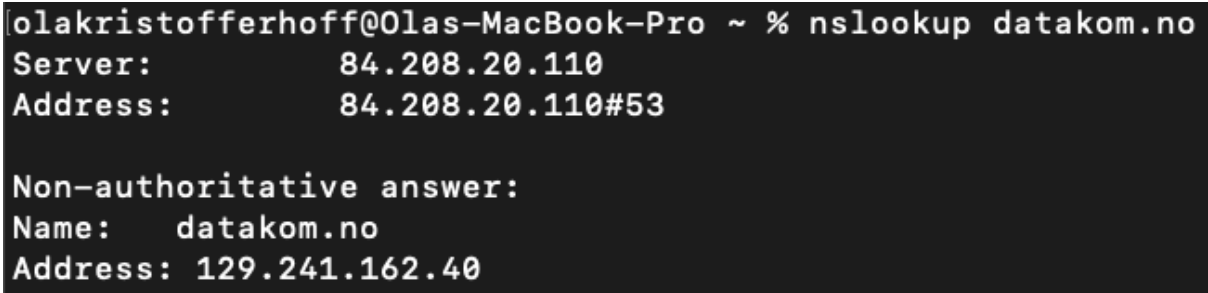


Datakommunikasjon – Øving 1

IDATT2104- Nettverksprogrammering

Gruppen består av Oline Amundsen og Ola Kristoffer Hoff. Vi har begge MacOS systemer og bruker derfor deres ekvivalente kommandoer (f.eks. *ifconfig* isteden for *ipconfig*). Noen av pakkefangstene våre er også tatt på forskjellige tidspunkt, pga. f.eks. ved gjentak av data, derfor kan resultat variere, men konklusjonene våre er like.

Oppgave 1: Wireshark og DNS

- a) 
- ```

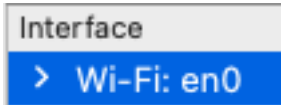
olakristofferhoff@Olas-MacBook-Pro ~ % nslookup datakom.no
Server: 84.208.20.110
Address: 84.208.20.110#53

Non-authoritative answer:
Name: datakom.no
Address: 129.241.162.40

```

Bilde 1: Skjermdump av terminalen, ved 'nslookup' til 'datakom.no'.

Server angir navnet på den lokale navnetjeneren, og Address angir dens IP-adressen. Svare på navneoppslaget består av Name som angir navnet på siden vi gjorde oppslag på, og Address gir oss IP-adressen til siden.



Bilde 2: Skjermdump fra Wireshark. Viser at Wi-Fi er nettverkskort en0.

- b) Ved bruk av *ifconfig* får vi listet ut 20 nettverkskort og deres informasjon. Vi spesifiserte oss inn på kort *en0*, som er Wi-Fi nettverkskortet på våre datamaskiner, ved kommandoen *ifconfig en0*.

```

olakristofferhoff@Olas-MacBook-Pro ~ % ifconfig en0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
 options=400<CHANNEL_IO>
 ether a4:83:e7:3a:3f:99
 inet6 fe80::cd9:8efe:2379:5a19%en0 prefixlen 64 secured scopeid 0x6
 inet 192.168.0.125 netmask 0xfffff00 broadcast 192.168.0.255
 nd6 options=201<PERFORMNUD,DAD>
 media: autoselect
 status: active

```

Bilde 3: Skjermdump av terminalen, ved 'ifconfig en0'.

Her fikk vi ikke med opplysninger om DNS-tjenere og standardruting, dette finner vi ved å bruke *netstat -r*. Da får vi listet opp frp alle nettverkskortene, så vi brukte heller kommandoen *netstat -r | grep en0*, som kun ga oss informasjon angående *en0* nettverkskortet.

```

olakristofferhoff@Olas-MacBook-Pro ~ % netstat -r | grep en0
default 192.168.0.1 UGSc en0
169.254 link#6 UCS en0
192.168.0 link#6 UCS en0
192.168.0.1/32 link#6 UCS en0
192.168.0.1 b8:d5:26:dd:14:d0 UHLWIir en0 1160
192.168.0.101 d6:73:8d:17:ee:15 UHLWii en0 1166
192.168.0.112 7a:44:f6:51:8f:c UHLWii en0 998
192.168.0.125/32 link#6 UCS en0
192.168.0.158 link#6 UHRLWI en0
192.168.0.159 link#6 UHRLWI en0
192.168.0.173 link#6 UHRLWI en0
224.0.0/4 link#6 UmCS en0
224.0.0.251 1:0:5e:0:0:fb UHmLWI en0
255.255.255.255/32 link#6 UCS en0
fe80::%en0 link#6 UCI en0
fe80::4c7:ed51:5ad 0:e0:4c:68:6c:c0 UHLWI en0
fe80::1054:5624:f9 7a:44:f6:51:8f:c UHLWI en0
fe80::bad5:26ff:fe b8:d5:26:dd:14:d0 UHLWii en0
ff00:: link#6 UmCI en0
ff01::%en0 link#6 UmCI en0
ff02::%en0 link#6 UmCI en0

```

Bilde 4: Skjermdump av terminalen ved kommandoene 'netstat -r | grep en0'.

Kolonene er henholdsvis: *Destination*, *Gateway*, *Flags*, *Netif* og *Expire*. Vi ser at den første raden i bilde over har *Destination: default* og at standardruterer (Gateway) er 192.168.0.1.

Videre ser vi at den ene navnetjeneren (DNS tjeneren) har samsvarende adresse med *Server* feltet i oppgave a. Dette stemmer bra, da vi sa at *Server* feltet var adressen til den lokale navnetjeneren.

```

DNS Servers:
84.208.20.110
84.208.20.111

```

Bilde 5: Skjermdump av DNS tjenerne som en0 (Wi-Fi) er konfigurert med. Funnet under avanserte nettverksinnstillinger.

- c) Vi ser at verdiene til *Type* og *Time to live* (levetid) er respektivt *A (Host Address) (1)* og *4394 (1 hour, 13 minutes, 14 seconds)*.

Verdi *A* i *Type* betyr *Address* som inneholder en IP-adresse som i dette tilfellet er *Host Address* altså IP-adressen til datakom.no

(129.241.162.40). I vårt tilfelle, ved et *nslookup*, brukes IP-adressen til å informere den som spør (klienten) om hva IP-adressen til datakom.no er.

```

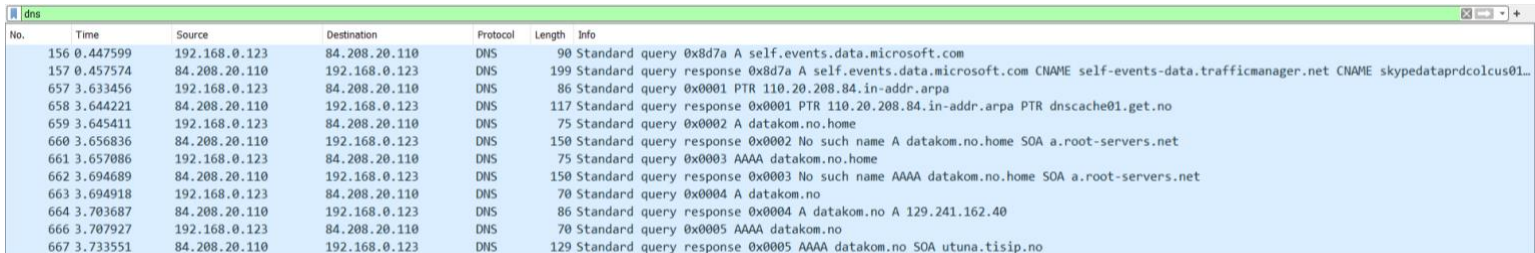
Answers
datakom.no: type A, class IN, addr 129.241.162.40
Name: datakom.no
Type: A (Host Address) (1)
Class: IN (0x0001)
Time to live: 4394 (1 hour, 13 minutes, 14 seconds)
Data length: 4
Address: 129.241.162.40

```

Bilde 6: Skjermdump av Wireshark. Her ser vi på 'Answer' delen i 'DNS' feltet til svarpakken til 'nslookup'.

Verdien *Time to live* er et tall som angir antall sekunder til den gitte ressursen burde fjernes fra *cache*'et (Kurose & Ross, 2017).

