Westerdals

Oslo School of Arts,
Communication and Technology

TK1100

TRANSPORTLAGET

8’ende forelesning

Del eksamen 1 - resultater

Antall studenter i klassen: 236

Antall fremmøtte eksamen: 219

Karakter A:	5%	90 – 100 poeng
Karakter B:	21%	75 – 89.5 poeng
Karakter C:	35%	60 – 74.5 poeng
Karakter D:	20%	45 – 59.5 poeng
Karakter E:	14%	30 – 44.5 poeng
Karakter F:	5%	0 – 29.5 poeng

Westerdals
Oslo A.C.T
Avdeling for Teknologi

Del eksamen 1 - resultater

- Tok du ikke eksamen?
  - Ta kontakt med studieadministrasjonen ASAP
  - Har du legeerklæring må du sende den ASAP
- Strøk du på eksamen?
  - Fortsett å følge forelesningene, du trenger alt du lærer i TK1100 for å mestre andre fag!
  - Ta kontakt med studieadministrasjonen, du skal kunne ta deleksamen 2 som normalt, og så kun konte denne du strøk
- Fikk du karakter A eller B?
  - Gratulerer, første eksamen bestått med glans!
- Fikk du karakter C?
  - Hvis du jobber hardt og klarer del eksamen 2 med perfekt besvarelse kan du kanskje fortsatt få en toppkarakter
- Fikk du karakter D eller E?
  - Du må nok jobbe mye mer med faget, hvis du fikk en E og i tillegg vet med deg selv at du svarte tynnt så var jeg snill med noen studenter – like tynn besvarelse til jul kan gi en F…

Westerdals
Oslo A.C.T
Avdeling for Teknologi

VIKTIG INFORMASJON

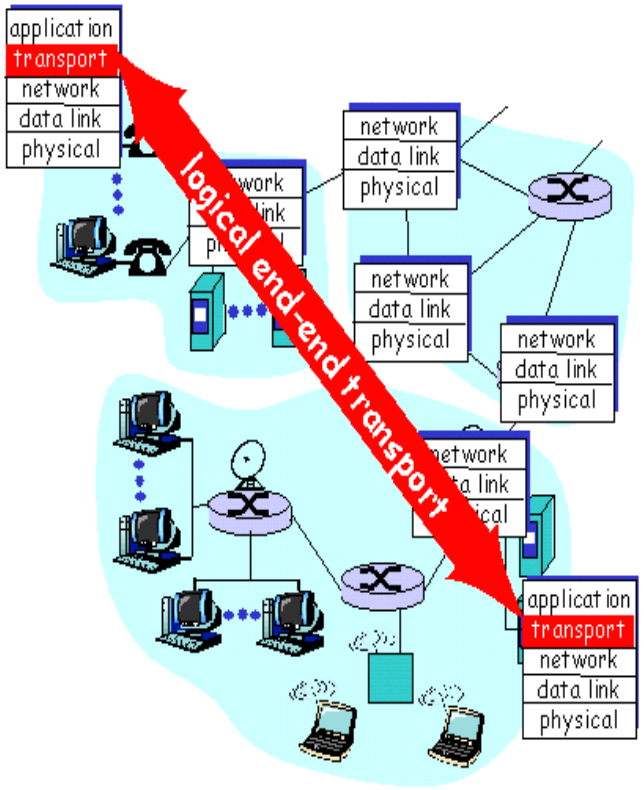
- 6. november; Nettverkslaget
  - Posthallen, ikke i Auditoriet 101
- 26. november; Linklaget 12.15 – 14.00
  Repetisjon 15.15 – 19.00
  - Obs: Dobbelt time
  - Obs: Mandag

Transportlaget: Agenda

1. Transportlagets tjenester
   - Multipleksing/demultipleksing
     - Portnummer
   - `netstat` (standard verktøy)
   - Transport uten fast forbindelse: **UDP**
2. Prinsipper for pålitelig dataoverføring
3. Transport med «fast» forbindelse: TCP
   - Pålitelig overføring
   - Flyt-kontroll
   - Kontroll og styring av forbindelsen

Transport tjenesten

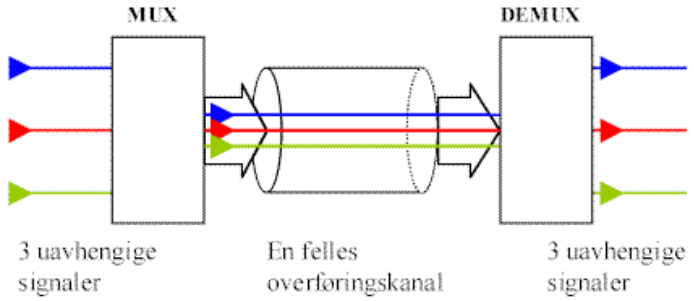
- Setter opp *logisk* kommunikasjon mellom applikasjons-prosesser på forskjellige klienter
- Transport-protokollen kjøres i hvert ende-system
- Transportlags-protokoll
  - Dataoverføring mellom prosesser
- Bruker Nettverklags-protokoll (IP)
  - Dataoverføring mellom systemer (vertsmaskiner)



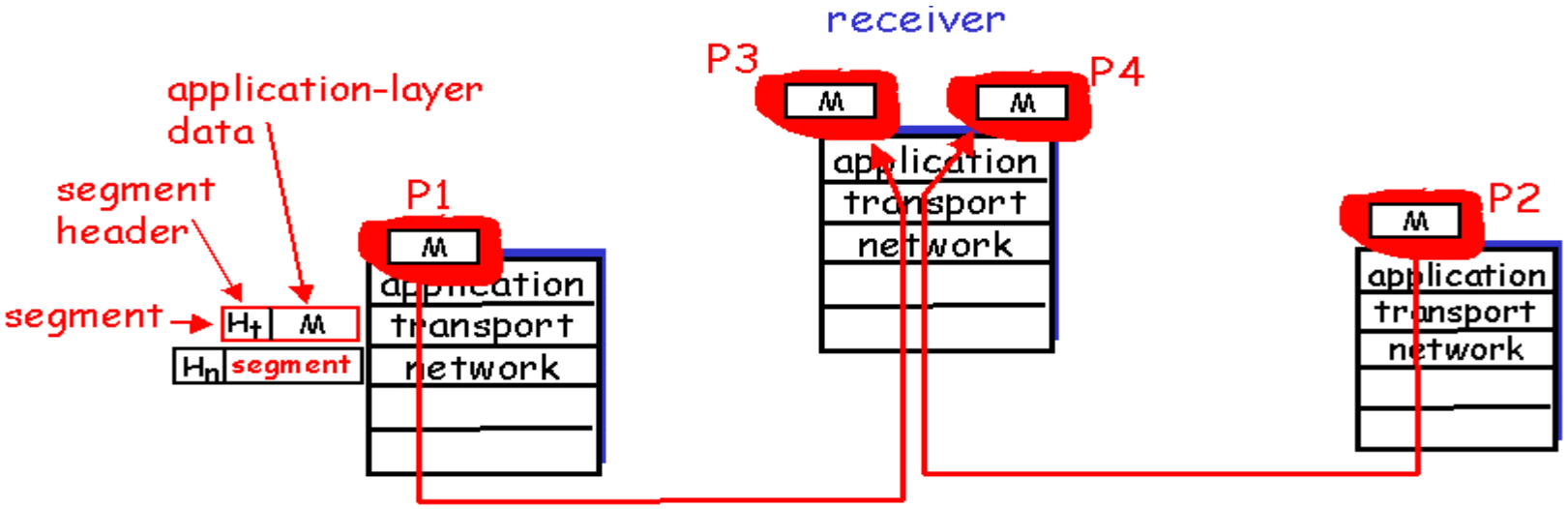
Transport-lag protokoller

- Internett bruker nettverks-protokollen IP
  - Gjør så godt den kan, men gir ingen garantier
  - «best effort»
- UDP
  - Sender et datagram, som kan bestå av flere deler, til mottaker og håper det kommer frem
  - Forbedrer IP bare med ende-til-ende kontroll og feil-sjekking
- TCP
  - Oppretter en ”fast” forbindelse
  - Legger inn flyt-kontroll,
    sekvens-nummer,
    kvittering,
    tidskontroll,
    feilsjekking og
    kontroll av trafikk-kork (metningskontroll)

Multipleksing/ demultipleksing

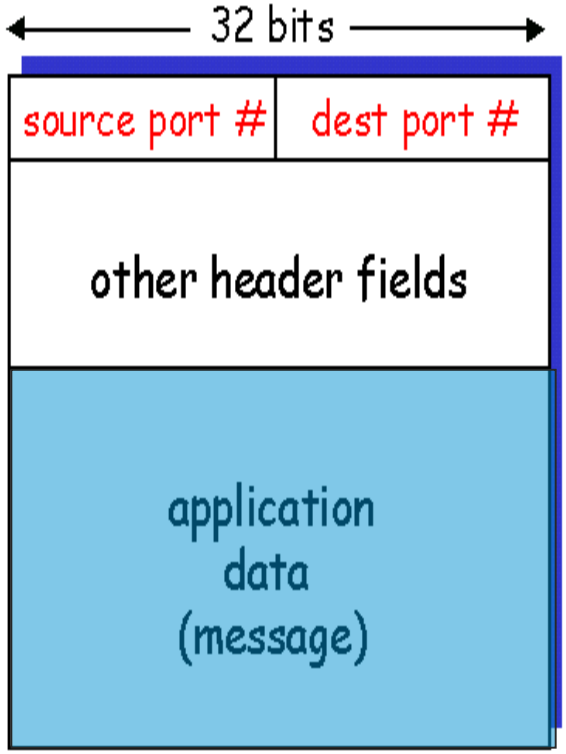


- Segment
  - Data-enhet som utveksles mellom transportlagene
  - TPDU (Transport Protocol Data Unit)
- **Demultipleksing**
  - Levere motatte segmenter til riktig prosess



Multipleksing <- portnummer

- Samler data fra applikasjons-prosesser og pakker disse med et hode (header)
- Hodet inneholder **senders** og **mottakers** portnummer
- Portnummer = **16 bit** unsigned **heltall**
- Portene **0-1023** er «**well known**» (RFC 1700)
  - Secure Shell: port 22
  - SMTP: port 25
  - DNS: port 53
  - HTTP: port 80
  - HTTP over TLS/SSL: port 443
- Andre porter deles opp i:
  - **Registrerte**
    - 1024-49151 (0x0400-0xBFFF)
    - Kan brukes til annet også, men er **registrert** for en tjeneste hos IANA
  - **Private/Dynamiske**:
    - 49152-65535 (0xC000-0xFFFF)



TCP/UDP segment format

Transportlaget 1

- Den første hovedoppgaven som løses på transportlaget er dermed å multiplexe/demultiplexe fra/til lokale prosesser og den felles kanalen (Internett)!

- Portnummer fungerer som **ID-nummer** for lokal og kontaktet prosess!

netstat

```
C:\>netstat -n

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    127.0.0.1:19872        127.0.0.1:55169        ESTABLISHED
  TCP    127.0.0.1:27015        127.0.0.1:55155        ESTABLISHED
  TCP    127.0.0.1:52965        127.0.0.1:52967        ESTABLISHED
  TCP    127.0.0.1:52967        127.0.0.1:52965        ESTABLISHED
  TCP    127.0.0.1:55155        127.0.0.1:27015        ESTABLISHED
  TCP    127.0.0.1:55169        127.0.0.1:19872        ESTABLISHED
  TCP
  TCP
  TCP
  TCP
  TCP    158.36.131.51:55812    174.36.30.34:80        ESTABLISHED
  TCP
  TCP    158.36.131.51:76892                           ESTABLISHED
  TCP
  TCP
  TCP    158.36.131.51:61306    174.129.195.86:443     CLOSE_WAIT
  TCP    158.36.131.51:62305    74.125.79.118:443      ESTABLISHED
  TCP    158.36.131.51:62434    158.36.191.141:443     ESTABLISHED
```

Klient IP-adresse: portnummer

Server IP-adresse: portnummer

Forbindelsens Tilstand

Windows

`netstat` er kommandoen som gir en oversikt over åpne porter:

`netstat -a` : også UDP

`netstat -s` : statistikk

`netstat -r` : routing tabell

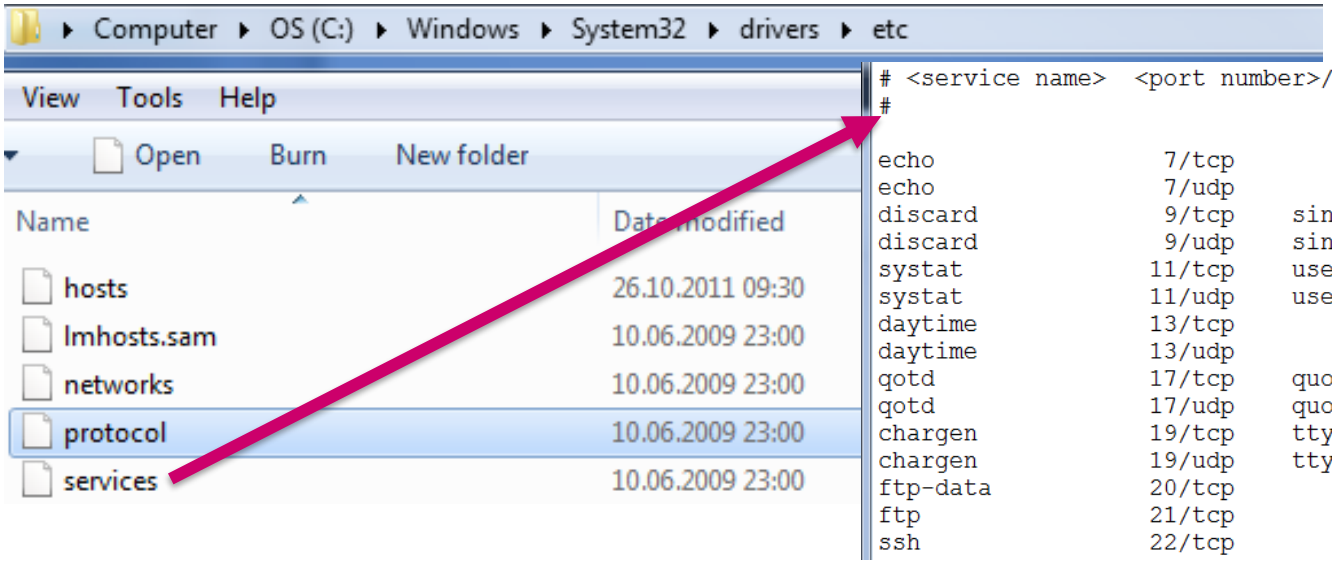
`netstat -n` : ikke DNS-navn

+ mange flere for å se hvilken prosess o.l. (–b på Win7 osv)

```
~->netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address               Foreign Address             State
tcp        0      0 localhost.localdomain:smtp  localhost.localdomain:36098 TIME_WAIT
tcp        0      0 nih-stud-web02.osl.ba:44573 nih-mysql01.osl.basef:mysql TIME_WAIT
tcp        0      0 nih-stud-web02.osl.ba:44572 nih-mysql01.osl.basef:mysql TIME_WAIT
tcp        0      0 nih-stud-web02.osl.base:ssh nith-vpn-nat03.osl.ba:29459 ESTABLISHED
Active UNIX domain sockets (w/o servers)
Proto RefCnt Flags       Type       State         I-Node Path
unix  14     [ ]         DGRAM                    9016   /dev/log
unix  2      [ ]         DGRAM                    1855   @/org/kernel/udev/udevd
```

OSX

Oversikt: Services-listen



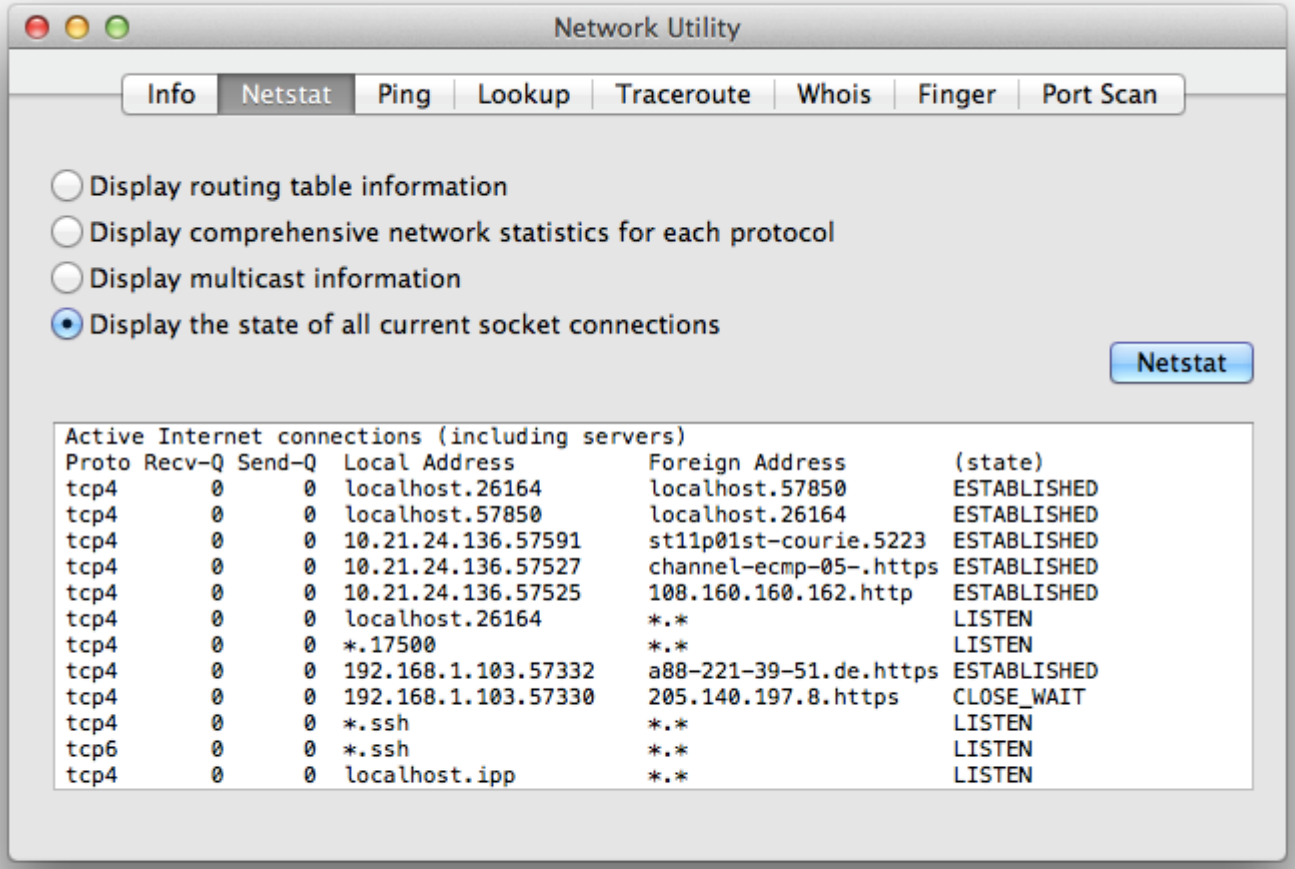

```
# <service name>  <port number>/<protocol>  [aliases...]   [#<comment>]
#

echo                7/tcp
echo                7/udp
discard             9/tcp    sink null
discard             9/udp    sink null
systat             11/tcp    users                    #Active users
systat             11/udp    users                    #Active users
daytime            13/tcp
daytime            13/udp
qotd               17/tcp    quote                    #Quote of the day
qotd               17/udp    quote                    #Quote of the day
chargen            19/tcp    ttytst source            #Character generator
chargen            19/udp    ttytst source            #Character generator
ftp-data           20/tcp                             #FTP, data
ftp                21/tcp                             #FTP. control
ssh                22/tcp                             #SSH Remote Login
```

- Både i Windows (se over) og i OSX (/etc) ligger en liste (`services`) med hvilke porter som er registrert med hvilke protokoller/tjenester
  - Det er denne som bestemmer hva som oppgis som protokollnavn av `netstat`

```
tcp        0      0 home.nith.no:ssh        blistog.nith.no:24606   ESTABLISHED
```

