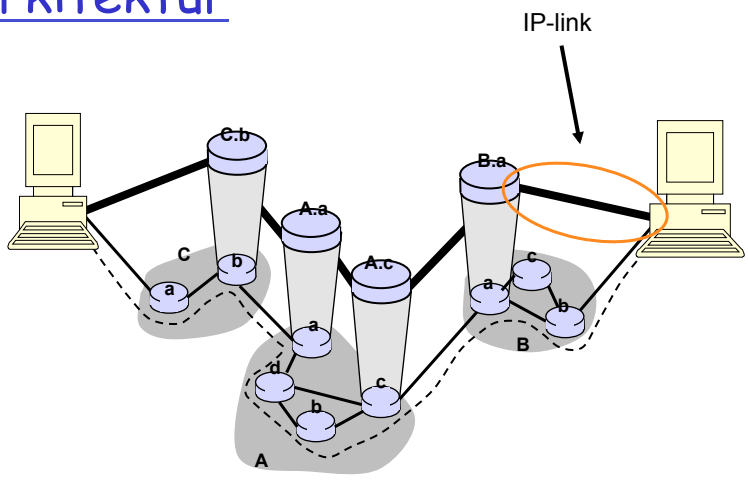
Linklaget

Olav Lysne
(med bidrag fra Stein Gjessing og Frank Eliassen)

Internettets Overlay Arkitektur



Link-typer

Tre typer av linker:

(a) Punkt-til-punkt (enkel kabel)

(b) Broadcast link (delt kabel eller annet medium; f. eks eldre Ethernet og trådløs)

(c) Svitsjet (f.eks svitsjet Ethernet og ATM)

Link Lagets tjenester

- ***Framing og linkaksess***:
  - Pakke data inn i en "ramme" (frame), og legge til hode og hale.
  - Implementere kanalaksess dersom det er delt medium.
  - 'fysiske adresser' blir brukt i rammeheaderene til å identifisere kilde og destinasjon når det er delt medium.
- ***Pålitelig levering***:
  - Sjelden brukt i fiber optikk, co-axial kabel og noen varianter av twisted pair pga. lav feil-rate.
  - Brukt på trådløslinker hvor målet er å redusere feil, og unngå ende til ende retransmisjoner.

Linklagets tjenester (mer)

- ***Flytkontroll***:
  - Hastighetsavpassing mellom sender og mottaker.
- **Feildeteksjon:**
  - Feil kommer av støy og signalreduksjon
  - Mottaker oppdager feil i mottatt ramme.
  - Den signalerer for retransmisjon, eller den bare kaster rammen.
- **Feilkorreksjon:**
  - Mekanisme hvor mottakeren retter feilen uten å be om retransmisjon.

Feilfinning/feilretting

- Oppgaver:
  - 1. Finne feil
  - 2. Rette feil
    - To alternativer til å rette feil:
    - A. Ha nok informasjon til å rette opp de mottatte dataene
    - B. Be om at dataene (rammen) blir sendt en gang til
    - (C. Gi blanke, det er ikke så farlig å miste litt data)
  - Generelt prinsipp i informatikken:
    Oppdag feilen så fort som mulig etter at den har oppstått !

Feil-deteksjon

- Bit-feil i rammer
  - behov for mekanismer som oppdager bit-feil
- Teknikker som ofte benyttes i datanett
  - Paritet - to-dimensjonal paritet
    - BISYNC ved ASCII overføring
  - Sjekksum
    - flere Internett-protokoller
  - Cyclic Redundency Check (CRC)
    - svært utbredt

Paritet (tversum)

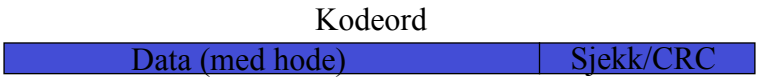
- Ett paritetsbit:
  - F.eks. 7 bit data, sendes som 8 bit
  - Like paritet dvs. et like antall enere i resultatet
  - Odde paritet dvs. et odde antall enere i resultatet
- Like paritet: 0110001 sendes som 01100011
- Odde paritet: 0110001 sendes som 01100010
- Mulig med flere paritetsbit
  - Generelt: Jo mer data til redundanse, jo flere feil oppdages.

