NTNU

Kunnskap for en bedre verden

# Datakommunikasjon mellom nettverksapplikasjoner

Rapport for dokumentasjon av datakommunikasjon mellom nettverksapplikasjoner i øvinger tre og fire i nettverksprogrammering


## Faglærere

Olav Skundberg

Ole Christian Eidheim

## Student

Joachim Grimen Westgaard

# Innholdsfortegnelse

1. Øving 3 – tjener og klient over TCP ..... 3
1.1 Innledning ..... 3
1.2 Tjener og klient ..... 3
1.3 Webtjener ..... 10

2. Øving 4 – tjener og klient over UDP og TLS ..... 15
2.1 Innledning ..... 15
2.2 UDP ..... 15
2.3 TLS ..... 18

3. Referanser ..... 24

# 1. Øving 3 – tjener og klient over TCP

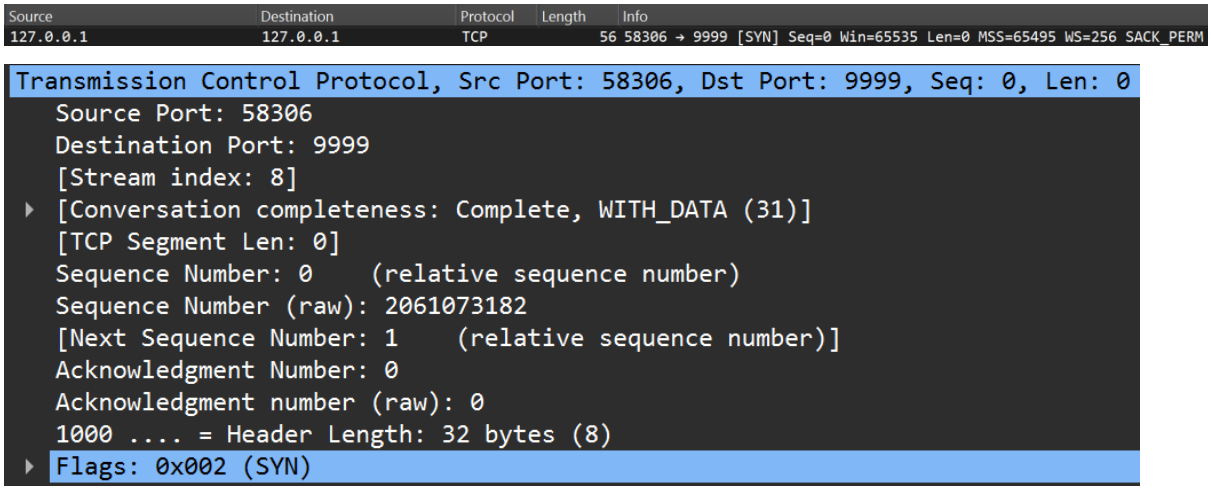
## 1.1 Innledning

Denne øvingen er delt i to oppgaver: programmering (i Java) av både tjener og klient, og en webserver hvor klienten er en selvvalgt nettleser. Tjener og klient vil dokumenteres først, og bli etterfulgt av webtjeneren.

## 1.2 Tjener og klient

I denne delen av øvingen er det programmert en tjener, servertråd og en klient. Tjeneren og klienten benytter seg av Java-klassene ServerSocket og Socket for etablering av og gjennomføring av datakommunikasjon. Tjeneren startes opp og får tildelt en valgt port på maskinens lokale tjener gjennom «localhost», og dermed vil klienten kunne etablere en forbindelse med tjeneren gjennom å koble seg opp mot den samme, spesifiserte porten. I dette tilfellet er den valgte porten 9999.
På transport- og nettverkslaget benyttes protokoller i TCP/IP-familien. Dette fordi Java-klassene ServerSocket og Socket i utgangspunktet benytter seg av TCP/IP (Oracle, 2024).

Pakkefangst i Wireshark bekrefter dette:



### 1.2.1 Oppkobling

Her kan vi se at klienten sender en forespørsel fra port 58306 til port 9999, hvor tjeneren lytter etter innkommende forbindelsesforespørsler. Dette er den første meldingen i kommunikasjonen, sendt av klienten, for å opprette en forbindelse med tjeneren. Dette kan leses av at TCP-pakken har aktivert SYN-flagget.

Kildekoden konfigurerer tjeneren til å benytte seg av port 9999 i følgende kodesnutt:

```
final int PORT_NUMBER = 9999;
int threadCounter = 0;

// Initialize server-socket on declared port
try (ServerSocket ss = new ServerSocket(PORT_NUMBER))
```

I programmet leses portnummeret til klienten inn og settes i dets socket i følgende kodesnutt:

```
System.out.println("Welcome!");
System.out.print("Enter the port you want to connect to: ");
int portNumber = Integer.parseInt(in.nextLine());

// Initializes the socket and establishes communication locally on 'localhost'
try (Socket clientSocket = new Socket( host: "localhost", portNumber);
```

Etter at tjeneren har mottatt TCP-pakken med SYN-flagget fra klienten, vil tjeneren sende et svar tilbake til klienten gjennom dets port (58306). I denne meldingen sendt av tjeneren, vil TCP-pakken ha et aktivt SYN- og ACK-flagg. ACK-flagget er en bekreftelse på at den har mottatt klientens SYN-flagg, at den anerkjenner klientens ønske om en forbindelse.

```
TCP          56 9999 → 58306 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
```

```
Transmission Control Protocol, Src Port: 9999, Dst Port: 58306, Seq: 0, Ack: 1, Len: 0
   Source Port: 9999
   Destination Port: 58306
   [Stream index: 8]
▸  [Conversation completeness: Complete, WITH_DATA (31)]
   [TCP Segment Len: 0]
   Sequence Number: 0    (relative sequence number)
   Sequence Number (raw): 3036989846
   [Next Sequence Number: 1    (relative sequence number)]
   Acknowledgment Number: 1    (relative ack number)
   Acknowledgment number (raw): 2061073183
   1000 .... = Header Length: 32 bytes (8)
▸  Flags: 0x012 (SYN, ACK)
```

Klienten svarer deretter på tjenerens forespørsel om å opprette en forbindelse (på veien fra tjeneren tilbake til klienten) ved en pakke med TCP sitt ACK-flagg aktivt.
Denne typen oppkobling kalles en «3-way handshake», og sikrer at begge parter selv er klare, og vet at den andre parten også er det.

Det bør bemerkes at både kilde- og destinasjonsadresser er 127.0.0.1, da dette er maskinens IP-adresse for IPv4 loopback, for å kunne sende og motta «internt» på maskinen mellom dets egne porter.
Tjeneren og klientens datakommunikasjon vil på transportlaget benytte seg av TCP-protokollen, og IPv4 på nettverkslaget.

## 1.2.2 Overføring av data og nedkobling

Over er oppkoblingen av forbindelsen mellom tjeneren og klienten dokumentert.
Overføringen av data og nedkoblingen følger.