- d) For denne oppgaven brukte vi en Windows-maskin, da macOS ikke gjør som teorien i læreboka (Hallsteinsen, Klefstad, & Skundberg, 2020) tilsier.



| No. | Time     | Source        | Destination   | Protocol | Length | Info                                                                                                                                    |
|-----|----------|---------------|---------------|----------|--------|-----------------------------------------------------------------------------------------------------------------------------------------|
| 156 | 0.447599 | 192.168.0.123 | 84.208.20.110 | DNS      | 90     | Standard query 0x8d7a A self.events.data.microsoft.com                                                                                  |
| 157 | 0.457574 | 84.208.20.110 | 192.168.0.123 | DNS      | 199    | Standard query response 0x8d7a A self.events.data.microsoft.com CNAME self-events-data.trafficmanager.net CNAME skypedataprdcolcus01... |
| 657 | 3.633456 | 192.168.0.123 | 84.208.20.110 | DNS      | 86     | Standard query 0x0001 PTR 110.20.208.84.in-addr.arpa                                                                                    |
| 658 | 3.644221 | 84.208.20.110 | 192.168.0.123 | DNS      | 117    | Standard query response 0x0001 PTR 110.20.208.84.in-addr.arpa PTR dnscache01.get.no                                                     |
| 659 | 3.645411 | 192.168.0.123 | 84.208.20.110 | DNS      | 75     | Standard query 0x0002 A datakom.no.home                                                                                                 |
| 660 | 3.656836 | 84.208.20.110 | 192.168.0.123 | DNS      | 150    | Standard query response 0x0002 No such name A datakom.no.home SOA a.root-servers.net                                                    |
| 661 | 3.657086 | 192.168.0.123 | 84.208.20.110 | DNS      | 75     | Standard query 0x0003 AAAA datakom.no.home                                                                                              |
| 662 | 3.694689 | 84.208.20.110 | 192.168.0.123 | DNS      | 150    | Standard query response 0x0003 No such name AAAA datakom.no.home SOA a.root-servers.net                                                 |
| 663 | 3.694918 | 192.168.0.123 | 84.208.20.110 | DNS      | 70     | Standard query 0x0004 A datakom.no                                                                                                      |
| 664 | 3.703687 | 84.208.20.110 | 192.168.0.123 | DNS      | 86     | Standard query response 0x0004 A datakom.no A 129.241.162.40                                                                            |
| 666 | 3.707927 | 192.168.0.123 | 84.208.20.110 | DNS      | 70     | Standard query 0x0005 AAAA datakom.no                                                                                                   |
| 667 | 3.733551 | 84.208.20.110 | 192.168.0.123 | DNS      | 129    | Standard query response 0x0005 AAAA datakom.no SOA utuna.tisip.no                                                                       |

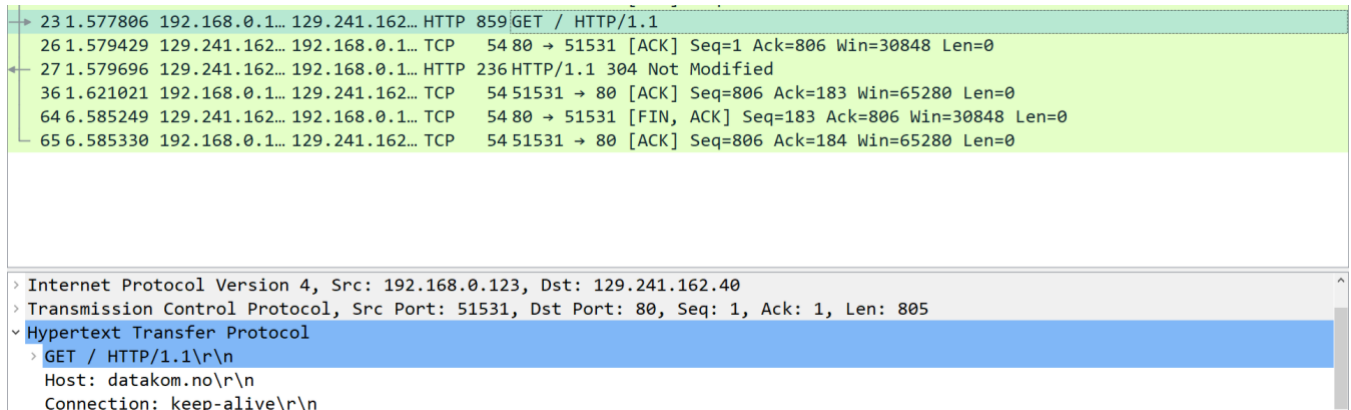
Bilde 7: Skjermdump fra Wireshark av navneoppslag på datakom.no

Vi ser i bildet over at det er flere DNS-oppslag på datakom.no, og alle med forskjellig transaction-id, for eksempel «0x0002». DNS-navnetjeneren vet ikke hva slags «type» datakom.no har, og spør derfor om flere typer frem til den får et svar. *Type* kan ha mange verdier, og som vi ser slår den opp på «A» (IPv4) og «AAAA» (IPv6), for datakom.no.home, og får svar om at den ikke finnes. Deretter slår den opp på henholdsvis A og AAAA datakom.no og får svar på A om en IP-adresse, 129.241.162.40. Dette er også svaret vi fikk av *nslookup*. Legg merke til at transaction-id'en inkrementeres per forespørsel, den starter med 0x0001 og den siste DNS-pakken var 0x0005.

## Oppgave 2: Wireshark og HTTP

Vi måtte bruke en Windows-maskin i denne oppgaven, da teorien ikke stemte i praksis på macOS.

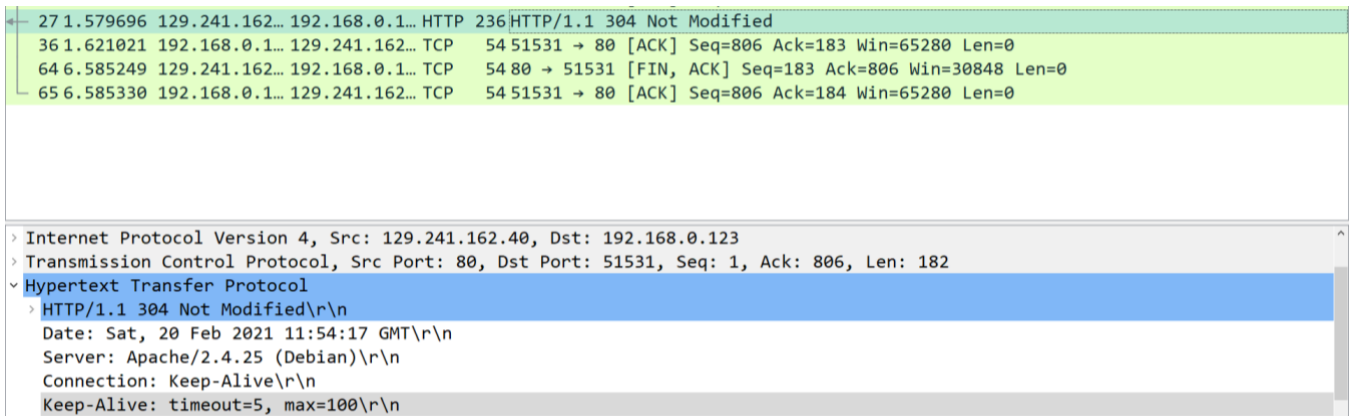
- a) HTTP-headerlinjen som brukes for vedvarende forbindelser er *Connection*, og den er satt (ved ønske om vedvarende forbindelse) til *Keep-Alive*. Når tjeneren mottar denne pakken kan den svare med et felt *Keep-Alive* og verdier som vi ser i bildet under (Bilde 9): «*timeout=5, max=100*». Dette betyr at tjeneren venter maks i 5 sekunder på en ny pakke fra klienten, og avslutter forbindelsen om det ikke kommer noen. Tallet 100 betyr at tjeneren tar maksimalt imot 100 pakker på denne forbindelsen, om dette nås, lukkes forbindelsen og en ny må bli opprettet om det skal sendes flere pakker. *Max* feltet er et sikkerhetsgrep for å sikre at man får jevnere fordeling av ressurser og at ingen sitter og holder alle linjene, det forhindrer også overbelastning av tjeneren.



The image shows a Wireshark packet capture. The top section displays a list of packets. Packet 27 is an HTTP GET request from 129.241.162 to 192.168.0.1. The packet details pane on the right shows the following structure:

- Internet Protocol Version 4, Src: 192.168.0.123, Dst: 129.241.162.40
- Transmission Control Protocol, Src Port: 51531, Dst Port: 80, Seq: 1, Ack: 1, Len: 805
- Hypertext Transfer Protocol
  - GET / HTTP/1.1\r\n
  - Host: datakom.no\r\n
  - Connection: keep-alive\r\n

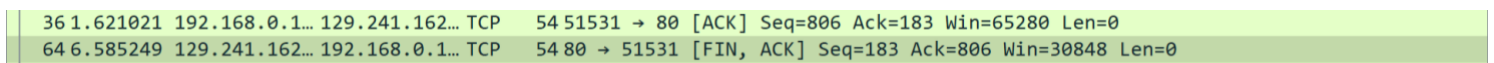
Bilde 8: Skjermdump fra Wireshark av 'Connection: keep-alive' i GET-pakken.



The image shows a Wireshark packet capture. The top section displays a list of packets. Packet 27 is an HTTP 304 Not Modified response from 129.241.162 to 192.168.0.1. The packet details pane on the right shows the following structure:

- Internet Protocol Version 4, Src: 129.241.162.40, Dst: 192.168.0.123
- Transmission Control Protocol, Src Port: 80, Dst Port: 51531, Seq: 1, Ack: 806, Len: 182
- Hypertext Transfer Protocol
  - HTTP/1.1 304 Not Modified\r\n
  - Date: Sat, 20 Feb 2021 11:54:17 GMT\r\n
  - Server: Apache/2.4.25 (Debian)\r\n
  - Connection: Keep-Alive\r\n
  - Keep-Alive: timeout=5, max=100\r\n

Bilde 9: Skjermdump fra Wireshark av tjenerens svar på GET-pakken.