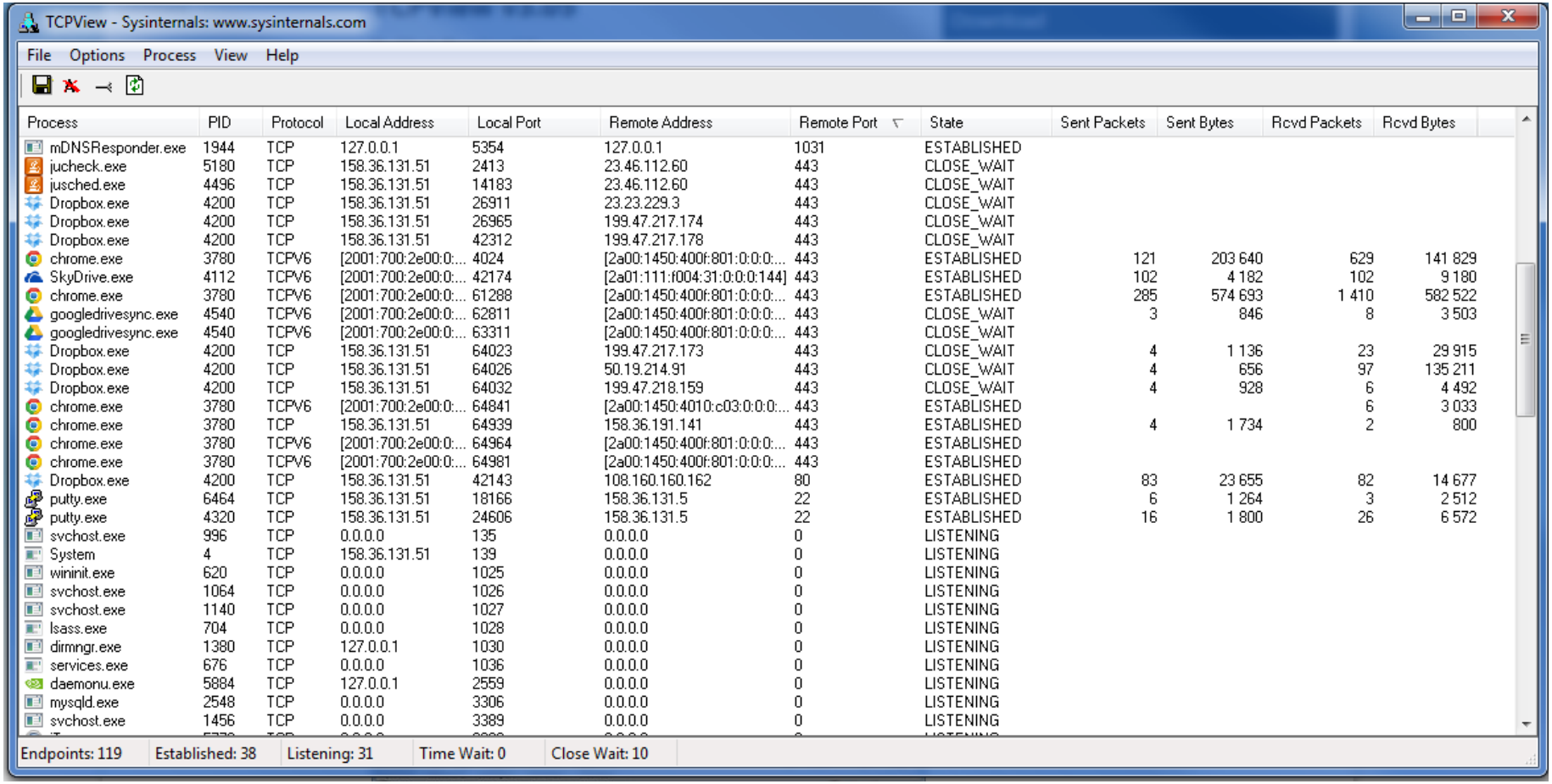
OSX: Network Utility

- OSX (under Applications/Utilities) har et GUI mot standardverktøyene



Win: TCPView

- Fra MS kan du laste ned TCPView
- GUI som viser «de fleste» tingene du kan finne med netstat



Process	PID	Protocol	Local Address	Local Port	Remote Address	Remote Port	State	Sent Packets	Sent Bytes	Rcvd Packets	Rcvd Bytes
mDNSResponder.exe	1944	TCP	127.0.0.1	5354	127.0.0.1	1031	ESTABLISHED				
jucheck.exe	5180	TCP	158.36.131.51	2413	23.46.112.60	443	CLOSE_WAIT				
jusched.exe	4496	TCP	158.36.131.51	14183	23.46.112.60	443	CLOSE_WAIT				
Dropbox.exe	4200	TCP	158.36.131.51	26911	23.23.229.3	443	CLOSE_WAIT				
Dropbox.exe	4200	TCP	158.36.131.51	26965	199.47.217.174	443	CLOSE_WAIT				
Dropbox.exe	4200	TCP	158.36.131.51	42312	199.47.217.178	443	CLOSE_WAIT				
chrome.exe	3780	TCPV6	[2001:700:2e00:0:...	4024	[2a00:1450:400f:801:0:0:0:...	443	ESTABLISHED	121	203 640	629	141 829
SkyDrive.exe	4112	TCPV6	[2001:700:2e00:0:...	42174	[2a01:111:f004:31:0:0:0:144]	443	ESTABLISHED	102	4 182	102	9 180
chrome.exe	3780	TCPV6	[2001:700:2e00:0:...	61288	[2a00:1450:400f:801:0:0:0:...	443	ESTABLISHED	285	574 693	1 410	582 522
googledrivesync.exe	4540	TCPV6	[2001:700:2e00:0:...	62811	[2a00:1450:400f:801:0:0:0:...	443	CLOSE_WAIT	3	846	8	3 503
googledrivesync.exe	4540	TCPV6	[2001:700:2e00:0:...	63311	[2a00:1450:400f:801:0:0:0:...	443	CLOSE_WAIT				
Dropbox.exe	4200	TCP	158.36.131.51	64023	199.47.217.173	443	CLOSE_WAIT	4	1 136	23	29 915
Dropbox.exe	4200	TCP	158.36.131.51	64026	50.19.214.91	443	CLOSE_WAIT	4	656	97	135 211
Dropbox.exe	4200	TCP	158.36.131.51	64032	199.47.218.159	443	CLOSE_WAIT	4	928	6	4 492
chrome.exe	3780	TCPV6	[2001:700:2e00:0:...	64841	[2a00:1450:4010:c03:0:0:0:...	443	ESTABLISHED			6	3 033
chrome.exe	3780	TCP	158.36.131.51	64939	158.36.191.141	443	ESTABLISHED	4	1 734	2	800
chrome.exe	3780	TCPV6	[2001:700:2e00:0:...	64964	[2a00:1450:400f:801:0:0:0:...	443	ESTABLISHED				
chrome.exe	3780	TCPV6	[2001:700:2e00:0:...	64981	[2a00:1450:400f:801:0:0:0:...	443	ESTABLISHED				
Dropbox.exe	4200	TCP	158.36.131.51	42143	108.160.160.162	80	ESTABLISHED	83	23 655	82	14 677
putty.exe	6464	TCP	158.36.131.51	18166	158.36.131.5	22	ESTABLISHED	6	1 264	3	2 512
putty.exe	4320	TCP	158.36.131.51	24606	158.36.131.5	22	ESTABLISHED	16	1 800	26	6 572
svchost.exe	996	TCP	0.0.0.0	135	0.0.0.0	0	LISTENING				
System	4	TCP	158.36.131.51	139	0.0.0.0	0	LISTENING				
wininit.exe	620	TCP	0.0.0.0	1025	0.0.0.0	0	LISTENING				
svchost.exe	1064	TCP	0.0.0.0	1026	0.0.0.0	0	LISTENING				
svchost.exe	1140	TCP	0.0.0.0	1027	0.0.0.0	0	LISTENING				
lsass.exe	704	TCP	0.0.0.0	1028	0.0.0.0	0	LISTENING				
dirmngr.exe	1380	TCP	127.0.0.1	1030	0.0.0.0	0	LISTENING				
services.exe	676	TCP	0.0.0.0	1036	0.0.0.0	0	LISTENING				
daemonu.exe	5884	TCP	127.0.0.1	2559	0.0.0.0	0	LISTENING				
mysqld.exe	2548	TCP	0.0.0.0	3306	0.0.0.0	0	LISTENING				
svchost.exe	1456	TCP	0.0.0.0	3389	0.0.0.0	0	LISTENING				

Endpoints: 119 | Established: 38 | Listening: 31 | Time Wait: 0 | Close Wait: 10

Porter og sikkerhet (brannmurer)

- Tilgangen til en kjørende applikasjon er via et portnummer (jf erfaringer med HTTP i øvinger)
- Portnummer kan dermed fortelle oss ganske mye om hvilke programmer som kjører på en maskin

- Vanlig verktøy for portscanning er:

  nmap
- Portscanning vil kunne fortelle deg (mye) om OS og hvilke tjenester det kjører
- De fleste software-brannmurer er der for å sørge for å filtrere ut slike «uønskede» forespørsler.

```
~->nmap 158.36.131.51

Starting Nmap 5.00 ( http://nmap.org ) at 2011-11-02 14:29 CET
Interesting ports on blistog.nith.no (158.36.131.51):
Not shown: 994 filtered ports
PORT     STATE SERVICE
80/tcp   open  http
135/tcp  open  msrpc
139/tcp  open  netbios-ssn
443/tcp  open  https
445/tcp  open  microsoft-ds
3389/tcp open  ms-term-serv

Nmap done: 1 IP address (1 host up) scanned in 4.75 seconds
```

U NIFIED

D ATAGRAM

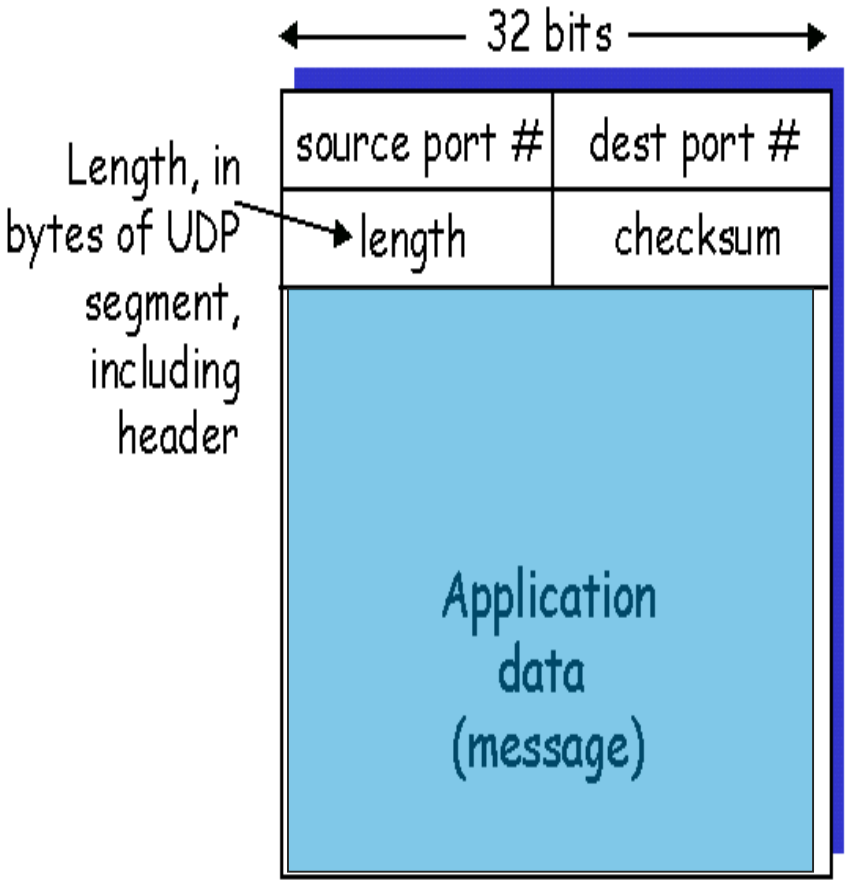
P ROTOCOL

UDP (User Datagram Protocol)

- Meget enkel Internett transportlag protokoll
  - Segmenter kan gå tapt
  - Segmenter kan leveres i feil rekkefølge
  - Ikke handshake mellom avsender og mottaker
- Hvorfor UDP?
  - Ingen etablering av forbindelse - ingen forsinkelse
  - Setter ikke opp noen felles tilstand for avsender og mottaker
  - Lite segment-hode
  - Ingen kontroll av trafikk-kork
  - Muliggjør broadcasting
  - Skulle du ha behov for pålitelighet kan du jo legge det inn i programmet på applikasjonslags nivå

UDP

- Brukes ofte i forbindelse med multimedia hvor den menneskelige hjerne kan korrigere feilene
- Andre bruksområder
  - DNS
  - SNMP, ICMP
- Mottakerens applikasjon kan besørge feil-håndtering



UDP segment format

UDP sjekksum

- Avsender
  - Oppfatter segmentet som sammensatt av 16 bits ord
  - Summerer alle ordene
  - Tar 1's komplement av summen (flipper)
  - Setter sjekksummen inn i headeren på segmentet
- Mottaker
  - Summerer alle 16 bits ordene i mottatt segment, inkl. sjekksummen
  - dersom sum = 1111 1111 1111 1111 => alt OK
- I beste fall gir dette bare en indikasjon på om feil er oppstått under overføringen

Ex: Internet sjekksummen

Merk: Mente i mest signifikante posisjon legges til LSb (Minst signifikante bit) !

Ex: To 16 bit deler av samlet pakke legges sammen

	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
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
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

HVORDAN OPPNÅ PÅLITELIG OVERFØRING GJENNOM EN UPÅLITELIG KANAL?

PRINSIPPENE BAK...

TRANSMISSION

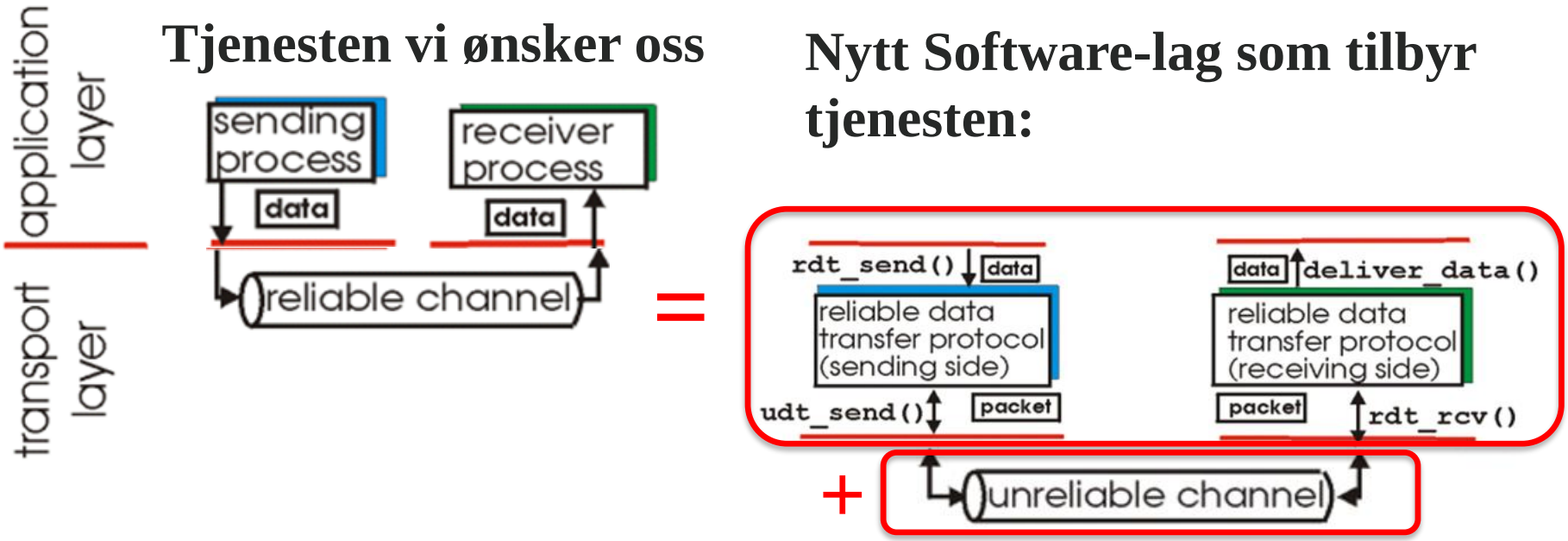
CONTROL

PROTOCOL

Pålitelig overføring

- UDP sikrer mux/demux
- UDP gjør det bare mulig for mottager å oppdage at bit-feil i en pakke kan ha oppstått
  - Ved feil er det vanlige at OS da dropper pakken
  - Ingen feilhåndtering e.l.
- TCP skal tilby en pålitelig forbindelse.
  - Da trenger må vi ta hensyn til de ulike feilkildene (støy/bitfeil, tap, …) og hvordan vi skal håndtere dem

Pålitelighet er å håndtere feil



- Dersom kanalen er pålitelig trenger vi bare mux/demux
- Det vi ønsker er å tilby tjenesten **pålitelig overføring** over en **upålitelig kanal**
- Da må vil lage en protokoll og software (metoder) som sørger dette.
- Kostand: Større **kompleksitet**

M.a.o.: Problem & Løsning

- Problem:
  - Vil ha pålitelig overføring av data over nettverk bygd opp av upålitelige media

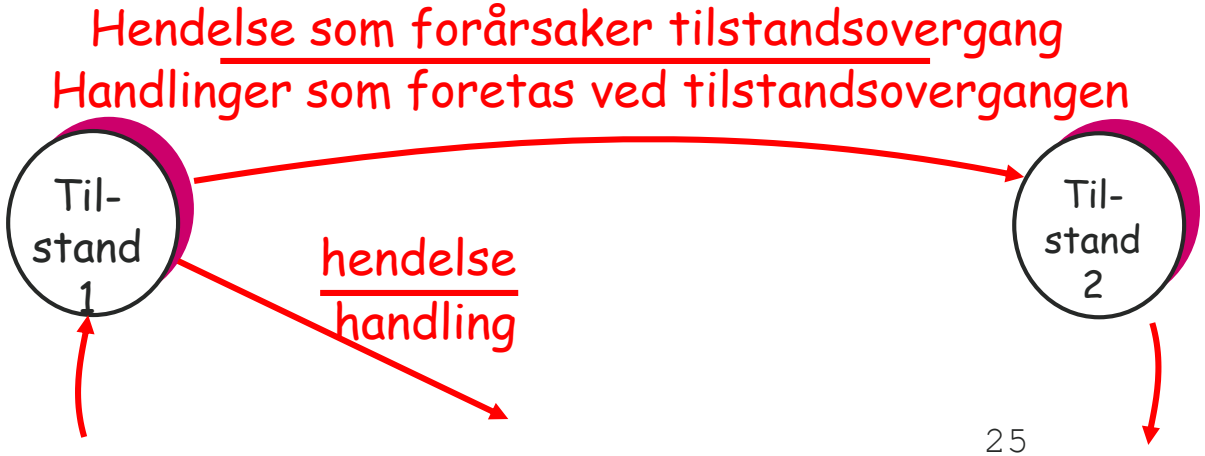
- Løsningsstrategi:
  1. Starter med idealtilstanden (media perfekt)
  2. Introduserer problemene forbundet med reelle media (støy og tap) **ett** for **ett**
  3. Konstruerer trinnvis en protokoll som håndterer problemene.

