

Internettværk og Web-programmering

Transport laget

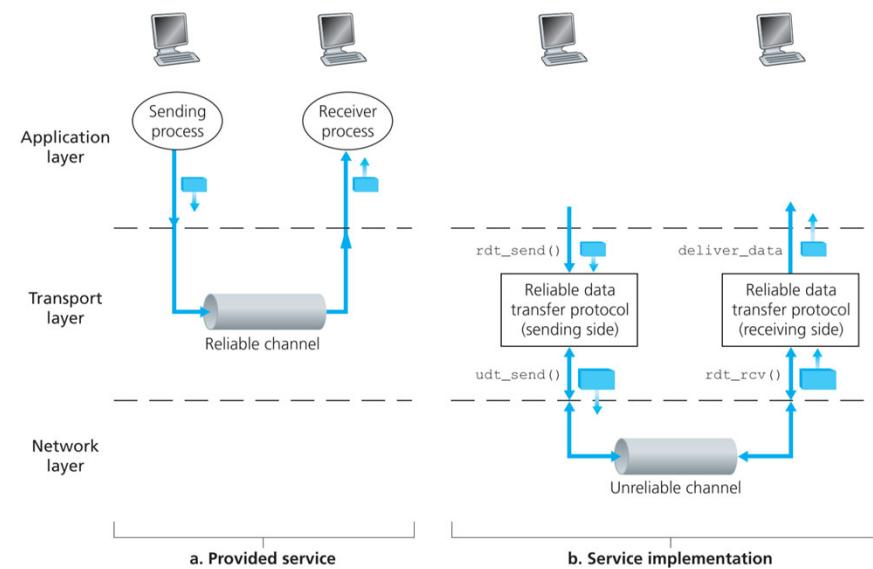
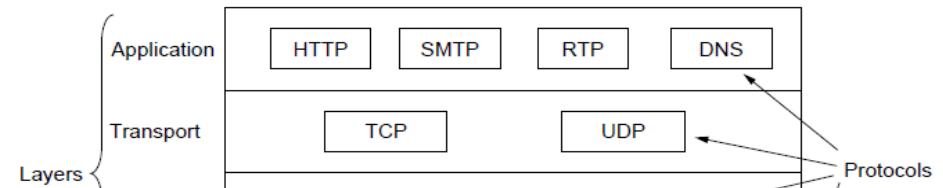
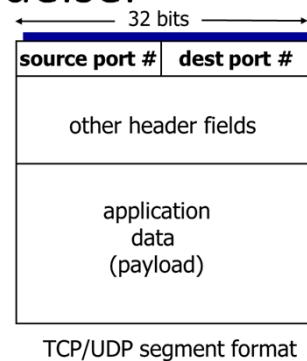
Forelæsning 11
Brian Nielsen

Distributed, Embedded, Intelligent Systems



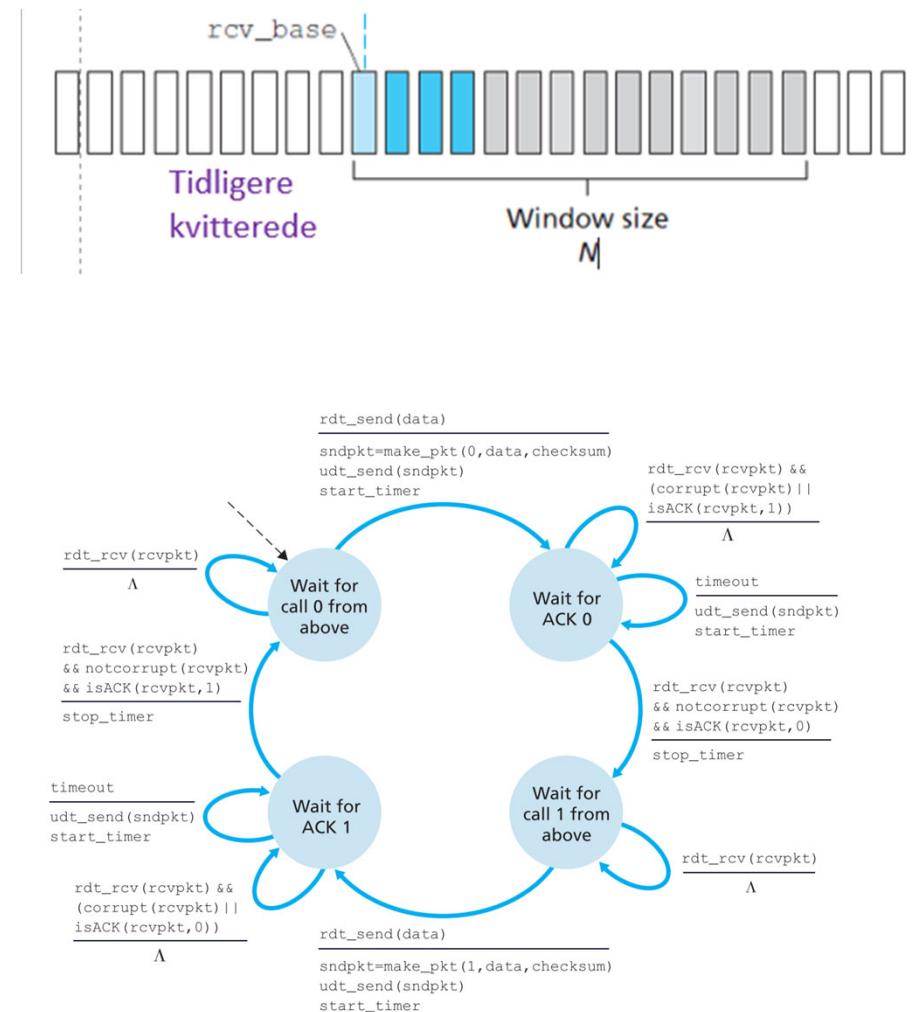
Læringsmål 1

- Forstå opgaver, services, og protokoller på transport-laget
 - TCP og UDP
- Hvordan disse opgaver løses
 - Bindeleddet mellem applikationslaget og transportlaget:
 - "primitiver" (operationer)
 - "sockets" og "demultiplexing"
 - Oprettelse af TCP forbindelser



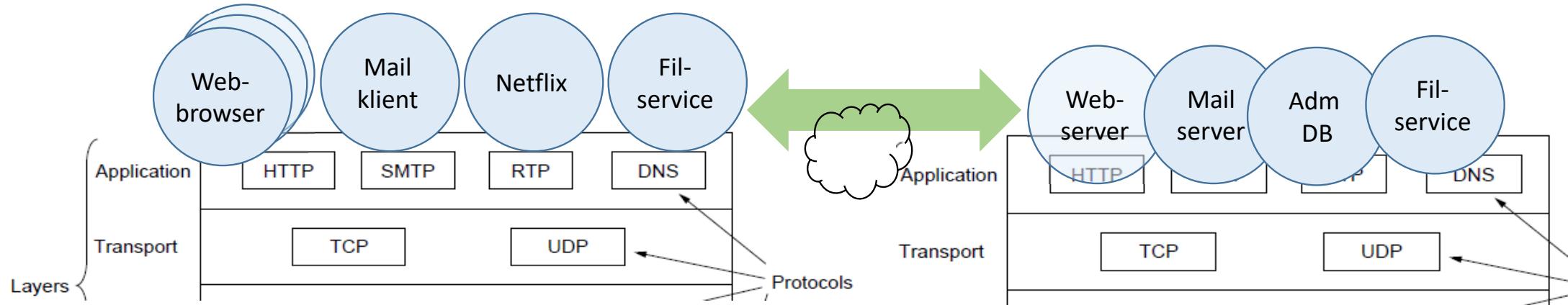
Læringsmål 2

- Hvordan kan en Host A sende en strøm af data til Host B tabsfrit.
 - Hvordan laver man pålidelig end-to-end kommunikation?
 - Detektion af tabte beskeder?
 - Retransmission
 - Brug af sekvensnumre?
 - (go-Back-N, Selektiv Repeat)
 - **"Sliding windows"**
- Beskrivelse af protokol som tilstandsmaskine, (illustration af scenarier via sekvensdiagrammer)



Transportlags protokoller

Transportlaget i TCP/IP modellen



Transportlags opgaver

- **END-TO-END** transport: logisk forbindelse imellem processer (proces=kørende program)
- Opdeling af data i mindre portioner (segmenter), der kan fremsendes af netværkslaget
 - Segmentation and re-assembly
- Fremsendelse af data til korrekte modtager proces
 - Multiplexing and demultiplexing
- **Fejl-korrektion**

TCP (transmission control protocol)

- Forbindelses-orienteret, og
- Pålidelig, ordnet, byte-stream service
- Flow- og congestion-kontrol

UDP (user datagram protocol)

- Forbindelsesløs, og
- Best-effort datagram service
 - Tabte- og omordnede segmenter

En pålidelig transport protokol

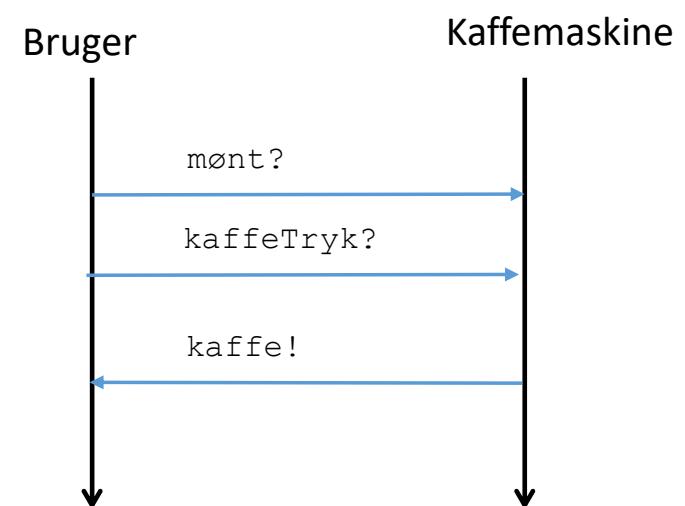
Hvordan sikrer man at data kan nå frem med pakketab?

Hvordan modelleres og specificeres en protokol?

Hvordan implementeres en protokol?

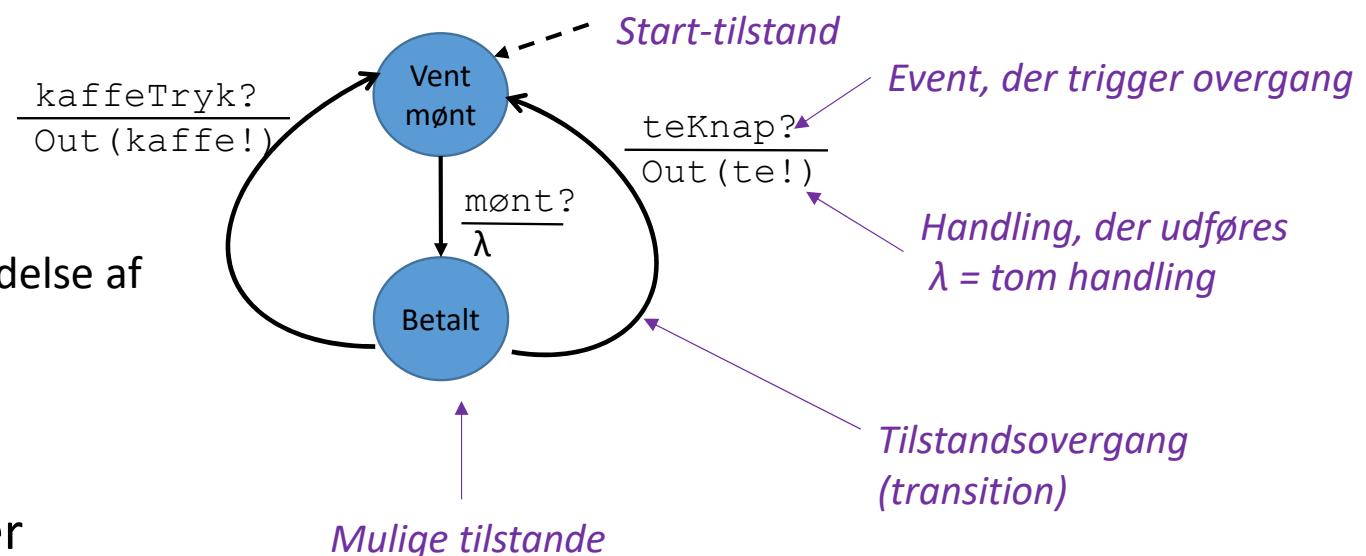
Sekvens diagram

- Generel metode til at beskrive et systems ”opførsel”
 - Interaktion mellem enheder i systemet
 - Viser ét muligt (ud af flere) interaktions-scenarie mellem komponenter

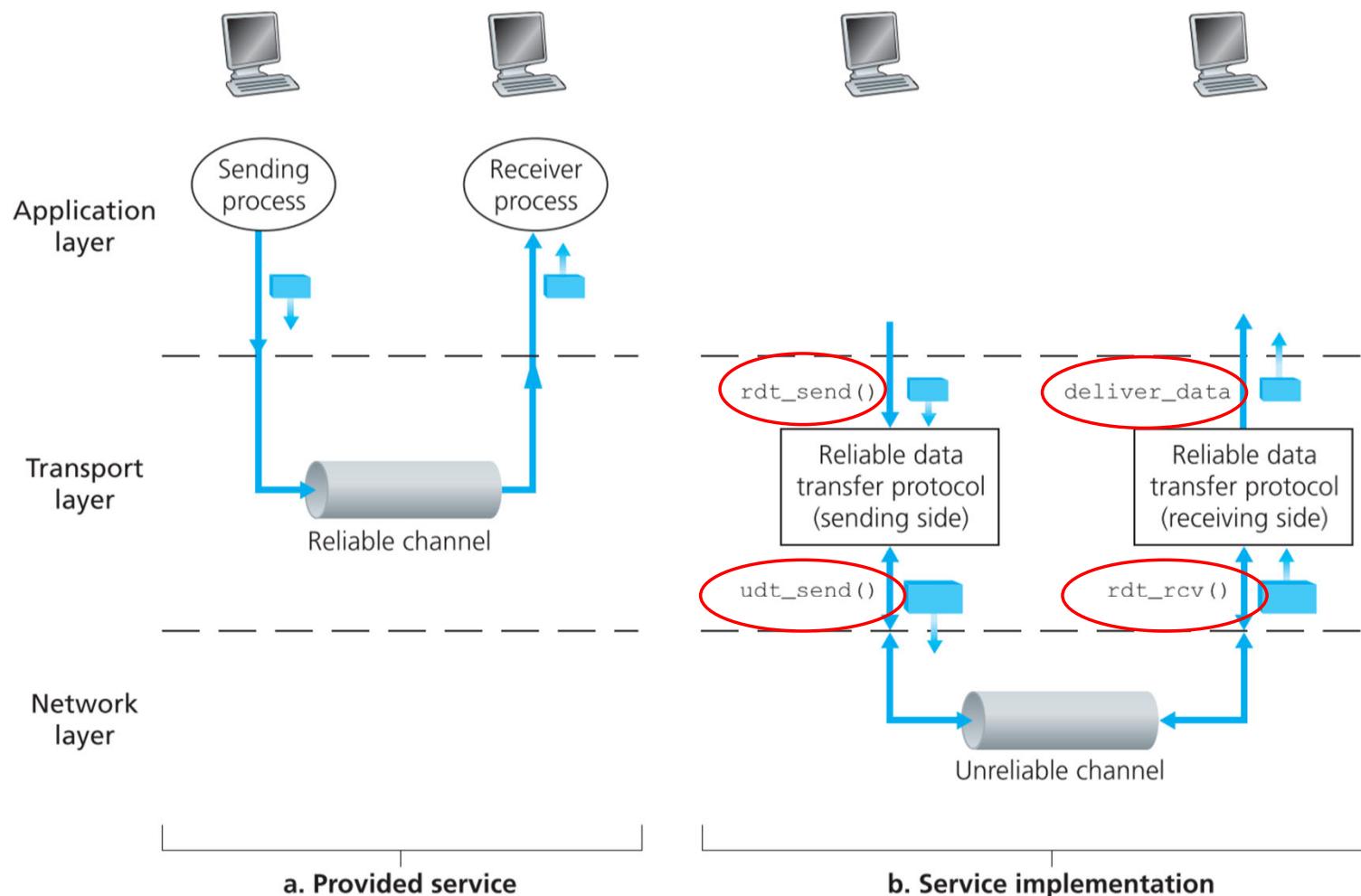


Tilstandsmaskiner

- Generel metode til at modellere en system komponents "opførsel"
 - Dens tilstande og tilstandsskift
 - Dens interaktion m. andre komponenter
- Forskellige varianter
 - "Automater"
 - Moore-maskiner
 - Mealy maskiner
- Anwendelser i andre fag
 - OO-Design
 - Syntax & Semantik (fx, genkendelse af reg-exp)
 - Compilere
 - Her protokol modellering
 - Formel Verifikation
- En graf, hvor knuder og kanter tillægges en tilstandsfortolkning

