

# Internetwork og Web-programmering

## Transport laget

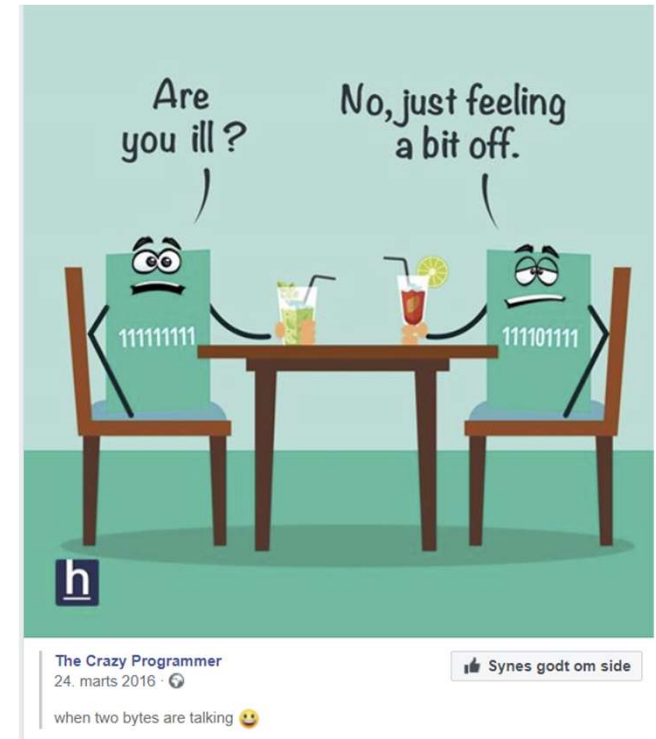
Forelæsning 8  
Brian Nielsen

Distributed, Embedded, Intelligent Systems



# Agenda

1. Transportlaget: services og protokoller
  1. Multiplexing/Demultiplexing
    - UDP
    - TCP
  2. UDP fejl-check
2. Principper for pålidelig data-kommunikation
  - Trinvis udvikling af en simpel protokol
3. Principper sliding window protokoller
  - Go-Back-B
  - Selective Repeat
4. Etablering og nedlukning af TCP forbindelser

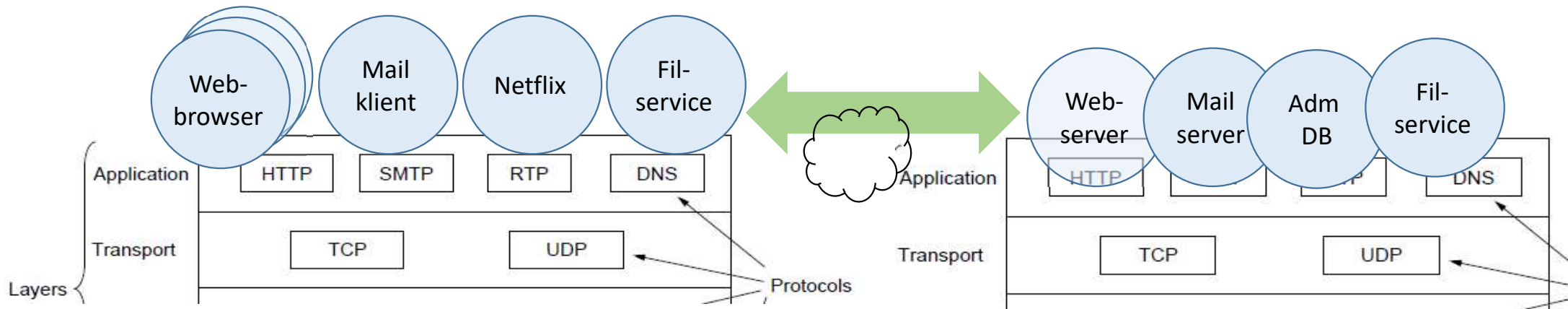


# Transportlags protokoller

Hvordan fremsendes data til modtager processerne?

Hvordan sikre vi mod fe

# Transportlaget i TCP/IP modellen



## Transportlags opgaver

- **END2END** 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 byte-stream service

## UDP (user datagram protocol)

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

# Multiplexing og demultiplexing

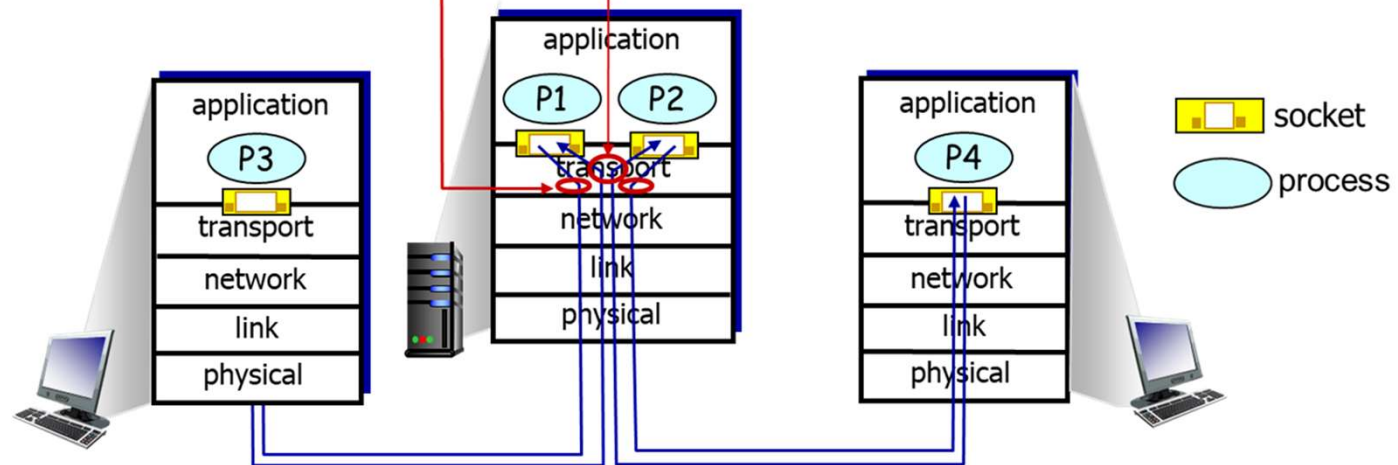
- Proces adresseres vha. hosts IP-nummer +port nummer indenfor en host
- Processer sender/modtager data via en **socket**, et bindeled/dør mellem applikationslag og transport lag
- En proces kan kommunikere med mange andre samtidigt (har dermed mange sockets)

## multiplexing at sender:

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

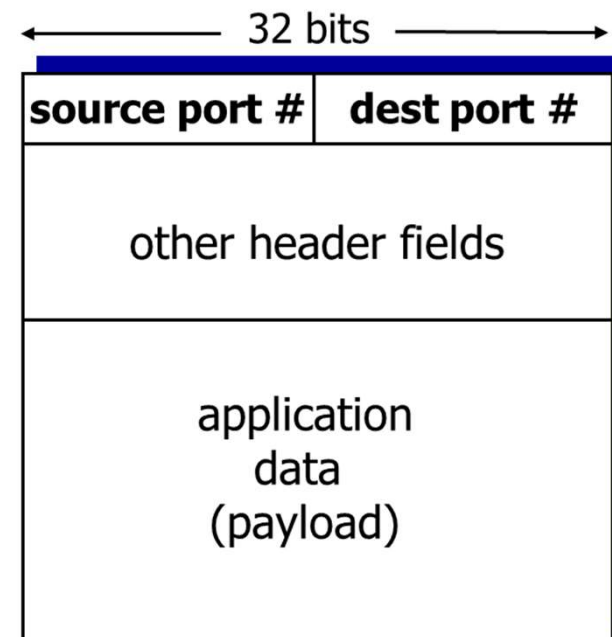
## demultiplexing at receiver:

use header info to deliver received segments to correct socket



# Demultiplexing

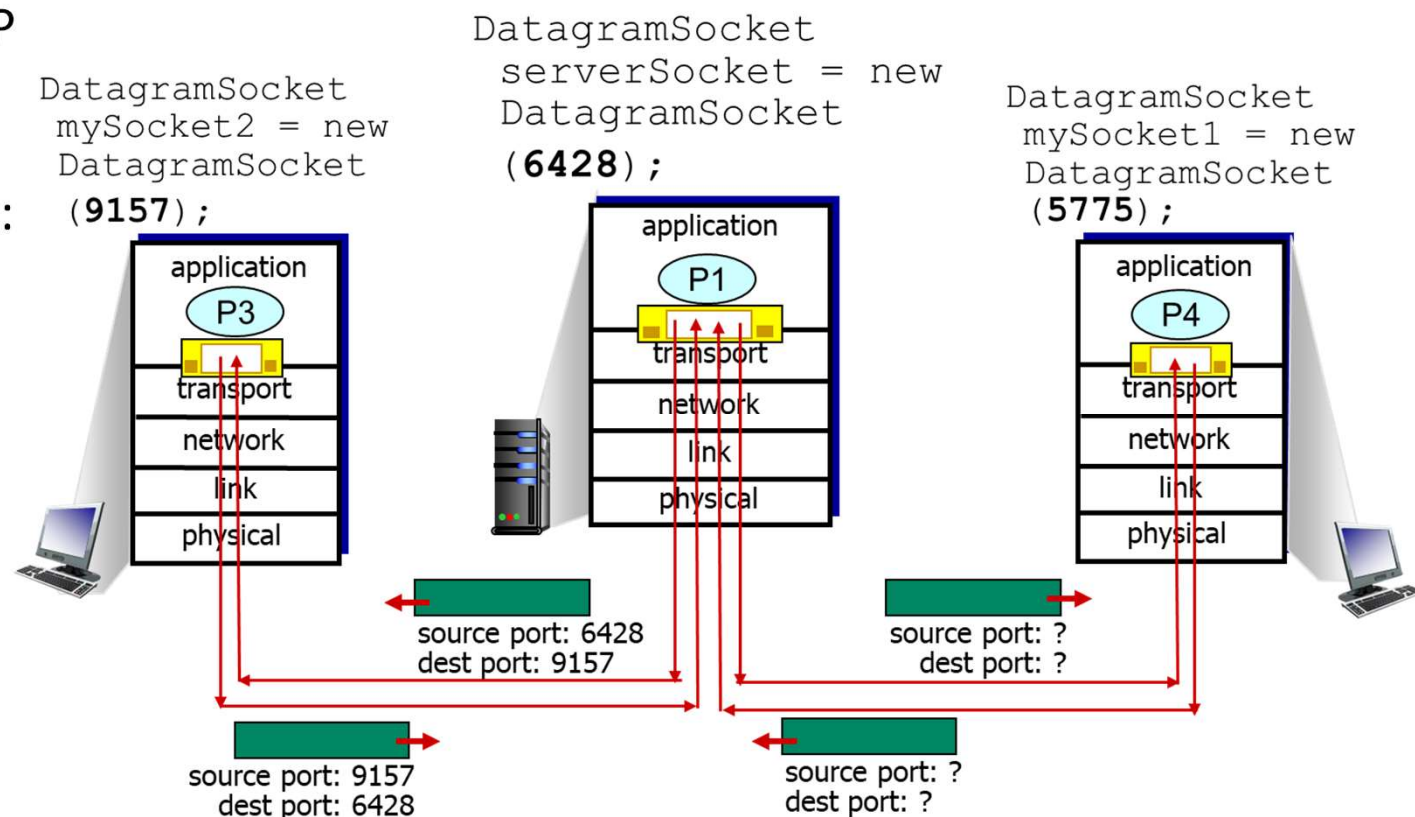
- Host modtager et IP datagram
  - Hvert datagram har et **source IP adresse, dest IP adresse**
  - Hvert datagram bærer ét transport-lags segment
  - Hvert segment har **en source og destinations port**
- Host bruger **IP-adresse og port numre** til at viderelevere data til den tiltænkte socket



TCP/UDP segment format

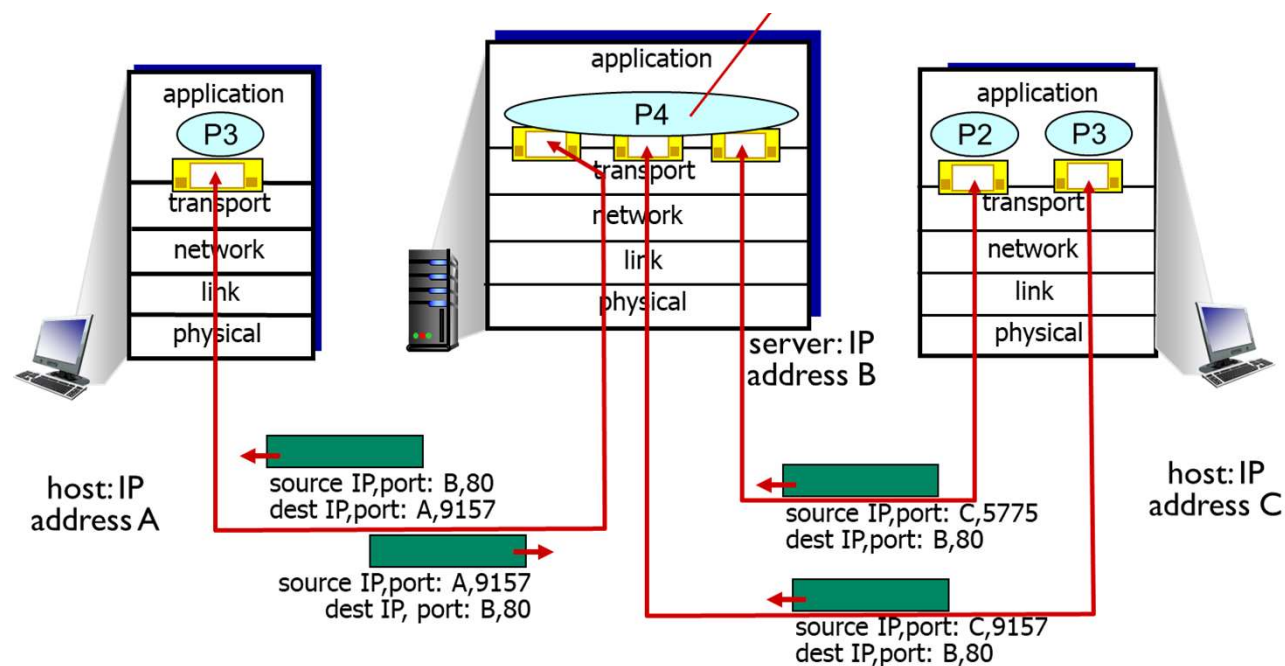
# De-multiplexing UDP

- 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
  - Videre sender 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



# Demultiplexing i TCP

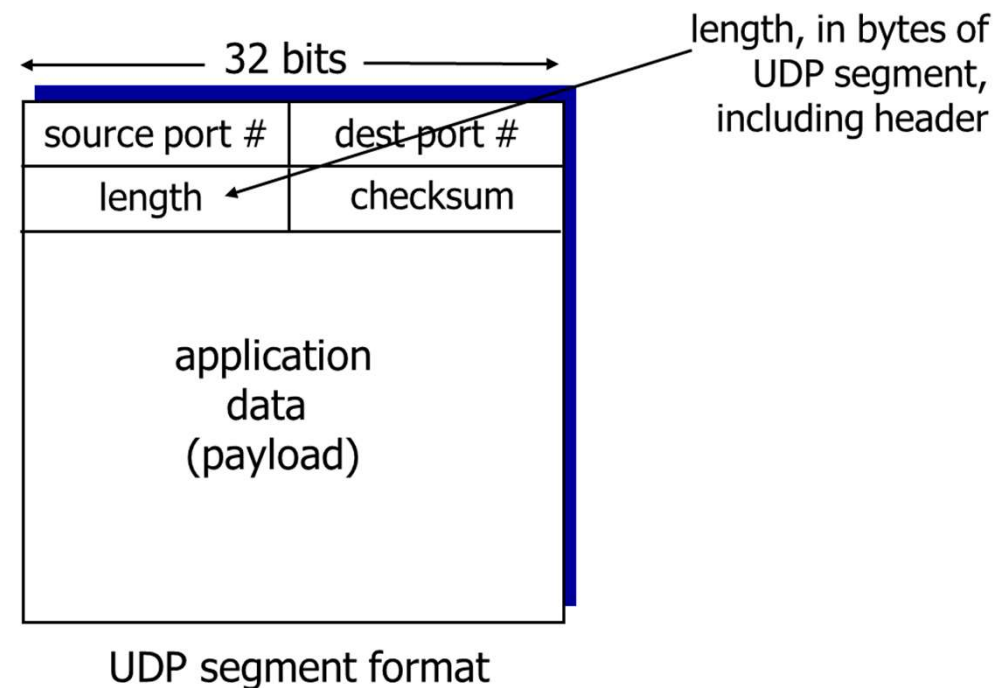
- TCP socket identificeres med 4-tupel:
  - **source IP adresse**
  - **source port nummer**
  - **dest IP adresse**
  - **dest port nummer**
- demux: modtager bruger alle 4værdier for at videresende til den tiltænkte socket
- server proces kan have mange samtidige TCP forbindelser (fx, til hver web-klient):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





# 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

## One's complement

5: 0101

+0: 0000

-0: 1111

-5: 1010

- 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!