Internett sjekksum algoritme

- Se på en melding som en sekvens av 16-biters heltall
  - Senderen adderer disse heltallene sammen ved bruk av 16-biters aritmetikk
  - Dette 16-biters tallet er sjekksummen
  - Mottaker utfører samme beregning og sammenligner resultatet med den mottatte sjekksum
  - Får mottaker feil resultat er det bitfeil enten i dataene eller i sjekksummen
- Benyttes ende til ende i Internett

CRC: Cyclic Redundancy Check

Generalisering av paritet
Punkter i et n-dimensjonalt rom



Like paritet: Kodeordet delt på 2 skal ikke gi rest
CRC: Kodeordet delt på et tall, G, skal ikke gi rest

Dette tallet vi deler på kaller vi Generatorpolynomet
Deling foregår med modulo-2 regning, dvs ikke mente eller låning.

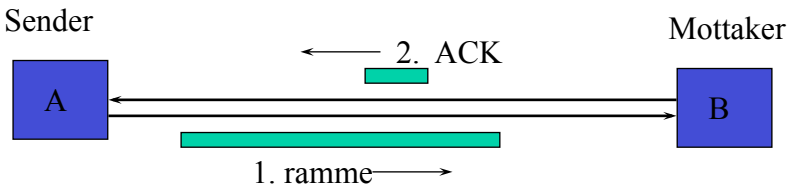
Pålitelig overføring

- rammer med feil CRC kastes
- Fint om vi kan rette opp feilen
- Hvis feilen ikke kan rettes opp, og vi trenger rammen, da
  må den sendes en gang til !
- Også her er det en avveiing:
  Ende-til ende eller mellom noder ?
  (Problemkomplekset med doblet/triplet (mm.) funksjonalitet)

Pålitelig overføring

- Når omsending av rammer er nødvendig:
- To fundamentale mekanismer
  - kvitteringer (engelsk: acknowledgements, ack)
  - tidsfrister (timeouts) vha. vekkeklokke (timer)
- Husk at også kvitteringer kan bli borte
- Ønsker vi at rammene skal komme frem i riktig rekkefølge?

Stop-and-Wait (stopp og vent)

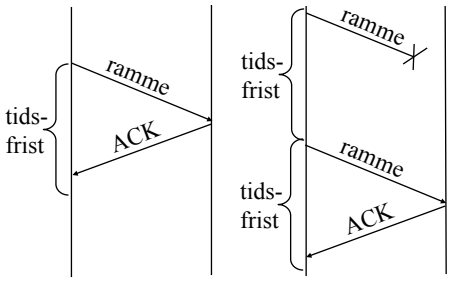


Mottaker sender ack tilbake når en ramme er mottatt,
og først når sender mottar ack, sendes ny ramme.
På denne måten blir ikke mottaker oversvømmet av rammer,
og avsender vet at alle rammer er kommet trygt fram.

Men hva hvis rammer blir borte?

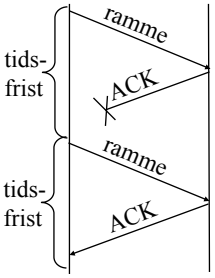
Stop-and-Wait (stopp og vent)

- Grunnleggende algoritme:
  - send én ramme og vent på kvittering (ACK ramme)
  - dersom ACK ikke mottatt innen gitt tidsfrist, send rammen på nytt.



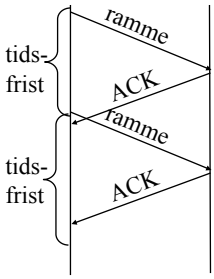
Stop-and-Wait

- Problem 1 med grunnleggende algoritme:
  - Men kanskje det var ACK som ble borte
  - Vi må kunne sende den samme rammen på nytt, selv om den allerede er kommet riktig frem



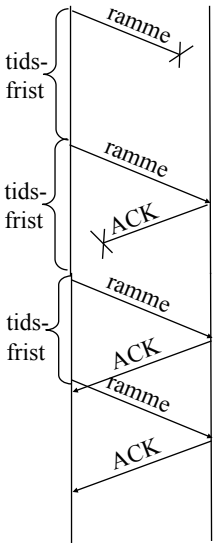
Stop-and-Wait

- Problem 2 med grunnleggende algoritme:
  - Kanskje vi sendte rammen omigjen for tidlig
  - Vi må godta at ACK kommer for sent

