Datakommunikasjon – Øving 2
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*).

Oppgave 1 – Adressering

a) Lag 2 og 3 er henholdsvis lenkelaget og nettverkslaget. «Nettverkslagets oppgave er å overføre datapakker mellom maskinene hvor applikasjonene kjører» (Hallsteinsen, Klefstad, & Skundberg, 2020). *Internet Protocol* (*IP*) er en typisk protokoll brukt av nettverkslaget. I IP-protokollen står det at IP-headeren på en IP-pakke skal ha et felt for avsenders IP-adresse og et for mottakers IP-adresse (32-bit hver i IPv4 og 128-bit i IPv6). Basert på IP-adressene i figuren ser man at det blir benyttet IPv4. Hvis vi tar for oss nettverkslagets adressering for pakken som går fra «webtjener» til «PC» så vil IP-pakken som blir lagd i «webtjener» ha «PC» sin IP-adresse som mottakers IP-adresse. Dette da nettverkslaget ikke «ser» alle punktene pakken fysisk må gå fra «webtjener» til «PC», men kun konsentrerer seg om å frakte pakken fra A til B, fra «webtjener» til «PC», dette lar den laget under finne ut av, nemlig lenkelaget. Så IP-pakken (pakken sett fra nettverkslaget) vil alltid ha mottaker adresse lik IP-adressen til «PC»: «155.155.1.55». Om ruterne hadde vært NAT-konfigurerte så hadde dette blitt annerledes da de hadde byttet IP-adressene i IP-pakken, men vi ser ikke noen grunn til at de er det. Hadde «PC» hatt en reservert privat adresse (10.0.0.0/8) kunne vi argumentert for at IP-adressen dens måtte blitt gjort om i ruteren (med NAT) for å at den skulle «funnet veien tilbake».

   «Ansvaret til lenkelaget er å overføre pakker mellom nettverkslagene på tilstøtende noder. Dette innebærer å klargjøre pakker, gjennom innramming og andre mekanismer, slik at innholdet kan overføres på det fysiske laget.» (Hallsteinsen, Klefstad, & Skundberg, 2020). Det finnes mange protokoller å velge mellom på lenkelaget (f.eks. Ethernet, LAN, ISDN, ol.), men alle bruker fysiske adresser, MAC-adresser, i innrammingen sin. Når nettverkslaget lager IP-pakken sin og sender den videre, så kommer den til lenkelaget og blir her innrammet til f.eks. en Ethernet-ramme. Dette er tilsvarende det nettverkslaget gjør når det mottar f.eks. et TCP-segment fra transportlaget, som da blir pakket inn til en IP-pakke.
   Når lenkelaget lager en ramme så må den knytte mottakers IP-adressen (hentet fra IP-pakken) til en fysisk MAC-adresse som skal ligge i rammen. MAC-adressen finnes ved hjelp av *Address Resolution Protocol (ARP)*. Om IP-adressen som blir spurt etter ikke befinner seg i samme subnett som den som spør (sender ARP, webtjener her) så må pakken først overføres til *Default Gateway* (*DGW*). Avsender (webtjener) bruker da DGW sin MAC-adresse i rammen den sender. I vårt tilfelle vil «webtjener» ikke få

noe svar på sin ARP-forespørsel, da «webtjener» og «PC» er i forskjellige subnett, dette kommer vi tilbake til. «webtjener» finner da DGW som er «ruter 1», «ruter 1» får rammen og ser at den er til seg selv. Den henter ut IP-pakken i rammen og sjekker om denne også er til seg selv, det er den ikke. Da vil den slå opp i rutingtabellen sin for å finne ut av hvilken «retning» den skal videresende pakken. Dette ser vi at i nettverket blir til «ruter 2». «ruter 2» gjør det samme og sender pakken til «svitsj 2» som sørger for at den blir sendt til «PC». «ruter 1» bruker naturlig nok MAC-adressen til «ruter 2» (som den fant i tabellen sin). «ruter 2» er ikke i samme subnett som «svitsj 2», men svitsjer har egentlig ikke adresser da de kun redirigerer rammer i riktig retning, men om den har det, som i figuren i oppgaveteksten vil den bli ansett som DGW inn i riktig subnett og «ruter 2» vil sende rammen dit (da bruke «svitsj 2» sin MAC-adresse). Om vi forholder oss til at svitsjer ikke har adresser med kun mottar rammer og sender de i riktig retning vil «ruter 2» sende rammen adressert til

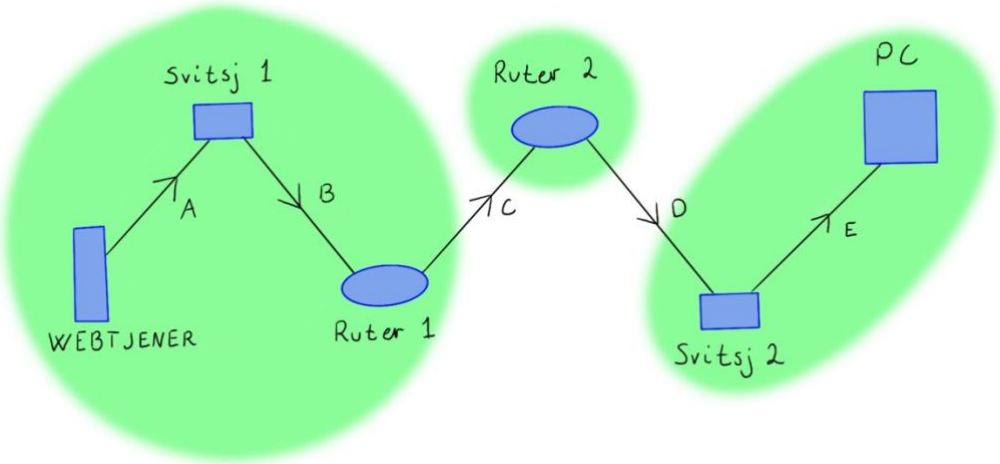


Figure 1: Illustrasjon av subnettene i nettverket.

«PC» sin MAC-adresse, ellers vil «svitsj 2» motta rammen fra «ruter 2» og se at IP-adressen ikke er sin egen, sende en ARP og får da svar av «PC» (fordi de er i samme subnett), og pakken vil bli innrammet og sendt med MAC-adressen til «PC» til «PC».

Vi har tre subnett i figuren subnett A (webtjener, svitsj 1, ruter 1), subnett B (ruter 2) og subnett C (svitsj 2 og PC). Vi anser disse som subnett da et subnett skal bestå av noder med samme nettadresse og som er koblet til internett gjennom samme ruter (DGW). Om vi ser på IP-adressene til de forskjellige nodene i figuren ser vi at de nodene vi har plassert i samme subnett har relativt like IP-adresser og vil da (med rett nettmaske, se oppgave b) ha samme nettadresse, og være forskjellige noder i samme subnett.

Delstrekk / Adresse	A Til Svitsj 1	B Til Ruter 1	C Til Ruter 2	D Til Svitsj 2	E Til PC
Mottaker IP	155.155.1.55	155.155.1.55	155.155.1.55	155.155.1.55	155.155.1.55
Mottaker MAC	22:BB	22:BB	33:CC	55:EE	44:DD

Table 1: Tabell over hvor pakken/rammen er adressert ved de forskjellige punktene i oppgavens figur. Antar her, som nevnt, at "svitsj 2" blir adressert.

b)

Noder	Nettverksadresser
Svitsj 1 & Ruter 1	166.166.1.0/24
Ruter 2	177.177.1.0/30
Svitsj 2 & PC	155.155.1.0/26

Table 2: Tabell over nettverksadressene som er i bruk i figuren.