Mente	1	1				1				1				1		
	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
	<hr/>															
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

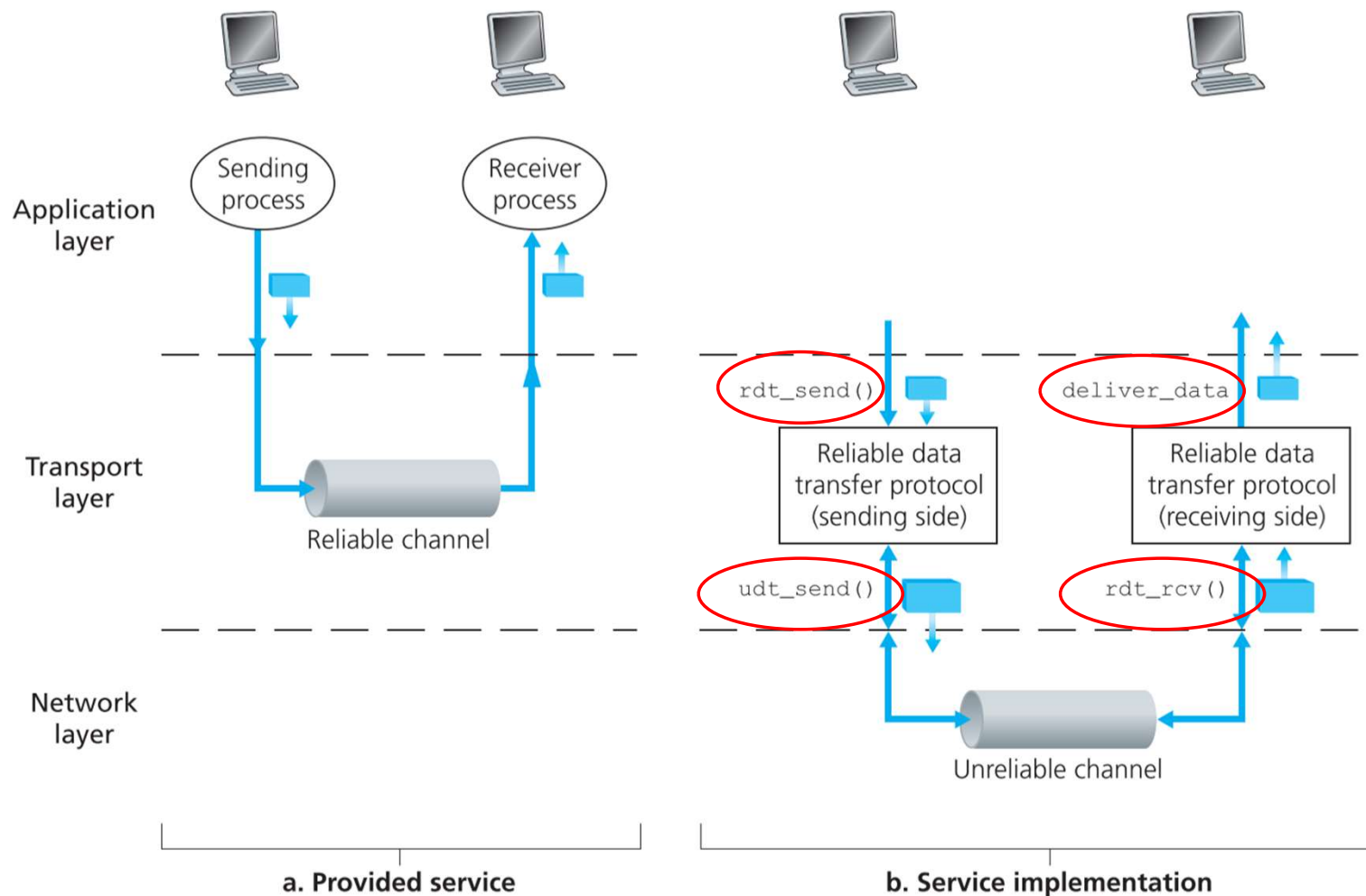
# En pålidelig transport protokol

Hvordan sikrer man at data kan nå frem med pakketab?

Hvordan modelleres og specificeres en protokol?

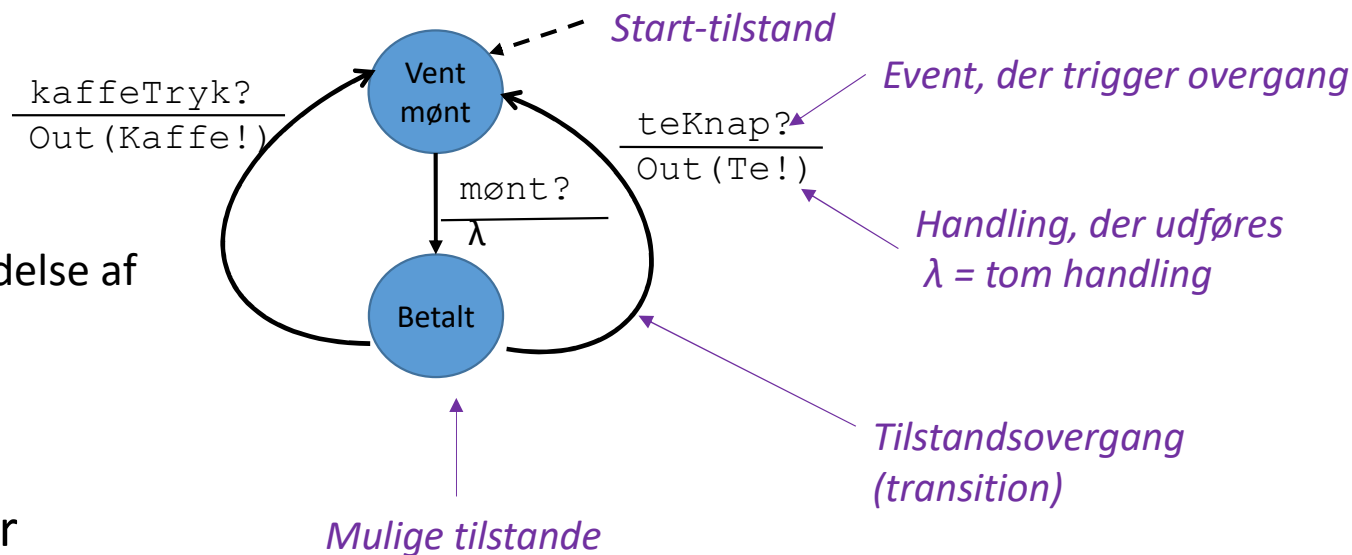
Hvordan implementeres en protokol?

# Abstraktion og Implementation



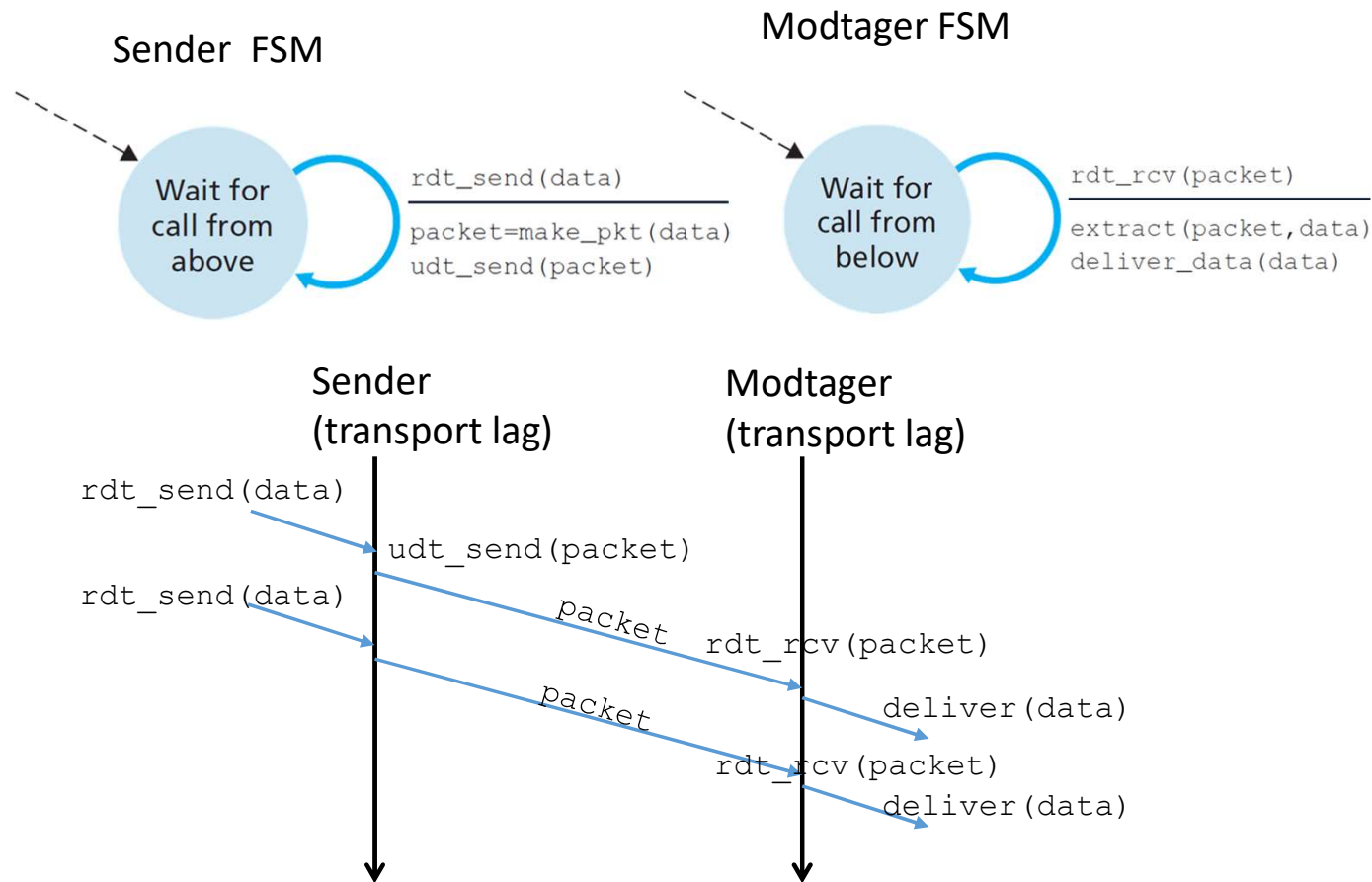
# Tilstandsmaskiner

- 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**
- Anvendelser 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



# Pålidelig kanal: rdt1.0

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



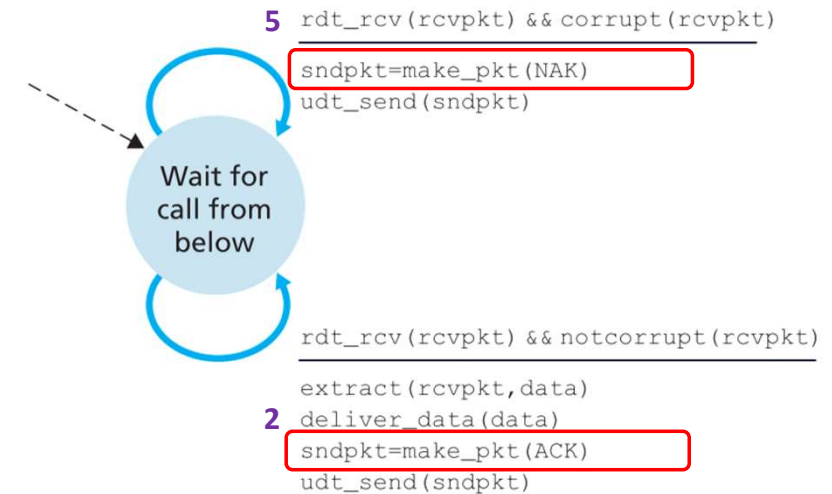
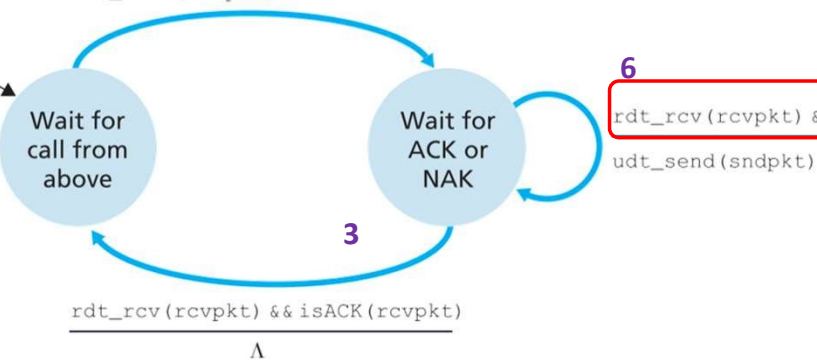
Sekvensdiagram (message sequence chart) viser ét muligt scenarie

# 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

```

rdt_send(data)      1    4
sndpkt=make_pkt(data,checksum)
udt_send(sndpkt)
    
```



Sender  
(transport lag)

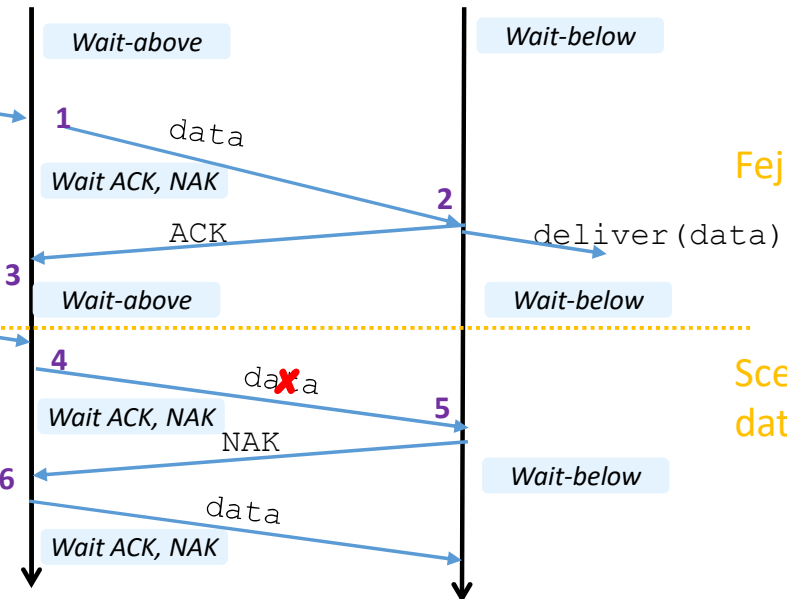
Modtager  
(transport lag)

rdt\_send(data)

rdt\_send(data)

```

6
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
    
```