Forskjellige grader av pålitelighet

- Vi lager nå en «leke-protokoll», som vi kaller RDT (Reliable Data Transport)
- Vi skal se på forskjellige nivåer av RDT og bygge disse opp gradvis
- Diskuterer bare data-transport i en retning
  - Samme som full duplex, men enklere å forklare
  - Kontroll-informasjon går i begge retninger

- Bruker FSM (Finite State Machines) for å spesifisere avsender og mottakers adferd

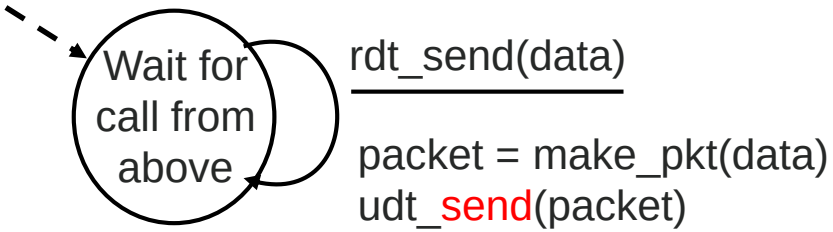
tilstand: når i denne "tilstanden" er neste tilstand entydig bestemt av neste hendelse



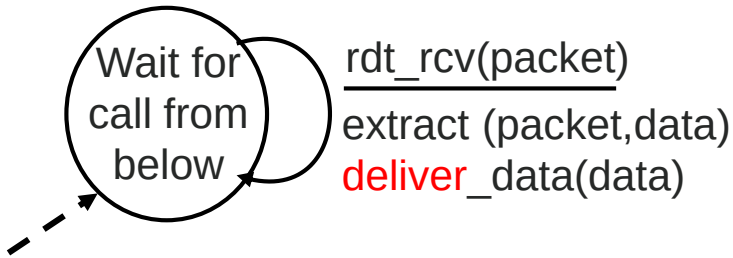
FSM: «Praktisk» Exempel

V. 1.0: Pålitelig overføring/kanal

- Kanalen helt pålitelig
  - Ingen bitfeil, ingen tap av pakker
- Separate FSM for avsender og mottaker
  - Kontroll ikke nødvendig



sender



mottager

v. 2.0: Kanal med bitfeil

A) Oppdag evt. feil:

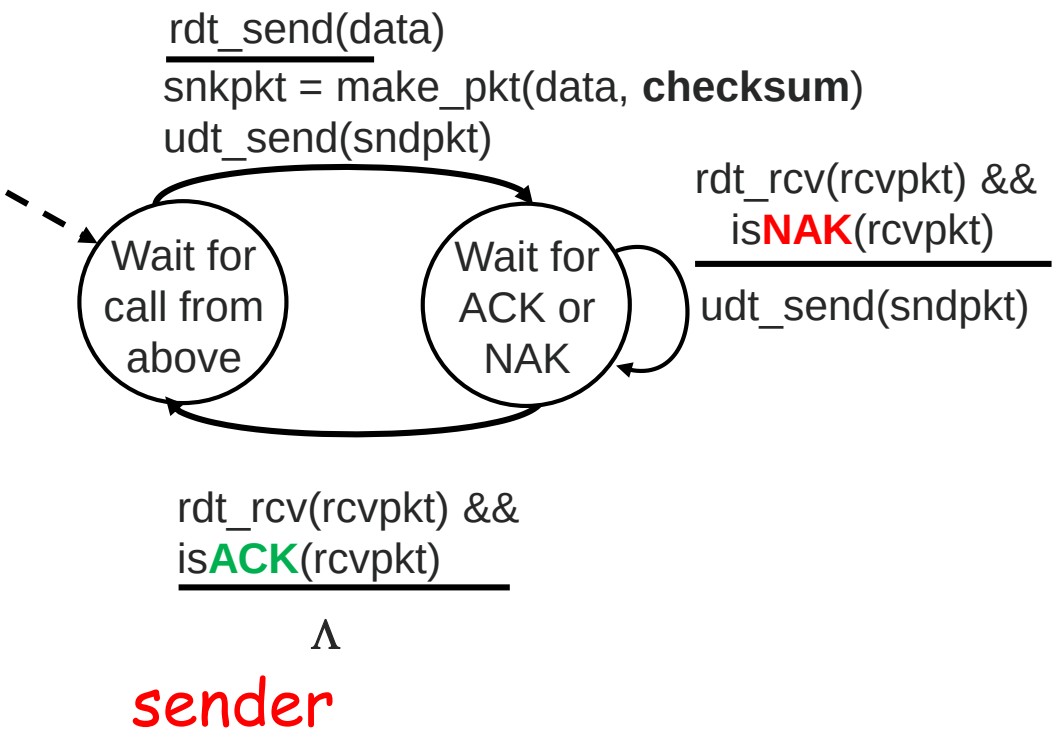
- Send sjekksum sammen med data-pakken
  - Feil kan påvises hos mottaker, men rettes ikke

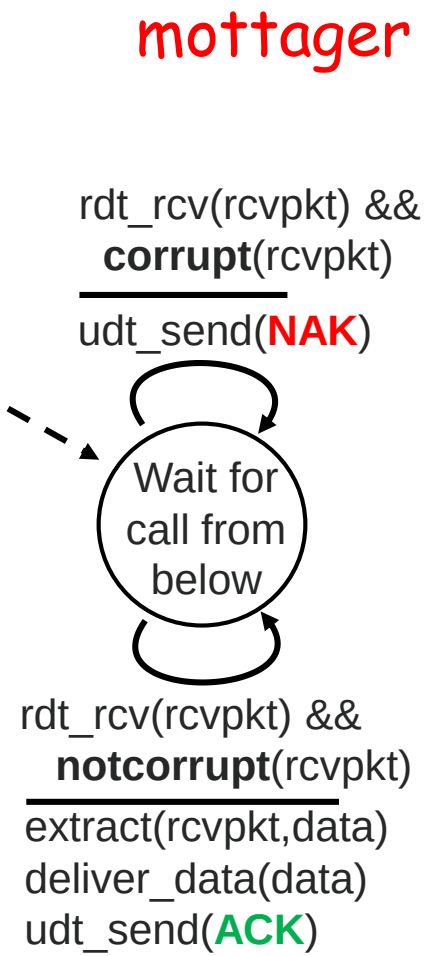
B) Gi tilbakemelding:

- Mottaker sender melding om feil til avsender
  - **ACK** (acknowledge) sendes når pakken er OK
  - **NAK** (negative ACK) sendes når pakken har feil
  - Avsender **sender** pakken **om** igjen ved **NAK**

- Tre nye mekanismer:
  1. Feil-detektering
  2. Kontroll-melding (kvittering) fra mottaker til avsender
  3. Omatt-sending ved feilmelding

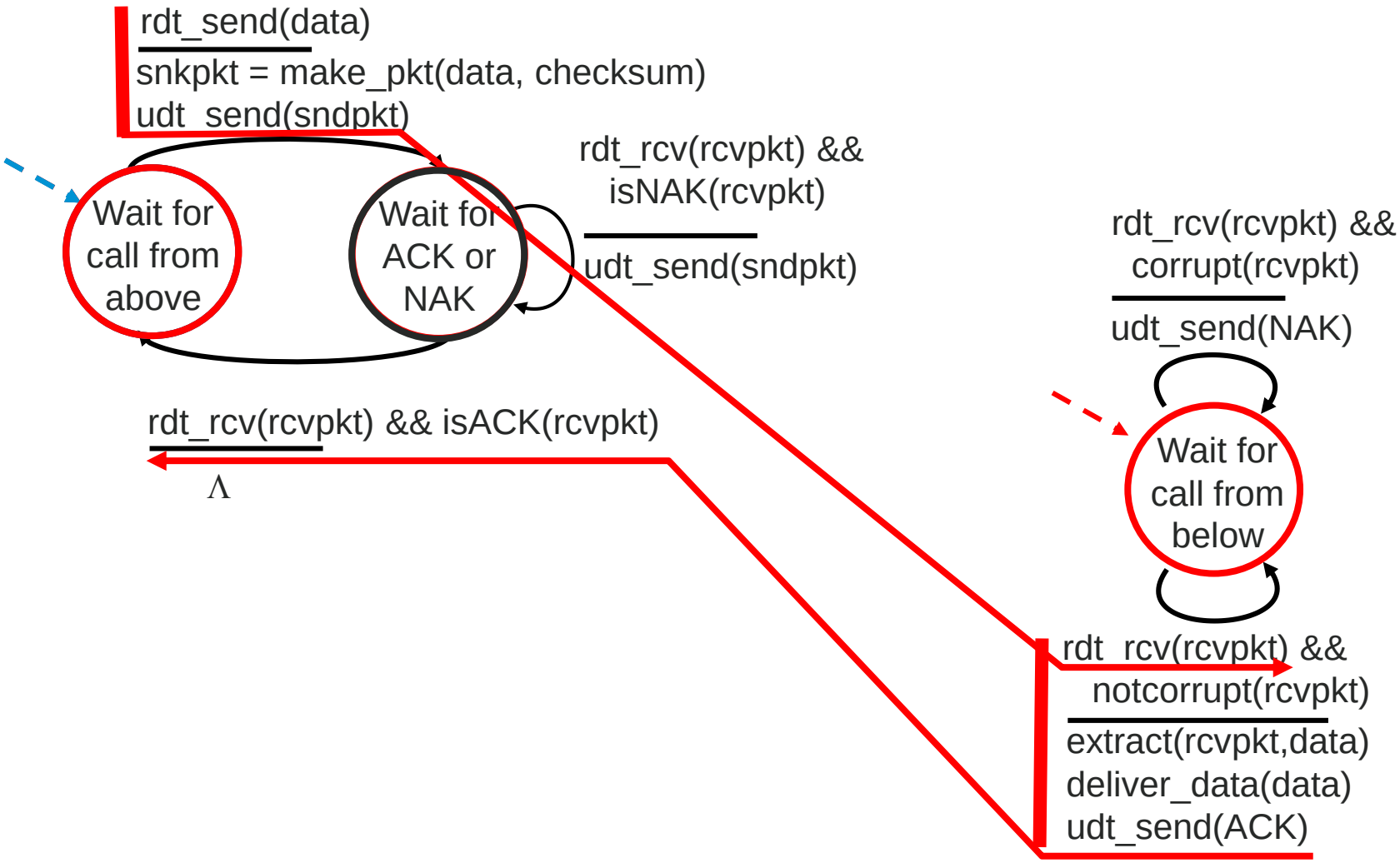
RDT 2.0: FSM modell



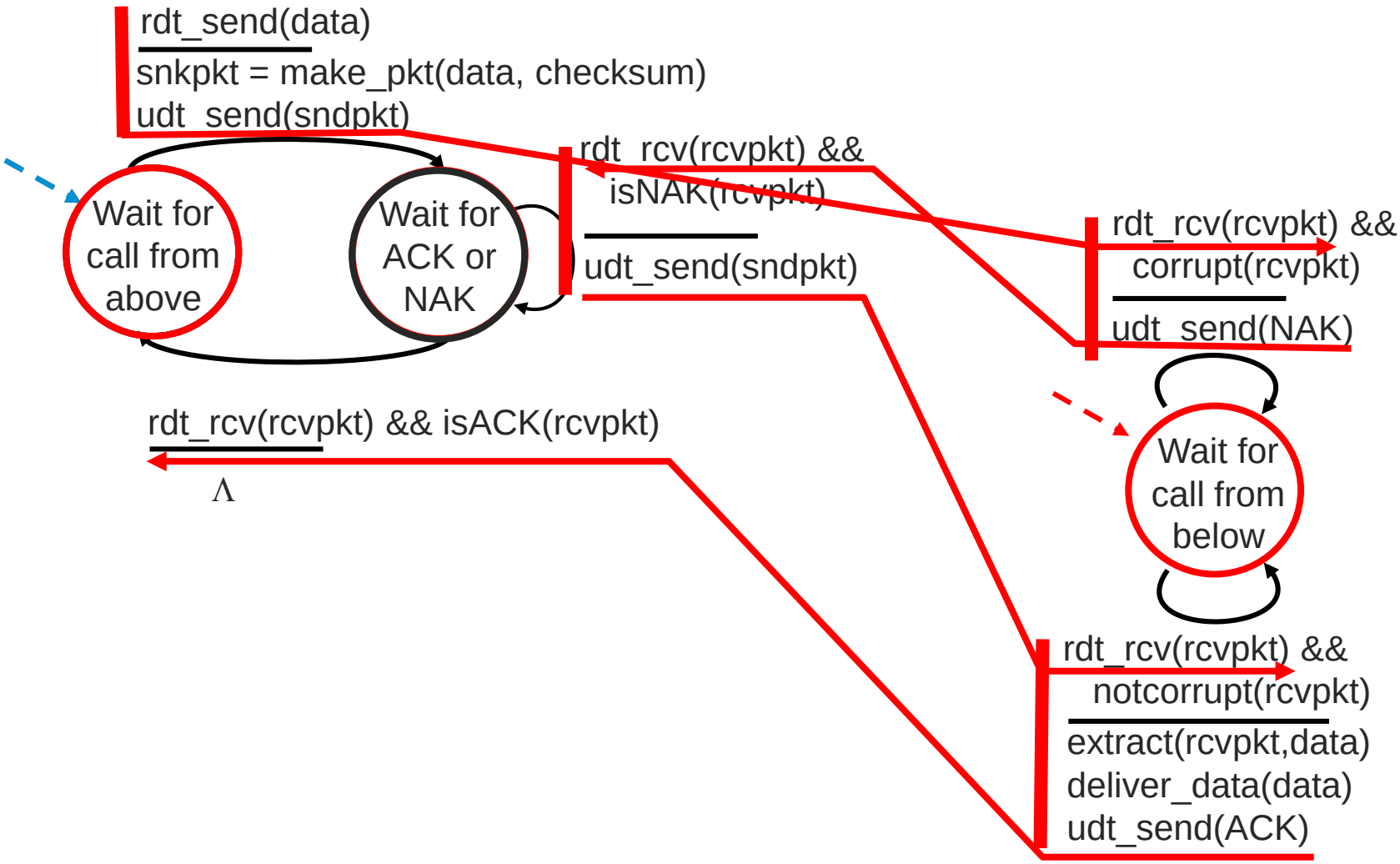


RDT 2.0:

FSM - ingen feil i overføring



v. 2.0: FSM – bitfeil i overføring..

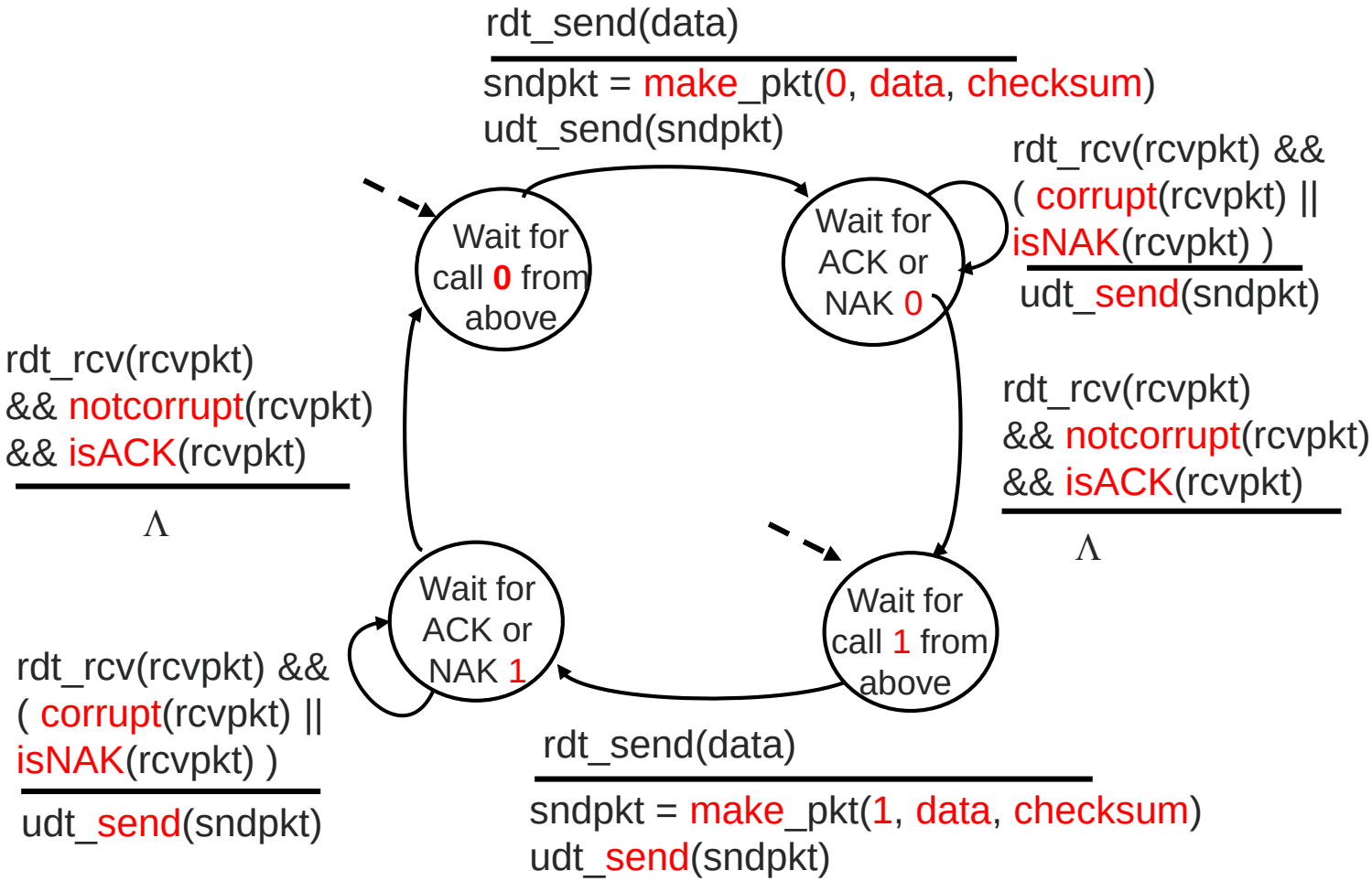


versjon 2.0: Fatal FEIL!!

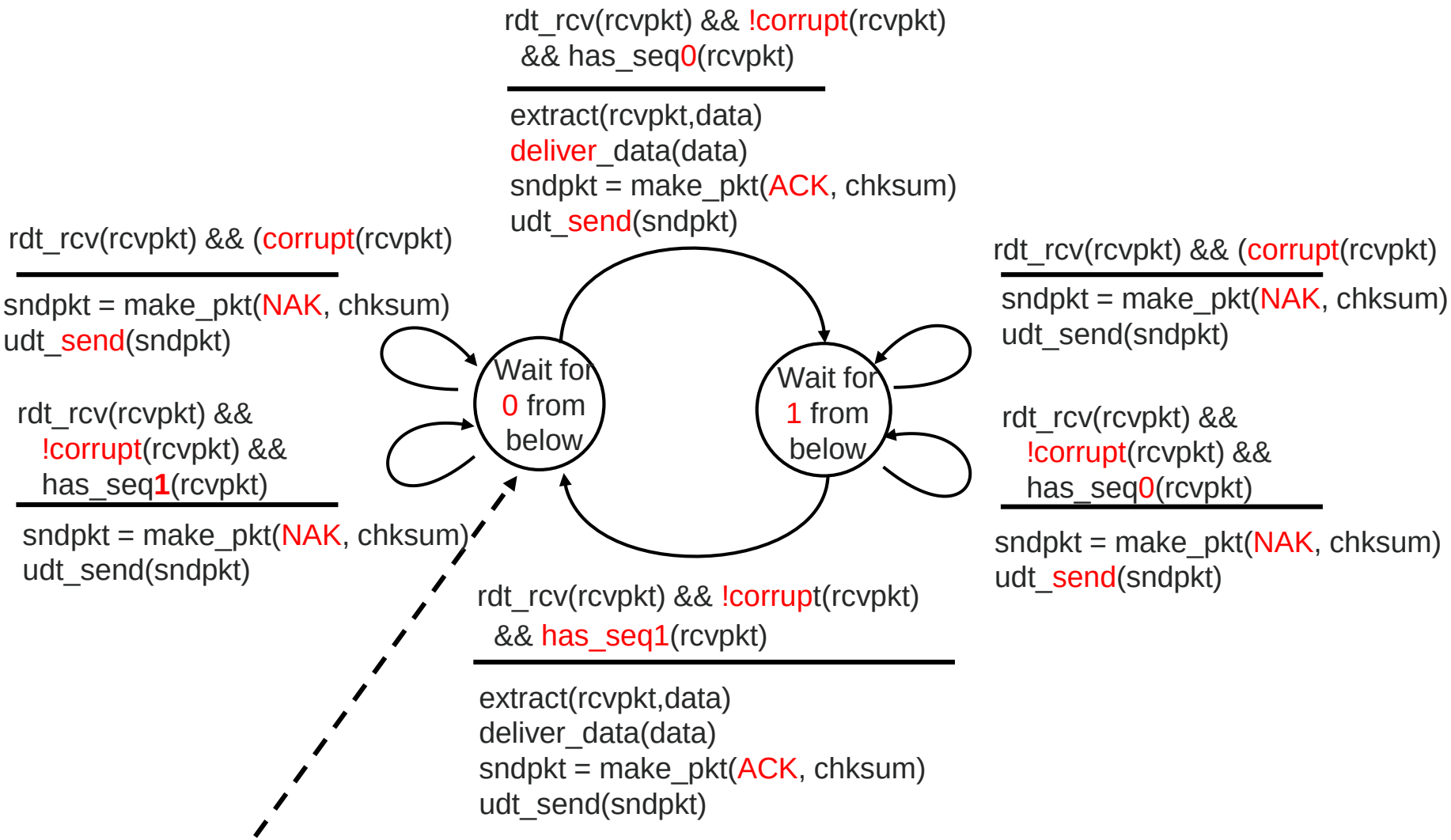
- Problemet oppstår hvis kontrollmelding svikter
  - Avsender kan da ikke vite hva som har hendt med pakken! (Den kom sikkert fram, men var det feil i den eller ikke?)
    - Ingen hensikt i å re-sende en pakke som er OK
    - Kan ikke re-sende pakken i frykt for **duplikat** (To like pakker vil for applikasjonen ende med feil data)
- Løsning
  - Avsender setter sekvensnummer på pakken
  - For en stopp/vent-protokoll trengs bare 1 bit sekvens-nummer
  - Medfører at vi må **doble antall tilstander** hos både avsender og mottager