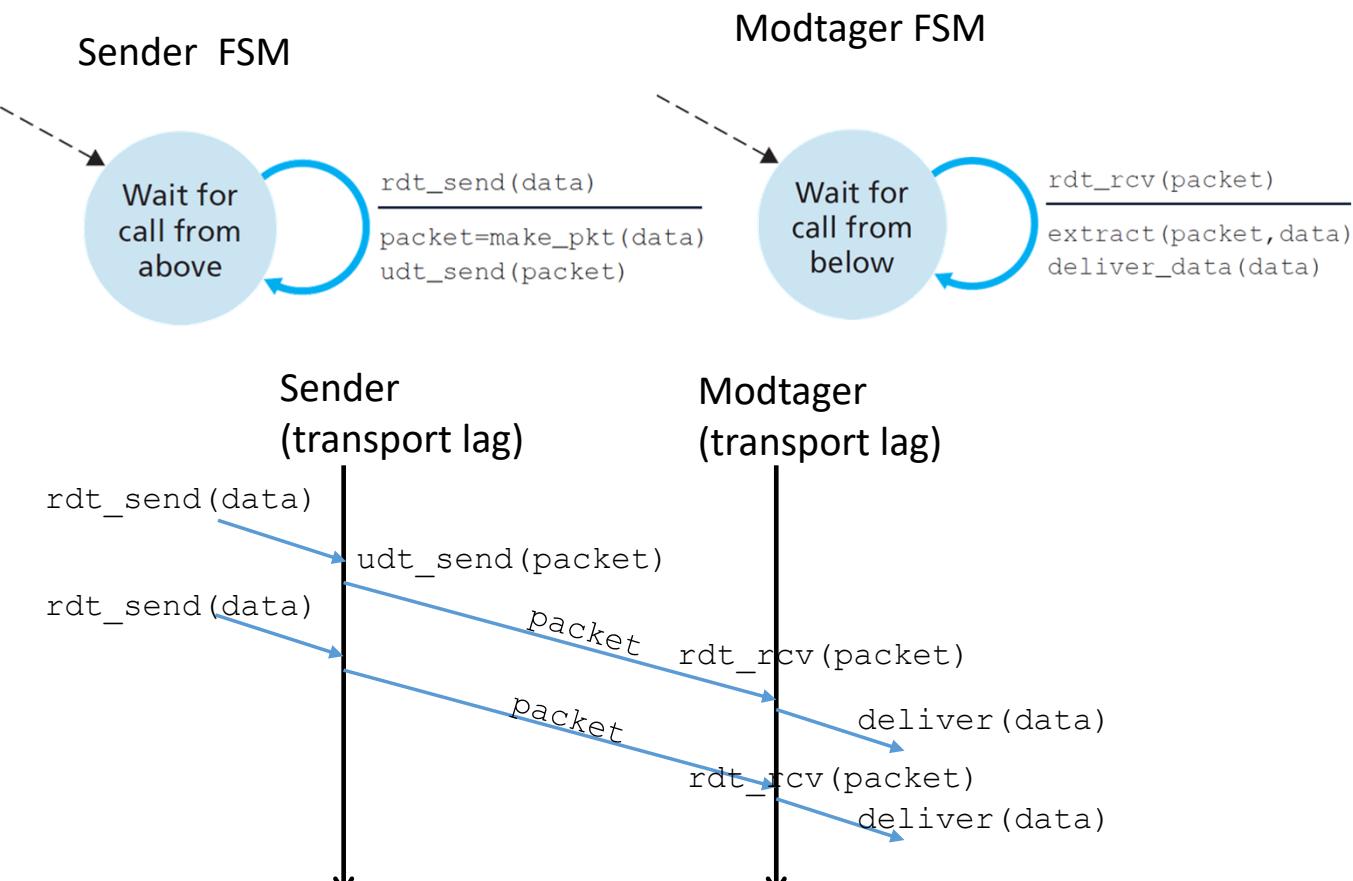


Abstraktion og Implementation



Pålidelig kanal: rdt1.0

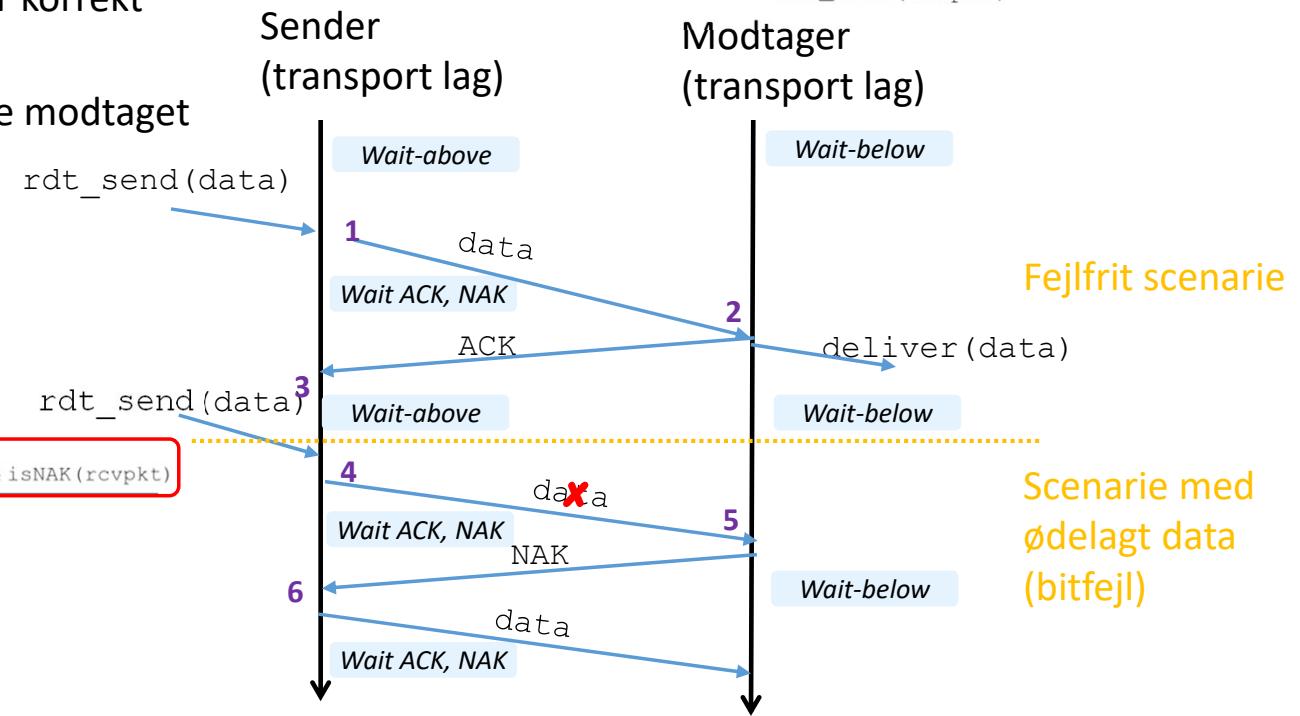
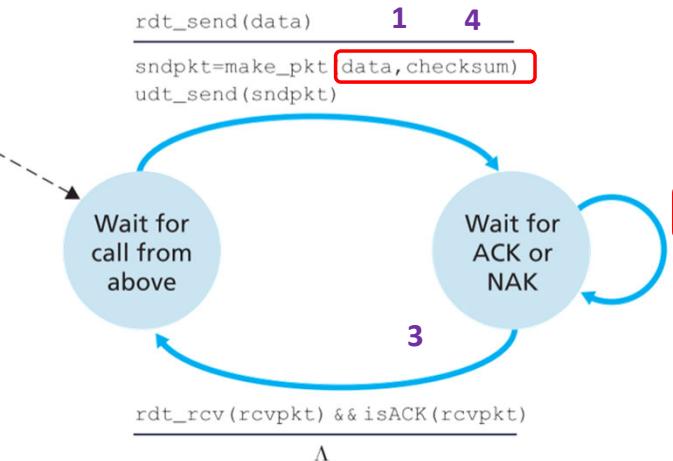
- Antagelser:
 - Kanalen er tabsfri
 - Pakker modtages fejlfri: Ingen bit-fejl
 - Bevarer rækkefølge
- Trivial!
- Sender og modtager følger hver sin tilstandsmaskine



Sekvensdiagram (message sequence chart) viser ét muligt scenarie

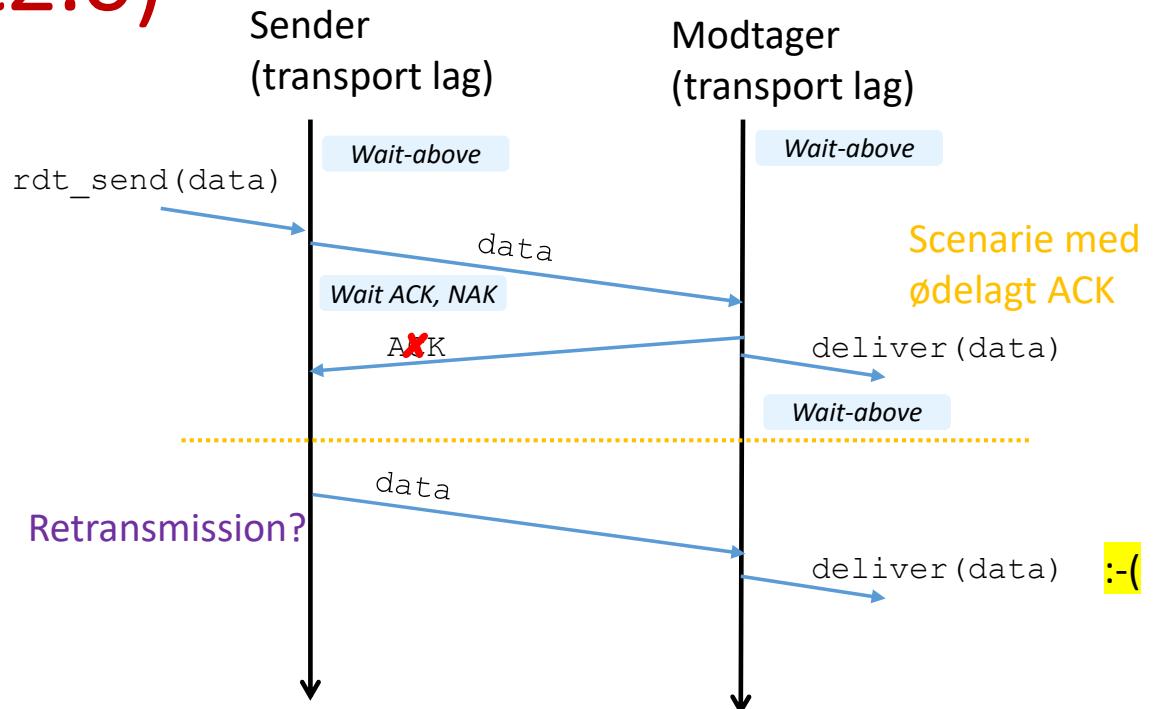
Kanal med bitfejl (rdt2.0)

- Antagelser:
 - Bitfejl kan forekomme (pakken kan ikke forstås)
 - Kan detekteres vha. checksum
 - Bevarer rækkefølge
 - **ACK** (acknowledge): meddelelsen kvitterer for korrekt modtagelse
 - **NAK** (negativ acknowledge): meddelelsen ikke modtaget korrekt \Rightarrow retransmission

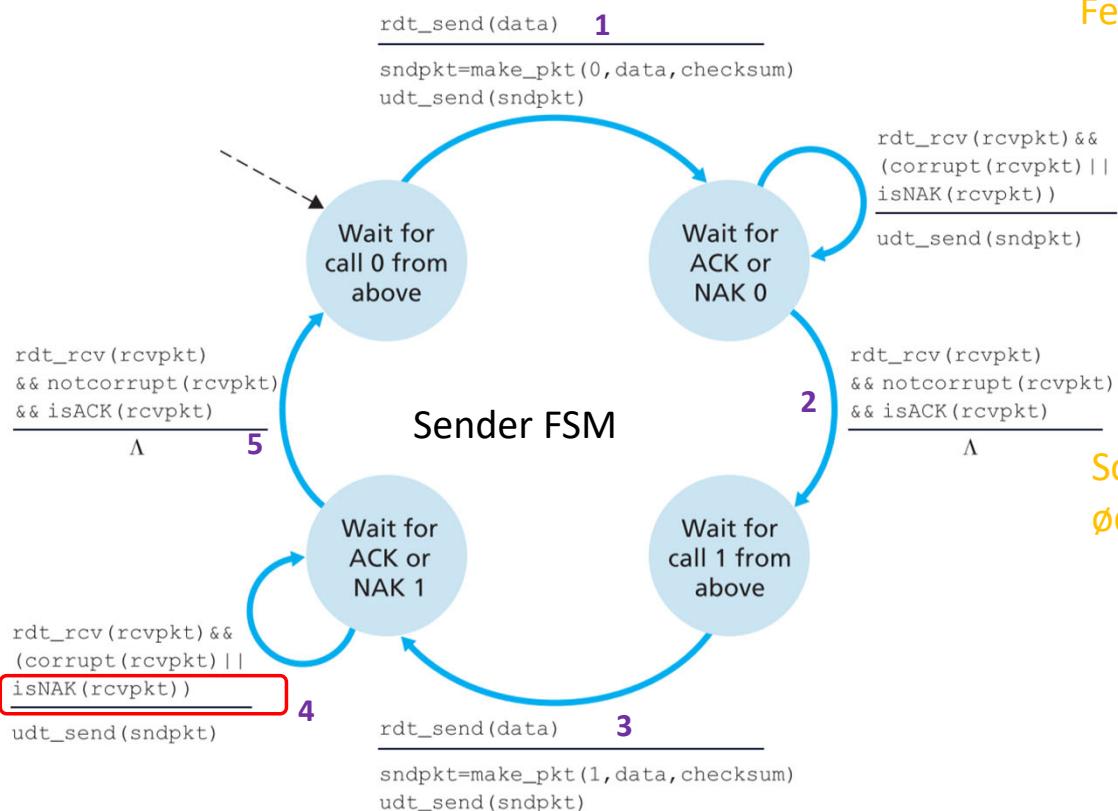


Kanal med bitfejl (rdt2.0)

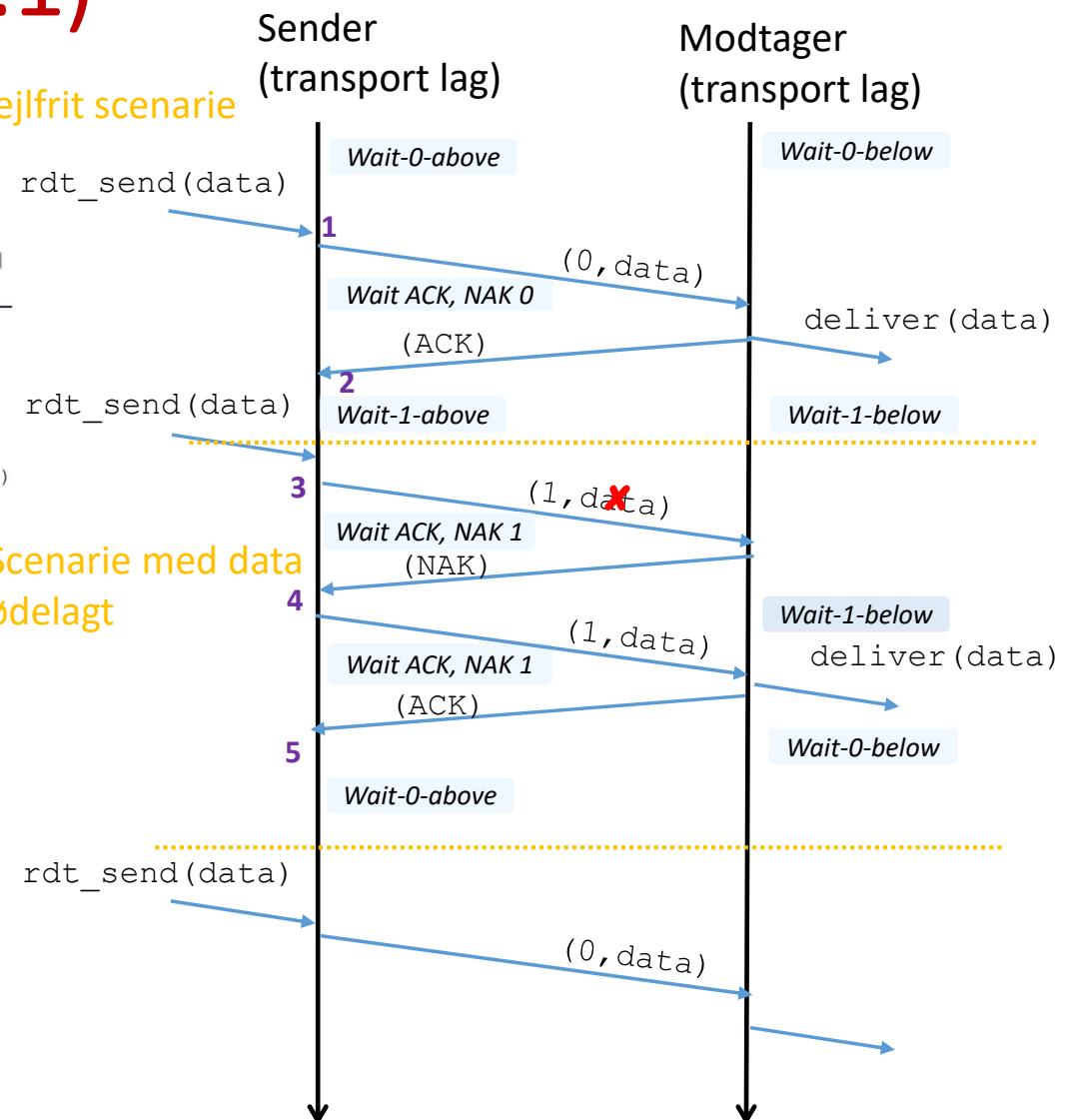
- Antagelser:
 - Bitfejl kan forekomme (pakken kan ikke forstås)
 - Kan detekteres vha. checksum
 - Bevarer rækkefølge
- **ACK/NAK** (kan selvfølgelig også rammes af bit-fejl og tabes)
 - ACK eller NAK?
 - Hænger fast i Wait ACK, NAK
 - Rtd2.0 duer ikke
- Hmm, sender må formode det værste (NAK) og gensende pakken?
 - Modtager leverer så samme data 2 gange (ved ikke om der er ny data eller gendsendt data).
 - ⇒ Brug sekvensnumre