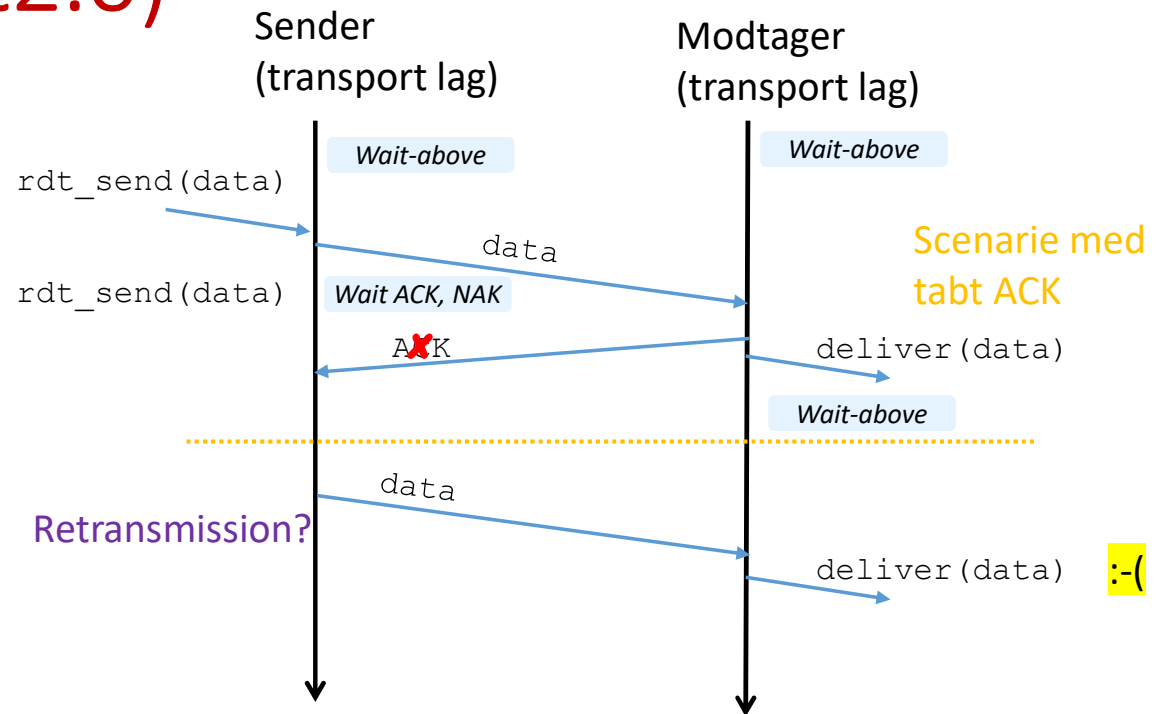


Fejlfrit scenarie

Scenarie med data tab

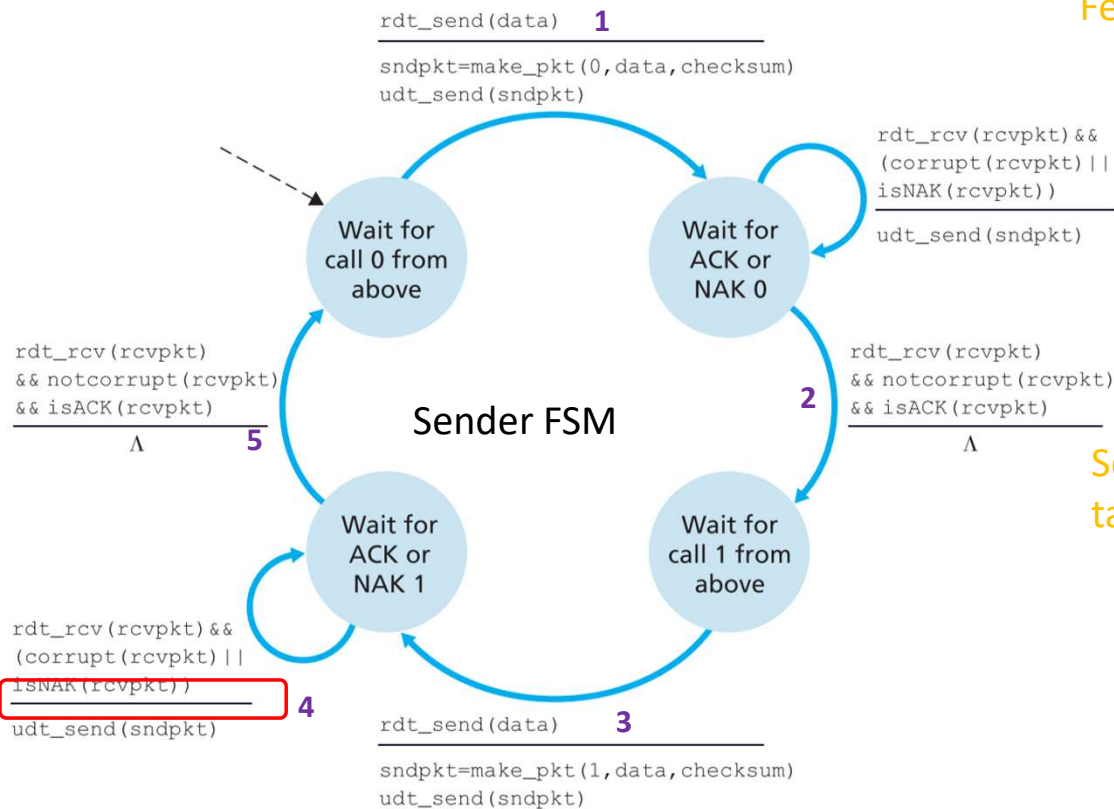
# Kanal med bitfejl (rdt2.0)

- 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 gendendt data).
  - ⇒ Brug sekvensnumre