Den første nettadressen er 166.166.1.0 da «ruter 1» hadde én høyere enn dette, som er standarden for DGW sin adresse. Nettmasken må være stor nok til at vi får med oss resten av nodeadressene i subnettet også, her er det «svitsj 1» med IP-adresse 166.166.1.166. Med litt testing av nettmasker så finner man at en maske på «/24» gir oss maks host adresse lik 166.166.1.254, som vi ser er «svitsj 1» sin adresse mellom nettadressen og øvre grense. Hadde nettmasken være «/25» ville maks host vært 166.166.1.126 som ikke inkluderer «svitsj 1» sin adresse. Derav er «/24» største nettmaske som kan velges.

Den andre nettadressen brukte vi samme logikk for, som ved den første. Vi trenger her et adresserom som har en nodeadresse til ruteren (standard er å velge den minste i adresserommet som ikke er reservert), vi har alltid to reserverte adresser (nettadressen og kringkastingsadressen). Totalt har dette subnettet behov for tre nodeadresser og vi finner da fort at nettmasken kan maksimalt være «/30» som gir oss fire nodeadresser.

Den tredje og siste nettadressen ser vi at begge nodene begynner på 155.155.1 som vi kan ta som utgangspunkt i og lage nettadressen 155.155.1.0/24, men denne har et mye større adresserom enn det vi trenger (155.155.1.1 – 155.155.1.254). Derfor kan vi gjøre nettmasken størst mulig og bruke «/26» som gir oss maks host lik 155.155.1.62 som da inneholder «PC» sin IP-adresse.

Oppgave 2 – MAC-adresser og ARP

a) Hensikten med ARP er å kunne finne ut hvilken MAC-adresse som er knyttet til en IP-adresse. IP-adresser er logiske adresser og lenkelaget opererer med fysiske adresser, nemlig MAC-adresser. Det vil si at når nettverkslaget sender en IP-pakke til lenkelaget så må lenkelaget finne ut hvilken MAC-adresse den skal sende pakken videre til, basert på IP-adressen. Lenkelaget har en todelt struktur *Logical Link Control (LLC)* og *Media Access Control (MAC)*. LLC er et grensesnitt mot nettverkslaget som ikke er teknologispesifikt, dette utfører oppgaver som retransmisjoner ved tapte pakker. Deretter kobles LLC videre til et MAC-

grensesnittet, her er det spesifikke variasjoner basert på de ulike teknologiene. Det er forskjellige MAC-grensesnitt mot teknologier som Ethernet, Fiber, Wi-Fi, ol. alle disse teknologiene ligger på det fysiske laget.
Når en maskin A i et nettverk skal sende en IP-pakke til en annen maskin B i samme nettverk (de er to noder i samme subnett), er A nødt til å finne ut hva MAC-adressen til B er. A har IP-adressen til B (fra IP-pakken), men lenkelaget opererer med MAC-adresser, og denne blir funnet med ARP. ARP starter med at A sender en kringkastingsforespørsel ut i nettverket, dette gjøres på MAC-adressen «FF:FF:FF:FF:FF:FF». A spør «hvem har denne IP-adressen?» (B sin IP-adresse), denne forespørselen vil bli fanget opp av alle nettverkskort i samme subnett. Det er kun maskinen som har den forespurte IP-adressen som svarer. I vårt tilfelle vil det være B som svarer, og svaret inneholder B sin MAC-adresse. Etter at A har knyttet IP-adressen i IP-pakken til en MAC-adresse vil den være klar til å sende IP-pakken til B.
Om A og B ikke er i samme subnett, dette sjekker A ved å se om de deler nettadresse, må IP-pakken sendes gjennom DGW (standardruteren). A bruker ARP til å finne MAC-adressen til DGW. A sender da IP-pakken ut i nettverket, adressert til DGW sin MAC-adresse.

b) Som nevnt i oppgave a, så blir ARP-forespørsler sendt på kringkastingsadressen «FF:FF:FF:FF:FF:FF». Det vil si at alle nettverkskort i samme subnett vil fange opp pakken og behandle den videre. Mottakere er da kjennetegnet ved at de er i samme subnett, altså at de deler samme nettadresse som avsender.

c) Under ser vi skjermklipp av kommandoen *«arp -a»*. Dette er innholdet i ARP-tabellen på vår maskin. Vi har mange nettverkskort på maskinen. De 24 første bit i MAC-adresser er leverandørspesifikke, og kalles *Organizationally Unique Identifier (OUI),* og de 24 resterende bit er enhetsspesifikke. Derfor kan vi finne produsenten av nettverkskortet basert på de 24 første. Vi kan finne produsenten ved å slå opp i IEEE RA sine register over OUI'er. Vi fant da produsentene av nettverkskortene ved å søke etter OUI'ene i dette registeret. Produsentene til de forskjellige nettverkskortene vises i *table 3* under.

```
olakristofferhoff@Olas-MacBook-Pro ~ % arp -a
? (192.168.0.1) at b8:d5:26:dd:14:d0 on en0 ifscope [ethernet]
? (192.168.0.1) at b8:d5:26:dd:14:d0 on en7 ifscope [ethernet]
? (192.168.0.101) at d6:73:8d:17:ee:15 on en0 ifscope [ethernet]
? (192.168.0.101) at d6:73:8d:17:ee:15 on en7 ifscope [ethernet]
? (192.168.0.112) at 7a:44:f6:51:8f:c on en0 ifscope [ethernet]
? (192.168.0.112) at 7a:44:f6:51:8f:c on en7 ifscope [ethernet]
? (192.168.0.119) at 2:61:7e:fd:37:c3 on en0 ifscope [ethernet]
? (192.168.0.119) at 2:61:7e:fd:37:c3 on en7 ifscope [ethernet]
? (192.168.0.172) at aa:75:7b:52:52:34 on en0 ifscope [ethernet]
? (192.168.0.172) at aa:75:7b:52:52:34 on en7 ifscope [ethernet]
? (224.0.0.251) at 1:0:5e:0:0:fb on en0 ifscope permanent [ethernet]
? (224.0.0.251) at 1:0:5e:0:0:fb on en7 ifscope permanent [ethernet]
```

Bilde 1: Viser 'arp -a' kommandoen i terminalen. Listen er over alle nettverkskortene på maskinen.

IP-adresse	MAC-adresse	Interface	Type	Leverandør
192.168.0.1	b8:d5:26:dd:14:d0	en0	Dynamisk	Zyxel Communications Corporation
192.168.0.1	b8:d5:26:dd:14:d0	en7	Dynamisk	Zyxel Communications Corporation
192.168.0.101	d6:73:8d:17:ee:15	en0	Dynamisk	Ikke funnet
192.168.0.101	d6:73:8d:17:ee:15	en7	Dynamisk	Ikke funnet
192.168.0.112	7a:44:f6:51:8f:c	en0	Dynamisk	Ikke funnet
192.168.0.112	7a:44:f6:51:8f:c	en7	Dynamisk	Ikke funnet
192.168.0.119	2:61:7e:fd:37:c3	en0	Dynamisk	Ikke funnet
192.168.0.119	2:61:7e:fd:37:c3	en7	Dynamisk	Ikke funnet
192.168.0.172	aa:75:7b:52:52:34	en0	Dynamsik	Ikke funnet
192.168.0.172	aa:75:7b:52:52:34	en7	Dynamsik	Ikke funnet
224.0.0.251	1:0:5e:0:0:fb	en0	Statisk	Reservert IPv4 mulitcast (Internet Engineering Task Force, 2021)
224.0.0.251	1:0:5e:0:0:fb	en7	Statisk	Reservert IPv4 mulitcast (Internet Engineering Task Force, 2021)

Table 3: Tabellen viser alle nettverkskortene på vår maskin, og deres respektive leverandører.