Versjon 2.1: FSM avsender

- Kan håndtere ødelagte ACK/NAK



2.1: FSM mottaker



2.1: Analyse

Avsender:

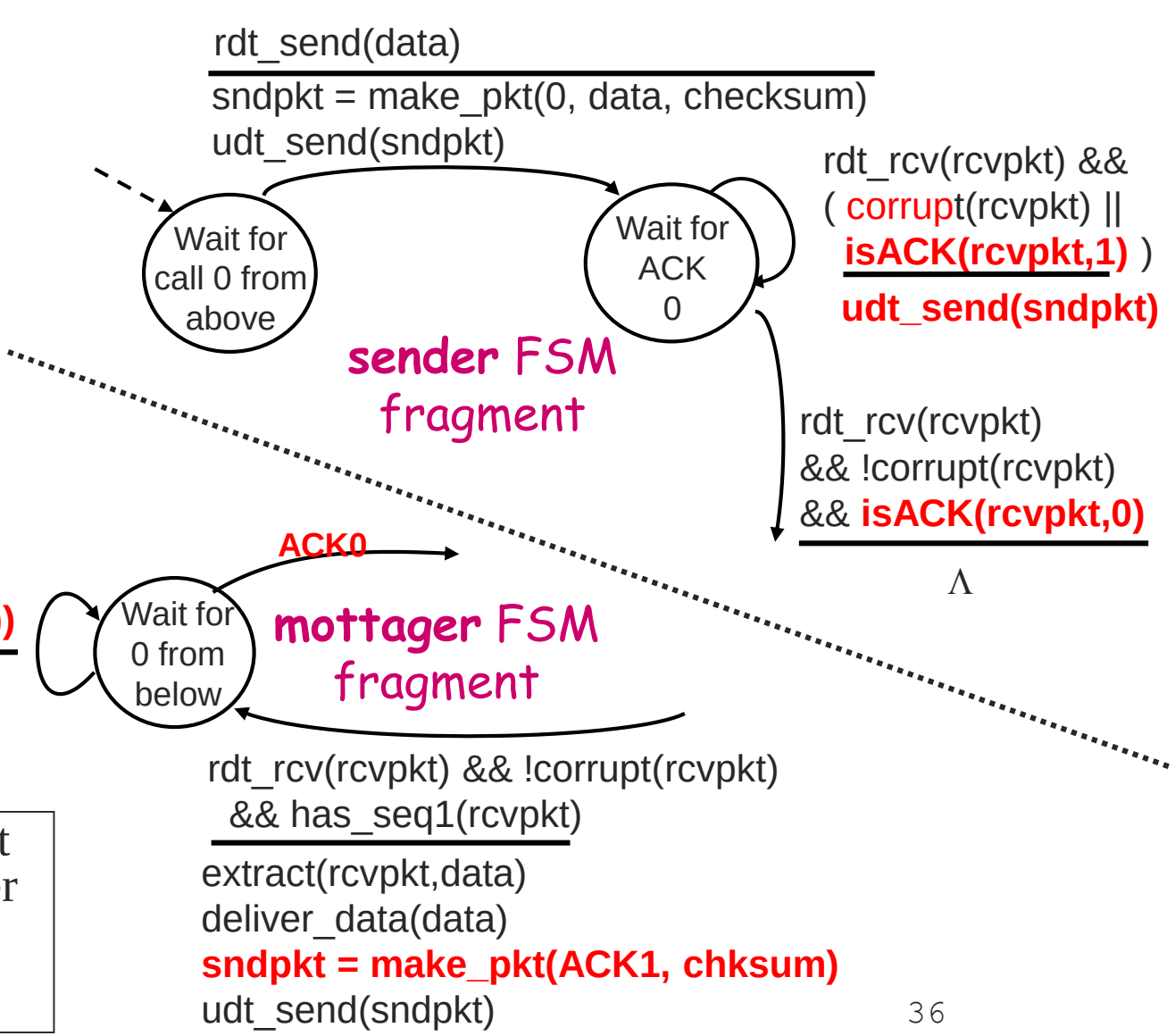
- Legger til pakkenummer (0, 1)
- Må sjekke om ACK/NAK-pakke er **korrupt**
- Må **huske nummer** på sist sendte pakke
- Dette gir kompleksitet; **2 tilstander**
  - Må ha to ulike tilstander for å ”huske” pakkenummeret til ”gjeldende” pakke.

Mottaker:

- Må sjekke om pakke er **duplikat**
  - Tilstanden angir om 0 eller 1 er forventet sekvensnummer
- NB! Mottaker kan ikke vite om sist sendte ACK/NAK er **mottatt** av avsender

versjon 2.2: NAK-fri protokoll

- Samme som 2.1, men uten egne NAK-meldinger
- Istedenfor NAK sender mottaker ACK for siste korrekt mottatte pakke
- Pakkenummer må da være med ACK'en
- Duplikat ACK til avsender medfører det samme som NAK i 2.1



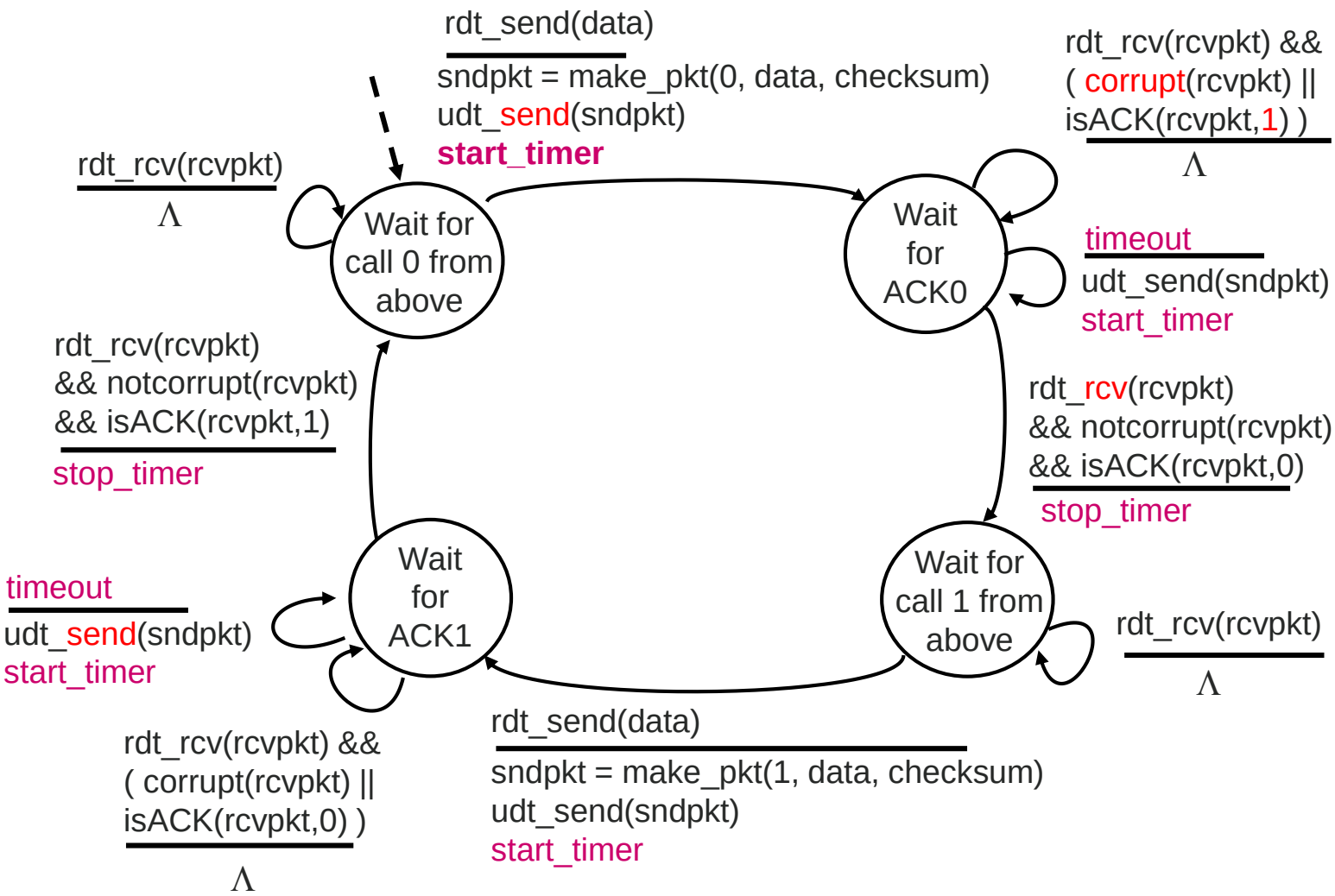
- ***Litt enklere***, men tatt med her fordi dette er en mekanisme som benyttes i TCP...

RDT 3.0: Kanal med både feil og tap

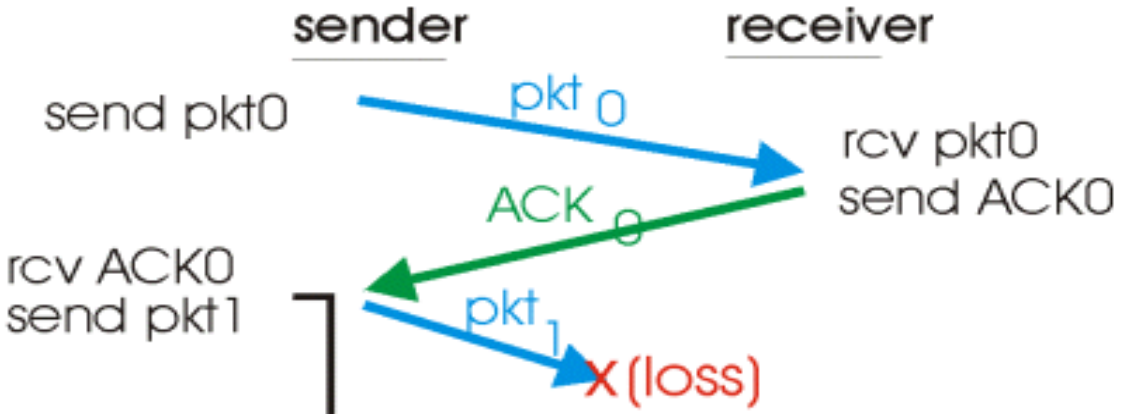
- Transportkanalen kan også *miste* pakker
  - Både data og ACK -pakker
- Løsning:
  Avsender venter et ”fornuftig” tidsrom og resender pakke hvis ingen kvittering dukker opp.

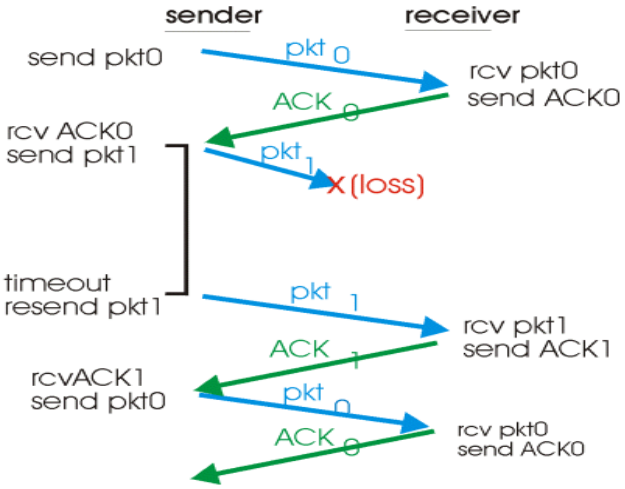
- Hvis pakke eller ACK bare er forsinket
  - Resending er da en duplikat, men sekvensnummer vil oppklare dette
  - Mottaker må spesifisere sekvensnummer på ACKen
- Denne løsningen krever en nedtellings-klokke (**timer**) hos avsender;
  som mottager kan vi fremdeles benytte versjon 2.2

RDT 3.0: FSM avsender

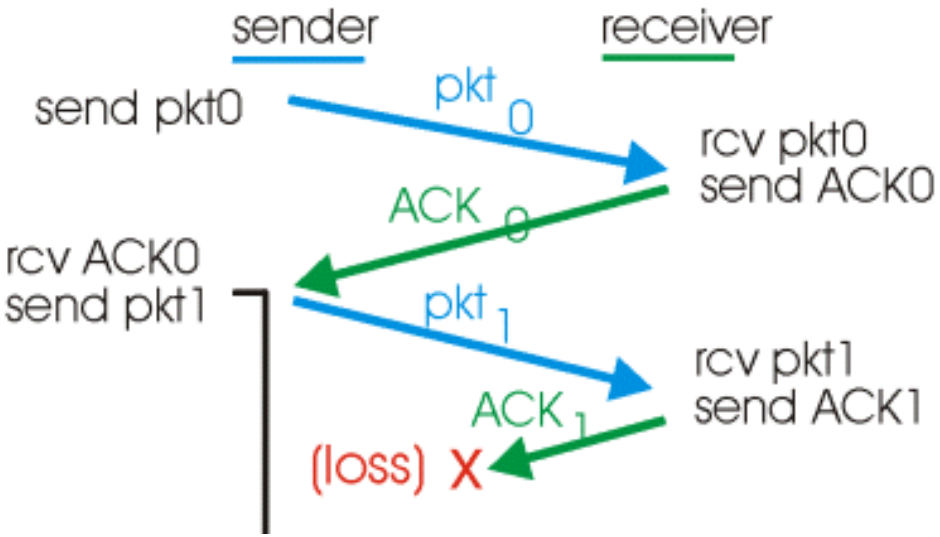


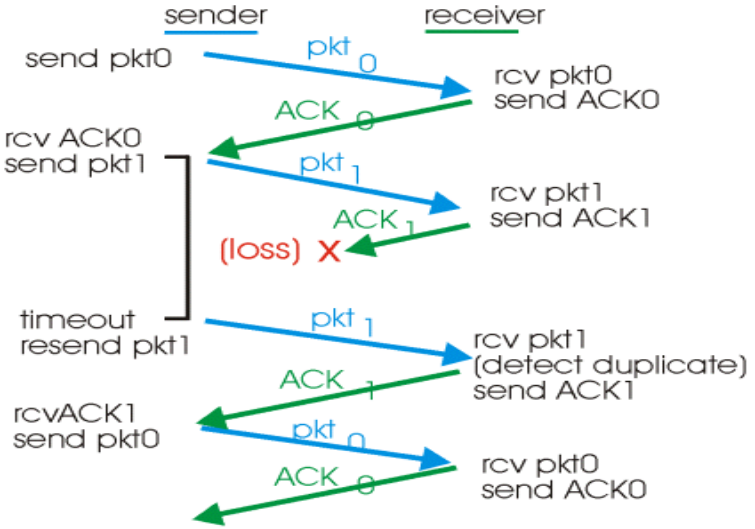
versjon 3.0: Tapt pakke





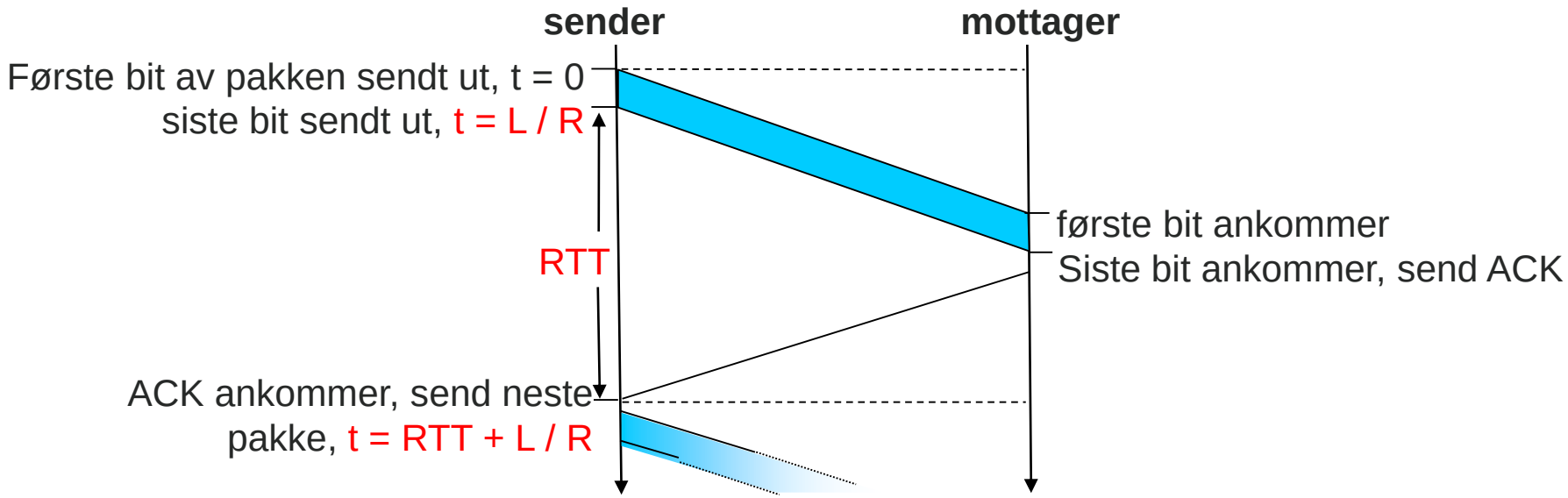
versjon 3.0: Tapt ACK





versjon 3.0: Stop&Wait Ytelse

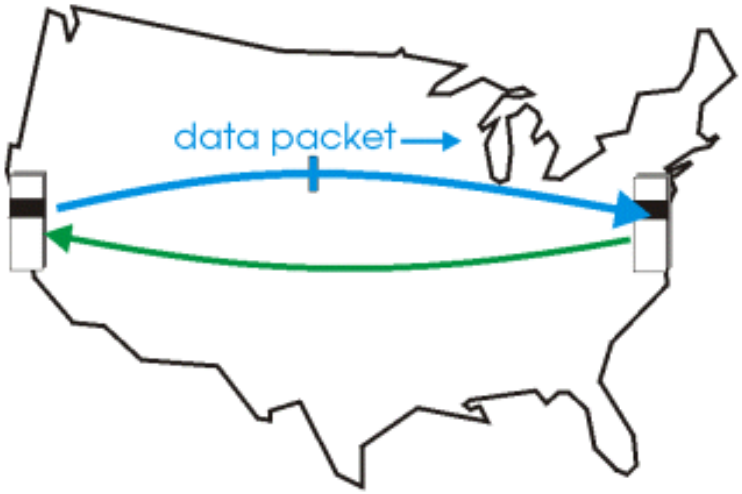
- versjon 3.0 virker for så vidt, men ytelsen er *lusen*
- Eksempel:
  - Pakke på 1 KB sendes fra Los Angeles til New York
  - Overføringstid på 4500 km er 15 millisekunder
  - Båndbredde på 1 Gb/s medfører 8 mikrosekunder for å få hele pakken ut på kanalen
  - Forutsetter like lang tid på ACKen
- Har da brukt 30,016 millisekunder på å overføre en pakke
- Kanalutnyttelsen blir: *0,14 promille(1/6667)*