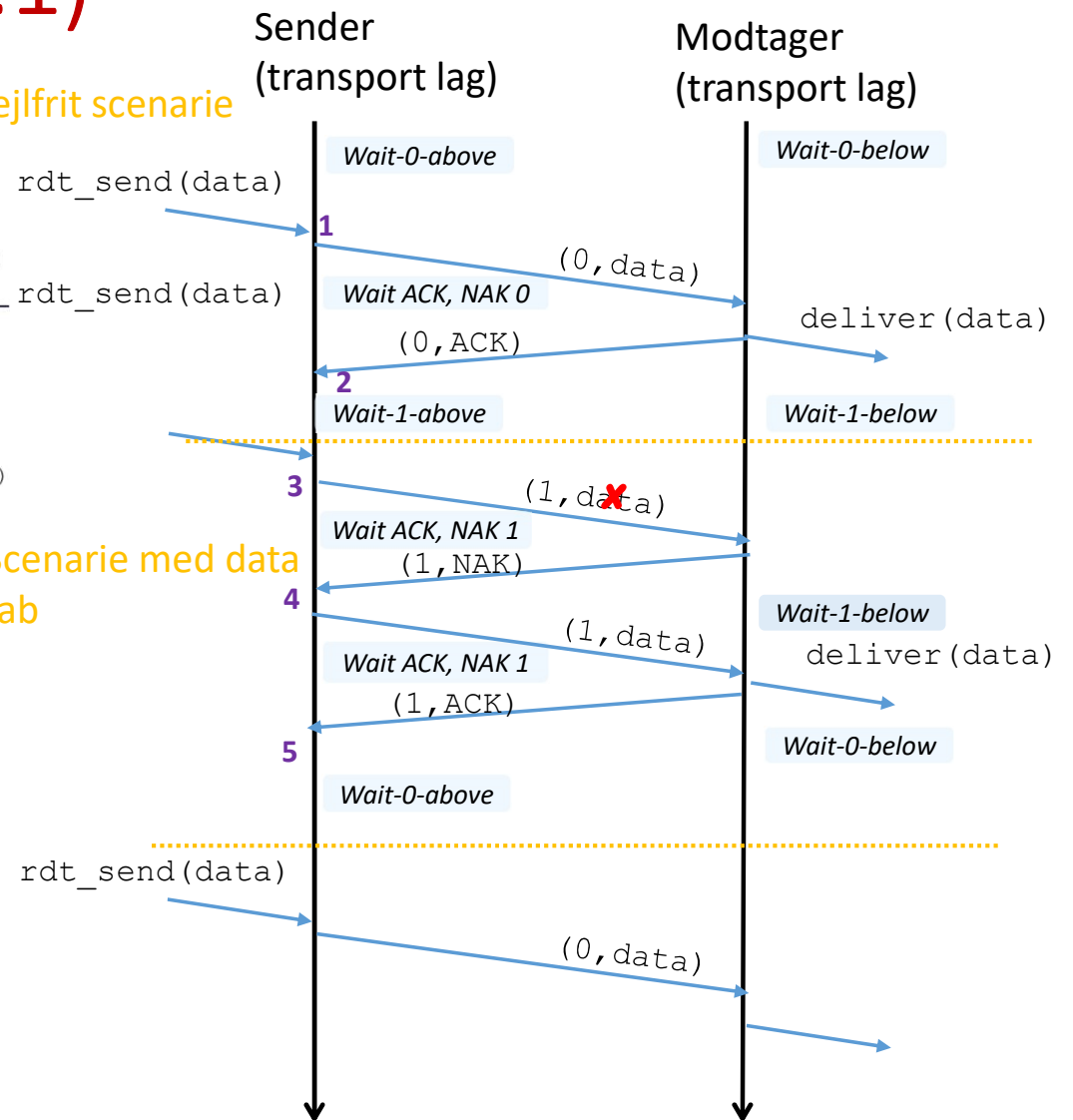


# Sekvens-numre 1 (rdt2.1)

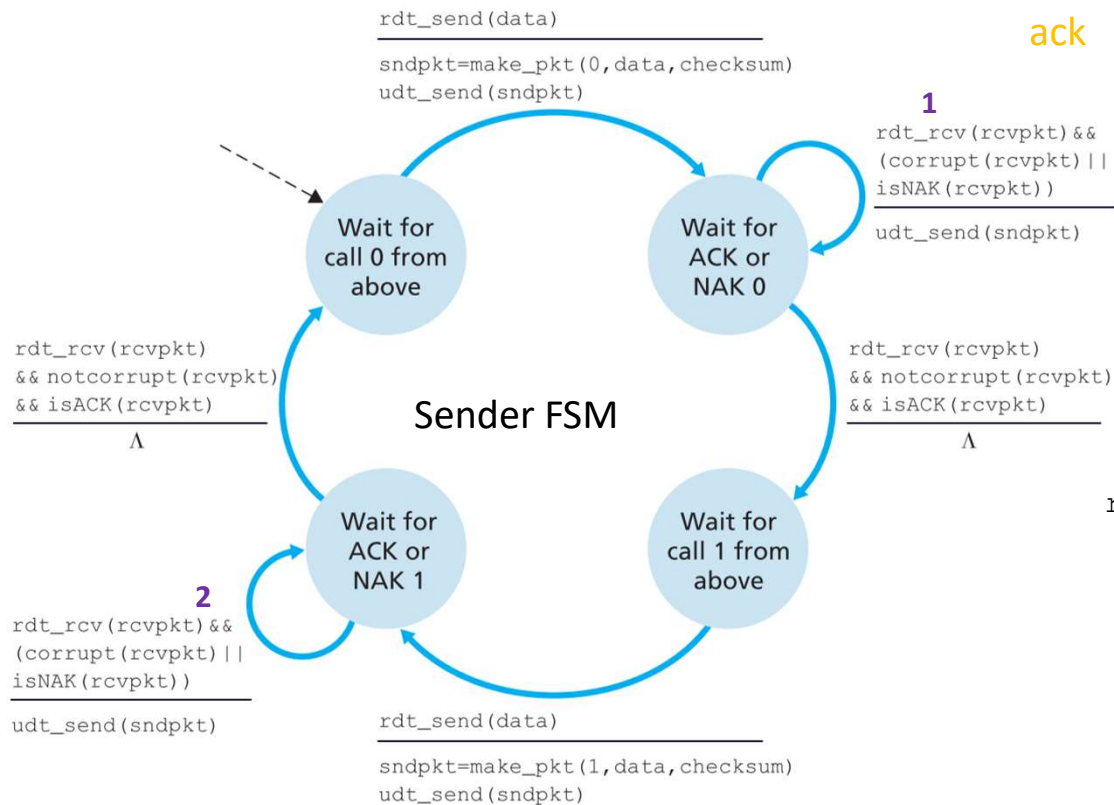


Fejlfrit scenarie

Scenarie med data tab

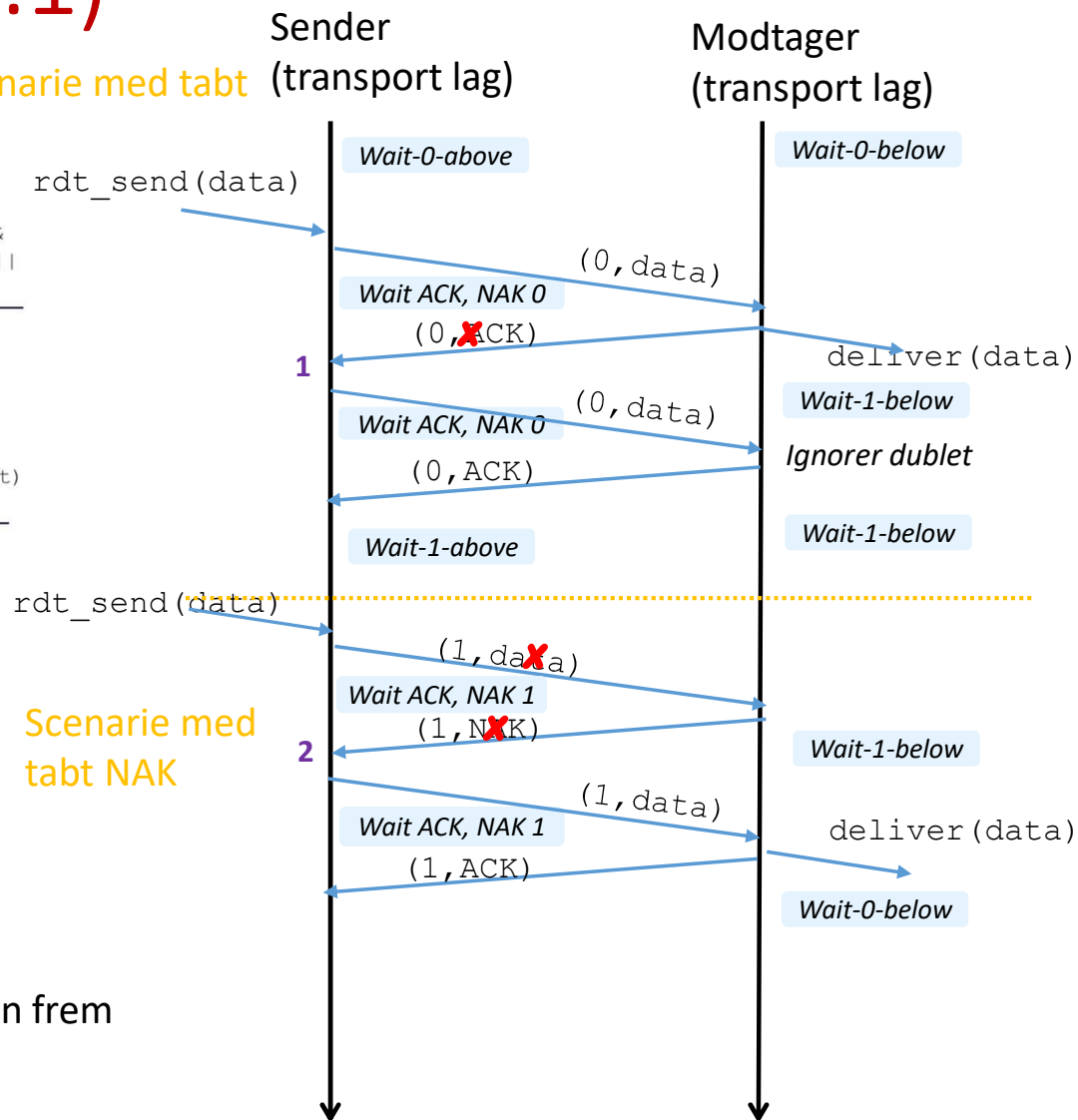


# Sekvens-numre 2 (rdt2.1)

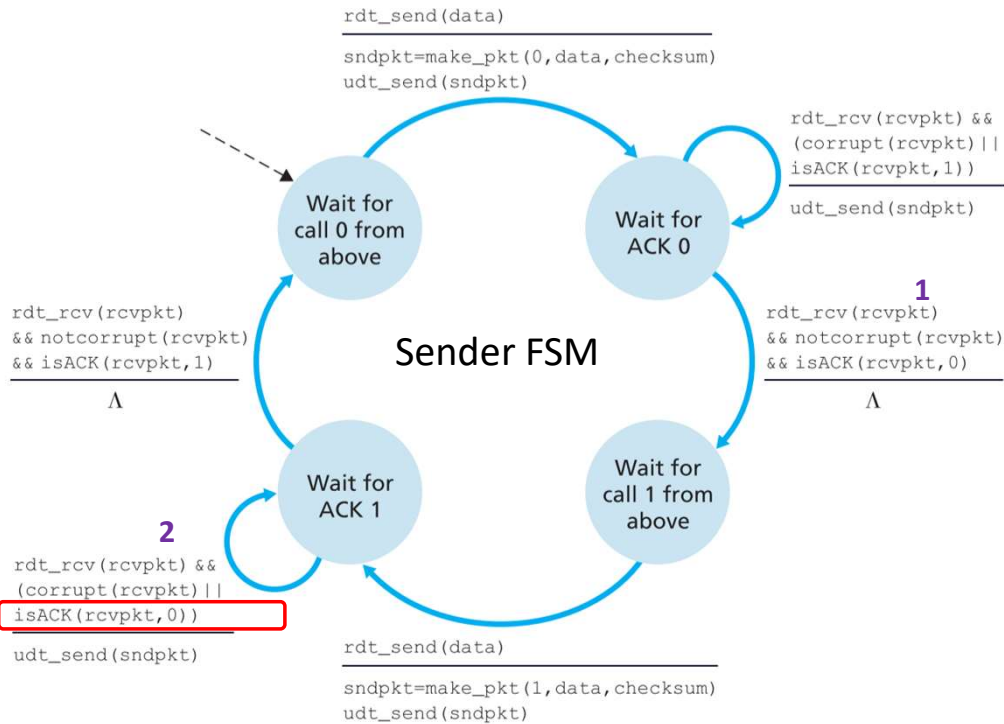


- Tabt ACK, NAK har samme effekt: retransmission
- Kun ACK med forventet sekvensnr, bringer protokollen frem
- $\Rightarrow$  Kan vi klare os med kun én af ACK, NAK?

Scenarie med tabt  
ack

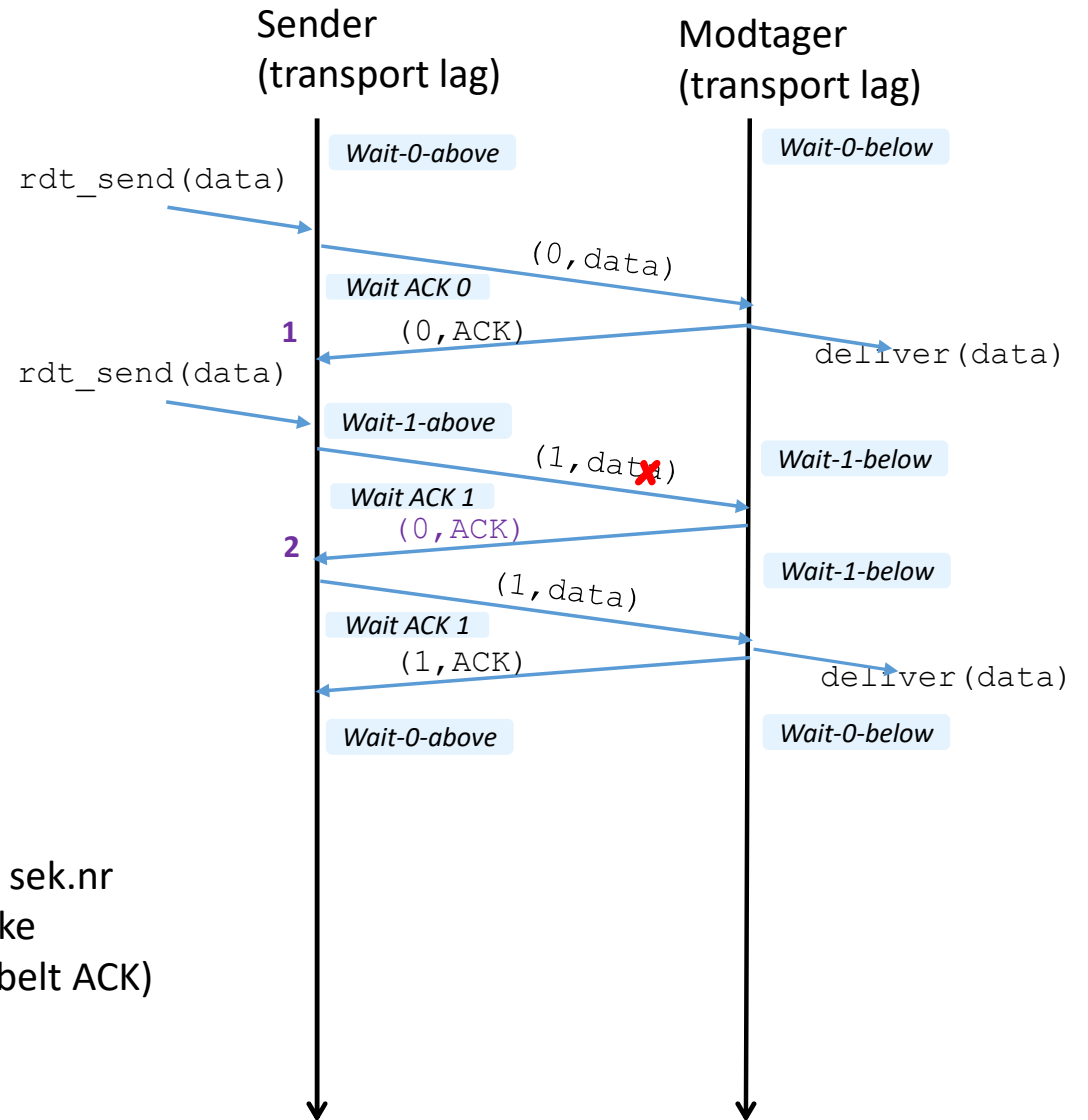


# Sekvens-numre og kun ACK (rdt2.2)

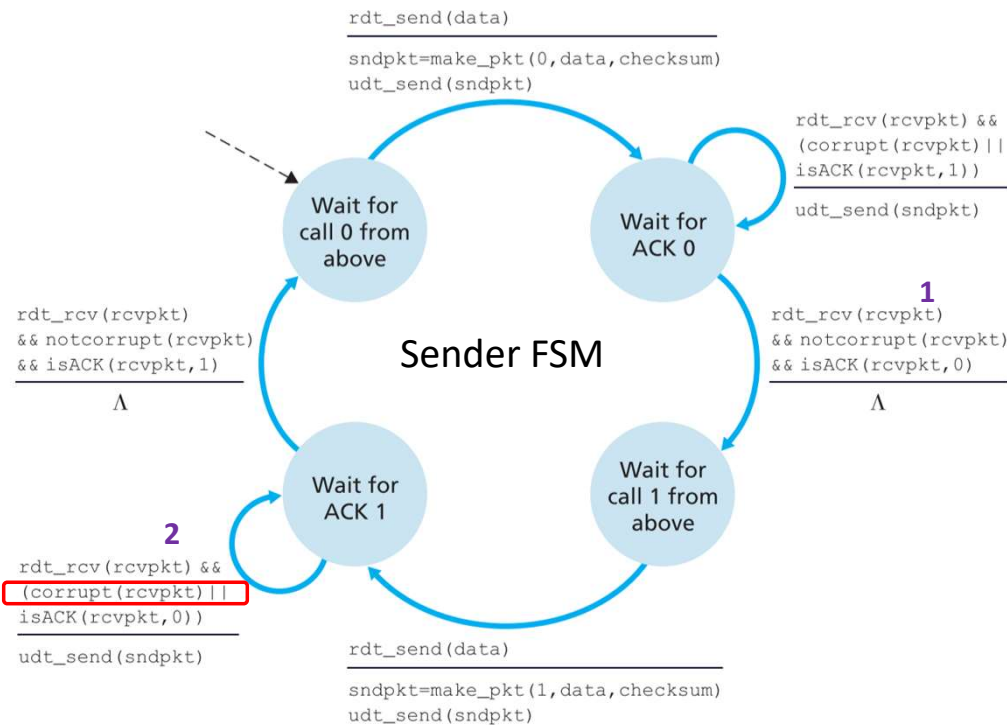


- Ved tabt data, kvitterer modtager med seneste korrekte sek.nr
- ACK inkluderer sekvensnr på den korrekt modtagne pakke
- Sender, der modtager ACK med uventet sekvensnr (dobbel ACK)
- $\Rightarrow$  retransmission

