som både økt-ID og entropi i sessjonsnøkkelen.

Det utføres en utveksling av såkalt "pre-master secret", som gjøres litt ulik avhengig av krypteringsmetode, f.eks RSA eller Diffie-Hellman. Her brukes gjerne tjenerens offentlige nøkkel, som klient fikk sammen med sertifikatet, for å sikre at utvekslingen foregår sikkert.

Når denne "pre-master secret" er etablert, blir den kombinert med "`client random`" og "`server random`" gjennom en såkalt "pseudo-random function" (PRF), hvilket resulterer i en "Master Secret".

"Master Secret" blir så brukt til å generere sessjonsnøklerne

Under er en formel for hvordan denne kombinasjonen foregår. "`master secret`" er her en konstant, definert i TLS-protokollen

```
1 | MasterSecret = PRF(PreMasterSecret, "master secret", ClientRandom +
    ServerRandom)
```

Med Master Secret på plass, brukes denne som grunnlag for å konstruere sessjonsnøklerne. Den deles opp og det lages flere "delnøkler" av den, som hver har sitt område, blant annet kryptering for sending og mottak av data og for å sikre dataintegritet.