**Multiplexing og Demultiplexing**

Processer adresseres med IP og port-nummer indenfor en end-host. Processerne sender og modtager data via sockets (som fungere som en dør mellem applikationslaget og transportlaget). **En proces kan kommunikere med mange brugere samtidigt. Altså en proces kan have mange sockets.**

**Multiplexing**

Multiplexing handler om at ”blande” mange beskeder sammen og få dem sendt. Dette sker fra sender-siden (sender-proces) ved at lave et segment ud af dataen og tilføjer en header med afsenderens IP-adresse og port hvor segmentet kom fra.

**Demultiplexing**

Demultiplexing er det modsatte af multiplexing. Her bruger modtageren informationen i segment-headeren (med IP og port) til at finde ud af hvilket socket som segmentet skal leveres over til. Her undersøger den datagrammet som er blevet sendt. *Hvert datagram har en destinations og source IP, hvert datagram indeholder et segment. Hertil har hvert segment også en destination og source port.* En end-host bruger IP-adresse og port til at vidrelevere data til den rigtige socket.

**Demultiplexing i UDP**

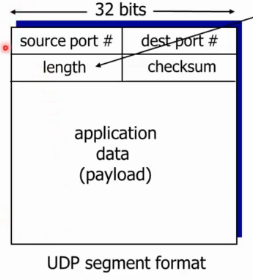
Modtager socket bruger destination IP og port. Når der bliver modtaget et UDP segment tjekkes der for en aktiv socket med destinations port-nummeret og sender den payload dataen vidre. Mange UDP segmenter kan leveres til samme port, selvom destinations IP’en ikke er den samme. Hvis applikationen som modtager segmentet skal kunne svarer tilbage, bruger den source IP og port.

**Demultiplexing i TCP**

TCP sockets, i modsætning til UDP sockets, bruger også afsender *OG* modtagers IP og port-nummer (UDP sockets bruger kun destinations IP og destinations-port). Dette er fordi TCP er en 2-vejs protokol.

**UDP (User Datagram Protokol)**

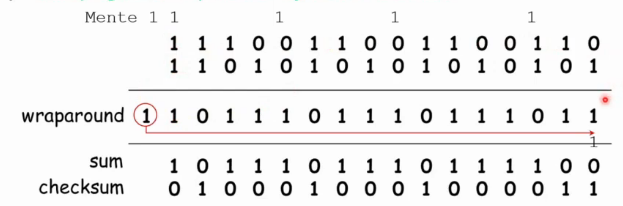
UDP transport er en ”best-effort” protol. Der er ingen etablering af forbindelser (man kan sende data med den hastighed som man ønsker) og der er ingen congestion control (der kan være pakke-tab). UDP-headeren er meget simpel og derfor kan UDP processeres af et netværk med lavt overhead.



Et UDP-segment bruger en checksum til at bestemme om segmentet er sendt korrekt. Checksummen er beregnet af senderen. Modtageren beregner også en checksum. Hvis de 2 summe er identiske antages pakken at være korrekt sendt.

**UDP-checksum beregning**

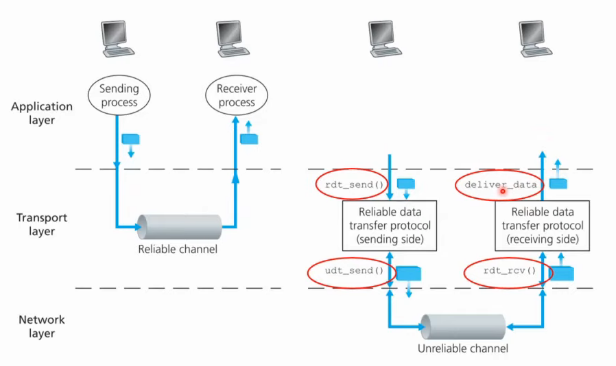
Et UDP-segment betragtes som en serie af 16-bit tal. Senderen laver denne summering af de 16-bit tal. Den tager komplementet til summen og bruger resultatet som checksummen.