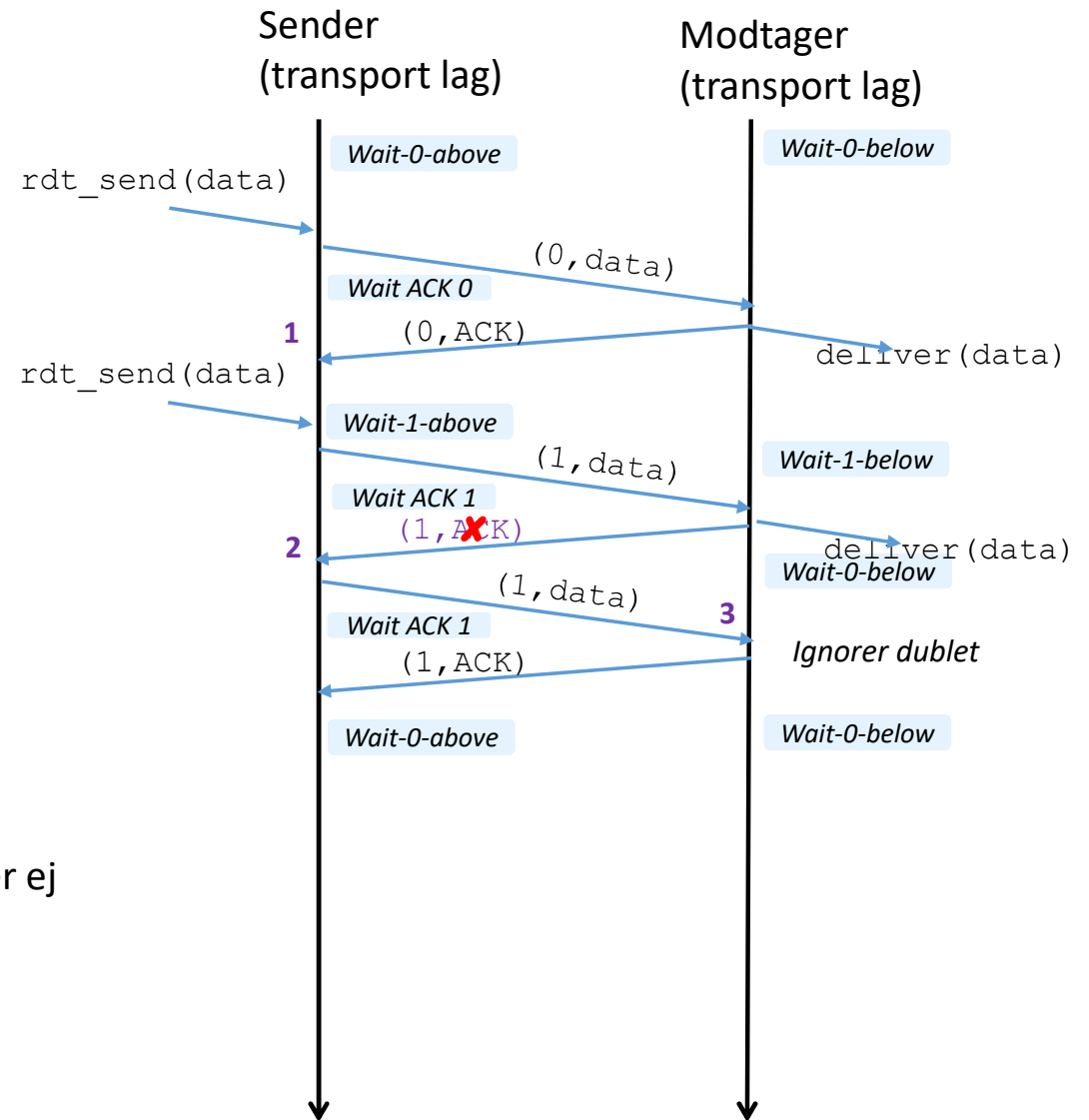
## Scenarie med tabt data



# Sekvens-numre og kun ACK (rdt2.2)



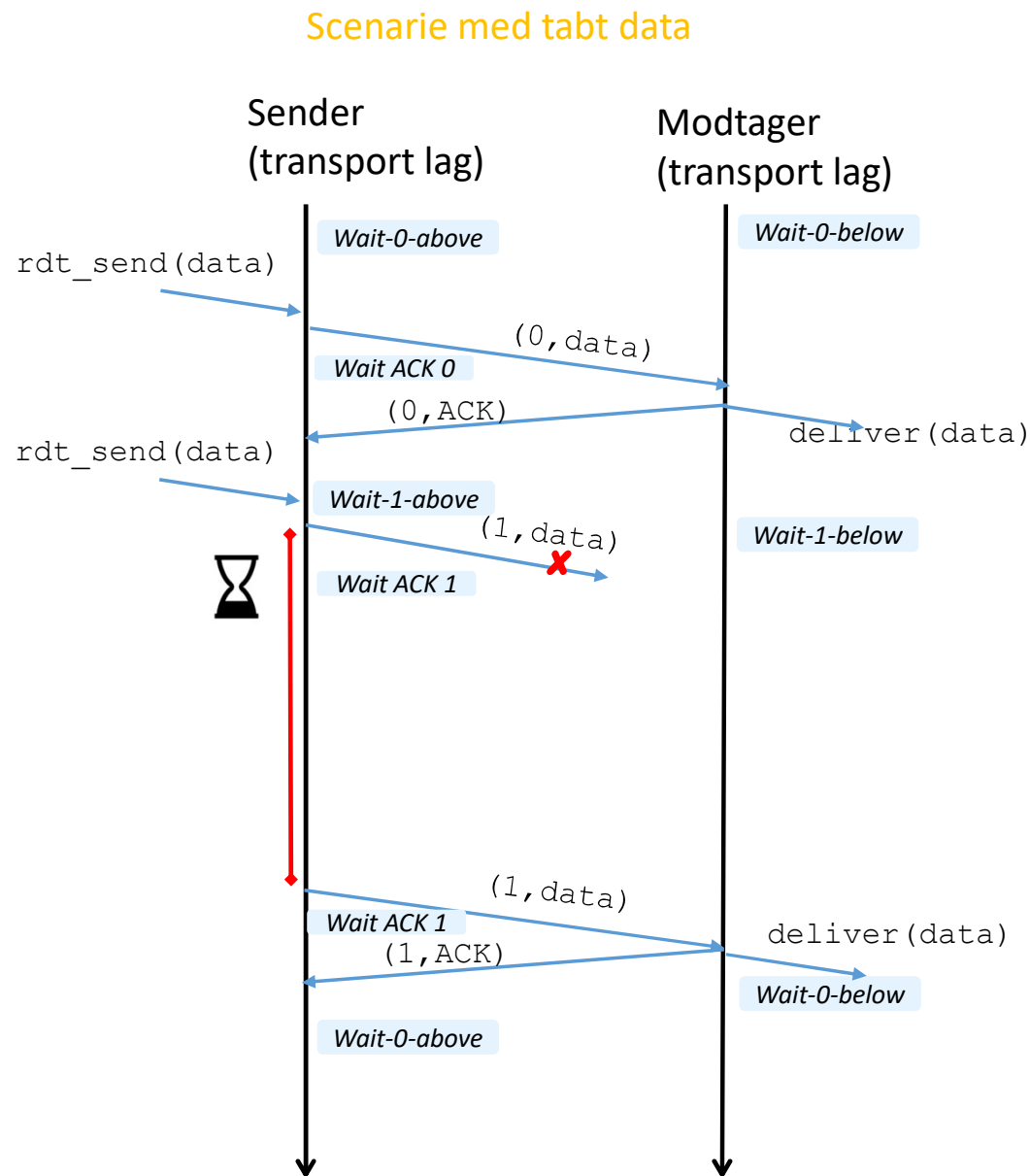
## Scenarie med tabt ACK



- Ved tabt ACK, ved sender ikke om data er modtager eller ej
- $\Rightarrow$  retransmission
- **3:** modtager dublet: ignorerer data, men gensend ACK

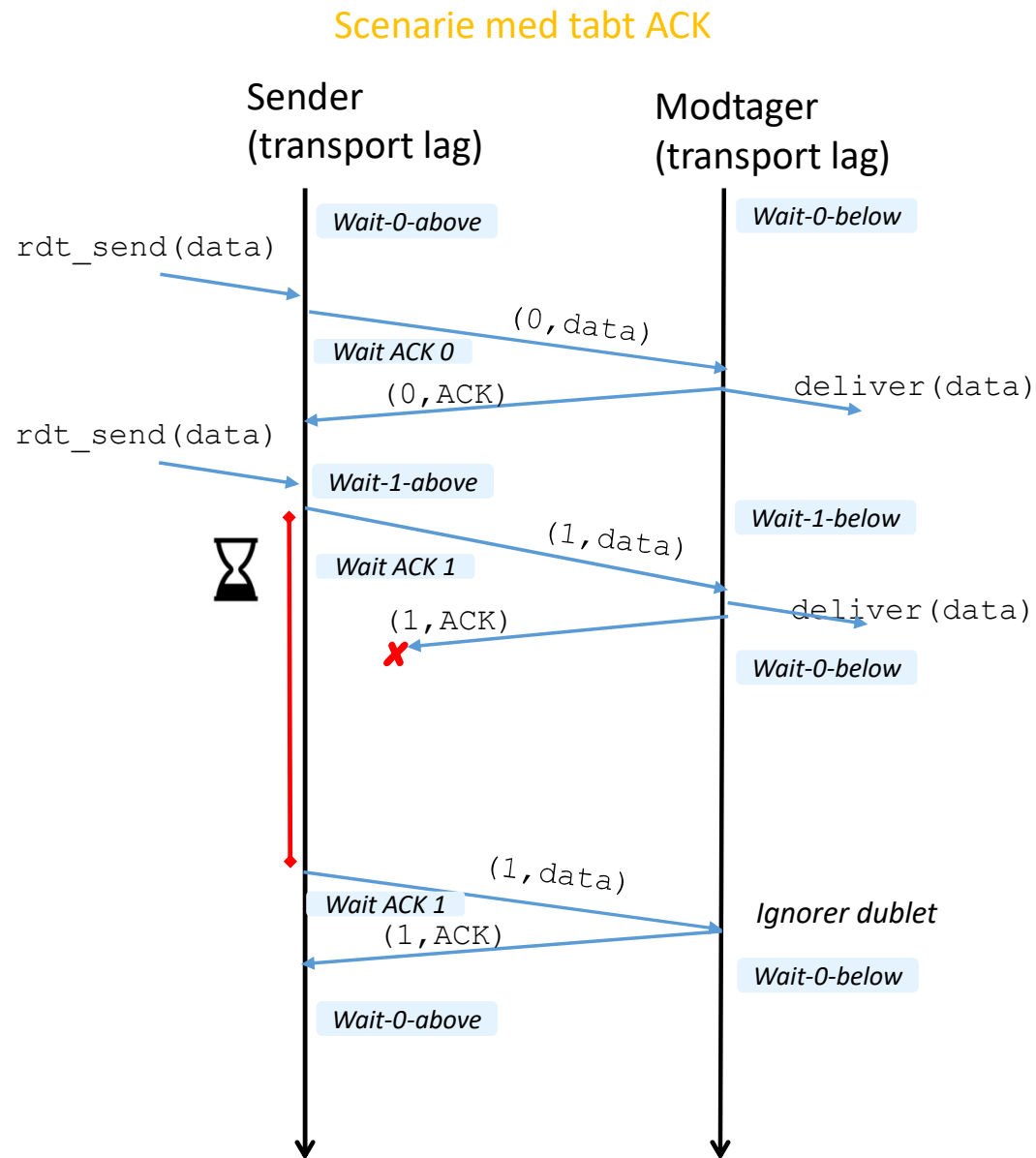
# Pakke tab 1: rtd3.0

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



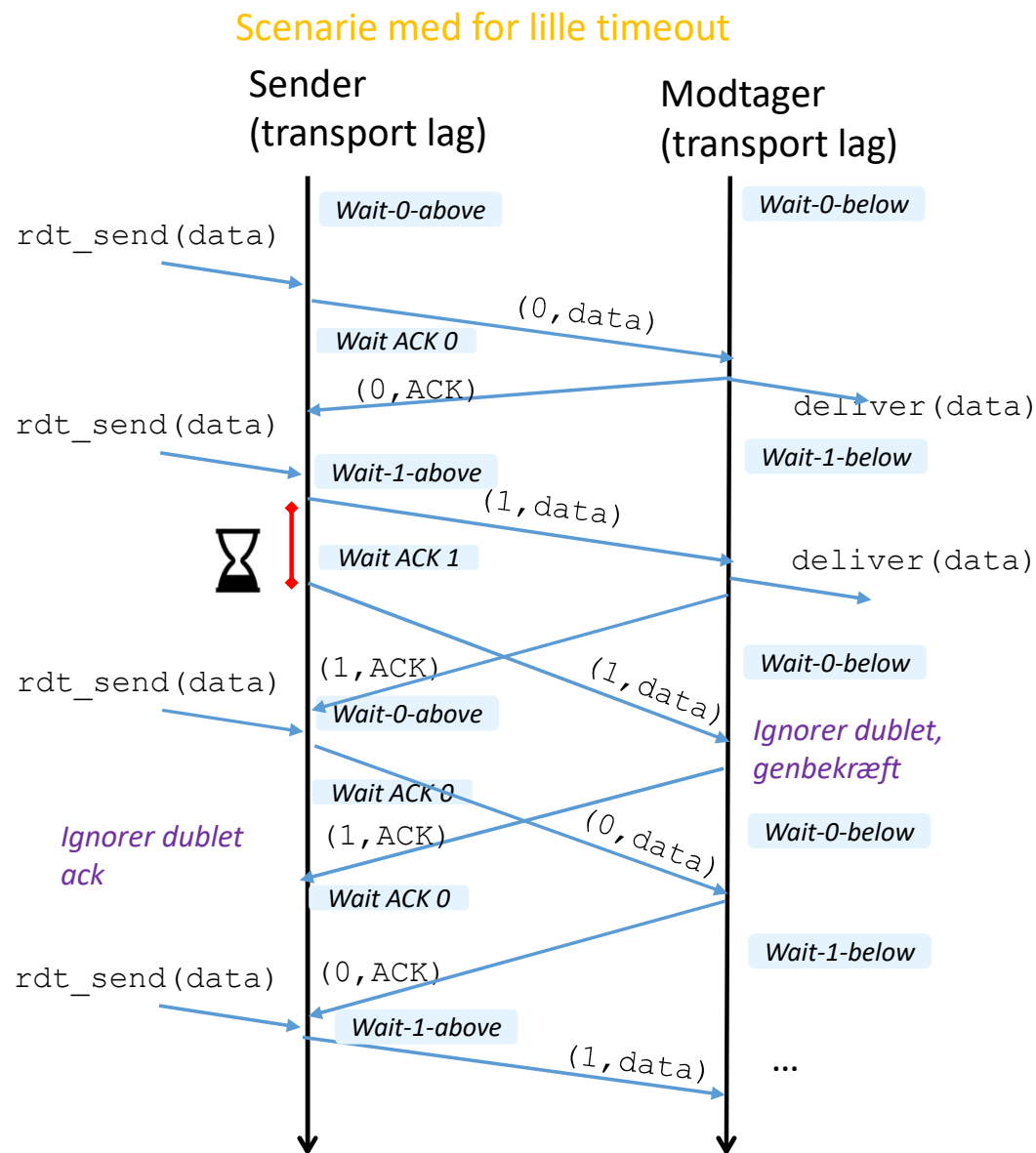
## Pakke tab 2: rtd3.0

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

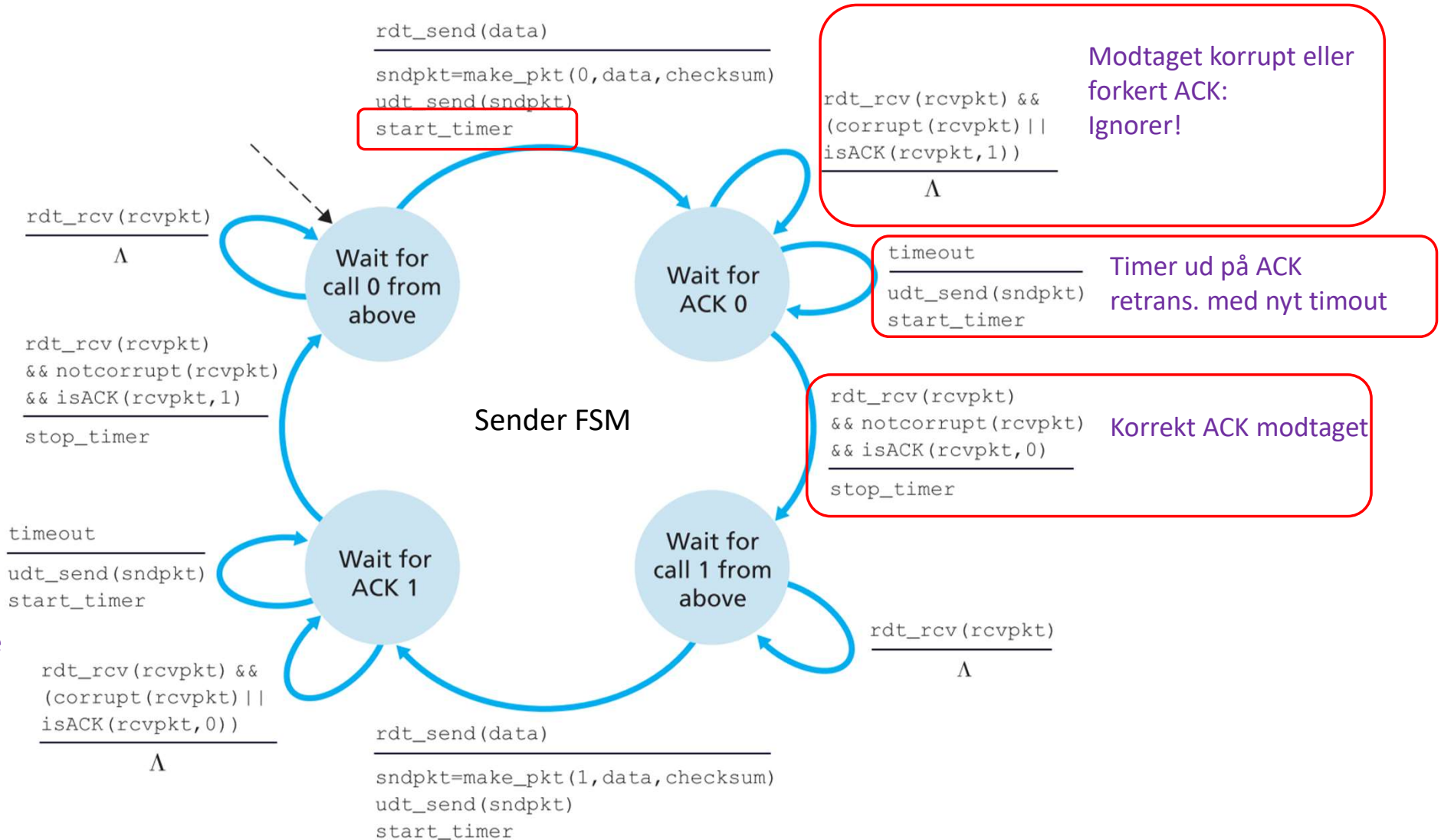


# Pakketab 3: rtd3.0

- 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



# Rtd 3.0

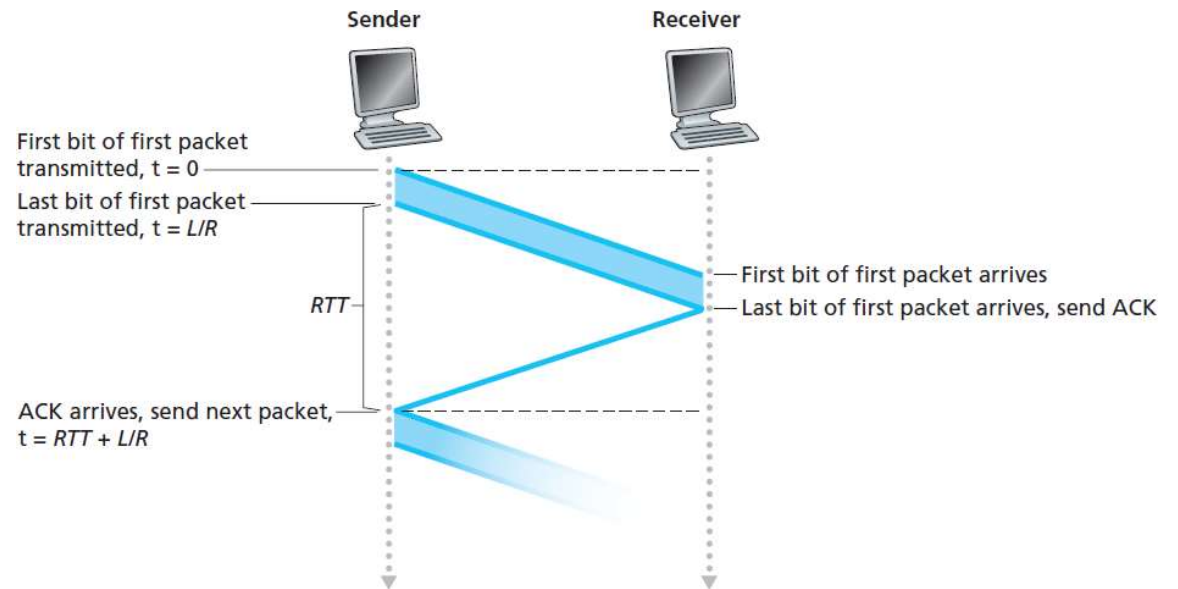


Symmetriske



# 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:

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

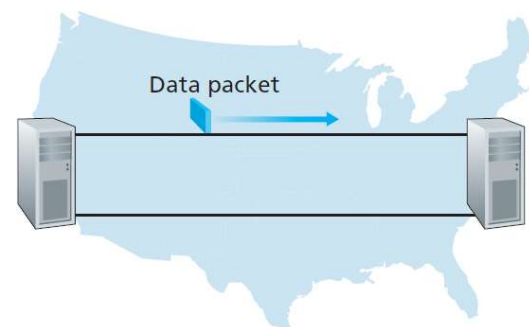
# Pipelinede protokoller

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

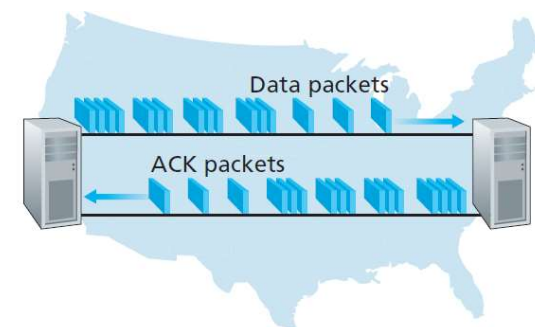
Kan vi undgå en stop&wait ?