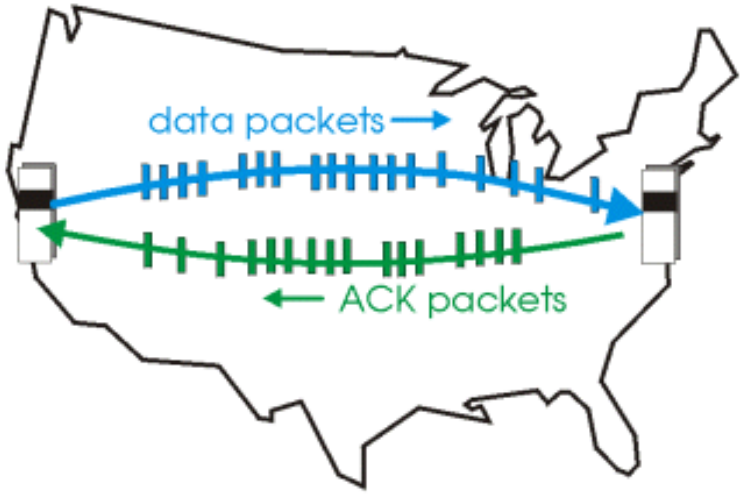
$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

Protokoller med pipelining

- Bruk av pipeline tillater mange pakker ”i luften” samtidig, mao «bedre båndbredde»
  - Sekvensnummer må da være store nok
  - Avsender/mottaker må etablere pakke-buffere



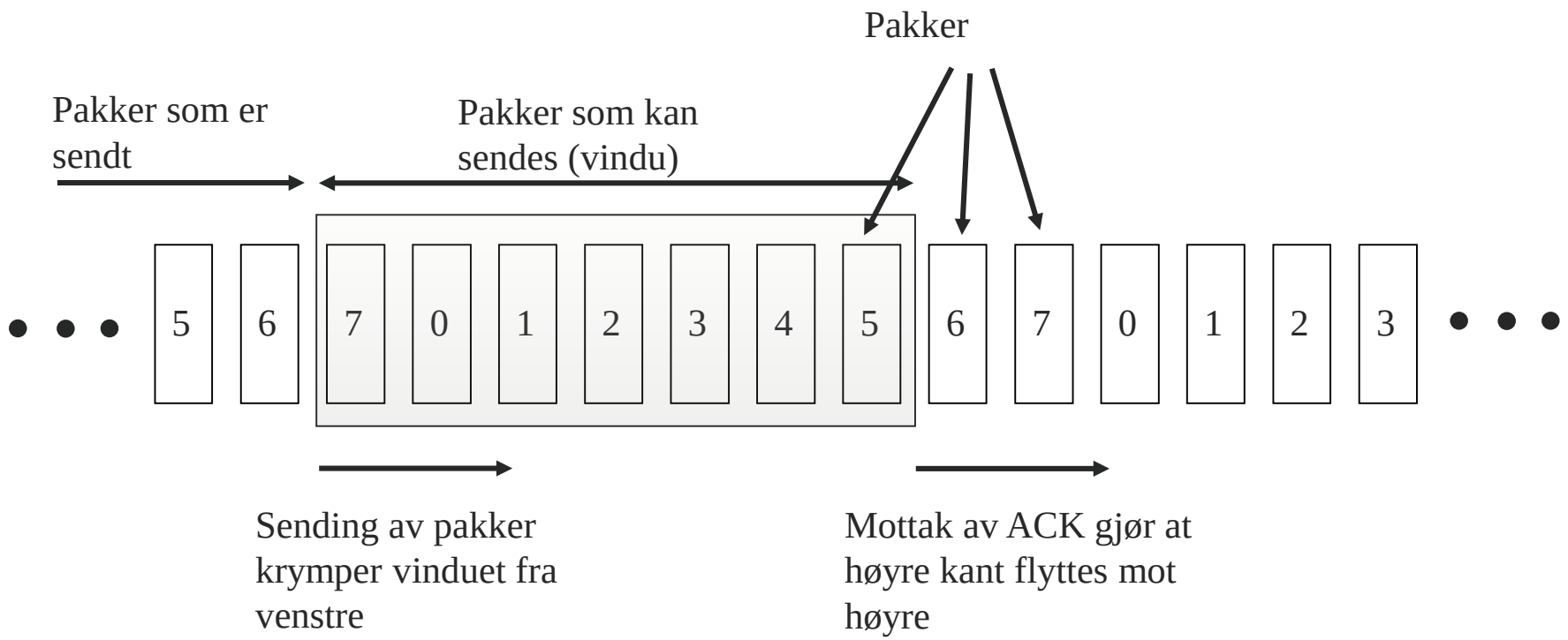
(a) a stop-and-wait protocol in operation



(b) a pipelined protocol in operation

Glidende vindu (Sliding window)

Eksempel med tre bits sekvensnummer



SR (Seletive Repeat) Prinsippet

- Mottaker gir ACK på hver eneste mottatte pakke
  - Må ha buffer for å sortere mottatte pakker
  - ACK også på omattsendte pakker
- Avsender sender bare om att pakker uten mottatt ACK
  - Må ha timer for hver pakke

Pålitelighet: Oppsummering

RDT	Situasjon	Problem	Løsning
1.0	Pålitelig kanal	Ingen	
2.0	Bitfeil på linja	Data blir tøv	•Feil-detektering (sjekksum) •Kvitteringsmeldinger (ACK,NAK) •Resending ved rapportert feil
2.1	Bitfeil	•Feil i kvitterings-meldinger •Duplikat	•(Stop-Wait) •Sekvensnummer på datapakker (0,1)
2.2	Bitfeil	”Kompleksitet”	•Fjerner NAK, sender heller ACK med sekvensnummer for siste **korrekt** mottatte pakke
3.0	+ Tap av pakker	Stop-Wait + pakker vekk i nettet	•Timer

- For å oppnå ytelse bruker vi pipeling
- Dette forutsetter at vi holder oversikt over hvilke pakker som er «i lufta» mhp kvitteringer
  - Krever buffere av ikke-kvitterte pakker som kan måtte sendes om igjen
  - TCP bruker både vindu og selektiv omattsending…

Så langt prinsipper…

Vi skjønner nå at dette er komplisert

Transmission

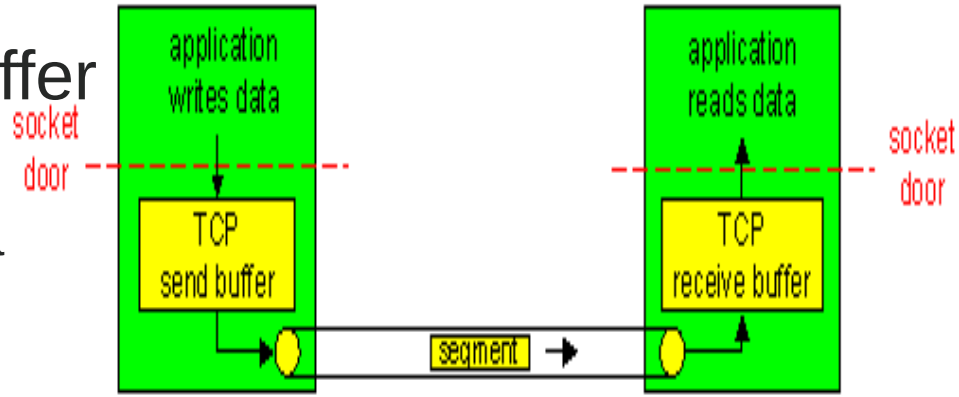
Control

Protocol

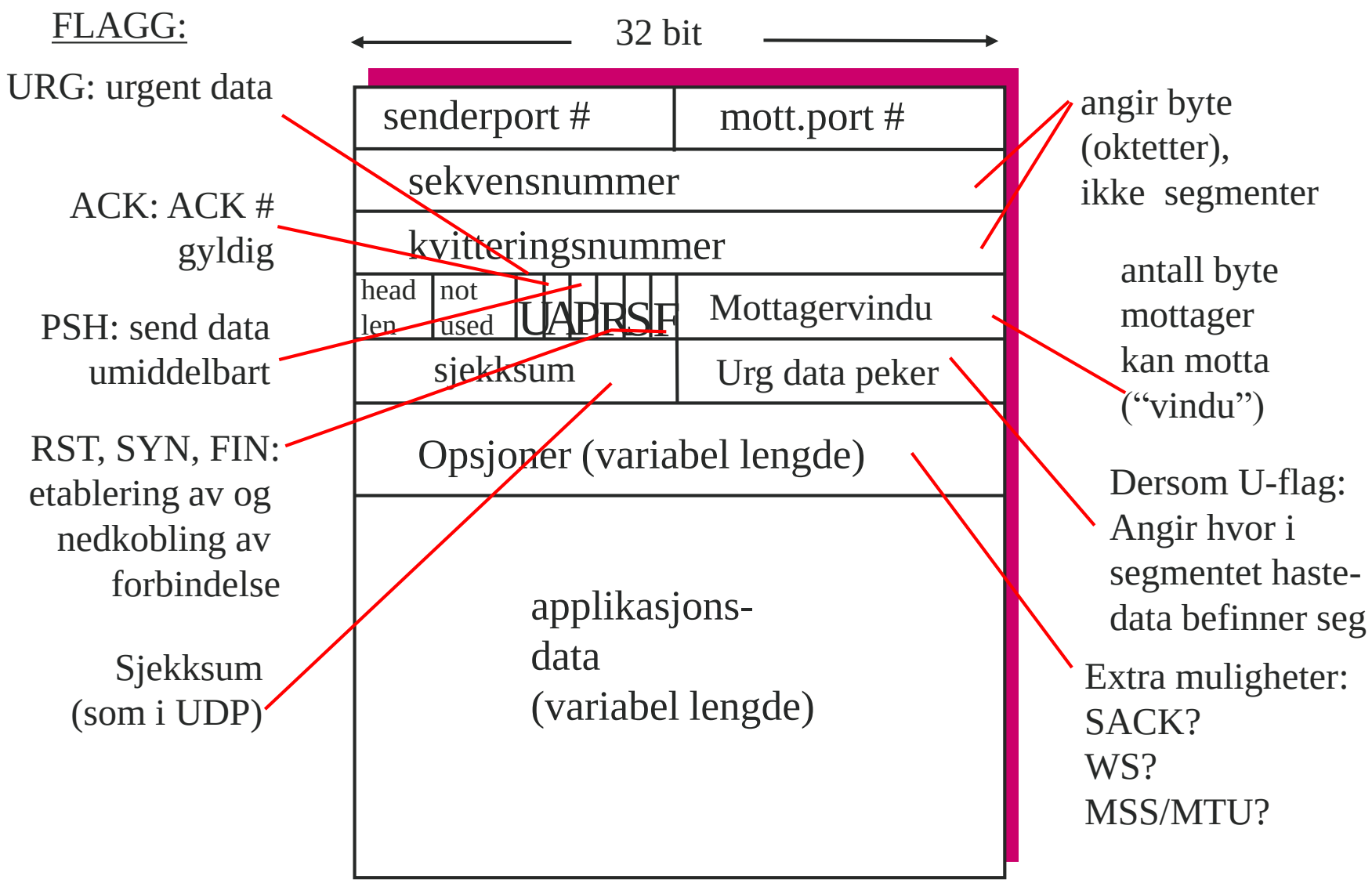
Hva med praksis?

TCP (Transmission Control Protocol)

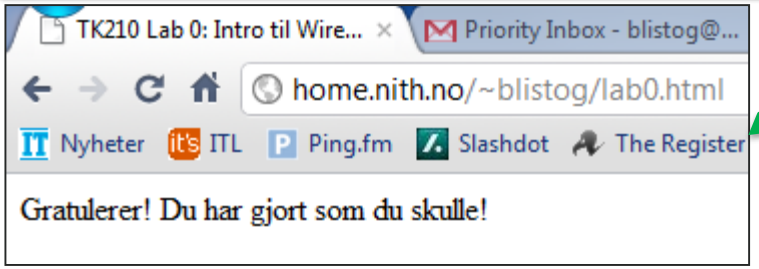
- Punkt til punkt
  - En avsender, en mottaker
- Pålitelig, ordnet byte-strøm
- Pipeline
  - Flyt- og metnings-kontroll bestemmer vindu-størrelse
- Avsender og mottaker-buffer
- Full duplex data
  - Begge kan sende og motta samtidig
- Forbindelses-orientert
  - Handshake før dataoverføring
- Flytkontroll
  - Avsender drukner ikke mottaker



Oppbygging av TCP-header



Wireshark: Enkel overføring



1) Experiment

2) Resultat

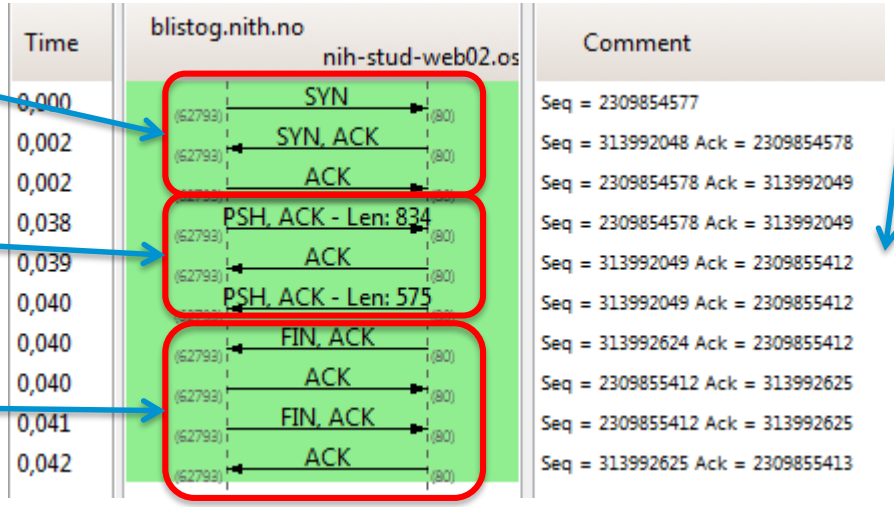
	Time	Source	Destination	Protocol	Info
5	3.012370	blistog.nith.no	nih-stud-web02.osl.	TCP	62793 > http [SYN] Seq=2309854577 Win=8192 Len=0 MSS=1460 WS=2 SACK_PERM=1
6	3.014275	nih-stud-web02.osl.	blistog.nith.no	TCP	http > 62793 [SYN, ACK] Seq=313992048 Ack=2309854578 Win=5840 Len=0 MSS=1380
7	3.014329	blistog.nith.no	nih-stud-web02.osl.	TCP	62793 > http [ACK] Seq=2309854578 Ack=313992049 Win=16560 Len=0
8	3.050158	blistog.nith.no	nih-stud-web02.osl.	HTTP	GET /~blistog/lab0.html HTTP/1.1
9	3.051843	nih-stud-web02.osl.	blistog.nith.no	TCP	http > 62793 [ACK] Seq=313992049 Ack=2309855412 Win=59 Len=0
10	3.052514	nih-stud-web02.osl.	blistog.nith.no	HTTP	HTTP/1.1 200 OK (text/html)
11	3.052515	nih-stud-web02.osl.	blistog.nith.no	TCP	http > 62793 [FIN, ACK] Seq=313992624 Ack=2309855412 Win=59 Len=0
12	3.052571	blistog.nith.no	nih-stud-web02.osl.	TCP	62793 > http [ACK] Seq=2309855412 Ack=313992625 Win=16416 Len=0
13	3.052912	blistog.nith.no	nih-stud-web02.osl.	TCP	62793 > http [FIN, ACK] Seq=2309855412 Ack=313992625 Win=16416 Len=0
14	3.054161	nih-stud-web02.osl.	blistog.nith.no	TCP	http > 62793 [ACK] Seq=313992625 Ack=2309855413 Win=59 Len=0

Oppkopling

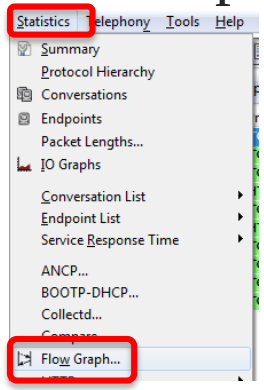
Dataoverføring

GET + ACK + 200OK

Nedkopling



3) Flow Graph



TCP: Oppstart av forbindelsen

- Sender og mottaker etablerer en forbindelse før data-segmenter utveksles
  - Initialiserer TCP-variable
    - Sekvens-nummer, buffere, vinduer.....
- Klient -> avsender -> mottaker -> server
  - Setter opp socket
- Klient sender et spesielt TCP-segment med SYN
  - SYN-flagget i headeren satt
  - Spesifiserer start sekvens-nummer
- Server svarer med SYN ACK
  - SYN og ACK-flaggene i headeren satt
  - Setter opp start sekvens-nummer, buffere, vinduer mm

Wireshark: SYN og SYN+ACK

Klient

```
Source port: 62793 (62793)
Destination port: http (80)
[Stream index: 0]
Sequence number: 2309854577
Header length: 32 bytes
Flags: 0x02 (SYN)
  000. .... .... = Reserved: Not set
  ...0 .... .... = Nonce: Not set
  .... 0... .... = Congestion Window Reduced (CWR)
  .... .0.. .... = ECN-Echo: Not set
  .... ..0. .... = Urgent: Not set
  .... ...0 .... = Acknowledgement: Not set
  .... .... 0... = Push: Not set
  .... .... .0.. = Reset: Not set
⊞ .... .... ..1. = Syn: Set
  .... .... ...0 = Fin: Not set
Window size: 8192
Checksum: 0x76be [correct]
Options: (12 bytes)
  Maximum segment size: 1460 bytes
  NOP
  Window scale: 2 (multiply by 4)
  NOP
  NOP
  TCP SACK Permitted Option: True
```

SYN

Tjener

```
Source port: http (80)
Destination port: 62793 (62793)
[Stream index: 0]
Sequence number: 313992048
Acknowledgement number: 2309854578
Header length: 28 bytes
Flags: 0x12 (SYN, ACK)
  000. .... .... = Reserved: Not set
  ...0 .... .... = Nonce: Not set
  .... 0... .... = Congestion Window Reduced (CWR)
  .... .0.. .... = ECN-Echo: Not set
  .... ..0. .... = Urgent: Not set
  .... ...1 .... = Acknowledgement: Set
  .... .... 0... = Push: Not set
  .... .... .0.. = Reset: Not set
⊞ .... .... ..1. = Syn: Set
  .... .... ...0 = Fin: Not set
Window size: 5840
Checksum: 0x5f08 [correct]
Options: (8 bytes)
  Maximum segment size: 1380 bytes
  NOP
  Window scale: 7 (multiply by 128)
```

