

# Kommunikasjon - Tjenester og nettverk

Skrevet av Henrik Høiness

Repository for teori og øvinger til Algoritmer og datastrukturer - TDT 4120.

Under står notater fra både forelesninger, kompendium og Computer Networking - A Top Down Approach (Kurose, Ross)

## Liste over kapitler

1. [Kapittel 1 - Computer Networks and the Internet](#)
2. [Kapittel 2 - Application Layer](#)
3. [Kapittel 3 - Transport Layer](#)
4. [Kapittel 4 - The Network Layer](#)
5. [Kapittel 5 - The Link Layer: Links, Access Networks, and LANs](#)
6. [Kapittel 6 - Wireless and Mobile Networks](#)
7. [Kapittel 7 - Multimedia Networking](#)
8. [Kapittel 8 - Security in Computer Networks](#)
9. [Kapittel 9 - Network Management](#)

## Kapittel 1 - Computer Networks and the Internet

### What Is the Internet

Internettet er et nettverk som kobler sammen flere hundre millioner dataenheter i verden. Alle disse enhetene kaller vi **hosts** eller **ende systemer**.

Endesystemene er koblet sammen gjennom et nettverk av kommunikasjonskoblinger (eng. *communication links*) og pakkesvitsjere (eng. *packet switches*).

- En **packet switch** tar en pakke som kommer på en av dets *innkommende* kommunikasjonskoblinger og videresender den pakken på en av dets *utgående* kommunikasjonskoblinger.

De mest brukte typene av pakkesvitsjere idag er *rutere* (i nettverkskjernen) og *link-layer-swictches* (i aksessnettverk)

Sekvensen av kommunikasjonskoblinger og pakkesvitsjere traversert av en pakke fra det sendende endesystemet til det mottakende endesystemet er kalt en **sti** (eng. *route el. path*) gjennom nettverket.

Vi kan på mange måter sammenlikne pakkesvitsje nettverk (som transporterer pakker) med en transportsystem av motorveier, veier, kryss (som transporterer trailer mellom fabrikker):

Pakker --> Trailer

Komm.koblinger --> Motorveier og veier

Pakkesvitsjer --> Veikryss

Fabrikker --> Endesystemer

Endesystemene aksesserer internettet gjennom **internetleverandører (ISP-er)** (eng *Internet Service Providers*). Her inngår bolig ISP-er (som kabel- eller telefonelskaper), bedrift ISP-er, universitets ISP-er o.l.

Hver ISP er i seg selv et nettverk av pakkesvitsjær og kommunikasjonskoblinger. ISP-er tilbyr forskjellige typer nettverkstilganger til endesystemer, som f.eks. kabelmodem, DSL, LAN, WLAN o.l. ISP-er tilbyr også internetttilgang til innholdsleverandører, som kobler opp nettsider til internettet. Det hele handler om å sammenkoble endesystemer.

Internettet bruker protokoller som kontrollerer sending og mottakelse av informasjon i internettet. **Transmission Control Protocol (TCP)** og **Internet Protocol (IP)** er de to viktigste protokollene i internettet.

IP protokollen spesifiserer formatet til pakkene som blir sent og mottatt over rutere og endesystemer. Internettets prinsipp protokoller er *kollektivt* kjent som **TCP/IP**

**Internet standarder** blir utviklet av Internet Engineering Task Force (IETF). IETF standard dokumentene er kalt for **request for comments (RFCs)**. RFCs er veldig tekniske og detaljserte. De definerer protokoller som TCP, IP, HTTP og SMTP.

Vi kan også se på internettet som en *infrastruktur som tilbyr tjenester til applikasjoner*. Internet applikasjoner kjører på endesystemer - de kjører ikke i pakkesvitsjene i nettverkskjernen.

- Endesystemer som er tilkoblet internettet har en **Application Programming Interface (API)** som spesifiserer hvordan et program som kjører på et endesystem spør internett-infrastrukturen om å levere data til en spesifik program-destinasjon som kjører på et annet endesystem.
  - Internet API-er er et sett med regler som det sendene programmet må følge slik at internettet kan levere dataen til program-destinasjonen.

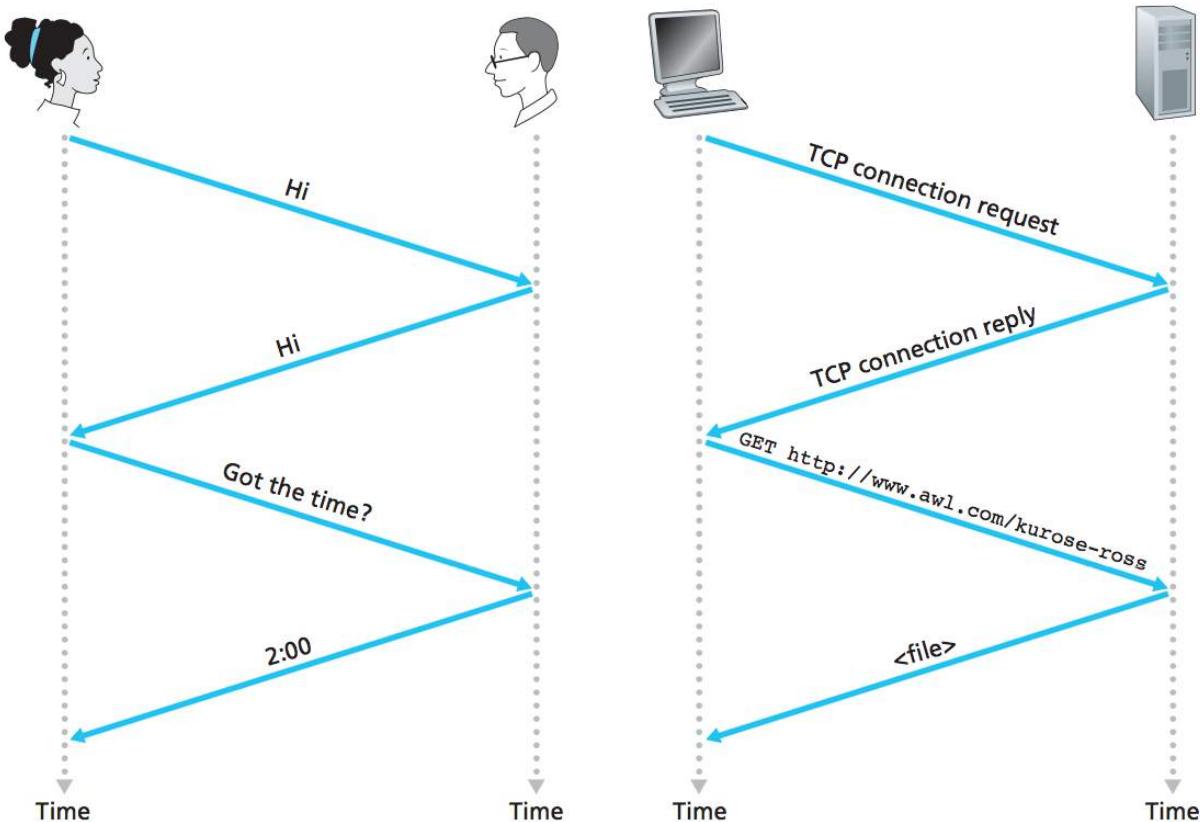
## What is a protocol?

### A Human Analogy

For å forstå hva en nettverksprotokoll er kan vi først se på noen humane analogier. Se for det at du ønsker å spørre noen om hva klokken er. Da vil man typisk:

1. Si "Hei" - initiere kommunikasjon med en annen
2. Få responsen "Hei" tilbake - En responsmelding som indikerer at du kan fortsette å spørre om hva klokken er
3. Du spør "Hva er klokken?" - siden det var en hyggelig respons, tør du å spørre hva klokken var.
4. Du får vite at klokken er "2:00:

Dersom den første responsmeldingen hadde vært uhyggelig, så hadde du kanskje ikke turt å spørre om tiden. Sånn fungerer også nettverksprotokollene.



**Figure 1.2** ♦ A human protocol and a computer network protocol

### Nettverksprotokoller

En nettverksprotokoll er lik til en human protokoll, bortsett fra at entitetene utveksler melinger og aksjoner er hardware eller software komponenter av en eller annen enhet.

All aktivitet i internettet som involverer to eller flere kommuniserende entiteter blir overvåket av en protokoll.

*A protocol defines the format and the order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event.*

### The Network Edge

Endesystemer blir også referert til som **verter** (eng. hosts) fordi de verter (som betyr, kjører) applikasjonsprogrammer som en Web-browser program, Web-server program, e-mail klient program eller en e-mail server program.

*host = end system*

Hoster blir i mange tilfeller delt videre inn i: **klienter** og **servere**. Klienter pleier å være skrivebords- eller mobile PCer, smarttelefoner osv, mens servere pleier å være sterke maskiner som lagrer og distribuerer Websider, strømmer videoer, releér eposter osv.

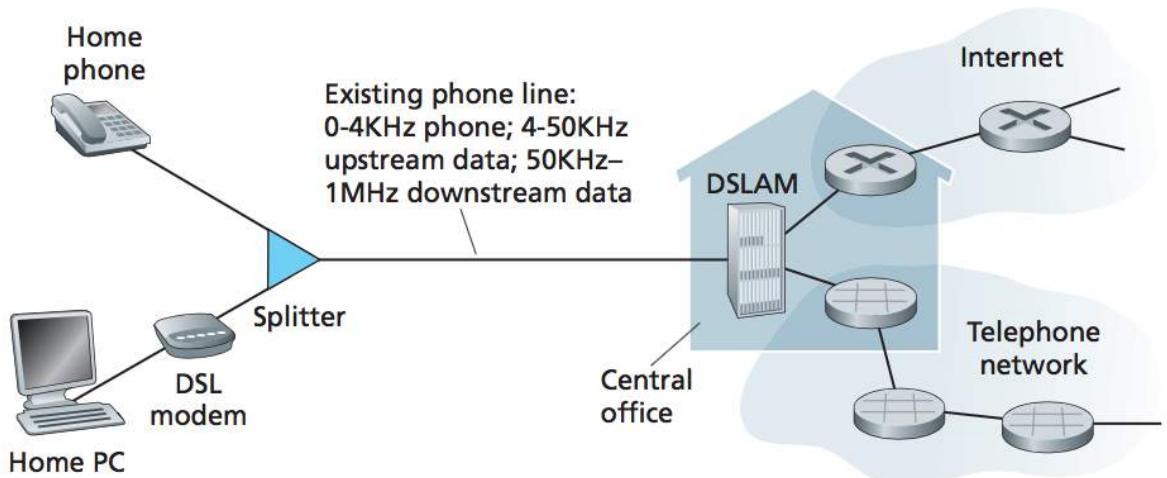
### Aksessnettverk

La oss nå se på aksessnettverket - nettverket som fysisk kobler sammen et endesystem til den første ruten (også kjent som "edge router") på en *sti* fra endesystemet til et annet endesystem.

I dag er de to vanligste typene av bredbåndsaksess i bolig **digital subscriber line (DSL)** og kabel.

- DSL Internettaksess får man ofte fra det lokale telefonselskapet (telco) - Når en bruker DSL er kundens telco også dens ISP.

På telco-en har de en DSLAM som skiller mellom data- og telefonsignaler

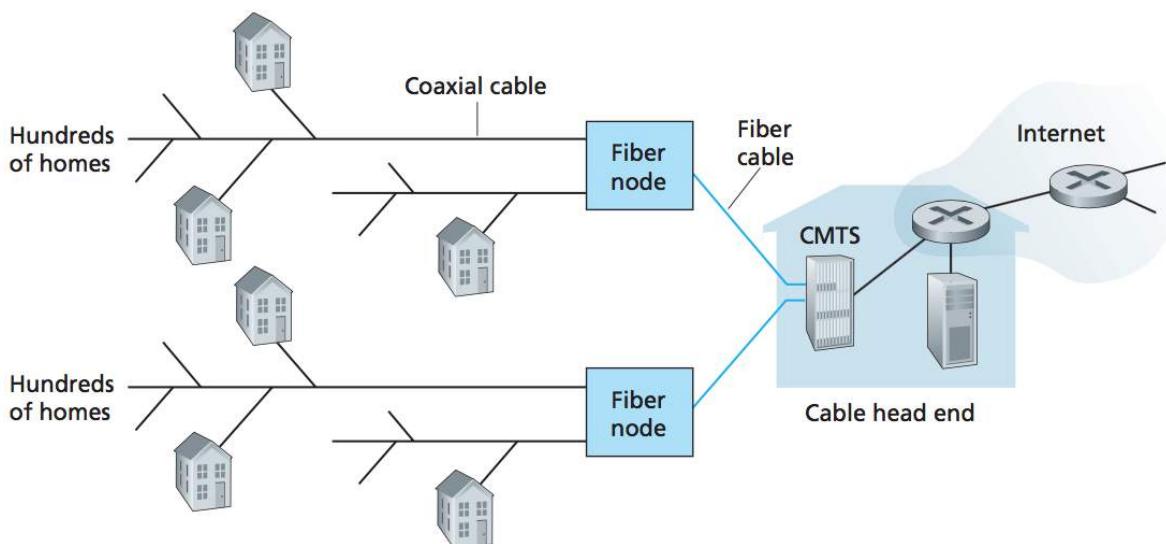


**Figure 1.5 ♦** DSL Internet access

- Kabel internettaksess bruker kabel-TV leverandørene sin eksisterende kabel-TV infrastruktur.
  - Kabel internettaksess trenger spesielle modemer - kalt kabelmodemer.

På kabelhode-enden har de en CMTS (cable modem termination system) som endrer de analoge signalene sendt fra kabelmodemet til digitalt format.

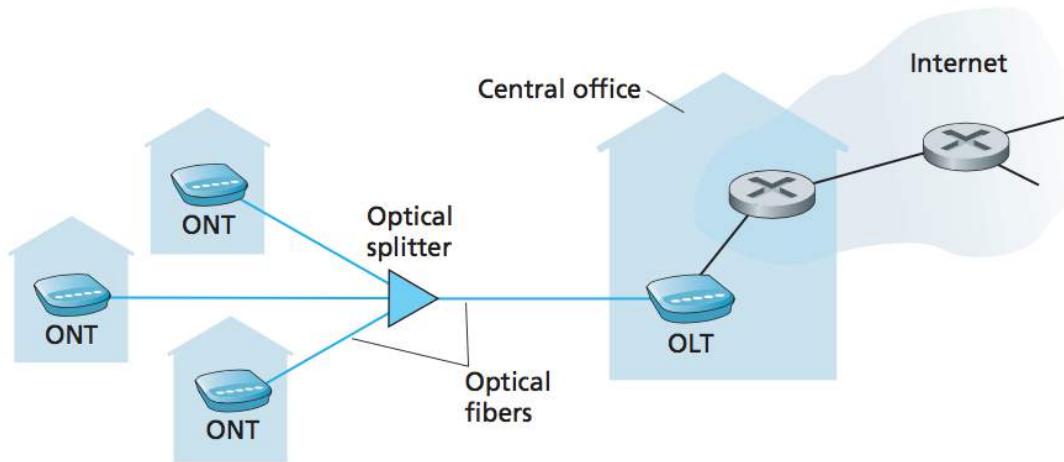
- DSL-standarder definerer overføringshastigheter av 12 Mbps nedstrøms og 1.8 Mbps oppstrøms [ITU 1999], og 23 Mbps nedstrøms og 2.5 Mbps oppstrøms [ITU 2003].



**Figure 1.6 ♦** A hybrid fiber-coaxial access network

Selvom DSL og kabelnettverk står for over 90% av bredbåndsaksess i bolig i USA, så er det kommet en ny teknologi **fiber to the home (FTTH)** - fibernett.

- Bruker PON distribusjonsarkitektur.
  - Hvert hjem har en optisk nettverksterminator (ONT), som er koblet, med optisk fiber, til en nabolagssplitter.
  - Splitteren kombinerer et antall hus på en enkel, felles optisk fiber, som kobles til en optisk linjeterminator (OLT) hos nettverksleverandøren



#### Wide-Area Wireless Access: 3G and LTE:

Alle mobil-enheter som sender mails, surfer og hører på musikk når de ikke er hjemme, er koblet til den samme trådløse infrastrukturen som brukes til mobiltelefoni til å sende/motta pakker via en basestasjon som drives av mobilnettleverandøren.

I motsetning til WiFi, trenger en bruker bare å være innenfor noen få tiier kilometer (i motsetning til noen få ti meter) av basestasjonen.

#### Fysiske medier:

- *Guided media:*
  - Twinnede kobberkabler
  - Coaxkabler
  - Fiber-optisk kabel
- *Unguided media:*
  - Trådløs LAN
  - Digital satellittkanal

#### Satellitter:

- Geostationary satellites
- Low-earth orbiting (LEO) satellites

## The Network Core

#### Packet Switching:

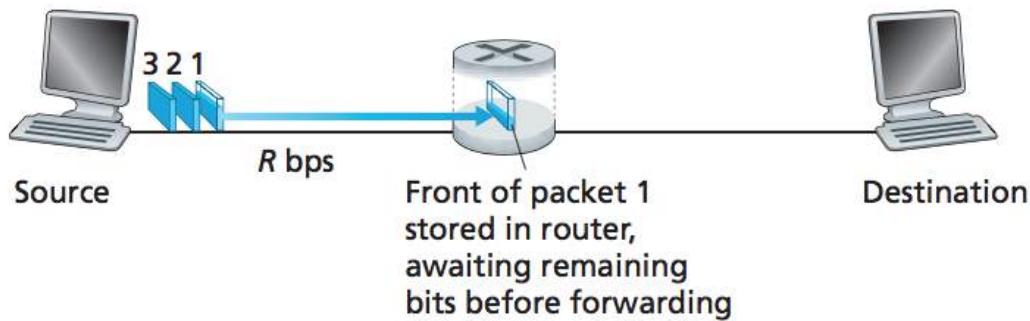
For å kunne sende en melding fra et kildeendesystem til en destinasjonsendesystem, må kilden dele opp lengre meldinger til mindre klumper med data, kjent som **pakker**.

Mellan kilden og destinasjonen, reiser hver pakke gjennom kommunikasjonskoblinger (eng. *comm. links*) og **pakkesvitjser** (som det finnes to typer, **rutere** og **link-layer svitsjer**). Pakkene blir sendt over hver kommunikasjonskobling med en rate lik en fulle **overføringsrate** (eng. *transmissionrate*) til koblingen.

Så dersom en kilde eller pakkesvitsj sender en pakke av  $L$  bit over en kobling med en transmissionrate  $R$  bits/sec, da tar det  $L/R$  sekunder å sende pakken over koblingen

#### Store-and-Forward Transmission:

De fleste pakkesvitjser bruker **store-and-forward transmission** på inputten på koblingen. Denne type overføring betyr at pakkesvitsjen må motta hele pakken før den kan begynne å sende den første biten av pakken på den utgående koblingen.



Kilen begynner å sende pakken ved tid 0, ved tid  $L/R$  har kilden sendt hele pakken og pakken har blitt mottatt og lagret på ruteren, og ruteren begynner å sende videre. Ved tid  $2L/R$  har ruteren sendt hele pakken og pakken har blitt mottatt av destinasjonen. Den totale delayen er da  $2L/R$ . Dersom det er flere koblinger, får vi at ende-til-ende forsinkelsen er:

$$d_{\text{end-to-end}} = N \frac{L}{R}$$

#### Queueing Delays and Packet Loss:

Hver pakkesvitsj har flere koblinger koblet til seg. For hver kobling har pakkesvitsjer en output queue, som lagrer pakkene som ruteren er på vei til å sende på koblingen. Dersom en kobling allerede frakter en pakke, må den neste pakken vente i bufferen/queue-en. Dette fører til queueing delays, disse forsinkelsene er varierende og avhenger av fordøyelsen til nettverket.

Siden størrelsen til en buffer er definert, så dersom en pakke kommer til en full buffer, vil **pakketap** oppstå.

#### Forwarding Tables and Routing Protocols:

Hver ruter har et *forwarding table* som mapper destinasjonsadressen til ruterens utgående koblinger. Når en pakke kommer til ruteren, ser den på pakkens adresse og søker i forward table-en sin, for å finne en passende utgående kobling. Ruteren leder deretter pakken til denne utgående koblingen. Internettet har en rekke **routing protokoller** som brukes til automatisk bestemme **forwarding tables**.

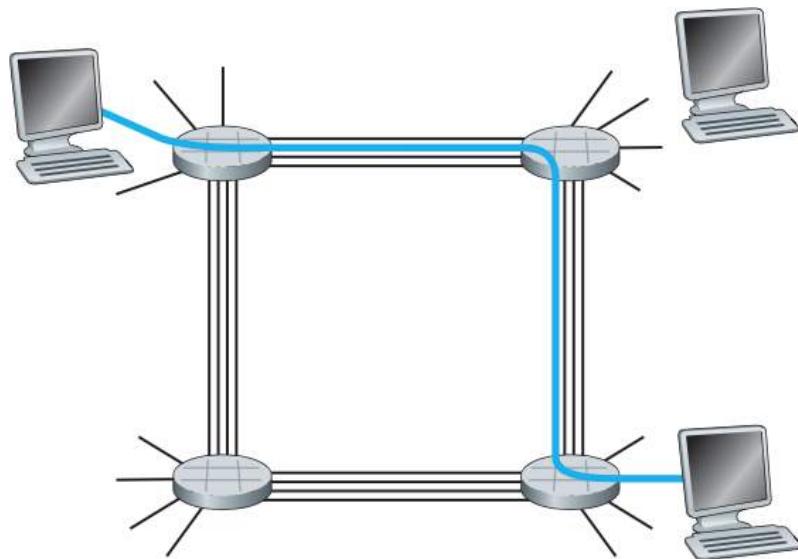
#### Circuit Switching:

Det er to fundamentale måter å flytte data gjennom et nettverk av koblinger og svitsjer: *circuit switching* og *packet switching*.

I circuit-switched nettverk er ressursene som trengs langs en sti (buffere, koblinger og overføringsrate) for å kunne kommunisere mellom endesystemene *reservert* varigheten av kommunikasjonsøkten mellom endesystemene.

I packet-switched nettverk er ikke disse ressursene reserverte, ressursene brukes etter etterspørsel, og som konsekvens må man kanskje vente (queue).

Vanlig telefonnettverk er eksempler på circuit-switched nettverk. Når man ringer noen, vil nettverket etablere en tilkobling mellom senderen og mottakeren, og linjen og overføringsraten vil være reservert helt til de leger på. Her har man garantert konstant rate.



**Figure 1.13** ♦ A simple circuit-switched network consisting of four switches and four links

For hver ikobling bruk av en ende-til-ende forbindelse, får denne forbindelsen en fjerdedel av den totale overføringskapasiteten under tilkoblingens varighet. Dersom hver kobling mellom tilstøtende svitsjer har en overføringshastighet på 1Mbps, da får hver ende-til-ende circuit-switch forbindelse 250 kbps av dedikert overføringshastighet.

Det blir ofte argumentert for at pakkesvitsjing ikke passer så bra for real-time tjenester (f.eks. telefoni og videokonferanser), på grun av variansen og den uforutsigbare ende-til-ende forsinkelsen. Pakkesvitsjing er billigere, mer effektiv og tilbyr en bedre deling av overføringskapasitet enn circuit-switching.