Oppgave 3 – IPv6

a) *Routing prefix* og *Subnet ID* er de to delene som til sammen utgjør *Network prefix* som er de 64 første bitsene i *General unicast address format*. I dette formatet så vil *Routing prefix* være 48 eller flere bits og *Subnet ID* henholdsvis 16 eller færre bits. Om vi har *Subnet ID* lik null vil en IPv6-adresse se slik ut «a:b:c:d:e:f:g:h/64» og vi har da en 64-bit-nettadresse (a:b:c:d) og en 64-bit-nodeadresse (e:f:g:h). Dette vil si at vi har plass til $2^{64} \approx 1.85 \cdot 10^{19}$ nodeadresser i dette ene subnettet. Her er som sagt *Subnet ID* satt til null og vi har da kun et subnett tilgjengelig, om vi øker denne vil vi få en øking i antall subnett tilgjengelig.

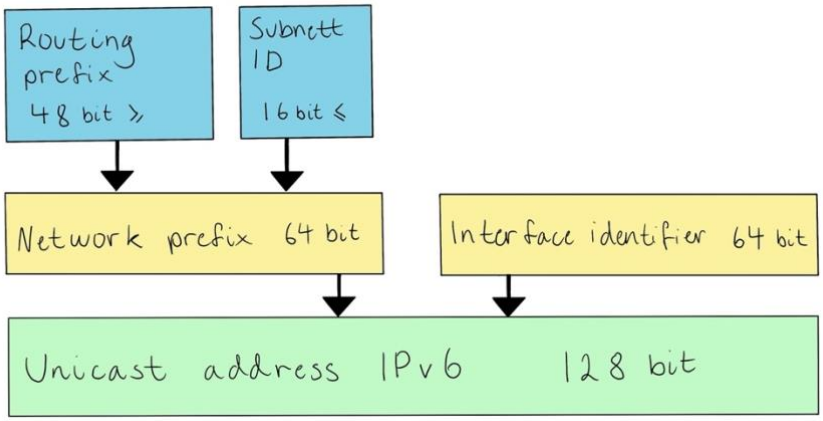


Figure 2: Illustrasjon av oppbygningen til IPv6 Unicast Address Format.

b) Se under: IPv6-adressen «2001:db8::/60», fire subnett med 25-125 kapasitet

Vi får behovet for fire subnett dekket av at adressen er «/60» da dette gir fire bit til *Subnet ID* som da er $2^4 = 16$ subnett, alle har hver sin 64 bit *Interface identifier* som alltid er 64 bit i en *Unicast address*, og gir hvert subnett $2^{64} \approx 1.85 \cdot 10^{19}$ nodeadresser hver. Det er da god plass til de 25-125 brukerne på de forskjellige subnettene. I *Table 4* ser vi startadressen til de fire laveste subnettene, derav deres laveste og høyeste adresse. Det er da også, som nevnt tidligere, alltid 64 bit med adresser i en *Unicast address*. Vi fikk som sagt 16 subnett og vi har brukt fire, da sitter vi igjen med tolv subnett som er ledige.

Prefiks	Subnettadresse start	Min host	Max host
60	2001:db8:0:0	2001:db8:0:0:0:0:0:0	2001:db8:0:0:ffff:ffff:ffff:ffff
60	2001:db8:0:1	2001:db8:0:1:0:0:0:0	2001:db8:0:1:ffff:ffff:ffff:ffff
60	2001:db8:0:2	2001:db8:0:2:0:0:0:0	2001:db8:0:2:ffff:ffff:ffff:ffff
60	2001:db8:0:3	2001:db8:0:3:0:0:0:0	2001:db8:0:3:ffff:ffff:ffff:ffff

Table 4: Tabellen viser de fire minste (i adresse, ikke størrelse) subnettene spurt etter i oppgaven.

Oppgave 4 – DHCP

a) Fra klient, Option(53) Message type: *Discover*.

MAC (3 første byte)		IP		UDP Portnummer	
DST	SRC	SRC	DST (2 byte)	SRC	DST
ff:ff:ff	44:85:00	0.0.0.0	255.255	68	67

Table 5: Tabellen viser innholdet i første pakke (Discover) i DHCP konfigureringen.

Fra tjener, Option(53) Message type: *Offer*.

MAC (3 første byte)		IP		UDP Portnummer	
DST	SRC	SRC	DST (2 byte)	SRC	DST
44:85:00	b8:d5:26	192.168.0.1	192.168	67	68

Table 6: Tabellen viser innholdet andre pakke (Offer) i DHCP konfigureringen.

Fra klient, Option(53) Message type: *Request*.

MAC (3 første byte)		IP		UDP Portnummer	
DST	SRC	SRC	DST (2 byte)	SRC	DST
ff:ff:ff	44:85:00	0.0.0.0	255.255	68	67

Table 7: Tabellen viser innholdet tredje pakke (Request) i DHCP konfigureringen.

Fra tjener, Option(53) Message type: *ACK*.

MAC (3 første byte)		IP		UDP Portnummer	
DST	SRC	SRC	DST (2 byte)	SRC	DST
44:85:00	b8:d5:26	192.168.0.1	192.168	67	68

Table 8: Tabellen viser innholdet fjerde pakke (ACK) i DHCP konfigureringen.

Source	Destination	Protocol	Length	Info
0.0.0.0	255.255.255.255	DHCP	343	DHCP Discover
192.168.0.1	192.168.0.123	DHCP	347	DHCP Offer
0.0.0.0	255.255.255.255	DHCP	369	DHCP Request
192.168.0.1	192.168.0.123	DHCP	372	DHCP ACK

Bilde 2: Viser, fra Wireshark, de fire DHCP-pakkene som ble observert ved gjennomføring av oppgaven.

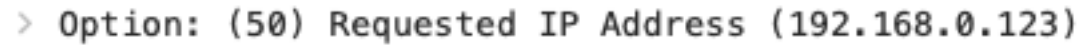
DHCP fungerer i teorien slik:

- Klient ønsker auto konfigurert TCP/IP. Setter da nettverkskortets IP-adresse til «0.0.0.0» da den ikke har noen IP-adresse ennå. Denne IP-adressen kan ikke brukes for å kontakte en maskin, men brukes her som avsender adresse i mangel av noe annet. Klienten sender da ut en DHCP *Discover* pakke ut på kringkastingsadressen «255.255.255.255», og «spør» de andre i nettet om noen kan tilby TCP/IP-informasjon.
- En tjener vil svare med en DHCP *Offer* pakke. Da klient ikke har noen IP-adresse enda, vil tjener sitt svar også sendes til kringkastingsadressen. Denne pakken inneholder all TCP/IP-informasjonen som klienten trenger.
- Klienten vil deretter akseptere tjeneren sitt tilbud med en DHCP *Request* pakke. Også denne vil bli sendt til kringkastingsadressen.
- Til slutt vil tjeneren svare med en DHCP *ACK* pakke. Når klienten får denne, kan den ta i bruk TCP/IP-konfigurasjonene den har fått.