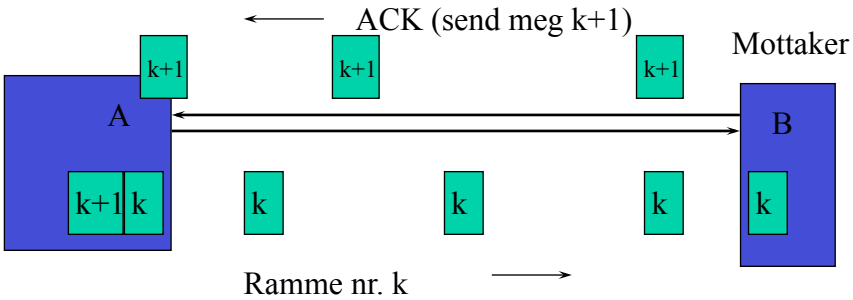


Stop-and-Wait

- Må sende rammen på nytt og på nytt helt til ACK kommer tilbake



Løsning: sekvensnummer



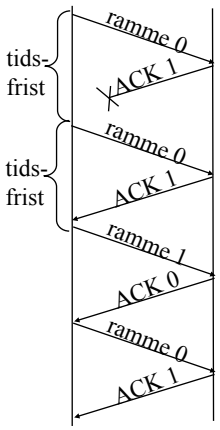
Neste bilde:
Det er nok med en en-bit teller (0 og 1)
k, k+1 regnes da ut modulo 2.
(En buffers "Sliding window" protokoll)

Sekvensnummer som 0 og 1

- 0 og 1 som sekvensnummere
  - En bit er nok når vi har én ramme ad gangen

  ACK 1: Send ramme med odde sekvensnummer
  ACK 0: Send ramme med like sekvensnummer

- Altså:
  ramme 0, 1, 2, 3, 4, 5, … sendes som
  ramme 0, 1, 0, 1, 0, 1, …



0 - 1 sekvensnummer

- Går dette bra?
- Ja, fordi:

  - like ramme (0) sendes ut
  - ignorerer (gamle) ACK 0
  - resender like ramme (0)

  I det ACK 1 kommer:

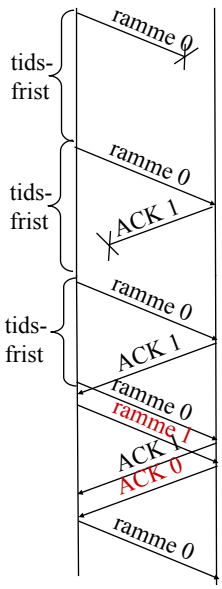
  - odde ramme (1) sendes ut
  - ignorerer (gamle) ACK 1
  - resender odde ramme (1)

  I det ACK 0 kommer:

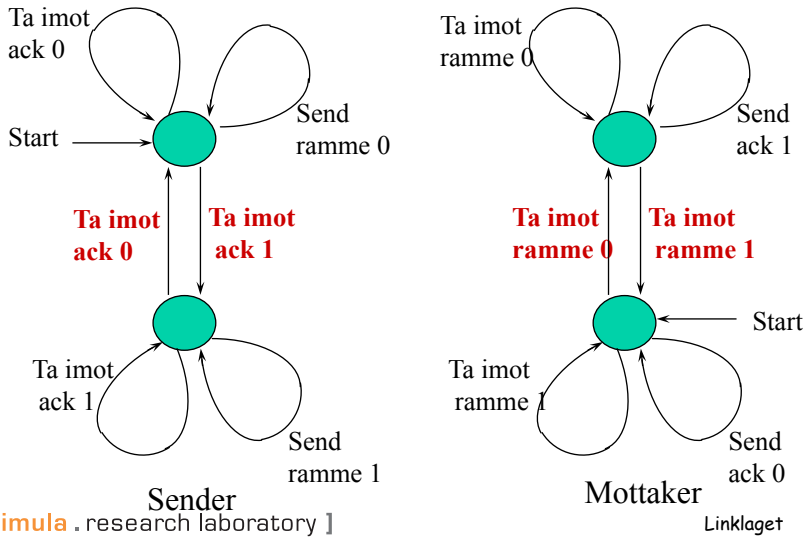
  - like ramme (0) sendes ut
  - ignorerer (gamle) ACK 0
  - resender like ramme (0)

  I det ACK 1 kommer:

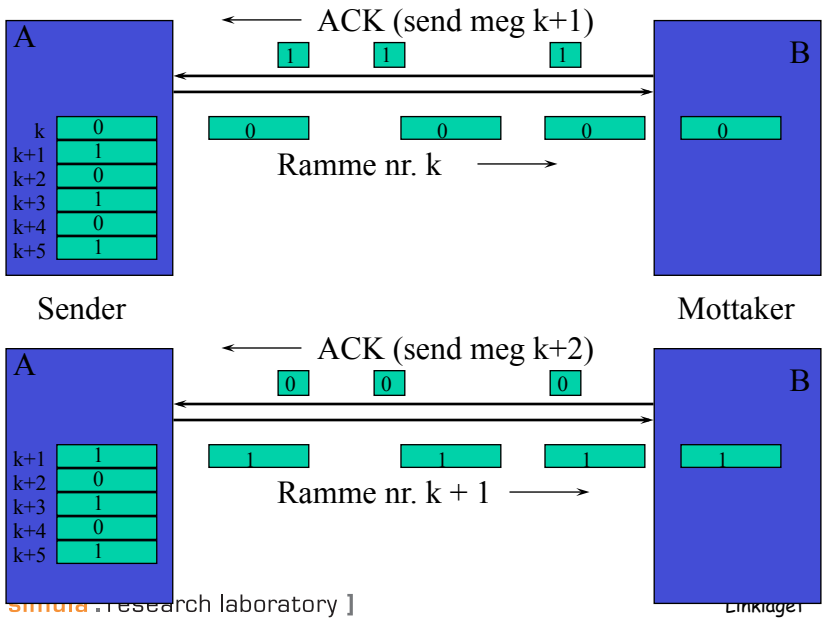
  osv.



Tilstandsmaskin for en bit protokoll

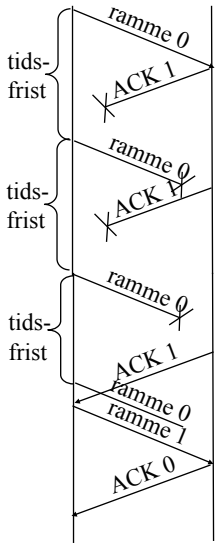


Et bit protokoll



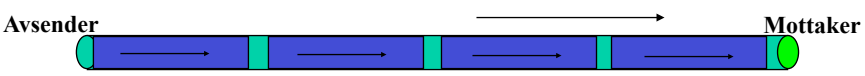
Stop-and-Wait

- Det gjør ikke noe om mottaker gjentar en ACK, men det er ikke vanlig, dvs. at det er bare vanlig å sende ACK når en ramme ankommer. Ved mye rammetap kan det være gunstig å gjenta ACK selv om det ikke kommer en ny ramme



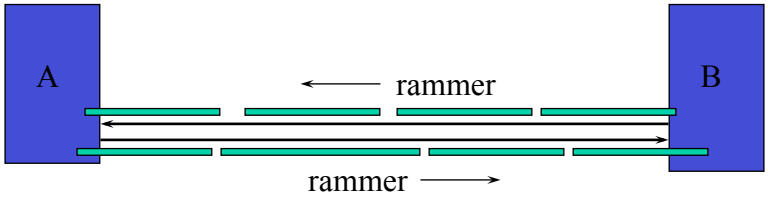
Stop-and-Wait

- Grunnleggende svakhet:
  - utnytter linjekapasiteten dårlig
    - senderen kan bare ha én utestående ramme til enhver tid
- Eksempel:
  - 1.5Mbps link x 45ms RTT = 67.5Kb (8KB)
    - dvs. 8KB data kan sendes før første ack kan ventes tilbake
  - anta rammestørrelse 1KB
  - stop-and-wait bruker ca. 1/8 av linjekapasiteten (om ingen feil)
  - Mål: senderen må kunne sende opp til 8 rammer før den må vente på en ACK
    - moralen er: "fyll opp røret"



Fyll opp røret

Utnytte linjen bedre.
Sender flere rammer rett etter hverandre:

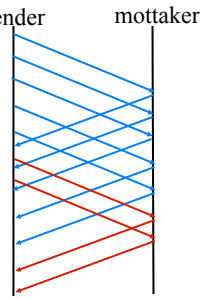


Putte ACK/NAK på ryggen til meldinger som går den andre veien ("piggyback")

Glidende vindu

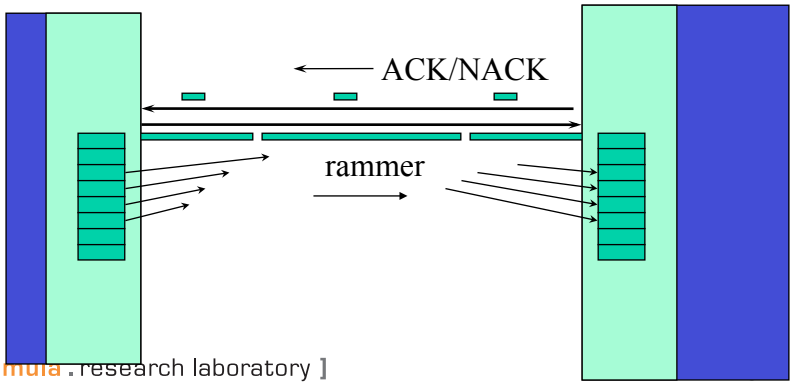
- Idé:
  - Tillat senderen å sende flere rammer før den mottar ACK for derved å holde "røret fullt".
  - Det må være en øvre grense på antall rammer som kan være utestående (som det ikke er mottat ACK for).