The image shows a Wireshark packet capture. The top section displays a list of packets. Packet 64 is a TCP FIN, ACK packet from 129.241.162 to 192.168.0.1. The packet details pane on the right shows the following structure:

- 36 1.621021 192.168.0.1... 129.241.162... TCP 54 51531 → 80 [ACK] Seq=806 Ack=183 Win=65280 Len=0
- 64 6.585249 129.241.162... 192.168.0.1... TCP 54 80 → 51531 [FIN, ACK] Seq=183 Ack=806 Win=30848 Len=0

Bilde 10: Skjermdump fra Wireshark av [FIN, ACK] og siste melding som blir sendt før den

Vi kan se i bildet at tiden det tar mellom siste svar fra webtjener og til det kommer en TCP-pakke med FIN-flagg satt er ca. 5 sekunder (1.621021 til 6.585249).

- b) Hvis maskinen har en lokal mellomlagring vil den sende med HTTP-headeren «*If modified since*» som angir når den lokale kopien ble lagret. Webtjeneren sjekker om den har en nyere versjon basert på datoen i «*If modified since*» feltet. Om det finnes en nyere versjon, sender webtjener hele denne fila. Dersom det ikke er en nyere versjon sender webtjener kun statusmelding «*304 not modified*», som vil si at klienten kan bruke det lokale mellomlageret sitt, vi slipper da å overføre fila som sparer tid for klient, og reduserer belastning på nettet.

```

260.732531 192.168.0.1... 129.241.162... HTTP 869 GET / HTTP/1.1
270.734234 129.241.162... 192.168.0.1... TCP 54 80 → 51437 [ACK] Seq=1 Ack=816 Win=30848 Len=0
280.735079 129.241.162... 192.168.0.1... HTTP 236 HTTP/1.1 304 Not Modified
370.778657 192.168.0.1... 129.241.162... TCP 54 51437 → 80 [ACK] Seq=816 Ack=183 Win=65280 Len=0

```

---

```

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate\r\n
Accept-Language: nb-NO,nb;q=0.9,en-US;q=0.8,en-GB;q=0.7,en;q=0.6,no;q=0.5\r\n
Cookie: __utma=174267319.1800770440.1613820443.1613820443.1; __utmz=174267319; __utmc=174267319; __utmv=174267319.1.1.1
If-None-Match: "1a0e-5affab3c0e000"\r\n
If-Modified-Since: Wed, 23 Sep 2020 13:02:56 GMT\r\n

```

Bilde 11: Skjermdump fra Wireshark av 'HTTP-GET'-melding

```

28 0.735079 129.241.162... 192.168.0.1... HTTP 236 HTTP/1.1 304 Not Modified
37 0.778657 192.168.0.1... 129.241.162... TCP 54 51437 → 80 [ACK] Seq=816 Ack=183 Win=65280 Len=0

> Internet Protocol Version 4, Src: 129.241.162.40, Dst: 192.168.0.123
> Transmission Control Protocol, Src Port: 80, Dst Port: 51437, Seq: 1, Ack: 816, Len: 182
> Hypertext Transfer Protocol
 > HTTP/1.1 304 Not Modified\r\n

```

Bilde 12: Skjerdump fra Wireshark av svar på 'HTTP-GET'-meldingen

### Oppgave 3: Wireshark og TCP

3A.

|                |                |     |    |            |            |                  |
|----------------|----------------|-----|----|------------|------------|------------------|
| 192.168.0.125  | 129.241.162.40 | TCP | 78 | 63013 → 80 | [SYN]      | Seq=0 Win=65535  |
| 129.241.162.40 | 192.168.0.125  | TCP | 74 | 80 → 63013 | [SYN, ACK] | Seq=0 Ack=1      |
| 192.168.0.125  | 129.241.162.40 | TCP | 66 | 63013 → 80 | [ACK]      | Seq=1 Ack=1 Win= |

Bilde 13: Skjermdump av de tre pakkene involvert i '3-way handshake'.

| SRC   | → | DST   | FLAGG    | SEQNR | ACKNR |
|-------|---|-------|----------|-------|-------|
| 63013 | → | 80    | SYN      | 0     | 0     |
| 80    | → | 63013 | SYN, ACK | 0     | 1     |
| 63013 | → | 80    | ACK      | 1     | 1     |

Her har vi den utfylte tabellen.

Sekvensnummeret den første byte'en med nyttelast vil alltid være 0. Kvitteringsnummeret bekrefter antall mottatte bytes ved å angi hva forventet sekvensnummer kommer til å være på neste pakke, altså sekvensnummeret til pakken pluss antall bytes med data. Så om vi kun



sender 1 byte med data, får vi sekvensnummer pluss antall bytes med data, dermed  $0 + 1$ , vi får altså kvitteringsnummer 1.

## 3B.

| NR | Avsender  | DST-port | SEQNR | ACKNR | TCP PAYLOAD | Merknad                                          |
|----|-----------|----------|-------|-------|-------------|--------------------------------------------------|
| 1  | Klient    | 80       | 1     | 1     | 352         | GET forespørsel til webtjener                    |
| 2  | Webtjener | 63013    | 1     | 353   | 0           | TCP kvittering (pakke 1)                         |
| 3  | Webtjener | 63013    | 1     | 353   | 1448        | Data fra klient                                  |
| 4  | Klient    | 80       | 353   | 1449  | 0           | TCP kvittering (pakke 3)                         |
| 5  | Webtjener | 63013    | 1449  | 353   | 1448        | Data fra klient                                  |
| 6  | Webtjener | 63013    | 2897  | 353   | 1448        | Data fra klient                                  |
| 7  | Klient    | 80       | 353   | 4345  | 0           | TCP kvittering (pakke 6)                         |
| 8  | Webtjener | 63013    | 4345  | 353   | 1448        | Data fra klient                                  |
| 9  | Webtjener | 63013    | 5793  | 353   | 1165        | OK melding fra webtjener (siste data fra klient) |
| 10 | Klient    | 80       | 353   | 5793  | 0           | TCP kvittering (pakke 8)                         |
| 11 | Klient    | 80       | 353   | 6958  | 0           | Bekrefter OK fra webtjener                       |

I weboppslaget sender klienten (oss) en GET forespørsel til webtjeneren om å få index-filen til datakom.no. Deretter følger en pakke fra webtjener om at forespørselen er mottatt og deretter kommer index-filen fordelt utover fem pakker. Den er delt opp fordi maksimalt antall byte med nyttelast i en TCP-pakke (i dette tilfellet) er 1448, og hele filen er på 6957 byte. Teoretisk sett vet vi at Ethernet en maksimal pakkestørrelse på 1500 byte, vi brukte derimot Wi-Fi 5.0G, og vi fant ikke den maksimale pakkestørrelsen på denne. Uansett tar TCP-pakkeheader og IP-pakkeheader 20 byte hver, så på en Ethernet-kobling ville vi hatt maksimal nyttelast på 1460 byte. I praksis ser vi at vi har 1448 og dette kan for eksempel komme av at vi bruker Wi-Fi og ikke Ethernet.

Mens webtjener sender pakker med data, sender klienten ACK-pakker, som ikke har noe data, men er der for å bekrefte mottakelse av data. Det er ikke noe fasit på hvor mange bekreftelser klienten sender, da dette varierer basert på belastningen på klientsiden og når pakkene til webtjener kommer fram. I vårt tilfelle sendte klienten fire slike bekreftelser.

```

HTTP 418 GET / HTTP/1.1
TCP 66 80 → 63013 [ACK] Seq=1 Ack=353 Win=30080 Len=0 TSval=
TCP 1514 80 → 63013 [ACK] Seq=1 Ack=353 Win=30080 Len=1448 TSv
TCP 66 63013 → 80 [ACK] Seq=353 Ack=1449 Win=130304 Len=0 TS
TCP 1514 80 → 63013 [ACK] Seq=1449 Ack=353 Win=30080 Len=1448
TCP 1514 80 → 63013 [ACK] Seq=2897 Ack=353 Win=30080 Len=1448
TCP 66 63013 → 80 [ACK] Seq=353 Ack=4345 Win=127424 Len=0 TS
TCP 1514 80 → 63013 [ACK] Seq=4345 Ack=353 Win=30080 Len=1448
HTTP 1231 HTTP/1.1 200 OK (text/html)
TCP 66 63013 → 80 [ACK] Seq=353 Ack=5793 Win=125952 Len=0 TS
TCP 66 63013 → 80 [ACK] Seq=353 Ack=6958 Win=124800 Len=0 TS

```

Bilde 14: Skjermdump fra Wireshark. Viser pakkene involvert i overføringen av index-fila til datakom.no.