Modtageren summere alle 16-bit tal (også checksummen). Denne sum bør give en serie af 1 taller. Hvis ikke, så er pakken ikke korrekt. Eksempel: 

*Opfriskning på bitvis-addition:* 0+1 = 1, 1+0 = 1, 1+1 = 0 (og 1 i mente), 1+1+1 = 1 (og 1 i mente).

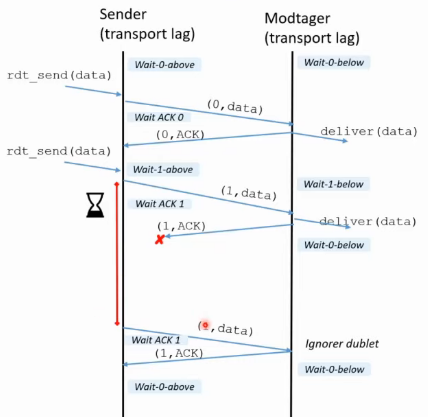
Wraparounds (som i eksemplet er den sidste bit). Denne wraparound skal summeres på resten af summen (addere den ekstra bit). Til sidst tager man komplementet til summen (altså laver en negation ved bit-shift). Denne negation er checksummen. Afsender og modtagers sum skal være den samme.

**RDT (Reliable data transfer)**

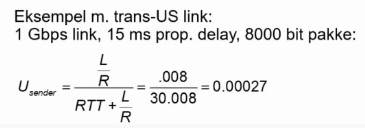


Når der bliver sendt data gennem transportlaget bruges RDT, som sender og modtager pakken. RDT er en tilstandsmaskine som venter på calls fra højere niveauer. Når rdt\_send() sender en pakke kan modtageren med rdt\_rcv() svarer med ACK (pakken er modtaget korrekt) eller NAK (pakken er ikke korrekt og skal derfor retransmiteres). ACK og NAK meddelelser bliver sendt tilbage til afsenderen som en ”kvitering”. Dog kan ACK og NAK meddelelser også gå tabt. Dette problem bliver løst med sekvensnumre hvor man sender hver pakke med et tal, som beskriver hvilken pakke der bliver sendt.

Modtageren sender en pakke med sekvens nummer 1 og venter på afsenderens ACK/NAK før den sender næste pakke med sekvens nummer 2. Hvis ACK/NAK ikke er korrekt, så bliver dataen retransmiteret og der opstår en duplet (som bliver ignoreret af modtageren). Normalt bruges NAK ikke, da det er nemmere at sende ACK og sekvensnummeret. Eksempel:



Ved pakke-tab bliver der ikke sendt noget ACK og sekvensnummer tilbage til afsenderen. Her bruges en timer. Her venter afsenderen på timeren løber ud. Hvis tiden løber ud bliver pakken retransmiteret. Hvis ACK går tabt, så venter afsenderen på timeren og retransmiterer. Dette timeout tid kan estimeres ud fra RTT (men det kan ikke undgås at timeouten er for hurtig/langsom).

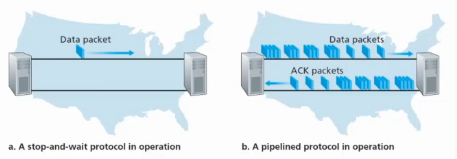
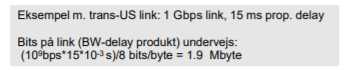
- hvilket ikke er effektivt

R – Transmissions rate (bps)