#### Nettverk av nettverk

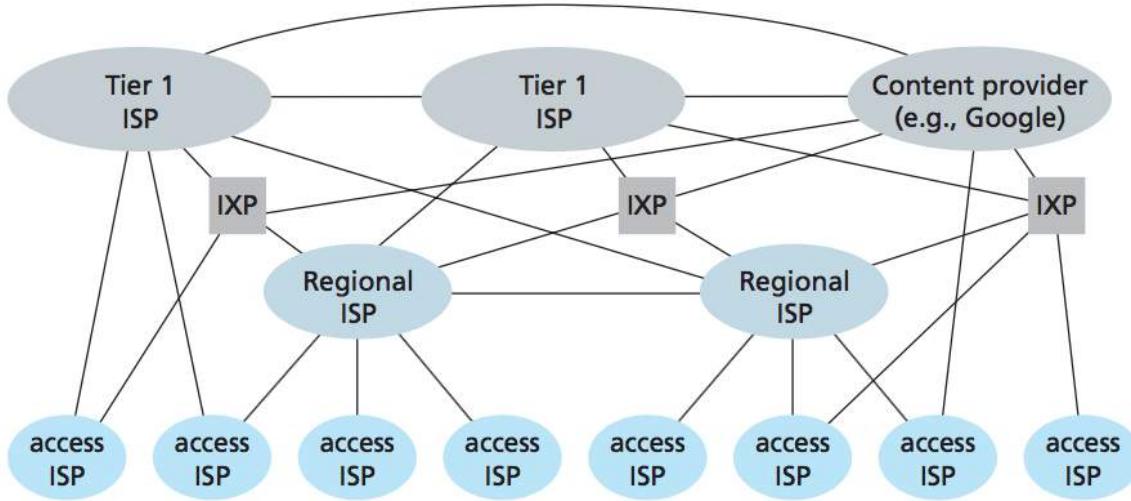
Endesystemer kobler seg til internettet gjennom en aksess ISP. ISP-ene selv må også være sammenkoblet. Dette gjøres ved å lage et *nettverk av nettverk*.

- *Network Structure 1*: En global transit ISP, **provider**, og mange aksess ISP-er, **customers**.
- *Network Structure 2*: Flere globale transit ISP-er, **providers**, og mange aksess ISP-er, **customers**.

I virkeligheten, selvom mange ISP-er er imponerende verdensdekkede og kobles direkte mot mange aksess ISP-er, er det ingen som har tilstedeværelse i hver eneste by i hverden. Istedet i enhver gitt region kan det være en **regional ISP** som aksess ISP-ene i regionen tilkobles. Hver regionale ISP kobler seg så til **tier-1 ISP-er**. Tier-1 ISP-er er like til den imaginære *global transit ISP-en*

For eksempel er det i Kina aksess ISP-er i hver by, som kobler seg til provinsielle ISP-er, som igjen kobler til nasjonale ISP-er, som til slutt kobler til ISP-ene i tier-1. Vi refererer til dette multi-tier hierarkiet, som fortsatt er bare en grov tilnærming av dagens internett, som *Network Structure 3*.

Internet Exchange Point (IXP) er møtepunktet hvor flere ISP-er kan peere sammen.



**Figure 1.15** ♦ Interconnection of ISPs

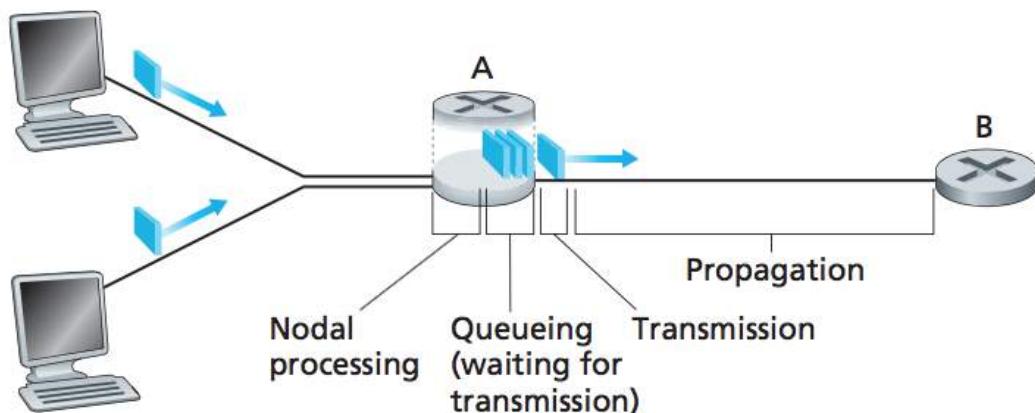
### Delay, Loss and Throughput in Packed-Switched Networks

#### Overblikk av forsinkelse i pakkesvitsjede nettverk

Når en pakke reiser fra en node (host eller ruter) til en påfølgende node (host eller ruter) langs denne veien vil pakken lide av flere typer forsinkelser ved hver node lags denne veien. De viktigste av disse forsinkelsene er:

- Nodal processing delay
- Queueing delay
- Transmission delay
- Propagation delay

Sammen gir disse en total nodalforsinkelse.



**Figure 1.16** ♦ The nodal delay at router A

#### I - Processing Delay

Tiden det tar å undersøke en pakkeheader og bestemme hvor man skal lede pakken er en del av prosesseringsforsinkelsen. Forsinkelsen kan også inkludere andre faktorer som tiden det tar for å skjekke for bit-nivå feil i pakken som skjedde under overføringen av pakkens bit fra oppstrømsnoden til ruter A. Etter dette retter ruten pakken til køen som går foran koblingen til ruter B.

#### II - Queuing Delay

I køen vil pakkene oppleve queuing delay mens den venter på å bli overført på koblingen. Lengen på køforsinkelsen til en spesifikk pakke er avhengig at antall tidlig ankommede pakkene som ligger i køen. Dersom køen er tom og det ikke er noen pakke som holder på å bli overført, vil køforsinkelsen være null. På den andre siden vil forsinkelsen være lang dersom det er stor trafikk og mange pakkene i køen.

### III - Transmission Delay

Dersom man antar at pakkene blir sendt på i first-come-first-served rekkefølge, som er vanlig i pakkesvitsjede nettverk, kan pakken først bli sendt etter at alle pakkene som kom før den har blitt overført. Pakken har  $L$  bits, og skal bli sendt over en kobling med en overføringsrate fra ruter A til ruter B på  $R$  bits/sec. Overføringsforsinkelsen blir da  $L/R$ .

Tiden ruteren bruker på å dytte ut pakken

### IV - Propagation Delay

Så fort en bit er pushet over på koblingen, må den forplanter seg til ruter B. Tiden det tar fra begynnelsen av koblingen til ruter B er forplantningsforsinkelsen. Biten forplanter seg med farten til koblingen. Forplantningshastigheten avhenger av det fysiske mediet til koblingen. Propagations delayet er avstanden mellom de to ruterne delt på forplantningshastigheten. Som blir  $d/s$  der  $d$  er avstanden mellom ruterne og  $s$  er forplantningshastigheten på koblingen.

#### Nodal forsinkelse:

Summen av alle forsinkelsene: I, II, III, IV.

$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

#### Ende-til-ende forsinkelse:

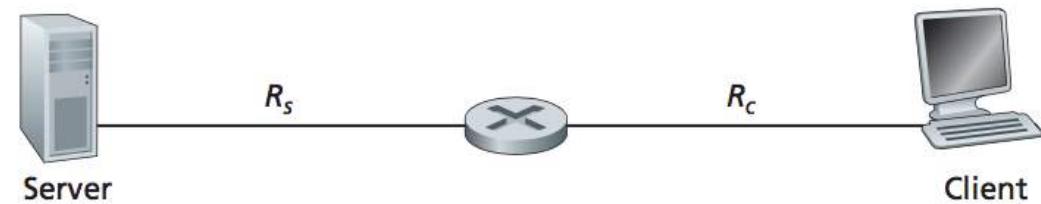
Anta at en har  $N$  koblinger, dvs.  $N-1$  rutere, mellom en kildehost og en destinasjonshost. Da vil summen av alle de nodale forsinkelsene gi ende-til-ende forsinkelsen:

$$d_{\text{end-end}} = N(d_{\text{proc}} + d_{\text{trans}} + d_{\text{prop}})$$

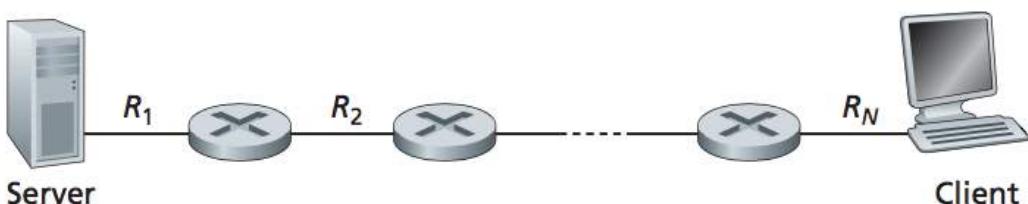
#### Gjennomstrømning i datanettverk:

- **Insantaneous throughput** ved enhver tid er raten (i bits/sec) som en host mottar en fil.
- **Average throughput** er gjennomsnittsraten ved overføring av en fil på  $F$  bit blir  $F/T$  bits/sec

Throughput er raten på antall vellykkede meldinger levert over en kommunikasjonskanal. Vi må se på flaskehals koblingen i ende-til-ende tilkoblingen for å finne gjennomstrømningen.



a.



b.

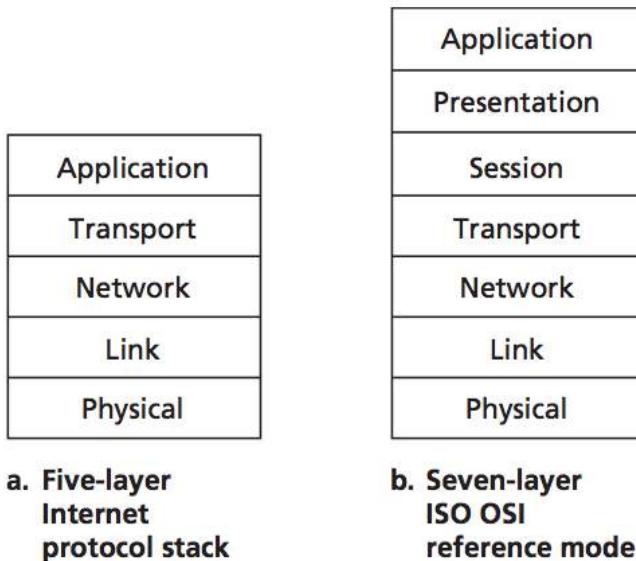
**Figure 1.19 ♦ Throughput for a file transfer from server to client**

a) Her vil gjennomstrømningen være  $\min \{R_c, R_s\}$ , det vil si overføringsraten til flaskehalskoblingen.

b) Her vil gjennomstrømningen være  $\min \{R_1, R_2, \dots, R_N\}$ , som igjen vil si overføringsraten til flaskehalskoblingen.

### Protocol Layers and Their Service Models

For å ha struktur i designet av nettverksprotokoller, organiserer vi protokollene i **layers**. Hver protokoll hører til i en av lagene. Vi er interessert i hvilke **services** som et lag har å tilby til laget over - den såkalte **service modellen**.



**Figure 1.23 ♦ The Internet protocol stack (a) and OSI reference model (b)**

### Application Layer

Applikasjonslaget er der nettverksapplikasjoner og deres applikasjonslag-protokoller ligger. Internettets applikasjonslag inkluderer mange protokoller som HTTP-, SMTP- og FTP-protokollen. Vi skal se at visse nettverksfunksjoner, som oversetting av menneskevennlige navn for internett-endesystemer som [www.ietf.org](http://www.ietf.org) til en 32-bit nettverksadresse, er gjort ved hjelp av en spesifikk applikasjonslag protokoll, DNS (The Domain Name System)

En application-layer protokoll er distribuert over flere endesystemer, med applikasjonen i et endesystem som bruker protokollen for å sende pakker av informasjon til applikasjonen på et annet endesystem. Vi referer til denne pakken av informasjon på applikasjonslaget for en **melding** (eng. message)

## Transport Layer

Internettets transportlag transporterer *applikasjonslag-meldinger* mellom applikasjon-endepunkter. I internettet er det to transportprotokoller, TCP og UDP, som hver og en kan transportere applikasjonslag-meldinger.

TCP protokollen tilbyr en tilkoblingsorientert tjeneste til sine applikasjoner. Denne tjenesten inkluderer garantert leveranse av app.lag-meldinger til destinasjonen og flytkontroll (vs. sender/mottaker hastighetsmatching, for å hindre en rask avsender fra å overvelde en langsom mottaker). TCP deler også opp lengre meldinger til kortere segmenter og tilbyr en overbelastsesstyring, slik at kilden kan endre overføringshastigheten sin når nettverket er overbelastet.

UDP protokollen tilbyr en tilkoblingsløs tjeneste til sine applikasjoner. Dette er en no-frills (ingenting ekstra) tjeneste som verken gir pålitelighet, flytkontroll og ingen overbelastningsstyring.

I denne boken refererer vi til en transportlagspakke som et **segment**.

## Network Layer

Internettets network layer er ansvarlig for å flytte nettverkslagpakker kjent som **datagrammer** fra en vert til en annen. Internett transportlagprotokoll (TCP eller UDP) i en kildevert sender et transportlagssegment og en destinasjonsadresse til nettverkslaget, slik som man leverer et brev til Posten.

Internettets nettverksprotokoll inkluderer IP protokollen, som definerer fletene i datagrammet i tillegg til hvordan endesystemer og rutere handler på disse feltene. Det er bare en IP protokoll, og alle internettkomponenter som har et nettverkslag må bruke IP protokollen. Nettverkslaget består også av routing protokoller som skal bestemme hvilke ruter datogrammene skal ta mellom kilde og destinasjon.

## Link Layer

Internettets nettverkslag sender et *datagram* gjennom en rekke rutere mellom kilden og destinasjonen. For å flutte en pakke fra en node (vert eller ruter) til den neste noden på en rute, bygger nettverkslaget på tjenestene til link layer-et (*koblingslaget*).

Ved hver node, gir nettverkslaget datagrammet ned til koblingslaget, som leverer datagrammet til neste node langs ruten. Ved neste node leverer koblingslaget datagrammet opp igjen til nettverkslaget.

Tjenestene som tilbys av koblingslaget avhenger av den spesifikke koblingslagsprotokollen som blir brukt over koblingen. Eksempler på koblingslagprotokoller er Ethernet og WiFi.

Nettverkslaget vil motta forskjellige tjenester fra hvert av de forskjellige koblingslagprotokollene. I denne boken referer vi til en koblingslagspakke som **frames** (nor. *rammer*)

## Physical Layer

Mens mye av jobben til koblingslaget er å flytte på *rammer* fra et nettverkselement til det etterfølgende nettverkselementet, er jobben til det fysiske laget å flytte de *individuelle bitene* innad i rammen fra en node til den neste.

Protokollene i dette laget er igjen koblingsavhengige og avhenger også på koblingens faktiske overføringsmedium (f.eks. tvinnede-kobberør, optisk fiber, etc.)

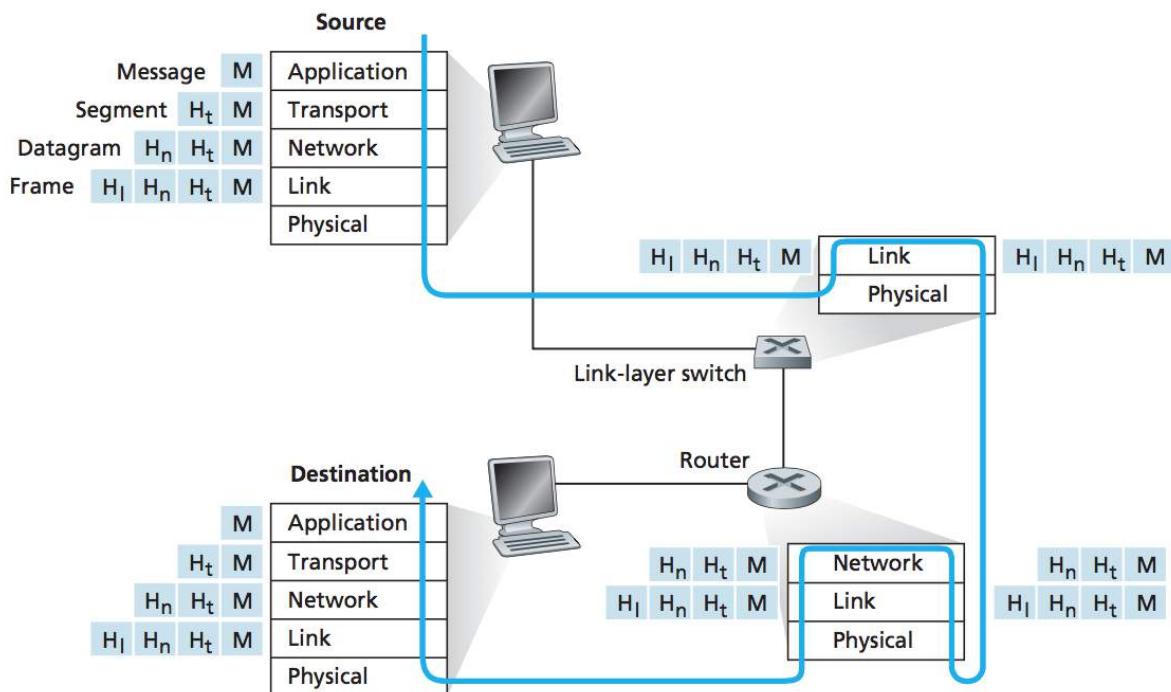
## The OSI Model

På slutten av 70-tallet ble det foreslått av The International Organization for Standardization (ISO) at datanettverk skulle være organisert rundt 7 lag, kalt Open Systems Interconnection (OSI) model [ISO 2012].

## Innkapsling

Figuren under viser den fysiske stien data tar ned et sendende endesystems protokollstakk, up og ned protokollstakkene til en mellomliggende linklagssvitsj og ruter, og opp protokollstakken til det mottakende endesystemet.

Ved den sendende verten, blir en **applikasjonslagsmelding** (*M*) sendt til transportlaget. Transportlaget tar meldingen og legger til ekstra informasjon (såkalt transportlagsheader-informasjon  $H_t$ ). Applikasjonslagsmeldingen og transportlagheaderen utgjør sammen **transportlagssegment**. Transportlaget får så segmentet, som så legger å nettverklagsheader-informasjon ( $H_n$ ), slik som kilde- og destinasjonsendesystemenes adresser, som utgjør **nettverklagsdatagram**. Datagrammet blir nå levert til koblingsagget som legger til sin egen koblingslagheader-informasjon ( $H_l$ ) og lager en **koblinglagsramme**.



**Figure 1.24** ◆ Hosts, routers, and link-layer switches; each contains a different set of layers, reflecting their differences in functionality

## Nettverk under angrep

På internettet finnes det veldig mye bra, men i tillegg til dette finnes det også noe som heter **skadelig programvare** (eng. *malware*), som kan bryte seg inn og angripe enhetene våre. Så fort skadelig programvare har infisert en enhet kan den gjøre mye slemt, slette alle filer, installere spyware som samler inn privat informasjon og sender det til de som angrep enheten. Man kan også bli en del av et nettverk av tusen like kompromitterte enheter, kjent som et **botnet**, som kan brukes til leveranse av spam e-mail og distributed denial-of-service attacks (DDOS) mot bestemte verter.

Mye av dagens skadelige programvare er **selvreproduserende**, som betyr at så fort det har infisert en vert, så vil den verten prøve å infisere andre verter. På denne måten kan selvreproduserende skadelig programvare spre seg eksponensielt fort. Skadelig programvare kan spre seg i form av virus eller worms.

**Viruser** er skadelig programvare som krever en form for interaksjon med brukeren for å infisere brukerens enhet. Et klassisk eksempel på dette er et epost-vedlegg som inneholder skadelig kjørbar kode.

**Worms** er skadelig programvare som kan gå inn i en enhet uten noen spesiell brukerinteraksjon. For eksempel at en bruker kan kjøre et sårbart nettverksprogram som en angriper kan sende skadelig programvare på.

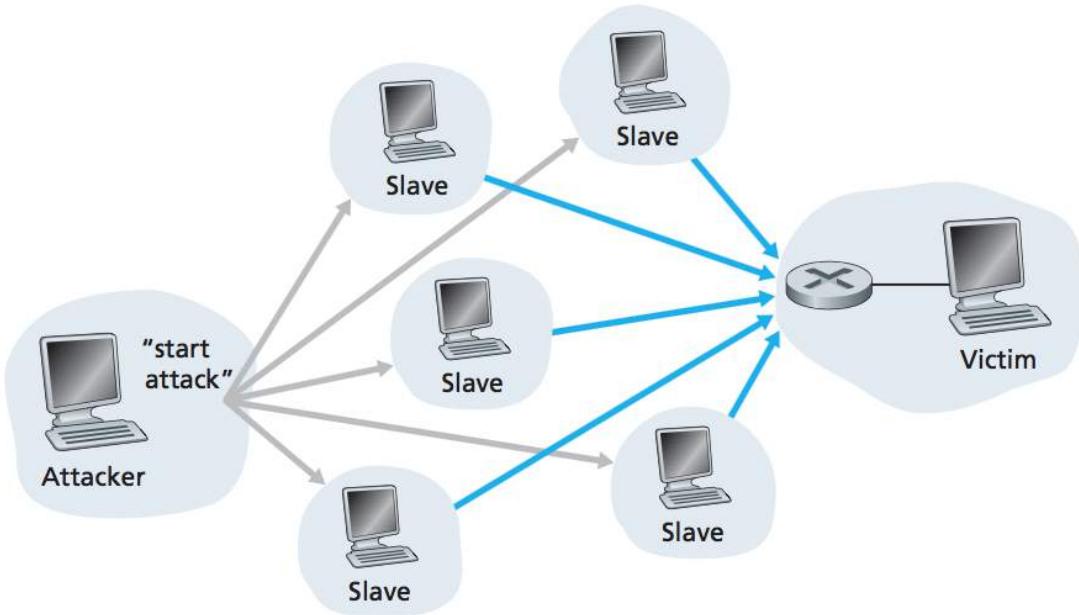
En annen sikkerhetstrussel er kjent som **denial-of-service (DoS) attacks**. Som navnet antyder, gjør et DoS-angrep et nettverk, vert eller annen infrastruktur ubruklig av legitime brukere.

De fleste Internett DoS-angrep faller inn i en av tre kategorier:

- **Sårbarhetsangrep.** Dette innebærer å sende noen velutviklede meldinger til et utfyllende program eller operativsystem som kjører på en målrettet vert. Hvis den riktige sekvensen av pakker sendes til et sårbart program eller operativsystem, kan tjenesten stoppe eller verre kan verten krasje.
- **Båndbreddeflom.** Angriperen sender en deluge av pakker til den målrettede verten, så mange pakker at målets tilgangslink blir tilstoppet, og hindrer at legitime pakker når serveren.

- *Tilkoblingsflom*. Angriperen etablerer et stort antall halvåpent eller fullt åpne TCP-tilkoblinger (TCP-tilkoblinger diskuteres i kapittel 3) på målverten. Verden kan bli så presset ned med disse falske forbindelsene at den slutter å godta legitime tilkoblinger.

I et distribuert DoS (DDoS) angrep, illustrert i figuren under, styrer angriperen flere kilder og lar hver kilde skyte trafikk på målet.



**Figure 1.25 ♦ A distributed denial-of-service attack**

En passiv mottaker kan lagre en kopi av hver pakke som flyr forbi, er kalt en **pakkesniffer**

Evnens til å injisere pakker til Internett med en falsk kildeadresse kalles IP-spoofing, og er bare en av mange måter som en bruker kan maskerer som en annen bruker.

## Kapittel 2 - Application Layer

### Principles of Network Applications

I en webapplikasjon er det to forskjellige programmer som kommuniserer med hverandre: nettleserprogrammet kjører i brukerens verts (skrivebord, bærbar PC, nettbrett, smarttelefon og så videre); og webserverprogrammet kjører i webserveren.

#### Arkitekturen til nettverkapplikasjoner

Applikasjonsarkitekturen er designet av applikasjonsutvikleren og dikterer hvordan applikasjonen er strukturert over flere endesystemer. Når man velger applikasjonsarkitekturen vil man gjerne velge mellom en av to dominerende arkitektske paradigmer brukt i moderne nettverksapplikasjoner: *the client-server architecture* eller *peer-to-peer (P2P) architecture*.