## Oppgave 4: Nettverk subnetting

| Nett | Nettadresse | CIDR | Min host  | Max host  | Broadcast | Tilgjengelig | Behov | Ledig |
|------|-------------|------|-----------|-----------|-----------|--------------|-------|-------|
| A    | a.b.c.0     | /25  | a.b.c.1   | a.b.c.126 | a.b.c.127 | 126          | 80    | 46    |
| L1   | a.b.c.128   | /26  | a.b.c.129 | a.b.c.190 | a.b.c.191 | 62           | 0     | 62    |
| C    | a.b.c.192   | /27  | a.b.c.193 | a.b.c.222 | a.b.c.223 | 30           | 20    | 10    |
| B    | a.b.c.224   | /28  | a.b.c.225 | a.b.c.238 | a.b.c.239 | 14           | 10    | 4     |
| L2   | a.b.c.240   | /28  | a.b.c.241 | a.b.c.254 | a.b.c.255 | 14           | 0     | 14    |

L1 og L2 er vilkårlige subnett som ikke i bruk. *Tilgjengelig* adresser vil være totalt adresserom (for eksempel a.b.c.0 - a.b.c.127 som totalt blir 128 mulige adresser) minus to, da den nederste er reservert til å være nettadresse og den øverste er *Broadcast*. Derav er også *Min host* en større enn nettadresse og *Max host* en mindre enn *Broadcast*. *Behov* er det som er oppgitt i oppgaven (for eksempel A har 80 (ansatte)), siden L1 og L2 er vilkårlige subnett som ikke er i bruk, satte vi *Behov* til null. *Ledig* vil være antall *Tilgjengelig* minus *Behov*. Kort sagt har vi gitt A, B og C subnett med minste tilgjengelighet større enn behovet deres.

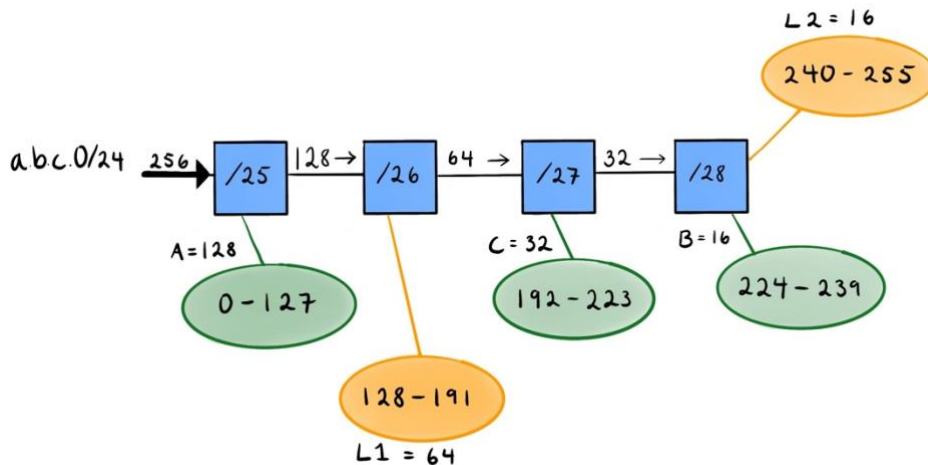


Figure 1: Illustrering av subnettene.

## Oppgave 5:

## Dokumentasjon av Programmeringsøving 1 – Tråder og socketprogrammering

## Øvingens del 1: Enkel tjener/klient (kalkulator)

a) Vi tar for oss tjenerne først.

```
try (ServerSocket serverSocket = new ServerSocket(PORT);
 Socket connection = serverSocket.accept())
```

Bilde 15: Kodelinje som oppretter en 'ServerSocket' og en 'Socket'.

I bilde over ser vi at en *ServerSocket* blir opprettet med *PORT* som argument, *PORT* er bare en *int* satt til 1250 (en tilfeldig valgt port i det kortlevde port intervallet). Deretter opprettes en *Socket* ved å kalle *.accept()*-metoden til *ServerSocket*, denne gjør at tjeneren venter på at en klient skal ta kontakt på denne porten (1250), når dette skjer blir det opprettet en *Socket* som tjener og klient kan kommunisere med.

```
inputStreamReader = new InputStreamReader(connection.getInputStream());
bufferedReader = new BufferedReader(inputStreamReader);
printWriter = new PrintWriter(connection.getOutputStream(), autoFlush: true);
```

Bilde 16: Kode som oppretter 'InputStreamReader', 'BufferedReader' og 'PrintWriter', for kommunikasjon gjennom 'Socket'.

Over ser vi at det blir opprettet *InputStreamReader*, med *Socket* sin *InputStream*. Kort sagt, så er jobben dennes å lese av alt som kommer inn til socket'en, altså alt tjener mottar fra klient. Så lages en *BufferedReader* som tar *InputStreamReader*'en som et argument, denne gjør det mulig å lagre det *InputStreamReader* leser i en buffer, så vi kan se på det senere. Til slutt opprettes en *PrintWriter* med *Socket* sin *OutputStream* (og *auto flush*'ing satt til *true*, det vil si at vi ikke manuelt må be den om å sende dataen, den gjør det selv.), denne sender da data gjennom *Socket* altså fra tjener til klient.

```
printWriter.println("Hei, du har kontakt med tjenersiden!");
```

Bilde 17: Kode for kontakt bekreftelse fra tjener til klient.

Tjeneren sender først en melding til klienten om at kommunikasjonen er koblet opp.

```
String message = bufferedReader.readLine();
while (message != null)
{
 System.out.println("En klient skrev: " + message);
 printWriter.println(calc(message));
 message = bufferedReader.readLine();
}
```

Bilde 18: Koden viser 'while'-løkken som sørger for at tjener kan motta mer enn en melding fra klienten.

Beskrevet i korte trekk leser vi her inn en melding fra klient ved '*readLine()*'-metoden til *BufferedReader*, så lenge det er data der vil vi kjøre logikken i '*while*'-løkken, om ikke vil vi koble ned kommunikasjonen. Inne i løkken skriver vi ut hva klienten sendte, deretter blir et svar beregnet med '*calc()*'-metoden og sendt tilbake til klient gjennom '*println()*'-metoden til *PrintWriter*. Til slutt leser vi av hva klienten svarer (Her vil koden vente i evig tid, det er ikke implementert noe maks tid på å få svar) tilsvarende som vi gjorde i starten.



```

printWriter.close();
bufferedReader.close();
inputStreamReader.close();

```

Bilde 19: Koden i bilde viser nedkoblingen av kommunikasjonen fra tjener siden.

Til slutt når klienten sender en «tom» melding (ikke noe data) og vi går ut av 'while'-løkken så vil vi gå inn i *finally*-blokken i koden. Her lukkes/stenges henholdsvis *PrintWriter*, *BufferedReader* og *InputStreamReader*. *ServerSocket* og *Socket* ble opprettet inne i *try*-blokken (try(her inne)) og vil derfor automatisk bli lukket/stengt når koden forlater *try*-blokken sitt *scope*.

Vi ser nå på klienten.

```

try (Socket connection = new Socket(host: "localhost", PORT))

```

Bilde 20: Koden viser at en 'Socket' blir opprettet med en 'host' og en 'port'.

Vi ser i bilde over at en *Socket* blir opprettet og tar to argumenter *host* og *port*. *PORT* er samme port som tjeneren er satt opp på (1250), og *host* argumentet tar IP-adressen til tjeneren, og siden vi kjører klient og tjener på samme maskin er den i dette tilfelle satt til 'localhost' (IP-adressen til maskinen klienten er på). Det blir altså opprettet en kommunikasjonslinje gjennom *Socket* mellom klient og tjener på en spesifisert port.

```

inputStreamReader = new InputStreamReader(connection.getInputStream());
bufferedReader = new BufferedReader(inputStreamReader);
printWriter = new PrintWriter(connection.getOutputStream(), autoFlush: true);

```

Bilde 21: Koden i bilde viser oppsettet av 'InputStreamReader', 'BufferedReader' og 'PrintWriter'.

Dette er akkurat det samme som skjer på tjener siden.

```

String answer = bufferedReader.readLine();
System.out.println("Tjener: " + answer);

```

Bilde 22: Koden viser innlesing av mottatt data fra tjener til klient.

I bilde over ser vi at klienten leser inn *answer* fra tjener, første gang er dette bare en beskjed om at vi har fått kontakt med tjeneren siden. Dette blir skrevet ut til klienten.

```
String message = scanner.nextLine();
while (!message.equals(""))
{
 printWriter.println(message);
 answer = bufferedReader.readLine();
 System.out.println("Tjener: " + answer);
 message = scanner.nextLine();
}
```

Bilde 23: Kode for å oppretthold kontakt med tjener frem til klienten velger å bryte den.

Denne koden er veldig lik tjener siden sin. I korte trekk så leser vi inn input fra klienten, om det sendes blankt avslutter vi, ellers sendes dataen til tjener. Om klienten sender inn data vil dette bli sendt til tjener gjennom *PrintWriter*. Klienten venter så på svar gjennom *BufferedReader* (Venter også her i evig tid, om den ikke får svar), og skriver ut svaret fra tjener til klienten. Til slutt leser vi igjen av input fra klienten, som i starten.

```
printWriter.close();
bufferedReader.close();
inputStreamReader.close();
```

Bilde 24: Kode for opprydding ved nedkobling av kommunikasjon.