# Pipelinede protokoller

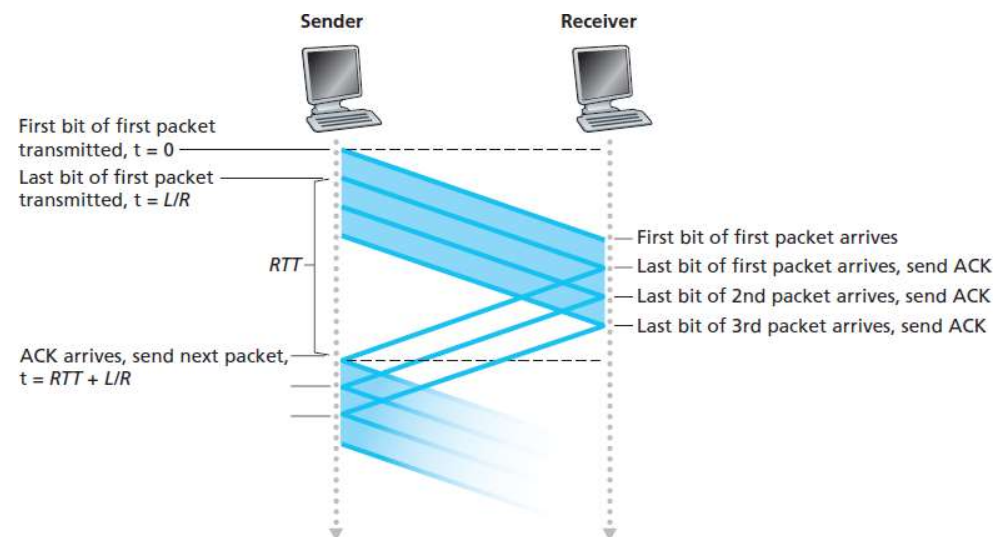
- Øg throughput (bps) ved at udsende flere pakker før vi afventer ACK
  - Udnytte kanalen
  - Uden at oversvømme modtager (flow-kontrol)
  - Uden at overbelaste netværket (congestion-kontrol)
- Pakker gemmes i buffere på sender og modtager til de er leveret
- 2 overordnede strategier for re-transmission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



a. A stop-and-wait protocol in operation



b. A pipelined protocol in operation

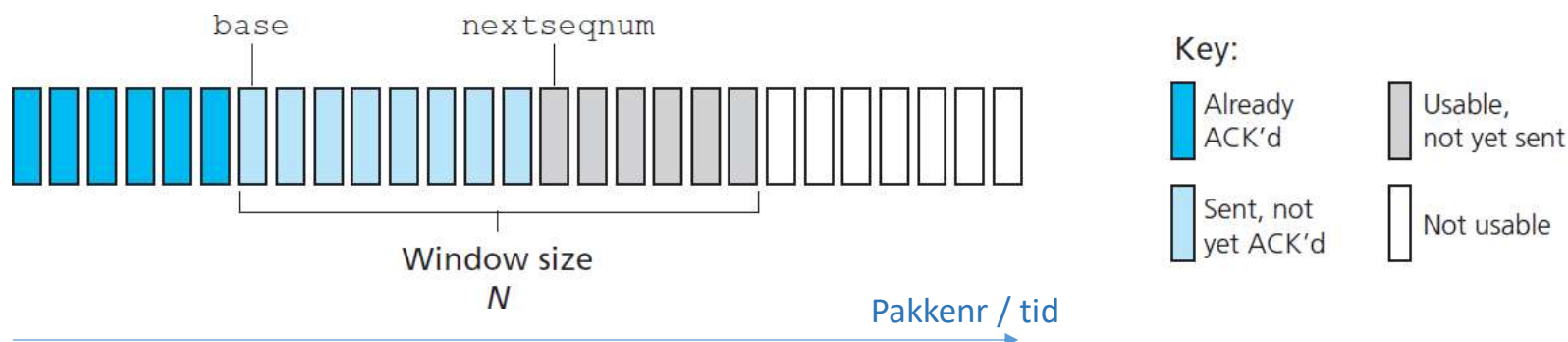


Eksempel m. trans-US link: 1 Gbps link, 15 ms prop. delay

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

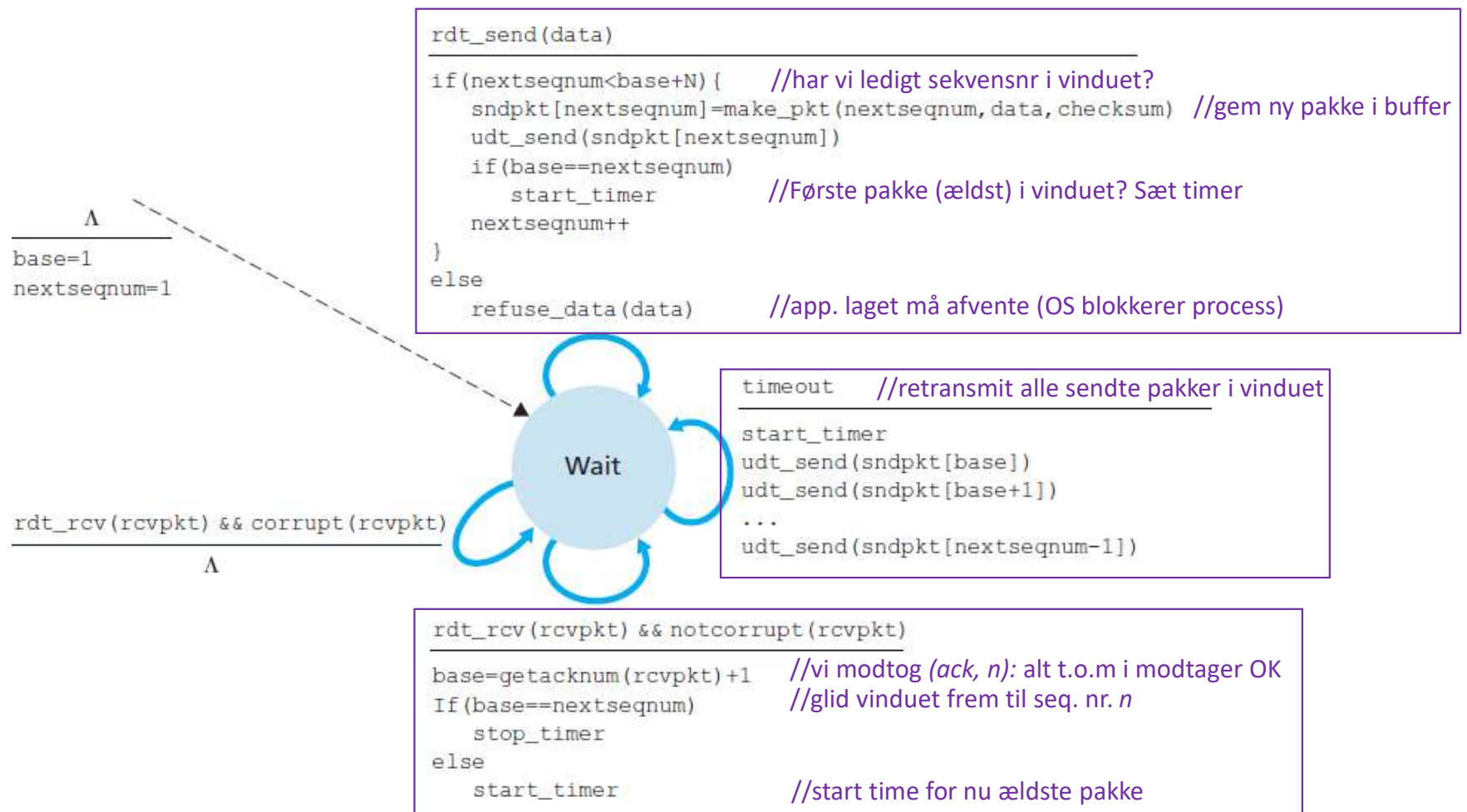
# Go-back-N

- Tillad at sender max har **N** ukvitterede pakker undervejs i "pipelinen"
- Sekvensnummer felt i header m.  $k$  bits:  $[0, 1, \dots, 2^k - 1]$
- Senders sekvensnumre:
  - **base**: start på aktuelt vindue (ældste ukvitterede pakke)
  - **nextSeqNum**: næste ledige sekvensnummer



- ACK ( $n$ ): alle pakker med sekvens numre t.o.m  $n$  er korrekt modtaget ("kumulativt" ACK)
- Sætter timer for ældste ukvitterede pakke
- Timeout => 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



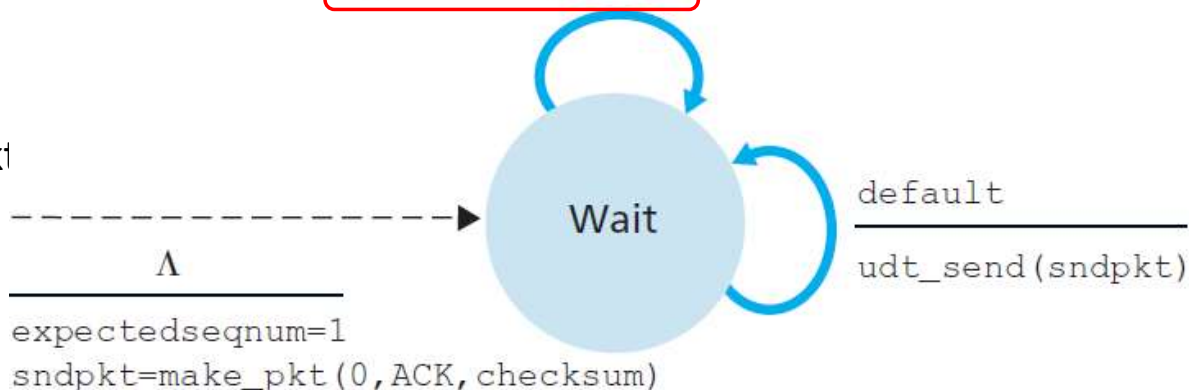
# Go-Back-N Modtager

- 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

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

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

ignore duplicate ACK



**pkt 2 timeout**

send pkt2  
 send pkt3  
 send pkt4  
 send pkt5

receiver

receive pkt0, send ack0  
 receive pkt1, send ack1  
 receive pkt3, discard,  
 (re)send ack1  
 receive pkt4, discard,  
 (re)send ack1  
 receive pkt5, discard,  
 (re)send ack1

rcv pkt2, deliver, send ack2  
 rcv pkt3, deliver, send ack3  
 rcv pkt4, deliver, send ack4  
 rcv pkt5, deliver, send ack5

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

Simpel sender/modtager!

I værste fald skal vi gå N skrid tilbage!

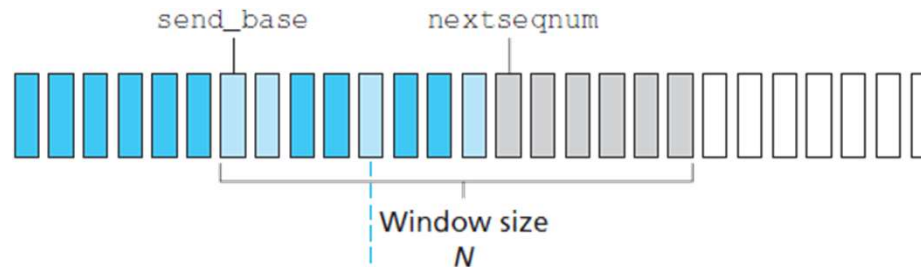
# Selective Repeat

- Tillad at sender har max ***N*** ukvitterede pakker undervejs i "pipelinen"
- 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 *individuel*t for hver modtager 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 **N** udestående pakker
- Sender og modtager vindue!
- Senders sekvensnumre:
  - **send\_base**: start på aktuelt vindue (ældste ukvitterede pakke)
  - **nextSeqNum**: næste ledige sekvensnummer
- Modtagers sekvens nr
  - **rcv\_base**: start på aktuelt vindue (ældste forventede pakke)
  - Accepterer at modtage N pakker frem

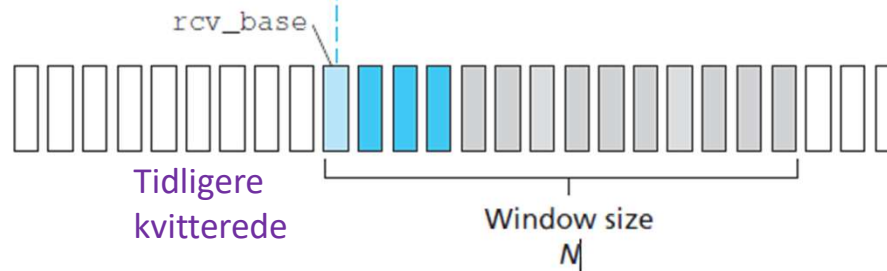


a. Sender view of sequence numbers

Key:

Already ACK'd  
 Sent, not yet ACK'd

Usable, not yet sent  
 Not usable



b. Receiver view of sequence numbers

Key:

Out of order (buffered) but already ACK'd  
 Expected, not yet received

Acceptable (within window)  
 Not usable

# 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  $[\text{sendbase}, \text{sendbase}+N-1]$ :
  - Marker  $n$  som modtaget
  - Hvis  $n$  var den mindste ukvitterede, skyd vinduet frem til næste ukvitterede pakke

## Modtager

- Modtaget pakke  $n$  i  $[\text{rcvbase}, \text{rcvbase}+N-1]:?$ 
  - 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  $[\text{rcvbase}-N, \text{rcvbase}-1]$ 
  - Send 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  
 [empty]