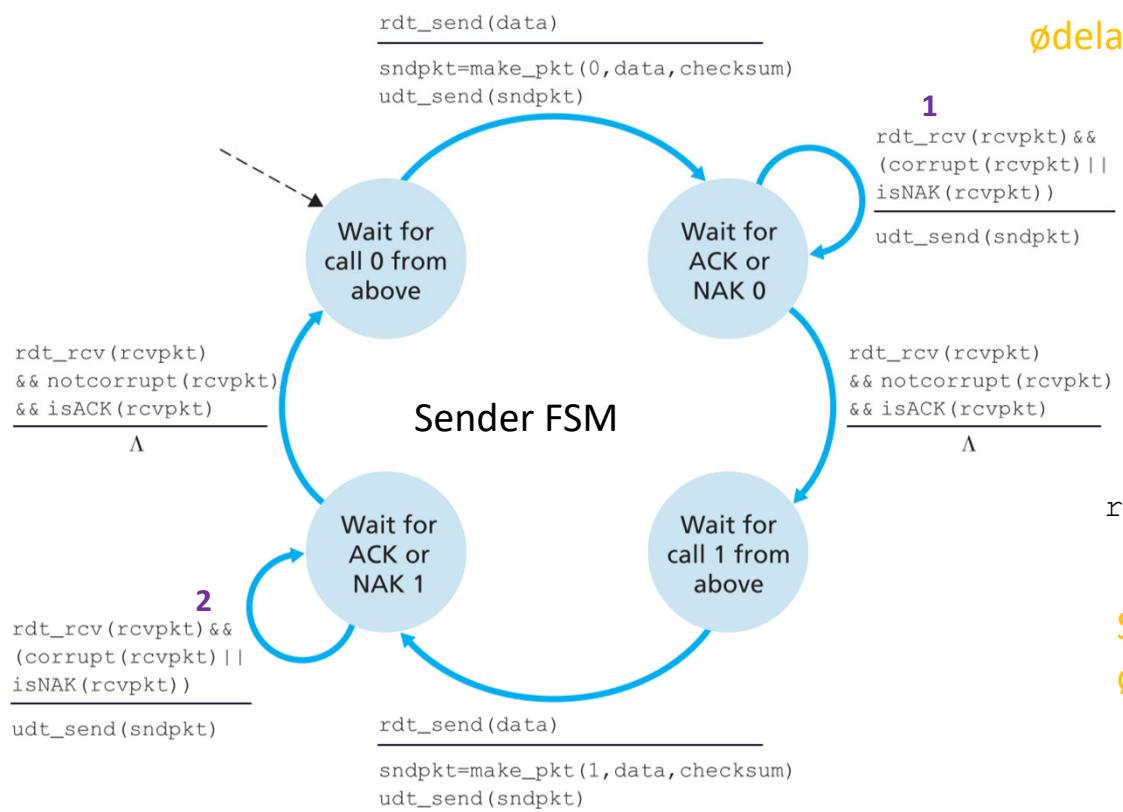
Sekvens-numre 1 (rdt2.1)



Fejlfrigt scenarie



Sekvens-numre 2 (rdt2.1)



Scenarie med
ødelagt ack

`rdt_send(data)`

`rdt_recv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))`
`udt_send(sndpkt)`

`rdt_recv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)`
`Λ`

`Λ`

`rdt_send(data)`

Sender FSM

Scenarie med
ødelagt NAK

`rdt_send(data)`

`sndpkt=make_pkt(1,data,checksum)`
`udt_send(sndpkt)`

`rdt_recv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))`
`udt_send(sndpkt)`

Sender
(transport lag)

Modtager
(transport lag)

`rdt_send(data)`

`Wait-0-above`

`Wait ACK, NAK 0`

`(0, data)`

`Wait ACK, NAK 0`

`(ACK)`

`Wait-1-above`

`rdt_send(data)`

`Wait ACK, NAK 1`

`(NAK)`

`Wait ACK, NAK 1`

`(ACK)`

`Wait-0-below`

`Wait-0-below`

`deliver(data)`

`Wait-1-below`

`Ignorer dublet`

`Wait-1-below`

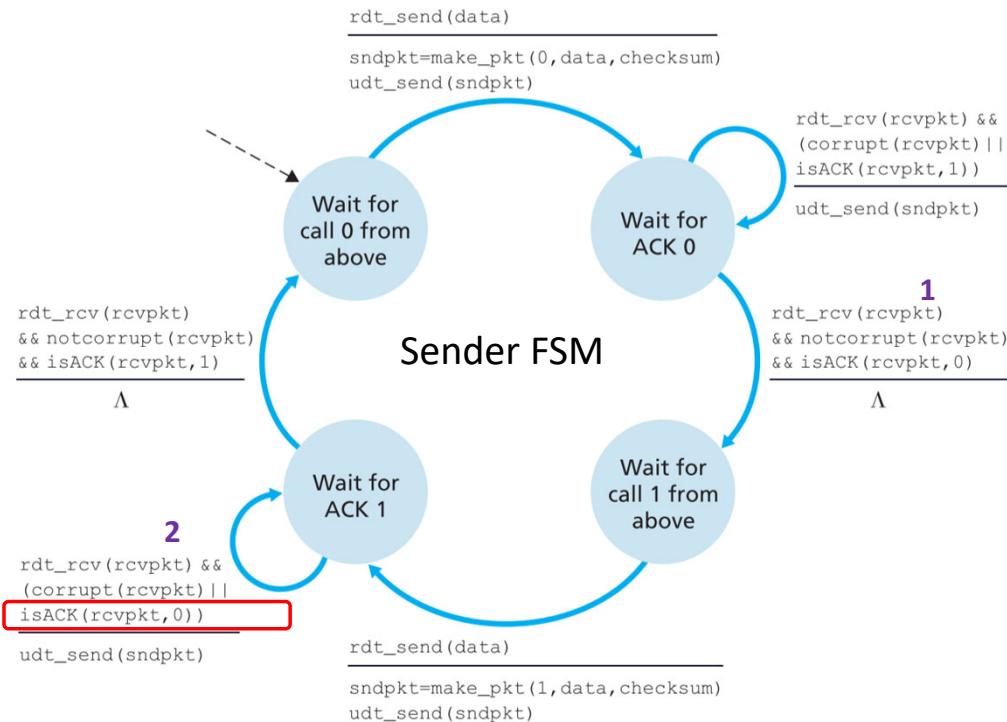
`Wait-1-below`

`deliver(data)`

`Wait-0-below`

- Tabt ACK, NAK har samme effekt: retransmission
- Kun ACK med forventet sekvensnr, bringer protokollen frem
- ⇒Kan vi klare os med kun én af ACK, NAK?

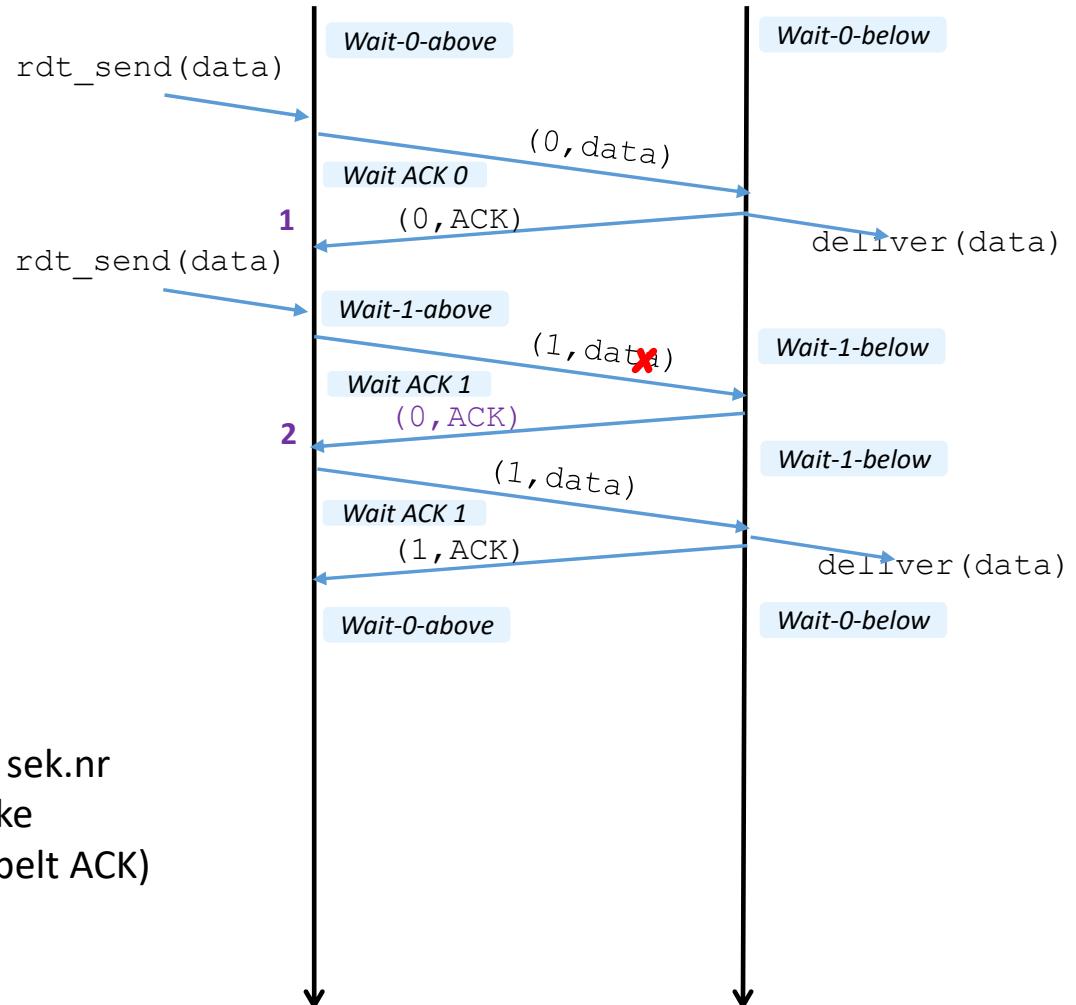
Sekvens-numre og kun ACK (rdt2.2)



Scenarie med ødelagt data

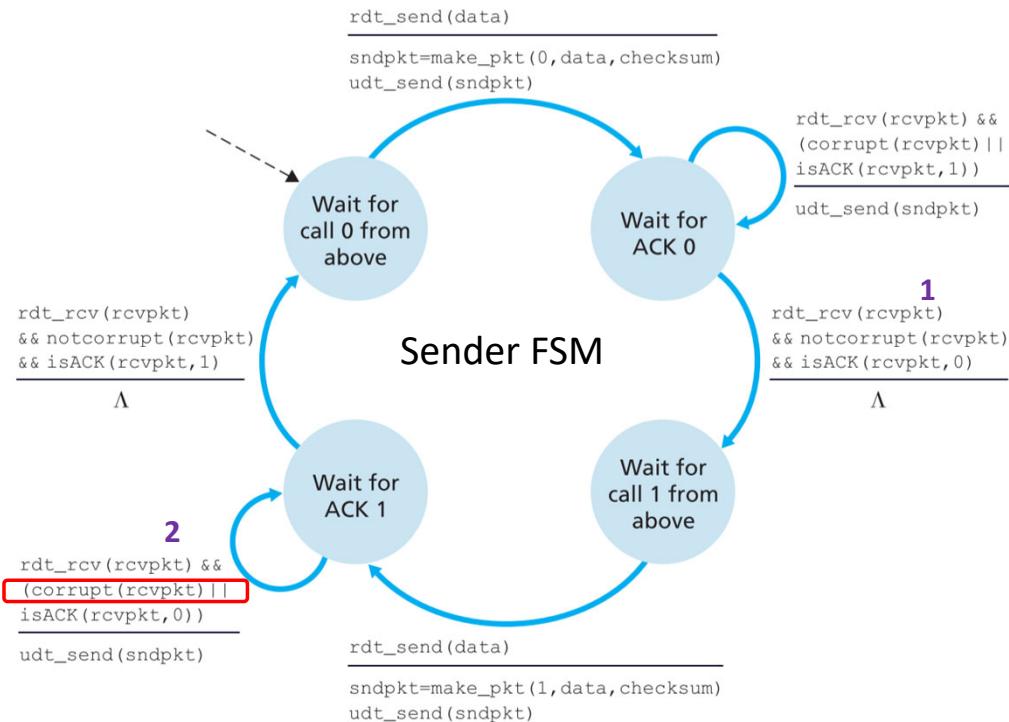
Sender
(transport lag)

Modtager
(transport lag)



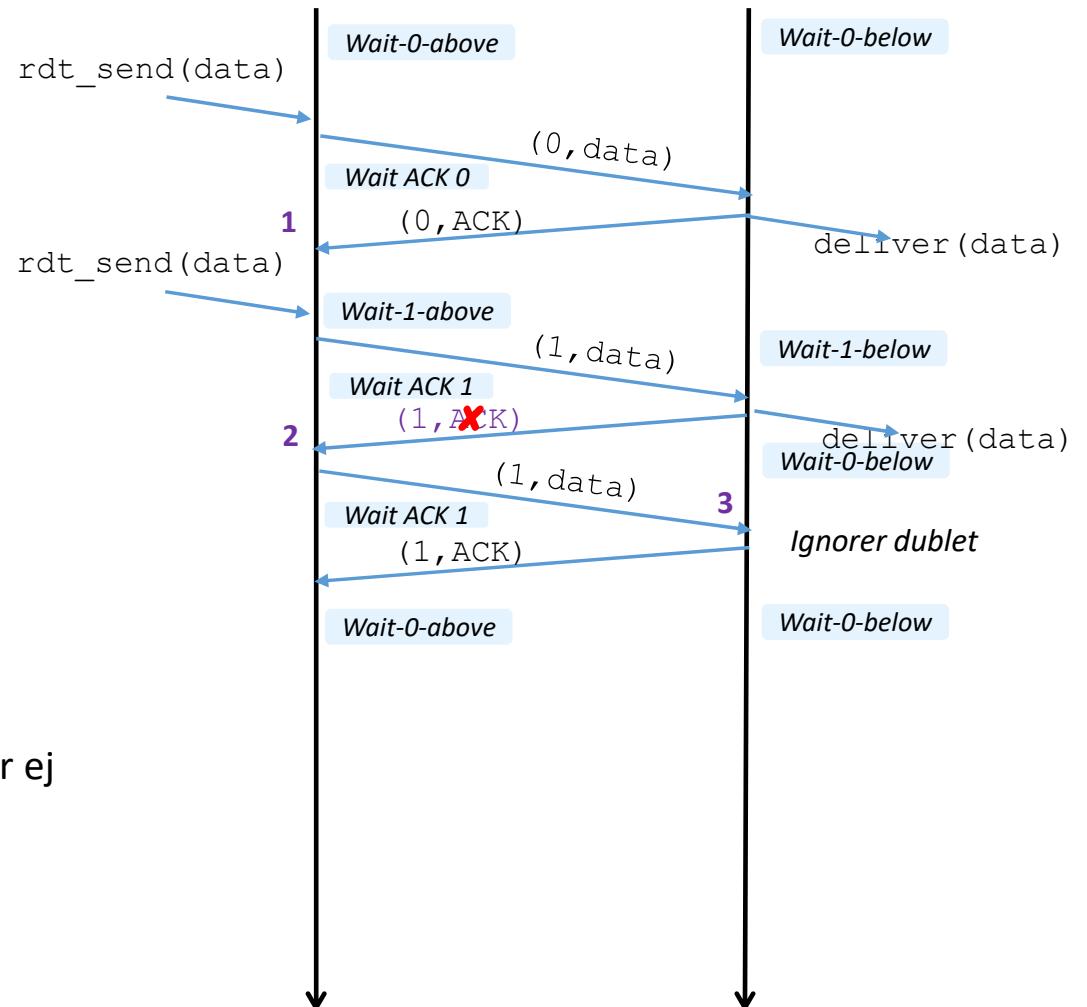
- Ved tabt data, kvitterer modtager med seneste korrekte sek.nr
- ACK inkluderer sekvensnr på den korrekt modtagne pakke
- Sender, der modtager ACK med uventet sekvensnr (dobbelt ACK)
- ⇒ retransmission

Sekvens-numre og kun ACK (rdt2.2)



Scenarie med ødelagt ACK

Sender
(transport lag)
Modtager
(transport lag)