(Hallsteinsen, Klefstad, & Skundberg, 2020)

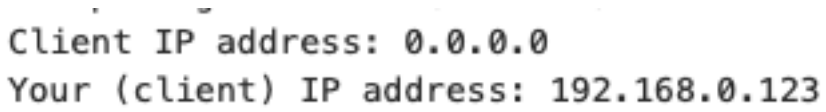
1) Fra klienten sin side så blir alle pakker sendt til kringkastingsadressen «ff:ff:ff:ff:ff:ff:ff:ff». Tjener svarer da alltid til klienten sin MAC-adresse, i dette tilfellet «44:85:00:d2:82:9b». (Dynamic Host Configuration Protocol, 2021) Dette stemmer overens med teorien.

2) Fra teorien vet vi at siden klienten ikke har noen IP-adresse før DHCP kallene er ferdig så vil alle pakker bli sendt til kringkastingsadressen «255.255.255.255» for at klienten skal kunne få dem, og siden den ikke har noe informasjon om TCP/IP-konfigurasjonene på nettet så må den også sende til kringkastingsadressen.



Bilde 3: Viser 'Option'-feltet med verdi '50' til DHCP-pakken 'Dicover'.

I praksis derimot observerer vi at klienten sender til kringkastingsadressen, men tjeneren bruker en annen IP-adresse, her «192.168.0.123», som er det IP-adressen som klient har spurt om å få, se *Bilde 3*. Ettersom pakkene går mellom vår maskin og DGW vil de være i samme subnett, så IP-adressen vil ikke ha noe å si da det kun er MAC-adressene som vil bli brukt til å sende DHCP pakken til sin destinasjon. Vi ser også i *Bilde 4* at tjeneren har registrert klienten sin IP-adresse som «0.0.0.0», og har gitt oss den forespurte IP-adressen, og adressert IP-pakken til denne adressen i stedet.



Bilde 4: Viser DHCP-feltene 'Client IP address' og 'Your IP address' i DHCP-pakken 'Offer'.

3) Klienten sender fra IP-adresse «0.0.0.0» da den ikke har fått konfigurert TCP/IP enda. Dette stemmer med teorien nevnt tidligere, og den vil konfigurere seg etter den får DHCP *ACK* pakken fra tjener.

b) I *Table 9* under ser vi hva DHCP konfigureringene fra tjener er satt til, disse er like i både DHCP *Offer* og -*ACK* pakkene. Dette stemmer overens med protokollen: «Any configuration parameters in the DHCPACK message SHOULD NOT conflict with those in the earlier DHCPOFFER message to which the client is responding.» (Internet Engineering Task Force, 2021)

DHCP pakke type	Option	Option navn	Option verdi
Offer	1	Subnet Mask	255.255.255.0
Offer	3	Router	192.168.0.1
Offer	6	Domain Name Server	84.208.20.110 og 84.208.20.111
ACK	1	Subnet Mask	255.255.255.0
ACK	3	Router	192.168.0.1
ACK	6	Domain Name Server	84.208.20.110 og 84.208.20.111

Table 9: Tabellen viser en oversikt over 'Option'-feltene 1, 3 og 6 for DHCP-pakkene 'Offer' og 'ACK'.

Oppgave 5 – Dokumentering av programmeringsøvinger 5

1) Øvingens hensikt
   Hensikten med øving 5 var å lage en nettside som lar en skrive inn kode (valgfritt hvilket programmeringsspråk, vi valgte C++), som blir sendt til en server hvor koden blir kompilert og kjørt i et virtuelt og trygt miljø. Virtualiseringen skulle bli gjort med Docker. Outputen til koden som ble kjørt skulle sendes tilbake til nettsiden.

2) Ordforklaringer
   TCP 3-way-handshake: TCP-pakker som sendes for å opprette en TCP-forbindelse. Dette må bli gjort før data kan overføres, dette fordi TCP er en pålitelig overføringsteknologi. En av partene må ta initiativ (så og si alltid klient i en klient-tjener-sammenheng), i vårt tilfelle nettsiden. Den sender da en *SYN*-pakke, en oppkoblingsforespørsel, som andre part svar og kvitterer for med en *SYN-ACK*-pakke. Første part må da kvittere andre part sin *SYN* for å bekrefte koblingen den veien, så den sender en *ACK*-pakke. Det blir da tre pakker, derav *3-way-handshake*. Nå er forbindelsen opprettet, og data kan bli overført over den.

   Når det skal overføres data vil TCP-protokollen dele dataen fra applikasjonen i segmenter (TCP-segmenter) som er store nok til å passe i nyttelasten til IP-pakkene. Alle TCP-segmenter har et sekvensnummer brukes til å validere om alle IP-pakker har kommet frem, om de ikke har det, vil de bli spurt etter på nytt, det er dette som gjør TCP til en pålitelig overføringsteknologi. For alle mottatte pakker vil det kvitteres med en *ACK*-pakke (en *ACK*-pakke kan kvittere for flere mottatte pakker), som da har *ACK*-nummer lik forventet sekvensnummer på neste pakke. Om *ACK*-nummeret er feil, vet avsender at ikke alle pakkene har kommer riktig frem.

   Når TCP-forbindelsen skal kobles ned tar en av partene initiativ ved å sende en *FIN*-pakke, da vil den andre parten kvittere denne mottatt og sende sin egen FIN-pakke (disse blir ofte da en pakke med både *FIN* og *ACK*) for å koble ned begge veier. Der

etter må første part kvittere for andre part sin *FIN*-pakke. For mer detaljert beskrivelse av nedkoblingsfasen se oppgave 2 b) i vedlegg A.

HTTP-pakker: HTTP-pakker har et er enten en *Request* eller en *Response*. En *Request*-pakke har en *request*-linje som kan være *GET, POST, PUT, DELETE,* osv., og alle disse er såkalte *request*-metoder. Disse sier noe om man spør etter data (*GET*), sende data (*POST*), osv. *Response*-pakken har en tilsvarende *status*-linje, som inneholder statuskoder (f.eks. 200 OK, 404 Page not found, 500 Internal server error) som forteller om en *request* var vellykket, om den ikke var gyldig, om noe galt skjedde på serversiden osv. Statuskoden «400 *Bad Request*» betyr at feilen ligger hos klient, det klienten sendte kommer ikke gjennom serverens validering og serveren vil ikke prosessere det. Statuskoden «500 Internal Server Error» betyr at noe feil har skjedd på serversiden som gjør at klienten ikke kan få det den har spurt etter, og serveren kan ikke være mer spesifikk en at noe gikk galt.

Videre struktur på *Request og Response*-pakker er lik. De består av et *header*-felt, med header-linjer, og en *body*. *Header*-linjer har informasjon som sier noe om HTTP-koblingen. Vi har *header*-linjen «Keep-Alive» som kan spesifisere hvor mange sekunder forbindelsen kan være oppe uten at pakker blir utvekslet (satt med f.eks. *timeout=5*, da fem sekunder) og hvor mange pakker som totalt kan bli sendt over samme forbindelse (satt med f.eks. *max=100*, da 100 pakker).

3) Program og datakommunikasjon

Først må server startes og «index.html»-fila åpnes. Deretter skrives det inn litt kildekode på nettsiden og knappen «Compile and run» starter datakommunikasjonen. I *Bilde 5* under ser vi kildekoden som ble sendt fra nettsiden og som er brukt videre i forklaringene.

```
#include <iostream>

int main()
{
   std::cout << "Kake!" << std::endl;
   return 0;
}
```

Bilde 5: Kildekode som er skrevet inn i nettsiden og brukt videre i forklaringene.