Til slutt kobles kommunikasjonen ned, og dette gjøre på akkurat samme vis som på tjenersiden.

| No. | Time      | Source    | Destination | Protocol | Length | Info                                                                                                            |
|-----|-----------|-----------|-------------|----------|--------|-----------------------------------------------------------------------------------------------------------------|
| 31  | 5.150094  | 127.0.0.1 | 127.0.0.1   | TCP      | 68     | 63624 → 1250 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=827669529 TSecr=0 SACK_PERM=1                    |
| 32  | 5.150153  | 127.0.0.1 | 127.0.0.1   | TCP      | 68     | 1250 → 63624 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=827669529 TSecr=827669529 SACK_PERM=1 |
| 33  | 5.150162  | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | 63624 → 1250 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=827669529 TSecr=827669529                                 |
| 34  | 5.150168  | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | [TCP Window Update] 1250 → 63624 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=827669529 TSecr=827669529             |
| 35  | 5.153940  | 127.0.0.1 | 127.0.0.1   | TCP      | 160    | 1250 → 63624 [PSH, ACK] Seq=1 Ack=1 Win=408256 Len=104 TSval=827669532 TSecr=827669529                          |
| 36  | 5.153958  | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | 63624 → 1250 [ACK] Seq=1 Ack=105 Win=408192 Len=0 TSval=827669532 TSecr=827669532                               |
| 43  | 9.178405  | 127.0.0.1 | 127.0.0.1   | TCP      | 62     | 63624 → 1250 [PSH, ACK] Seq=1 Ack=105 Win=408192 Len=6 TSval=827673548 TSecr=827669532                          |
| 44  | 9.178466  | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | 1250 → 63624 [ACK] Seq=105 Ack=7 Win=408256 Len=0 TSval=827673548 TSecr=827673548                               |
| 45  | 9.182782  | 127.0.0.1 | 127.0.0.1   | TCP      | 58     | 1250 → 63624 [PSH, ACK] Seq=105 Ack=7 Win=408256 Len=2 TSval=827673552 TSecr=827673548                          |
| 46  | 9.182807  | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | 63624 → 1250 [ACK] Seq=7 Ack=107 Win=408192 Len=0 TSval=827673552 TSecr=827673552                               |
| 48  | 10.985010 | 127.0.0.1 | 127.0.0.1   | TCP      | 61     | 63624 → 1250 [PSH, ACK] Seq=7 Ack=107 Win=408192 Len=5 TSval=827675351 TSecr=827673552                          |
| 49  | 10.985053 | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | 1250 → 63624 [ACK] Seq=107 Ack=12 Win=408256 Len=0 TSval=827675351 TSecr=827675351                              |
| 50  | 10.985715 | 127.0.0.1 | 127.0.0.1   | TCP      | 107    | 1250 → 63624 [PSH, ACK] Seq=107 Ack=12 Win=408256 Len=51 TSval=827675351 TSecr=827675351                        |
| 51  | 10.985785 | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | 63624 → 1250 [ACK] Seq=12 Ack=158 Win=408128 Len=0 TSval=827675351 TSecr=827675351                              |
| 52  | 12.821429 | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | 63624 → 1250 [FIN, ACK] Seq=12 Ack=158 Win=408128 Len=0 TSval=827677186 TSecr=827675351                         |
| 53  | 12.821507 | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | 1250 → 63624 [ACK] Seq=158 Ack=13 Win=408256 Len=0 TSval=827677186 TSecr=827677186                              |
| 54  | 12.825355 | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | 1250 → 63624 [FIN, ACK] Seq=158 Ack=13 Win=408256 Len=0 TSval=827677189 TSecr=827677186                         |
| 55  | 12.825410 | 127.0.0.1 | 127.0.0.1   | TCP      | 56     | 63624 → 1250 [ACK] Seq=13 Ack=159 Win=408128 Len=0 TSval=827677189 TSecr=827677189                              |

Bilde 25: Pakkefangst fra Wireshark av kommunikasjon mellom klient og tjener. Med filter på port 1250.

Over ser vi pakkefangsten fra Wireshark av kommunikasjonen mellom klient og tjener programmene våre. Vi observerte på *loopback* (nettverkskort *lo0*) og filtrerte pakkene på port 1250, da vi vet at det er på denne porten de kommuniserer.

```
Hei, du har kontakt med tjenersiden! Skriv et regne stykke med +/- og jeg gir deg svar. (tall +/- tall)
1 + 3
4
kake
Husk å skriv inn på riktig format: tall +/- tall
```

Bilde 26: Bilde viser TCP-stream i klartekst.

Her ser vi hva tjener (i blått) og klient (i rødt) har skrevet til hverandre. Måtte også sette formatet til UTF-8, da det var dette vi brukte i koden.

## Øvingens del 2: Enkel webtjener (returnere HTTP)

a)

| No. | Time     | Source    | Destination | Protocol | Length | Info                                                                                                                      |
|-----|----------|-----------|-------------|----------|--------|---------------------------------------------------------------------------------------------------------------------------|
| 19  | 7.326786 | :::1      | :::1        | TCP      | 88     | 63701 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=828662920 TSecr=0 SACK_PERM=1                                |
| 20  | 7.326893 | :::1      | :::1        | TCP      | 88     | 80 → 63701 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16324 WS=64 TSval=828662920 TSecr=828662920 SACK_PERM=1             |
| 21  | 7.326902 | :::1      | :::1        | TCP      | 76     | 63701 → 80 [ACK] Seq=1 Ack=1 Win=407744 Len=0 TSval=828662920 TSecr=828662920                                             |
| 22  | 7.326909 | :::1      | :::1        | TCP      | 76     | [TCP Window Update] 80 → 63701 [ACK] Seq=1 Ack=1 Win=407744 Len=0 TSval=828662920 TSecr=828662920                         |
| 23  | 7.379356 | :::1      | :::1        | HTTP     | 427    | GET / HTTP/1.1                                                                                                            |
| 24  | 7.379373 | :::1      | :::1        | TCP      | 76     | 80 → 63701 [ACK] Seq=1 Ack=352 Win=407424 Len=0 TSval=828662970 TSecr=828662970                                           |
| 25  | 7.392267 | :::1      | :::1        | TCP      | 603    | 80 → 63701 [PSH, ACK] Seq=1 Ack=352 Win=407424 Len=527 TSval=828662982 TSecr=828662970 [TCP segment of a reassembled PDU] |
| 26  | 7.392320 | :::1      | :::1        | TCP      | 76     | 63701 → 80 [ACK] Seq=352 Ack=528 Win=407232 Len=0 TSval=828662982 TSecr=828662982                                         |
| 27  | 7.394578 | :::1      | :::1        | HTTP     | 76     | HTTP/1.0 200 OK (text/html)                                                                                               |
| 28  | 7.394626 | :::1      | :::1        | TCP      | 76     | 63701 → 80 [ACK] Seq=352 Ack=529 Win=407232 Len=0 TSval=828662984 TSecr=828662984                                         |
| 29  | 7.394702 | :::1      | :::1        | TCP      | 76     | 63701 → 80 [FIN, ACK] Seq=352 Ack=529 Win=407232 Len=0 TSval=828662984 TSecr=828662984                                    |
| 30  | 7.394781 | :::1      | :::1        | TCP      | 76     | 80 → 63701 [ACK] Seq=529 Ack=353 Win=407424 Len=0 TSval=828662984 TSecr=828662984                                         |
| 31  | 7.429187 | :::1      | :::1        | TCP      | 88     | 63702 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=828663017 TSecr=0 SACK_PERM=1                                |
| 32  | 7.429202 | :::1      | :::1        | TCP      | 64     | 80 → 63702 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0                                                                             |
| 33  | 7.429572 | 127.0.0.1 | 127.0.0.1   | TCP      | 68     | 63703 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=828663017 TSecr=0 SACK_PERM=1                                |
| 34  | 7.429582 | 127.0.0.1 | 127.0.0.1   | TCP      | 44     | 80 → 63703 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0                                                                             |

Bilde 27: Pakkefangst fra Wireshark av kommunikasjon mellom klient (Safari) og webtjener. Med filter på port 80.

Over ser vi alle pakkene som ble utvekslet mellom klienten (Safari) og webtjeneren vår.

```
GET / HTTP/1.1
Host: localhost
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_6) AppleWebKit/605.1.15 (KHTML, like Gecko)
Version/14.0.3 Safari/605.1.15
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: keep-alive

HTTP/1.0 200 OK
Content-Type: text/html; charset=utf-8

<html><body>
<h1>En Hilsen</h1>
GET / HTTP/1.1
Host: localhost
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_6) AppleWebKit/605.1.15 (KHTML, like Gecko)
Version/14.0.3 Safari/605.1.15
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: keep-alive
</body></html>
```

Bilde 28: Bilde viser TCP-stream i klartekst, av pakkene mellom klient (Safari) og webtjener.

I bilde over ser vi først klienten sin melding til webtjener og deretter webtjener sin respons.