Eksempel:
maks. 5
utestående
rammer



Glidende vindu

Flere bufre hos sender og flere bufre hos mottaker,
mange rammer med forskjellige nummer og mange ack/nack med forskjellige nummer underveis hele tiden.

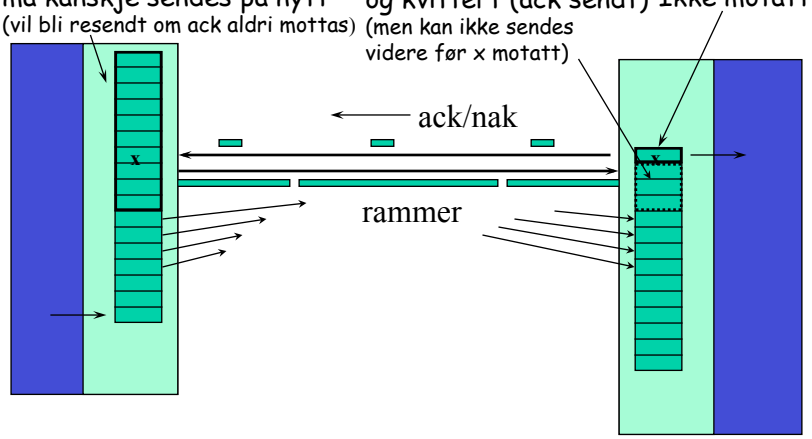


Glidende vindu

Sendt men ikke fått ack,
må kanskje sendes på nytt
(vil bli resendt om ack aldri mottas)

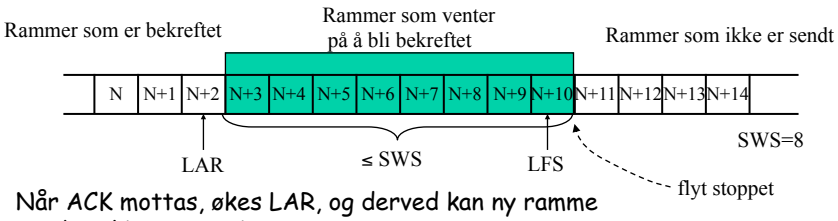
Disse tre er motatt
og kvittert (ack sendt)

Ikke motatt
(men kan ikke sendes
videre før x motatt)



Glidende vindu: sender

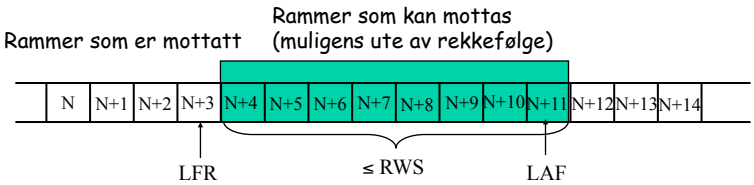
- Tilordner sekvensnummer til hver ramme (SeqNum)
- Vedlikeholder tre tilstandsvariable
  - send window size (**SWS**)
  - last acknowledgment received (**LAR**)
  - last frame sent (**LFS**)
- Vedlikeholder invariant: **LFS - LAR ≤ SWS**



Når ACK mottas, økes LAR, og derved kan ny ramme sendes slik at LFS økes.
Buffer for opptil SWS rammer, dvs SWS rammer i "røret" samtidig

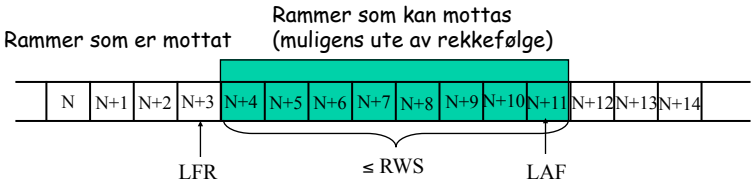
Glidende vindu: mottaker

- Vedlikeholder tre tilstandsvariable
  - receive window size (**RWS**)
  - largest acceptable frame (**LAF**)
  - last frame recived (**LFR**) (med alle "mindre" rammer også motatt)
- Vedlikeholder invariant: **LAF - LFR ≤ RWS**