- **Klient-server arkitektur:** I denne arkitekturen er det en alltid-på vert, kalt serveren som gir tjenestegjør forespørsler fra mange andre verter, kalt *klienter*.
  - Et klassisk eksempel er webapplikasjonen som en kontinuerlig webserver tjenester forespørsler fra nettlesere som kjører på klientverter.

- **Peer-to-peer arkitektur:** I en P2P-arkitektur er det minimal (eller ingen) avhengighet av dedikerte servere i *datasentre*. I stedet utnytter programmet direkte kommunikasjon mellom par av periodisk tilkoblede verter, kalt *peers*. Klientene eies ikke av tjenesteleverandøren, men er i stedet desktops og bærbare datamaskiner kontrollert av brukerne, med de fleste kolleger bosatt i boliger, universiteter og kontorer. Fordi peer-ene kommuniserer uten å passere gjennom en dedikert server, kalles arkitekturen peer-to-peer. Mange av dagens mest populære og trafikkintensive applikasjoner er basert på P2P-arkitekturer.

- En av de mest overbevisende funksjonene i P2P-arkitektur er deres **selvskalering**. For eksempel, i et P2P-fildelingsprogram, selv om hver peer genererer arbeidsbelastning ved å be om filer, legger hver peer også tjenestekapasitet til systemet ved å distribuere filer til andre peers.

Men fremtidige P2P-applikasjoner står overfor tre store utfordringer:

1. *ISP Friendl/y*. De fleste boliger (inkludert DSL- og kabelleverandører) er dimensjonert for "asymmetrisk" båndbreddebruk, det vil si for mye nedstrøms enn oppstrøms trafikk. Men P2P-videoostreaming- og fildistribusjonsapplikasjoner skifter oppstrøms trafikk fra servere til boliger, og legger dermed betydelig vekt på Internett-leverandørene.
2. *Sikkerhet*. På grunn av deres svært distribuerte og åpne natur kan P2P-applikasjoner være en utfordring å sikre
3. *Insentiver*. Suksessen til fremtidige P2P-applikasjoner avhenger også av overbevisende brukere til å frivillig gi båndbrede, lagrings- og beregningsressurser til applikasjonene, som er utfordringen med insentivdesign (Insentiv = noe som motiverer noen til å utføre en bestemt handling)

### Klient- og server-prosesser

Vi sier at det ikke er programmer, men prosesser som kommuniserer. For eksempel i en webapplikasjon vil en klient-prosess utvekle meldinger med webserver-prosessen. For hvert par av kommuniserende prosesser, merker vi vanligvis en av de to prosessene som **klienten** og den andre prosessen som **serveren**. Vi definerer klient- og server-prosesser som følgende:

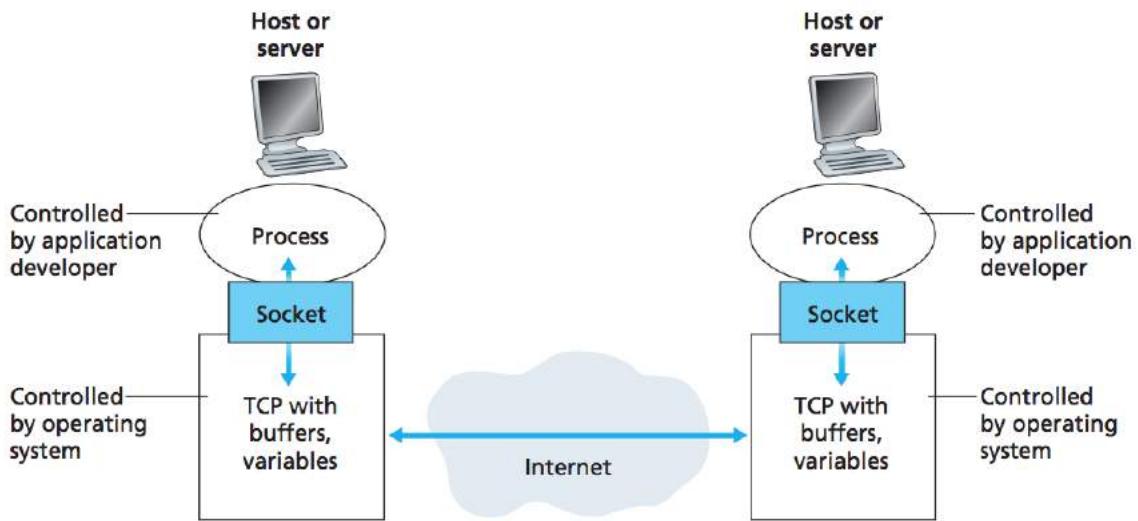
- I sammenheng med en kommunikasjonsøkt mellom et par prosesser merkes prosessen som initierer kommunikasjonen (det vil si i utgangspunktet kontakten med den andre prosessen i begynnelsen av økten) merket som *klienten*.
- Prosessen som venter på å bli kontaktet for å starte økten er *serveren*.

### Grensesnittet mellom prosessen og datanettverket

En prosess sender meldinger inn i, og mottar meldinger fra, nettverket gjennom et programgrensesnitt som kalles en **socket**.

- Dersom vi ser på en prosess som et hus, vil socketen være døren.

En socket er grensesnittet mellom applikasjonslaget og transportlaget innad en vert. Det refereres også som **Application Programming Inter-face (API)** mellom applikasjonen og nettverket.



**Figure 2.3** ♦ Application processes, sockets, and underlying transport protocol

### Addresseringsprosesser

For at en prosess kjørende på en vert skal kunne sende pakker til en annen prosess som kjører på den andre verten, så trenger den mottakende prosessen en adresse. For å identifisere mottakende prosesser, må to typer informasjon være spesifisert:

1. Adressen til verten
2. En identifikator som spesifiserer den mottakende prosessen i destinasjonsverten

I internettet er en vert gjenkjent ved dens **IP adresse**. I tillegg til vite adressen til verten, må den sendende prosessen også kunne identifisere den mottakende prosessen (mer spesifikt, den mottakende socket-en). Et destinasjons **portnummer** tjener denne hensikten.

Populære applikasjoner har fått tidelt spesifikke portnummer. For eksempel er en webserver identifisert med portnummer 80, en mailserver-prosess (SMTP) har portnummer 25, og resten av listen finnes [her](#).

### Transporttjenester tilgjengelige for applikasjoner

Minnes om at en socket er grensesnittet mellom applikasjonsprosessen og transportlagsprotokollen. Hva er tjenestene som en transportlagsprotokoll kan tilby for applikasjoner som påberoper det? Vi kan i stor grad klassifisere de mulige tjenestene langs fire dimensjoner: pålitelig dataoverføring, gjennomstrømning, timing og sikkerhet.

#### Pålitelig dataoverføring:

Dersom en protokoll kan garantere at data som blir sendt fra en ende vil bli levert korrekt og komplett til den andre enden av applikasjonen, sier vi at den tilbyr **pålitelig overføring**.

Når en transportlagprotokoll ikke gir pålitelig dataoverføring, kan noen av dataene som sendes av den sendende prosessen aldri komme til den mottakende prosessen. Dette kan være akseptabelt for **tapstolerante applikasjoner**, særlig multimedieapplikasjoner som konversasjonslyd/video som kan tåle noe tap av data.

#### Gjennomstrømning:

Programmer som har gjennomstrømningskrav, sies å være **båndbreddefølsomme** applikasjoner. Mange nåværende multimedieapplikasjoner er følsomme for båndbrede, selv om enkelte multimedieapplikasjoner kan bruke adaptive kodeteknikker for å kode digitalisert stemme eller video med en hastighet som samsvarer med den nåværende tilgjengelige gjennomføringen.

Selv om båndbreddefølsomme applikasjoner har spesifikke krav til gjennomstrømning, kan **elastiske applikasjoner** benytte så mye eller lite gjennomstrømning som tilfeldigvis er tilgjengelig. Dette kan være elektronisk mail, filoverføring o.l.

**Timing:** En transportlagsprotokoll kan også gi tidsgarantier. Som med gjennomstrømningsgarantier kan tidsgarantier komme i mange former og former. Et eksempel på garanti kan være at hver bit som avsenderen pumper inn i socket-en, kommer til mottakerens socket, ikke mer enn 100 ms senere.

**Sikkerhet:** En transportprotokoll kan gi et program med en eller flere sikkerhetstjenester. Eksempelvis kan en transportprotokoll kryptere alle data som sendes av sendeprosessen, i mottakeren, og transportlagsprotokollen dekrypterer dataene før de leverer dataene til mottaksprosessen. En slik tjeneste vil gi konfidensialitet mellom de to prosessene, selv om dataene på en eller annen måte observeres mellom sende- og mottaksprosesser.

## TCP Services

Når en applikasjon bruker TCP som sin transportprotokoll, vil applikasjonen få begge tjenestene fra TCP:

- *Connection-oriented service:* TCP får klienten og serveren til å utveksle transportlagskontroll informasjon med hverandre før applikasjonslagsmeldingene kan begynne å flyte. Dette er den såkalte handshaking-prosedyren, som lar klientene og serverene til å bli klare til å sende/motta pakker.
- *Reliable data transfer service:* De kommuniserende prosessene kan vite at TCP levererer all data som blir sendt uten noen feil og i riktig rekkefølge.

TCP inkluderer også en opphopningskontroll-mekanisme, en tjeneste som endrer hastigheten på en sendingsprosess når nettverket er mett mellom sender og mottaker.

## UDP Services

UDP er en *no-frills*, lett nettverksprotokoll, som tilbyr minimalt med tjenester. UDP er *connectionless*, så det er ingen handshaking før de to prosessene starter å kommunisere.

- UDP tilbyr en unreliable data service, det betyr at det ikke er noen garanti på at pakker kommer frem.
- UDP tilbyr ingen opphopningskontroll-mekanisme.

## Application-Layer Protocols

En applikasjonslagsprotokoll definerer:

- Typen meldinger som blir utvekslet, for eksempel request-meldinger og response-meldinger
- Syntaxen til de varierende meldingstypene, sånn som hvilke felt som er i meldinger og hvordan feltene er avgrenset.
- Semantikken til feletene, som betyr meningen til informasjonen i feltene
- Reglene for å bestemme når og hvordan en prosess skal sende og respondere til meldinger

## The Web and HTTP

Protokollen **HyperText Transfer Protocol (HTTP)** er hjertet til internettet. HTTP er implementert i to programmer: et klientprogram og et serverprogram. Programmene kjører på forskjellige endesystemer, snakker med hverandre ved å utveksle HTTP-meldinger. HTTP definerer strukturen til disse meldingene og hvordan klienten og serveren utveksler disse meldingene.

En **nettside (document)** består av objekter. Et **objekt** er en fil, som en HTML-fil, et JPEG bilde, som kan addresses til gjennom en enkelt URL. De fleste nettsider består av en **base HTML fil** og flere refererte objekter.

Når en bruker *requester* om en nettside sender browseren en HTTP request-meldinger for objektene på siden til serveren. Serveren mottar disse request-meldingene, og responderer med HTTP response-meldinger som inneholder objektene.

HTTP bruker **TCP** som sin underliggende transportprotokoll. HTTP-klienten initierer først en TCP-tilkobling til serveren. Når tilkoblingen er etablert, vil browser og server prosessene aksessere TCP gjennom socket-interfacet sitt.

- Som beskrevet over tilbyr TCP pålitelig dataoverføring til HTTP. Dette impliserer at hver HTTP request-melding sendt av en klient prosess kommer intakt ved serveren, på samme måte vil hver HTTP response-melding etterhvert komme intakt hos klienten.
- Det er viktig å merke seg at serverene ikke lagrer noen informasjon om klientene. På grunn av at en HTTP server ikke vedlikeholder noen informasjon om sine klienter, er HTTP en **stateless protocol**.

## Non-Persistent and Persistent Connections

- **Persistent (norsk. vedvarende)**

Når en klient-server interaksjon finner sted over TCP, må applikasjonen ta en viktig avgjørelse, skal hvert request/response-par sendes over en *separat* TCP-forbindelse, eller skal hver av disse requestene og deres korresponderende responser bli sendt over *samme* TCP-forbindelse.

En applikasjon er dermed sagt å være **ikke-vedvarende tilkoblinger** eller motsatt, **vedvarende tilkoblinger**.

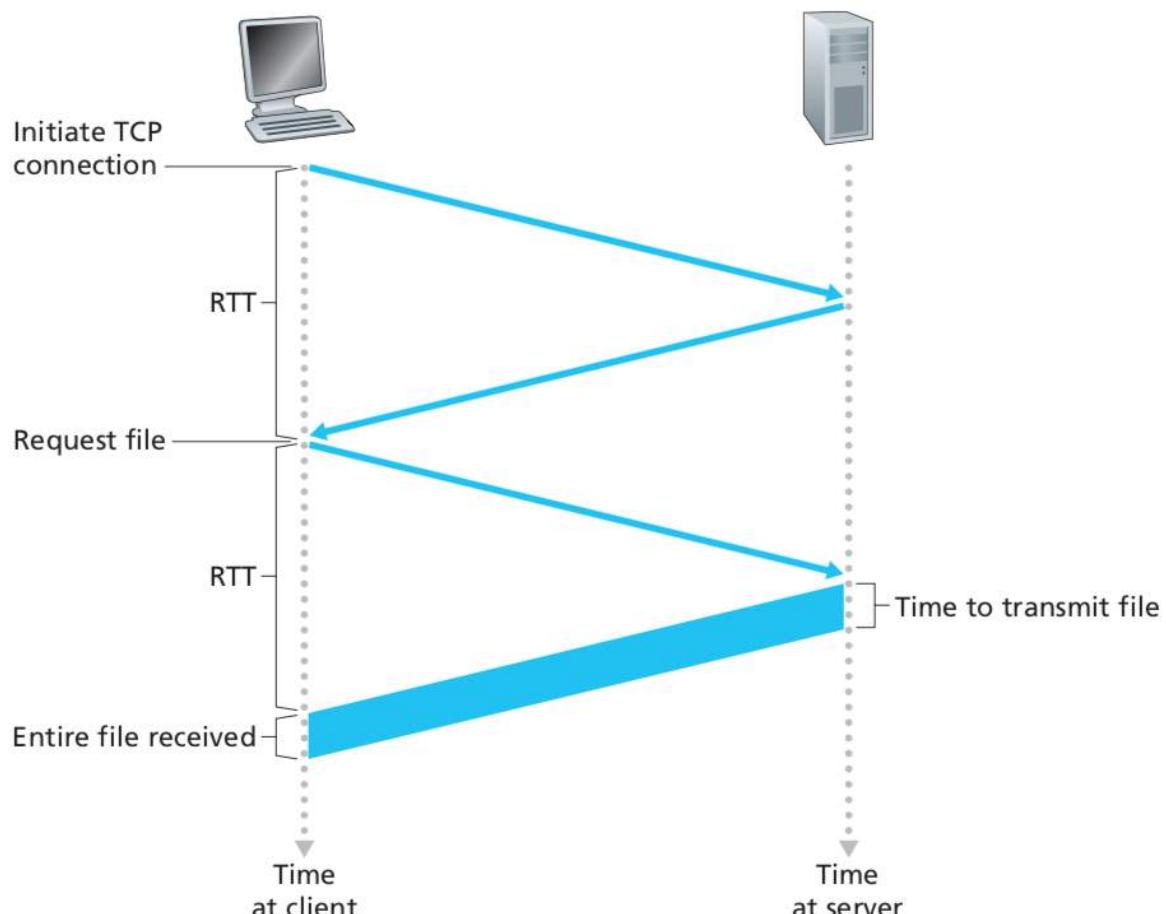
### HTTP with Non-Persistent Connections

La oss anta at en nettside innholder en HTML-fil og 10 JPEG bilder, og at alle disse 11 elementene ligger på samme server. Dette vil da skje:

1. HTTP-klienten vil initiere en TCP-tilkobling med serveren.
2. HTTP-klienten sender en HTTP request-melding til serveren.
3. HTTP-server-prosessen sender en HTTP response-melding til klienten.
4. HTTP-server-prosessen forteller TCP å lukke TCP-tilkoblingen når meldingen kommer frem.
5. HTTP-klienten mottar response-meldingen. TCP-tilkoblingen avsluttes. Klienten leser HTML-filen og finner referansene til de 10 JPEG-objektene.
6. De fire første stegene blir så gjentatt for hvert JPEG-objekt.

Før vi fortsetter, la oss estimere hvor lang tid det tar fra en klient *requester* HTML-basefilen til hele filen er mottatt av klienten. Som sagt, definerer vi **round-trip time (RTT)** til å være tiden det tar for en pakke å reise fra en klient til server og tilbake til klienten.

Som vist i figuren under tvinger dette nettleseren til å initiere en TCP-tilkobling mellom nettleseren og Web-serveren; som involverer en "three-way handshake" - klienten sender et TCP-segment til serveren, serveren anerkjenner (eng. *acknowledges*) og responderer med et TCP-segment, og til slutt, klienten anerkjenner tilbake til serveren.



- **Første og andre del av "three-way handshake"** tar en RTT. Etter dette sender klienten en HTTP request-melding kombinert med den **tredje** del av håndhilsingen (anerkjennelsen) inn i TCP-tilkoblingen.

- Når request-meldingen kommer til serveren, sender serveren HTML-filen inn i TCP-tilkoblingen. Denne HTTP request/respons bruker enda en RTT.

#### HTTP with Persistent Connections

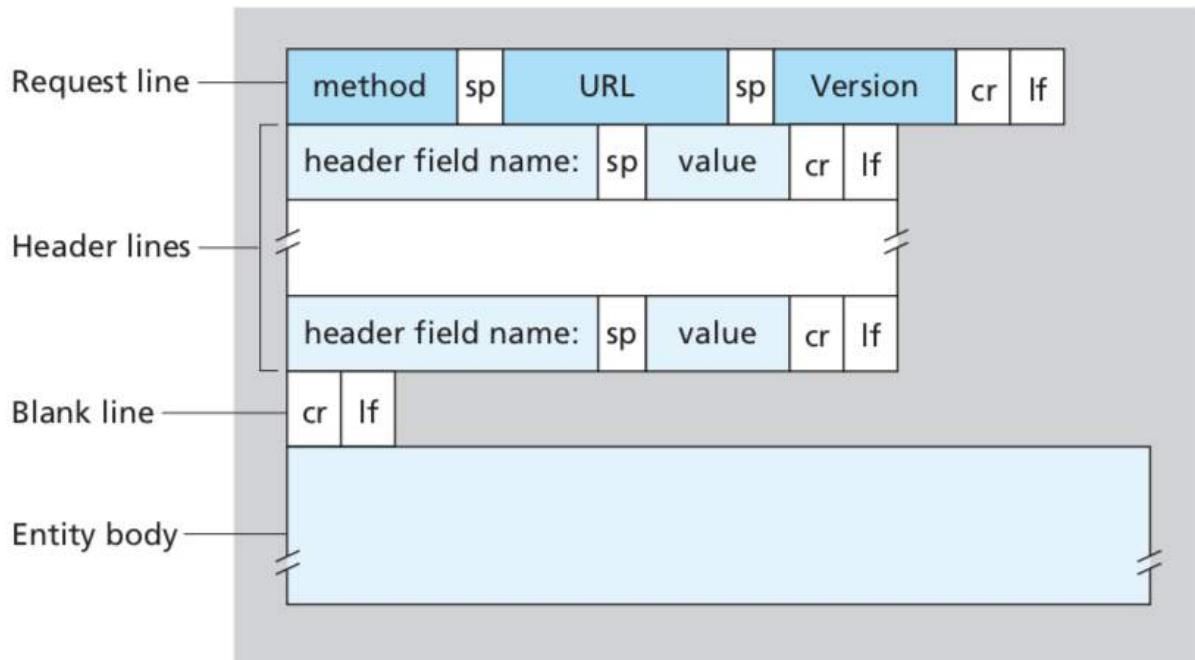
Ikke-vedvarende tilkoblingen kan ha noen mangler. For det første, en helt ny tilkobling må bli etablert og vedlikeholdt for *hvert forespurte objekt*. For hver av disse tilkoblingene må det tildeles TCP-buffere og -variabler. Dette kan være en byrde for Web-serveren, som muligens gjør slike forespørsler for flere hundre klienter samtidig. Som beskrevet vil hvert objekt få leveringsforsinkelse på to RTT-er.

Med vedvarende tilkoblinger kan serveren la TCP-tilkoblingen være åpen etter den har sendt en respons. Senere requester og responser mellom samme klient og server kan bli sendt over samme tilkobling.