Alle ACK-pakkene som sendes uten data er «rene» kvitteringsmeldinger, da deres eneste funksjon er å kvittere at pakker er mottatt. Det blir alltid kvittert i alle pakker med ACK-flagget satt. Det sendes kvitteringsmeldinger til alle pakker ellers vil sender av data anta pakken/pakkene som tapt og sende på nytt etter en viss tid. Derfor hender det at mottaker må sende «rene» kvitteringsmeldinger og ikke bake disse inn i andre pakker (for eksempel å kvittere når den sender data selv).

b)

```

 printWriter.println(createHTML(messages));
 }
 catch (Exception e)
 {
 e.printStackTrace();
 }
 finally
 {
 printWriter.close();
 bufferedReader.close();
 inputStreamReader.close();
 }

```

Bilde 29: Bilde viser kode for nedkobling av kommunikasjon.

Vi ser at etter vi sender svar til klienten, med *PrintWriter*, så går vi videre til 'finally'-blokken hvor vi kobler ned kommunikasjonen. Dette er nøyaktig samme som vi har sett tidligere med den enkle kalkulatoren.

TCP	76	63701 → 80	[FIN, ACK]	Seq=352	Ack=529	Win=407232	Len=0	TSval=828662984	TSecr=828662984
TCP	76	80 → 63701	[ACK]	Seq=529	Ack=353	Win=407424	Len=0	TSval=828662984	TSecr=828662984

Bilde 30: Pakker involvert i nedkobling av kommunikasjon.

Som sagt tidligere vil *Socket* koblingen mellom klient og tjener automatisk lukkes når koden forlater *try*-blokkens *scope*. Når dette skjer vil da [FIN, ACK]-pakken bli sendt fra tjener til klienten, og klienten svarer med en ACK-pakke og med det er, i dette tilfellet, kommunikasjonen ferdig nedkoblet. Dette stemmer ikke overens med teorien, men dette kan komme av at webtjeneren vår lukker kommunikasjonen og stenger porten, og at det kan resultere i at FIN-pakken fra klient ikke når frem til nettverkskortet vi observerer på (med Wireshark). Det kan hende vi tvinger tjeneren ut av «FIN\_WAIT\_2» og til «CLOSED» ved å stoppe hele programmet.

Teoretisk sett vil nedkoblingsfasen gå som beskrevet under, men her tar vi ikke for oss alle de mange scenarioene som kan forekomme (f.eks hvis begge parter sender FIN-pakker samtidig e.l.), vi ser på en typisk nedkobling mellom klient og tjener. Part A vil sende en FIN-pakke

og gå stadiet «FIN\_WAIT\_1». Part B mottar da en FIN-pakke og går i stadiet «CLOSE\_WAIT» og sender en ACK-pakke på FIN-pakken. Part A mottar ACK-pakken og går i «FIN\_WAIT\_2». Nå venter part A på at part B skal sende sin FIN-pakke, som er nøyaktig det part B gjør, og deretter går i «LAST\_ACK» hvor den venter på en ACK-pakke fra part A. Part A mottar FIN-pakken til part B og sender en ACK-pakke og går i «TIME\_WAIT». Part B mottar ACK-pakken fra A som bekrefter FIN-pakken dens, og går da i «CLOSED», B er nå ferdig nedkoblet. A derimot må vente i «TIME\_WAIT» i en gitt tid (typisk noen minutter, men dette varierer fra implementasjon til implementasjon). Om A ikke får noen flere pakker fra B i løpet av den gitte tiden, vil også A gå til «CLOSED» og da være ferdig nedkoblet. Det er jo et scenario at ACK-pakken til A (som kvitterer FIN-pakken til B) ikke kommer frem til B, da vil B prøve å sende FIN-pakken på nytt og A mottar da denne og vil sende ACK-pakken på nytt og resette tiden i «TIME\_WAIT». (Kurose & Ross, 2017) (McKinney, 2021)

Kort sagt vil man typisk se (f.eks i Wireshark) at man har pakke «møsteret» FIN-ACK, ACK, FIN-ACK, ACK (se Figure 2).

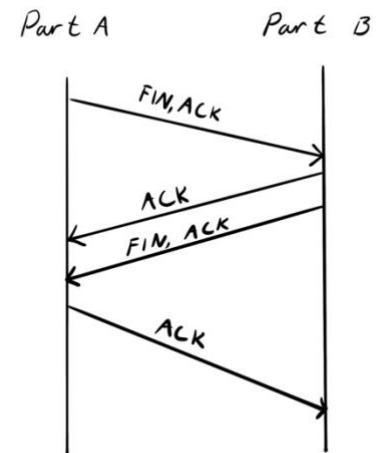


Figure 2: Illustrasjon av nedkoblingsfasen til TCP-protokollen.

## Oppgave 6

Oppgave ‘a’ er besvarelsen vår før den reviderte versjon, den skal dekke alt den reviderte spør etter også.

a)

```
datagramPacket = new DatagramPacket(buffer, buffer.length, InetAddress.getByName(host), port: 1234);
datagramSocket.send(datagramPacket); //Sender en pakke
```

Bilde 31: Kode for å opprette en DatagramPacket og sende den som en UDP-pakke til tjener fra klient.

```
1 0.000000 127.0.0.1 127.0.0.1 UDP 37 60975 → 1234 Len=5
```

Bilde 32: Skjermdump fra Wireshark av UDP-pakke fra klient til tjener.

I bildene over ser vi at en pakke til port 1234 (*DatagramPacket*) blir laget og deretter sendt gjennom *DatagramSocket*, da som UDP-pakke. Vi ser pakken igjen i Wireshark.

```
this. datagramSocket = new DatagramSocket(port: 1234);
```

Bilde 33: Kode som oppretter en DatagramSocket på port 1234.

```
this._datagramSocket.send(new DatagramPacket(buffer,
 buffer.length, datagramPacket.getAddress(),
 datagramPacket.getPort())); //Sender svar til
```

Bilde 34: Kode for å sende en DatagramPacket gjennom DatagramSocket, fra tjener til den som sendte mottatt packet.



```
DatagramPacket datagramPacket = new DatagramPacket(buffer, buffer.length);
this._datagramSocket.receive(datagramPacket); //Venter på en pakke
```

Bilde 35: Kode som lager en *DatagramPacket* og venter på å motta data gjennom *DatagramSocket*, for så å legge det i *DatagramPacket*.

I bildene over ser vi tjenersiden. Tjeneren oppretter en *DatagramSocket* på en port (port 1234 i dette tilfelle), og lager en *DatagramPacket* som den bruker til å legge mottatt *DatagramPacket* i. Den venter på at en klient skal sende den noe med *recive()*-metoden til *DatagramSocket*. Til slutt sender tjeneren et svar tilbake til den som sendte pakken tjeneren mottok, adressen til den klienten finner tjeneren i *DatagramPacket*'en den mottok. Vi ser i *Wireshark* at denne pakken har blitt sendt tilbake til klienten.

Vi kan også se på UDP-stream'en som klartekst og se hva klient og tjener har sendt til hverandre.

1 + 34

Bilde 36: Skjermdump av *Wireshark*, UDP-stream i form av klartekst.

2	0.000559	127.0.0.1	127.0.0.1	UDP	33	1234 → 60975	Len=1
---	----------	-----------	-----------	-----	----	--------------	-------

Bilde 37: Skjermdump av *Wireshark* av UDP-pakke fra tjener til klient.

Forskjellen mellom øving 1 og 2 er at vi brukte TCP i øving 1 og UDP i øving 2. I øving 1 måtte det mer «forsiktig» programmering til da TCP-protokollen er strengere og gir flere feilmeldinger. Eksempler på dette er at TCP krever kvitteringer for alle pakker som blir sendt, for å være en pålitelig dataoverføring. Dette kan da selvfølgelig gi en del feil, om vi ikke får opprettet kontakt med tjener, eller tjener ikke svarer. Alle feil skal også tolkes og formidles til bruker om hva som har skjedd. Ved øving 2 så var det UDP-protokollen vi måtte forholde oss til. UDP er upålitelig dataoverføring og krever derav ikke noen kvitteringer på pakker, ingen oppkobling eller nedkobling, man bare sender pakker ditt man mener de skal og håper på å få svar. Det er da ikke like mye feilmeldinger som ved TCP, da det er færre restriksjoner. Rent programmerings messig var det mer utfordrerne å sette opp UDP da tjener ikke vet hvor mye data en klient vil prøve å sende og derfor ikke vet hvor mye plass den skal sette av til en innkommende melding, dette kan naturligvis gjøres dynamisk, men det må da gjøres. Med TCP kunne man bare sende tilsynelatende en stor pakke, og oppdelingen av pakken ble gjort automatisk.

Fordeler ved valg av TCP er ganske klart at det er en pålitelig dataoverføring og om ønsker er at all data kommer frem så er TCP et godt valg. TCP har som konsekvens av påliteligheten sin en ganske stor overhead (hvert fall sammenliknet med UDP) som gjør at pakkene bruker lengere tid. Dette fordi om en part ikke får kvitteringer på pakkene sender den de på nytt og den bruker også ressurser på å holde styr på hva som er kvittert og ikke. En følge av dette er at det sendes ikke nye pakker og det sendes færre da TCP-protokollen tar hensyn til belastningen på nettet og reduserer utsendingen sin om det mye trafikk. TCP vil ved pakketap (ukvitterte pakker) være veldig treg, dette stammer fra TCP sin metningskontroll protokoll, mer spesifikt «tilbake-til-start» og «sakte start» fasene. TCP starter alltid med én pakke og



øker antall pakker den har ukvittert eksponentielt (fram til en terskel, deretter er det lineær vekst), men ved et pakketap, vil den starte på nytt med å kun sende én pakke og bygge seg opp igjen. Dette fungerer veldig bra for å ikke overbelaste nettverk og fordele kapasitet blant brukere, men det vil være ganske ubrukelig for f.eks. telefonsamtaler, live streamer, e.l.