Glidende vindu: mottaker

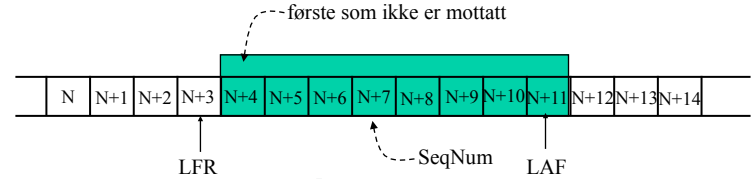
- Kumulativ kvittering ("go back n" protokoll), dvs. vi ack-er ikke nye rammer hvis det er hull i sekvensen av mottatte rammer
  - **if** LFR < MottatRamme.SeqNum < LAF **then**
    ta-imot-rammen-og-legg-den-på-plass;
    beregn-nye-grenser (MottatRamme.SeqNum); // se neste lysark
    else
    kast rammen;
    send ACK (LFR + 1)



Glidende vindu: mottaker

- Invariant: LFR + 1 er første ramme som ikke er mottatt
- bergen-nye grenser(`seqNo`)

```
• if seqNo = LFR + 1
  {    beregn ny LFR og LAF:
       for (i= LFR + 1; ramme i er mottatt; i++) { } ;
       LFR = i-1;
       LAF = LFR + RWS;
  }
```



Glidende vindu: mottaker

- Varianter til "go back n" protokoll
  - Negativ kvittering (NAK)
    - mottaker sender NAK på rammer som savnes
  - Selektiv kvittering (SAK)
    - mottaker sender ACK på nøyaktig de rammer som mottas
    - disse behøver da ikke re-sendes
  - Både SAK og NAK øker kompleksiteten til implementasjonen, men kan potensielt bedre utnyttelsen av linjen

Glidende vindu: endelige sekvensnummer

- I praksis representeres sekvensnummer med et endelig antall biter.
- n biters sekvensnummer => intervall sekv. nr. = (0..2^n-1)
- I HDLC: n = 3, dvs. sekvensnummerintervall (0..7)
  - ⇒ sekvensnummer må gjenbrukes

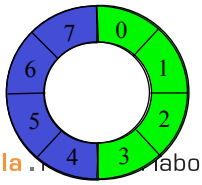
	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	

Problem
skille mellom ulike inkarnasjoner av samme numre

Minimum krav:
sekvensnummerintervallet må være større enn maks. antall utestående rammer

Glidende vindu: endelige sekvensnummer

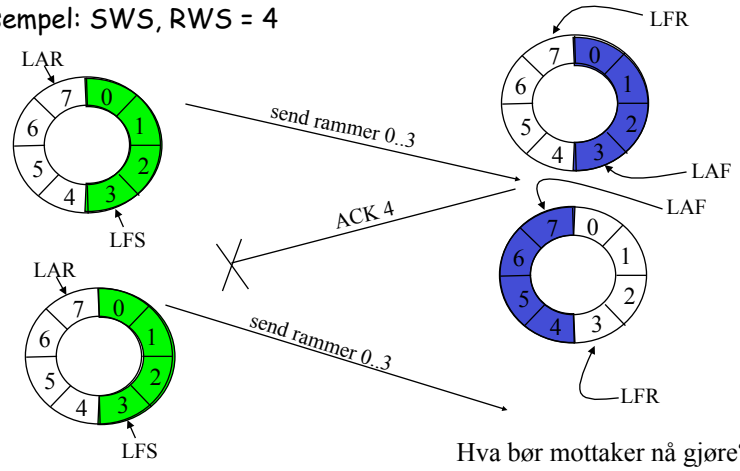
- `SWS ≤ MaxSeqNum+1` er ikke tilstrekkelig
  - anta 3 biters `SeqNum` felt (0..7)
  - `SWS=RWS=8`
  - senderen transmitterer rammene `0..6`
  - mottas uten feil, men ACK går tapt
  - senderens tidsfrist utløper, rammene `0..6` sendes på nytt
  - mottaker forventer `7,0..5`, men mottar andre inkarnasjon av `0..5`
- `SWS,RWS ≤ (MaxSeqNum+1)/2` er riktig regel



Hindrer overlapp mellom *nedre kant* av sendervinduet og *øvre kant* av mottakervinduet.