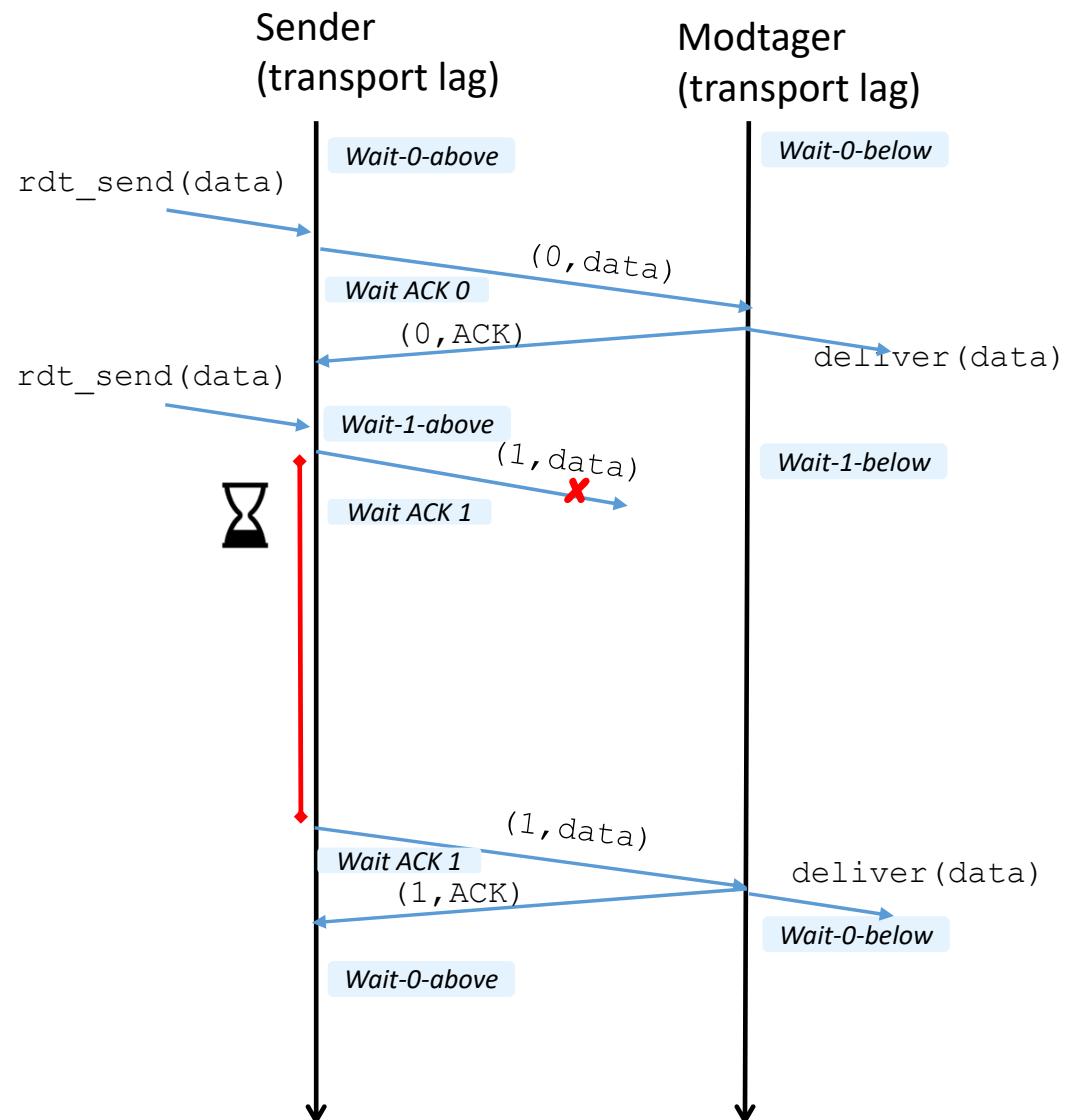


- Ved tabt ACK, ved sender ikke om data er modtager eller ej
- ⇒ retransmission
- 3: modtager duplet: ignorerer data, men gensend ACK

Pakketab 1: rtd3.0

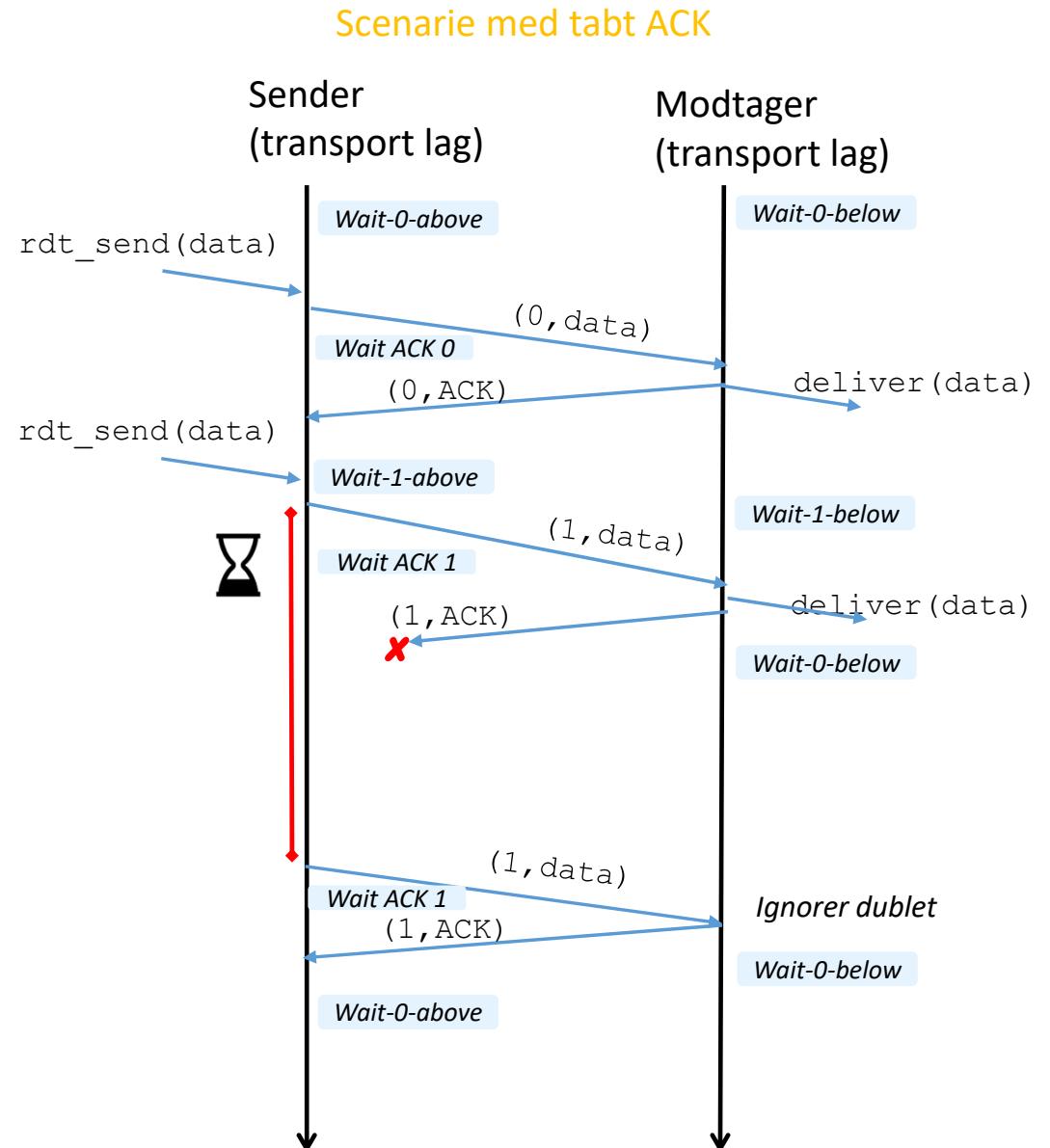
- Antagelser:
 - Kanalen kan **tabe pakker**
 - Mulighed for bit fejl
 - Bevarer rækkefølge
- Sender afventer "rimelig tid" på et ACK, ellers formodes data tabt.
- \Rightarrow retransmission

Scenarie med tabt data



Pakketab 2: rtd3.0

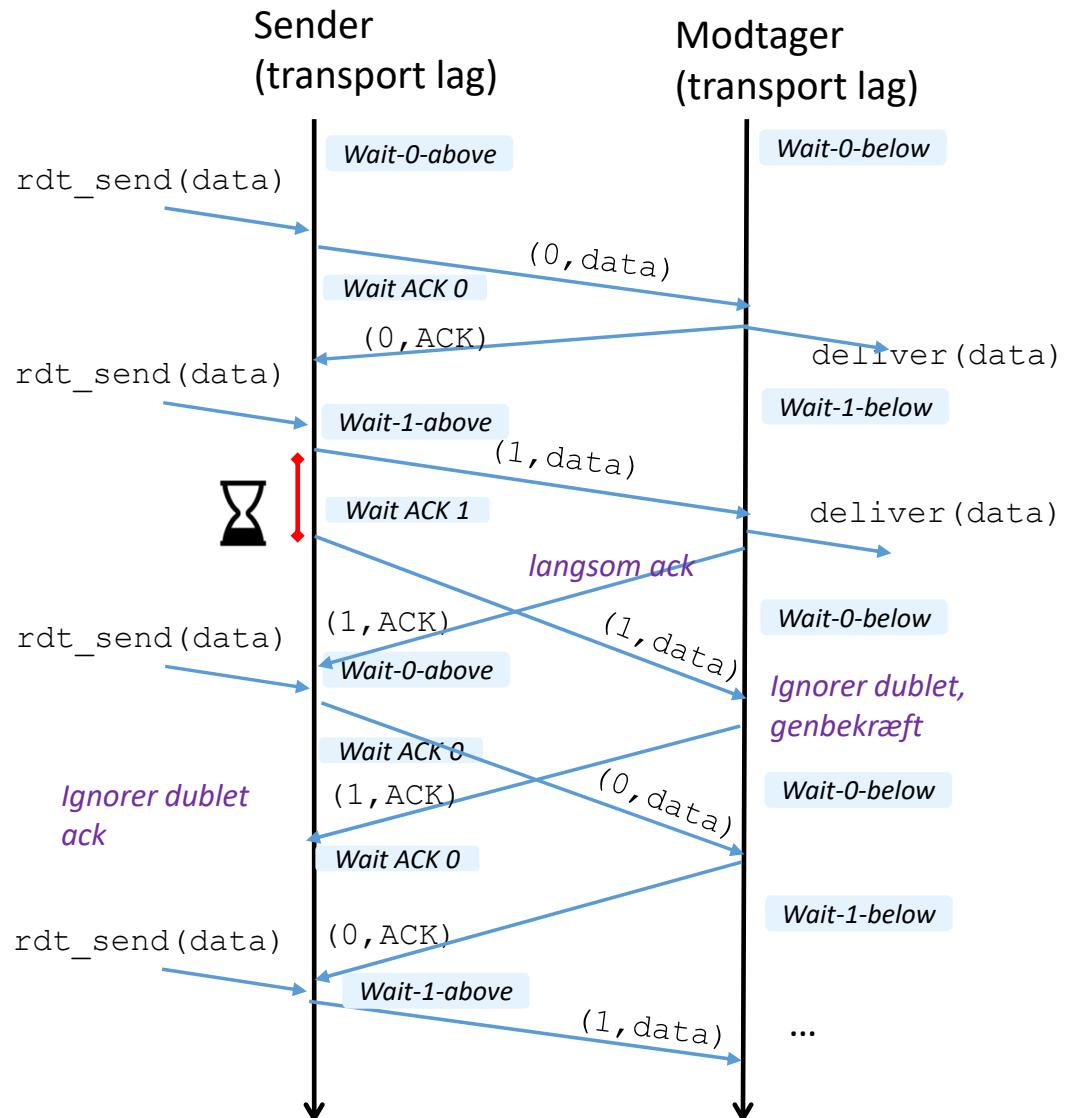
- Antagelser:
 - Kanalen kan tage pakker
 - Mulighed for bit fejl
 - Bevarer rækkefølge
- Sender afventer "rimelig tid" på et ACK, ellers formodes data tabt.
- \Rightarrow retransmission



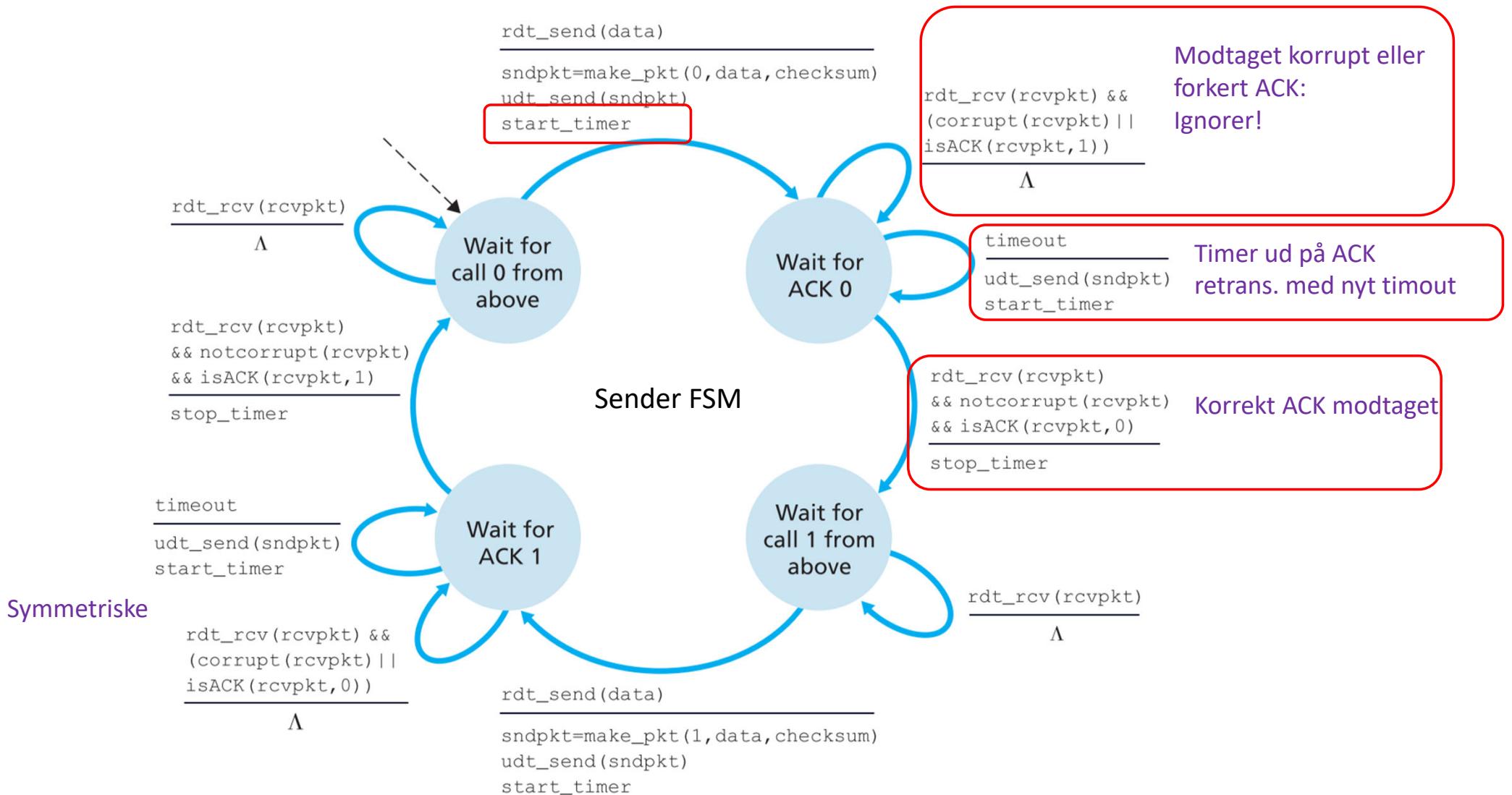
Pakketab 3: rtd3.0

- Hvordan bestemmes "timeout" tid?
 - For lang: Unødvendig langsom
 - For kort: unødvendig retransmission af data og ACK
- Estimeres ud fra RTT
 - Varierer dynamisk efter netværksbelastning
 - Finde god timeout værdi, men kan aldrig undgå for tidlig / for langsom timeout

Scenarie med for lille timeout

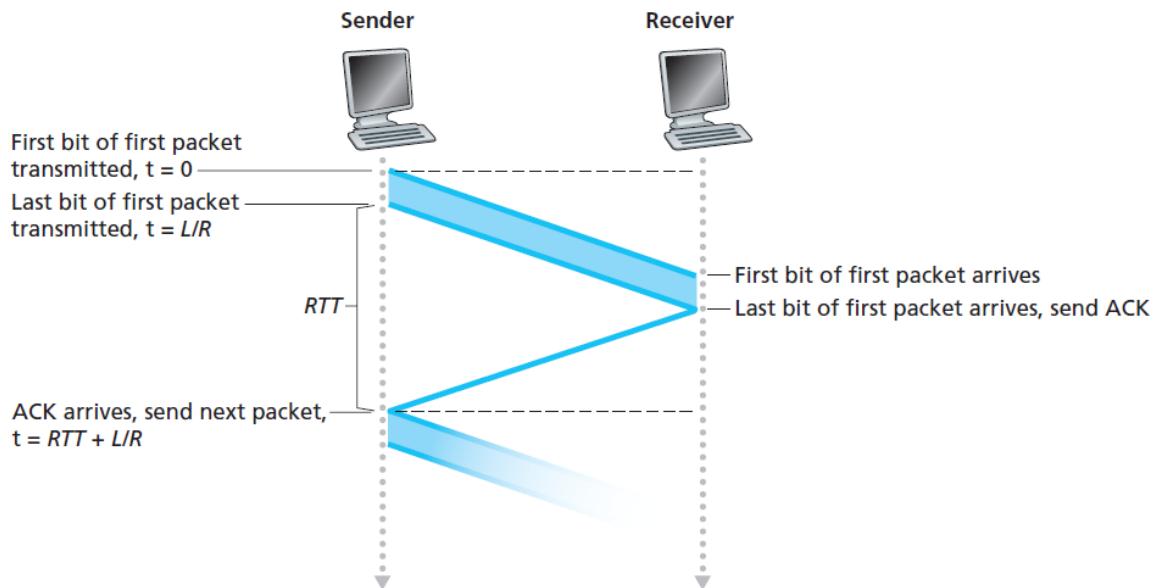


Rtd 3.0



Rtd3.0

- "Den alternerende bit-protokol"
- Mange scenarier!
 - Protokol test og verifikation?
- En passende detaljeret og præcis FSM kan "nemt" laves om til et program m. event-drevet programmering
- Stop&Wait
- Korrekt! Men Håbløs langsom!