Svakhetene til TCP er nettopp styrkene til UDP. UDP har ingen metningskontroll, den bare sender så mange pakker den får til og tar imot så mye den får til. Så UDP egner seg ikke for Svakheterne til TCP er nettopp styrkene til UDP. UDP har ingen metningskontroll den bare sender så mange pakker den klarer, og tar imot så mye den klarer. UDP egner seg ikke for overføring av data som må være korrekt, men heller data som må gå kjapt. UDP har pga. sine færre restriksjoner også en mindre header. Det stilles heller ingen krav til størrelsen på nyttelasten, men dersom pakken er for stor for mottaker kan den bli forkastet. Dette er et sjeldent problem, da UDP-pakker vanligvis er korte. UDP har sin beste bruk da nettopp da ved f.eks. telefonsamtaler og live streamer. Nedlastning av programvare vil ikke være gunstig for UDP, men passer veldig bra for TCP.

## Øvingens Del 2: Etablere sikker kommunikasjon med TLS (Revisjon TLS)

### Fra programmeringsøvingen

- a) Besvarelsen som følger, er hovedsakelig hentet fra dokumentasjonen vår av programmeringsøving 2.

Vi opprettet `JavaSSLServer.java` og `JavaSSLClient.java` (Ukjent, 2021). Deretter opprettet vi en jar-fil til hver av disse med følgende steg:

```
olineamundsen@Olines-MBP server % javac ../../src/JavaSSLServer.java -d .
```

Bilde 38: Lager '.class'-fil.

Lager .class-fil av java-fila.

```
1 Main-Class: JavaSSLServer
2
```

Bilde 39: Innhold i 'MANIFEST.MF'-filen.

Lager MANIFEST.MF-fil, og linker main-klassen og legger den i samme mappe som class-fila.

```
olineamundsen@Olines-MBP server % jar cvmf MANIFEST.MF JavaSSLServer.jar *
added manifest
adding: JavaSSLServer.class(in = 1954) (out= 1060)(deflated 45%)
adding: MANIFEST.MF(in = 26) (out= 28)(deflated -7%)
```

Bilde 40: Lager en 'JAR'-fil og kobler den med 'MANIFEST.MF'-filen.

Lager jar-fil i samme mappe. Gjør dette for både server og client.

```

olineamundsen@Olines-MBP øving 2 % mkdir mykeystore
olineamundsen@Olines-MBP øving 2 % cd mykeystore
olineamundsen@Olines-MBP mykeystore % /usr/bin/keytool -genkey -alias signFiles
-keystore examplestore -keyalg RSA
Enter keystore password:
Re-enter new password:
What is your first and last name?
[Unknown]: k
What is the name of your organizational unit?
[Unknown]: k
What is the name of your organization?
[Unknown]: k
What is the name of your City or Locality?
[Unknown]: k
What is the name of your State or Province?
[Unknown]: k
What is the two-letter country code for this unit?
[Unknown]: k
Is CN=k, OU=k, O=k, L=k, ST=k, C=k correct?
[no]: y

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) wi
th a validity of 90 days
for: CN=k, OU=k, O=k, L=k, ST=k, C=k

```

Bilde 41: Setter opp en 'keystore'-mappe, og spesifiserer at vi skal bruke RSA-kryptering.

Oppretter en keystore i en egen mappe i prosjektet. Det er her privat nøkkel lagres.

```

olineamundsen@Olines-MBP øving 2 % java -jar -Djavax.net.ssl.keyStore=ex
amplestore -Djavax.net.ssl.keyStorePassword=123456 "bin/server/JavaSSLSe
rver.jar"
SSL ServerSocket started
[SSL: ServerSocket[addr=0.0.0.0/0.0.0.0,localport=8000]]

```

Bilde 42: Kjører server (tjener) sin 'JAR'-fil og linker den med 'keystore'-mappen.

Kjører deretter komandoen:

```
java -jar -Djavax.net.ssl.keyStore=examplestore -Djavax.net.ssl.keyStorePassword=123456
"bin/server/JavaSSLServer.jar"
```

Som kjører jar-fila og bruker keystore. Starter da server og deretter client, og har derav kryptert meldinger gjennom TLS/SSL.

```

olineamundsen@Olines-MBP øving 2 % java -jar -Djavax.net.ssl.trustStore=examples
tore -Djavax.net.ssl.trustStorePassword=123456 "bin/client/JavaSSLClient.jar"
Enter something:
1 + 3
1 + 3
Enter something:
dette funker
dette funker
Enter something:
q

```

Bilde 43: Terminal fra klienten sin side.

```

olineamundsen@Olines-MBP øving 2 % java -jar -Djavax.net.ssl.keyStore=examplestore -Djavax.net.ssl.keyStorePassword=123456 "bin/server/JavaSSLServer.jar"
SSL ServerSocket started
[SSL: ServerSocket[addr=0.0.0.0/0.0.0.0,localport=8000]]
ServerSocket accepted
1 + 3
dette fungerer
Closed

```

Bilde 44: Terminal fra server(tjener) sin side.

Vi kjørte Wireshark, og observerte på loopback under kommunikasjonen. Vi observerte at innholdet var kryptert med TLS.

```

> Frame 13: 126 bytes on wire (1008 bits), 126 bytes captured (1008 bits) on interface lo0, id 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 8000, Dst Port: 49524, Seq: 134, Ack: 392, Len: 70
> Transport Layer Security
 TLSv1.3 Record Layer: Application Data Protocol: Application Data
 Opaque Type: Application Data (23)
 Version: TLS 1.2 (0x0303)
 Length: 65
 Encrypted Application Data: Sec261522476e809b262131e7f28d7b45ffa11cebbf0093bac5ae6a75ead3f194c172fe6...

```

Bilde 45: Viser i Wireshark at dataen som sendes mellom klient og server(tjener) er kryptert.

Ser at port 8000 er i bruk som vi spesifikt satte opp i koden, og at dataen er kryptert.

- b) TLS 1.3 opprettes ved en «Client Hello»-pakke fra klienten og en «Server Hello»-pakke fra tjeneren, i disse pakkene utveksler de den nødvendige informasjonen for å sette opp krypteringen. Vi ser at klienten tilbyr tjeneren 49 forskjellige siffersuiter å velge blant. Denne informasjonen ligger i TLSv1.3-protokollen i «Client Hello»-pakken fra klient til tjener.

3	17.755897	127.0.0.1	127.0.0.1	TCP	68	49524 → 8000 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TS=0
4	17.755994	127.0.0.1	127.0.0.1	TCP	68	8000 → 49524 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344
5	17.756001	127.0.0.1	127.0.0.1	TCP	56	49524 → 8000 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=80089
6	17.756006	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 8000 → 49524 [ACK] Seq=1 Ack=1 Win=4082
7	22.434600	127.0.0.1	127.0.0.1	TLSv1.3	447	Client Hello
8	22.434648	127.0.0.1	127.0.0.1	TCP	56	8000 → 49524 [ACK] Seq=1 Ack=392 Win=407872 Len=0 TSval=800
9	22.464546	127.0.0.1	127.0.0.1	TLSv1.3	183	Server Hello
10	22.464583	127.0.0.1	127.0.0.1	TCP	56	49524 → 8000 [ACK] Seq=392 Ack=128 Win=408128 Len=0 TSval=8
11	22.475373	127.0.0.1	127.0.0.1	TLSv1.3	62	Change Cipher Spec
12	22.475400	127.0.0.1	127.0.0.1	TCP	56	49524 → 8000 [ACK] Seq=392 Ack=134 Win=408128 Len=0 TSval=8
13	22.477923	127.0.0.1	127.0.0.1	TLSv1.3	126	Application Data

Bilde 46: Skjermdump fra Wireshark av pakker mellom klient og tjener, som er TLS (1.3) kryptert.

Vi ser at tjeneren velger siffersuiten  
‘TLS\_AES\_256\_GCM\_SHA384’. Denne informasjonen  
ligger i TLSv1.3-protokollen i ‘*Server Hello*’-pakken fra  
tjener til klient, som er svaret på ‘*Client Hello*’-pakken  
fra klienten.