Når knappen blir trykt på kjøres JavaScript koden i *Bilde 6*. Først opprettes et «*XMLHttpRequest*»-objekt og denne klargjøres til en *POST*-melding i tredje linje, denne meldingen skal da gå til den spesifiserte URL-en på linje to.

```
const Http = new XMLHttpRequest(); //Opprett
const URL = "http://localhost:8080/compile";
Http.open("POST", URL); //Åpner for en post
```

Bilde 6: JavaScript koden som kjøres når knappen på nettsiden trykkes på.

Når tredje linje kjøres vil det bli opprettet kontakt mellom nettsiden og serveren. Dette gjennom et *TCP 3-way-handshake*. Dette ser vi klart i *Bilde 7* at nettsiden (på port 57097) starter *3-way-handshake* med en *TCP SYN*-pakke.

```
TCP        88  57097 → 8080 [SYN] Seq=0
TCP        88  8080 → 57097 [SYN, ACK] S
TCP        76  57097 → 8080 [ACK] Seq=1
```

Bilde 7: Viser pakker, fra WireShark, involvert i 3-way-handshake.

Når nettsiden og serveren har satt opp en TCP-forbindelse så vil linje en i *Bilde 8* bli kjørt. Denne koden henter kildekoden (som er skrevet inn) fra nettsiden, og sender denne som en *HTTP-POST*-melding over TCP forbindelsen. Dette ser vi ved pakke en og tre i *Bilde 9* som går fra nettsiden til serveren. Deretter ser vi at server bekrefter pakken fra nettsiden som mottatt. Den tredje pakken som er merket som en *HTTP-POST*-pakke markerer at nettsiden er «ferdig» med å sende data. Den andre kodelinjen i *Bilde 8* forteller bare brukeren av nettsiden at den venter på svar fra serveren. Den siste kodelinjen vil vente på svar fra server, denne ser vi mer på senere.

```
Http.send(encodeURIComponent(document.getElementById("input-field").value));
document.getElementById("output-text").innerText = "Compiling...";
Http.onreadystatechange = function ()
```

Bilde 8: Kode linjer for sending av kildekoden fra nettside til server.

```
TCP        428  57097 → 8080 [PSH, ACK] Seq=1 Ack=1
TCP         76  8080 → 57097 [ACK] Seq=1 Ack=353 Win=
HTTP       279  POST /compile HTTP/1.1  (text/plain)
TCP         76  8080 → 57097 [ACK] Seq=1 Ack=556 Win=
```

Bilde 9: Pakker, fra WireShark, fra klient, med data, da kildekoden fra nettsiden.

Den første TCP-pakken i *Bilde 9* ser vi *payload*-en til i *Bilde 11* hvor vi kan se at det ligger en *HTTP-POST* melding med alle *header*-ene til *HTTP*-en. Resten av innholdet som blir sendt fra nettsiden til tjener kommer i den tredje pakken. Som vi ser innholdet til i *Bilde 10*. Om man ser litt etter kan man se kildekoden som var skrevet inn på nettsiden som innholdet i denne pakken. Til slutt vil serveren sende en kvittering på at den har mottatt pakkene fra nettsiden.

```
·······z ···z%23i
nclude%2 0%3Ciost
ream%3E% 0A%0Aint
%20main( )%0A%7B%
0A%20%20 %20%20st
d%3A%3Ac out%20%3
C%3C%20% 22Kake!%
22%20%3C %3C%20st
d%3A%3Ae ndl%3B%0
A%20%20% 20%20ret
urn%200% 3B%0A%7D
%0A%20%2 0%20%20%
20%20%20 %20%20%2
0%20%20
```

Bilde 10: Payload-en til den tredje pakken i Bilde 9.

```
·······z ···zPOST
 /compil e HTTP/1
.1··Host : localh
ost:8080 ··Conten
t-Type:  text/pla
in;chars et=UTF-8
··Origin : file:/
/··Conne ction: k
eep-aliv e··Accep
t: */*·· User-Age
nt: Mozi lla/5.0
(Macinto sh; Inte
l Mac OS  X 10_15
_6) Appl eWebKit/
605.1.15  (KHTML,
 like Ge cko) Ver
sion/14.  0.3 Safa
ri/605.1  .15··Con
tent-Len gth: 203
··Accept -Languag
e: en-us ··Accept
-Encodin g: gzip,
```

Bilde 11: Payload-en til den første pakken i Bilde 9.

Når serveren mottar pakken så vil det skje på linje en og to i *Bilde 12*. Videre vil Java koden til metoden på linje to bli kjørt (inputSourceCode(String)). Denne tar inn kildekoden nettsiden sender som en *String* og gjør litt enkel formatering og validering.

```
@PostMapping(value = "/compile")
public ResponseEntity<String> inputSourceCode(@RequestBody String input)
```

Bilde 12: Viser Javakoden som mottar pakkene fra klienten.

Deretter kjører den koden som kompiler kildekoden i et *Docker Image* og henter ut *output*-en som blir sendt tilbake til nettsiden. Dette ser vi i *Bilde 13*, hvor vi ser at server sender tilbake *output* fra Docker i linje 2 og har *Http status OK*.

```
String outputFromDocker = new String(runDockerImage.getInputStream().readAllBytes());
return new ResponseEntity<>(outputFromDocker, HttpStatus.OK); //returner output koden
```

Bilde 13: Javakoden som kjører kildekoden fra nettsiden i Docker, og returnerer output-en.

Om noe feil skjer med valideringen vil koden i *Bilde 14* bli kjørt som forteller bruker av nettsiden at det var en *Bad Request*, og nettsiden vil vise at kompileringen feilet.

```
return new ResponseEntity<>( body: "Compilation failed, check syntax!", HttpStatus.BAD_REQUEST);
```

Bilde 14: Javakoden som kjører om kompilering feilet.

Om noe feil skjer med kjøringen av Docker vil koden i *Bilde 15* bli kjørt. Her forteller vi bruker av nettsiden at det har skjedd en server feil (*Internal Server Error*).

```
return new ResponseEntity<>( body: "Server error!", HttpStatus.INTERNAL_SERVER_ERROR);
```

Bilde 15: Javakoden som kjører om det skjer noe feil ved Docker.

Nå ser vi på tilfellet av at alt har gått bra på server siden og den returnerer *output*-dataen til nettsiden, de andre tilfellene ville vært helt like, men med annen data og statuskode.

I *Bilde 16* viser to pakker, den første er en *HTTP-OK* pakke fra server til nettsiden med *output*-dataen fra kildekoden som ble kjørt, denne dataen ser vi i *Bilde 17*. Den siste pakken er en bekreftelse fra nettsiden på at den har mottatt pakken fra server. Her ser vi en *HTTP*-melding med *header*-felt og innholdet (i blått, «Kake!»).

```
HTTP        243 HTTP/1.1 200   (text/plain)
TCP          76 57097 → 8080 [ACK] Seq=556
```

Bilde 16: Pakker fra server til klient, svar på kompileringen.

```
·····; ···zHTTP
/1.1 200  ··Conte
nt-Type:  text/pl
ain;char set=UTF-
8··Conte nt-Lengt
h: 6··Da te: Sat,
 13 Mar  2021 17:
19:04 GM T··Keep-
Alive: t imeout=6
0··Conne ction: k
eep-aliv e····Kak
e!·
```

Bilde 17: Payload-en til pakke en i Bilde 16.