Eksempel m. trans-US link:
1 Gbps link, 15 ms prop. delay, 8000 bit pakke:

$$U_{\text{sender}} = \frac{\frac{L}{R}}{RTT + \frac{L}{R}} = \frac{.008}{30.008} = 0.00027$$

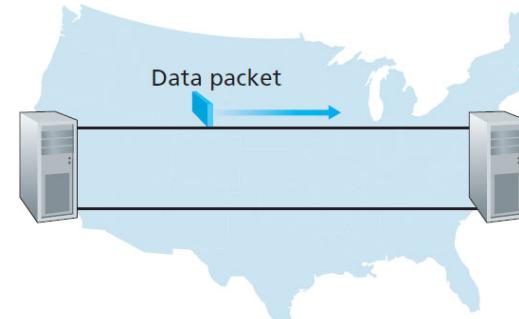
Pipelinede protokoller

Hvordan får vi hurtigt transporteret en masse data korrekt til modtager?

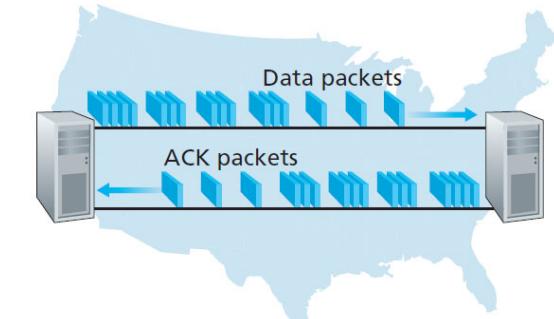
Kan vi undgå en stop&wait ?

Pipelinede protokoller

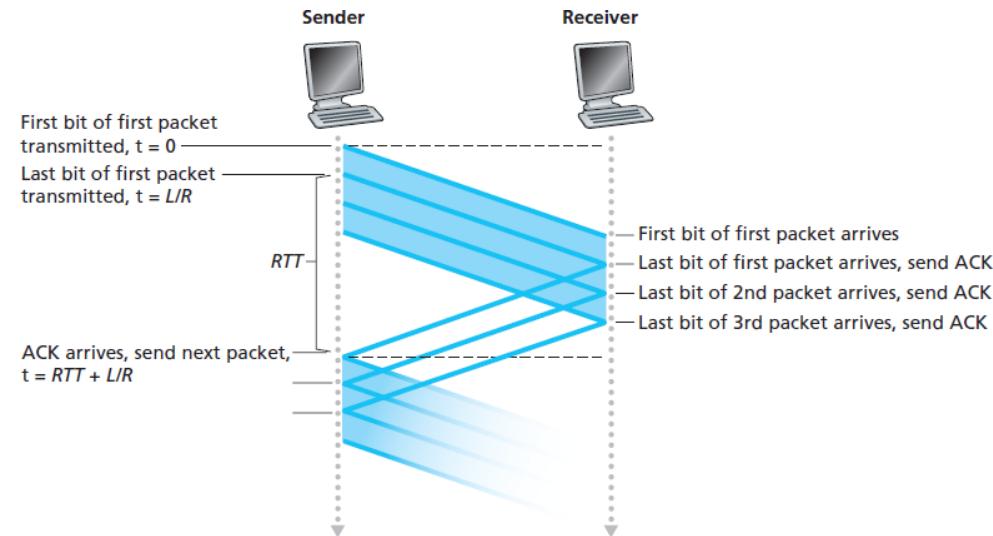
- Øg throughput (bps) ved at udsende flere pakker før vi afventer ACK
 - Udnytte kanalen
 - Uden at oversvømme modtager (flow-kontrol)
 - Uden at overbelaste netværket (congestion-kontrol)
- Pakker gemmes i buffere på sender og modtager til de er leveret
- 2 overordenede strategier for re-transmission
 - Go-Back-N
 - Selective-repeat



a. A stop-and-wait protocol in operation



b. A pipelined protocol in operation

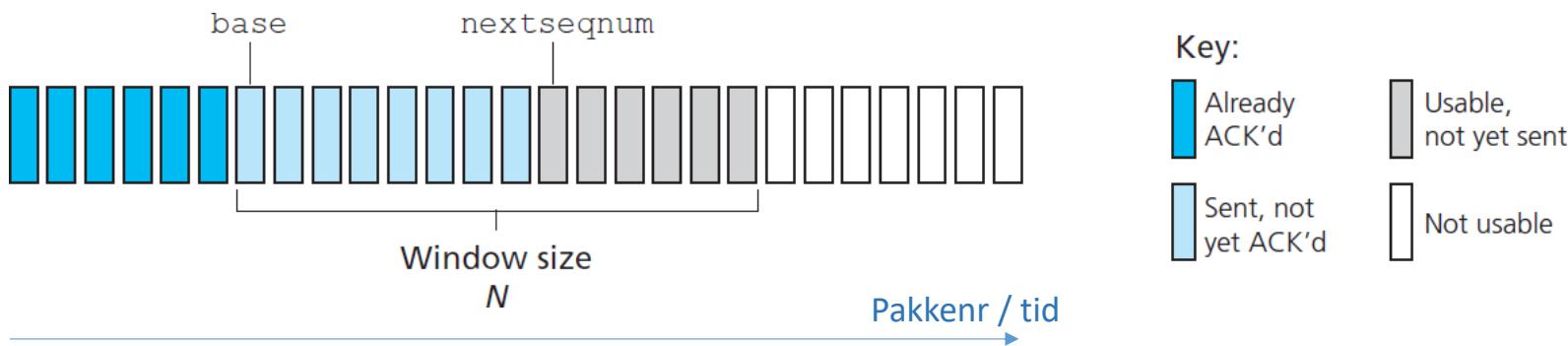


Eksempel m. trans-US link: 1 Gbps link, 15 ms prop. delay

Bits på link (BW-delay produkt) undervejs:
 $(10^9 \text{bps} * 15 * 10^{-3} \text{s}) / 8 \text{ bits/byte} = 1.9 \text{ Mbyte}$

Go-back-N

- Tillad at sender max har max N pakker undervejs i ”pipelinen”
 - Begrænse hvor meget data sender og modtager skal være klar til at bufferere
- Sekvensnummer felt i header med k bits: $[0, 1, \dots, 2^k - 1]$ (fx $k=16 \Rightarrow [0, 65535]$)
- Senders sekvensnumre:
 - **base**: start på aktuelt vindue (ældste sendte ukvitterede pakke)
 - **nextSeqNum**: næste ledige sekvensnummer



- Sender får ACK (n): modtager har fået alle pakker med sekvens numre t.o.m n er korrekt (“kumulativt” ACK)
- Sætter timer for ældste ukvitterede pakke
- Timeout \Rightarrow gensender alle pakker i nuværende vindue, der er sendt efter n
- Vinduet glider en tak frem, hver gang ældste pakke kvitteres

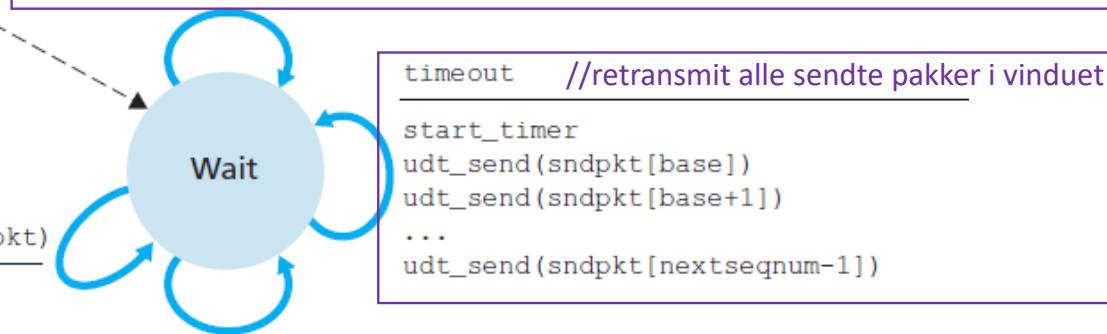
Go-Back-N Sender

Λ
base=1
nextseqnum=1

```
rdt_send(data)
if(nextseqnum<base+N) {      //har vi ledigt sekvensnr i vinduet?
    sndpkt[nextseqnum]=make_pkt(nextseqnum, data, checksum) //gem ny pakke i buffer
    udt_send(sndpkt[nextseqnum])
    if(base==nextseqnum)
        start_timer          //Første pakke (ældst) i vinduet? Sæt timer
    nextseqnum++
}
else
    refuse_data(data)      //app. laget må afvente (OS blokkerer process)
```

Λ
rdt_rcv(rcvpkt) && corrupt(rcvpkt)

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
base=getacknum(rcvpkt)+1   //vi modtog (ack, n): alt t.o.m i modtager OK
If(base==nextseqnum)        //glid vinduet frem til seq. nr. n
    stop_timer
else
    start_timer            //start time for nu ældste pakke
```

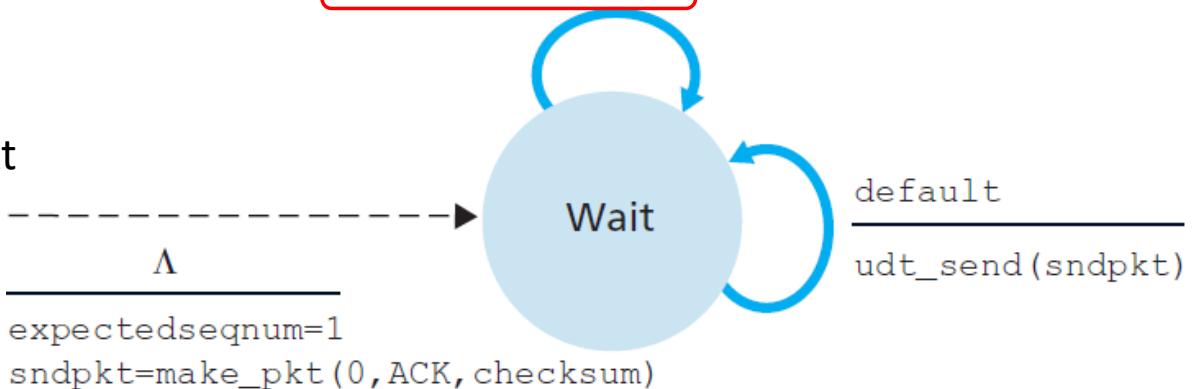


Go-Back-N Modtager

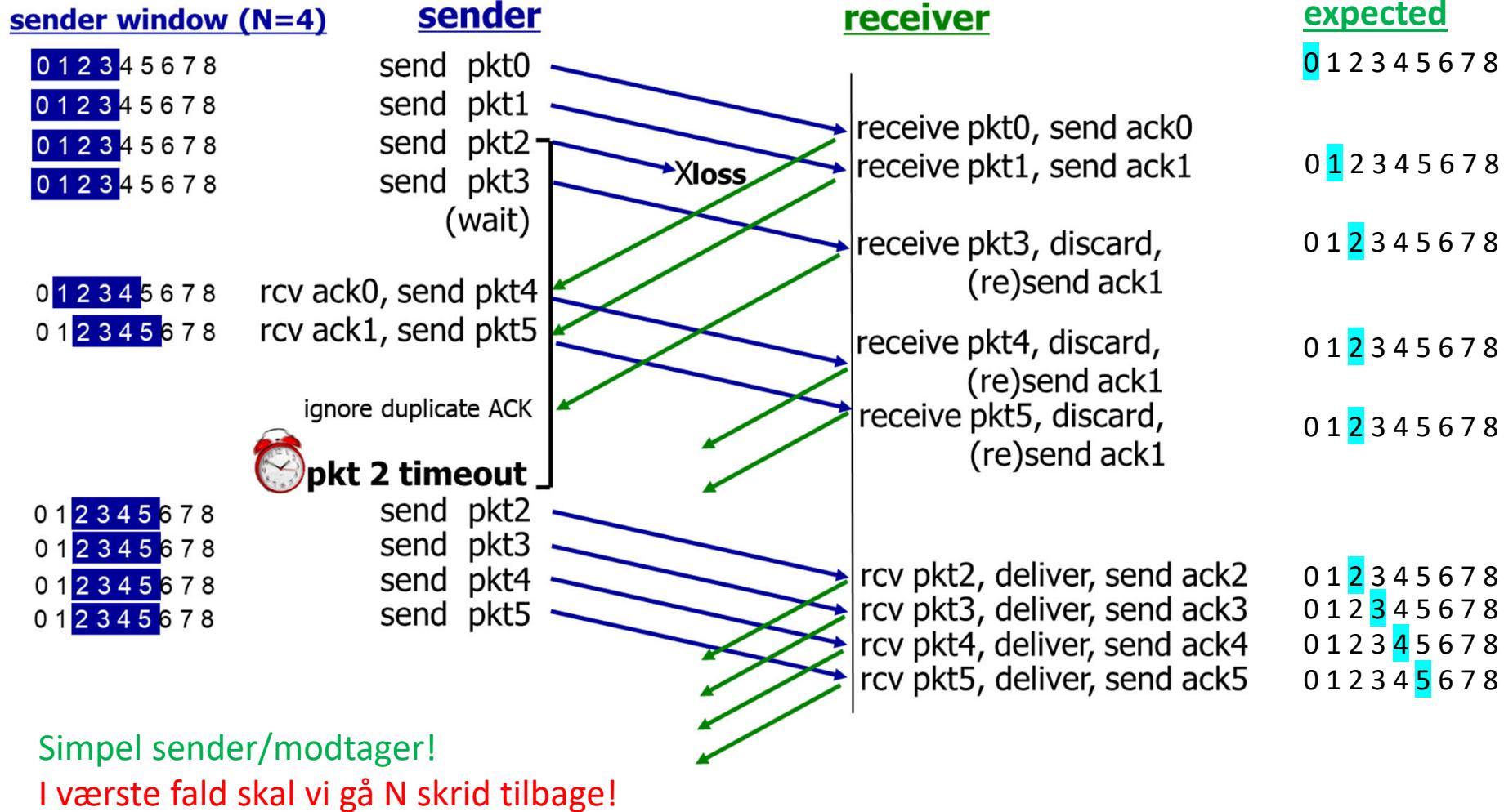
- Forventer at modtage data i stigende sekvens-orden, uden "huller": **in-order**
 - Tæller: **Expectedseqnum**
- Sender ACK for den korrekt modtagne pakke med størst **in-order** seknr.
 - Kan give dublerede ACKs
- Pakker der ankommer "out-of-order"
 - Bortkastes: ingen buffer på modtager siden
 - Gensende ACK med højeste korrekt modtaget sek nr.

```
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt) //expected == modtaget seknr?
&& hasseqnum(rcvpkt, expectedseqnum)
```