L – Længden af pakken i bits

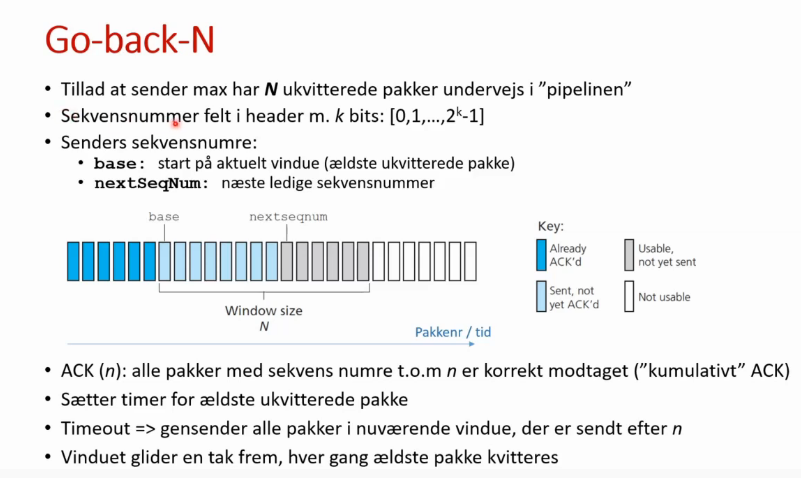
**Pinelinede protokoller**

Ift. En Stop and wait protokol, så sender pinelined protrokoler flere pakker af gangen og afventer ACK fra modtageren. Dette er mere effektivt end Stop & wait, fordi man udnytter mediet vi sender over bedre. (flere pakker bliver sent over en RTT). Eksempel fra før:

Dog kan de stresse netværket at sende mange pakker. Her bruger man sliding window protokoller.

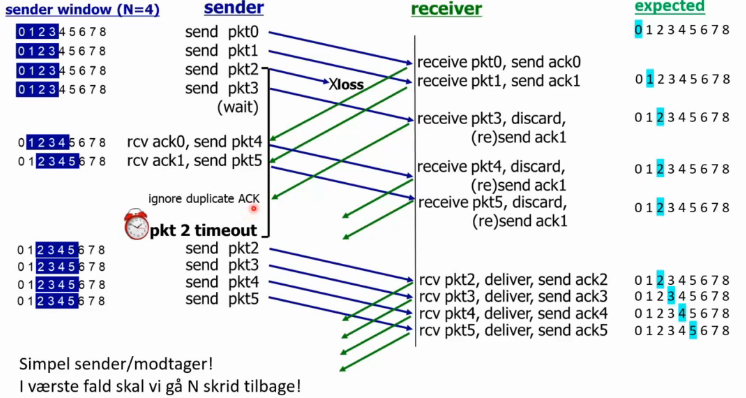
**Go-Back-N**



*Protolregler for sender:* Hvis der kommer et timeout retransmiteres ALLE pakker i vinduet. Hvis der kommer et ACK, så flyttes vinduet en ”tak” frem.

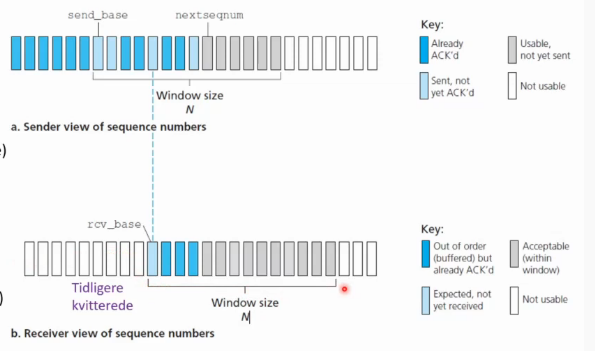
Protokolregler for modtager: Alle pakker forventes at være in-order, altså der ikke er ”huller”. Modtageren skal sende det korrekte sekvensnummer som forventes, duplerende ACKs er godkendt.