Glidende vindu: endelige sekvensnummer

- Eksempel: SWS, RWS = 4

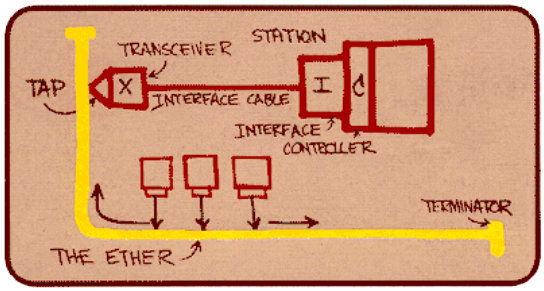


Hva bør mottaker nå gjøre?

Ethernet

"dominant" wired LAN technology:

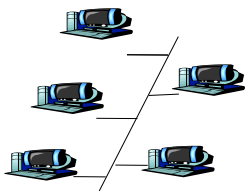
- cheap $20 for NIC
- first widely used LAN technology
- simpler, cheaper than token LANs and ATM
- kept up with speed race: 10 Mbps – 10 Gbps



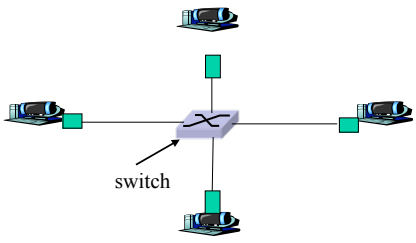
Metcalfe's Ethernet sketch

Star topology

- bus topology popular through mid 90s
  - all nodes in same collision domain (can collide with each other)
- today: star topology prevails
  - active *switch* in center
  - each "spoke" runs a (separate) Ethernet protocol (nodes do not collide with each other)



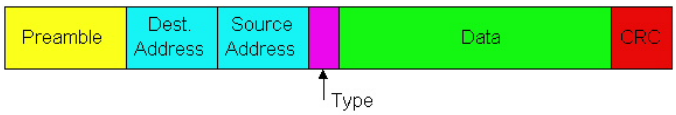
bus: coaxial cable



star

Ethernet Frame Structure (more)

- Addresses: 6 bytes
  - if adapter receives frame with matching destination address, or with broadcast address (eg ARP packet), it passes data in frame to network layer protocol
  - otherwise, adapter discards frame
- Type: indicates higher layer protocol (mostly IP but others possible, e.g., Novell IPX, AppleTalk)
- CRC: checked at receiver, if error is detected, frame is dropped



Ethernet: Unreliable, connectionless

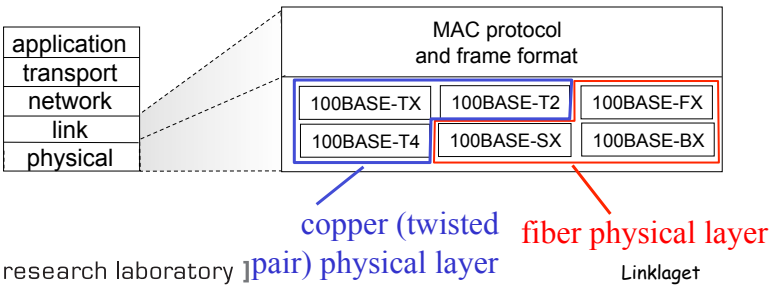
- connectionless: No handshaking between sending and receiving NICs
- unreliable: receiving NIC doesn't send acks or nacks to sending NIC
  - stream of datagrams passed to network layer can have gaps (missing datagrams)
  - gaps will be filled if app is using TCP
  - otherwise, app will see gaps
- Ethernet's MAC protocol: unslotted CSMA/CD

Ethernet CSMA/CD algorithm

1. NIC receives datagram from network layer, creates frame
2. If NIC senses channel idle, starts frame transmission If NIC senses channel busy, waits until channel idle, then transmits
3. If NIC transmits entire frame without detecting another transmission, NIC is done with frame !
4. If NIC detects another transmission while transmitting, aborts and sends jam signal
5. After aborting, NIC enters **exponential backoff**: after *m*th collision, NIC chooses *K* at random from $\{0,1,2,\ldots,2^m-1\}$. NIC waits $K \cdot 512$ bit times, returns to Step 2