```
extract(rcvpkt, data)
deliver_data(data)
sndpkt=make_pkt(expectedseqnum, ACK, checksum)
udt_send(sndpkt)
expectedseqnum++
```



Go-Back-N Scenarie

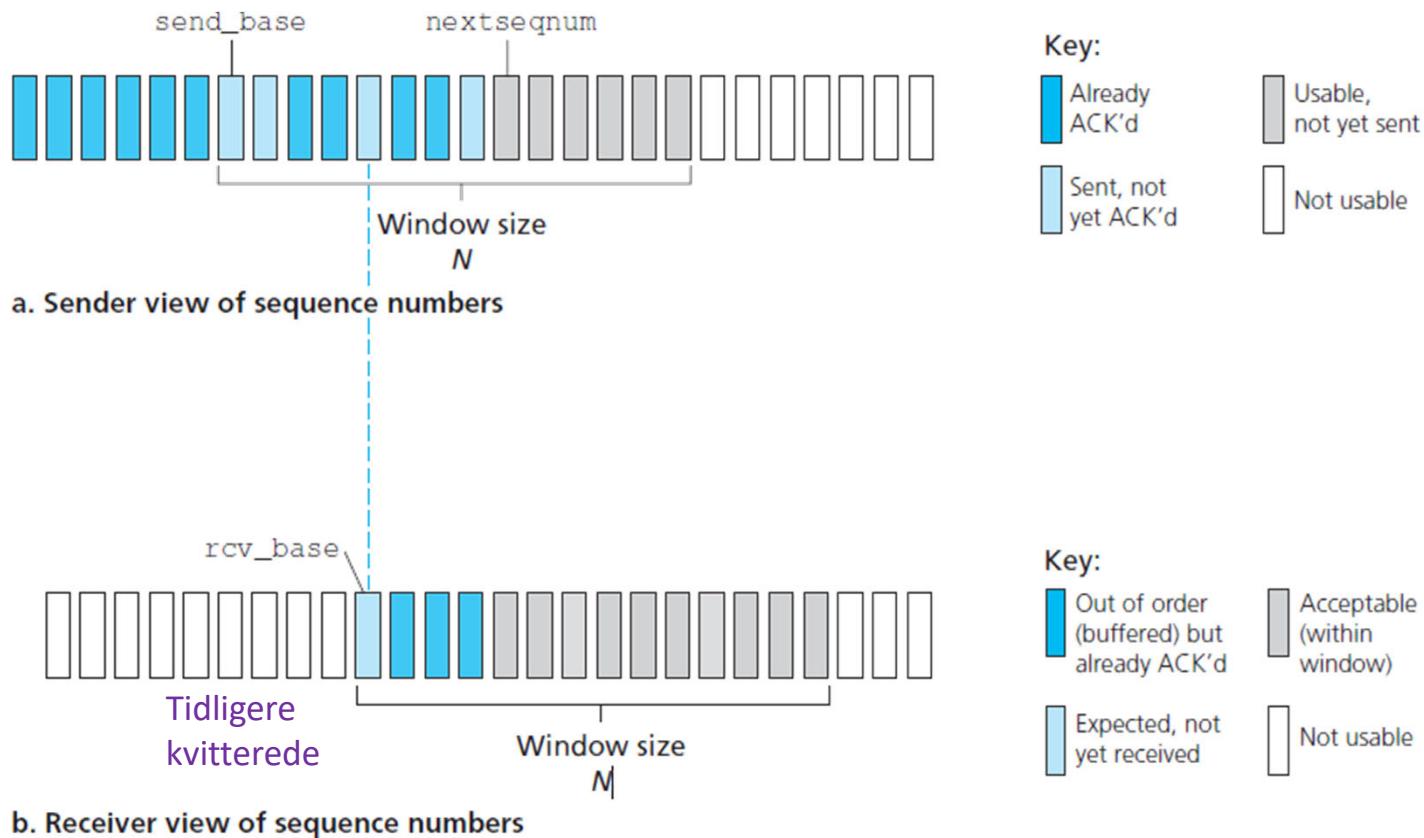


Selective Repeat

- Tillad at sender har max N pakker undervejs i ”pipelen”
- GBN bortkaster korrekte pakker der ankommer out-of-order
- I Selektive Repeat:
 - Modtager gemmer pakker, der ankommer out-of-order i en buffer
 - Modtager kvitterer *individuelt* for hver modtagne pakke
- Sender *retransmitterer kun de pakker der mangler kvittering*
- Sender sætter en timer for hver enkelt pakke den sender

Selective Repeat

- Tillad at sender har max N udestående pakker
 - Bestemt fra ældste ukvitterede pakke
- Senders sekvensnumre:
 - **send_base**: start på aktuelt vindue (ældste ukvitterede pakke)
 - **nextSeqNum**: næste ledige sekvensnummer
- Både sender og modtager vindue!
- Modtagers sekvens nr
 - **rcv_base**: start på aktuelt vindue (ældste forventede pakke)
 - Accepterer at modtage N pakker frem



Selective Repeat

Sender

- Data klar fra app-lag?
 - Hvis ledigt seknr (n) i vinduet, send pakke n .
 - Start timer for pakke n
- Timeout (n):
 - gensend pakke n
 - Genstart timer for pakke n
- ACK (n) i $[sendbase, sendbase+N-1]$:
 - Marker n som modtaget
 - Hvis n var den første ukvitterede, forskyd vinduet frem til næste ukvitterede pakke

Modtager

- Modtaget pakke n i $[rcvbase, rcvbase+N-1]$:?
 - (pakken er indenfor det forventede område)
 - Send ACK n .
 - Gem pakken n i buffer
 - Aflever alle in-order pakker til app laget
 - Skyd vinduet frem til næste forventede pakke
- Modtaget pakke n i $[rcvbase-N, rcvbase-1]$
 - (pakken er tidligere kvitteret/leveret)
 - Gensend ACK n .
- ELLERS
 - Drop pakken

Selective repeat Scenarie

sender window (N=4)

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

sender

send	pkt0
send	pkt1
send	pkt2
send	pkt3
(wait)	

rcv ack0, send pkt4
rcv ack1, send pkt5

pkt 2 timeout
send pkt2

record ack4 arrived
record ack5 arrived

receiver

receive pkt0, send ack0

receive pkt1, send ack1

receive pkt3, buffer,
send ack3

receive pkt4, buffer,
send ack4

receive pkt5, buffer,
send ack5

rcv pkt2; deliver pkt2,
pkt3, pkt4, pkt5; send ack2

Receiver window
expected

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

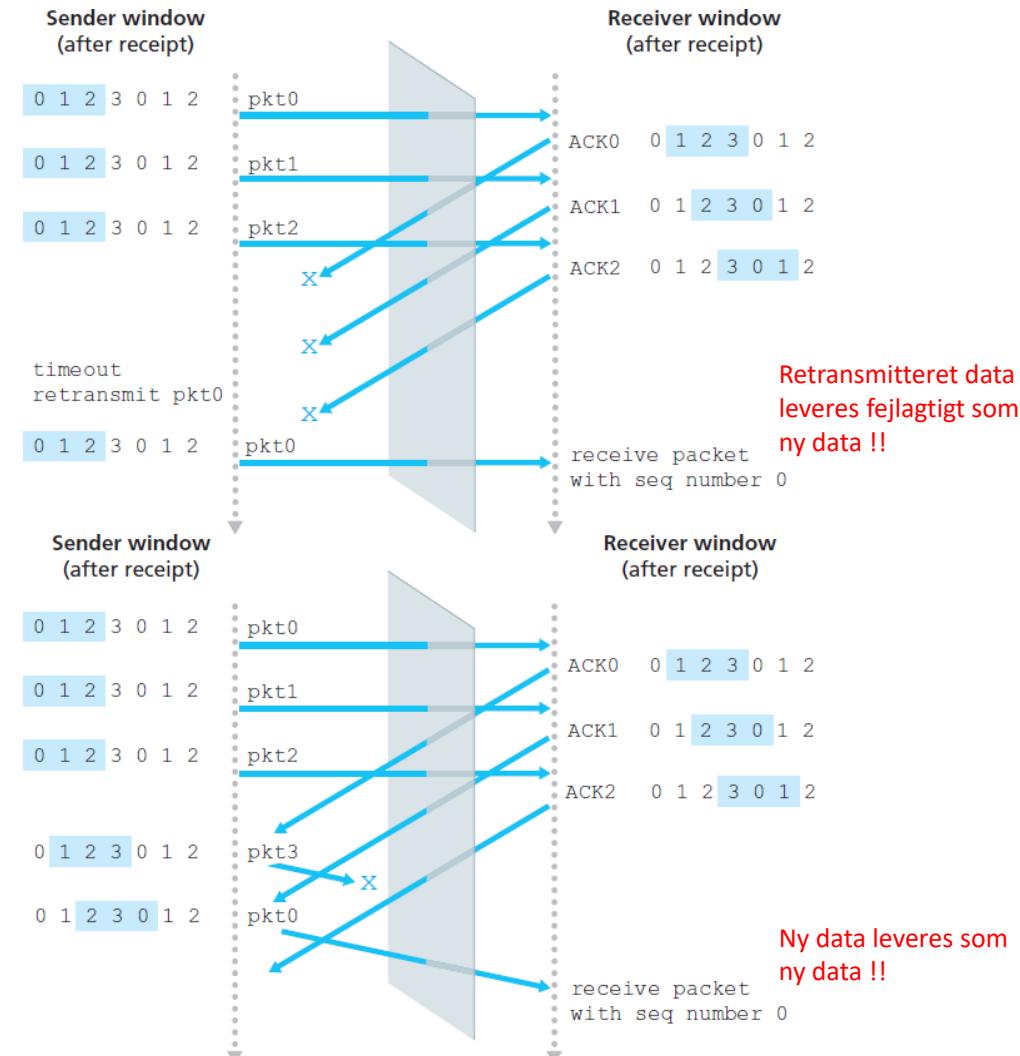
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

Q: what happens when ack2 arrives?

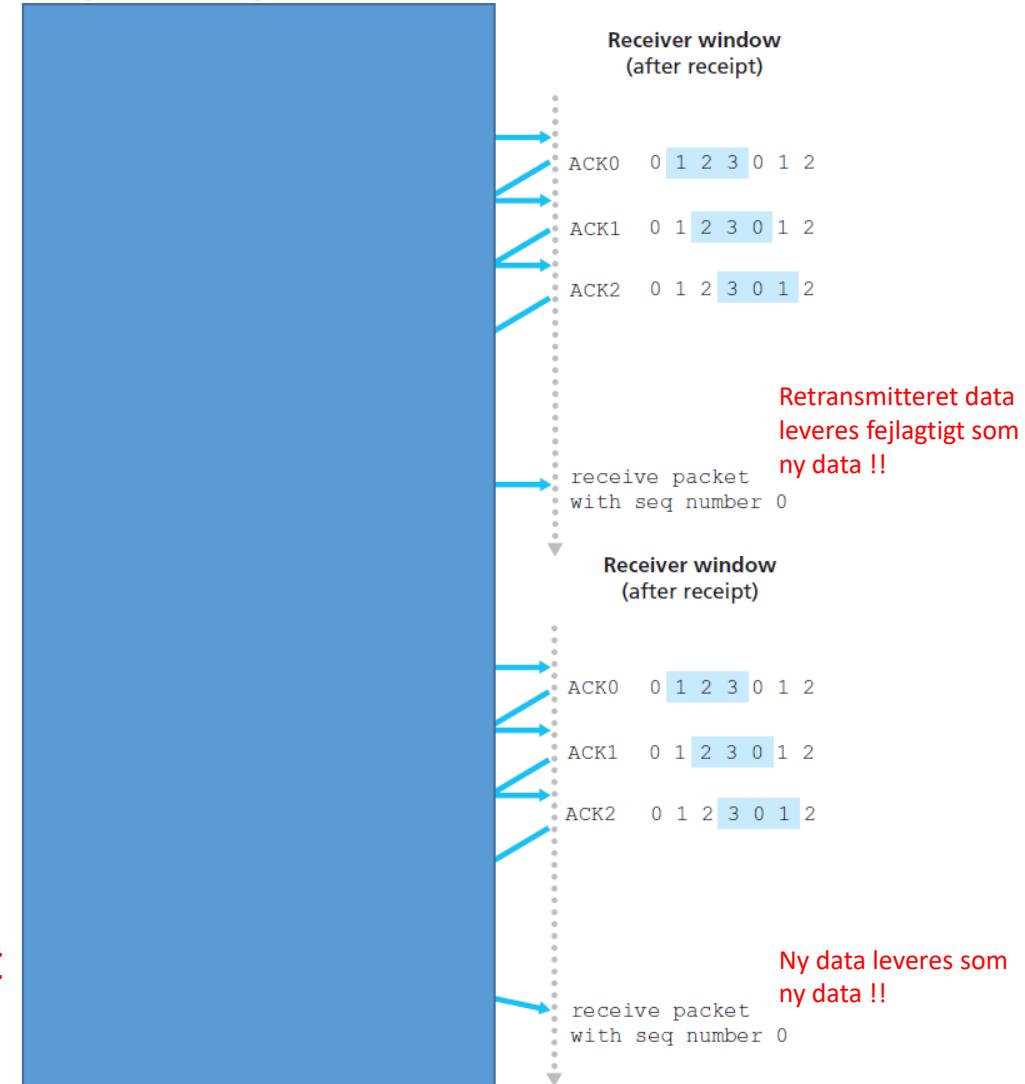
Dilemma omkring sekvensnumre

- Header har kun plads til $2^k - 1$ sekvensnumre
 - Wrap rundt, start forfra med 0
- Antag sekvens numre 0,1,2,3
- N=3
- 2 Scenarier
 - ACK tab
 - Data tab



Dilemma omkring sekvensnumre

- Header har kun plads til $2^k - 1$ sekvensnumre
 - Wrap rundt, start forfra med 0
- Antag sekvens numre 0,1,2,3
- N=3
- 2 Scenarier
 - ACK tab
 - Data tab
- Modtager kan ikke se forskel!
 - Retransmitteret data accepteres som nyt



Udeståender

- Genbrug af sekvens numre?
 - Scenarie b indikerer at: $2N < k^2 - 1$
- Hvad med en kanal som omordner pakker?
 - **Uproblematisch:** Hvis den ikke er for gammel, så fx buffererer selektiv repeat den, og leverer i rækkefølge
 - **Mere problematisk:** En *meget meget* gammel pakke kan have et sekvens nr, der passer ind i modtagers nye vindue: vi leverer forkert data?!
 - På internettet antages at pakker ikke lever ud over en max tid (3 min)
 - IP datagrammer har et "Time to live" (TTL / hop limit) felt som tælles ned hver gang de videresendes
 - NB: i et computer netværk er det problematisk at anvende klokken som tidsstempel, da alle computere har sit eget ur, der ikke kan synkroniseres (præcist.)
- Flow- og Congestion kontrol? Næste lektion!

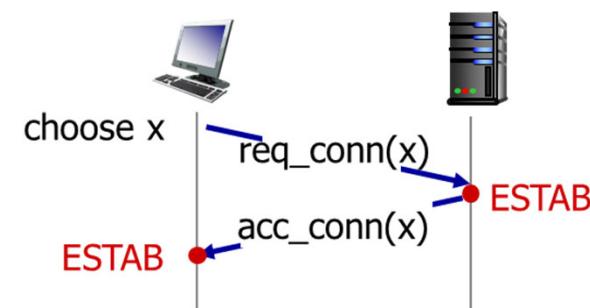
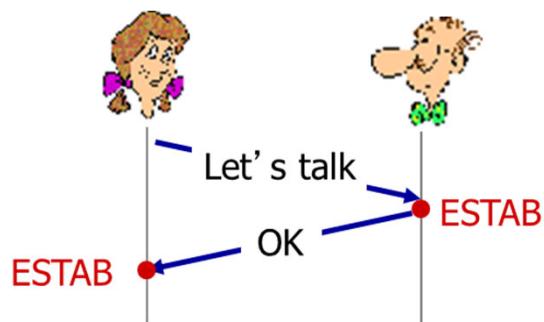
Etablering af TCP forbindelser

Hvordan forbinder en klient sig til en server?

Hvordan stopper de kommunikationen og nedriver forbindelsen igen?

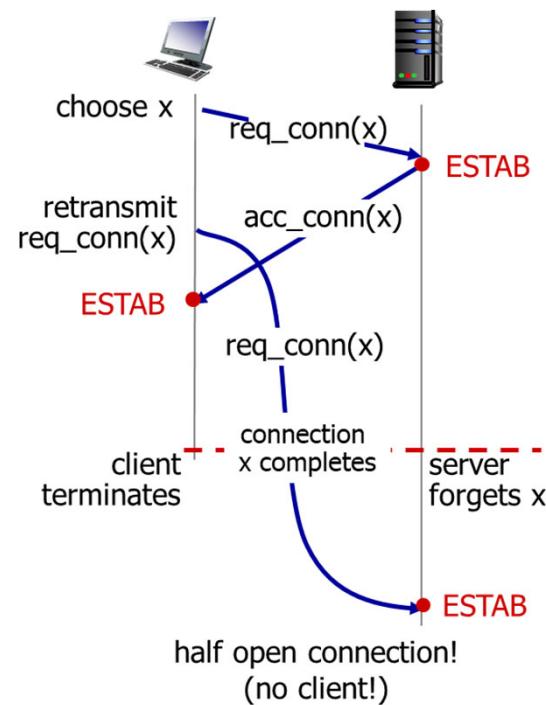
Etablering og nedlægning af forbindelser

- Sender Klient og server skal blive enige om, at de har en forbindelse
 - Afsætte buffer-plads;
 - Initialisere "sliding window" parametre: sekvens nummer + vindues størrelse
 - I begge retninger da TCP er bi-direktionel
- Komplikationer
 - Gamle pakker (fx fra re-transmissioner på tidligere forbindelse) må ikke medgå i ny forbindelse
 - Ved nedlukning: afvente at alt sendt data er leveret til modtager, selv i tilfælde af, at retransmission er nødvendigt.
 - Special pakker til oprettelse og nedlukning af forbindelser kan gå tabt!

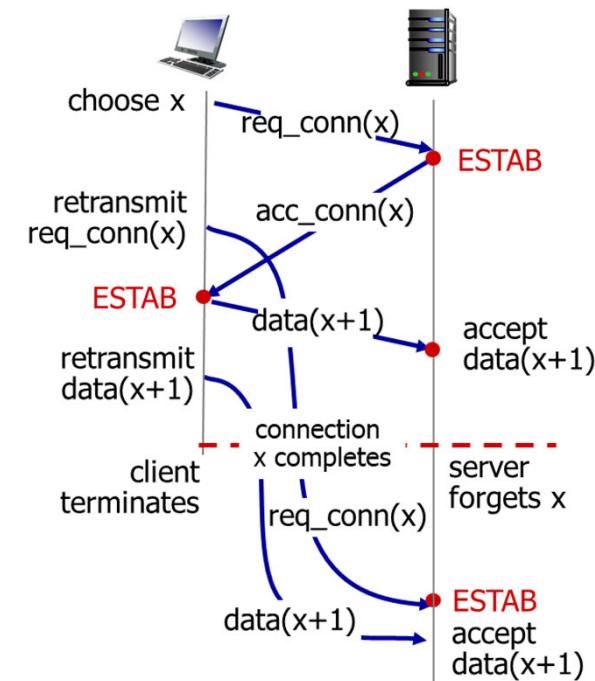


2-vejs handshake ?

- Eksempler på problematiske scenarier



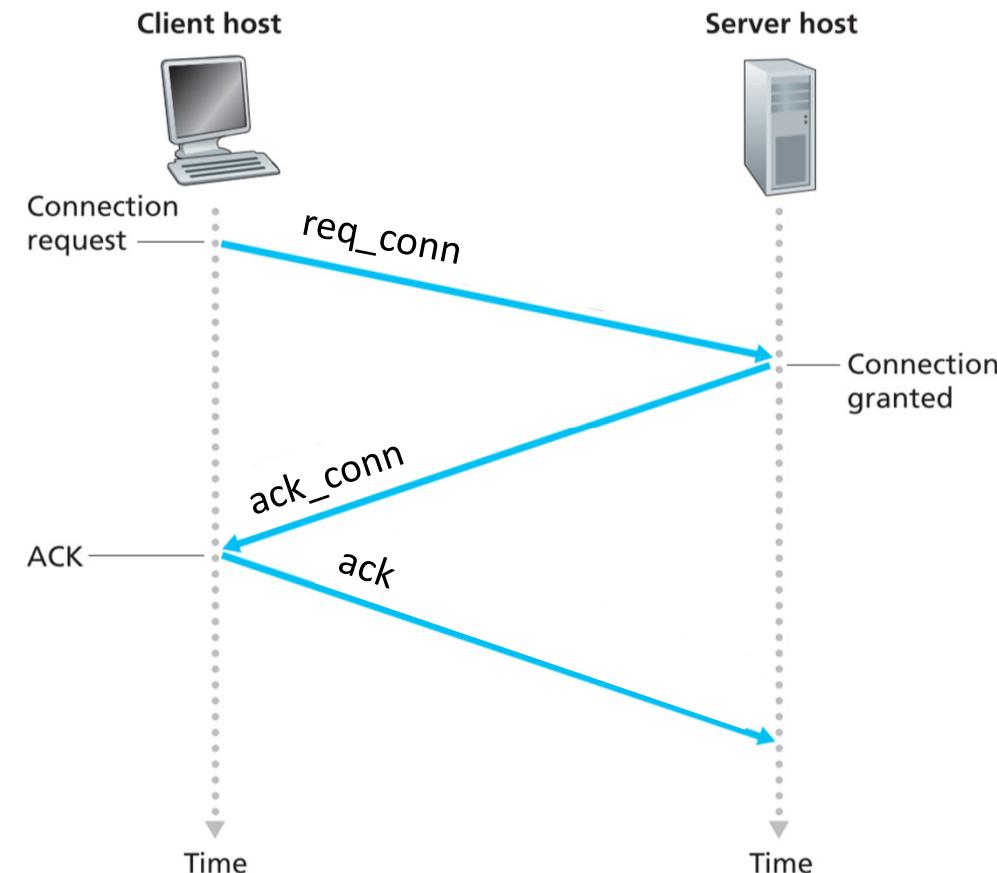
Server afsætter ressourcer til klient,
der ikke findes



Server modtager gammel data.

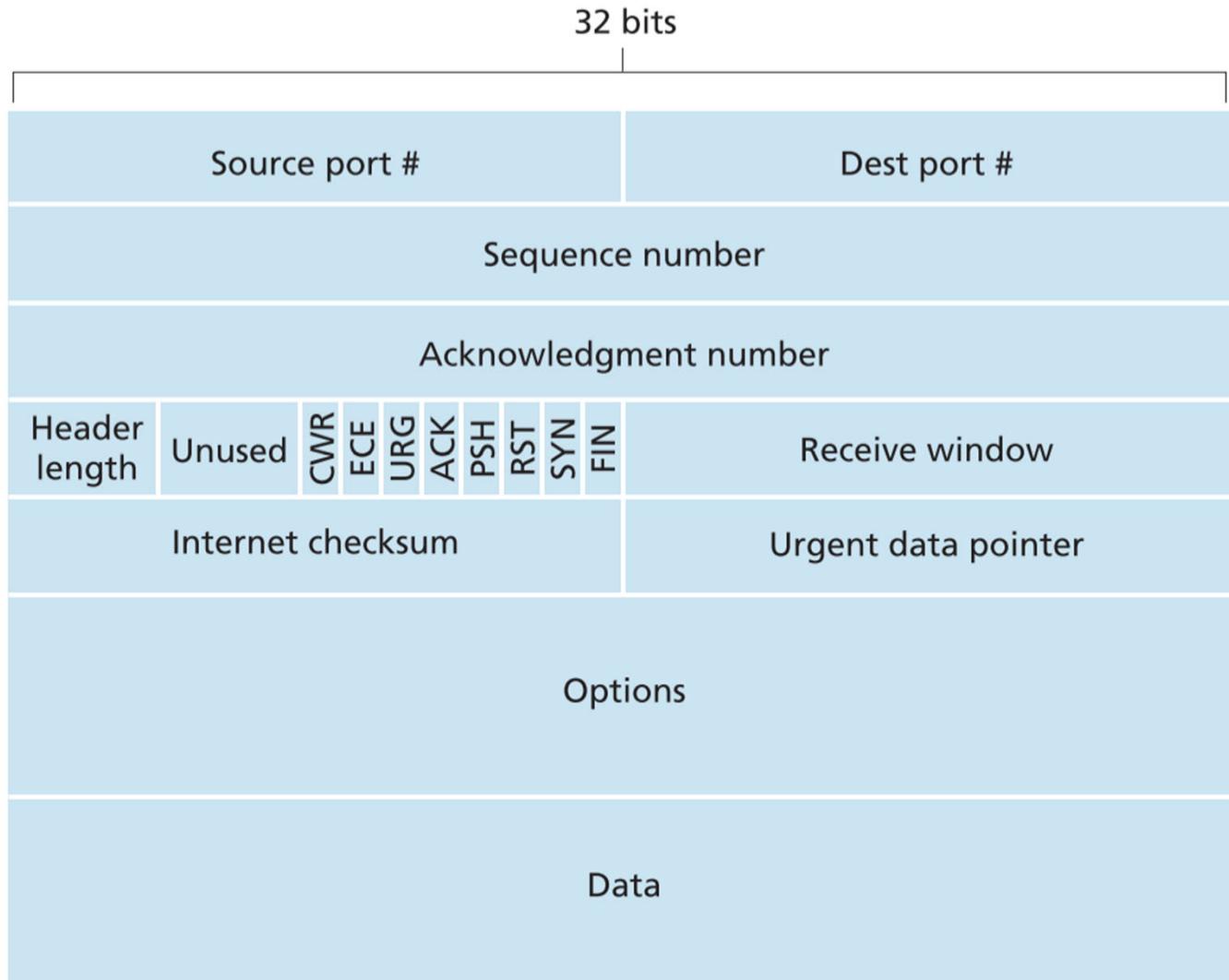
3-way-handshake

- Forudsætning:
 - Server-process lytter på socket (bundet til ønsket port)
 - Klient server process vil forbinde sig til server
- Transport laget foretager et 3-vejs handshake
 - Gensidig kvittering



TCP header

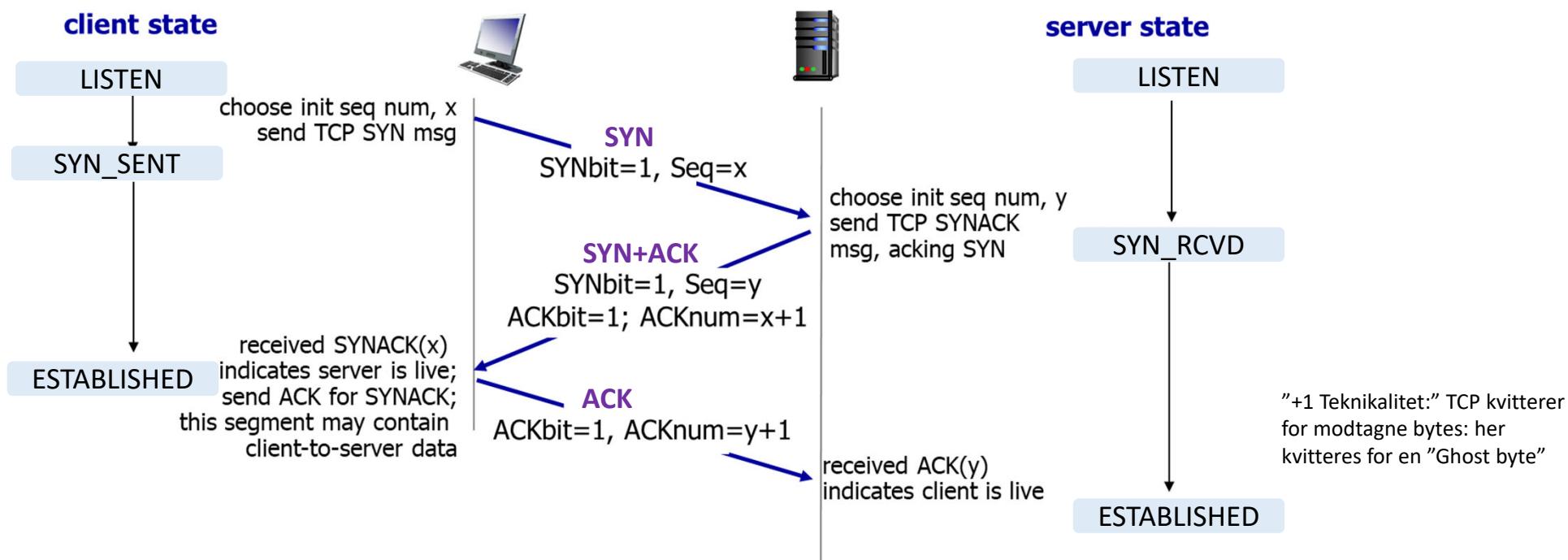
- TCP *Segment*
- Kontrol-bits til etablering og nedrivning af forbindelser:
 - SYNchronize
 - FINish
 - ACKnowledge (også til data)
 - ACK=1 \Rightarrow gyldig info i ack no feltet.
- Sekvens og ack. numre



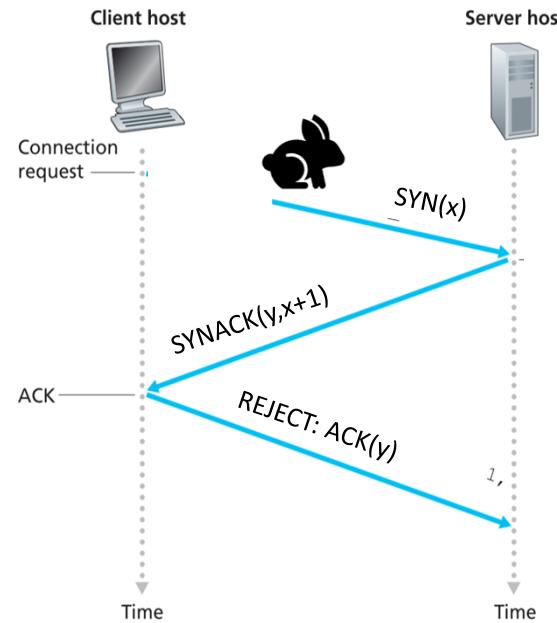
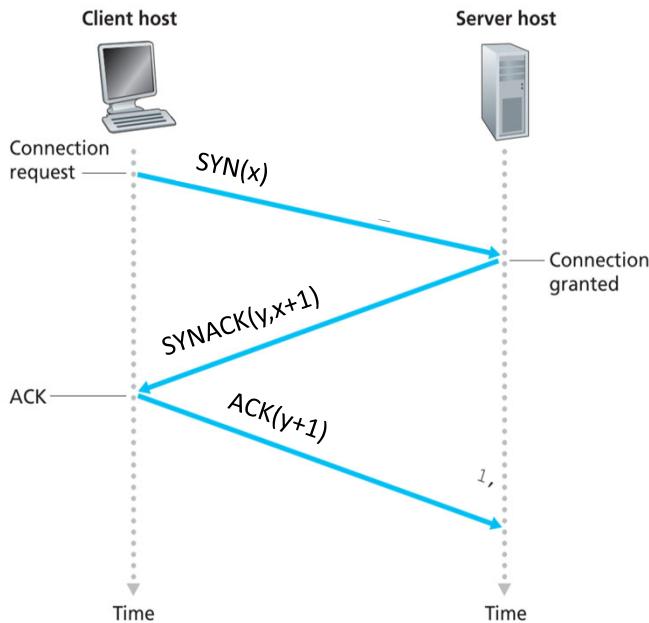
3-way-handshake

- **req_conn:** TCP **SYN** segment (SYN=1, Seq=x):
 - x=klients (tilfældigt) valgte init sekvensnr
- **ack_conn:** TCP **SYN+ACK** segment (SYN=1, ACK=1, Seq=y, AckNo=x+1):
 - y=servers (tilfældigt) valgte sekvensnummer
 - Server bekræfter; forventer at næste segment nr. har sekvens nr. x+1
- **ack :** TCP **ACK** (SYN=0, ACK=1, AckNo=y+1)
 - Klient bekræfter; forventer at næste segment nr. er sek. y+1

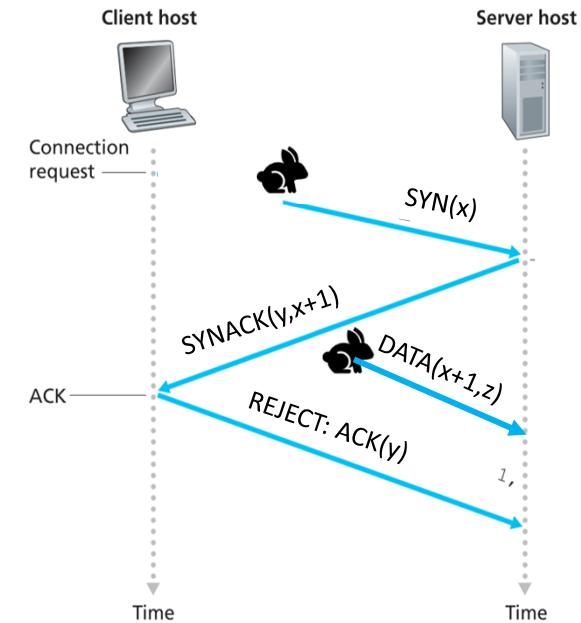
Se evt. denne video med
[wireshark analyse af TCP 3-way handshake](#)



3-way-handshake



Gammel dublikat SYN segment:
forbindelse afvises



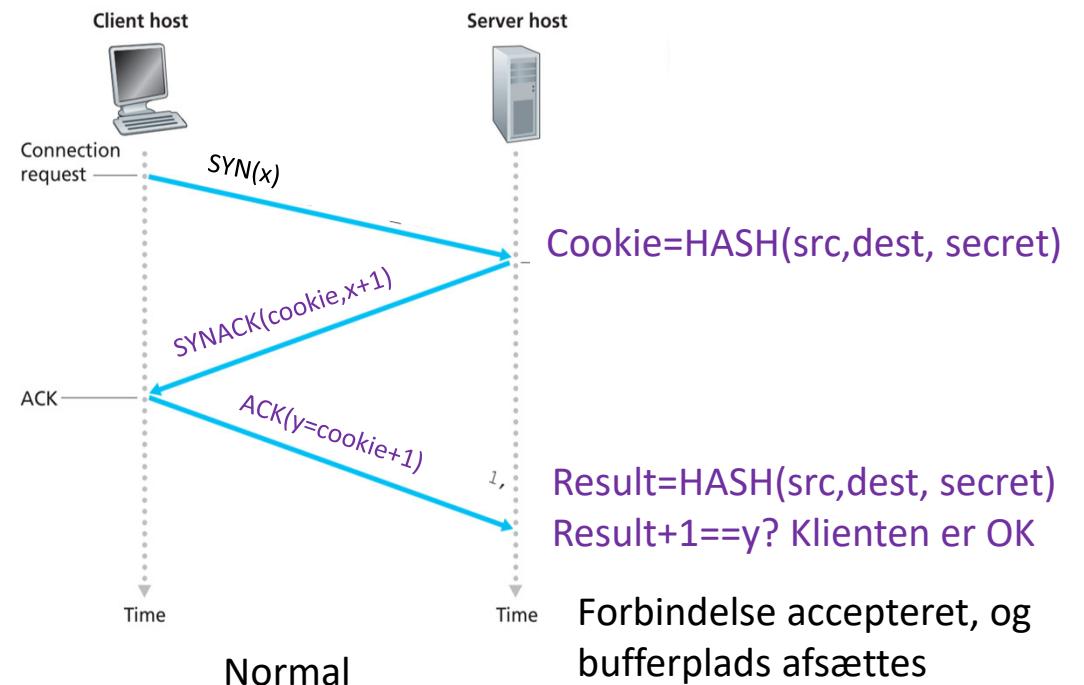
Gammel dublikat SYN og Data:
forbindelse afvises

SYN-flood angreb

- Angriber konstruerer falske SYN pakker
 - Modtagelse af SYN vil normalt få Server til at afsætte plads til buffere mv.
 - ”halv åben” forbindelse
 - Tilpas mange vil overvælde server

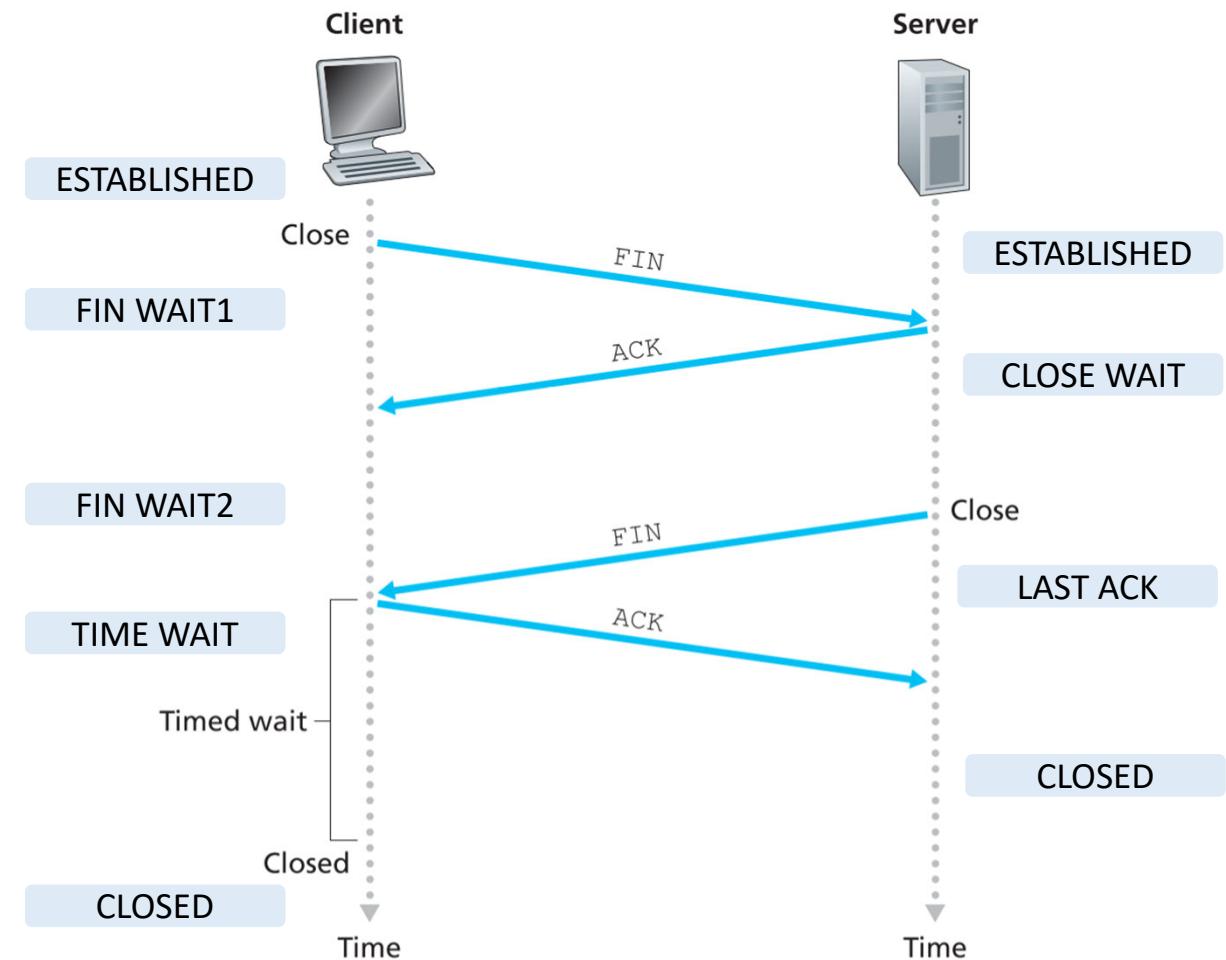
• SYN Cookie Forsvar

- Afsæt kun ressourcer når der er en ”levende” / ”legitim” client bag forespørgslen



Kontrolleret nedlægning

- Klient og server lukker begge forbindelsen
- I scenariet:
 - Klient starter
 - Server følger med sin egen
- TIME_WAIT: giv tid (fx 30-120 sec.) til at gensende sidste ACK
- Mindst lige så mange ”interessante” scenarier
- (faktisk teoretisk umuligt at de bliver ”enige” om nedlukning: two-army problem”)



Multi-plexing/De-multiplexing

Hvordan skelnes mange forbindelser?

Hvordan får data til/fra rette socket?

Multiplexing og demultiplexing

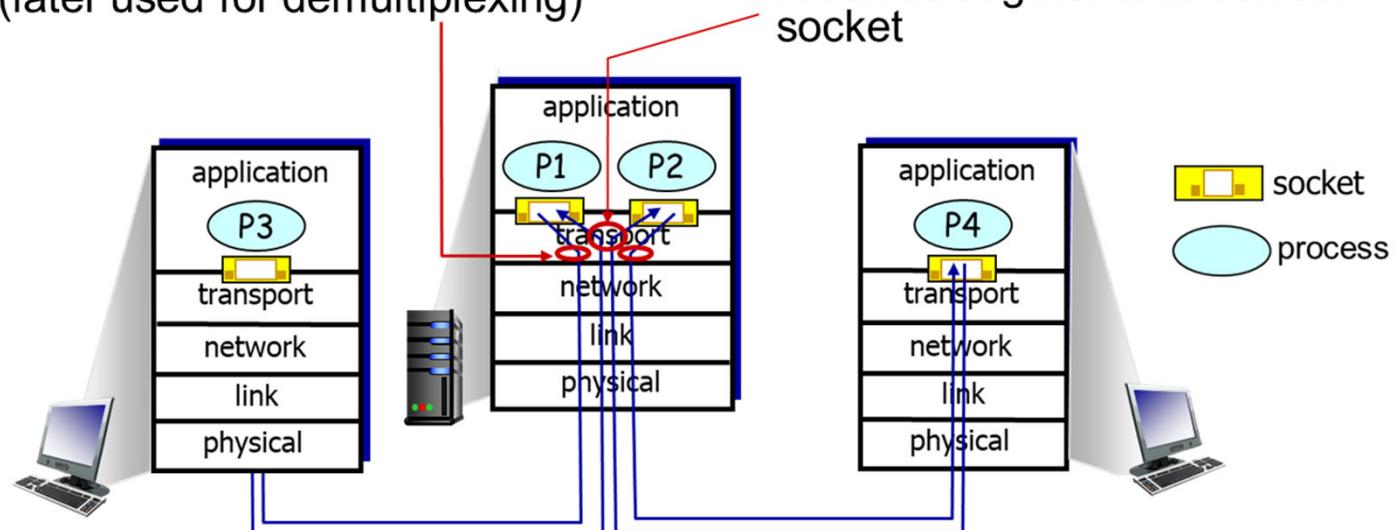
- Proces adresseres vha. hosts **IP-nummer +port nummer** indenfor en host
- Processer sender/modtager data via en **socket**, et bindeled mellem applikations- og transport-lag
- Der er mange processer på en host, dermed mange som transport laget skal betjene
 - En proces kan kommunikere med mange andre samtidigt (kan dermed have mange sockets)
- Hvordan leveres data til den rette socket?

multiplexing at sender:

handle data from multiple
sockets, add transport header
(later used for demultiplexing)

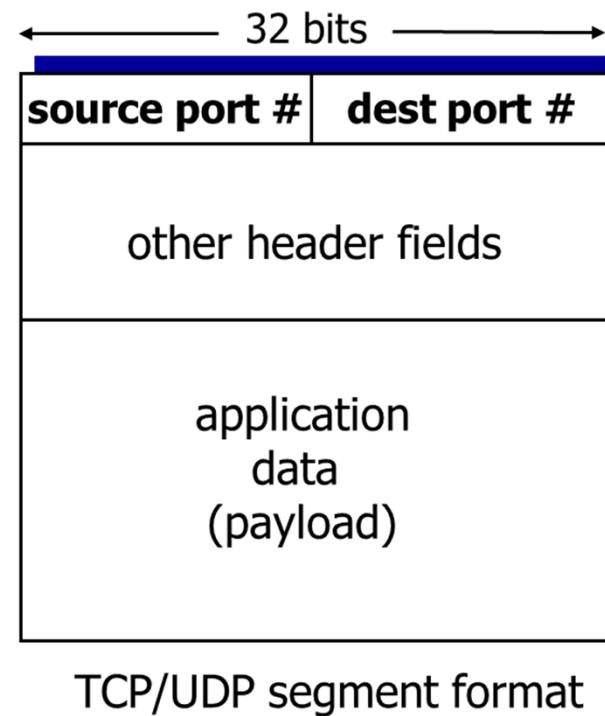
demultiplexing at receiver:

use header info to deliver
received segments to correct
socket



Demultiplexing

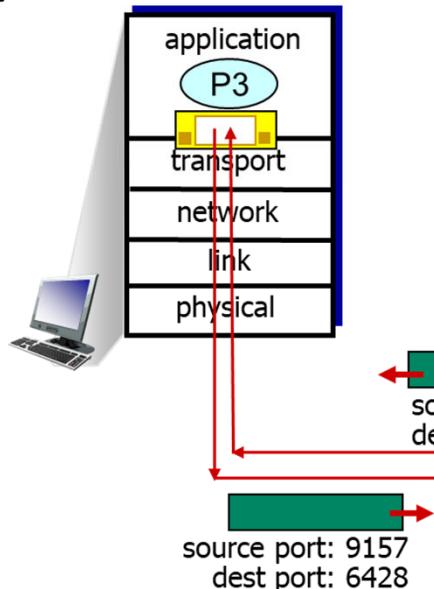
- Host modtager et IP datagram
 - Hvert datagram har et **source IP adresse, dest IP adresse**
 - Hvert datagram bærer ét transport-lags segment
 - Hvert transport-lags segment har **en source og destinations port**
- Host bruger **IP-adresse og port numre** til at videreføre data til den tiltænkte socket



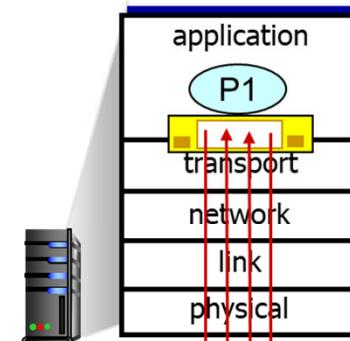
De-multiplexing i UDP

- Modtager socket identificeres ved modtager IP og porte (dest IP, dest port)
- Modtagelse af UDP-segment:
 - Checker for om der findes en aktiv socket på dest-port
 - Videresender data til denne
- NB! 2 UDP segmenter med samme dest, men forskellig src leveres til samme socket
- NB! Source IP og port skal kendes hvis sender skal kunne besvare modtager

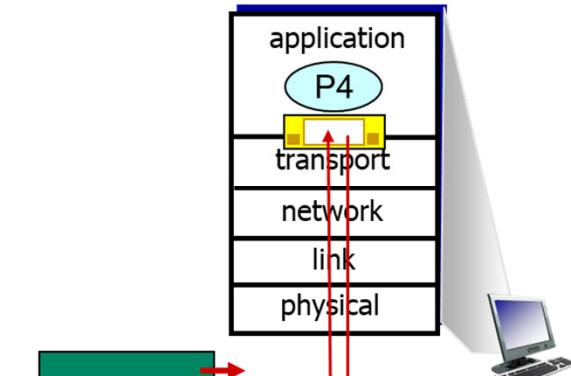
```
DatagramSocket  
mySocket2 = new  
DatagramSocket  
(9157);
```