Etter at klienten har bekreftet tjenerens ønske om en forbindelse til seg, vil den sende sin første melding til tjeneren. Samtidig vil tjeneren også sende en melding til klienten.
Meldingen fra tjeneren sendes her (i klassen SocketServerThread):

```
try (BufferedReader reader = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
     PrintWriter writer = new PrintWriter(clientSocket.getOutputStream(), autoFlush: true);
) {
    // Send welcome message to client
    writer.println("### Welcome to the server on localhost:" + clientSocket.getLocalPort() + " running on thread " +
            threadNumber + "! ###");
```

Meldingen fra klienten sendes her (i klassen SocketClient):

```
writer.println("Hello from the client:)");
```

De vil hver for seg lese meldingene de har mottatt i følgende kode:

Tjener:

```
// Read acknowledgment from client
System.out.println(reader.readLine());
```

Klient:

```
// Reads the welcome-message from the server
System.out.println(reader.readLine());
```

Etter at de har ønsket hverandre velkommen til samtalen, vil klienten lese inn to tall og en operator (addisjon eller subtraksjon) fra brukeren av klientprogrammet, og deretter sende denne informasjonen til tjeneren. De sendes (i dette programmet) i tre ulike pakker. Dette kunne vært gjort i én enkelt pakke, men da hadde tjeneren vært nødt til å ha logikk for å tolke pakken i de tre ulike verdiene (de to tallene og operatoren). Derfor ble det for enkelthets skyld gjort i tre ulike pakker.

```java
Double firstNumber = null;
Double secondNumber = null;
int operator = 0;

while (firstNumber == null) {
    System.out.print("Enter the first number: ");
    firstNumber = Double.parseDouble(in.nextLine());
}

while (secondNumber == null) {
    System.out.print("Enter the second number: ");
    secondNumber = Double.parseDouble(in.nextLine());
}

while (operator != 1 && operator != 2) {
    System.out.print("Do you want to add (1) or subtract (2)? ");
    operator = Integer.parseInt(in.nextLine());
}

// Send the data to the server and read its response
System.out.print("Sending... ");
writer.println(firstNumber);
writer.println(secondNumber);
writer.println(operator);
```

Klienten forsøker å lese inn brukerens to tall og operator, før de sendes i hver sine pakker i de tre siste, avbildede, linjene.

Overføringen av disse tre pakkene kan vi finne igjen i Wireshark:

| | | | |
|---|---|---|---|
| 127.0.0.1 | 127.0.0.1 | TCP | 49 58306 → 9999 [PSH, ACK] Seq=26 Ack=71 Win=2161152 Len=5 |
| 127.0.0.1 | 127.0.0.1 | TCP | 44 9999 → 58306 [ACK] Seq=71 Ack=31 Win=2161152 Len=0 |
| 127.0.0.1 | 127.0.0.1 | TCP | 49 58306 → 9999 [PSH, ACK] Seq=31 Ack=71 Win=2161152 Len=5 |
| 127.0.0.1 | 127.0.0.1 | TCP | 44 9999 → 58306 [ACK] Seq=71 Ack=36 Win=2161152 Len=0 |
| 127.0.0.1 | 127.0.0.1 | TCP | 47 58306 → 9999 [PSH, ACK] Seq=36 Ack=71 Win=2161152 Len=3 |
| 127.0.0.1 | 127.0.0.1 | TCP | 44 9999 → 58306 [ACK] Seq=71 Ack=39 Win=2161152 Len=0 |

Her kan vi se at tjeneren anerkjenner og kvitterer klientens sendte pakker ved hjelp av kvitteringsnummeret (Ack). Dette nummeret vil være tjenerens måte å vise klienten hvilken byte den forventer som den neste. Sekvensnummeret til linje 1 (klienten), indikerer at dets neste byte for overføring er byte nummer 26. Størrelsen på linje 1 er 5 byte (Len=5). Dette vil si at klienten overfører bytene nummer 26, 27, 28, 29 og 30. Derfor vil tjeneren i sin

anerkjennelse på linje 2 indikere at den forventer byte nummer 31 som den neste i klienten sin neste overføring. Det må altså være en overenstemmelse mellom klientens sekvensnummer og tjenerens kvitteringsnummer. Dette forsikrer partene om at alle de ønskede bytene faktisk har blitt overført.

Vi kan få en oversikt over kommunikasjonen så langt (fra oppkobling til og med overføring av velkomstmeldinger, tall og operator) ved hjelp av tabellen under.

| Nr. | Innhold | KLIENT | | | TJENER | | |
|---|---|---|---|---|---|---|---|
| | | Sekv.nr | Kvitt.nr | TCP Length/ Payload | Sekv.nr | Kvitt.nr | TCP Length/ Payload |
| 446 | SYN | 0 | - | 0 bytes | - | - | - |
| 447 | SYN, ACK | - | - | - | 0 | 1 | 0 bytes |
| 448 | ACK | 1 | 1 | 0 bytes | - | - | - |
| 449 | PSH | 1 | 1 | 25 bytes | - | - | - |
| 450 | ACK | - | - | - | 1 | 26 | 0 bytes |
| 451 | PSH | - | - | - | 1 | 26 | 70 bytes |
| 452 | ACK | 26 | 71 | 0 bytes | - | - | - |
| 489 | PSH | 26 | 71 | 5 bytes | - | - | - |
| 490 | ACK | - | - | - | 71 | 31 | 0 bytes |
| 491 | PSH | 31 | 71 | 5 bytes | - | - | - |
| 492 | ACK | - | - | - | 71 | 36 | 0 bytes |
| 493 | PSH | 36 | 71 | 3 bytes | - | - | - |
| 494 | ACK | - | - | - | 71 | 39 | 0 bytes |
| 495 | PSH | - | - | - | 71 | 39 | 5 bytes |
| 496 | ACK | 39 | 76 | 0 bytes | - | - | - |

Følgende rader ble uthentet fra Wireshark ved å sette display-filteret til «tcp.port == 9999».

Vi kan finne verdiene til radene i tabellen over under TCP-headeren i Wireshark:

```
▾ Transmission Control Protocol, Src Port: 58306, Dst Port: 9999, Seq: 39, Ack: 76, Len: 0
    Source Port: 58306
    Destination Port: 9999
    [Stream index: 8]
  ▸ [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 39    (relative sequence number)
    Sequence Number (raw): 2061073221
    [Next Sequence Number: 39    (relative sequence number)]
    Acknowledgment Number: 76    (relative ack number)
```

### 1.2.3 Nedkobling

Avslutningsvis i kommunikasjonen mellom klienten og tjeneren, velger brukeren av klienten å avbryte forbindelsen fra klienten sin side. Dermed vil klienten sende en TCP-pakke med FIN- og ACK-flagget aktivt. Dette signaliserer at klienten ønsker å lukke den vedvarende forbindelsen fra klienten til tjeneren. Tjeneren bekrefter mottagelsen av denne meldingen ved en ACK-melding, før tjeneren selv sender en FIN- og ACK-melding til klienten for å lukke forbindelsen ifra tjeneren til klienten. Dette vil klienten svare med et ACK-flagg. Etter dette, er forbindelsen lukket.