Som nevnt tidligere så venter JavaScript koden på svar fra serveren, noe den får nå. Da vil koden i *Bilde 18* bli kjørt, denne koden tar ganske enkelt og setter svaret fra server inn i et tekstfelt i *HTML* koden, og svaret fra server dukker da opp på nettsiden.

```
Http.onreadystatechange = function ()
{
    document.getElementById("output-text").innerText = Http.responseText;
}
```

Bilde 18: Viser JavaScript-koden som kjører når nettsiden får svar fra server.

Til slutt, om bruker av nettsiden ikke trykker noe mer på knappen, vil TCP-forbindelsen være stille. Etter en stund vil server eller klient si at det er for lenge siden det har vært noe aktivitet, dette er 60 sekunder som vi ser stemmer med hva *Keep-Alive*-header-feltet i HTTP-pakken fra server sier, dette ser vi i *Bilde 19*.

```
Keep-Alive: timeout=60\r\n
```

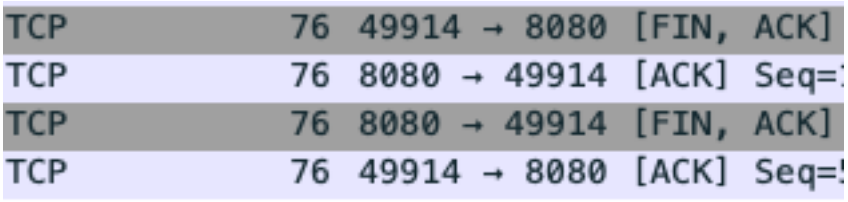
Bilde 19: Header-felt 'Keep-Alive' fra HTTP-pakke 1 fra Bilde 16.

I *Bilde 20* ser vi at tidsforskjellen mellom kvitteringen fra nettsiden til server (på mottatt svar) og første *FIN*-pakke stemmer godt overens med *timeout*-feltet.

```
2.812426    ::1    ::1    TCP    76 49806 → 8080 [ACK] Seq=
60.061137   ::1    ::1    TCP    76 49806 → 8080 [FIN, ACK]
```

Bilde 20: Viser tidsforskjell mellom fåregående pakke og første FIN-pakke. Porten stemmer ikke overens, dette da det ble gjort gjentak av pakkefangst.

Første *FIN*-pakke setter i gang nedkoblingen av TCP-forbindelsen. Det vil da skje en vanlig *nedkoblingsfase*, som vi ser i *Bilde 21*.



Bilde 21: Pakkene involvert i nedkoblingsfasen, fra WireShark. Porten stemmer ikke overens, dette da det ble gjort gjentak av pakkefangst.

4) Sammendrag
   Øvingen gikk greit å gjennomføre. Det var mye knot grunnet mangel på kunnskap til gjennomføring. Dette spesifikt i kobling mellom nettside og server, da dette ikke hadde blitt undervist i «Full-stack applikasjonsutvikling, IDATT2105». Da dette var løst var det resterende relativt greit, da Docker var undervist og gitt gode kilder til i «Nettverksprogrammering, IDATT2104».
   Øvingen ga et klarere bilde av sammenhengen mellom *frontend* og *backend*, dette burede kanskje ha vært forklart tidligere, da det er vanskelig å koble disse sammen, uten forståelse for hvordan dette vanligvis henger sammen. Sitter igjen med et godt inntrykk av hva man potensielt kan bruke Docker til, og ser på Docker som et veldig kraftig verktøy.

Oppgave 6 – Dokumentering av programmeringsøvinger 6

1) Hensikt
   Hensikten med øving 6 var å sette opp en *WebSocket*-server som skulle kunne utføre *handshake* med klienter, lese korte meldinger fra klienter og sende korte meldinger til alle tilknyttede klienter. Mer spesifikt skulle *WebSocket*-serveren sende de korte meldingene den mottok fra en klient til alle andre tilknyttede klienter.

2) Ordforklaring
   For forklaring på TCP og HTTP-protokollene se oppgave 5.2.

   WebSocket-protokollen er definert av RFC6455. WebSocket-teknologi gir mulighet for toveis kommunikasjon mellom klient og tjener. For å opprette en WebSocket-forbindelse må det utføres en *handshake*, dette gjøre gjennom HTTP, hvor HTTP-meldingen inneholder *header*-linjene: «Upgrade: websocket», «Connection: Upgrade» og «Sec-WebSocket-Key: ...». Sist nevnte *header* er en streng som server

må regne ut riktig svar til, dette for å sikre at klienten vet at serveren støtter WebSocket, og ikke er proxy-server. Metoden for å regne ut den riktige svarverdien er definert i RFC6455. Det innebærer å legge til en predefinert streng, kjøre SHA1-hashalgoritme på disse (samlet), og til slutt *base64*-kode dem for å få riktig svar verdi. I serveren sitt svar må tilhørende *header*-linjene: «Upgrade: websocket», «Connection: Upgrade» og «Sec-WebSocket-Accept: ...». Om serveren har riktig *accept*-verdi vil det nå være opprettet en WebSocket-forbindelse.

Nå kan både server og klient sende melinger over denne forbindelsen når de vil. Klienten må alltid maskere meldingene den sender til server, og server skal aldri maskere sine.

3) Program og datakommunikasjon
Server siden startes, og man kobler seg til på «localhost:3000». I *Bilde 22* ser vi koden som starter serveren. Linje en starter en ny prosess som kjører en JavaScript-kode (med Node.js) som gir nettsiden *HTML* koden, som videre har en annen JavaScript-kode som kobler seg til port «localhost:3001» som Java-serveren kjører på, som vi ser på linje to spesifiserer vi at serveren skal ha port 3001. Linje tre starter selve serveren.

```
Process process = Runtime.getRuntime().exec( command: "node src/client.js");
Server server = new Server( port: 3001);
server.start();
```

Bilde 22: Javakode som starter serveren.

I *Bilde 23* ser vi JavaScript-koden som blir kjørt i en egen prosess. Denne koden er fra vedlegget til øvingen. Det blir satt opp en HTTP-server som lytter på port 3000. Når en klient kobler seg opp mot «localhost:3000» vil serveren plukke opp dette. Når en klient kobler seg opp, vil «connection.on(...)»-metoden på linje tre (inne i net.createServer(...)) bli kjørt. Her vil den i *content*-variabelen sette opp *HTML*-koden (denne er fjernet i bildet, men er vist i *Bilde 28*). Deretter sender serveren *content* til klienten som har koblet seg opp.

```
const net = require('net');

// Simple HTTP server responds with a simple WebSocket client test
const httpServer = net.createServer((connection) => {
  connection.on('data', () => {
    let content = ``; //Fjernet innholdet for bildet
    connection.write('HTTP/1.1 200 OK\r\nContent-Length: ' + content.length + '\r\n\r\n' + content);
  });
});
httpServer.listen(3000, () => {
  console.log('HTTP server listening on port 3000');
});
```

Bilde 23: JavaScript-kode som blir kjørt for å sette opp server for redirigering til WebSocket-serveren.