```
DatagramSocket  
serverSocket = new  
DatagramSocket  
(6428);
```

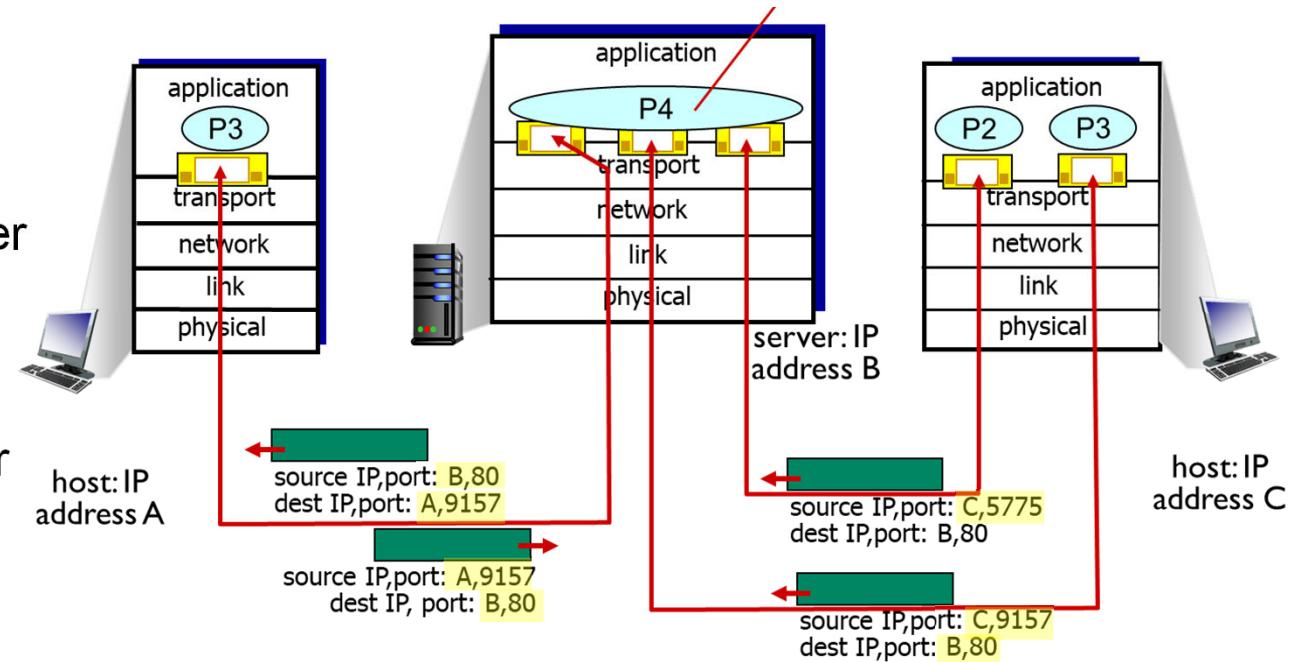