#### HTTP Message Format

##### HTTP Request Message

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/5.0
Accept-language: fr
```

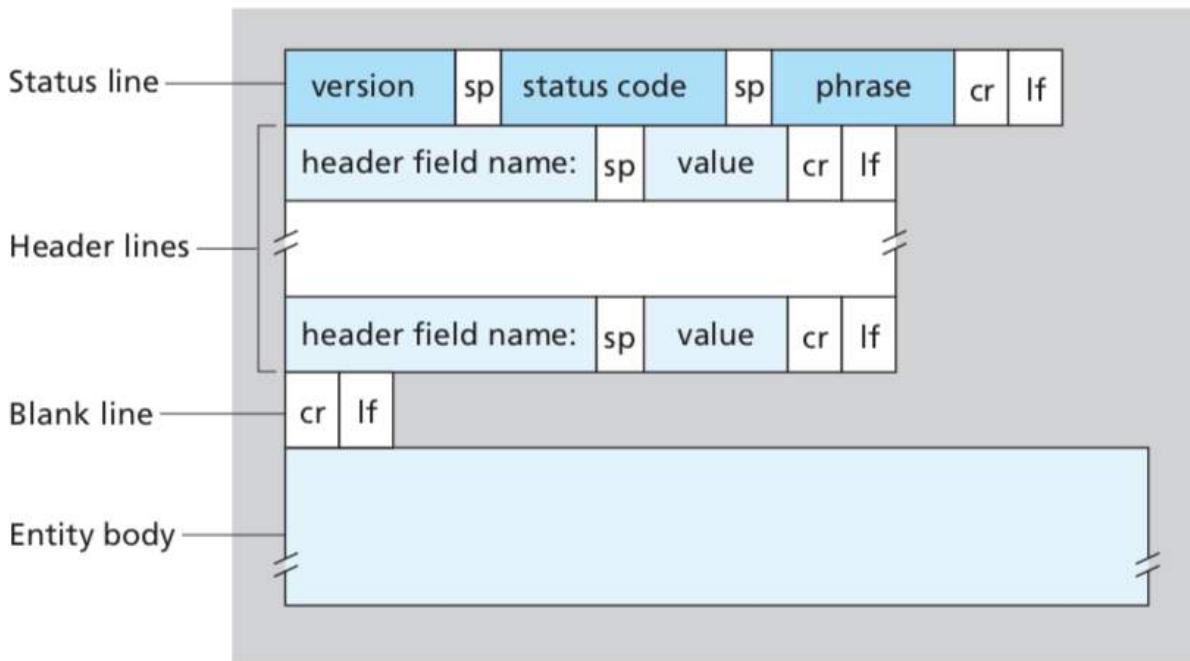


- Første linje i meldingen kalles en **forespørsel-linje** (eng. *request line*), de påfølgende linjene er kalt **header-linjer**. Requestlinjen har tre felt: *metodefelt*, *URL-felt* og *HTTP-versjon-felt*.
  - Metodefeltet kan ta inn forskjellige verdier, inkludert `GET` , `POST` , `HEAD` , `PUT` , og `DELETE` .
  - Headerlinjen `Host:` spesifiserer hosten hvor objektet ligger. Ved å inkludere `Connection: close` i headerlinjen, forteller browseren serveren at den ikke trenger en vedvarende tilkobling (persistent).
  - Headerlinjen `User agent:` spesifiserer brukeragenten, som er browsertypen som gjør forespørselen på serveren.
  - Entitets-bodyen til en `GET`-request er tom, men blir brukt med `POST`-metoden.
- `HEAD` -metoden er lignende `GET` -metoden. Når en server mottar en `HEAD`-metode, responderer den med en HTTP-melding, men uten det forespurte objektet.
  - Applikasjonsutviklere bruker ofte `HEAD`-metoden for debugging eller testing
- `PUT` -metoden blir ofte brukt til å laste opp et objekt på en spesifik path (dir) på en spesifik webserver.
- `DELETE` -metoden tillater en bruker, eller en applikasjon, til å slette et objekt på en webserver.

## HTTP Response Message

```
HTTP/1.1 200 OK
Connection: close
Date: Tue, 09 Aug 2011 15:44:04 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 09 Aug 2011 15:11:03 GMT Content-Length: 6821
Content-Type: text/html

(data data data data data ...)
```



- Den består av tre deler: en **statuslinje**, seks **header-linjer** og **entitets-bodyen**.
- Entitetskroppen/bodyen er kjøttet til meldingen - den inneholder det forespurte objektet selv, representert med  
(data data data data ...)
  - Statuslinjen* har tre felt: protokollen, versjonskoden og en korresponderende statusmelding.
  - Headerlinjen `Connection: close` forteller klienten at den kommer til å lukke TCP-tilkoblingen etter sendelsen av meldingen. `Date:` -headeren forteller tiden og datoan HTTP-responsen ble laget og sent av serveren. `Server:` -headerlinjen indikerer hva slags server som har generert meldingen. `Last-modified:` -feltet forteller når objektet ble laget eller sist oppdatert. `Content-type:` -feltet forteller oss hva slags filtype objektet er.

### Kjente statuskoder:

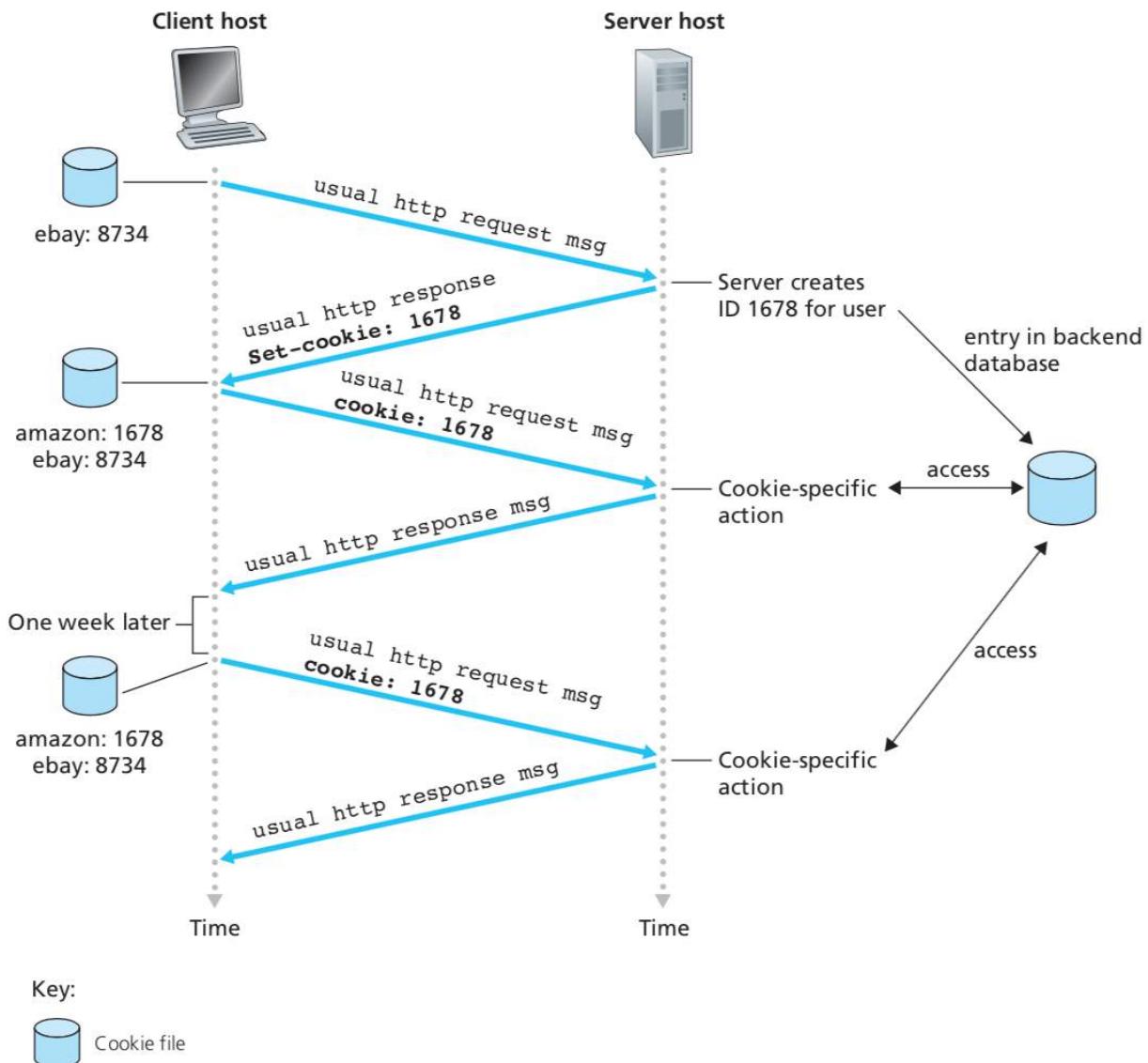
- 200 OK : Request suksessfull og informasjon returnert i respons
- 301 Moved Permanently : Forespurt objekt er flyttet til en ny URL, spesifisert i `Location:` -feltet i responsen
- 400 Bad Request : Denne responsen kommer når serveren ikke forstod request-meldingen
- 404 Not Found : Det forespørte dokumentet eksisterer ikke på serveren
- 505 HTTP Version Not Supported : Den forespørte HTTP-protokollversjonen støttes ikke av serveren

### User-Server Interaction: Cookies

Vi nevnte at en HTTP-server er tilstandsløs. En server kan håndtere tusen av samtidige TCP-tilkoblinger. Men ofte er det ønsket at en ønsker å kunne identifisere brukere, enten fordi serveren ønsker å begrense brukertilgang eller fordi den ønsker å vise innhold som en funksjon av brukeridentiteten. For disse formålene bruker HTTP Cookies (*norsk informasjonskapsler*). Cookies tillater nettsteder å holde oversikt over brukere. De fleste store kommersielle nettsteder bruker informasjonskapsler i dag.

Som vist i figuren under har informasjonskapselteknologien fire komponenter (1) en cookie header.linje i HTTP response-meldingen, (2) en cookie headerlinje i HTTP request-meldingen, (3) en cookie-fil lagret på brukerens endesystem og styrt av brukerens nettleser, og (4) en backend-database på nettstedet.

- HTTP request-meldingen har headerfeltet `Cookie:` og HTTP response-meldingen har `Set-cookie:`-headerfeltet.

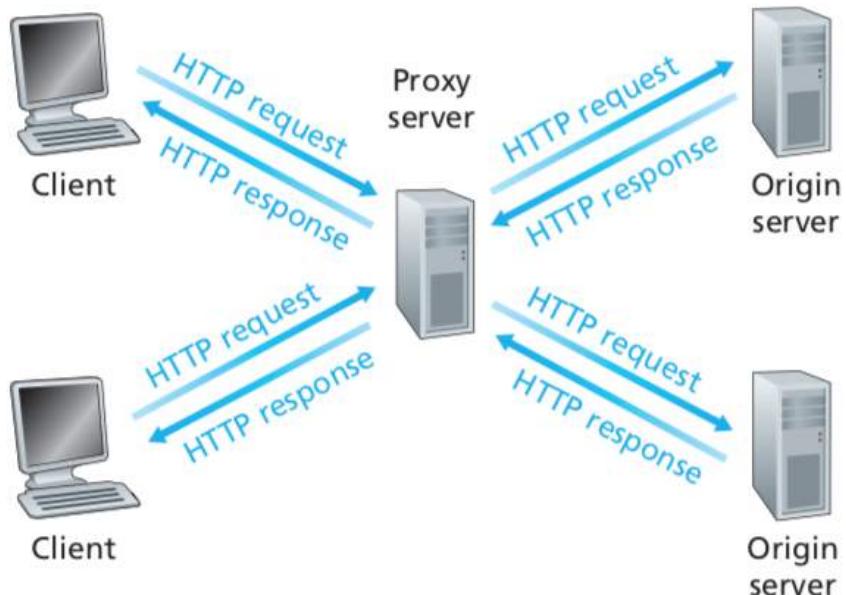


**Figure 2.10 ♦ Keeping user state with cookies**

Som man ser i figuren over, så vil cookien bli i lengre tid, og når man etter en uke går tilbake til samme sted vil cookien man fikk tildelt sist bestemme hvordan respons man får av serveren.

### Web Caching

En **web cache** - også kalt en **proxy server** - (norsk. *nettbuffer*) er en nettverksentitet som tilfredsstiller HTTP forespørsler på vegne av en webserver. Nettbufferen har en egen lagringseenhet og holder kopier av nylig forespurte objekter. Som ivst i figuren under kan en brukers nettleser konfigureres slik at alle nettleser-forespørsler rettes mot nettbufferen.



**Figure 2.11 ♦ Clients requesting objects through a Web cache**

Dette skjer:

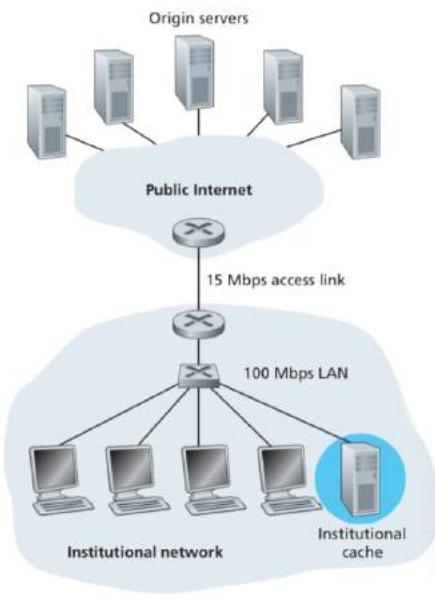
1. Nettleseren lager en TCP-tilkobling til nettbufferen og sender den en HTTP-request for objektet på nettbufferen.
2. Nettbufferen sjekker om den har en kopi lagret lokalt. Dersom den har det returnerer den objektet med en HTTP-response til klient-nettleseren.
3. Dersom nettbufferen *ikke* er objektet, åpner nettbufferen en TCP-tilkobling til den originale serveren. Nettbufferen forespør så om objektet, nettserveren sender objektet med en HTTP-response til nettbufferen
4. Når nettbufferen får objektet, lagrer den en kopi lokalt og sender en kopi, i en HTTP response-melding, til klient-nettleseren (over den eksisterende TCP-tilkoblingen)

Typisk er en nettbuffer kjøpt og installert av en nettverksleverandør

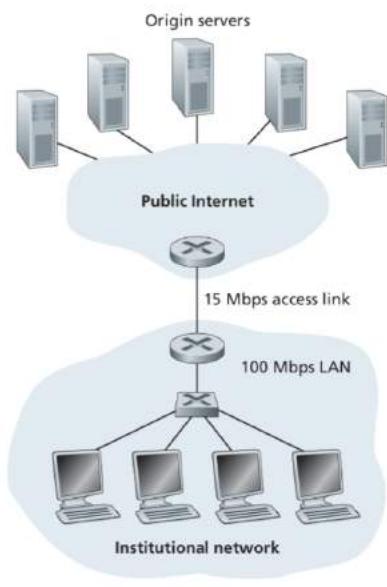
Nettbuffring blir implementert på Internettet av to grunner:

- For det **første** kan en nettbuffer redusere responstiden for en klientforespørsel vesentlig, spesielt hvis flaskehalsbåndbredden mellom klienten og opprinnelsesserveren er mye mindre enn bottenhalsbåndbredden mellom klienten og hurtigbufferen.
- For det **andre**, som snart illustreres med et eksempel, kan web-caches betydelig redusere trafikken på institusjonens tilgangskoblingen til internettet. Ved å redusere trafikken trenger institusjonen (for eksempel et firma eller et universitet) ikke å oppgradere båndbredden så fort, og dermed redusere kostnadene.

Videre kan nettbuffere betydelig redusere webtrafikk på Internett som helhet, og dermed forbedre ytelsen for alle applikasjoner.



**Figure 2.13** ♦ Adding a cache to the institutional network



**Figure 2.12** ♦ Bottleneck between an institutional network and the Internet

Anta at den gjennomsnittlige objektstørrelsen er 1 Mbits, og at gjennomsnittlig forespørselshastighet fra institusjonens nettleseere til opprinnelsesserverne er 15 forespørsler per sekund.

Trafikkintensiteten på LAN-et er

- $(15 \text{ requests / s}) * (1 \text{ Mbits / requests}) / (100 \text{ Mbps}) = 0.15$

mens trafikkintensiteten på tilgangslinken (fra Internett-ruteren til institusjonen-ruteren) er

- $(15 \text{ forespørsler / s}) (1 \text{ Mbits / requests}) / (15 \text{ Mbps}) = 1$

Denne typen intensitet vil gi veldig stor forsinkelse på koblingen, og kan være opp mot minutter. Dersom man legger til en nettbuffer, og antar at den har 40% hitratio, vil gjennomsnittlig forsinkelse være:

- $0.4 * (0.01 \text{ s}) + 0.6 * (2.01 \text{ s}) \approx 1.2 \text{ s}$

Som vil er en kjempeforbedring enn det det var. Også mye billigere enn å øke aksesskoblingen mot internettet.

#### The Conditionally GET

Selvom buffring kan redusere bruker-oppfattede responstider, introduserer den et nytt problem - kopien av et objekt som ligger i hurtigbufferen, kan være foreldet. Med andre ord kan objektet lagret i bufferen være endret siden kopien ble cachet ved klienten.

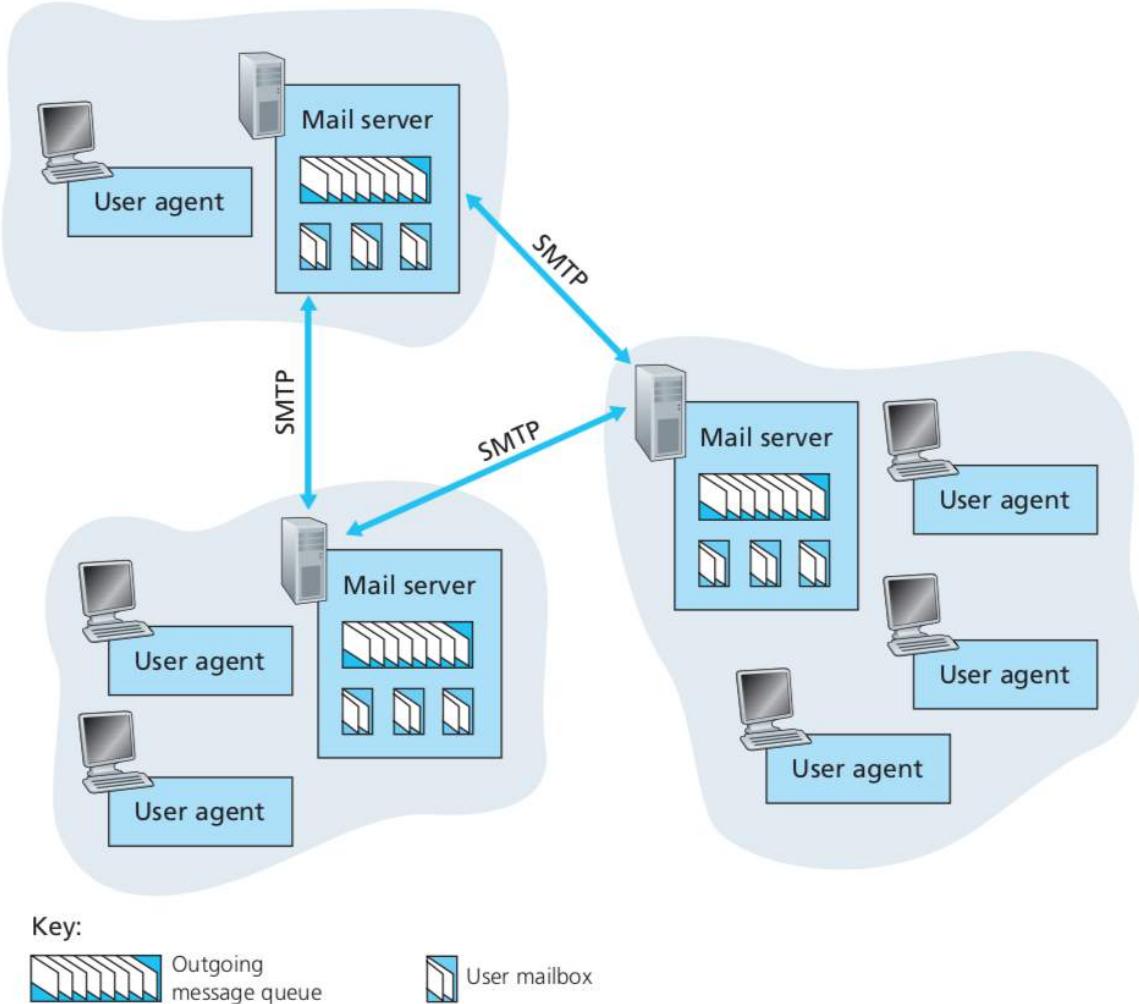
Heldigvis har HTTP en mekanisme som lar en nettbuffer verifisere at objektene sine er oppdaterte. Denne mekanismen kalles **conditional GET** (norsk. *betinget*). En HTTP request-melding er en såkalt betingen GET melding hvis (1) request-meldingen bruker GET-metoden og (2) request-meldingen inkluderer en *If-Modified-Since*: -headerlinje.

- Anta at en klient forespør et objekt og nettbufferen henter dette fra en webserver, og lagrer en kopi i bufferen sin. Nettbufferen lagrer dette objektet sammen med headerlinjen *Last-Modified*: .
- En uke senere kommer en ny klient og ber om samme objekt. Siden objektet kan være endret den siste uken, så gjør nettbufferen en *betinget GET-request* til serveren, og får vite om filen er endret eller ikke (304 Not Modified-status).

#### Electronic Mail in the Internet

Som vanlig post er eposter en asynkront kommunikasjonsmedium - folk svarer og leser meldinger når det passer dem. I motsetning til vanlig post er epost raskt, enkelt å distribuere og billig. Man kan legge til vedlegg, hyperlinker og HTML-formatert tekst.

Vi skal å se på applikasjonslag protokollene som er hjertet av internettets eposter.

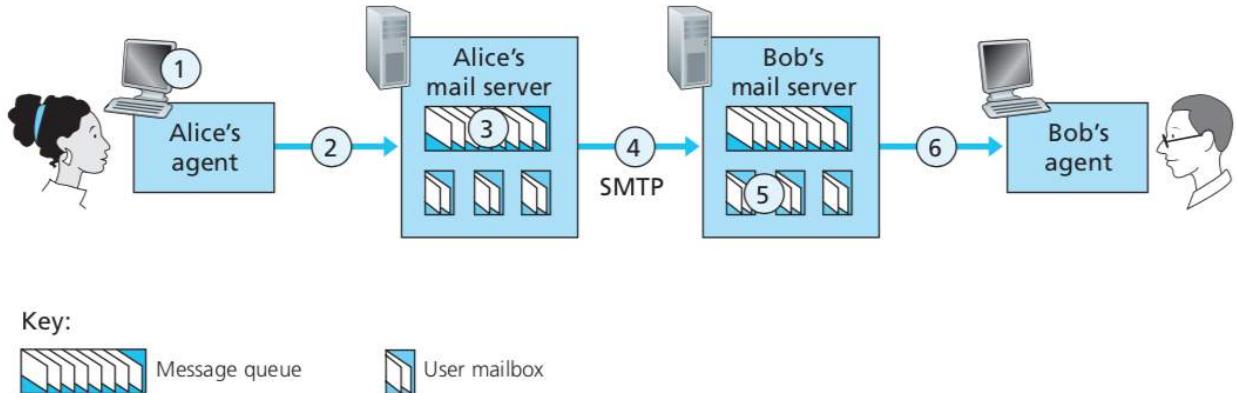


**Figure 2.16** ♦ A high-level view of the Internet e-mail system

Denne figuren inneholder mange komponenter som **brukeragenter**, **mailservere**, og **SMTP-protokollen** (eng. *Simple Mail Transfer Protocol*). Christian sender en epost til en mottaker Bob. brukeragentene lar brukere lese, svare, videresende, lagre meldinger. Apple Mail og Outlook er eksempler på brukeragenter for epost. Når Christian er ferdig med å skrive eposten sin sender brukeragenten hans meldingen til mailserveren hans, hvor meldingen blir plassert i mailserverens utgående meldingskø. Når Bob ønsker å lese meldinger vil lese meldingen, henter brukeragenten hans meldingen fra mailboksen i mailserveren hans.

SMTP er hoved applikasjonsprotokollen for elektronisk post. Den bruker den pålitelige dataoverføringsprotokollen TCP.

- Når en mailserver sender en mail til en annen mailserver, oppfører den seg som en SMTP-klient.
- Når en mailserver mottar en mail fra en annen mailserver, oppfører den seg som en SMTP-server.



**Figure 2.17 ♦ Alice sends a message to Bob**

1. Alice bruker brukeragenten hennes for epost, oppgir Bobs epost-adresse, skriver en melding og ber agenten sende eposten.
2. Alice sin brukeragent sender meldingen hennes til mailserveren hennes, og plasserer den i meldingskøen.
3. Klientsiden av SMTP, som kjører på Alice sin mailserver, ser meldingen i køen. Åpner en TCP-tilkobling til en SMTP-server, kjørende på Bob sin mail server.
4. Etter SMTP-handshaking, sender SMTP-klienten Alice sin melding gjennom TCP-tilkoblingen.
5. Ved Bob sin mailserver, vil server-siden av SMTP motta meldingen. Mailserveren til Bob lagrer så meldingen i mailboksen til Bob.
6. Bob spør så brukeragenten sin om å lese meldingen når han ønsker.

SMTP-klienten etablerer en TCP-tilkobling på port 25 hos SMTP-serveren. Når TCP-tilkoblingen er etablert, vil serveren og klienten utføre noen applikasjonslag-handshaking. SMTP-klientene og serverene introduserer hverandre før de sender informasjon.

#### *Handshaking:*

Server (S - Hostname: `hamburger.edu`)

Client (C - Hostname: `crepes.fr`)

```

S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr ... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection

```

#### Sammenligning med HTTP

Begge protokoller brukes til å overføre filer fra et endesystem til en annet. HTTP overfører filer (kalt objekter) fra en webserver til en webklient (typisk nettleser). SMTP overfører filer (som er epost-meldinger) fra en mailserver til en annen mailserver.

Når filene overføres bruker begge protokollene vedvarende/persistent tilkoblinger.

HTTP er i hovedsak en **pull protocol** - noen vil ha informasjon fra en nettserver og bruker HTTP for å hente informasjonen ved serveren etter deres ønske. Spesielt, TCP-tilkoblingen blir initialisert på maskinen som ønsker filen.

SMTP er på den andre siden en **push protocol** – den sendende mailserveren sender dytter filen til den mottakende mailserveren. Spesielt, TCP-tilkoblingen blir initialisert på maskinen som ønsker å sende filen.

En annen forskjell er at SMTP krever at hver melding, inkludert kroppen til hver melding, skal være i 7-biters ASCII-format. Hvis meldingen inneholder tegn som ikke er 7-biters ASCII (for eksempel franske tegn med aksenter) eller inneholder binære data (for eksempel en bildefil), må meldingen kodes inn i 7-biters ASCII. HTTP-data pålegger ikke denne begrensningen.

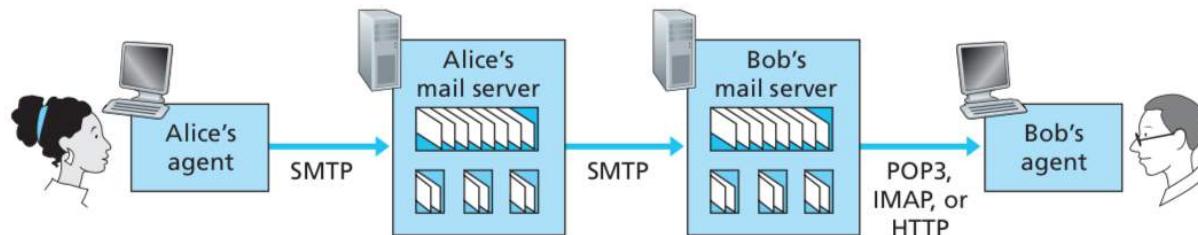
HTTP innkapsler hvert objekt i sin egen HTTP-svarmelding. Internett-post plasserer alle meldingsobjektene i en melding.

#### Mail-format

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Searching for the meaning of life.

(melding)
```

#### Mail-aksess protokoller



**Figure 2.18** ♦ E-mail protocols and their communicating entities

#### POP3 - Post Office Protocol v. 3

- Enkel epost-aksessprotokoll, derav liten funksjonalitet.
- OP3 begynner når en brukeragent (klienten) åpner en TCP-tilkobling til mailserveren (serveren) på port 110. Med TCP-tilkoblingen etabler begynner POP3 å gå igjennom tre faser: autorisering, transaksjon og oppdatering.
  - i. Autorisering: Brukeragent sender brukernavn og passord, for å autentisere brukeren. `user <username> og pass <password>`
  - ii. Transaksjon: Brukeragent bruker kommandoene `list`, `retr`, `dele` og `quit` for å få listet alle mailenes størrelser, retrievet innholdet i mail og slettet mail.
  - iii. Oppdatering: Etter å ha behandlet `quit`-kommandoen, går POP3-serveren i oppdateringsfasen og fjerner meldinger fra postkassen.
- Under POP3-sessionen mellom en brukeragent og mailserveren, lagrer POP3-serveren noen tilstander, spesielt hvilke eposter som har blitt markert som slettet. Men lagrer det ikke til senere sessions.