Vi kan visualisere denne nedkoblingen ved hjelp av tabellen under:

| Nr. | Innhold | KLIENT | | | TJENER | | |
|---|---|---|---|---|---|---|---|
| | | Sekv.nr | Kvitt.nr | TCP Length/ Payload | Sekv.nr | Kvitt.nr | TCP Length/ Payload |
| 493 | FIN, ACK | 39 | 76 | 0 bytes | - | - | - |
| 494 | ACK | - | - | - | 76 | 40 | 0 bytes |
| 495 | FIN, ACK | - | - | - | 76 | 40 | 0 bytes |
| 496 | ACK | 40 | 77 | 0 bytes | - | - | - |

Disse verdiene kan finnes i følgende rader i Wireshark:

```
127.0.0.1   127.0.0.1   TCP   44 58306 → 9999 [FIN, ACK] Seq=39 Ack=76 Win=2161152 Len=0
127.0.0.1   127.0.0.1   TCP   44 9999 → 58306 [ACK] Seq=76 Ack=40 Win=2161152 Len=0
127.0.0.1   127.0.0.1   TCP   44 9999 → 58306 [FIN, ACK] Seq=76 Ack=40 Win=2161152 Len=0
127.0.0.1   127.0.0.1   TCP   44 58306 → 9999 [ACK] Seq=40 Ack=77 Win=2161152 Len=0
```

### 1.2.4 Porter, IP- og MAC-adresser

Den enkle klient-tjener kommunikasjonen over foregår lokalt på maskinen, derfor benyttes maskinens IP-adresse som refererer til loopback, at trafikken ut ønskes direkte inn igjen på maskinen. Denne finner vi igjen i IP-adressen 127.0.0.1. Maskinen har tjeneren kjørende på port 9999 (spesifisert i den egendefinerte Java-klassen SocketServer), mens SocketClient-programmet kjører på en port valgt av operativsystemet, i dette tilfellet 58306.

Det vil ikke være noen MAC-adresse forbundet med denne kommunikasjonen, da det fysiske nettverkskortet ikke tas i bruk. Loopback-trafikken på «localhost 127.0.0.1» er kun et virtuelt grensesnitt.

## 1.3 Webtjener

I denne delen av øvingen er det laget to klasser, én webtjener og én webtjener-tråd. Webtjeneren fungerer som en «resepsjonist», som har i oppgave å ta imot klienter og gi de videre til webtjener-tråden for kommunikasjonshåndtering.

Webtjeneren er satt til å lytte på port 80, som er standardporten for kommunikasjon over HTTP-protokollen. Dermed kan en nettleser fungere som en klient ved å kun søke opp «localhost» uten å spesifisere portnummer.

På transport- og nettverkslaget benyttes fremdeles protokoller ifra TCP/IP-familien, på lik linje som den enkle tjener-klient-kommunikasjonen. Forskjellen i webtjeneren, er at den benytter seg av HTTP-protokollen på applikasjonslaget. Dette kan vi bekrefte i Wireshark ved å sette display-filteret til «http»:

| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| ::1 | ::1 | HTTP | 756 | GET / HTTP/1.1 |
| ::1 | ::1 | HTTP | 64 | HTTP/1.0 200 OK (text/html) |
| ::1 | ::1 | HTTP | 677 | GET /favicon.ico HTTP/1.1 |
| ::1 | ::1 | HTTP | 64 | HTTP/1.0 200 OK (text/html) |

Her kan vi se at det foregår intern kommunikasjon på maskinen. «::1» er IPv6-standarden for loopback-trafikk, på lik linje med 127.0.0.1 i IPv4. Dette gjør at vi heller ikke her vil benytte et fysisk nettverkskort i kommunikasjonen mellom tjener og klient.

Vi kan i følgende kodesnutt se at webtjeneren fungerer relativt likt som tjeneren ifra det enkle tjener-klient eksemplet.

```
final int PORT_NUMBER = 80;
int threadCounter = 0;

try (ServerSocket ss = new ServerSocket(PORT_NUMBER)) {
    System.out.println("Waiting for connections...");

    while (true) {
        try {
            threadCounter++;
            WebServerThread wst = new WebServerThread(threadCounter, ss.accept());
            wst.start();
```

Webtjeneren vil lytte etter innkommende forespørsler på port 80, og videreføre klienten til et objekt av WebServerThread-klassen ved opprettelsen av en forbindelse. Vi kan i følgende kodesnutt se hvordan tråd-klassen tar imot klientens Socket-objekt:

```java
public WebServerThread(int threadNumber, Socket webSocket) {
    this.threadNumber = threadNumber;
    this.webSocket = webSocket;
}

@Override
public void run() {

    System.out.println("Web-client connected on thread no. " + threadNumber);

    try (BufferedReader reader = new BufferedReader(new InputStreamReader(webSocket.getInputStream()));
         PrintWriter writer = new PrintWriter(webSocket.getOutputStream(), autoFlush: true);
    ) {

        System.out.println("Link established with client residing on port " + webSocket.getPort() + "\n");
```

Her vil det også bli sjekket hvilken port klienten bruker. Dette kan være nyttig ved kryssreferering i Wireshark.

I det testede eksemplet på maskinen, kjører Google Chrome som klient på port 60376. Kodesnutten over gir følgende resultat i terminalen:

```
Web-client connected on thread no. 1
-> Connection established in thread no. 1
Link established with client residing on port 60376
```

## 1.3.1 Oppkobling

I Wireshark ønsker vi å finne HTTP-pakker sendt mellom port 80 (webtjeneren) og port 60376 (klienten). Følgende skjermbilde fra Wireshark viser HTTP-kommunikasjonen mellom port 80 og port 60376:

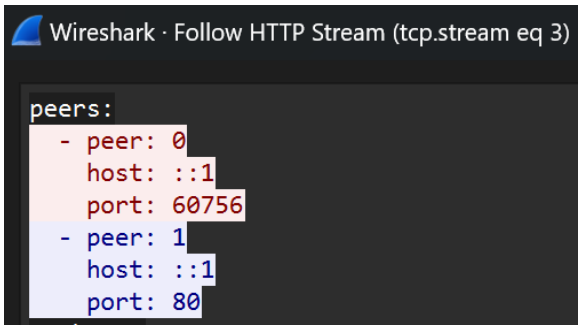
| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| ::1 | ::1 | HTTP | 756 | GET / HTTP/1.1 |
| ::1 | ::1 | HTTP | 64 | HTTP/1.0 200 OK (text/html) |
| ::1 | ::1 | HTTP | 677 | GET /favicon.ico HTTP/1.1 |
| ::1 | ::1 | HTTP | 64 | HTTP/1.0 200 OK (text/html) |

Linje 1 har klienten som avsender og webtjeneren som mottaker. Linje 2 har webtjeneren som avsender og klienten som mottaker. Linje 3 har det samme som linje 1, mens linje 4 er lik linje 2. Det er altså en vekselsvis kommunikasjon mellom webtjeneren og klienten.