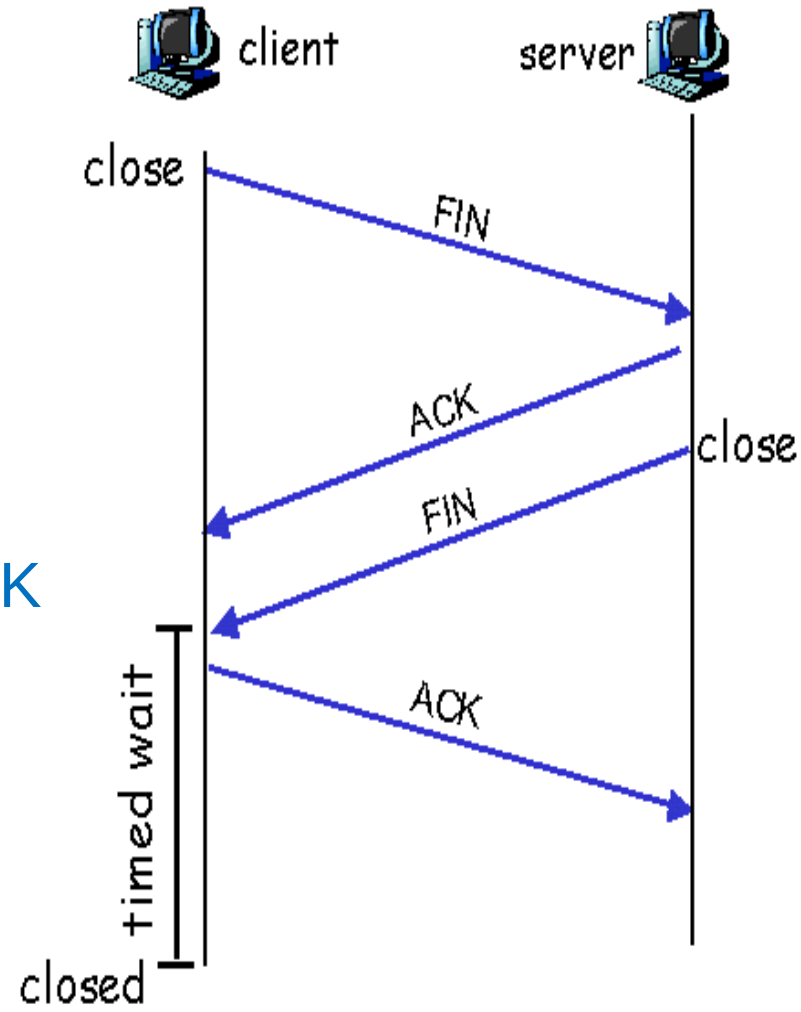
SYN + ACK

- Utveksler sekvensnummer
- Avtaler MSS
- Avtaler/utveksler Window scaling

Nedkobling av forbindelsen

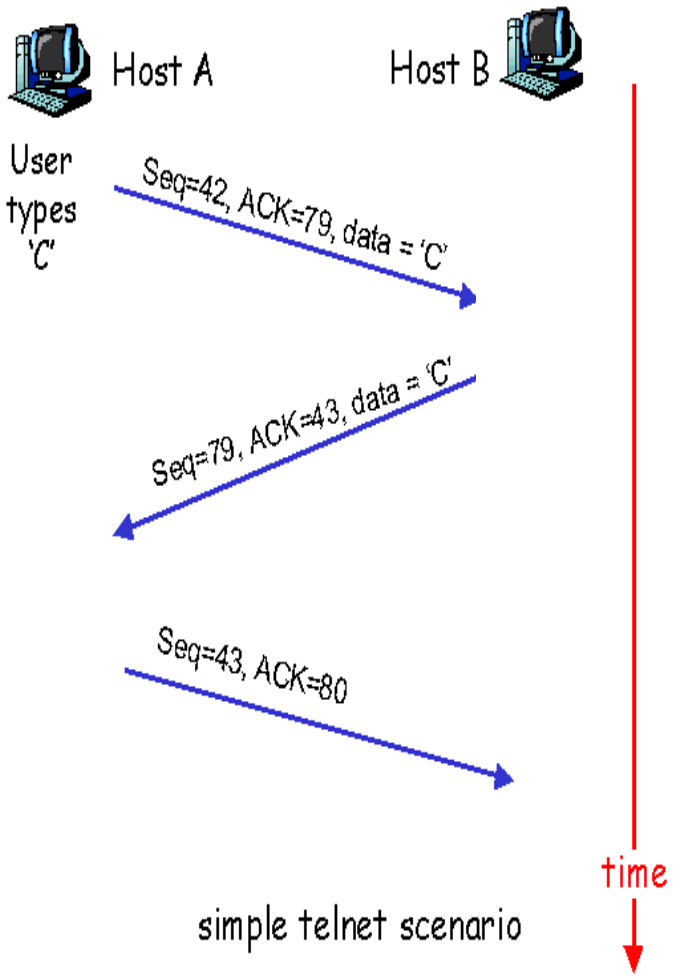
- Klient-app lukker socket
- Klient-OS sender TCP FIN til server
- Server-OS mottar FIN, sender ACK
- Server-app lukker socket
- Server-OS sender FIN til klient
- Klient-OS mottar FIN, sender ACK
- Server-OS mottar ACK
- Forbindelsen avsluttet

- NB! Andre metoder benyttes også!!
  - F. eks RESET-flagget (fra Server)
  - Three Way: FIN, FIN+ACK, ACK



Sekvensnummer og ACK

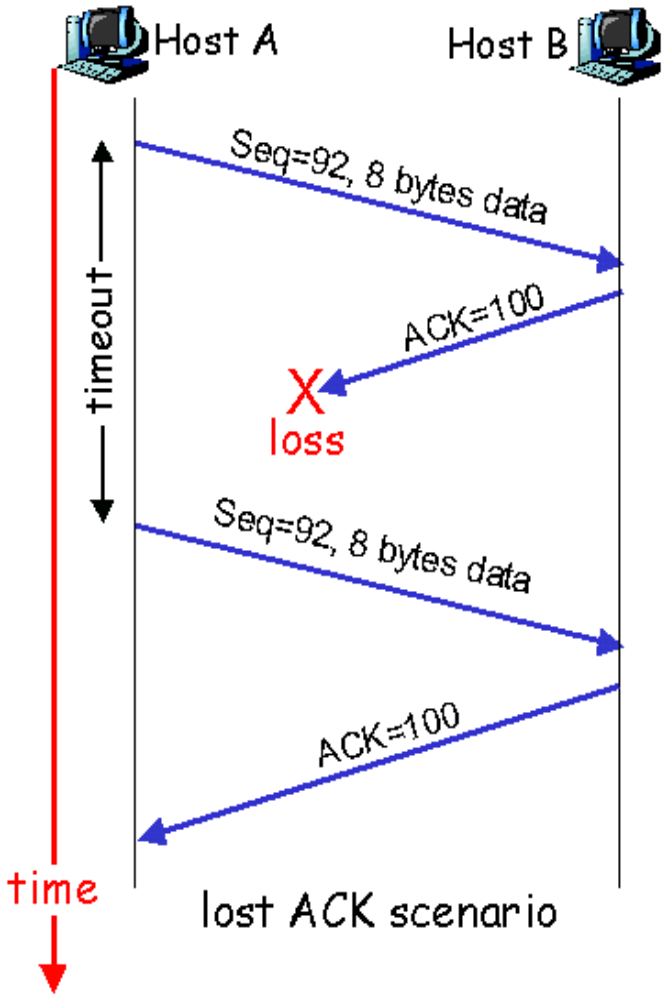
- TCP er byte-orientert
- Sekvensnummer
  - Bytestrøm-nummer for første byte i segmentet
- ACK-nummer
  - Sekvensnummer til neste byte som forventes fra den andre siden
  - Kumulativ ACK
- Segmenter utenfor rekkefølge
  - Ikke dekket av TCP-spesifikasjonen, men må håndteres i implementasjonen

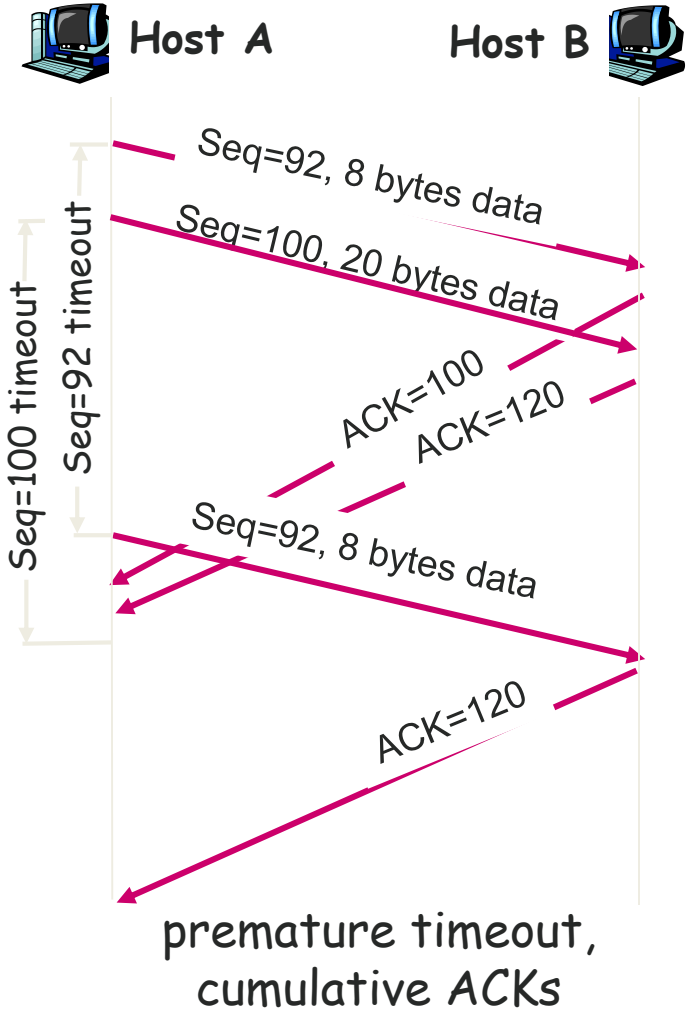


TCP ACK generering [RFC 1122, RFC 2581]

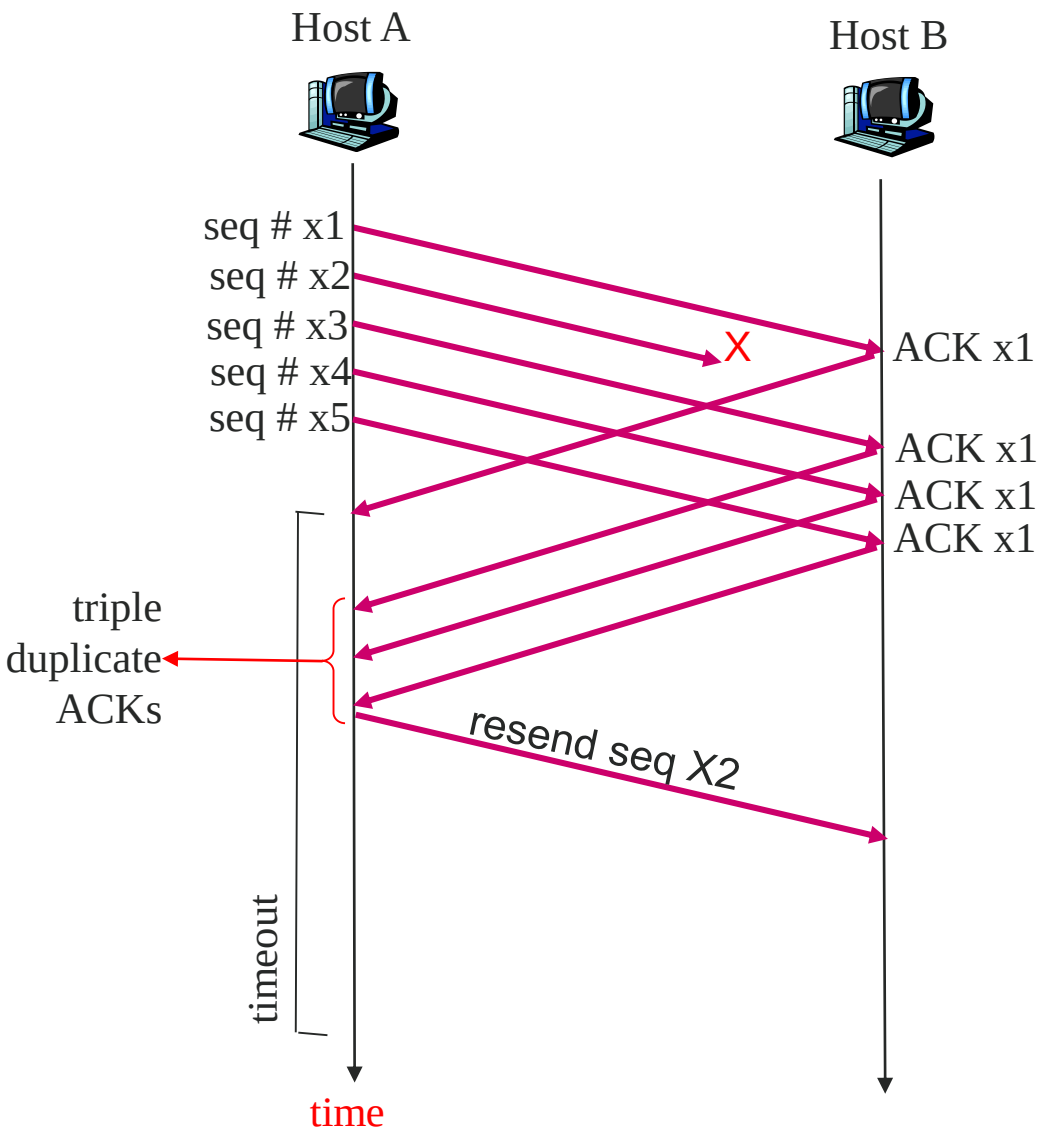
Hendelse hos Mottager	TCP Mottager Håndtering
Ankomst av segment i riktig rekkefølge med forventet seq #. Alle data allerde ACKet	Utsatt (delayed) ACK. Vent inntil 0,5 s på neste segment. Dersom det ikke kommer, send ACK.
Ankomst av segment i riktig rekkefølge med forventet seq #. Et tildligere segment er ankommet, men ikke ACKet	Send umiddelbart kummulativ ACK (fungerer som ACK for begge)
Ankomst av segment ute av rekkefølge med høyere seq # enn forventet (Gap oppdaget)	Send umiddelbart *duplikat ACK*, som indiker seq # for neste **forventede** byte
Ankomst av segment som del-vis eller helt fyller gap	Send umiddelbar ACK (forutsatt at segmentet starter på “bunnen av gapet”)

Resending


Host A
Host B
Seq=92, 8 bytes data
ACK=100
timeout
X
loss
Seq=92, 8 bytes data
ACK=100
time
lost ACK scenario


Host A
Host B
Seq=92, 8 bytes data
Seq=100, 20 bytes data
Seq=100 timeout
Seq=92 timeout
ACK=100
ACK=120
Seq=92, 8 bytes data
ACK=120
premature timeout,
cumulative ACKs

Rask omsending



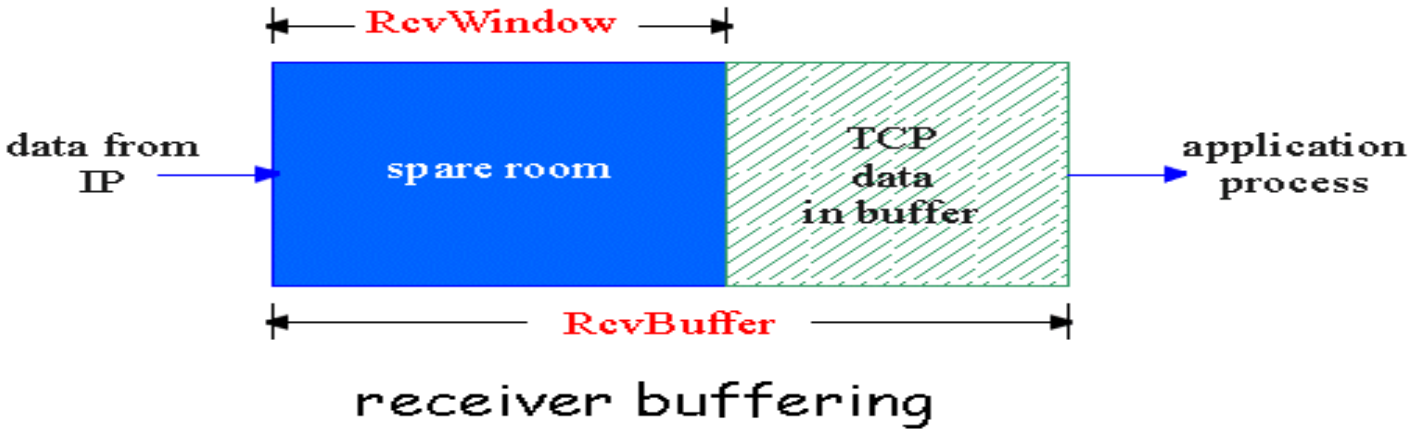
- Timeout-perioden er ofte relativt lang
- Dersom sender mottar 3 ACK på samme data før timeout tolkes det som pakketap –> omsending av påfølgende segment

Flytkontroll = Window

- **Avsender skal ikke «drukne» mottaker ved å sende for mye, for fort**
- Mottaker informerer avsender om fri buffer-kapasitet
  - *RcvWindow* i TCP segmentet
- Avsender tar hensyn til dette

RcvBuffer = size of TCP Receive Buffer

RcvWindow = amount of spare room in Buffer



receiver buffering

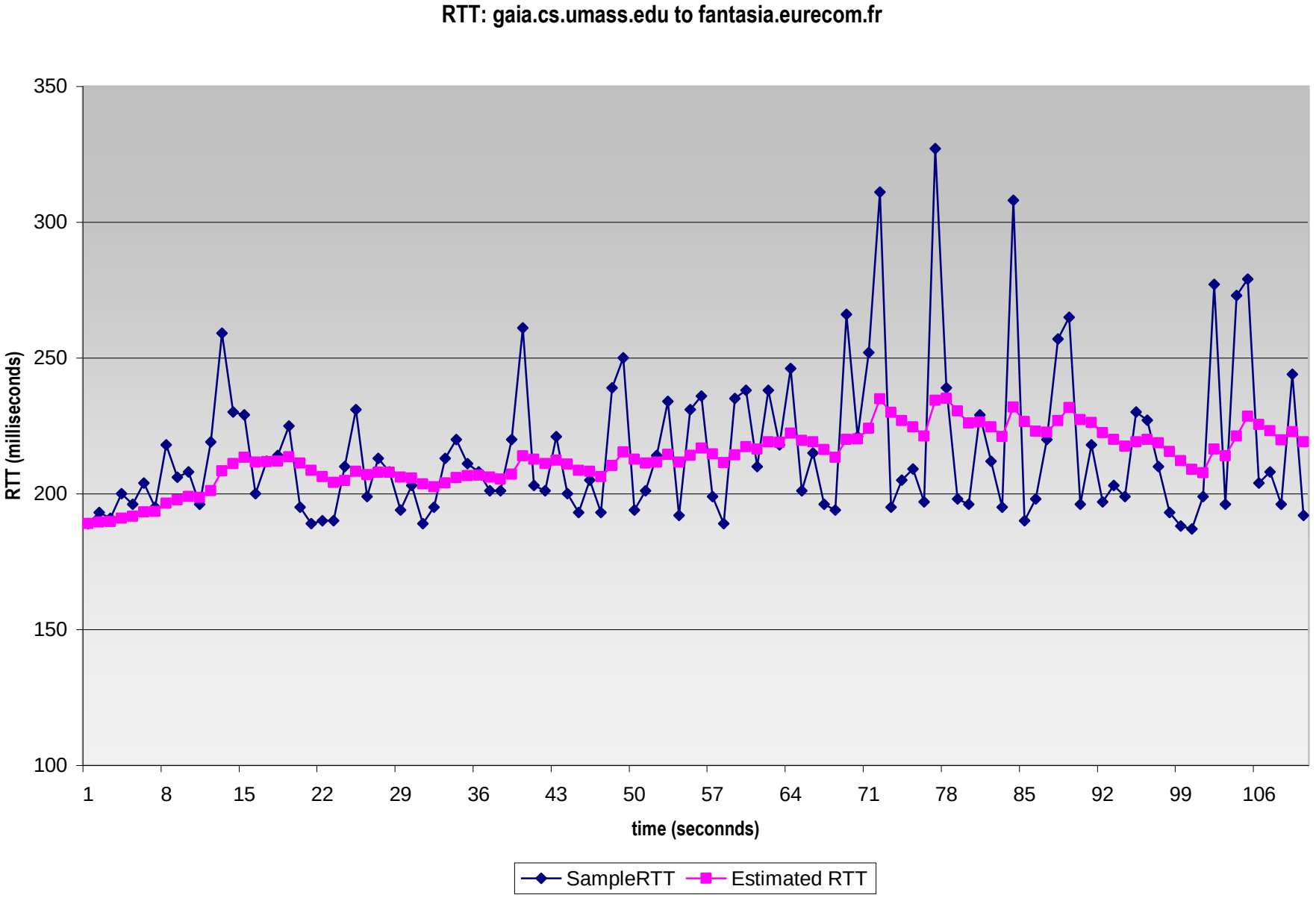
RTT (Round Trip Time) og timeout

- Hvor stor skal en velge timeout-verdien?
  - Lengre enn RTT
    - RTT varierer!
  - For kort timeout-verdi gir unødvendig omsending
  - For lang timeout-verdi gir for dårlig reaksjon på tap av segment

- Måler tiden fra avsendt segment til mottatt ACK
  - Tar ikke med omsend og kumulativ ACK i målingene
  - Må ta hensyn til at RTT varierer
  - Mest interessert i de ferskeste målingene