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

record ack3 arrived  
 pkt 2 timeout  
 send pkt2

record ack4 arrived  
 record ack5 arrived

**Q: what happens when ack2 arrives?**

receiver

receive pkt0, send ack0  
 receive pkt1, send ack1  
 [Xloss]  
 receive pkt3, buffer,  
 send ack3  
 receive pkt4, buffer,  
 send ack4  
 receive pkt5, buffer,  
 send ack5  
 rcv pkt2; deliver pkt2,  
 pkt3, pkt4, pkt5; send ack2

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

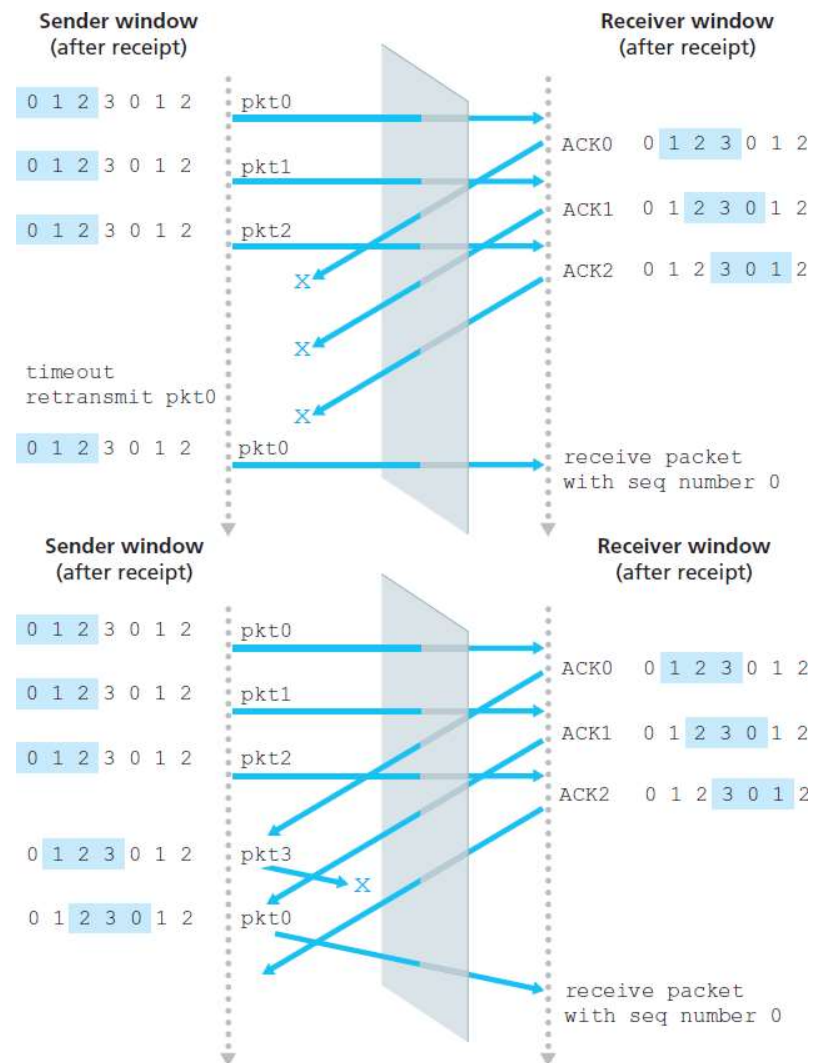
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8 9

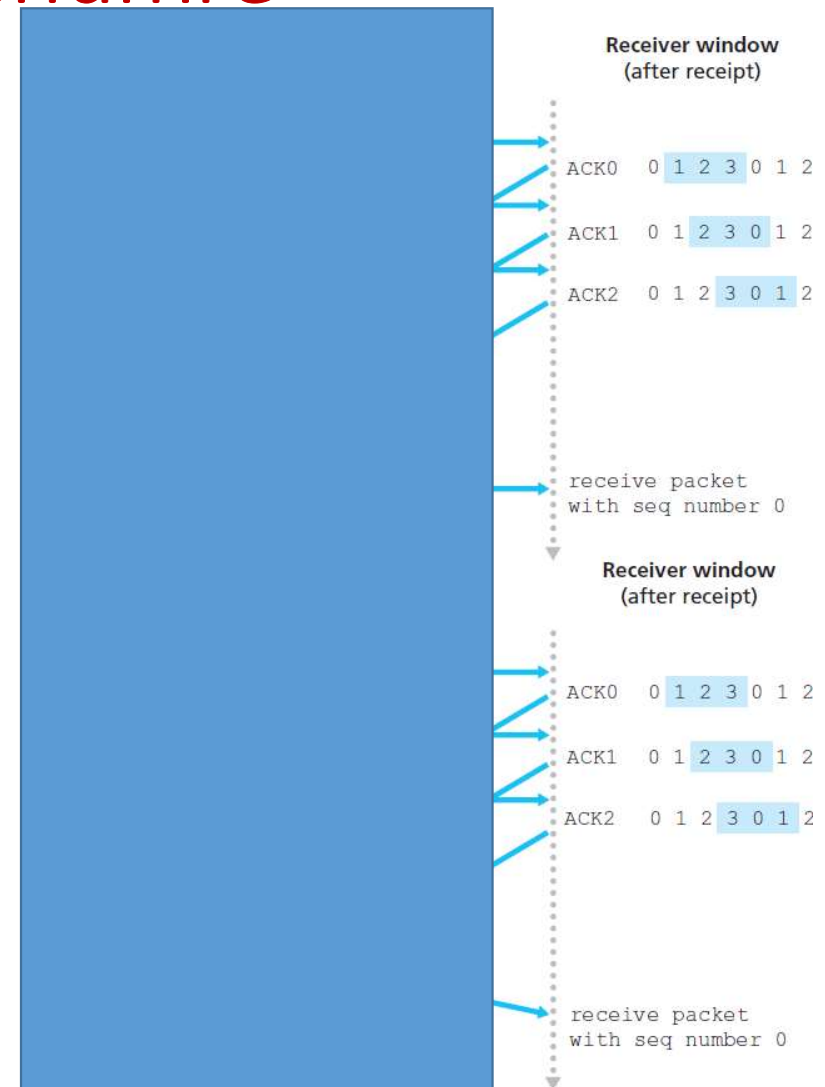
# Dilemma omkring sekvensnumre

- Header har kun plads til  $2^k - 1$  sekvensnumre
  - Wrap rundt
- 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
- 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?
  - **Uproblematiske**: Hvis den ikke er for gammel, så bufferer 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)
    - 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!

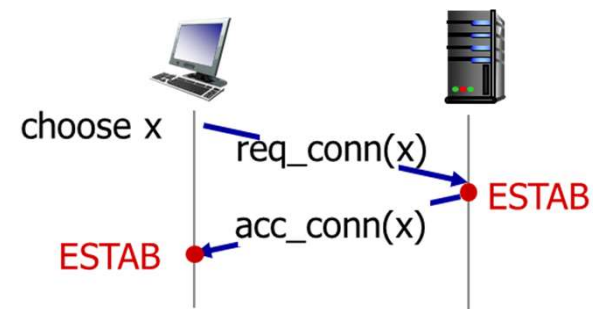
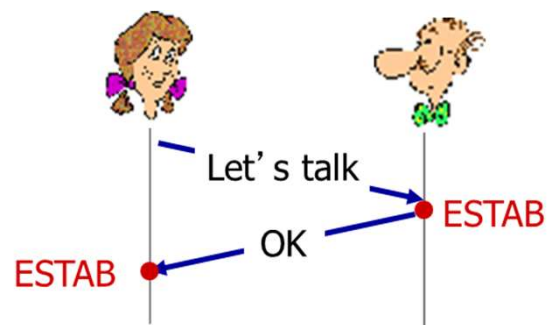
# Etablering af TCP forbindelser

Hvordan forbinder en klient sig til en server?

Hvordan stopper de kommunikationen og nedriver forbindelsen igen?

# Etablering og nedlægning af forbindelser

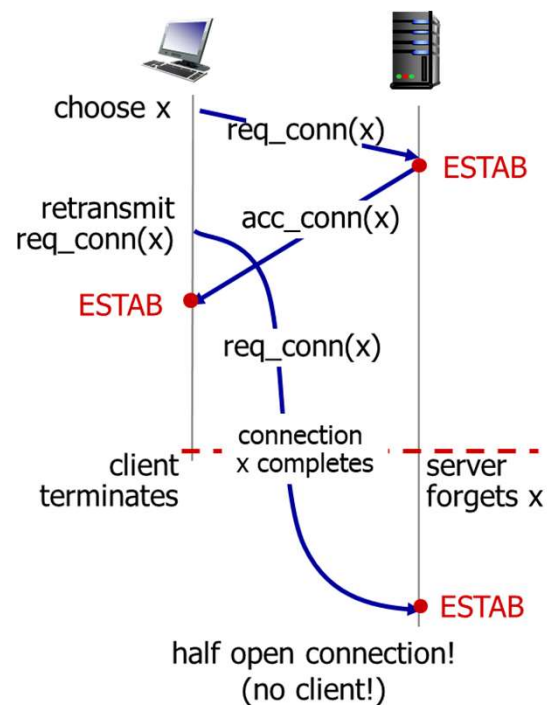
- Sender Klient og server skal blive enige om at de har en forbindelse
  - Afsætte buffer-plads;
  - Initialisere "sliding window" parametre: sekvens nummer + vindues størrelse
  - I begge retninger da TCP er bi-direktionel
- Komplikationer
  - Gamle pakker (fx fra re-transmissioner på tidligere forbindelse)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



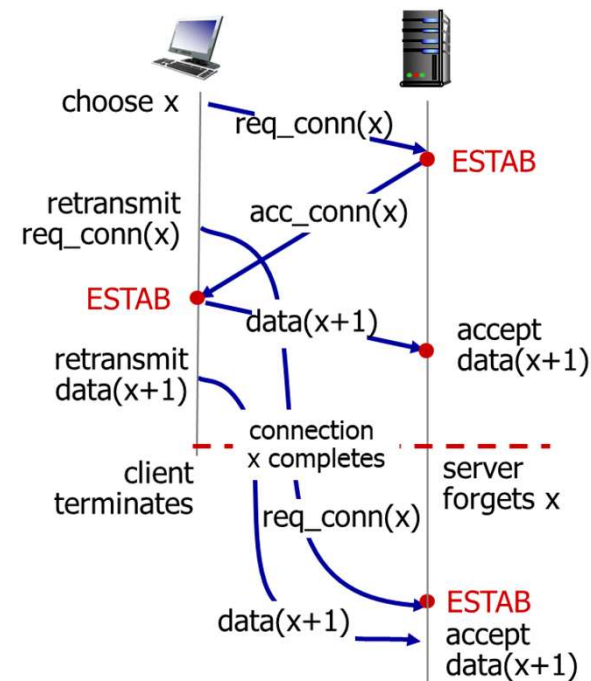


# 2-vejs handshake ?

- Eksempler på problematiske scenarier for



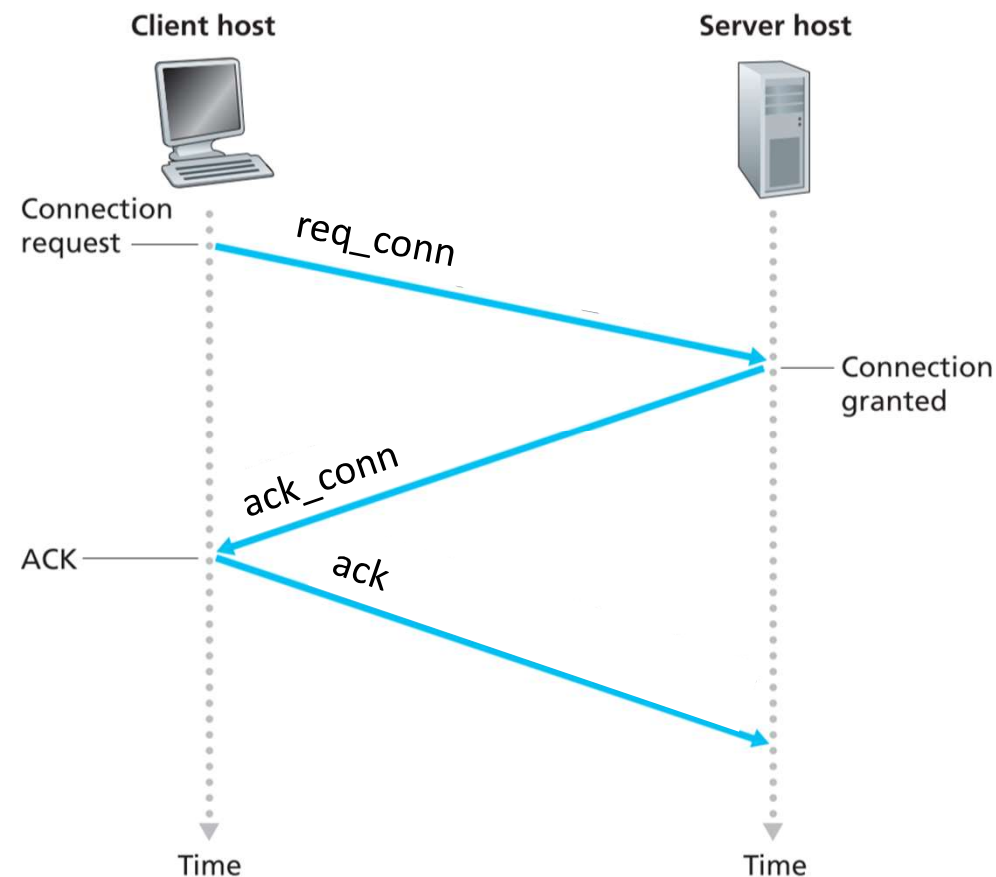
Server afsætter ressourcer til klient, der ikke findes



Server modtager gammel data.