#### IMAP - Internet Message Access Protocol

Med POP3-tilgang så kan Bob slette, lagre, markere og organisere mailene sine i mapper og strukturer på sin lokale maskin. Men dersom han ønsker å kunne aksessere denne strukturen på enhver maskin, må denne organiseringen lagres på en ekstern server. Dette kan man med IMAP-protokollen.

En IMAP-server vil assosiere en melding med en mappe, når en mail først innkommer er den assosiert med INBOX-mapen. IMAP-protokollen tillater en bruker å lage mapper, og flytte eposter til forskjellige mapper.

#### Web-Based E-Mail

Flere og flere begynner å sende og aksessere epostene sine gjennom nettleseprogrammene. Med en slik tjeneste er brukeragenten en vanlig nettleseprogram som kommuniserer med den eksterne mailboksen gjennom HTTP.

Nettleseren bruker HTTP for å snakke med mailserveren istedet for POP3- eller IMAP-protokollen

## DNS - The Internet's Directory Service

- En internett vert er identifisert med `hostname`, som `cnn.com` eller `www.yahoo.com`, også IP-adresser, som `121.7.106.83`

### Tjenester tilbuddt av DNS

Vi har nettopp sett at det er to måter å identifisere en vert ved, med vertsnavn og en IP-adresse. Folk foretrekker den mer lettleselige vertsnavnidentifikatoren, mens rutere foretrekker IP-adresser. For å forene disse innstillingene trenger vi en katalogtjeneste som oversetter vertsnavn til IP-adresser. Dette er hovedoppgaven til internettets **domain name system (DNS)**.

DNS-en er (1) en distribuert database implementert i et hierarki av DNS-servere, og (2) en applikasjonslagsprotokoll som tillater verter å sende spørreningar til den distribuerte databasen

- DNS brukes vanligvis av andre protokoller for applikasjonslag, inkludert HTTP og SMTP, for å oversette brukerleverte vertsnavn til IP-adresser.

Dette skjer dersom en nettleser ber om URL-en `www.someschool.edu/index.html`:

1. Samme brukermaskin kjører klientsiden til DNS-applikasjonen
  2. Nettleseren henter vertsnavnet, `www.someschool.edu`, fra URL-en og gir det til klientsiden av DNS-applikasjonen.
  3. DNS-klienten sender en spørring med vertsnavnet til DNS-applikasjonen
  4. DNS-klienten mottar etterhvert en respons, inkludert IP-adressen for vertsnavnet
  5. Når nettleseren får IP-adressen fra DNS, kan den initialisere en TCP-tilkobling til HTTP-server-prosessen på port 80 på den IP-adressen.
- **Vertsnavn aliaser:** Verter med kompliserte navn kan ha vertsnavnaliaser for å gjøre vertsnavnene lettere for mennesker å lese og huske. Da må nettleseren bruke DNS for å få hentet det kanoniske (lange og kompliserte) vertsnavnet sammen med den korresponderende IP-addresen
  - **Mailserver aliaser:** Nettsider med stor trafikk er ofte replikert over flere servere, der hver server kjører på forskjellige endesystemer med hver sin IP-adresse. For replikerte webservere, et sett av IP-adresser, er da assosiert med ett kanonisk vertsnavn, som `cnn.com`. Når en nettleser spør om IP-adressen til vertsnavnet får man returnert hele settet med IP-adresser, som nettleseren velger en fra, gjerne den første i settet.
  - **Lastfordeling:** DNS er også brukt til lastfordeling på replikerte servere, som replikerte webservere. Når en nettleser ber om en IP-adresse til et vertsnavn med replikerte webservere, får den et sett med IP-adresser. Rekkefølgen på disse IP-adressene blir bestemt av DNS-en..

### Hvordan funker DNS

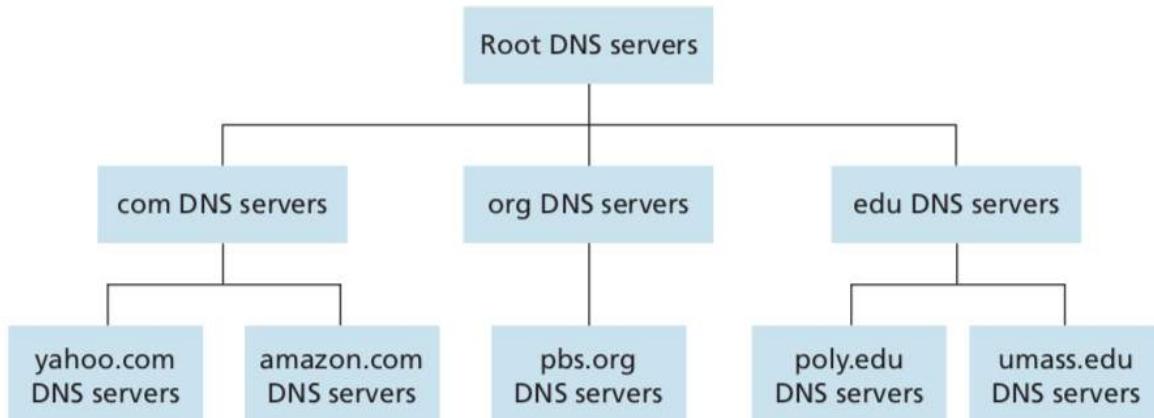
- Applikasjonen vil bruke klientsiden av DNS, og spesifiserer vertsnavnet som skal bli oversatt. På UNIX-baserte maskiner, brukes ofte funksjonen `gethostbyname()`. DNS-en lager så en spørring som sendes i nettverket. Alle DNS-spørreningar sendes med UDP datagrammer på port 53.
- Etter en forsinkelse på alt fra millisekunder til sekunder, får DNS-en i brukerens vert DNS-responsmeldingen med den ønskede mappingen.

I et enkelt design for DNS, ville det ha vært en DNS-server som har alle mappingsene. I dette sentraliserte designet kan det være flere problemer:

- *Single point of failure.* Dersom DNS-serveren krasjer, gjør også hele internettet
- *Traffic volume.* En enkel DNS server må håndtere alle DNS-spørreningar (for alle HTTP-requestene og epostene generert av flere undre millioner verter)
- *Langveis sentralisert database.* En enkelt DNS-server kan ikke være nærmest alle de spørrende klientene. Dette kan føre til forsinkelser.

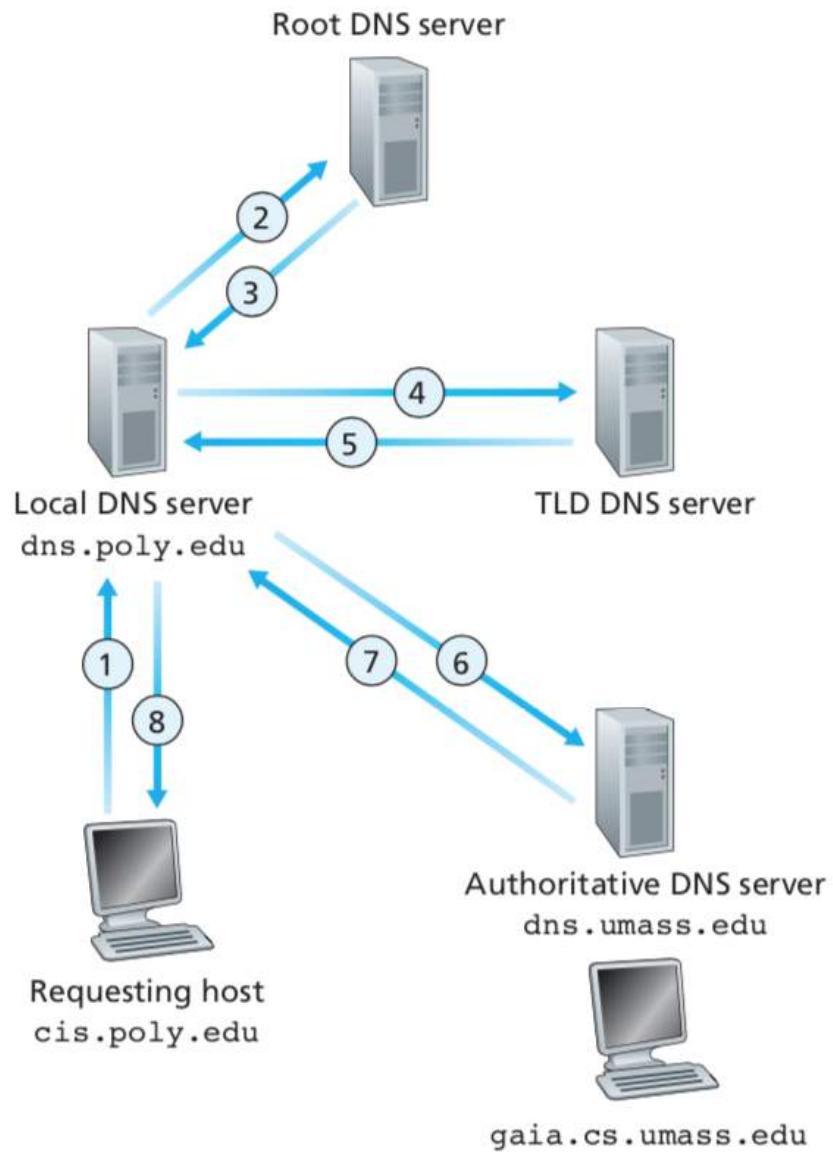
- Vedlikeholding. EN enkelt DNS-server vil måtte holde data for alle verter på internettet. Ikke bare vil denne sentraliserte databasen være stor, men måtte oppdateres

En distribuert, hierarkisk database:

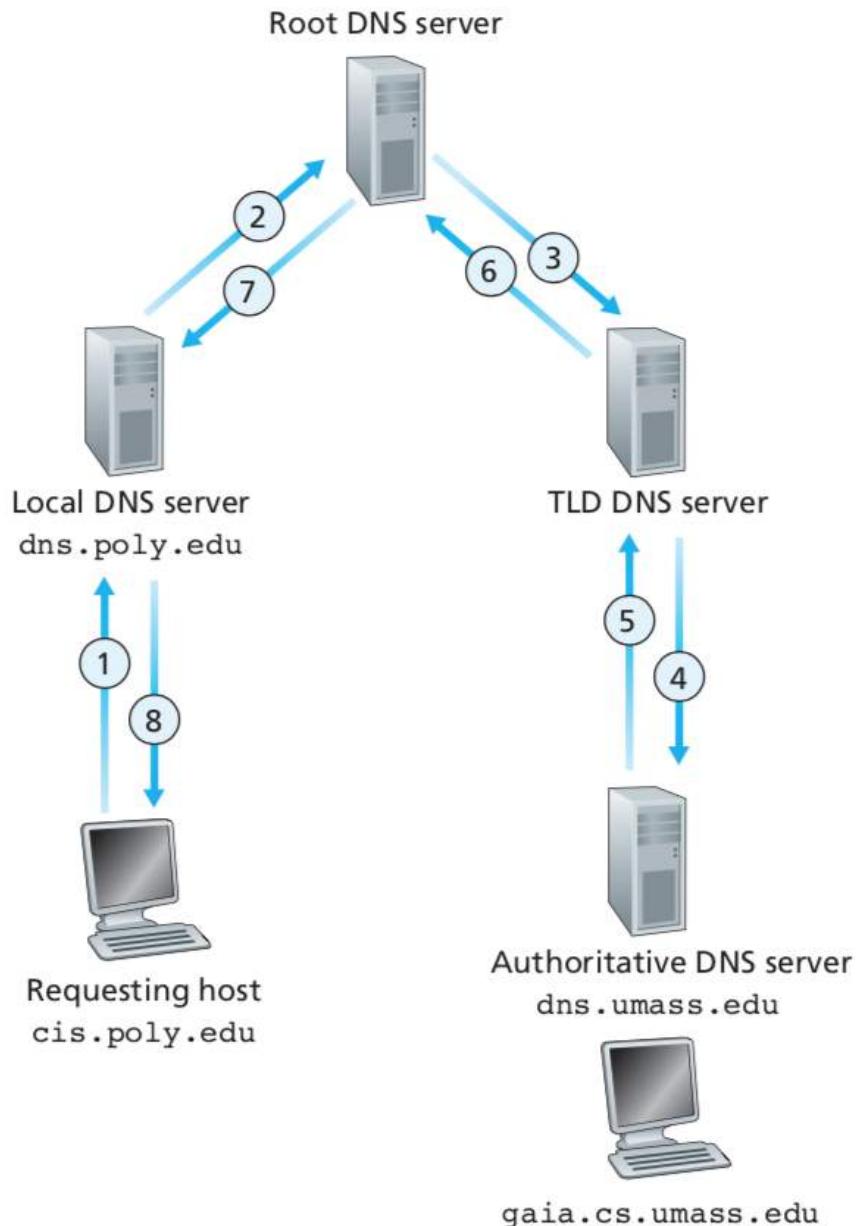


**Figure 2.19** ♦ Portion of the hierarchy of DNS servers

- Dersom en klient vil ha IP-adressen til *amazon.com*, vil man først spørre en av rot-serverene som vil gi ip-adresser for TLD-serverene (*top-level-domain*) på top-nivå-domenet *com*. Klienten kontakter så en av disse TLD-serverene og får returnert IP-adressen til *amazon.com*
- **Rot-DNS-servere.** Finnes 13 stk i verden (A-M). En rot-DNS-server er et nettverk av replikerte servere.
- **Top-nivå-domene-servere (TLD).** Disse serverene er ansvarlige for top-nivå-domener slik som *com*, *net*, *edu* og *gov*.
- **Autoritative DNS-servere.** Hver organisasjon med offentlig tilgjengelige verter på internettet på gi offentlig aksessbare DNS-data som mapper alle vernetavnene til IP-adresser.
- **Lokal DNS-server.** En lokal DNS-server hører ikke til i hierarkiet av servere, men er likevel sentral i DNS-arkitekturen. Hver ISP har en lokal DNS-server. Når en vert kobler seg på en ISP, vil ISP'en gi vertens IP-adressen til en av dens lokale DNS-servere. Når man gjør en DNS-forespørsel vil den først gå gjennom den *lokale DNS-serveren*, som vil oppføre seg som en proxy.



**Figure 2.21** ♦ Interaction of the various DNS servers



**Figure 2.22 ♦ Recursive queries in DNS**

#### DNS Caching:

En kritisk del av DNS-systemet. DNS utnytter i stor grad DNS-caching for å forbedre forsinkelsesytelsen og redusere antall DNS-meldinger som sendt rundt på internettet.

Ideen bak DNS-caching er veldig enkel. I en spørringeskjede, når en DNS-server mottar et DNS-svar (som for eksempel inneholder en mapping fra et vertsnavn til en IP-adresse), kan det cache mappingen i sitt lokale minne.

#### DNS poster og meldinger

DNS-serverne som sammen implementerer den distribuerte DNS-databasen lagrer **ressursposter (RRs - Resource Records)**, inkludert RR-er som gir vertsnavn-til-IP-adresse mappinger. Hver DNS response-melding bærer en eller flere ressursposter.

En ressurspost er en fir-tuppel med følgende felter:

(Name, Value, Type, TTL)

- Dersom Type=A , da er Name vertsnavn og Value er IP-adressen til vertsnavnet
- Dersom Type=NS , da er Name et domene og Value er vertsnavnet til den autorative DNS-serveren

- Dersom Type=CNAME , da er Name vertsnavn-alias og Value er kanonisk vertsnavn
- Dersom Type=MX , da er Name vertsnavn-alias og Value er kanonisk vertsnavn for mailserver

#### DNS-meldinger

Jeg har tidligere nevt DNS-spørninger og respons-meldinger. Dette er de to typene vi har av DNS-meldinger. Slik er semantikken til DNS-meldinger:

- De første 12 bytene er *header-seksjonen*, som har et antall felter. Første feltet er et 16-bit nummer som identifiserer spørringen. Denne identifikatoren blir kopiert over i respons-meldingen. Det er også et antall flagg. En 1-bit *query/reply-flagg* indikerer spørring(0) og respons(1). Andre 1-bit flagg er *authoritative-, recursion-desired* og *recursion-available*. Ellers er det også også 4 antall-av-felt. Disse feltene indikerer antall av forekomster av daseksjonene under headeren.
- *Spørsmålseksjonen* inneholder informasjonen om spørringen som skal bli gjort. Denne seksjonen inneholder:
  - Navn-felt med navn på det skal bli forespurt
  - Typefelt som indikerer typen spørring (Type A eller f.eks. Type MX)
- *Svarseksjon*, i et svar fra en DNS-server, inneholder ressurspostene for navnet som i utgangspunktet ble forespurt.
- *Autoritetsseksjonen* inneholder poster av andre autoritative servere.
- *Tilleggsseksjonen* inneholder andre hjelphulde poster.

#### Innsetting i DNS Databasen

Når man lager et nytt domene må man gjøre dette gjennom en **registrar**. En registrar er en kommersiell enhet som bekrefter det unike domenenavnet, går inn i domenenavnet i DNS-databasen, og samler en liten avgift fra deg for sine tjenester.

Dersom en ny nettside skal innsettes, og en antar at navnene og IP-adressene er *dns1.networkutopia.com*, *dns2.networkutopia.com*, 212.212.212.1, og \*212.212.212.2 \*. Da må en sette inn følgende ressurspostene i DNS-systemet:

- (*networkutopia.com*, *dns1.networkutopia.com*, NS)
- (*dns1.networkutopia.com*, 212.212.212.1, A)

### Peer-to-Peer Application

Applikasjonene introdusert hittil - inkludert Internettet, epost og DNS - er alle klient-server arkitekturen som avhenger av alltid-på infrastruktur servere.

I P2P-arkitektur er det minimal (eller ingen) avhengighet av alltid-på-infrastruktur servere. I stedet kommuniserer par av tilkoblede verter, kalt peers, direkte med hverandre.

Vi skal se på applikasjoner som er spesielt velegnet for P2P-design. Den vi skal se på er filfordeling, hvor applikasjonen fordeler en fil fra en enkelt kilde til et stort antall peers. Fildistribusjon er et fint sted å starte vår undersøkelse av P2P, da det tydeligvis avslører selvskalering av P2P-arkitekturen.

#### P2P Fildistribusjon

Vi skal se på å distribuere en stor fil fra en enkelt server til et stor antall verter (kalt peers). I en P2P fildistribusjon kan hver peer redistribuere enhver del av filen som den har mottatt fra andre peers, og dermed hjelpe serveren med å distribusjonsprosessen.

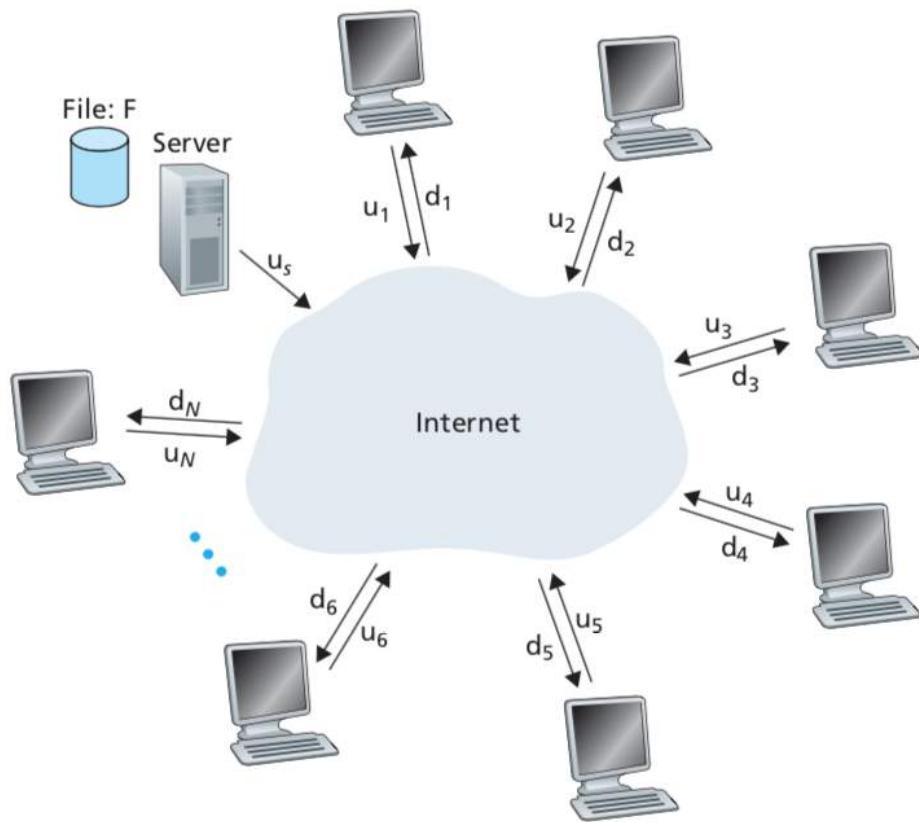
I 2012 var den mest fildistribusjonsprotokollen BitTorrent.

#### Skalerbarhet av P2P-arkitekturen

For å sammenlikne klient-server arkitekturen med P2P-arkitekturen og illustrere den iboende selv-skalerbarheten til P2P, skal vi se på en enkelt modell for å distribuere en fil til et antall peers med begge arkitekturene.

Som vist i figuren under er både server og klienter koblet til internettet med aksesskoblinger. Betegn opplastningsraten til serveren er  $u_s$ , opplastningsraten til det  $i$ -te peers aksesskobling er  $u_i$ , og nedlastningsraten til det  $i$ -te peers aksesskobling er  $d_i$ .