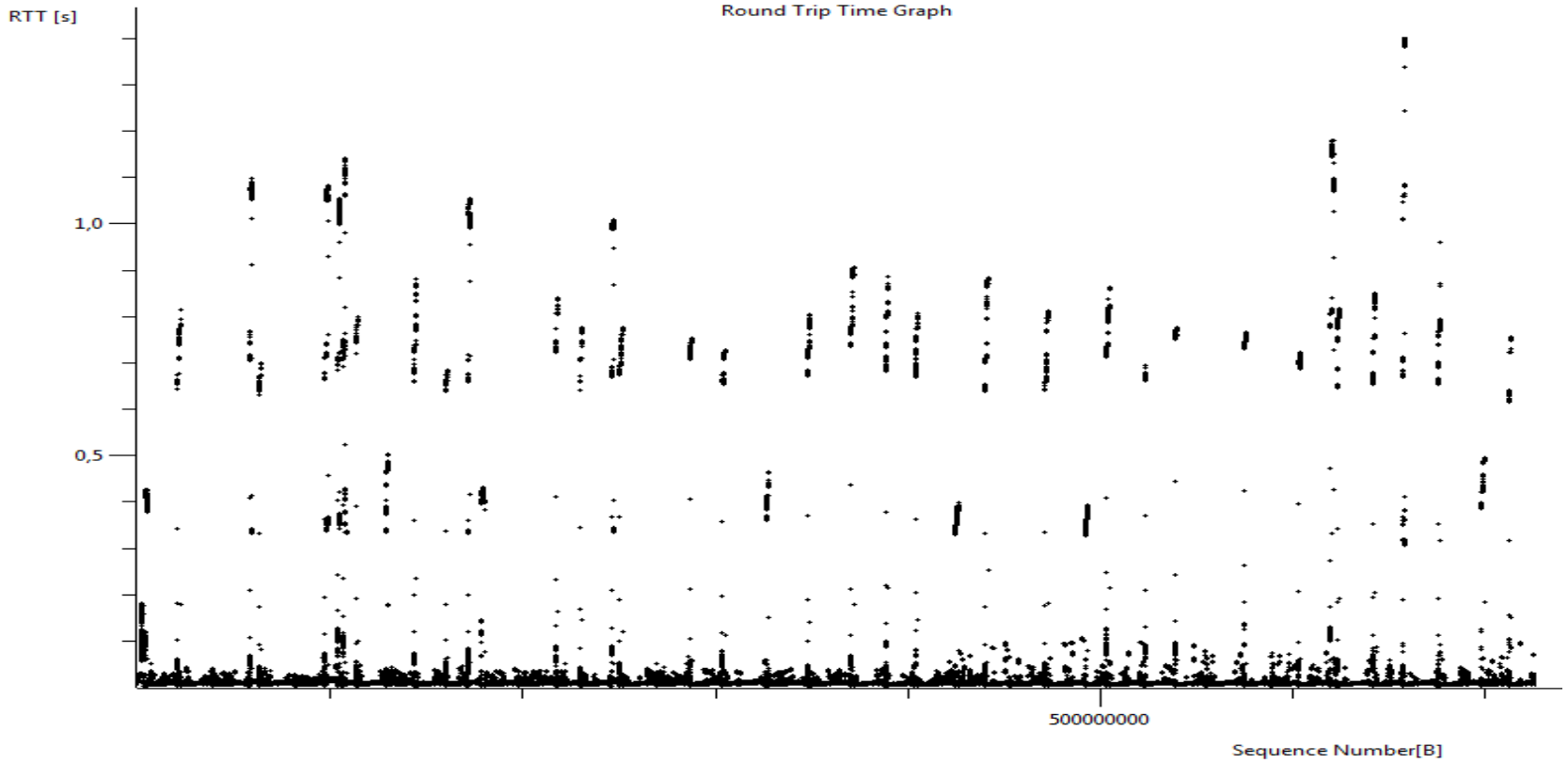
Westerdals
Oslo A.C.T
Avdeling for Teknologi

Eksempel RTT estimering:



Wireshark: RTT

- Laster opp en 1 GB til home.nith.no
- RTT *varierer* mellom langt under 1/10 ms og 1,4 s (ca faktor 10.000)



Beregning av timeout

- Utregning av utjevnet RTT

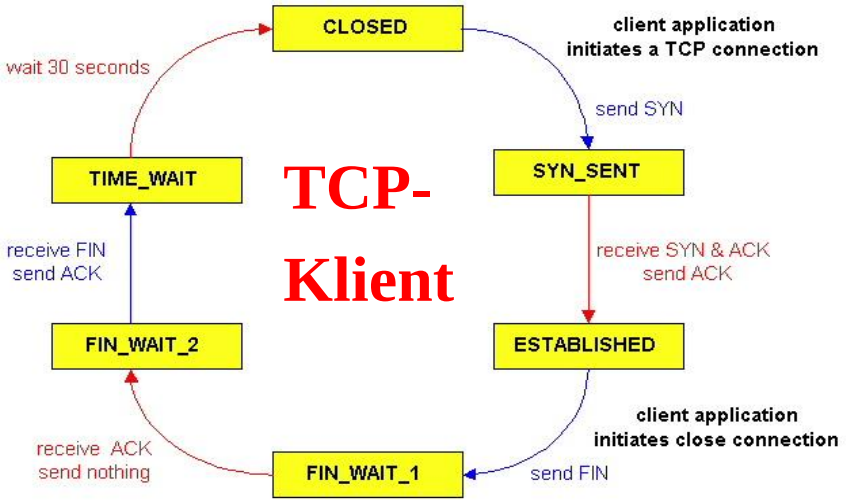
```
EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT
```

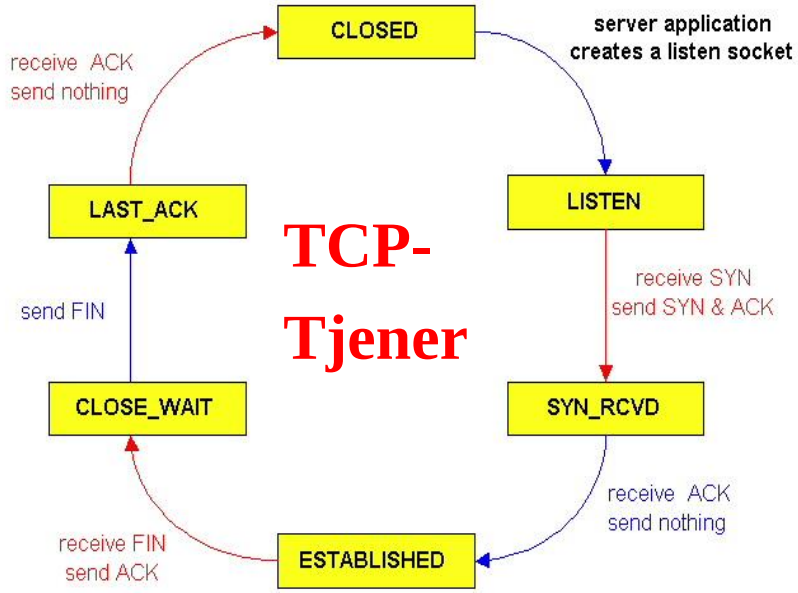
  - Eksponensielt veiet, løpende gjennomsnitt
  - Siste målinger veier tyngst
  - Typiske verdier for x: 1/8 - 1/10
- Utregning av timeout
  - *EstimatedRTT* pluss en sikkerhets-margin
  - Jo større variasjon i *EstimatedRTT*, jo større sikkerhets-margin

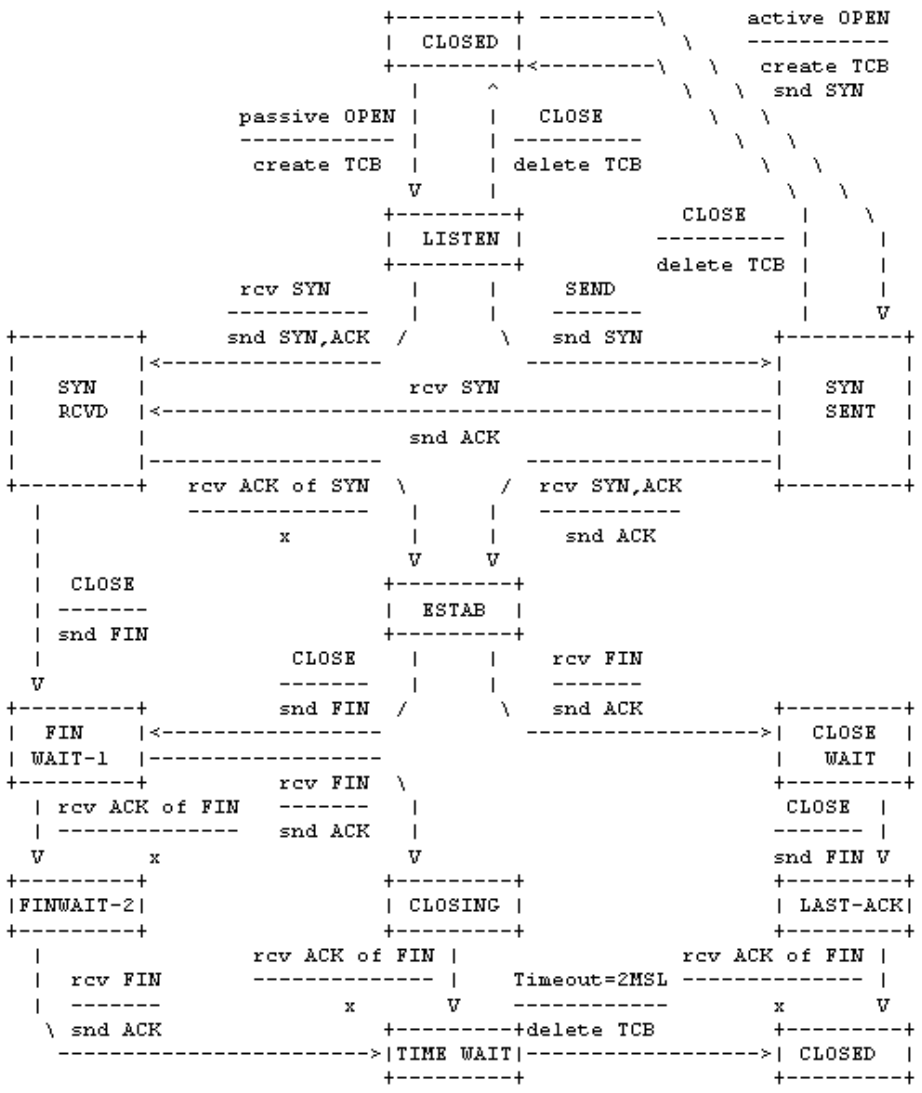
```
Timeout = EstimatedRTT + 4*Deviation
Deviation = (1-x)*Deviation + x*|SampleRTT-EstimatedRTT|
```

Westerdals
Oslo A.C.T
Avdeling for Teknologi

TCP: Livssyklus







TCP Connection State Diagram
Figure 6.

Om hvordan andre faktorer kan ha like mye å si som «båndbredden» du får av ISP…

TCP METNINGSKONTROL «CONGESTION CONTROL»

Trafikk-kork/**Metning** (”congestion”)

- For mange kilder sender for mye data for fort til at nettverket (routerene) klarer å håndtere det
  - Dette er *forskjellig fra flyt-kontroll* som avhenger av endesystemenes kapasitet og styres med utveksling av vindusstørrelser (RWIN)
- Resulterer i
  - tapte pakker (drukner i ruter-buffer)
  - lange forsinkelser (kø i ruter-buffer)

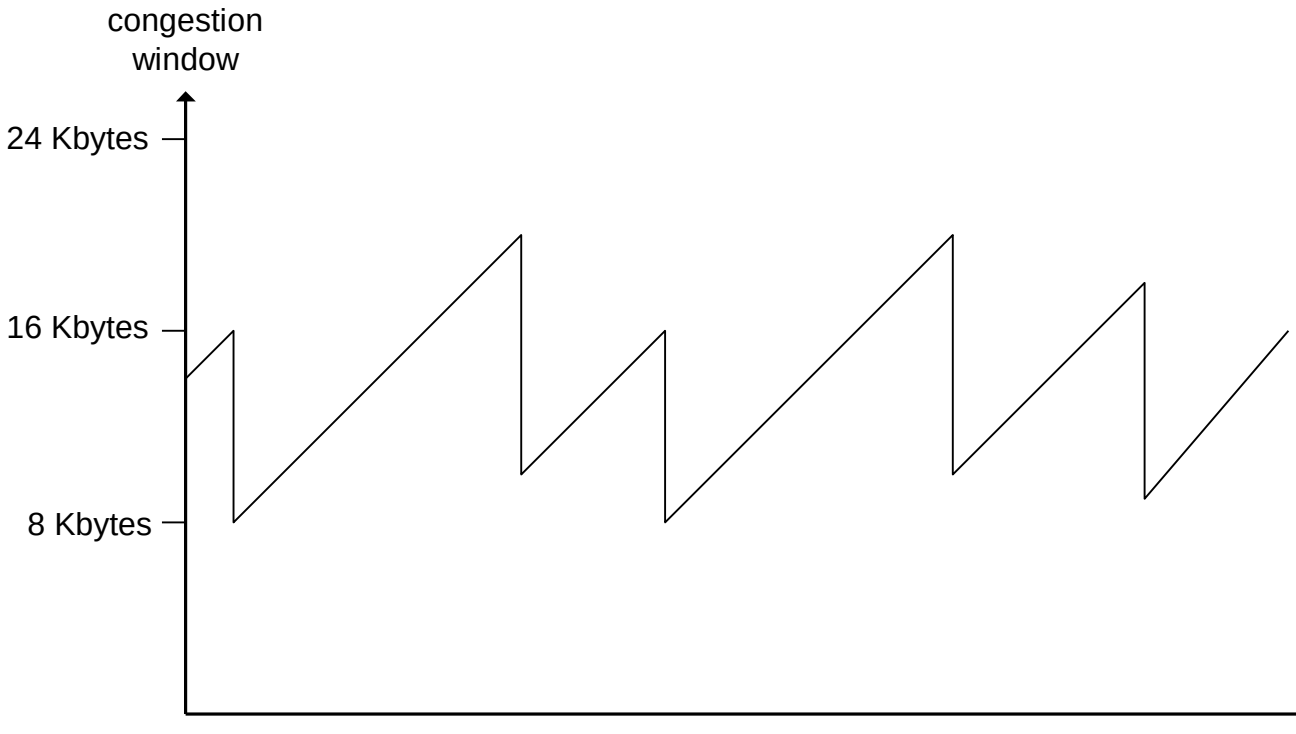
- Dette kan være, og er ofte, et stort problem!

Trafikk-kork kontroll prinsipper

- Ende-til-ende kontroll
  - Ingen tilbakemelding fra nettverket
  - Endepunkt finner selv ut om det er problemer
    - Symptomer: Forsinkelse, tap av pakke
  - TCP bruker dette prinsippet
- Nettverk-assistert kontroll
  - Ruterne gir tilbakemelding til endene
    - Setter bit-flagg i pakkehodet (SNA, ATM.....)
    - Direkte tilbakemelding (choke packet)

TCP trafikkork kontroll (AIMD)

- Metode: øk utsendelsraten (vindusstørrelsen) forsiktig, let etter tilgjenglig båndbredde, inntil det kommer pakketap.
  - Additativ økning: øk CongWin med 1 MSS (Max Segment Size) hver RTT inntil det oppstår pakketap
  - multiplikativ senkning: kutt CongWin til det halve etter pakketap.



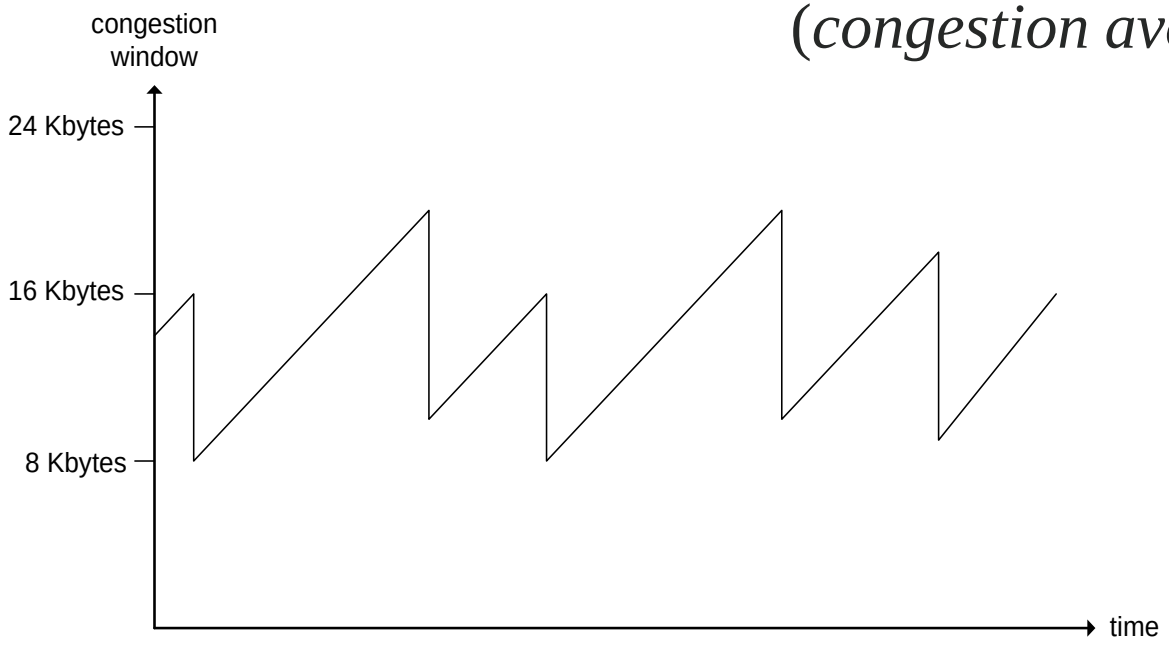
Sagtann adferd:

- Tester bånd-bredden til det blir pakketap
- Reduserer utsendelse kraftig
- Tester på nytt
- …

TCP AIMD

multiplikativ reduksjon (MD):
halvér **`CongWin`** etter segmenttap (minimum én MSS)

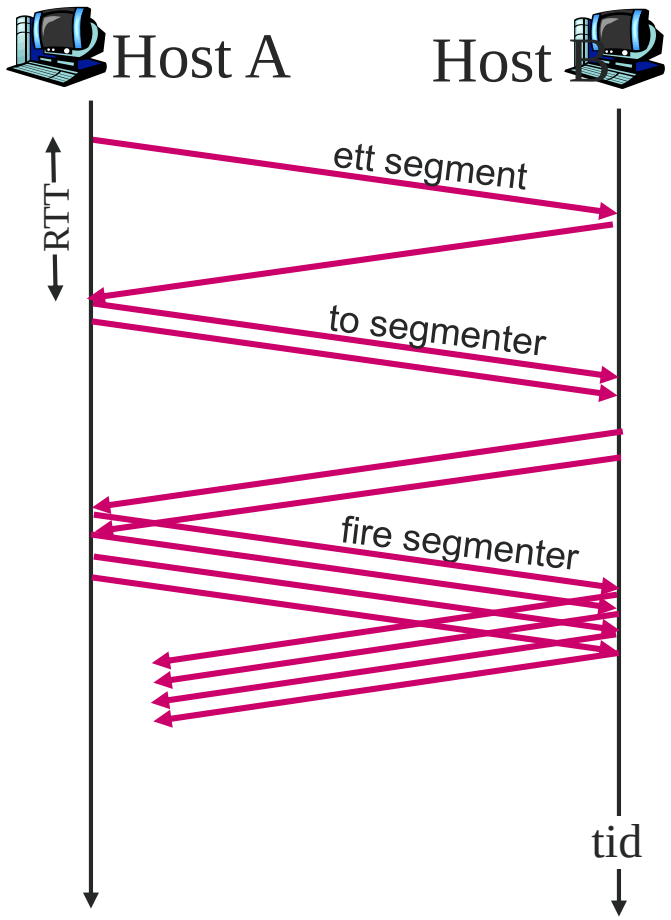
additiv økning (AI):
`CongWin` økes med én MSS for hver RTT under fravær av segmenttap (*congestion avoidance* fase)



Langlivet TCP-forbindelse

TCP Slow-start

- Ved starten på forbindelse økes raten eksponentielt inntil man opplever segmenttap:
  - dobler **`CongWin`** hver RTT
  - gjøres ved å øke **`CongWin`** for hver mottatt ACK
- Oppsummert: initiell rate er lav, men øker eksponentielt



Reaksjon på segmenttap

- Etter tre dupliserte kvitteringer:
  - **`CongWin`** halveres
  - vinduet vokser deretter lineært
- Men etter en timeout:
  - **`CongWin`** settes til én MSS
  - vinduet vokser så eksponentielt opp til halvparten av verdien før timeout, og vokser deretter lineært

Filosofi:

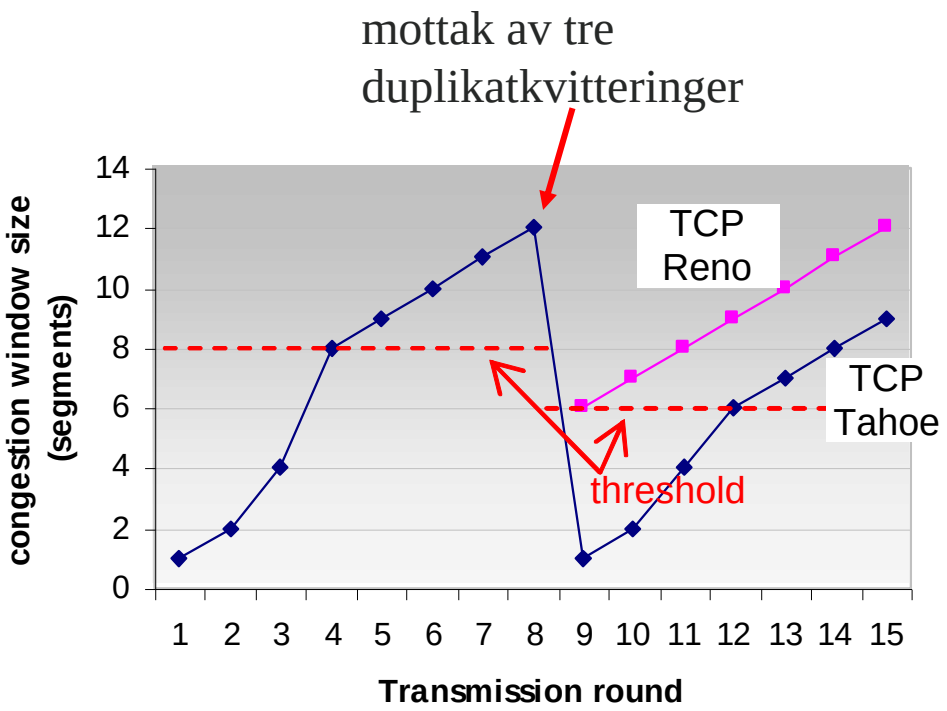
• 3 dupliserte ACK indikerer at nettet faktisk kan levere en del segmenter i og med at disse kommer gjennom

• Timeout før 3 dupliserte ACK er mer alarmerende, da ser det ut til at svært lite kommer gjennom

Reaksjon på segmenttap (forts)

Q: Når skal eksponentiell økning endres til lineær?