Dersom vi utfører Wireshark-funksjonen «Follow->HTTP -stream» ved å høyreklikke, på linje 1, vil vi få et mer detaljert innblikk i meldingen sendt fra klienten til webtjeneren. Klienten vil i en HTTP «samtale» alltid sende en melding til webtjeneren først, akkurat som en 3-way-handshake. Denne handshaken kan finnes igjen etter å ha kjørt «Follow->HTTP-stream», fordi display-filteret da vil bli satt til «tcp.stream eq {tall}». Følgende Wireshark-rader viser handshaken mellom klienten og tjeneren:

| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| ::1 | ::1 | TCP | | 76 60376 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM |
| ::1 | ::1 | TCP | | 76 80 → 60376 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM |
| ::1 | ::1 | TCP | | 64 60376 → 80 [ACK] Seq=1 Ack=1 Win=327168 Len=0 |

I tillegg kan følgende leses av vinduet som dukker opp ved «Follow->HTTP -stream»:




Her ser vi hvem den følgende samtalen er mellom. TCP-stream nummer 3 er altså i denne pakkefangsten mellom klienten på port 60756 og webtjeneren på port 80. Legg merke til at dette skjermbildet ble tatt ved en annen kjøring enn de andre skjermbildene, derfor er ikke portnummeret lenger 60376, da dette er noe som kan variere ved oppstart av nettleseren (og andre programmer).

## 1.3.2 Overføring av data

Etter oppkoblingen, sender klienten en GET-forespørsel til webtjeneren, hvor den gir webtjeneren beskjed om hvilke ressurser den ønsker å få tak i av tjeneren. I dette tilfellet er det ingen ressurs å hente direkte, slik som det vanligvis gjøres ved et oppslag på en nettside. Her vil det kun være «hardkodet» tekst fra tjeneren som sendes og vises hos klienten (i nettleseren). Dersom det hadde eksistert en «index.html»-fil som tjeneren kunne ha vist klienten, ville webtjeneren ha gitt den til klienten dersom den gav tjeneren beskjed om at den for eksempel ønsket å se på nettsiden «192.168.0.54/hjemmeside». Dette er ikke tilfellet i dette enkle eksemplet.

Webtjeneren vil på lik linje med det forrige eksemplet overføre data ved hjelp av Java-klassen «PrintWriter», noe som vil medføre at den overfører innholdet ved TCP-pakker. I dette tilfellet overføres det tekst og et bilde, noe som vil resultere i at meldingen deles opp mange små TCP-segmenter.

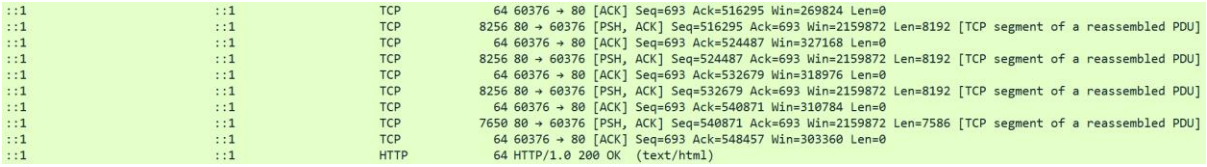
Følgende kodesnutt viser innholdet som sendes fra webtjeneren til klienten:

```java
// Send HTTP-header and a message to the client
System.out.println("Sending HTTP-header and some HTML to the client...");
writer.println(
        "HTTP/1.0 200 OK\n" +
                "Content-Type: text/html; charset=utf-8\n\n" +
                "<html><body>\n" +
                "<h1>This is a message from the client!</h1>\n" +
                "This client is running on thread no. " + threadNumber + "\n" +
                "<ul>\n" +
                "<li>Element 1</li>\n" +
                "</ul>\n" +
                "</body></html>"
);
System.out.println("HTML sent!\n");

System.out.println("Sending image in HTML...");
byte[] imageBytes = Files.readAllBytes(new File( pathname: "src/WebServer/idurogjugem.jpg").toPath());
String base64EncodedImage = Base64.getEncoder().encodeToString(imageBytes);

writer.println(
        "HTTP/1.0 200 OK\n" +
                "Content-Type: text/html\n\n" +
                "<html><body>" +
                "<h1>Image title!!! <:) &#128511;&#128128;&#x1F480;</h1>" +
                "<img src=\" " + base64EncodedImage + "\" " +
                "width=450px alt=\"Your image\">\n" +
                "</body></html>"
);
System.out.println("Image in HTML sent!\n");
```

Først sendes det tekst i HTML-format, som klienten vil visualisere. Etter dette, overføres det et bilde, som har blitt omgjort fra bytes til en base64-streng. Dette kan tolkes av nettleseren direkte dersom vi spesifiserer at datatypen til «img»-taggen er et bilde/jpeg, og at formatet er i base64.

De følgende skjermbildene av begynnelsen og slutten av dataoverføringen fra webtjeneren til klienten er hentet fra Wireshark:

| | | | |
|---|---|---|---|
| ::1 | ::1 | HTTP | 756 GET / HTTP/1.1 |
| ::1 | ::1 | TCP | 64 80 → 60376 [ACK] Seq=1 Ack=693 Win=2159872 Len=0 |
| ::1 | ::1 | TCP | 262 80 → 60376 [PSH, ACK] Seq=1 Ack=693 Win=2159872 Len=198 [TCP segment of a reassembled PDU] |
| ::1 | ::1 | TCP | 64 60376 → 80 [ACK] Seq=693 Ack=199 Win=327168 Len=0 |
| ::1 | ::1 | TCP | 8256 80 → 60376 [PSH, ACK] Seq=199 Ack=693 Win=2159872 Len=8192 [TCP segment of a reassembled PDU] |
| ::1 | ::1 | TCP | 64 60376 → 80 [ACK] Seq=693 Ack=8391 Win=318976 Len=0 |
| ::1 | ::1 | TCP | 8256 80 → 60376 [PSH, ACK] Seq=8391 Ack=693 Win=2159872 Len=8192 [TCP segment of a reassembled PDU] |
| ::1 | ::1 | TCP | 64 60376 → 80 [ACK] Seq=693 Ack=16583 Win=310784 Len=0 |
| ::1 | ::1 | TCP | 8256 80 → 60376 [PSH, ACK] Seq=16583 Ack=693 Win=2159872 Len=8192 [TCP segment of a reassembled PDU] |
| ::1 | ::1 | TCP | 64 60376 → 80 [ACK] Seq=693 Ack=24775 Win=302592 Len=0 |



| | | | |
|---|---|---|---|
| ::1 | ::1 | TCP | 64 60376 → 80 [ACK] Seq=693 Ack=516295 Win=269824 Len=0 |
| ::1 | ::1 | TCP | 8256 80 → 60376 [PSH, ACK] Seq=516295 Ack=693 Win=2159872 Len=8192 [TCP segment of a reassembled PDU] |
| ::1 | ::1 | TCP | 64 60376 → 80 [ACK] Seq=693 Ack=524487 Win=327168 Len=0 |
| ::1 | ::1 | TCP | 8256 80 → 60376 [PSH, ACK] Seq=524487 Ack=693 Win=2159872 Len=8192 [TCP segment of a reassembled PDU] |
| ::1 | ::1 | TCP | 64 60376 → 80 [ACK] Seq=693 Ack=532679 Win=318976 Len=0 |
| ::1 | ::1 | TCP | 8256 80 → 60376 [PSH, ACK] Seq=532679 Ack=693 Win=2159872 Len=8192 [TCP segment of a reassembled PDU] |
| ::1 | ::1 | TCP | 64 60376 → 80 [ACK] Seq=693 Ack=540871 Win=310784 Len=0 |
| ::1 | ::1 | TCP | 7650 80 → 60376 [PSH, ACK] Seq=540871 Ack=693 Win=2159872 Len=7586 [TCP segment of a reassembled PDU] |
| ::1 | ::1 | TCP | 64 60376 → 80 [ACK] Seq=693 Ack=548457 Win=303360 Len=0 |
| ::1 | ::1 | HTTP | 64 HTTP/1.0 200 OK (text/html) |

Overføringen av teksten og bildet vil foregå fra det øverste bilde, til og med det siste. Etter at overføringen har funnet sted, vil webtjeneren sende en HTTP-pakke med koden 200, som indikerer at overføringen er ferdig og foregikk uten feil. Dette leses av den siste linjen på det andre bildet.