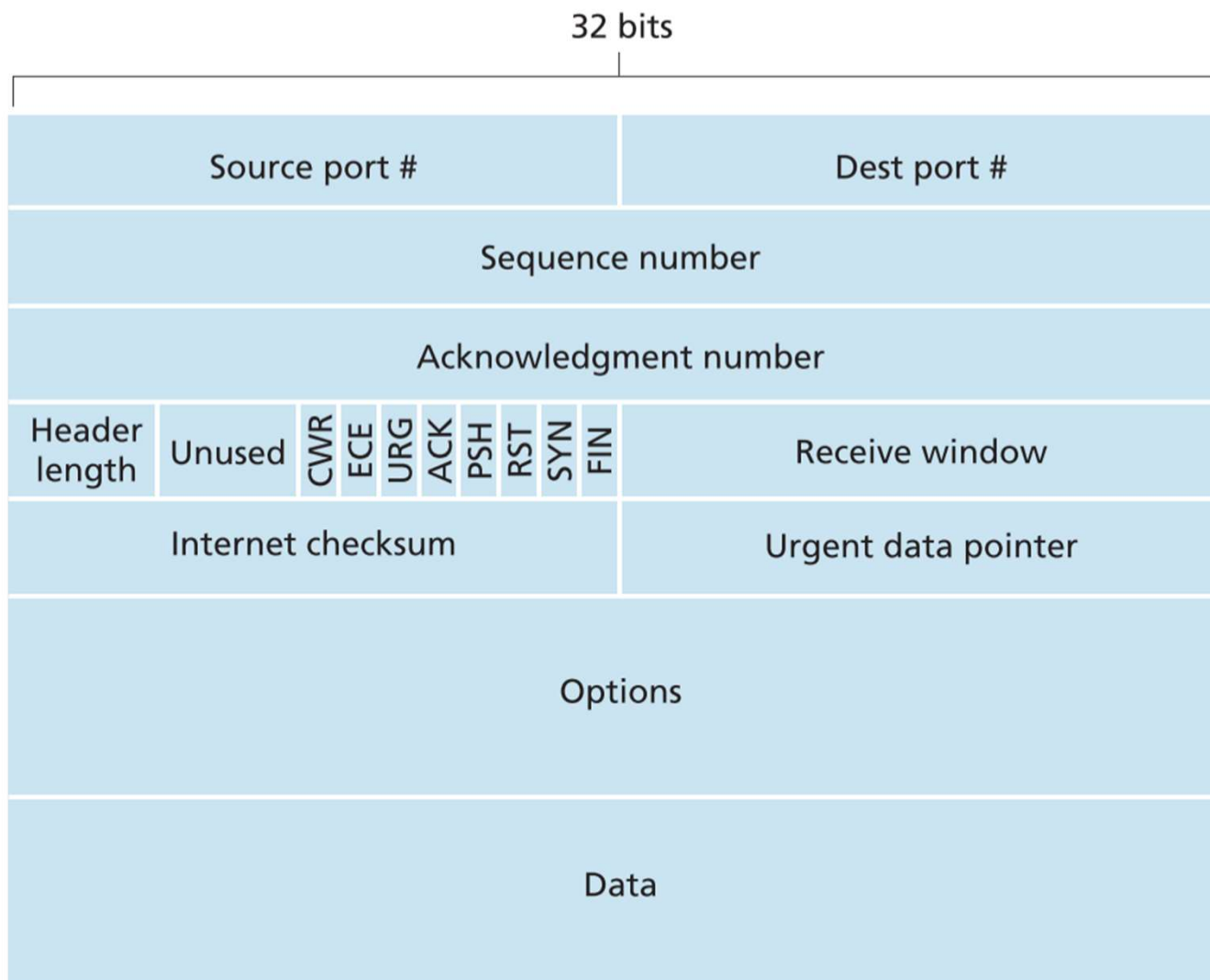
# 3-way-handshake

- 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

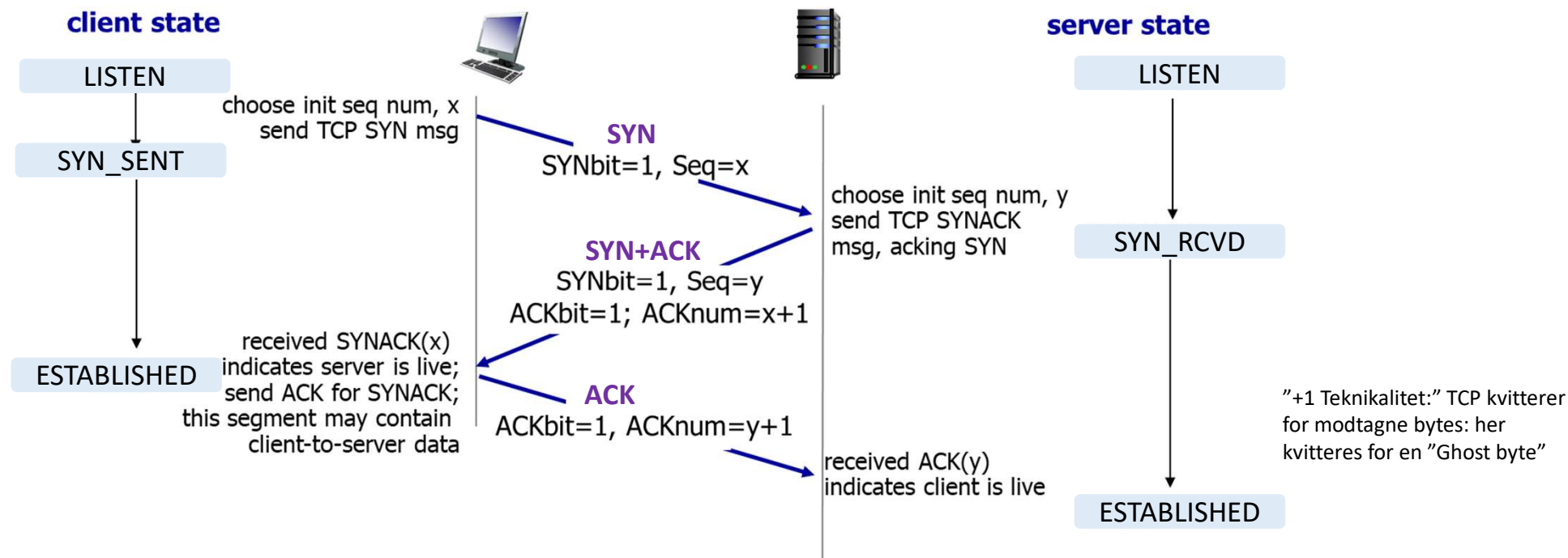
- TCP *Segment*
- Kontrol-bits til etablering og nedrivning af forbindelser:
  - SYNchronize
  - FINish
  - ACKnowledge (også til data)
    - ACK=1  $\Rightarrow$  gyldig info i ack no feltet.
- Sekvens og ack. numre



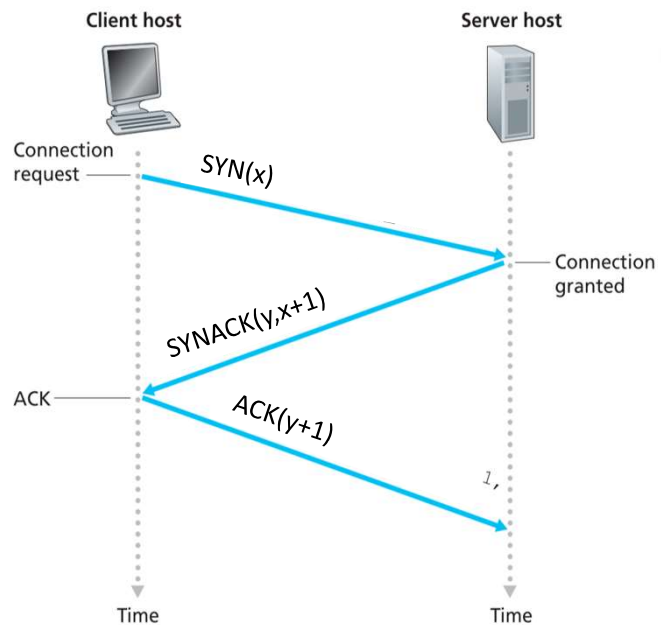
# 3-way-handshake

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

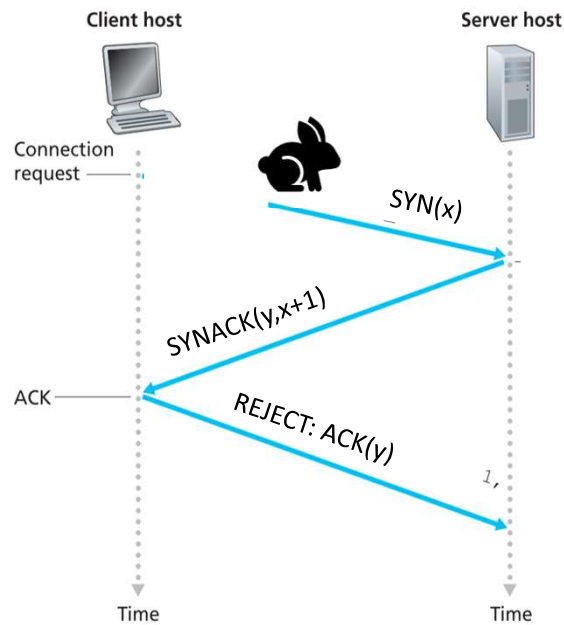
- **req\_conn**: TCP **SYN** segment (SYN=1, Seq=x):
  - x=klients (tilfældigt) valgte init sekvensnr
- **ack\_conn**: TCP **SYN+ACK** segment (SYN=1, ACK=1, Seq=y, AckNo=x+1):
  - y=servers (tilfældigt) valgte sekvensnummer
  - Server bekræfter; forventer at næste segment nr. har sek. x+1
- **ack**: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



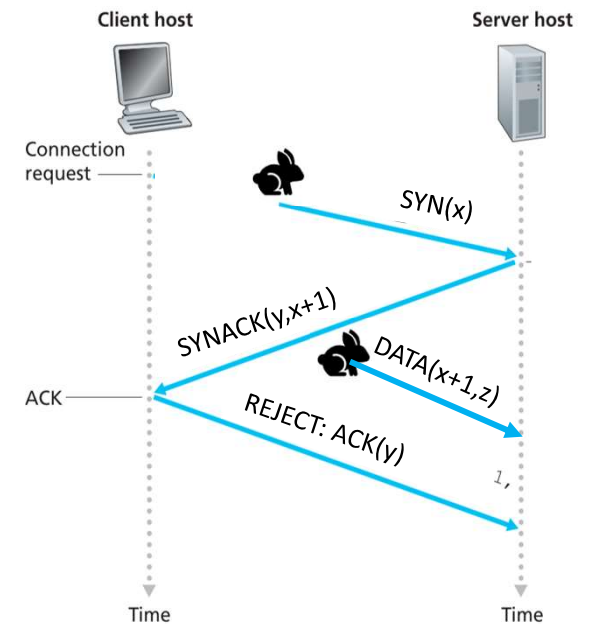
# 3-way-handshake



Normal



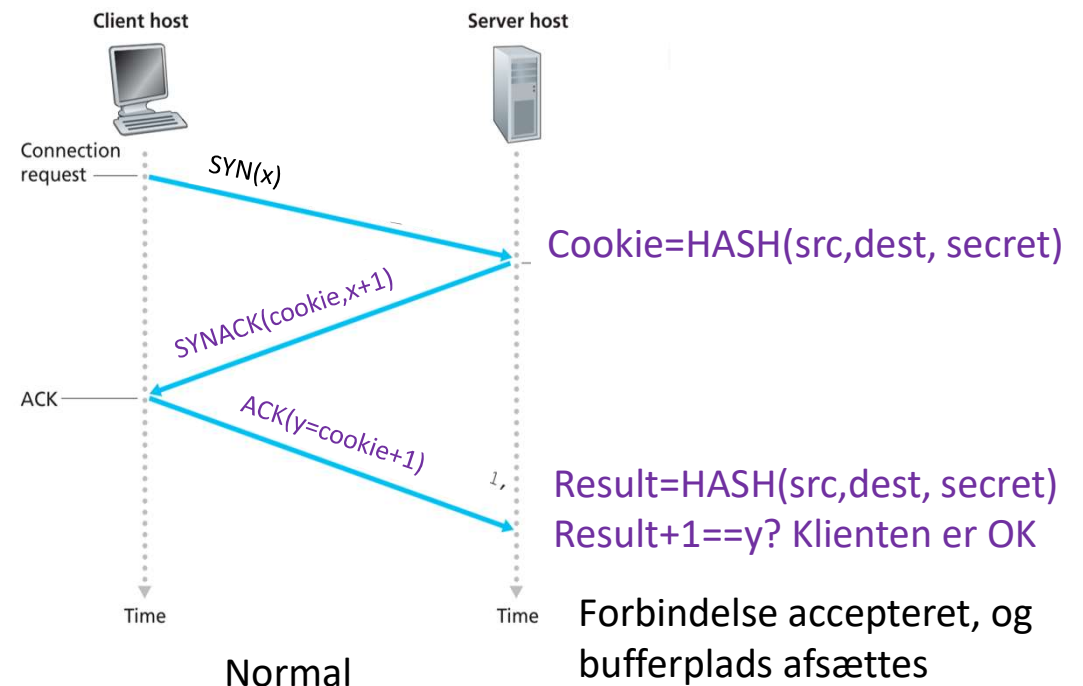
Gammel SYN segment:  
forbindelse afvises



Gammel SYN og Data:  
forbindelse afvises

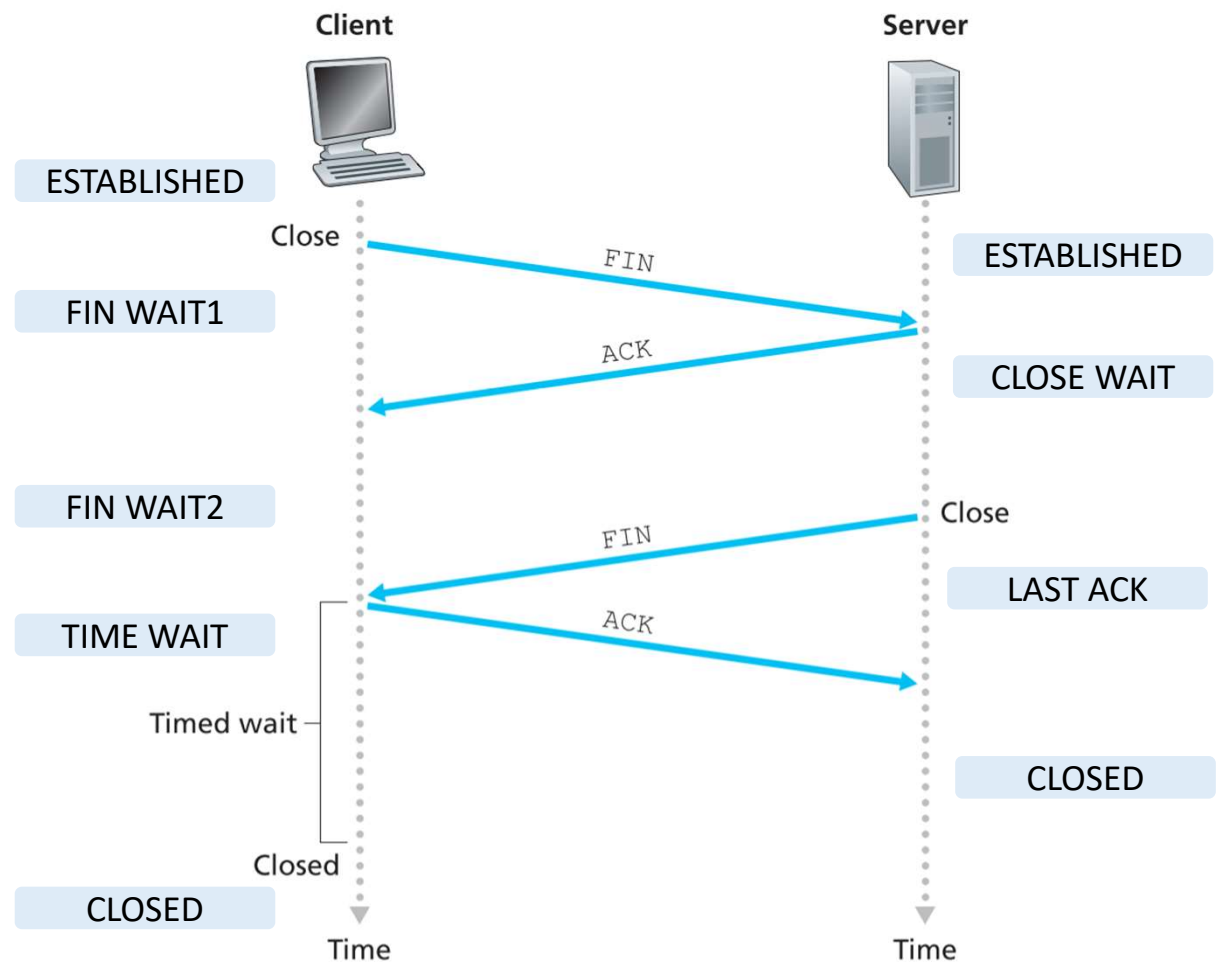
# SYN-flood angreb

- 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



# 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")



SLUT