Når klient kobler seg til, må et *3-way-handshake* forkomme for å sette opp TCP-forbindelsen. Dette er de tre første pakkene i *Bilde 24*. Deretter i pakke fem sender klient en standard *HTTP-GET*-forespørsel. Denne kvitterer server på med pakke seks, og sender svar i pakke sju, som klient kvitterer på med pakke åtte.
Innholdet i klienten sin *GET*-pakke ser vi i *Bilde 25*.

```
TCP        88  50625 → 3000 [SYN] Seq=
TCP        88  3000 → 50625 [SYN, ACK]
TCP        76  50625 → 3000 [ACK] Seq=
TCP        76  [TCP Window Update] 300
HTTP      432  GET / HTTP/1.1
TCP        76  3000 → 50625 [ACK] Seq=
HTTP      614  HTTP/1.1 200 OK
TCP        76  50625 → 3000 [ACK] Seq=
```

Bilde 24: Pakkene som inngår i oppstarten av kommunikasjonen mellom klient og tjener.

```
GET / HTTP/1.1
Host: localhost:3000
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
```

Bilde 25: Innholdet i GET-melding fra klient til tjener.

Svaret fra server ser vi i *Bilde 26*. Her kan vi legge merke til at alt fra *content*-variabelen (se *Bilde 28*) er sendt med som data i denne pakken.

```
HTTP/1.1 200 OK
Content-Length: 498

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
  </head>
  <body>
    WebSocket test page, kake
    <script>
      let ws = new WebSocket('ws://localhost:3001');
      ws.onmessage = event => alert('Message from server: ' + event.data);
      ws.addEventListener('open', (event) => {
        ws.send('Kake er super duper godt');
      });
      ws.addEventListener('error', function (event) {
        console.log('WebSocket error: ', event);
      });
    </script>
  </body>
</html>
```

Bilde 26: Innholdet i svaret fra tjener på klienten sin GET-melding.

Etter dette vil ikke klienten og server ha mer å utveksle. Og etter en gitt tid vil nedkoblingsfasen starte, pakkene involvert i nedkoblingen ser vi i *Bilde 27*.

```
TCP        76  3000 → 50625 [FIN, ACK]
TCP        76  50625 → 3000 [ACK] Seq=
TCP        76  50625 → 3000 [FIN, ACK]
TCP        76  3000 → 50625 [ACK] Seq=
```

Bilde 27: Pakker involvert i nedkoblingsfasen mellom JavaScript-serveren og klienten.

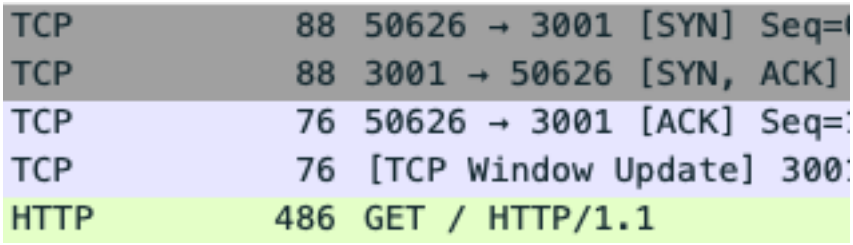
Når klienten mottar *HTML*-koden og denne blir tolket, vil JavaScript-koden i den bli kjørt. Hele *HTML*-koden ser vi i *Bilde 28*.

```
    let content = `<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
  </head>
  <body>
    WebSocket test page, kake
    <script>
      let ws = new WebSocket('ws://localhost:3001');
      ws.onmessage = event => alert('Message from server: ' + event.data);
      ws.addEventListener('open', (event) => {
        ws.send('Kake er super duper godt');
      });
      ws.addEventListener('error', function (event) {
        console.log('WebSocket error: ', event);
      });
    </script>
  </body>
</html>
`;
```

Bilde 28: Innholdet i 'content'-variabelen i JavaScript-koden til serveren.

Som vi ser i *script*-et i *HTML*-koden blir en *WebSocket* opprettet opp mot «ws://localhost:3001» som er Java-serveren vår. Så settes det opp kode for hva som skal gjøres når *WebSocket*-en mottar data, dette er «ws.onmessage = ...», alt som skjer da er at en *Alert*-boks dukker opp med en melding fra server. Så settes det opp kode for når *WebSocket*-åpner seg, dette er «ws.addEventListener(‘open’, ...)», denne sender tesktmeldingen «Kake er super duper godt!» til serveren. Den andre «EventListener»-metoden tar hånd om generelle *error* som kan oppstå og logger dette i nettleseren.

I *Bilde 29* ser vi pakkene som sendes fra klienten til server når *HTML*-koden blir satt opp. De tre første pakkene er en *3-way-handshake* for å opprett TCP-forbindelsen. Så i pakke fem sendes en *HTTP-GET*-forespørsel, det er i denne opprettingen av *WebSocket* begynner.



Bilde 29: Pakker for oppstart av TCP-forbindelsen mellom klient og Java-serveren.

Innholdet i *GET*-pakken ser vi i *Bilde 30*. Her ser vi *header*-en «Upgrade: websocket», som betyr at klienten ønsker å oppgradere forbindelsen til en *WebSocket*-forbindelse. Det følger også med feltet «Sec-WebSocket-Key» som server trenger for å opprette forbindelsen.

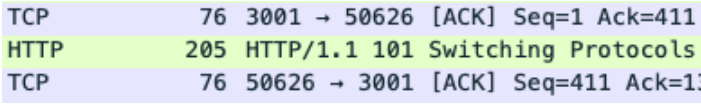
```
GET / HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: localhost:3001
Origin: http://localhost:3000
Pragma: no-cache
Cache-Control: no-cache
Sec-WebSocket-Key: Opd+8otBp5XkZjBLIGnXcg==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: x-webkit-deflate-frame
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_6) AppleWebKit/605.1.15 (KHTML, like Gecko)
Version/14.0.3 Safari/605.1.15
```

Bilde 30: GET-pakke fra klient, som spør om oppgradering til en WebSocket-forbindelse.

Server svarer med *HTTP*-meldingen i *Bilde 31* og aksepterer med denne *WebSocket*-forbindelsen. Denne HTTP-meldingen ser vi i *Bilde 32* som pakke nummer to. Pakke nummer en er kvitteringen fra server til klient om at den mottok *HTTP-GET*-forespørselen, og den tredje pakken er klienten sin kvittering på å ha mottatt serveren sitt svar på *WebSocket*-oppgraderingen.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: RyQ5MMqkMonBVa9RfJyKJKz433Y=
```

Bilde 31: Java-serveren sitt svar på oppgradering til en WebSocket-forbindelse.



Bilde 32: Pakker over Java-server sitt svar om WebSocket oppgradering.

Med dette er det opprettet en *WebSocket*-forbindelse og TCP-forbindelsen som server og klient har kommunisert over så langt vil bli nedkoblet på samme vis som vist tidligere.

På server siden sa vi at den ble startet og kjørte «start()»-metoden. Denne ser vi i *Bilde 33*. Den oppretter da en *serverSocket* på port 3001 («this.PORT», objektvariabel i Server-klassen, som ble satt ved initialisering). Serveren går da i en evig *loop*. Første linje i *loop*-en oppretter et *Socket*-objekt, dette skjer når noen kobler seg opp mot port 3001. Deretter opprettes en *ServerThread* som bruker *Socket*-en og har tilgang på en liste over alle *ServerThreads*. Deretter blir den lagt i nevnte liste, og startet.