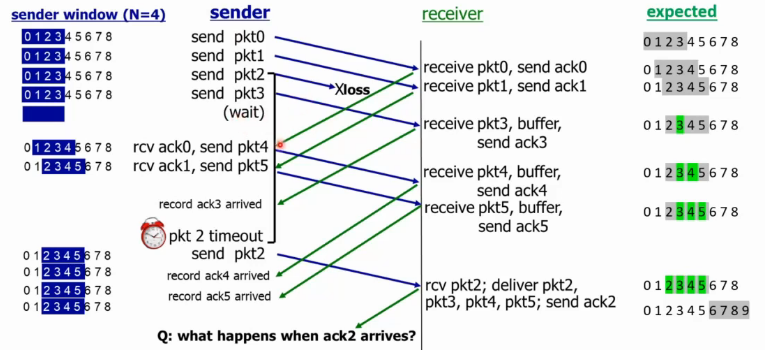
Eksempel på et Go-back-N scenarie:



**Selective Repeat**

Hvor Go-back-N er nødtil at gensende alle pakker hvis en går tabt, så bruger selective repeat en ”out-of-order” buffer, som gemmer pakkerne i en buffer indtil de kan blive ACK’d. Dette er smart, fordi sender skal ikke sende pakker som er modtaget korrekt. Senderen skal altså kun retransmitere de pakker der mangler ACK. Dette kræver dog, at hver pakke har sin egen timer, i modsætning til Go-back-N som har en fælles timer for alle sendte pakker i vinduet.



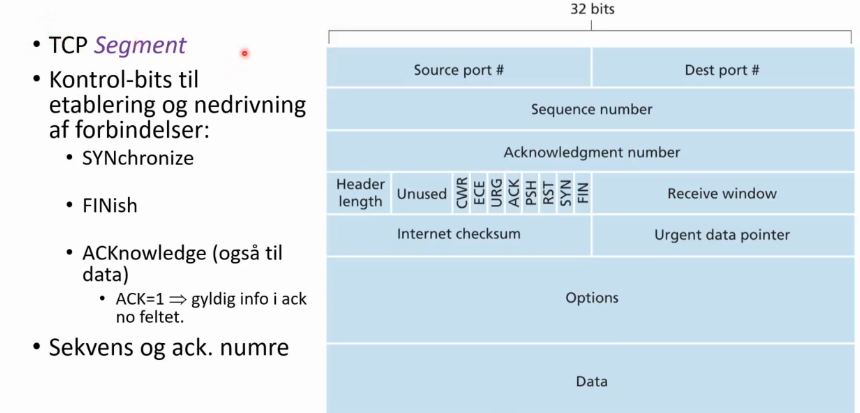


**Etablering af TCP forbindelser**

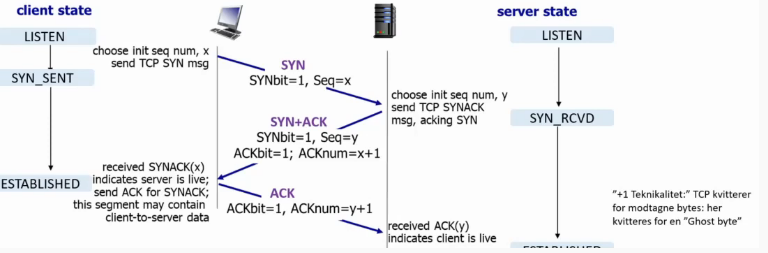
Klient og server skal blive enige om en forbindelse til at afsætte buffer-plads, blive enige om sekvensnumre og vinduestørrelse.

Her opstår der nogle komplikationer ved gamle pakker fra en tidligere forbindelse ikke må medgå i en ny forbindelse. Når forbindelsen lukkes ned, kræves det også at alt data er leveret til modtageren korrekt. Her bruger man specielle pakker til at starte og slutte en forbindelse (disse pakker kan også gå tabt). For at der ikke sker fejl, bruger man derfor et 3-vejs handshake. Clienten sender en req\_connect til serveren som giver en ACK\_connection og til sidst sender clienten et ACK om at den har modtaget serverens ACK.

TCP bruger også et 3-vejs handshake til at oprette en forbindelse. Eksempel på et TCP-segment:



Hvordan et TCP-handhake foregår:



Når forbindelsen skal lukkes, bruges FIN-flaget som starter fra clienten og serveren sender en ACK tilbage. Serveren nedlægger bagefter forbindelsen ved at sende sit FIN-flag til clienten og clienten sender et ACK tilbage.