### 1.3.3 Nedkobling

Etter at teksten og bildet har blitt overført, lukker brukeren av klienten/nettleseren programmet. Dermed vil den vedvarende TCP-forbindelsen mellom webtjeneren og klienten kobles ned. I dette tilfellet vil det kun være klienten som sender en TCP-pakke med et aktivt FIN- og ACK-flagg, mens webtjeneren kun vil svare med et ACK-flagg.

### 1.3.4 Porter, IP- og MAC-adresser

I dette tilfellet foregår kommunikasjonen mellom klienten på port 60376 og webtjeneren på port 80. Kommunikasjonen foregår over IPv6 protokollen lokalt på maskinen (localhost ved IPv6 slåes opp ved «[::1]» i nettleseren). Det vil ikke være i bruk noe fysisk nettverkskort, da trafikken går gjennom maskinens egne loopback-funksjon. MAC-adresse er derfor ikke-eksisterende og irrelevant i dette eksemplet.

# 2. Øving 4 – tjener og klient over UDP og TLS

## 2.1 Innledning

Denne øvingen består av to oppgaver. Den første er lik oppgave 1 i øving 3, bortsett fra at dataoverføringen skal foregå ved hjelp av UDP på transportlaget, fremfor TCP. Oppgave 2 består av dataoverføring mellom en tjener- og klientklasse ved hjelp av TCP og TLS.

## 2.2 UDP

Målet med programmet er å overføre en UDP-pakke med to tall og en operator til en tjener. Tjeneren skal kalkulere uttrykket og gi klienten et svar på dette uttrykket. Klienten sender de tre verdiene i en streng, separert med kommaer. Tjeneren er derfor nødt til å skille disse tre verdiene fra hverandre, løse regnestykket, og deretter svare klienten.

Tjeneren og klienten kjører lokalt på den samme maskinen, derfor vil «localhost/127.0.0.1» bli brukt også i denne øvingen. MAC-adresse er ikke relevant, da det ikke tas i bruk et fysisk nettverkskort. Tjeneren kjører på den selvvalgte porten 9999, mens klienten (i denne kjøringen) benytter seg av port 51465.

Programmet kjøres, mens Wireshark «sniffer» på loopback-trafikk i bakgrunnen. Display-filteret settes til «udp», og følgende linjer vises:

| Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|
| 127.0.0.1 | 127.0.0.1 | UDP | 41 | 51465 → 9999 Len=9 |
| 127.0.0.1 | 127.0.0.1 | UDP | 36 | 9999 → 51465 Len=4 |

Dette stemmer godt overens med Java-koden, hvor klienten sender én enkelt pakke:

```
// Send the data to the server
String sendDataString = firstNumber + "," + secondNumber + "," + operator;
sendData = sendDataString.getBytes();
sendPacket = new DatagramPacket(sendData, sendData.length, InetAddress.getByName(iNetAddress), portNumber);
clientSocket.send(sendPacket);
System.out.println("sent!");
```

Etter dette sender tjeneren også kun én enkelt pakke tilbake:

```
// Respond to the client
System.out.print("Sending response ... ");
String resultString = String.format("%.2f", result);
byte[] responseMessage = resultString.getBytes();
DatagramPacket responsePacket = new DatagramPacket(responseMessage, responseMessage.length,
        clientIPAddress, port);
ss.send(responsePacket);
System.out.println("sent!");
```

Denne UDP-kommunikasjonen kan verifiseres til å foregå mellom programmets tjener og klient ved å printe ut klientens lokale port:

```
System.out.println("This client is using port " + clientSocket.getLocalPort());
```

Denne linjen gir følgende resultat i terminalen:

```
This client is using port 51465
```

Samtidig vil tjenerens port være satt i følgende kode:

```
final int PORT_NUMBER = 9999;
```

```
// Initialize server-socket on declared port
try (DatagramSocket ss = new DatagramSocket(PORT_NUMBER)) {
```

### 2.2.1 Dataoverføring

Ved å inspisere linje 1 i Wireshark nærmere, kan det leses av at pakken sendes gjennom IPv4 (lokalt i dette tilfellet) på nettverkslaget, og ved bruk av User Datagram Protocol (UDP) på transportlaget. Lengden til UDP-pakken sitt innhold/payload er på 9 bytes. Dette er UDP-pakken sendt av klienten til tjeneren.

Den neste linjen er svaret fra tjeneren til klienten. Dette er en UDP-pakke med en payload på størrelse 4 bytes. Dette gir mening, da det kun returneres én tallverdi. Pakken fra klienten til tjeneren vil være større, da det sendes en String med to tall og én operator.

UDP-overføringen er forbindelsesløs, som vil si at det ikke opprettes en forbindelse mellom klient og tjener før pakken sendes. Dette vil medføre at UDP-pakker kan bli sendt til «ingenting» dersom klienten ikke spesifiserer den riktige porten som tjeneren benytter seg av. Samtidig vil det ikke være viktig at tjeneren svarer klienten på rett vis, altså at tjeneren henter ut klientens port ved å lese av UDP-pakken sin del av headeren som beskriver avsenders IP-adresse og port. Dette gjøres ved følgende kode i Java:

```
// Receive UDP-packet and read its values
System.out.print("\nWaiting for a packet... ");
receivePacket = new DatagramPacket(receiveData, receiveData.length);
ss.receive(receivePacket);
System.out.println("received!");
System.out.print("Reading packet... ");
InetAddress clientIPAddress = receivePacket.getAddress();
int clientPort = receivePacket.getPort();
```