Vi kan bryte ned hva siffersuiten betyr:

TLS – protokollen som brukes

AES – krypterings algoritmen

256 – størrelsen på nøkkelen

GCM – modusen til nøkkelen

SHA384 – ‘MAC’-en brukt av algoritmen (Helme, 2021).

```

Handshake Protocol: Server Hello
Handshake Type: Server Hello (2)
Length: 118
Version: TLS 1.2 (0x0303)
Random: e1a87a3bcfa854e82267825004eece22c09446
Session ID Length: 32
Session ID: 10aea86877800292de939c0d72830f41c4
Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
Compression Method: null (0)

```

Bilde 47: Skjermdump fra Wireshark av hvilken siffersuite tjener velger.

```

Cipher Suites Length: 98
Cipher Suites (49 suites)
Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xc0a9)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xc0a8)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
Cipher Suite: TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0xc00f)
Cipher Suite: TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xc0aa)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_GCM_SHA384 (0x00a3)
Cipher Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0xc00e)
Cipher Suite: TLS_DHE_DSS_WITH_AES_128_GCM_SHA256 (0x00a2)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc023)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)
Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 (0xc006b)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA256 (0xc006a)
Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA256 (0xc0067)
Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA256 (0xc0040)
Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02e)
Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384 (0xc032)
Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02d)
Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256 (0xc031)
Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384 (0xc026)
Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384 (0xc02a)
Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256 (0xc025)
Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256 (0xc029)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0xc039)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0xc038)
Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0xc033)
Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0xc032)
Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA (0xc005)
Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA (0xc00f)
Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA (0xc004)
Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA (0xc00e)
Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0xc009d)
Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0xc009c)
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA256 (0xc03d)
Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA256 (0xc03c)
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0xc035)
Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0xc02f)
Cipher Suite: TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0xc0ff)

```

Bilde 48: Skjermdump fra Wireshark av alle 'siffersuitene' som klienten tilbyr tjeneren å velge blant (TLSv1.3).

## Fra Blackboard kommunikasjon

### c) Steg 1 oppkobling: Client Hello

På bilde til høyre ser vi at klienten tilbyr  
tjeneren 27 forskjellige algoritmer for  
krypteringen.

```

Cipher Suites Length: 54
Cipher Suites (27 suites)
Cipher Suite: Reserved (GREASE) (0x5a5a)
Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xc0a9)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xc0a8)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc023)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0xc009d)
Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0xc009c)
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA256 (0xc03d)
Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA256 (0xc03c)
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0xc035)
Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0xc02f)
Cipher Suite: TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc008)
Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0xc00a)

```

Bilde 49: Skjermdump fra Wireshark av alle 'siffersuitene' som klienten tilbyr tjeneren (TLSv1.2)

## d) Steg 2A: Server Hello, siffersuite

Under ser vi at tjeneren valgte siffersuiten

«TLS\_ECDHE\_RSA\_WITH\_AES\_128\_GCM\_SHA256». Dette skal følge teorien være den sikreste algoritmen som både tjeneren og klienten har tilgjengelig.

Vi kan bryte ned hva siffersuiten betyr:

TLS – Protokollen som brukes

ECDHE – Nøkkelbytte mekanismen

RSA – Algoritmen for autentiseringsnøkkelen

WITH – Er bare bindeord i denne kontekst også

AES – Den symmetriske krypteringsalgoritmen

128 – Størrelsen på nøkkelen

GCM – Modusen til nøkkelen

SHA256 – ‘MAC’-en brukt av algoritmen (Helme, 2021)

```

v Handshake Protocol: Server Hello
 Handshake Type: Server Hello (2)
 Length: 85
 Version: TLS 1.2 (0x0303)
 > Random: 0816b6bb0e12177a78854f1e40ac7d38aca8610c939569a8b79e94f3aa1e587d
 Session ID Length: 32
 Session ID: 590a531c9aaa03af725fbabb64c29ca2d7155eab2ee3981bc5545df7b1360646
 Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)

```

Bilde 50: Skjermdump fra Wireshark av hvilken siffersuite tjener velger (TLSv1.2).

## e) Steg 2B: Server Hello, sertifikat

Som vi kan se på bildene under er det fire sertifikater i «Certificate» feltet i pakken fra tjeneren. Det ene bilde viser alle fire, og det andre viser detaljene i en av dem, verdiene for alle «Issuer» og «Subject» ligger ser vi i tabellen under. Som vi ser så er «Issuer» i en rad lik «Subject» i raden under. Så vi kan da si at endepunktet for «Issuer» er «Starfield Class 2 Certification Au», som er da den som «gir» sertifikatet videre til det når endepunktet til «Subject», som er «\*.blackboard.com». Dette gir mening, det er denne siden vi vil bruke.

Nr.	Issuer/common name	Subject/common name
1	Amazon	*.blackboard.com
2	Amazon Root CA 1	Amazon
3	Starfield Services Root Certificate Authority	Amazon Root CA 1
4	Starfield Class 2 Certification Au	Starfield Services Root Certificate Authority

```

v Certificates (4822 bytes)
 Certificate Length: 1390
 > Certificate: 3082056a30820452a00302010202100e795c100a919055924c82d4500f4775300d06092a.. (id-at-commonName=*.blackboard.com)
 Certificate Length: 1101
 > Certificate: 3082044930820331a0030201020213067f94578587e8ac77deb25325bbc998b560d300d.. (id-at-commonName=Amazon,id-at-organizationalUnitName=Server CA 1B,id-at-organizationName=Amazon,id-at-countryName=US)
 Certificate Length: 1174
 > Certificate: 308204923082037aa0030201020213067f944a2a27cdf3fac2ae2b01f908eeb9c4c6300d.. (id-at-commonName=Amazon Root CA 1,id-at-organizationName=Amazon,id-at-countryName=US)
 Certificate Length: 1145
 > Certificate: 308204753082035da003020102020900a70e4a4c3482b77f300d06092a864886f70d0101.. (id-at-commonName=Starfield Services Root Certificate Authority ,id-at-organizationName=Starfield Technologies, Inc.,id-at-localityName=Scottsdale,id-

```

Bilde 51: Skjermdump fra Wireshark som viser fire sertifikat for TLS 1.2, i pakke fra tjener.



```

 Certificate: 3082056a30820452a00302010202100e795c100a919055924c82d4500f4775300d06092a... (id-at-commonName=*.blackboard.com)
 signedCertificate
 version: v3 (2)
 serialNumber: 0x0e795c100a919055924c82d4500f4775
 > signature (sha256WithRSAEncryption)
 issuer: rdnSequence (0)
 > rdnSequence: 4 items (id-at-commonName=Amazon,id-at-organizationalUnitName=Server CA 1B,id-at-organizationName=Amazon,id-at-countryName=US)
 > validity
 subject: rdnSequence (0)
 > rdnSequence: 1 item (id-at-commonName=*.blackboard.com)
 > subjectPublicKeyInfo
 > extensions: 10 items
 algorithmIdentifier (sha256WithRSAEncryption)
 Padding: 0
 encrypted: b13b00475bd5cb01907398d3a2f57256bc821059408e8411e1a769942c46047b7c380c83...

```

Bilde 52: Skjermdump fra Wireshark av en av sertifikatene i TLS 1.2 pakken fra tjener.

## f) Steg 3: Nøkkelutveksling

Under ser vi på bildene hva DH parameterne i «Server Key Exchange» er gitt.

```

 Handshake Protocol: Server Key Exchange
 Handshake Type: Server Key Exchange (12)
 Length: 329
 EC Diffie-Hellman Server Params
 Curve Type: named_curve (0x03)
 Named Curve: secp256r1 (0x0017)
 Pubkey Length: 65
 Pubkey: 0425abb5f74216ed2ad8042158991331823d6b3cc08a7f9575bee9d39e8836c6f118d538...
 Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
 Signature Hash Algorithm Hash: SHA512 (6)
 Signature Hash Algorithm Signature: RSA (1)
 Signature Length: 256
 Signature: 08e08293a588537bad75c3b4160c3763de3b439c08606a1050e445261c2eaeaaa92ae0b4...

```

Bilde 53: Skjermdump fra Wireshark av "Server Key Exchange" og DH parameterne.

```

 Handshake Protocol: Client Key Exchange
 Handshake Type: Client Key Exchange (16)
 Length: 66
 EC Diffie-Hellman Client Params
 Pubkey Length: 65
 Pubkey: 046a72588ddee6361179588b6ab61226fc1012e9667d91cdb6542af3d59acf114c6b8998...

```

Bilde 54: Skjermdump fra Wireshark av "Client Key Exchange" og dens DH paramenterne.



## Sitat kilder

Hallsteinsen, Ø., Klefstad, B., & Skundberg, O. (2020). *Innføring i datakommunikasjon*, 2.

*Utgave*. Stiftelsen TISIP og Gyldendal Norsk Forlag 2008.

Kurose, J., & Ross, K. (2017). *Computer Networking A Top-Down Approach, Seventh Edition*. Pearson.

Ukjent. (2021, Feb 1). *Java example of SSL Server and Client, and how to generate keystore*.

Hentet fra java-buddy.blogspot.com: <http://java-buddy.blogspot.com/2016/07/java-example-of-ssl-server-and-client.html>

Helme, S. (2021, Feb 15). *HTTPS Cheat Sheet*. Hentet fra <https://scotthelme.co.uk/https-cheat-sheet/>

McKinney, G. (2021, Feb 20). *TCP/IP State Transition Diagram (RFC793)*. Hentet fra [users.cs.northwestern.edu](https://users.cs.northwestern.edu/~agupta/cs340/project2/TCPIP_State_Transition_Diagram.pdf):

[https://users.cs.northwestern.edu/~agupta/cs340/project2/TCPIP\\_State\\_Transition\\_Diagram.pdf](https://users.cs.northwestern.edu/~agupta/cs340/project2/TCPIP_State_Transition_Diagram.pdf)