Betegn også størrelsen på filen til å bli distribuert med  $F$ , og antall peers som vil ha filen er  $N$ .



**Figure 2.24** ♦ An illustrative file distribution problem

Distribusjonstiden er tiden det tar for at alle peers-ene får en kopi av filen.

La oss først regne ut distribusjonstiden for **klient-server-arkitekturen**, som vi betegner  $D_{CS}$ . I denne arkitekturen hjelper ingen av peers-ene med på å distribuere filen. Vi gjør følgende observasjoner:

- Serveren må distribuere  $F$  bits til  $N$  peers. Det vil si sende  $NF$  bits. Siden serveren har en opplastningsrate på  $u_s$ . Da blir distribusjonstiden minst  $NF/u_s$ .
- La  $d_{min}$  være nedlastningsraten til peeren med den laveste nedlastningsraten. Derfor vil distribusjonstiden være minst  $F/d_{min}$ .

Setter vi sammen disse observasjonene får vi,  $D_{CS} \geq \max \{ NF/u_s, F/d_{min} \}$

La oss nå regne ut distribusjonstiden for **peer-to-peer-arkitekturen**, som vi betegner  $D_{P2P}$ . Vi gjør følgende observasjoner:

- Ved begynnelsen av distribusjonen, er det kun serveren som har filen. For å få denne filen til et fellesskap med peers, må serveren sende filen minst en gang gjennom aksesskoblingen. Dermed må distribusjonstiden være minst  $F/d_{min}$  sekunder.
- Som i klient-server-arkitekturen, kan ikke peeren med lavest nedlastningsrate få filen kjappere enn  $F/d_{min}$  sekunder. Som blir et minimum for distribusjonstiden.
- Til slutt, observer at den totale opplastningskapasiteten til systemet som en helhet er opplastningsraten til serveren pluss opplastningsraten til alle de individuelle peers-ene. Systemet må laste opp  $F$  bits til hver av de  $N$  peer-ene, altså  $NF$  bits. Dette kan ikke bli gjort kjappere enn  $NF/(u_s + u_1 + \dots + u_N)$ .

Setter vi sammen disse observasjonene får vi,  $D_{P2P} = \max \{ F/u_s, F/d_{min}, NF/(u_s + \sum u_i) \}$

Nå har vi sett på et antall viktige nettverksapplikasjoner, nå skal vi se på hvordan nettverk applikasjons-programmer blir laget. Husk fra Kap. 2 at en typisk nettverksapplikasjon består av et par av programmer - et klientprogram og et serverprogram - liggende på to forskjellige endesystemer.

Når disse programmene startes, startes en klientprosess og en serverprosess, og disse prosessene kommuniserer med hverandre gjennom sockets (*norsk. nettverkssocket*)

Det finnes to typer nettverksapplikasjoner:

- Den ene typen er en implementering der operasjoner er spesifisert i en protokollstandard, for eksempel en RFC eller et annet standarddokument. Et slikt program kalles noen ganger "åpen", siden reglene som angir operasjonen er kjent for alle. For en slik implementering må klient- og serverprogrammene overholde de regler som er angitt av RFC.
- Den andre typen nettverksapplikasjon er et proprietært nettverksprogram. I dette tilfellet bruker klient- og serverprogrammene en protokoll for applikasjonslag som ikke har blitt offentliggjort i en RFC eller et annet sted. Utvikleren har full kontroll over hva som skjer i koden.

Når en skal utvikle en klient-server applikasjon, må man først bestemme seg for om applikasjonen skal kjøre over TCP eller over UDP.

- TCP: Koblingsorientert og pålitelig dataoverføring
- UDP: Koblingsløs og sender uavhengige pakker av data uten leveringsgaranti.

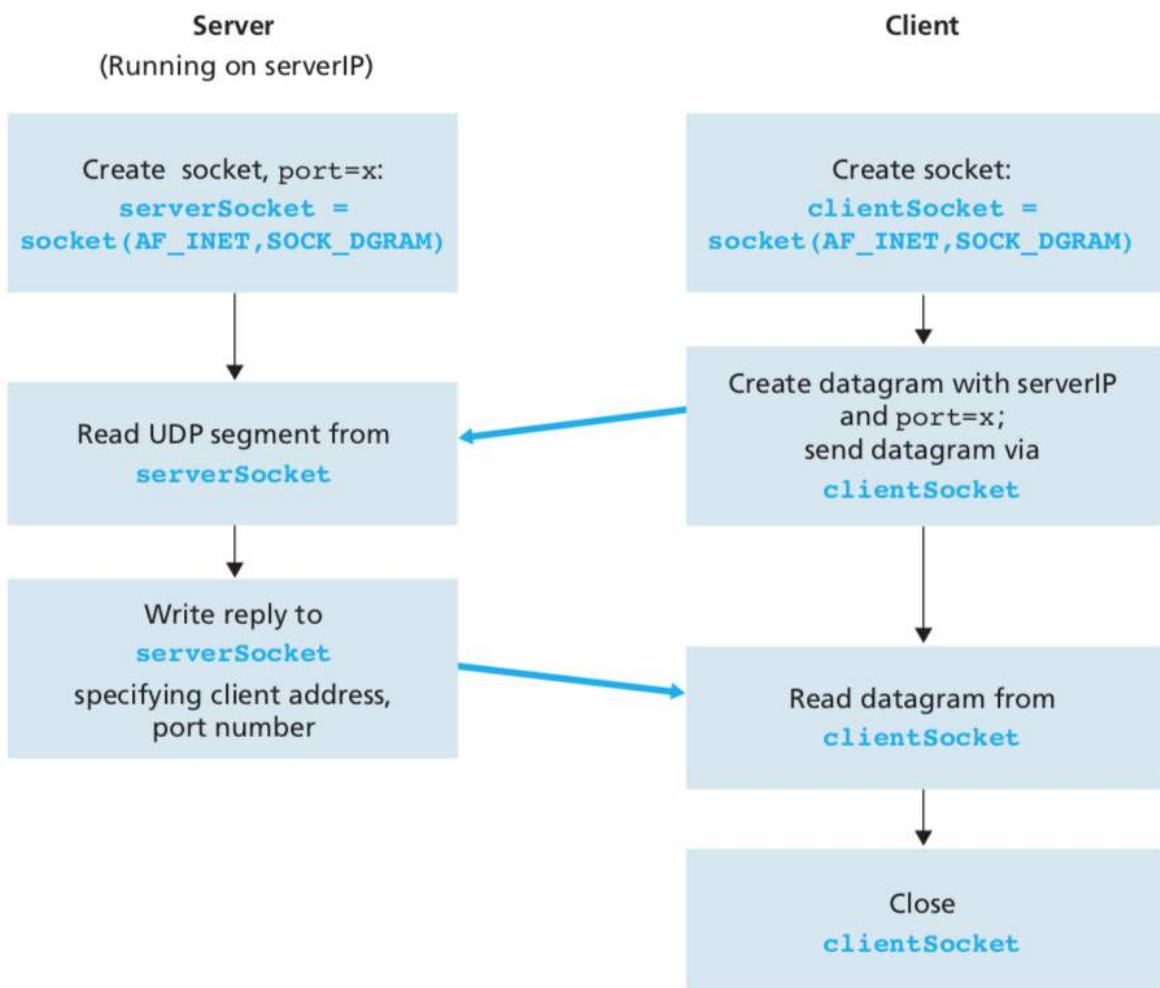
### Sock Programming with UDP

Vi kan se på hver prosess som et hus og hver socket som døren. Når man bruker UDP, må man først legge til destinasjonsadressen på pakken. Etter at pakken går igjennom senderens socket, vil internettet bruke destinasjonsadressen til å guide pakken frem til socketen på den mottakende prosessen.

Denne destinasjonsadressen er destinasjonsvertens *IP-adresse*, samt *portnummer* til socketen. I tillegg legges også ved senderens IP-adresse og portnummer, men dette gjøres ikke av UDP-applikasjonskoden, men det underliggende systemet.

Vi bruker følgende klient-server-applikasjon for å demonstrere programmering for både UDP og TCP:

1. Klienten leser linjer av karakterer (data) fra tastaturen og sender dataen til serveren.
2. Serveren mottar dataen og konverterer karakterene til blokkbokstaver.
3. Serveren sender den modifiserte dataen til klienten.
4. Klienten mottar den modifiserte dataen og viser linjen på skjermen.



UDPClient.py:

```

from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message,(serverName, serverPort))
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print modifiedMessage
clientSocket.close()
  
```

UDPServer.py:

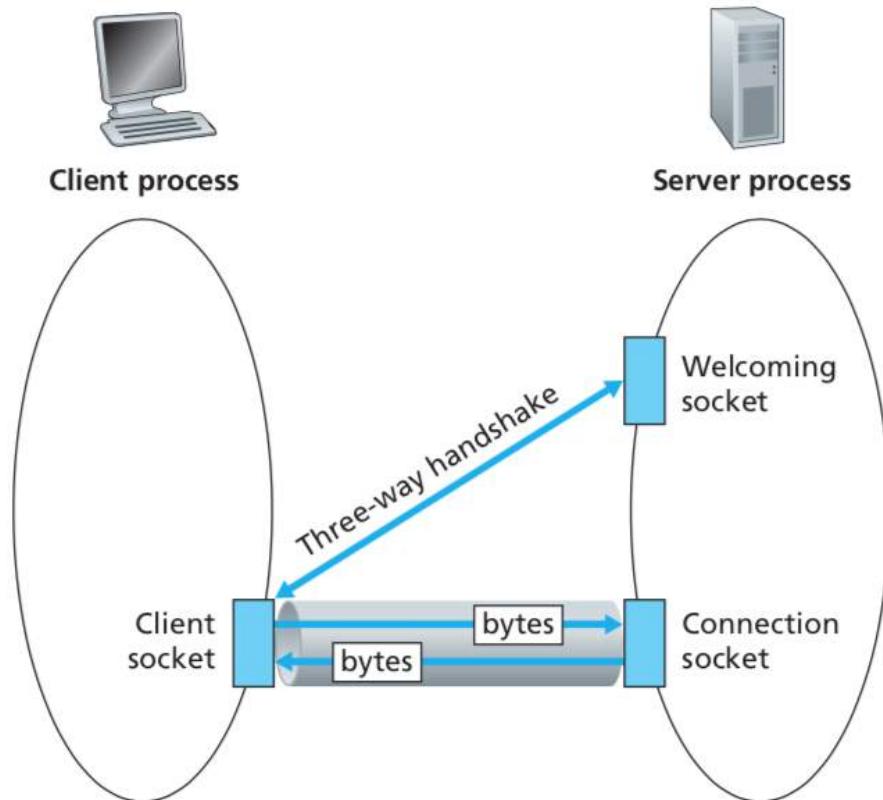
```

from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print "The server is ready to receive"
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, clientAddress)
  
```

## Socket Programming with TCP

I motsetning til UDP, er TCP en tilkoblings-orientert protokol. Dette betyr at klienten og serveren må handshake og etablere en TCP-tilkobling før de kan sende data til hverandre.

Her kan serveren eller klienten kun slippe data ned i TCP-koblingen, uten å måtte legge ved ekstra data som ved UDP.



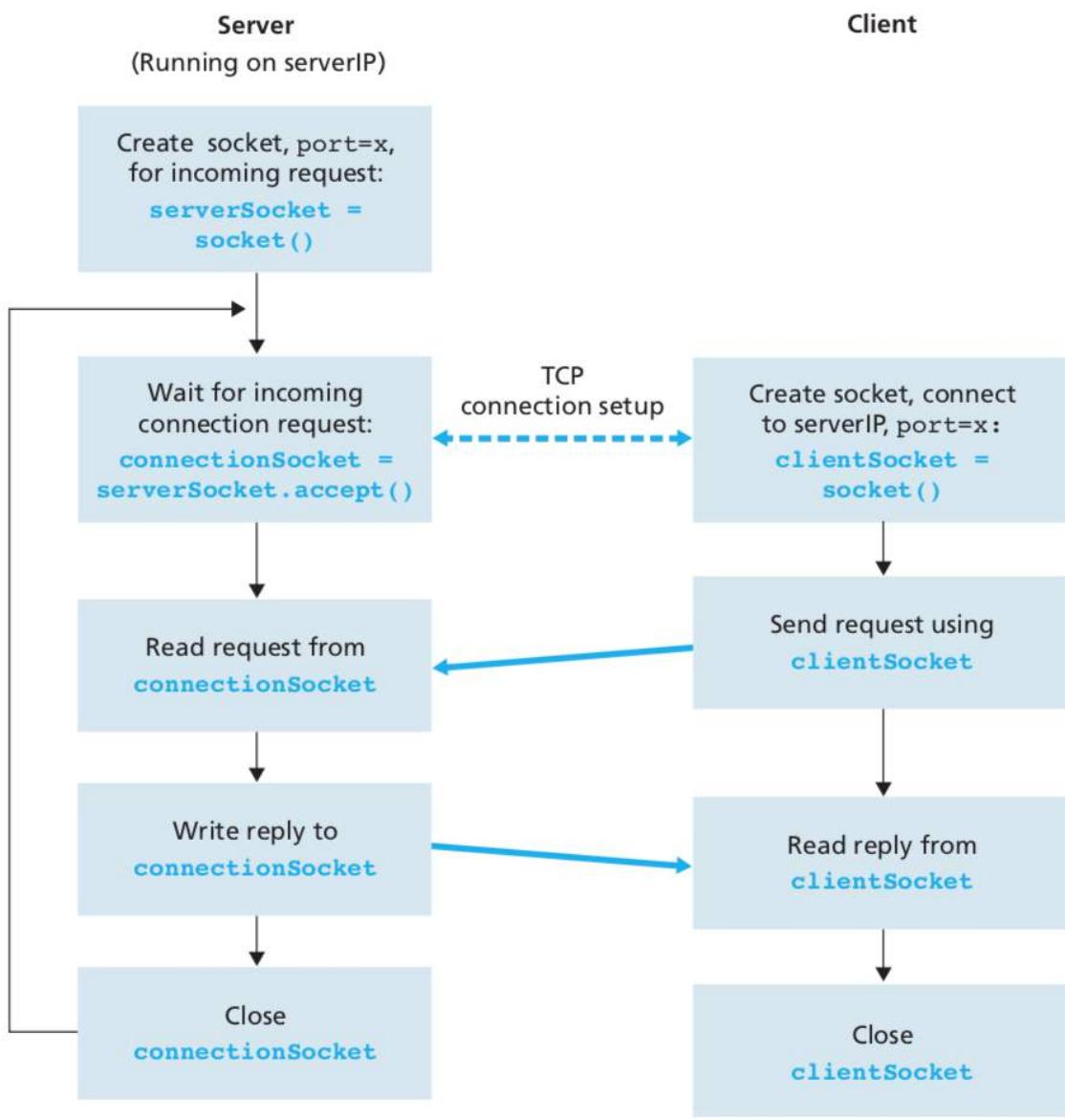
**Figure 2.29 ♦** The TCPServer process has two sockets

TCPClient.py:

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

TCPClient.py:

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```



**Figure 2.30** ♦ The client-server application using TCP

## Oppsummering

- Applikasjonslagsprotokoller: HTTP, FTP, SMTP, POP3, IMAP og DNS
- Transportlagprotokoller: TCP, UDP

## Kapittel 3 - Transport Layer

### Introduction and Transport-Layer Services

En transportlagsprotokol tilbyr **logisk kommunikasjon** mellom applikasjonsprosesser kjørende på forskjellige verter. Med *logisk kommunikasjon* snakker vi fra en applikasjonsprosess, det blir som at vertene er direkte sammenkoblet, men i virkeligheten kan det hende at de er på forskjellige sider av jorden.

Transportlagsprotokoller er implementert i endesystemer, men ikke i rutere.

På den **sendende** siden vil trasportlaget konvertere applikasjonlags-meldingene den mottar til transportlag-pakker, som er kjent som transportlag **segmenter**. Dette blir gjort ved å dele applikasjonsmeldingene opp i enda mindre biter, og legge på transportlags-headeren til hver bit, for å lage transportlagssegmentet. Transportlaget sender så segmentet til netverkslaget, hvor segmentet blir innkapslet i en nettverklagspakke (*datagram*), og sendt til destinasjonen. Det er viktig å notere seg at nettverksrutere bare benytter seg av nettverklagsfeltene på datagrammet.

På den **mottakende** siden, henter nettverkslaget segmentet innkapslet i datagrammet opp til transportlaget. Transportlaget prosesserer det mottatte segmentet, og gjør dataen tilgjengelig for den mottakende applikasjonen.

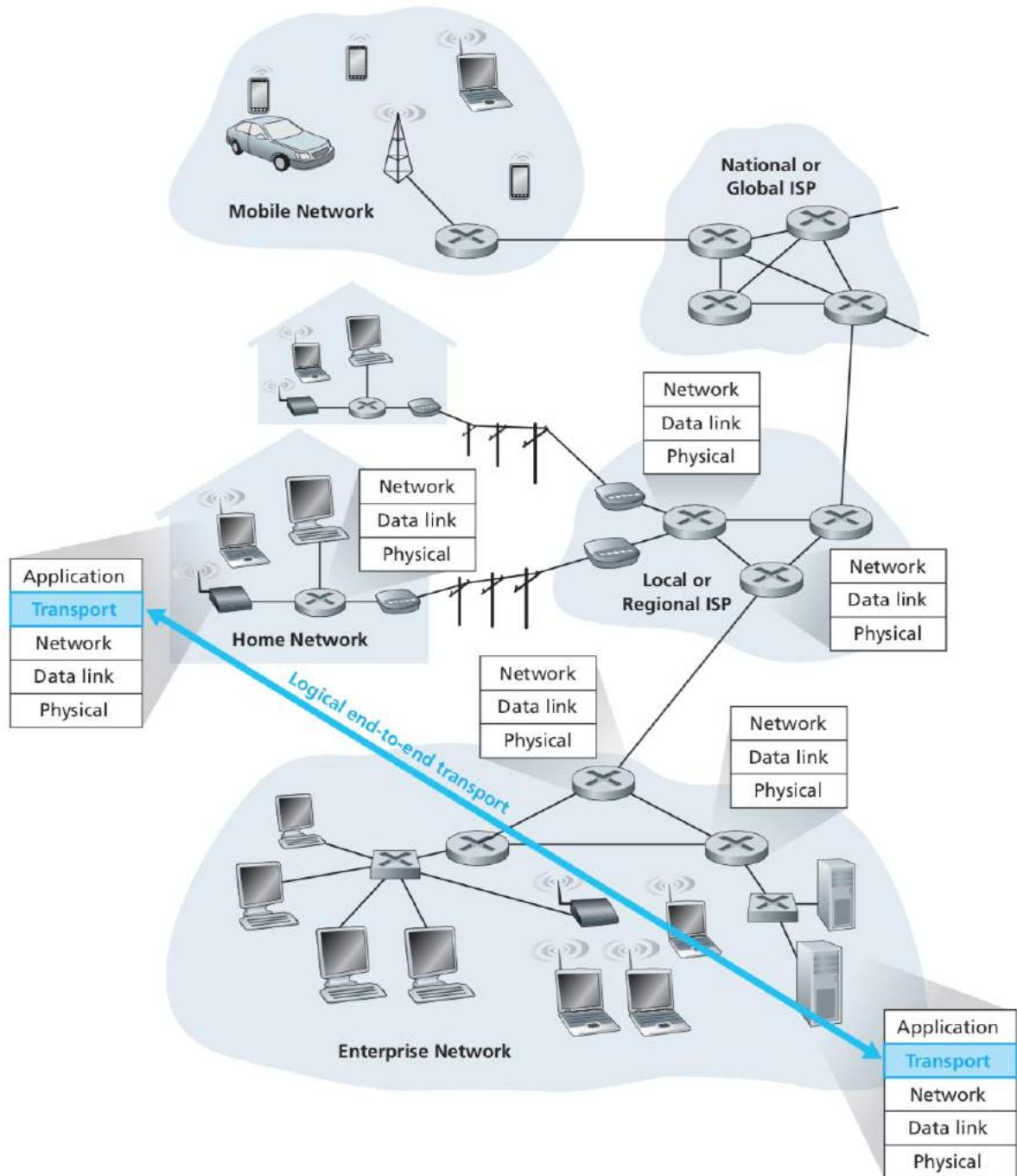
- Det er flere enn en transportlagsprotokoll som er tilgjengelig for nettverksapplikasjoner. F.eks. har internettet to protokoller - TCP og UDP. Hver av disse protokollene tilbyr forskjellige sett med transportlagstjenester.

### Forholdet mellom transport- og nettverkslaget

Minnes om at transportlaget ligger rett over nettverklaget i protokollstakken. Mens en transportlagsprotokoll gir logisk kommunikasjon mellom prosesser som kjører på forskjellige verter, gir en nettverkslagprotokoll logisk kommunikasjon mellom verter. Denne forskjellen er subtil men viktig. La oss undersøke dette skillet ved hjelp av en husstandsanalyse.

- Se på to hus, ett på østkanten og et på vestkanten, og i hvert hus bor et dusin barn. Barna på østkanten søskenhvile med de på vestkanten. De liker å skrive brev til hverandre. I begge husstandene er det ett barn - Ann på vestkanten og Bill på østkanten - som er ansvarlige for å hente og levere post. Hver uke går Ann og Bill rundt og henter brev fra søskenen sine, leverer og henter posten, og leverer brev til søsknene sine.
  - I dette eksempelet tilbyr posten logisk kommunikasjon mellom de to husene. Ann og Bill tilbyr logisk kommunikasjon mellom søskenhvile. Analogien var slik:

applikasjonsmeldinger = brev i konvolutt  
prosesser = søskenhvile  
verter (endesystemer) = hus  
transportlagsprotokoller = Ann og Bill  
nettverksprotokoller = postvesenet



**Figure 3.1** ♦ The transport layer provides logical rather than physical communication between application processes

Visse tjenester kan bli tilbuddt av transportprotokoller, selv når de underliggende nettverksprotokollene ikke tilbyr samme tjeneste på nettverkslaget. For eksempel kan en transportprotokoll tilby pålitelig dataoverføring til en applikasjon, selvom den underliggende nettverksprotokollen er upålitelig.

#### Oversikt over transportlaget på internettet

Minnes at internettet, eller mer generelt et TCP/IP internett, har to distinkte transportlagsprotokoller tilgjengelige for appliksjonslaget.

- Den første er **UDP (User Datagram Protocol)**, som tilbyr en upålitelig, koblingsløs tjeneste.
- Den andre er **TCP (Transmission Control Protocol)**, som tilbyr pålitelig, koblingsorientert tjeneste til den brukende applikasjonen.
  - Tilbyr flow control, congestion control, sequence numbers, acknowledgments og timere

Når man skal lage en nettverksapplikasjon må man velge mellom UDP og TCP når man skal lage nettverkssockets.

En protokoll som tilbyr pålitelig dataoverføring og metningskontroll er nødvendigvis kompleks.

- Vi refererer til transportlagspakker som **segmenter** og nettverkspakker som **datagrammer**.

Internettets nettverksprokol kalles **IP (Internet Protocol)**, som tilbyr logisk kommunikasjon mellom verter. IP-tjenestemodellen er en **best-effort delivery service**. Hvilket betyr at IP gjør så godt den kan for å levere segmenter mellom verter, men *gir ingen garanti*, IP er derfor sagt å være en **upålitelig tjeneste**.

For å oppsummere:

Det mest grunnleggende ansvaret for UDP og TCP er å utvide IPs leveringstjeneste mellom to endesystemer til en leveringstjeneste mellom to prosesser som kjører på endesystemene. Utvidelse av **vert-til-vert-levering** til **prosess-til-prosess-levering** kalles *transportlagsmultipleksing* og *demultipleksing*.