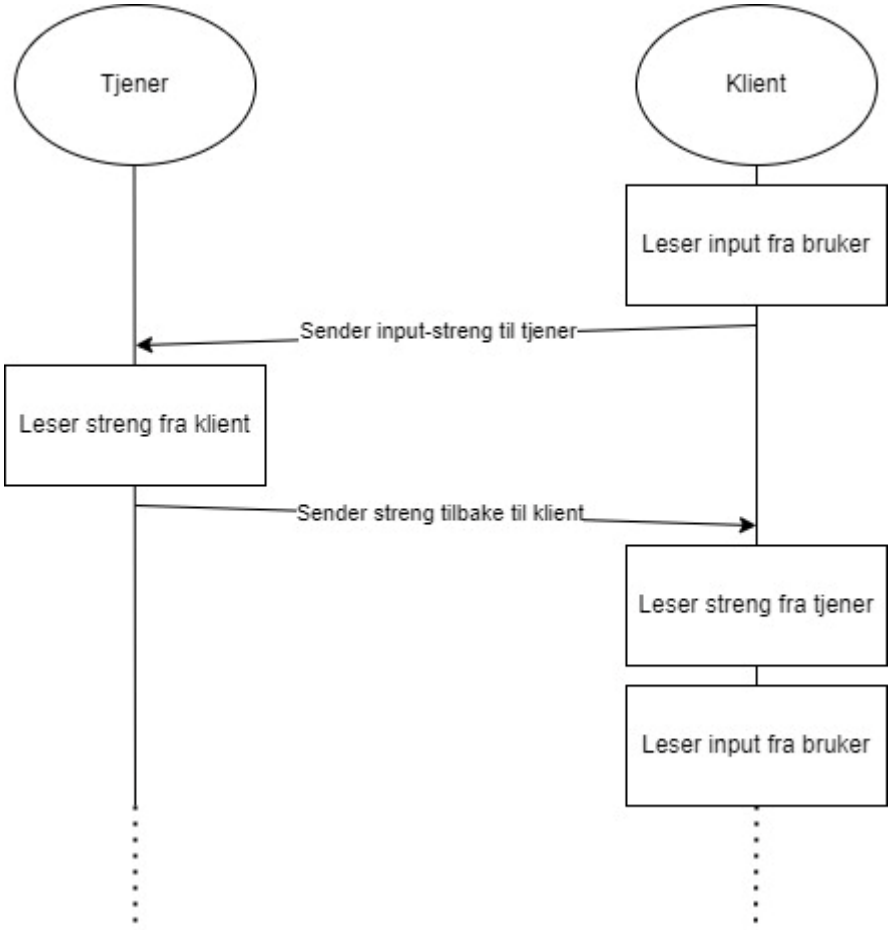
Her vil klientens (i dette tilfellet: IPv4-adresse) leses av og lagres i variabelen «clientIPAddress» og porten lagres i «clientPort».

### 2.2.2 Porter, IP- og MAC-adresser

Denne dataoverføringen foregår mellom en klient på port 51465 og en tjener på port 9999. Kommunikasjonen foregår lokalt på maskinen, gjennom IP-loopback (localhost/127.0.0.1). På bakgrunn av at trafikken går gjennom loopback, vil ikke maskinens fysiske nettverkskort benyttes, derfor er MAC-adresse ikke relevant.

## 2.3 TLS

Dette programmet er en enkel klient-tjener program, hvor dataoverføringen benytter seg av TLS for å etablere en sikker dataoverføring mellom tjener og klient. Det overføres data ved at brukeren av klienten skriver inn en String i terminalen, som deretter sendes til tjeneren. Tjeneren vil altså speile det som blir skrevet inn i klientens terminal. Samtidig vil tjeneren lese og sende denne Stringen tilbake til klienten. Kommunikasjonen kan visualiseres:



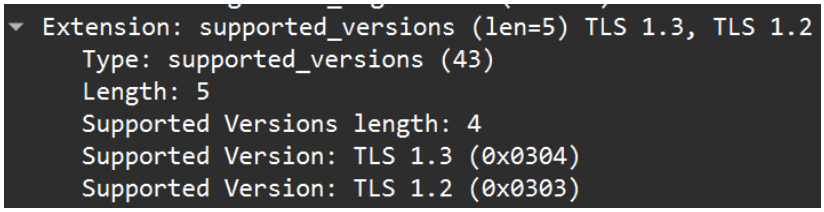

De stiplede linjene markerer at programmet vil fortsette i en løkke, fram til det blir avbrutt av klienten.

### 2.3.1 Oppkobling – TLSv1.3

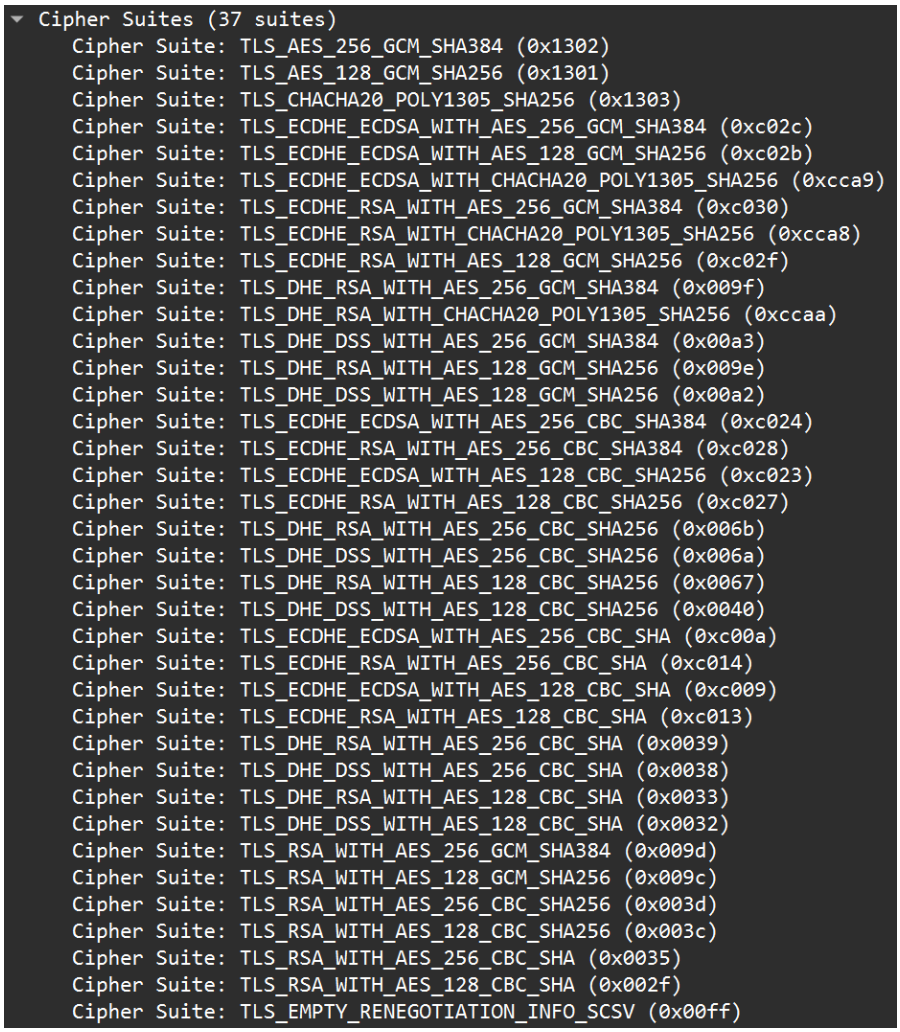
Klienten og tjeneren oppretter en kobling over TCP med en 3-way handshake. Deretter vil det sendes en pakke med en TLS-protokoll for å kunne etablere en forbindelse med kryptert dataoverføring. Display-filteret er satt til «tls», men det bør merkes at alle dataoverføringer vil bli kvittert fra den mottagende parten, som i en ordinær TCP-forbindelse.