```
DatagramSocket  
mySocket1 = new  
DatagramSocket  
(5775);
```



Demultiplexing i TCP

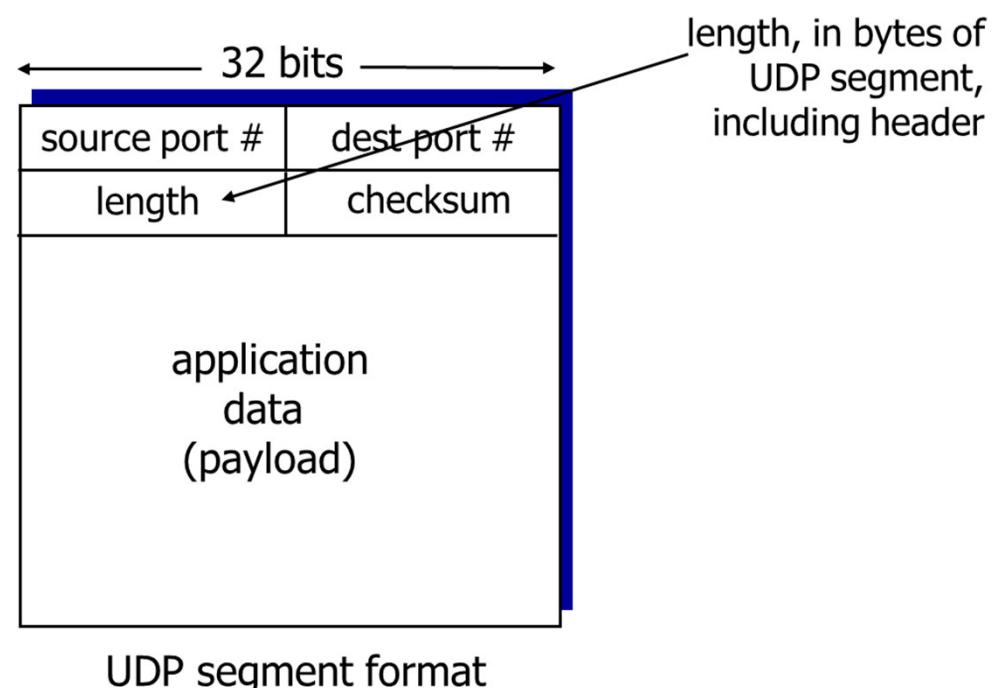
- TCP socket identificeres med 4-tupel *):
 - **source IP adresse**
 - **source port nummer**
 - **dest IP adresse**
 - **dest port nummer**
- demux: modtager bruger alle 4 værdier for at videresende til den tiltænkte socket
- server proces kan have mange samtidige TCP forbindelser (fx, til hver web-klient):
 - Hver socket identificeres med sin egen 4-tupel



*) når forbindelsen er etableret.

UDP-transport

- Ingen ventetid på etablering af forbindelser
- Ingen congestion kontrol
- Meget simpel og lille header
 - Lavt overhead
- Checksum!
 - Støj på linien kan ”flippe” en eller flere bits
 - 01011 modtages som **11011**
 - Sender beregner en checksum på sendte data
 - Inkluderer dem i segmentet
 - Modtager beregner checksum på modtagne data
 - Pakken formodes at være korrekt modtaget hvis modtaget og beregnet checksum stemmer overens



UDP Checksum

- UDP segment betragtes som en serie af 16 bit tal
 - Senderen:
 - Summerer serien af 16-bit tal efter one's complement metoden
 - Tager komplementet til denne sum; bruger resultatet som checksum
 - På modtageren
 - Summerer igen alle 16-bit tal, inkl. checksummen
 - Sum bør give 1111111111111111
 - Hvis ja: **sandsynlighed for at pakken er fejl-ramt er mindsket!**

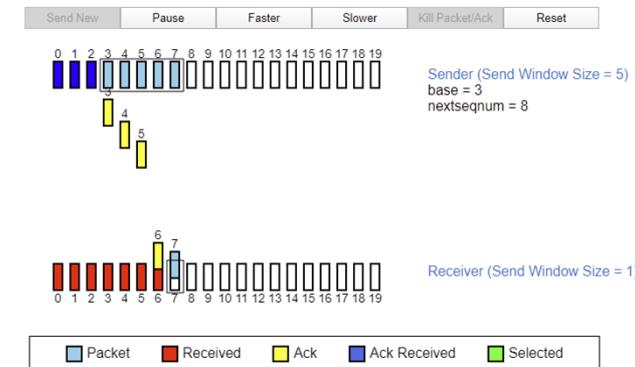
Opgaverne idag

- Review: Har man forstået grundlæggende begreber
 - Demultiplexing,
 - Retransmissionsstrategier: Gå-N-tilbage, selektiv retransmission,
- Øvelser:
 - Styring af protokol-tilstandsmaskine: alt-bit
 - Sliding windows
- Praktiske: Kan anvende netværksværktøjerne
 - Portscan med NMAP: hvilke applikationer lytter på en given HOST?

Prøv simulator

Go-Back-N Protocol.

This interactive animation brings to life the Go-Back-N protocol. In this demo, the sending window lies between sender and receiver. To simulate loss, select a moving data packet or ack, and then press "retransmissions" (i.e., timeouts).



SLUT