802.3 Ethernet Standards: Link & Physical Layers

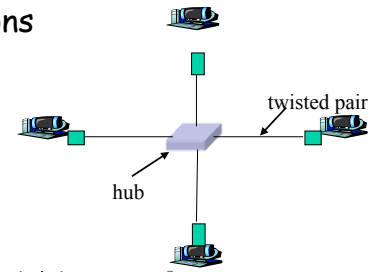
- *many* different Ethernet standards
  - common MAC protocol and frame format
  - different speeds: 2 Mbps, 10 Mbps, 100 Mbps, 1Gbps, 10G bps
  - different physical layer media: fiber, cable



Hubs

… physical-layer ("dumb") repeaters:

- bits coming in one link go out *all* other links at same rate
- all nodes connected to hub can collide with one another
- no frame buffering
- no CSMA/CD at hub: host NICs detect collisions

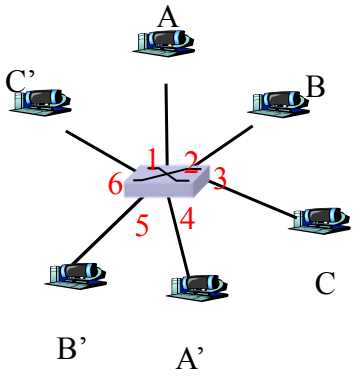


Switch

- link-layer device: smarter than hubs, take *active* role
  - store, forward Ethernet frames
  - examine incoming frame's MAC address, selectively forward frame to one-or-more outgoing links when frame is to be forwarded on segment, uses CSMA/CD to access segment
- *transparent*
  - hosts are unaware of presence of switches
- *plug-and-play, self-learning*
  - switches do not need to be configured

Switch: allows *multiple* simultaneous transmissions

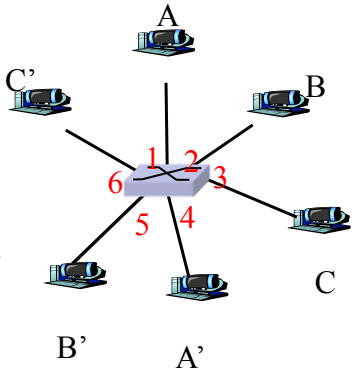
- hosts have dedicated, direct connection to switch
- switches buffer packets
- Ethernet protocol used on *each* incoming link, but no collisions; full duplex
  - each link is its own collision domain
- *switching:* A-to-A' and B-to-B' simultaneously, without collisions
  - not possible with dumb hub



switch with six interfaces (1,2,3,4,5,6)

Switch Table

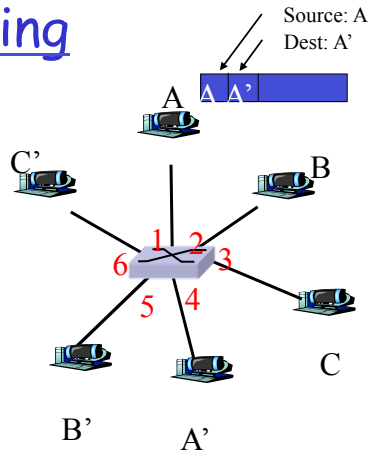
- *Q:* how does switch know that A' reachable via interface 4, B' reachable via interface 5?
- *A:* each switch has a switch table, each entry:
  - (MAC address of host, interface to reach host, time stamp)
- looks like a routing table!
- *Q:* how are entries created, maintained in switch table?
  - something like a routing protocol?



switch with six interfaces (1,2,3,4,5,6)

Switch: self-learning

- switch *learns* which hosts can be reached through which interfaces
  - when frame received, switch "learns" location of sender: incoming LAN segment
  - records sender/location pair in switch table



MAC addr	interface	TTL
A	1	60

Switch table (initially empty)

Switch: frame filtering/forwarding

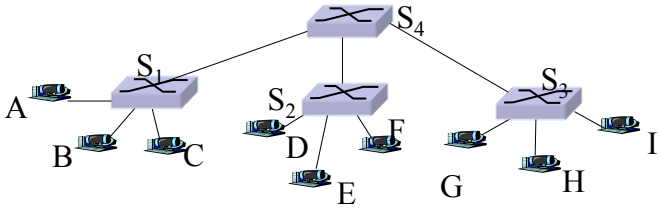
When frame received:

1. record link associated with sending host
2. index switch table using MAC dest address
3. **if** entry found for destination
   then {
   if dest on segment from which frame arrived
   then drop the frame
   else forward the frame on interface indicated
   }
   else flood

forward on all but the interface on which the frame arrived

Interconnecting switches

- switches can be connected together



  - Q: sending from A to G - how does S_1 know to forward frame destined to F via S_4 and S_3?
  - A: self learning! (works exactly the same as in single-switch case!)