Klienten starter med å sende en «Client Hello» til tjeneren. I denne meldingen vil klienten sende en liste med støttede TLS-versjoner, støttede krypteringsalgoritmer og annen informasjon tjeneren må forholde seg til.

For eksempel kan det leses av de støttede TLS-versjonene under «Extensions: supported_versions». I dette tilfellet støtter klienten både TLS versjon 1.2 og 1.3:

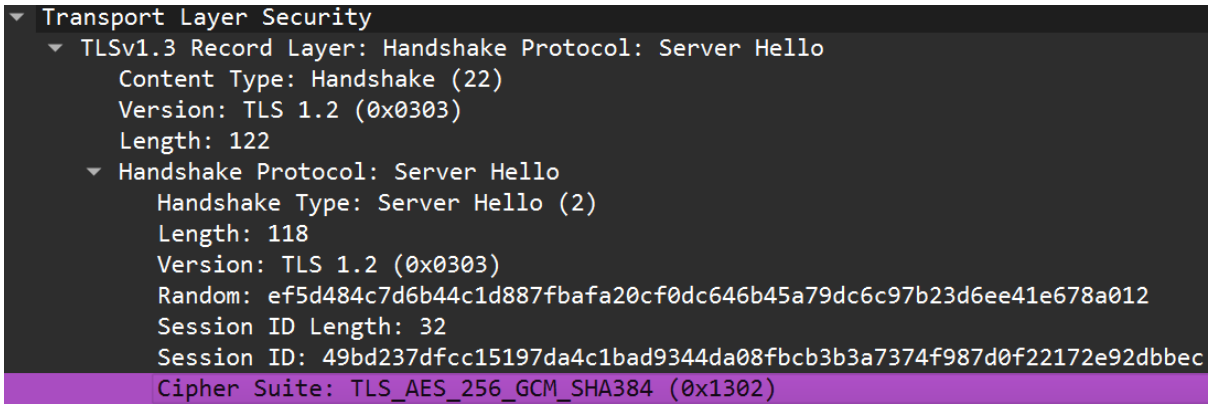



I tillegg kan vi lese av de støttede settene med krypteringsalgoritmer under «Cipher Suites»:

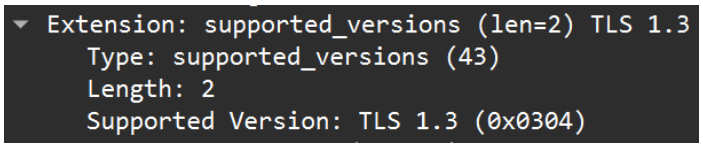


Det neste steget i opprettelsen av en TLS-forbindelse er at tjeneren leser «Client Hello» meldingen og dets innhold, før den blant annet velger en Cipher Suite og TLS versjon. Tjeneren velger blant annet en Cipher Suite og en TLS-versjon, og sender dette svaret til klienten gjennom en «Server Hello»-melding.

I dette tilfellet velger tjeneren Cipher Suiten TLS_AES_256_GCM_SHA384 (0x1302):




Selv om det her kan virke som at den valgte TLS-versjonen er 1.2, vil dette ikke stemme. Tjenerens valgte TLS-versjon finnes under «Extensions: supported_versions»:




Her har tjeneren valgt versjon 1.3. «Version»-feltet under «TLSv1.3 Record Layer» vil være satt til defaulten (0x0303) dersom TLSv1.3 er valgt, da TLSv1.3 ikke benytter seg av dette feltet til vanlig.

I «Server Hello»-meldingen vil tjeneren også gi informasjon om kryptografiske parametere til klienten, som vil si noe om den følgende nøkkel-byttingen som straks vil skje. Etter «Server Hello»-meldingen, er dataoverføringen kryptert. Tjeneren sitt sertifikat vil ligge under pakken «Application Data» i Wireshark. Denne kan vi naturligvis ikke lese av, da den er kryptert. Dette er ikke tilfelle i TLSv1.2, hvor sertifikatet kunne leses ut ifra «Server Hello» sin «Handshake protocol: Certificate». Sertifikatet er i TLSv1.3 altså kryptert og ikke lesbart i Wireshark (Rescorla, 2018).

## 2.3.2 Oppkobling - TLSv1.2

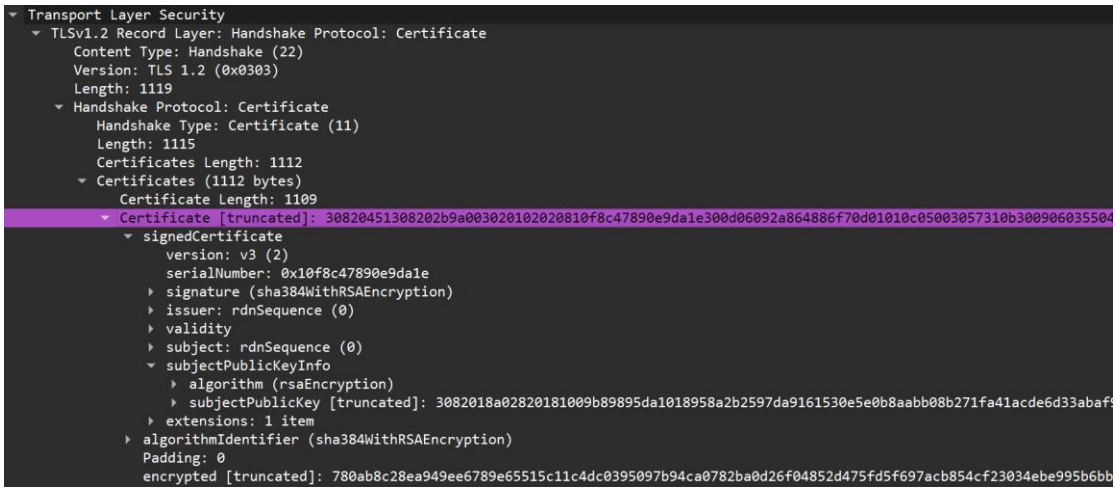
Programmet kan derimot tvinges til å kjøre over TLSv1.2 i stedet for TLSv1.3. Dette kan gjøres ved å kjøre .class-filene med følgende tilleggsargument:

«-Djdk.tls.client.protocols=’TLSv1.2’»

Med TLSv1.2 som TLS-protokoll dukker det opp flere linjer i Wireshark, som under TLSv1.3 er krypterte:

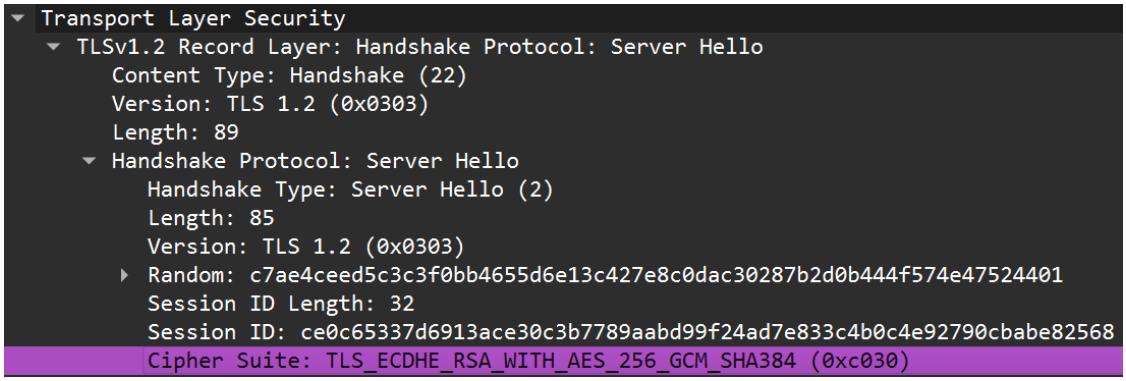
```
127.0.0.1    127.0.0.1    TLSv1.2    1168 Certificate
127.0.0.1    127.0.0.1    TLSv1.2     477 Server Key Exchange
127.0.0.1    127.0.0.1    TLSv1.2      53 Server Hello Done
127.0.0.1    127.0.0.1    TLSv1.2      86 Client Key Exchange
```

I TLSv1.2 er ikke overleveringen av sertifikatet eller nøkkelutvekslingene krypterte. Tjeneren identifiseres seg for klienten med sertifikatet som kan finnes under «Handshake Protocol: Certificate» i Wireshark:




Sertifikatet er lengre enn det som vises på bildet, totalt litt over 1KB.

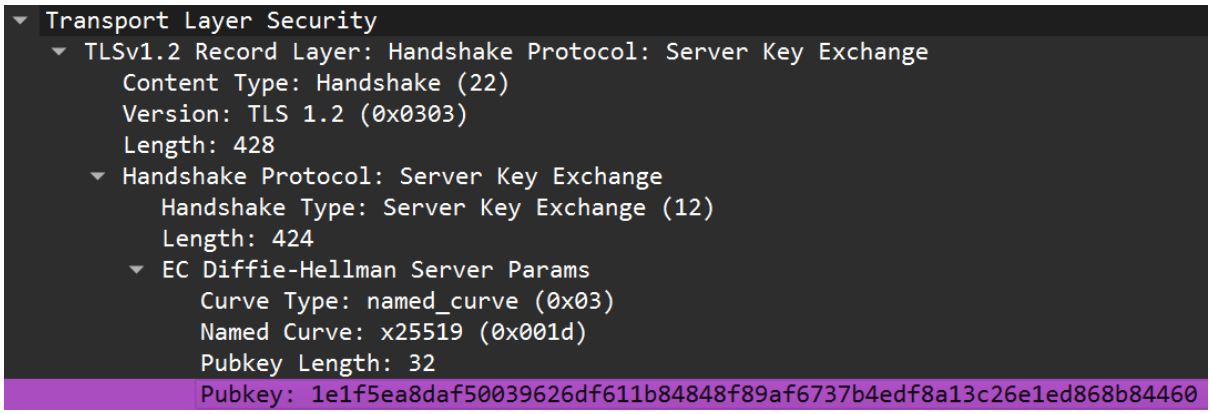
I dette tilfellet velges en annen kryptografisuite (Cipher Suite) enn ved TLSv1.3-eksempelet:



Nøkkelutvekslingen er det neste steget som tas før kommunikasjonen mellom tjener og klient blir kryptert. Dette skjer i de følgende linjene i Wireshark:

```
TLSv1.2       1168 Certificate
TLSv1.2        477 Server Key Exchange
TLSv1.2         53 Server Hello Done
TLSv1.2         86 Client Key Exchange
```

Tjeneren sender i «Server Key Exchange» sin offentlige nøkkel under «pubkey»:




Både tjeneren og klienten har hver sine private nøkler, som ikke deles med hverandre eller noen andre. De kombinerer denne offentlige nøkkelen med deres egne private nøkler, før de deretter deler deres «blandede» nøkler med hverandre. De blandede nøklene kommer av å mikse deres private nøkler med den offentlige nøkkelen utsendt av tjeneren. Deretter tilsetter de deres egne private nøkkel til den motstående parten sin «blandede» nøkkel. Til slutt vil de begge ende opp med den samme, hemmelige nøkkelen. Denne nøkkelen brukes videre til å danne sesjonsnøkler for den symmetriske krypteringen av meldingene mellom tjener og klient. Kommunikasjonen mellom tjener og klient foregår nå i en TLS-sesjon.

Denne fremgangsmåten tar utgangspunkt i at det er svært vanskelig å reversere en blanding av nøklene.

Etter at de hemmelige nøklene, og grunnlaget for sesjonsnøklene, er fullført, sender tjeneren en «Server Hello Done», som indikerer at tjeneren har fått delt sin nødvendige informasjon med klienten.

Tjener og klient deler altså sine «blandede» nøkler med hverandre i stegene «Server Key Exchange» og «Client Key Exchange». Etter dette, vil kommunikasjonen mellom partene være kryptert.

### 2.3.3 Dataoverføring

Overføringen av data mellom tjener og klient vil være kryptert i både TLSv1.3 og TLSv1.2. Derfor kan ikke pakkene her tolkes utover en ordinær TCP-kommunikasjon.

### 2.3.4 Nedkobling

For å kunne gjennomføre en trygg nedkobling, er TLS-sesjonen nødt til å opphøre før de ordinære TCP-flaggene FIN og ACK veksles mellom tjener og klient. Dette gjøres slik at tjener og klient kan være sikker på at det er en av dem, og ikke en tredjeparts angriper, som ønsker å avslutte forbindelsen. Denne TLS-nedkoblingen skjer i følgende to pakker fra Wireshark (linje 1 er fra klient til tjener, mens linje 3 er fra tjener til klient):

```
TLSv1.2          75 Encrypted Alert
TCP              44 8000 → 62519 [ACK] Seq=3189 Ack=492 Win=2160640 Len=0
TLSv1.2          75 Encrypted Alert
TCP              44 62519 → 8000 [ACK] Seq=492 Ack=3220 Win=2158080 Len=0
```

Etter at TLS-sesjonen er avsluttet, nedkobles TCP-forbindelsen på ordinært vis ved hjelp av FIN- og ACK-flagg-pakker.

### 2.3.5 Porter, IP- og MAC-adresser

I denne kjøringen ble det benyttet protokoller fra TCP/IP-familien på «localhost/127.0.0.1». Klienten benyttet seg av port 61971 og tjeneren av port 8000. Det ble benyttet loopback ved IPv4, derfor er ikke MAC-adresse relevant, da det ikke ble benyttet et fysisk nettverkskort i kommunikasjonen mellom tjener og klient.

# 3. Referanser

Oracle. (2024, 8. januar). *ServerSocket - setPerformancePreferences*. Retrieved from Java SE 8: https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html

Rescorla, E. (2018). *The Transport Layer Security (TLS) Protocol Version 1.3*. Retrieved from RFC Editor: https://www.rfc-editor.org/rfc/rfc8446#page-11