## Multipleksing og Demultipleksing

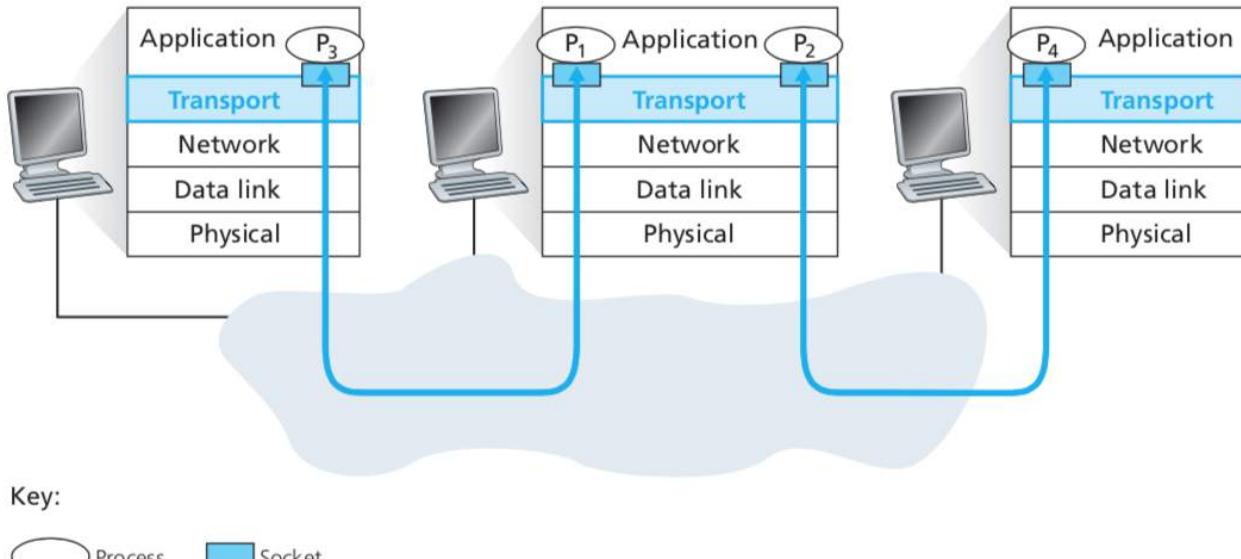
Ved destinasjonsverten mottar transportlaget segmenter fra nettverkslaget rett under. Transportlaget har ansvaret for å ivare daten i segmentene til den riktige applikasjonsprosessen hos verten.

- Anta at du laster ned nettsider, samtidig som du laster ned en fil med FTP og har to Telnet-sessioner. Da har man 4 prosesser kjørende - to Telnet-, en FTP- og en HTTP-prosess. Når transportlaget mottar data fra nettverkslaget må den sende dataen til en av disse prosessene.

En prosess kan ha mer enn en **socket**, som fungerer som dører som data går igjennom fra nettverket til prosessen, og motsatt. Som Figur 3.2 viser, så vil transportlaget på den mottakende verden levere dataen til en mellomliggende socket, ikke direkte til prosessen.

Hvert transportlags-segment har et sett med felt i segmentet. Ved mottakeren undersøker transportlaget disse feltene for å identifisere den mottakende socketen og leder deretter segmentet til denne socketen.

Denne jobben med å levere dataene i et transportlagsegment til *riktig socket* kalles **demultiplexing**. Arbeidet med å samle databiter ved kildeverten fra forskjellige sockets, innkapsling av hver data-bit med header-informasjon (som senere vil bli brukt i demultiplexing) for å opprette segmenter, og å sende segmentene til nettverkslaget kalles **multiplexing**.

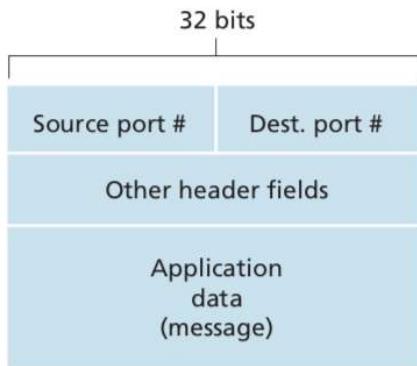


**Figure 3.2 ♦ Transport-layer multiplexing and demultiplexing**

Transportlags multipleksing krever:

1. Sockets har unike identifikatorer
2. Hvert segment har spesielle felt, som indikerer hvilken socket segmentet skal bli levert hos.

Disse spesielle feltene er **kilde-port-nummeret** og **destinasjons-port-nummeret**:



**Figure 3.3** ♦ Source and destination port-number fields in a transport-layer segment

Hvert portnummer er et 16-bitters nummer, alt fra 0 til 65535. Når vi lager en ny applikasjon må vi tildele applikasjonen et portnummer.

#### Koblingsløs multipleksing og demultipleksing - UDP

Dersom vi kobler lager en UDP-socket og kobler den opp til et portnr, kan vi beskrive hvordan UDP multipleksing/demultipleksing.

```
```python
clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)
clientSocket.bind(('', 19157))
````
```

En UDP-port er identifisert med en tuppel av (*destinasjons IP-adresse, destinasjons port*)

La oss tenke oss at Vert A, med UDP-port 19157, ønsker å sende en haug med data til en prosess med UDP-port 46428 hos Vert B. Da lager transportprotokollen i Vert A et transportlags-segment som inkluderer dataen, kildeportnummeret (19157), destinasjonsportnummeret (46428), og to andre verdier (ikke viktig enda). Transportlaget sendr så segmentet til nettverkslaget, som innkapsler segmentet i et IP-datagram og prøver å levere det.

Dersom segmentet kommer frem til Vert B, vil transportlaget eksaminere destinasjonsportnummeret på segmentet (46428), og levere det hos socketen identifisert med portnr 46428.

Kildeportnummeret brukes som en *retur-adresse*, dersom Vert B ønsker å sende noe til Vert A. I python brukes metoden `recvfrom()` for å hente kildeportnummeret.

To segmenter med samme kilde-portnummer og/eller kilde-IP-adresse vil kunne sendes til samme socket.

#### Koblings-orientert multipleksing og demultipleksing - TCP

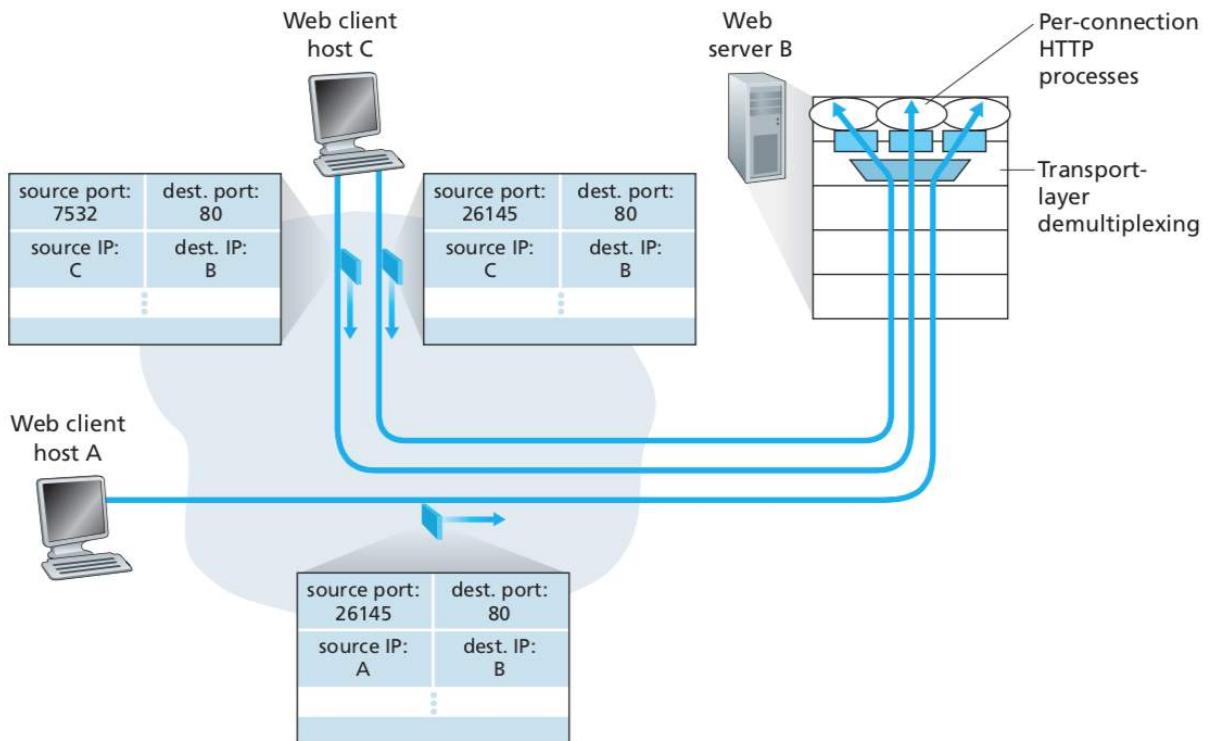
En forskjell på en TCP-socket og en UDP-socket er at en TCP-socket er identifisert med en firtuppel (*kilde-IP-adresse, kilde-portnummer, destinasjons IP-adresse, destinasjons portnummer*). Når et TCP-segment ankommer verten, brukes alle fire verdiene for å demultiplekse segmentet til riktig socket.

I kontrast med UDP, vil to ankommende TCP-segmenter med forskjellig kilde IP-adresser eller kilde-portnummer (med unntak av et TCP-segment som bærer den opprinnelige tilkoblings-estableringsforespørselen) sendes til to forskjellige porter.

La oss se på TCP klient-server programmeringen fra forrige kapittel:

- TCP-serverapplikasjonen har en "welcoming socket" som venter på tilkoblingsforespørsler fra TCP-klienter.
- TCP-klienter lager en socket og snider en tilkoblingsforespørsel til TCP-serveren.
- En tilkoblingsforespørsel er bare et TCP-segment med destinasjonsportnummer, og et spesielt tilkoblingsforespørsel-bit-set i TCP-headeren, i tillegg til kilde-portnummer.
- Når serveren får tilkoblingsforespørselen og aksepterer forespørselen, og serverprosesen lager en ny socket.
  - Serveren noterer følgende felt fra tilkoblingsforespørsel-segmentet: (1) kilde-portnummer, (2) kildens IP-adresse, (3) destinasjons-portnummeret, (4) sin egen IP-adresse. Alle senere TCP-segmenter med disse verdiene vil bli

demultiplexet til denne porten.



**Figure 3.5** ♦ Two clients, using the same destination port number (80) to communicate with the same Web server application

Situasjonen i Figur 3.5, der Vert C initialiserer to HTTP-sesjoner til server B, og Vert A initialiserer én HTTP-sesjon til server B. Vert A, C og server B har unike IP-adresser. Vert C tilegner to forskjellige portnummer (26145 og 7532) for sine to HTTP-koblinger. Siden A har valgt et kilde-portnummer uavhengig av C, som kanskje også er samme som C, er ikke dette noe problem å ha samme kilde-portnr siden de to tilkoblingene har forskjellige kilde-IP-adresser.

Både de opprinnelige tilkoblingsforespørsels-segmentene og segmentene som bærer HTTP-request-meldinger, vil ha destinasjonsport 80.

### Connectionless Transport: UDP

UDP gjør omtrent like lite som en transportprotokoll kan gjøre. Bortsett fra multiplesking og demultiplexing og noen få enkle feilskjekker, legger den ingenting til IP.

Faksik hvis applikasjonsutvikleren velger UDP istedet for TCP, snakker applikasjonen nesten direkte med IP. UDP tar imot meldinger fra applikasjonsprosessen, legger til feltene *kilde-* og *destinasjonsportnummer* for multipleksing / demultiplexing-tjenesten, legger til to andre små felt og overfører det resulterende segmentet til nettverkslaget.

Nettverkslaget innkapsler transportlagssegmentet i et IP-datagram og gjør deretter sitt beste forsøk på å levere segmentet til mottakeren. Hvis segmentet kommer til mottakeren, bruker UDP destinasjonsportnummeret til å levere segmentets data til den riktige socketen / applikasjonsprosessen.

I UDP er det **ingen handshaking** mellom de sendende og mottakende transportlags-entitetene før sending av et segment. På grunn av dette sies UDP å være **koblingsløs**.

DNS er et eksempel på en applikasjonslagsprotokoll som bruker UDP. Når DNS-applikasjonen hos en vert ønsker å gjøre en spørring så lager den en DNS-spørringsmelding og sender meldingen til UDP. Dersom DNS-applikasjonen hos den spørrende verten ikke får svar vil den sende enten en ny spørring til en annen DNS-server eller fortelle applikasjonen at den ikke får svar.

Grunnen for at mange applikasjoner vil heller bruke UDP, fremfor TCP:

- *Finere applikasjonsnivå kontroll over hvilke data som er sendt, og når.* Når man sender data til UDP vil UDP pakke daten inn i et UDP-segment og sende pakket umiddelbart til nettverkslaget. TCP har derimot en overbelastningskontroll (*congestion control*) som sprer transportlagets TCP-avsender når en eller flere koblinger mellom kilde- og destinasjonsverten blir for overbelastede.
- *Ingen tilkoblingsetablering.* TCP bruker en tre-veis handshake, før den begynner å sende data. UDP, derimot, begynner å sende data uten noen formell innledning. Dermed er det ingen forsinkelse hos UDP for å opprette tilkobling. Dette er nok hovedgrunnen til at DNS kjører over UDP istedet for TCP. (Ville være mye tregere over TCP)
- *Ingen tilkoblingstilstand.* TCP vedlikeholder tilkoblingstilstanden i endesystemene. Denne tilkoblingstilstanden inkluderer motta- og sende-buffere, congestion-control-parametere, og sekvens- og bekreftelsesnummer-parametere. UDP har ingen av disse parameterene, og en server kan typisk støtte flere aktiv klienter når applikasjonen går over UDP enn TCP.
- *Små pakke-header overhead.* TCP-segmentet har 20 bytes av header overhead (sum av alt som ikke er data) i hvert segment, derimot har UDP kun 8 bytes av overhead.

| Application            | Application-Layer Protocol | Underlying Transport Protocol |
|------------------------|----------------------------|-------------------------------|
| Electronic mail        | SMTP                       | TCP                           |
| Remote terminal access | Telnet                     | TCP                           |
| Web                    | HTTP                       | TCP                           |
| File transfer          | FTP                        | TCP                           |
| Remote file server     | NFS                        | Typically UDP                 |
| Streaming multimedia   | typically proprietary      | UDP or TCP                    |
| Internet telephony     | typically proprietary      | UDP or TCP                    |
| Network management     | SNMP                       | Typically UDP                 |
| Routing protocol       | RIP                        | Typically UDP                 |
| Name translation       | DNS                        | Typically UDP                 |

**Figure 3.6** ♦ Popular Internet applications and their underlying transport protocols

Som nevnt tidligere har UDP *ingen congestion control*. Dersom alle skulle begynt å strømme høy-bit-rate video uten noen congestion control, ville det ha vært så stor overflow hos ruterne at veldig få UDP-pakker ville suksessfullt ha traversert over kilde-til-destinasjonsruten.

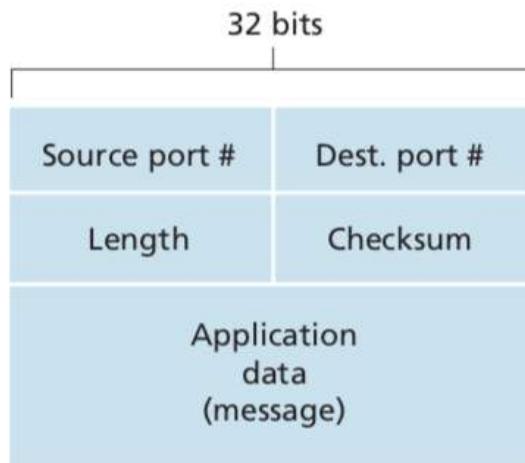
Dermed kan mangelen på congestion control i UDP resultere i høye tapsrater mellom en UDP-avsender og mottaker, og overbelastningen av TCP-økter.

Selvom man bruker UDP, som er upålitelig, er det mulig for en applikasjon å ha pålitelig dataoverføring med UDP. Dette kan bli gjort med å implementere pålitelighet i applikasjonen selv (f.eks. med bekreftelses- eller re-sending-mekanismser).

#### UDP-segmentstruktur

Strukturen til et UDP-segment er som beskrevet i Figur 3.7. Applikasjonsdataen tar datafeltet i segmentet. F.eks. for DNS, vil data-feltet inneholde en spørrmelding eller en responsmelding.

Header-feltet har kun fire fleter, hver bestående av to bytes. Som diskutert i forrige delavsnitt lar portnummerene destinasjonsverten levere dataen til riktig prosess (demultipleksingsprosessen). Length -feltet spesifiserer antall bytes i UDP-segmentet (*header + data*). Checksum -en er brukt av den mottakende verden for å sjekke om det har skjedd noen feil med segmentet.



**Figure 3.7** ♦ UDP segment structure

#### UDP Checksum

UDP Checksum-en sørger for feilsøking. Det bety, at checksummen blir brukt for å bestemme om noen av bitene i UDP-segmentet har blitt endret på under reisen fra kilde til destinasjon.

UDP på den sendende siden utfører 1s komplement på **summen** av alle 16-bit ord i segmentet. Dette resultatet blir puttet i Checksum -feltet til UDP-segmentet.

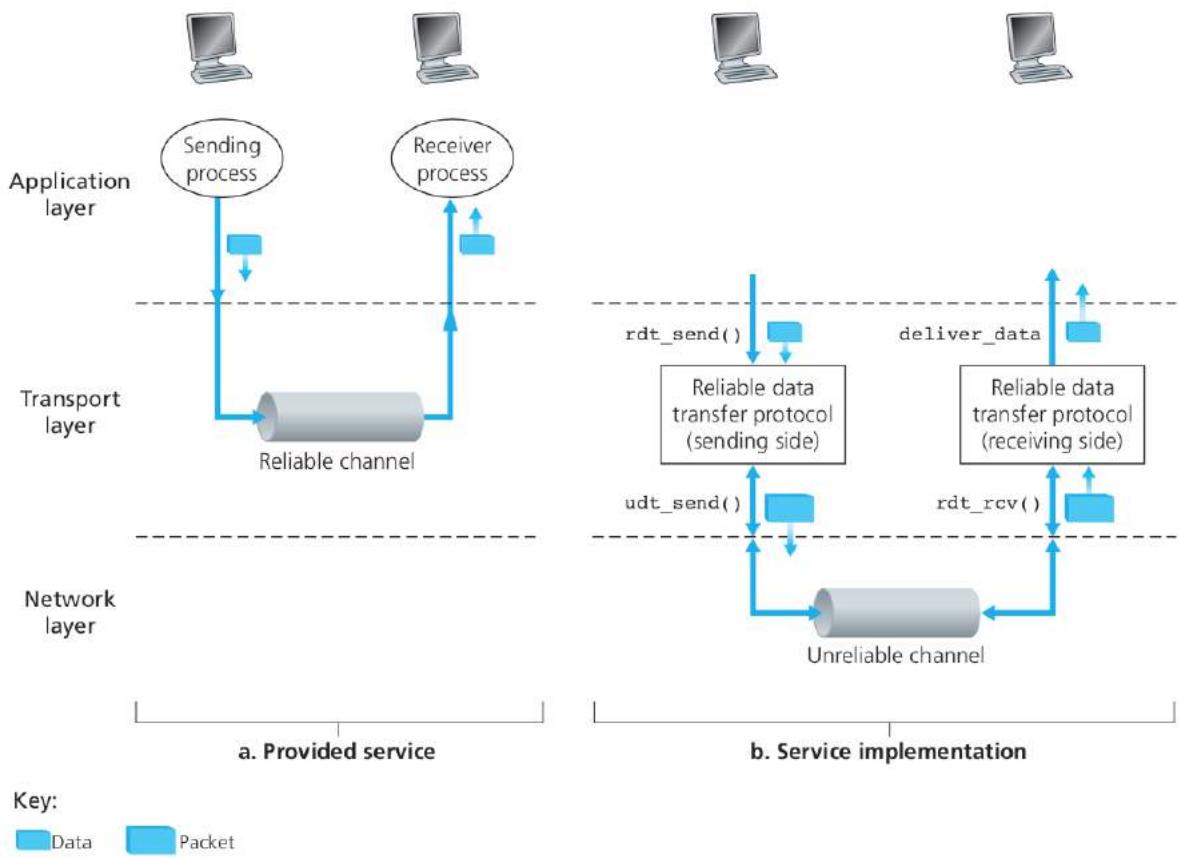
**16-BIT ORD:** I: 011001 //: 010101 ///: 100011 **SUM I = I + //:** 011001 + 010101 = 101110 **SUM II = SUM I + ///:** 101110 + 100011 = 010001 1s **KOMPL. SUM II:** = 101110 **Checksum = 101110**

#### Principles of Reliable Data Transfer

Dersom en tjenestemodell er pålitelig blir ingen overførte data-bit korrupte (vendt fra 0 til 1, eller omvendt) eller mistet, og alle leveres i den rekkefølgen de ble sendt i. Dette er nettopp tjenestemodellen som tilbys av TCP til Internett-applikasjoner som påberoper den.

Det er ansvaret for en *pålitelig dataoverføringsprotokoll* for å implementere denne tjenesteabstraksjonen.

Denne oppgaven er vanskelig av det faktum at laget under den pålitelige dataoverføringsprotokollen *kan* være *upålitelig*. For eksempel er TCP en pålitelig dataoverføringsprotokoll som implementeres på toppen av et upålitelig (IP) ende-til-ende nettverkslag.



**Figure 3.8** ♦ Reliable data transfer: Service model and service implementation

*rdt = reliable data transfer*

*udt = unreliable data transfer*

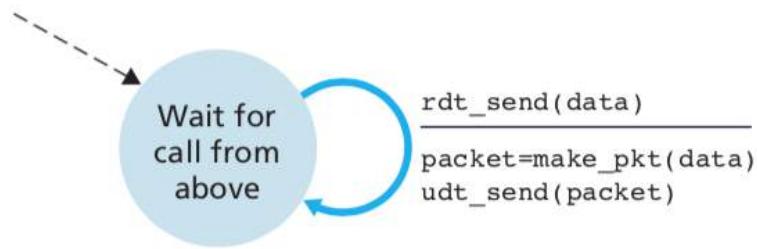
Vi skal å se på tilfellet av **enveis dataoverføring**, det vil si dataoverføring fra sendingen til mottakssiden. Saken om pålitelig **toveis** (det vil si fullsidig) **dataoverføring** er begrepsmessig ikke vanskeligere, men betydelig mer kjedelig å forklare.

#### Bygge en pålitelig dataoverføringsprotokoll

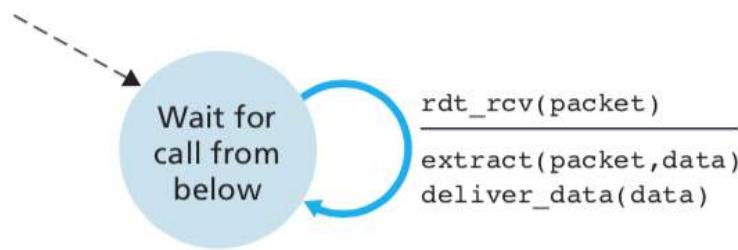
Vi skal nå se på en rekke protokoller, der hver blir mer kompleks og slutter med en feilfri, pålitelig dataoverføringsprotokoll.