A: Når **`CongWin`** blir halvparten av dens verdi før timeout.



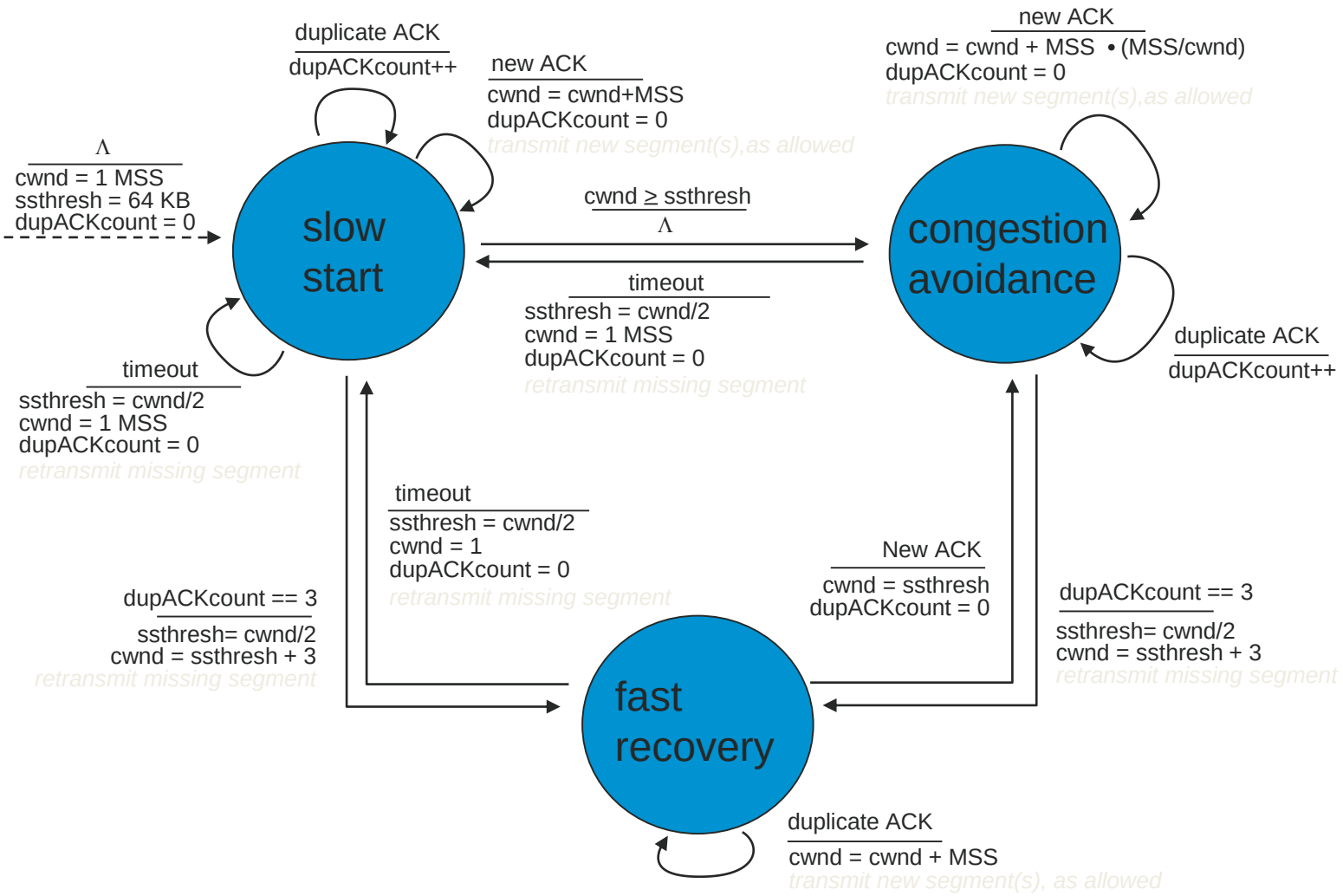
Implementasjon:

- Variabel terskel
- Ved segmenttap settes terskelen til halvparten av det CongWin var før segmenttapet

Oppsummering: TCP metningskontroll

- Når **`CongWin`** er lavere enn **`Threshold`** er senderen i slow-start fasen; vindu øker eksponentielt.
- Når **`CongWin`** er større enn **`Threshold er`** senderen i congestion-avoidance fasen; vindu øker lineært.
- Når sender mottar en trippel duplisert ACK, settes **`Threshold`** til **`CongWin/2`** og **`CongWin`** settes til **`Threshold`**.
- Når sender får timeout, settes **`Threshold`** til **`CongWin/2`** og **`CongWin`** til én MSS.

Tilstandsmaskinen...



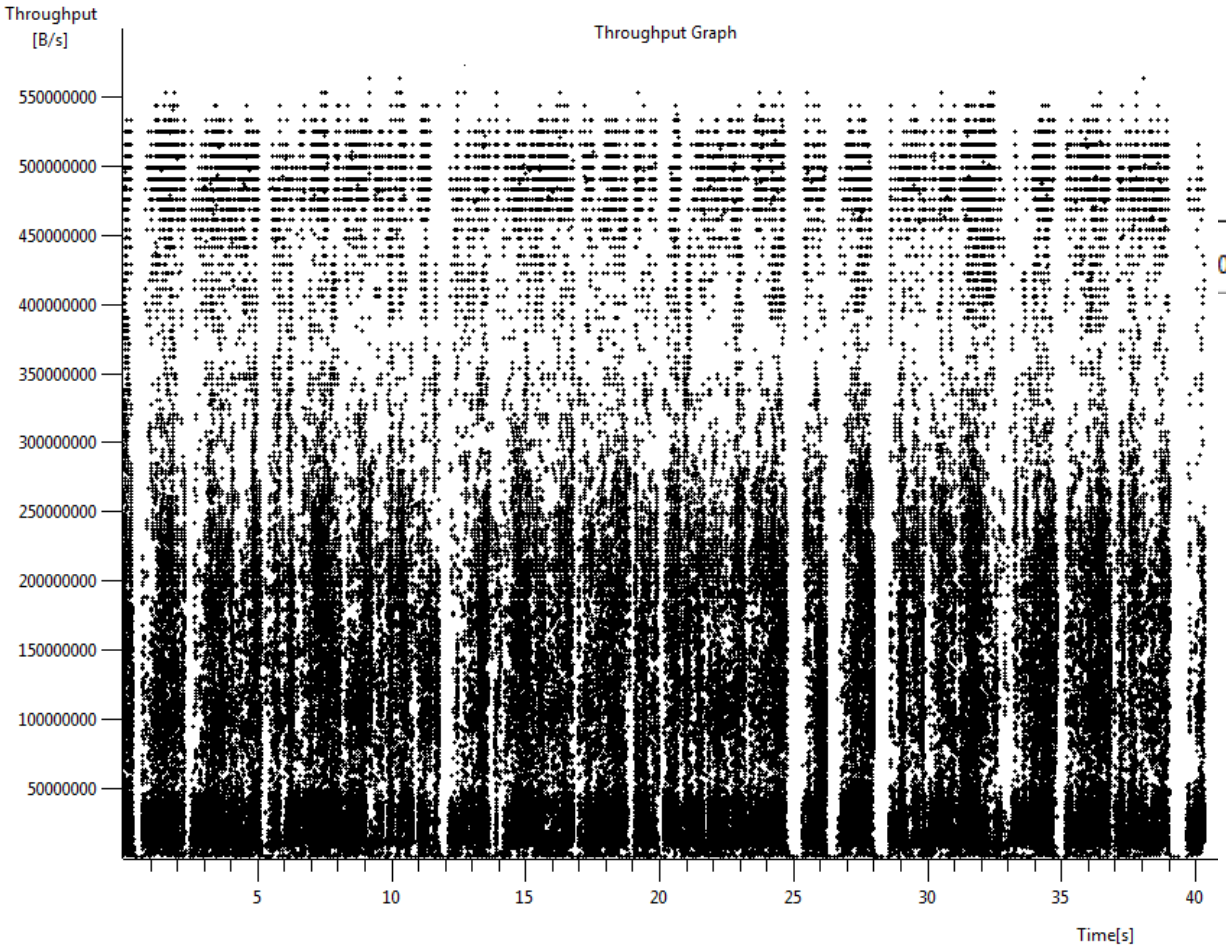

So what?

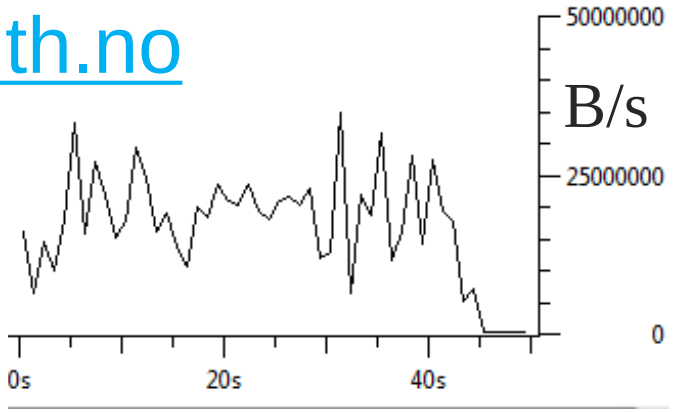
Effektiv bitrate (eksempel)

- Benytter en server i Tromsø (www.uit.no)
- RTT = ca 22 ms (målt med `pathping`)
- I klassisk TCP («vanilla») er max vindustørrelse 64 KiB
- Max (teoretisk) gjennomstrømning blir da $\frac{65535 \cdot 8\ b}{0{,}022\ s} = 23{,}8\ Mbps$,
  men det er bare dersom det aldri er segmenttap…
- Automatisk Windowscaling vil øke dette i mange moderne systemer, men dette forutsetter at også serveren støtter scaling
- **RFC 1323** angir hvordan man kan gå over til Jumbo-vinduer og oppnå enda større max gjennomstrømning

Wireshark: Gjennomstrømning

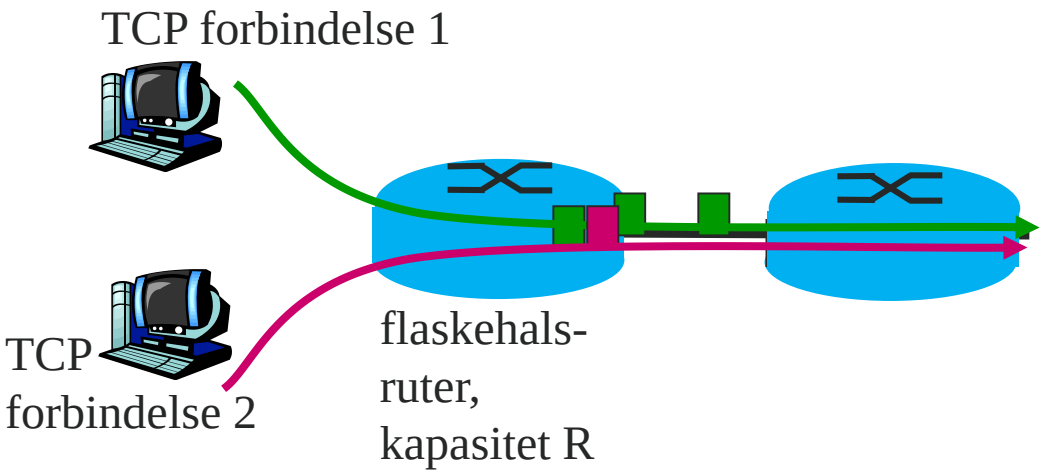
- Laster opp en 1 GB til home.nith.no





TCP rettferdighet (fairness)

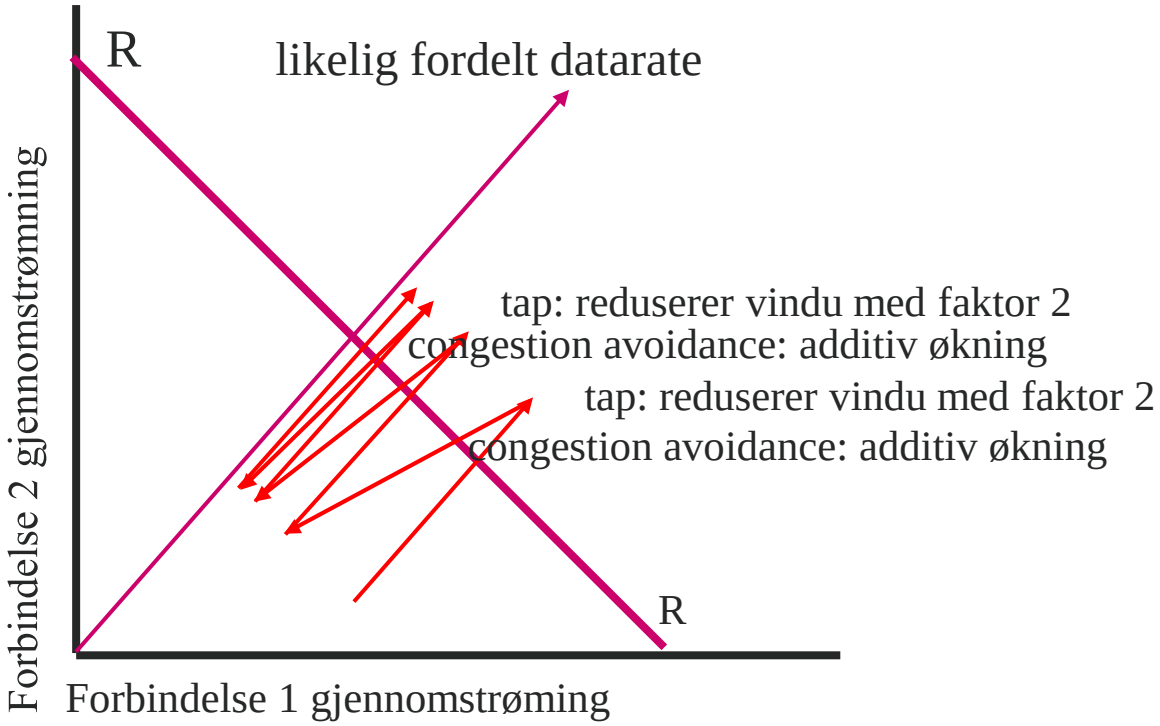
Mål: dersom K TCP-sesjoner deler samme flaskehals-link med datarate R, skal hver av dem ha en gjennomsnittlig datarate på R/K



Hvorfor er TCP rettferdig?

To konkurrerende sesjoner:

- Additiv økning gir helning på 1, som øker etterhvert
- multiplikativ reduksjon reduserer gjennomstrømning proporsjonalt



Rettferdighet (forts)

Rettferdighet og UDP

- Multimedia applikasjoner benytter «sjelden» TCP
  - ønsker ikke datarate begrenset pga trafikkork-kontroll
- Benytter isteden UDP:
  - pumper audio/video på konstant rate, tolererer pakketap
- Forskningsområde: TCP-vennlig oppførsel

Rettferdighet og parallelle TCP-forbindelser

- ingenting forhindrer appl å åpne parallelle forbindelser mellom to maskiner.
- Nettlesere gjør ofte dette
- Eksempel: link med rate R med 9 forbindelser:
  - ny appl ber om én TCP-forbindelse, får rate R/10
  - ny appl ber om 11 TCP-forbindelser, får rate R/2 !

TCP: Versjoner

- Den største forskjellen mellom ulike varianter er nettopp hvordan de håndterer metning
- Tahoe («vanilje»), Reno, New Reno, Vegas, BIC/CUBIC (Linux 2.6->), CTCP (Windows Vista/7 ->,..)
- Alle forsøker å få opp bitraten raskere igjen etter pakketap
  - Uten å utkonkurrere «vanilje» TCP
  - Nye versjoner skal være Tahoe-vennlige!

TCP: Nagle's algoritme

- TCP + IPv4 legger på 20 byte header hver
- Dersom vi bare skal overføre **en** bokstav medfører dette en stor «overhead».
  - Bare 1/41 = 2,4% av pakken er data
  - Med ACK fra server: bare 1,2% av båndbredden benyttes til noe nyttig (enda mindre dersom vi tar med linklag-headeren
- Nagle's algoritme
  - Lagrer data som skal til samme server inntil ACK på forrige pakke mottatt, eller mengden data blir >= 1 MSS
  - Problematisk i sanntidsapplikasjoner (f.eks. **online spill**) pga «delayed ACK» fra serversiden
  - Kan/må løses i programmeringen av applikasjonen

TCPs fremtid 1: TCP over "lange, tykke rør"

- Eksempel: 1500 Byte segment, 100ms RTT,
  Vi vil gjerne ha 10 Gbps gjennomstrømning
- Dette krever da en minimum vindusstørrelse på W = 83.333 segmenter på linja
- Gjennomstrømning i forhold til tapsrate (L):

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- ➜ L = $2 \cdot 10^{-10}$ = *kun ca* ***to av ti milliarder pakker kan gå tapt***
- *TCP også problematisk i trådløse nettverk, der pakketap er* ***mye*** *vanligere enn i kablede*
- **Det trengs nye versjoner av TCP!**

TCPs fremtid 2: interaktivitet over «tynne rør»

- Kan bruke UDP, men må da legge tilpassede pålitelighetsmekanismer a la TCP inn i selve applikasjonen
  - **QUIC** (Google, Chrome)
  - Kompliserer
- Fokus i FoU har så langt vært på «tykke rør» og høy latens, ikke på interaktivitet
  - Petland(2009) viser at i spillet Anarchy Online fungerer gamle (New Reno) bedre enn noen av de nyere TCP-implementeringene!
- Det trengs tillegg til TCP som oppdager og tar hensyn til sanntid og tynnt rør.

OPPSUMMERING

Hva skal vi kunne?

- Prinsipper bak transportlag-service
  - Multipleksing/demultipleksing
  - Pålitelig dataoverføring
    - Sjekksum, kvittering, sekvensnummer, timere
  - Flyt kontroll
  - Metnings-kontroll
- Implementering for Internett
  - Portnummer
  - UDP
  - TCP