```
public void start()
{
    try (ServerSocket serverSocket = new ServerSocket(this.PORT))
    {
        while (true)
        {
            Socket socket = serverSocket.accept();
            ServerThread serverThread = new ServerThread(socket, this.serverThreads);
            this.serverThreads.add(serverThread);
            serverThread.start();
        }
    }
```

Bilde 33: Start-metoden i Server-klassen.

I «*run()*»-metoden i *ServerThread* (denne metoden er *Override*-et fra Thread-klassen, så når «*start()*»-metoden blir kjørt blir «*run()*»-metoden kjørt) brukes et *InputStream*-objekt for å lese bytes som kommer fra *Socket* (se *Bilde 34*). Tilsvarende er det et *OutputStream*-objekt som brukes for å skrive til *Socket* (se *Bilde 35*).

```
try
{
    InputStream inputStream = this.socket.getInputStream();
    while (this.running)
    {
```

Bilde 34: Viser 'InputStream' fra ServerThread-klassen.

```
this.outputStream = this.socket.getOutputStream();
```

Bilde 35: Viser 'OutputStream' fra ServerThread.

Når *ServerThread* har lest av melding fra klient, vil denne meldingen bli tolket. Om den inneholder «GET» så vil programmet tolke den som en forespørsel om å oppgradere til *WebSocket*, den vil da hente ut verdien til «Sec-WebSocket-Key» og ta denne, legge til «258EAFA5-E914-47DA-95CA-C5AB0DC85B11» (denne strengen er predefinert i RFC6455) bak nøkkelen. Deretter blir den nye nøkkelen kjørt gjennom SHA1-hashingalgoritmen, og til slutt vil den bil *Base64-encoded*. Resultatet av dette er verdien i «Sec-WebSocket-Accept» i serveren sitt svar til klienten (se *Bilde 31*). Ellers er resten av innholdet i serveren sitt svar på *WebSocket*-oppgraderingen kun det som er krevd av RFC6455 standarden. Svaret som blir sendt er pakke to i *Bilde 32*.

Når klienten mottar pakken om at serveren har akseptert *WebSocket*-oppgraderingen, og om server har riktig «Sec-WebSocket-Accept», så vil den sende meldingen «Kake er super duper godt!», som sagt tidligere. Denne pakken sendes da over *WebSocket*-forbindelsen og vi ser den som pakke nummer en i *Bilde 36*, pakke nummer to er serveren sitt svar.

```
WebSocket    106  WebSocket Text [FIN] [MASKED]
WebSocket     94  WebSocket Text [FIN]
```

Bilde 36: Viser pakkene over WebSocker-forbindelsen.

Serveren leser nå av meldingen fra klient på nytt, etter det tolker den dette til å være en melding fra klient. Da vil den dekode innholdet slik det er beskrevet i RFC6455. Programmet vårt leser kun korte tekstmeldinger. I *Bilde 37* ser vi hva klienten har sendt server, hvor stor denne *payload*-en var, og den dekodede strengen til slutt.

```
From Client:
□□a□□□*k□A€□ ≠□ □□□ €□ Û□
Payload: 24
Kake er super duper godt
```

Bilde 37: Serveren sin dekodede versjon av det klient sendte.

Når serveren har dekodet meldingen sender den en melding til alle andre tilknyttede klienter. I *Bilde 38* ser vi hva som skjer inne i en *loop* som går gjennom alle klientene. Her er *total* hva serveren sender til klientene, denne meldingen er da også formatert til RFC6455 sine spesifikasjoner og er satt til *final* og at det er en tekstmelding, som vi ser er pakke to i *Bilde 36*.

```
if (this != serverThread)
{
    serverThread.outputStream.write(total, off: 0, total.length);
}
```

Bilde 38: Logikken i loop-en Java-serveren bruker.

I *WireShark* kan vi se den komplette kommunikasjonen mellomklient og server, se *Bilde 39*. Her ser vi i klartekst at klienten sin melding er maskert og uleselig, mens serveren sitt svar, «Kake er godt ja!» er umaskert og helt lesbart.

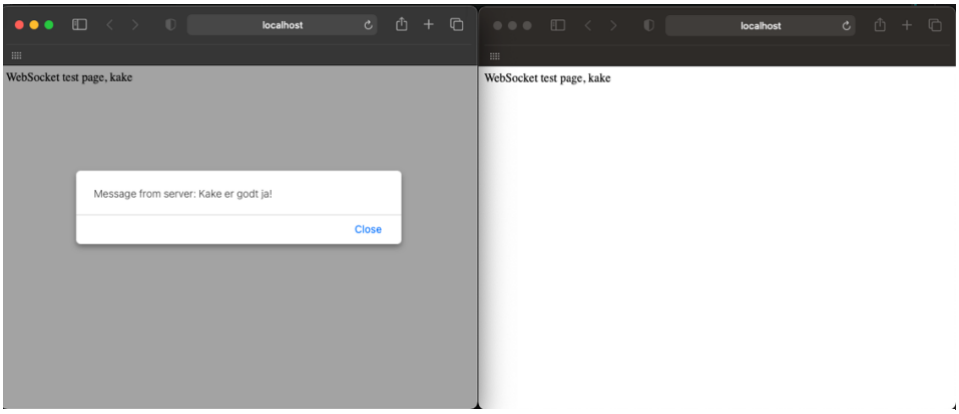
```
GET / HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: localhost:3001
Origin: http://localhost:3000
Pragma: no-cache
Cache-Control: no-cache
Sec-WebSocket-Key: Opd+8otBp5XkZjBLIGnXcg==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: x-webkit-deflate-frame
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_6) AppleWebKit/605.1.15 (KHTML, like Gecko)
Version/14.0.3 Safari/605.1.15

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: RyQ5MMqkMonBVa9RfJyKJKz433Y=

...
.
.k.o.o.*...o.*...o.*.e.~..Kake er godt ja!
```

Bilde 39: Bilde av follow TCP-stream fra WireShark, viser kommunikasjonen mellom klient og tjener i klartekst.

Vi ser i *Bilde 40* hvordan dette kommunikasjonen ser ut for en klient. Til venstre er en klient som har koblet seg til, deretter har en ny klient koblet seg til på høyre side. Når klienten til høyre sender melingen sin til serveren vil serveren sende sin melding til alle andre klienter enn den som sendte meldingen, så altså klienten til venstre. Vi ser her at klienten til venstre har fått opp *Alert*-boksen med meldingen fra serveren.



Bilde 40: Viser hvordan klient siden (i en nettleser) ser ut.

4) Sammendrag

Øvingen gikk greit å gjennomføre. Alt vi trengte var blitt undervist og/eller lagt ved som kilde. Vi valgte å ta utgangspunkt i JavaScript koden som var vedlagt oppgaven, og lage *WebSocket*-serveren i Java. Selve *handshake* mellom klient og server gikk veldig greit. Det var litt knot med å lese og tolke meldinger fra klient, men RFC6455 var godt dokumentert og ga oss svar på tilnærmet alt vi hadde av problemer.

Vi sitter igjen med god forståelse for hvordan *WebSocket* teknologien fungerer, og ser potensialet i å kunne ha toveis kommunikasjon mellom klient og tjener. Det er også veldig nyttig erfaring å kunne navigere og bruke ressurser som RFC6455.

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.

Internet Engineering Task Force. (2021, Mars 12). *IANA Considerations and IETF Protocol Usage for IEEE 802 Parameters.* Hentet fra tools.ietf: https://tools.ietf.org/html/rfc5342

Dynamic Host Configuration Protocol. (2021, Mars 13). Hentet fra Wikipedia: https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol

Internet Engineering Task Force. (2021, Mars 13). *Dynamic Host Configuration Protocol*. Hentet fra tools: https://tools.ietf.org/html/rfc2131#section-3