#### Pålitelig dataoverføring over en perfekt pålitelig kanal: rdt1.0:

Tilstandsmaskinen (eng. *finite state machine - FSM*) definert for rdt1.0 er vist i Figur 3.9. Siden alt skjer over en helt fullstendig pålitelig kanal, trenger ikke den mottakende siden å gi noen feedback til den sendende siden - siden ingenting kan gå galt i kanalen.



a. rdt1.0: sending side



b. rdt1.0: receiving side

**Figure 3.9 ♦ rdt1.0 – A protocol for a completely reliable channel**

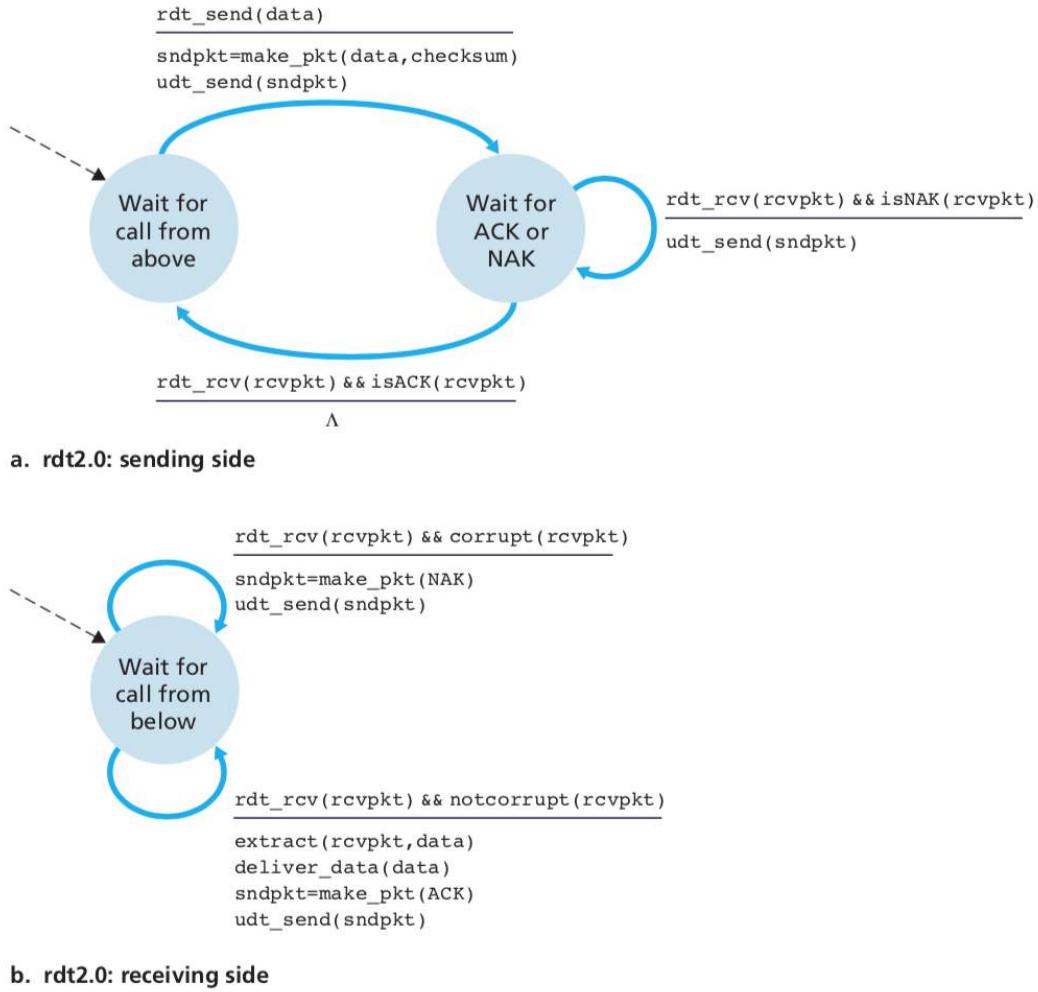
#### Pålitelig dataoverføring over en kanal med bitfeil: rdt2.0

Her implementeres feedback-meldinger som **positive acknowledgements** ("OK") og **negative acknowledgements** ("Vær snill å gjenta."). Pålitelig dataoverføringsprotokoller som baserer seg på retransmitting er kjent som **ARQ** (Automatic Repeat reQuest) protokoller.

Tre tilleggstjenester er krevt i ARQ-ptrokoller for å håndtere forekomst av bitfeil:

- *Feilsøking*. En mekanisme som lar mottaker sjekke om det har skjedd noen bit-endringer. Her benyttes *checksum*-feltet.
- *Tilbakemelding fra mottaker*. Siden sender og mottaker typisk kjører på forskjellige endesystemer, må mottakeren eksplisitt gi tilbakemelding til senderen. Positive (ACK) eller negative (NAK) bekreftelser er eksempler på slik tilbakemelding. Her kan 0 (NAK) eller 1 (ACK) benyttes.
- *Re-sending*. En pakke som mottas med feil ved mottakeren, blir sendt igjen av avsenderen.

Figuren under (3.10) viser tilstandsmaskinen til *rdt2.0*, en dataoverføringsprotokoll som har feilsøking, positive og negative bekreftelser.



**Figure 3.10** ♦ rdt2.0—A protocol for a channel with bit errors

Her vil ikke senderen kunne sende noe mer data, før den har fått vite at mottakeren har mottatt pakken. På grunn av dette kaller vi slike protokoller for **stop-and-wait**-protokoller.

Det ser kanskje ut som protokoll *rdt2.0* fungerer, men uheldigvis så har den en kjempefeil. Vi har ikke tatt i beregning av at ACK eller NAK pakkene kan være korrupte!

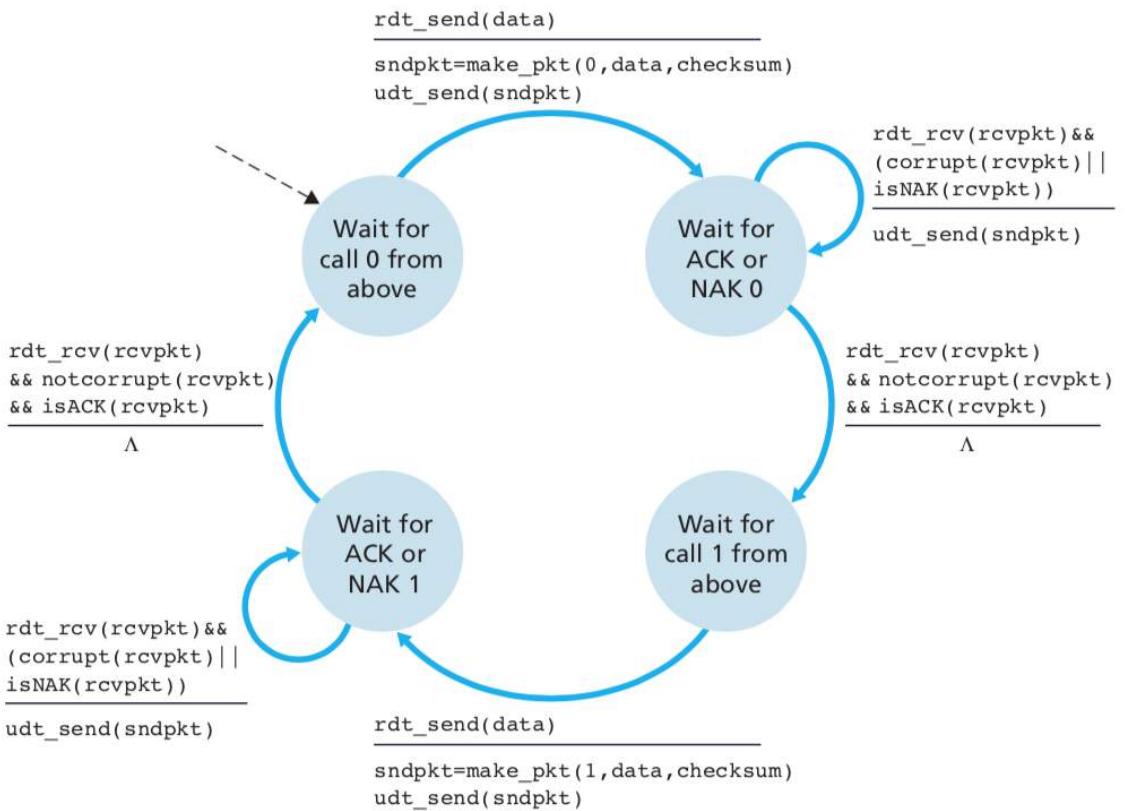
Vi kan innføre sekvensnummer. Mottakeren trenger kun å sjekke dette sekvensnummeret for å mestemme om den motatte pakken er en re-sending.

Figurene under er tilstandsmaskinen for *rdt2.1*, som nå har dobbelt så mange tilstander.

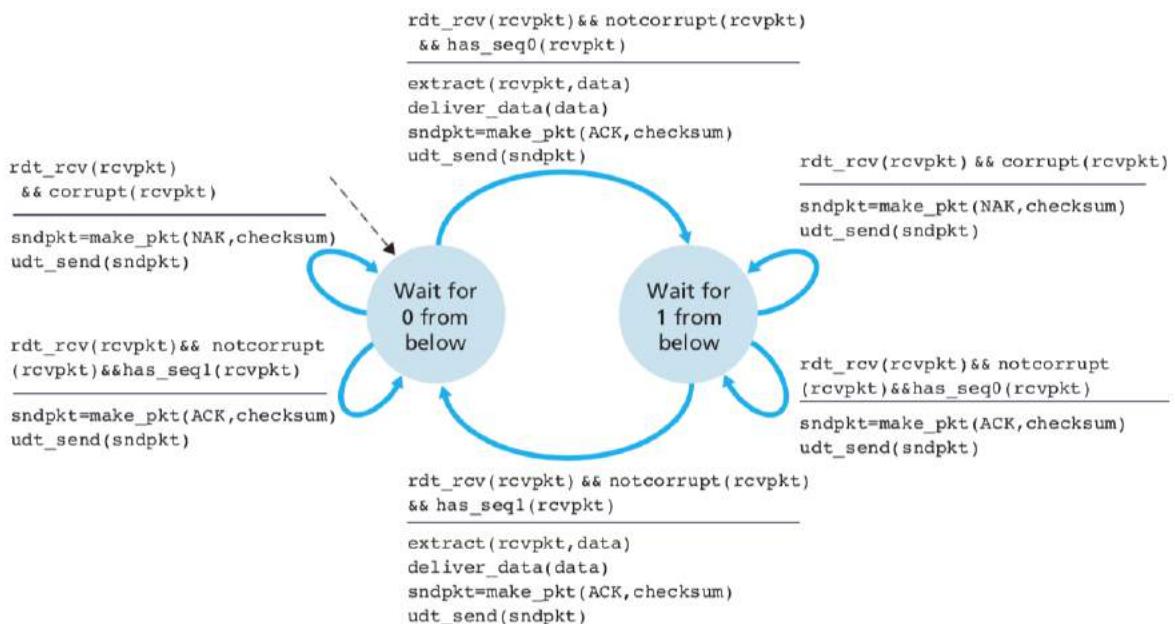
Dette skyldes at protokollstaten nå må gjenspeile om pakken som nå sendes (av avsenderen) eller er forventet (ved mottakeren), skal ha et sekvensnummer på 0 eller 1. Merk at handlingene i de tilstandene der en 0-nummerert pakke blir sendt eller forventet er speilbilder av de der en 1-nummerert pakke sendes eller forventes - De eneste forskjellene har å gjøre med håndteringen av sekvensnummeret.

Protokoll *rdt2.1* bruker både positive og negative bekreftelser fra mottakeren til avsenderen.

- Når en out-of-order-pakke mottas, sender mottakeren en positiv bekreftelse for pakken den har mottatt.
- Når en ødelagt pakke er mottatt, sender mottakeren en negativ bekreftelse.
  - Vi kan oppnå samme effekt som en NAK hvis vi, i stedet for å sende en NAK, sender en ACK for den sist riktig mottatte pakken.
- En avsender som mottar to ACKs for samme pakke (det vil si mottar duplikat ACKer) vet at mottakeren ikke mottok riktig pakken etter pakken som blir ACKed to ganger



**Figure 3.11** ♦ rdt2.1 sender



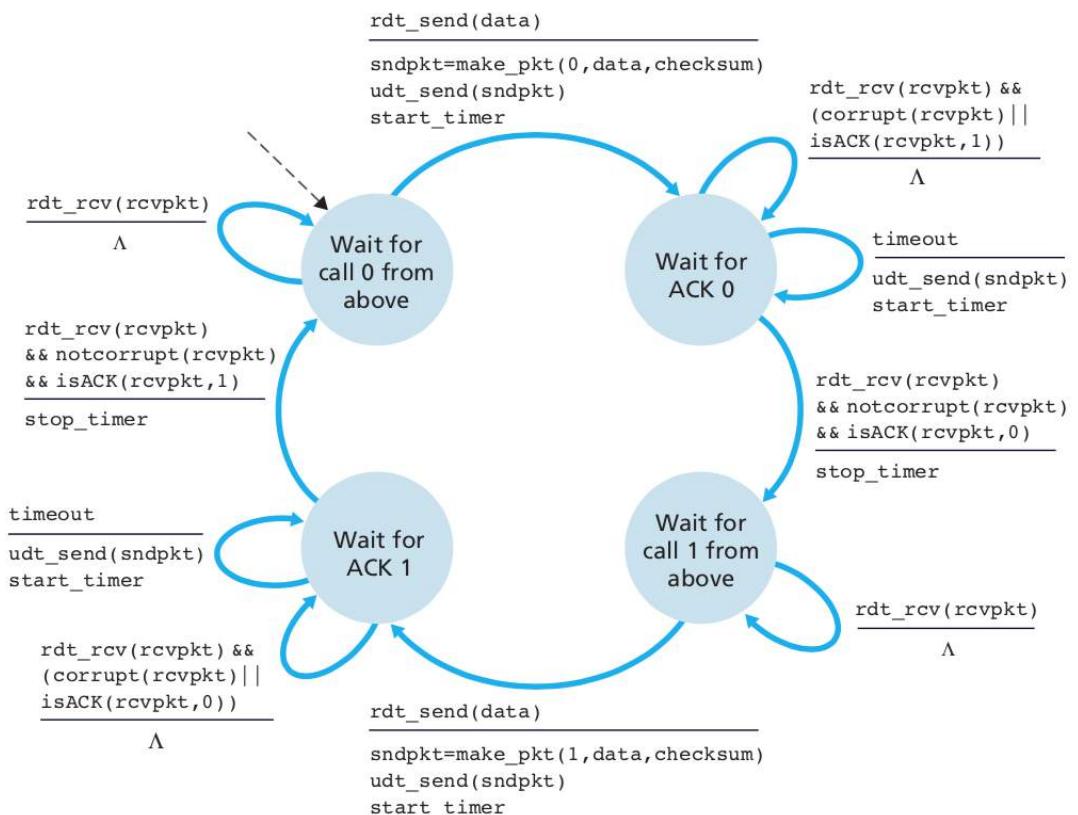
**Figure 3.12** ♦ rdt2.1 receiver

Dersom vi legger til sekvensnummeret til pakken som blir ack-et, kan vi få en NAK-fri pålitelig dataoverføringsprotokoll.

#### Pålitelig dataoverføring over en tapende kanal med bitfeil: rdt3.0

Dersom en pakke ikke kommer frem i det hele tatt, så må senderen re-sende pakken. Dersom man implementerer en timer, som re-sender pakken når nedtellingstimeren slår ut.

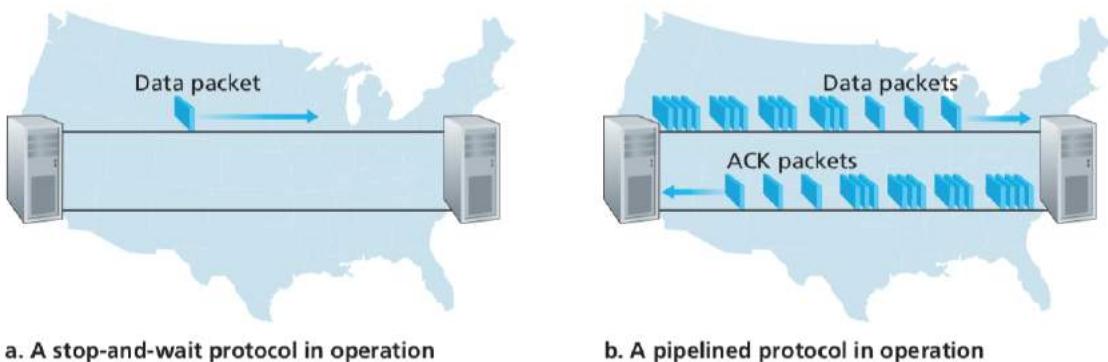
En slik protokoll der sekvensnummeret alternerer mellom 0 og 1 er også kjent som *alternating-bit protocol*



**Figure 3.15** ♦ rdt3.0 sender

#### Pipelinet pålitelig dataoverføringsprotokoller

Protokoll rdt3.0 er en funksjonell korrekt protokoll, men det er lite sannsynlig at noen vil være fornøyd med ytelsen dens. Slike stop-and-go-protokoller vil bruke mye døttid på å vente, og det vil gi forsinkelser.

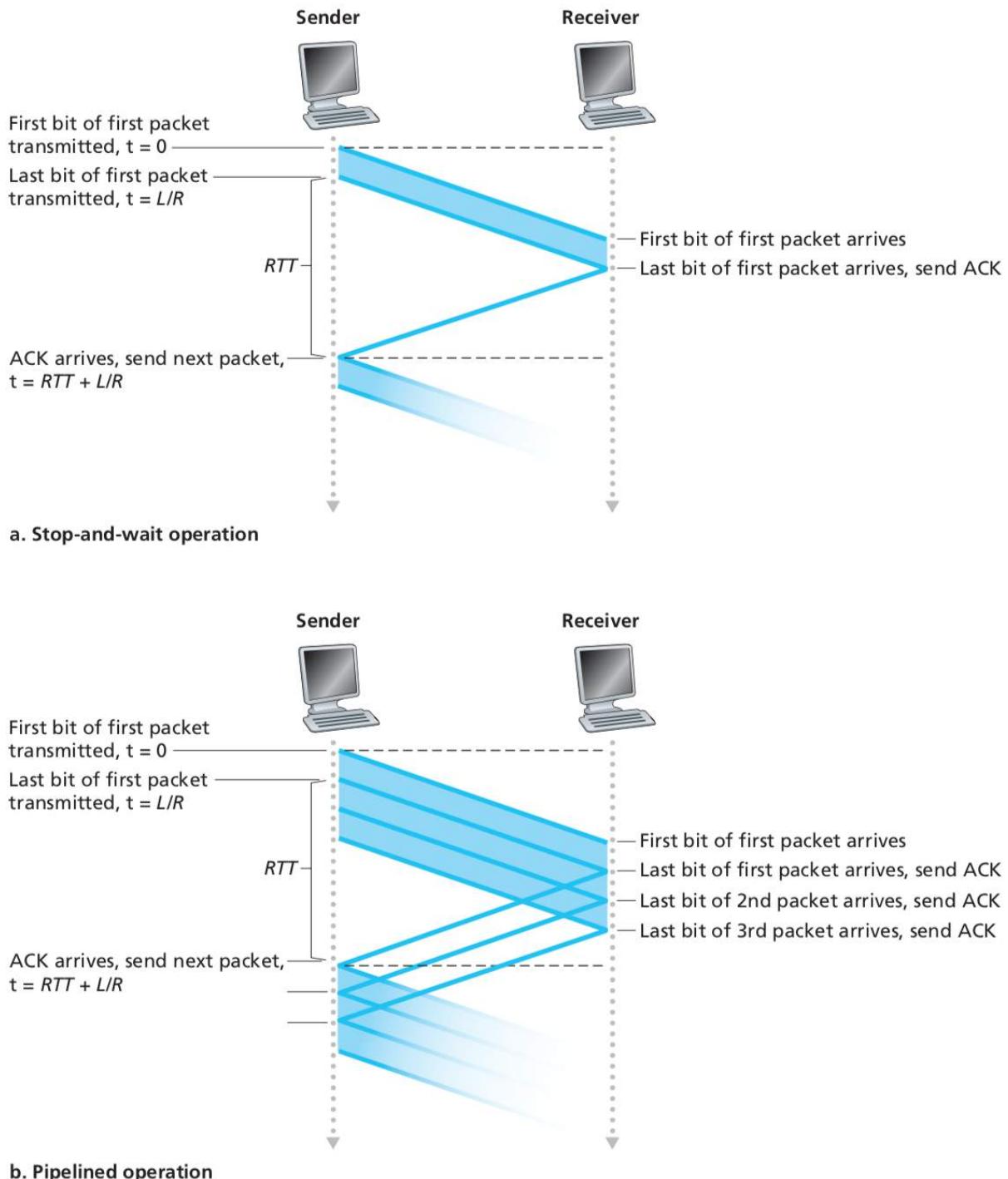


**Figure 3.17** ♦ Stop-and-wait versus pipelined protocol

Løsningen på dette problemet er ganske enkel. Istedet for å operere på en stop-and-go-metode, vil senderen tillate å sende flere pakker, uten å vente på bekrefteelse, illustrert over. Denne teknikken kalles **pipelining**. Pipelining har følgende konsekvenser for pålitelige dataoverføringsprotokoller:

- Utvalget av sekvensnumre må økes, siden hver transittpakke (ikke teller re-sendinger) må ha et unikt sekvensnummer og det kan være flere, uoverskådete in-transit-pakker.
- Avsender- og mottakersiden av protokollene må kanskje lagre mer enn én pakke. Minst må avsenderen bufferere pakker som er overført, men ikke bekreftet. Buffering av riktig mottatt pakke kan også være nødvendig hos mottakeren, som beskrevet nedenfor.
- Antall sekvensnumre som kreves og bufferingskravene vil avhenge av måten en dataoverføringsprotokoll svarer til tapte, korrupte og altfor forsinkede pakker. To grunnleggende tilhærminger mot pipelined feilgjenopprettning kan

identifiseres: Go-Back-N og selektiv gjentakelse.



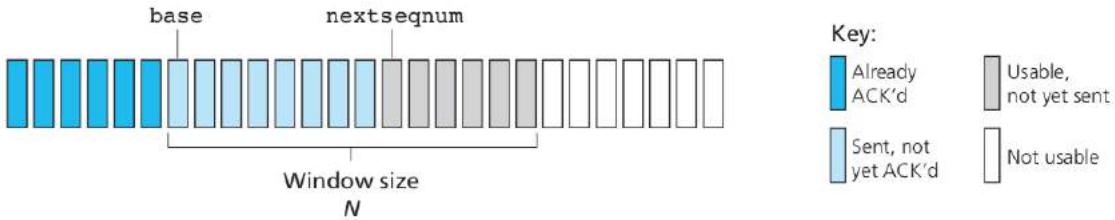
**Figure 3.18** ♦ Stop-and-wait and pipelined sending

#### Go-Back-N (GBN)

I en Go-Back-N (GBN) protokoll har senderen ov til å sende flere pakker uten å vente på bekreftelse (ACK), men er begrenset til å ikke ha mer enn et maksimalt lovlig nummer, N, av unacknowledged pakker i pipelinien.

Dersom vi definerer `base` til å være sekvensnummeret til den eldste unacknowledgede pakken, og `nextseqnum` til å være det minste ubrukte sekvensnummeret (dvs. sekvensnummeret til neste pakke som skal sendes). Da har vi at:

- $[0, \text{base} - 1]$  - er alle sendte og bekreftede pakker
- $[\text{base}, \text{nextseqnum} - 1]$  - er alle sendte men ikke bekreftede pakker
- $[\text{nextseqnum}, \text{base} + N - 1]$  - er alle usendte pakker, som er i vinduet
- $[\text{base} + N, \infty)$  - er alle pakker som ikke er sendt som er utenfor vinduet



**Figure 3.19** ♦ Sender's view of sequence numbers in Go-Back-N

Som vist i Figur 3.19 kan serien av tillate sekvensnumre for sending, men som ikke er blitt bekreftet, sees på som et vindu av størrelse  $N$  over serien av sekvensnumre. Av denne grunn blir  $N$  ofte referert til som *vindustørrelsen* og GBN-protokollen selv en **sliding-window protocol**.

GBN-senderen må kunne respondere til tre typer hendelser:

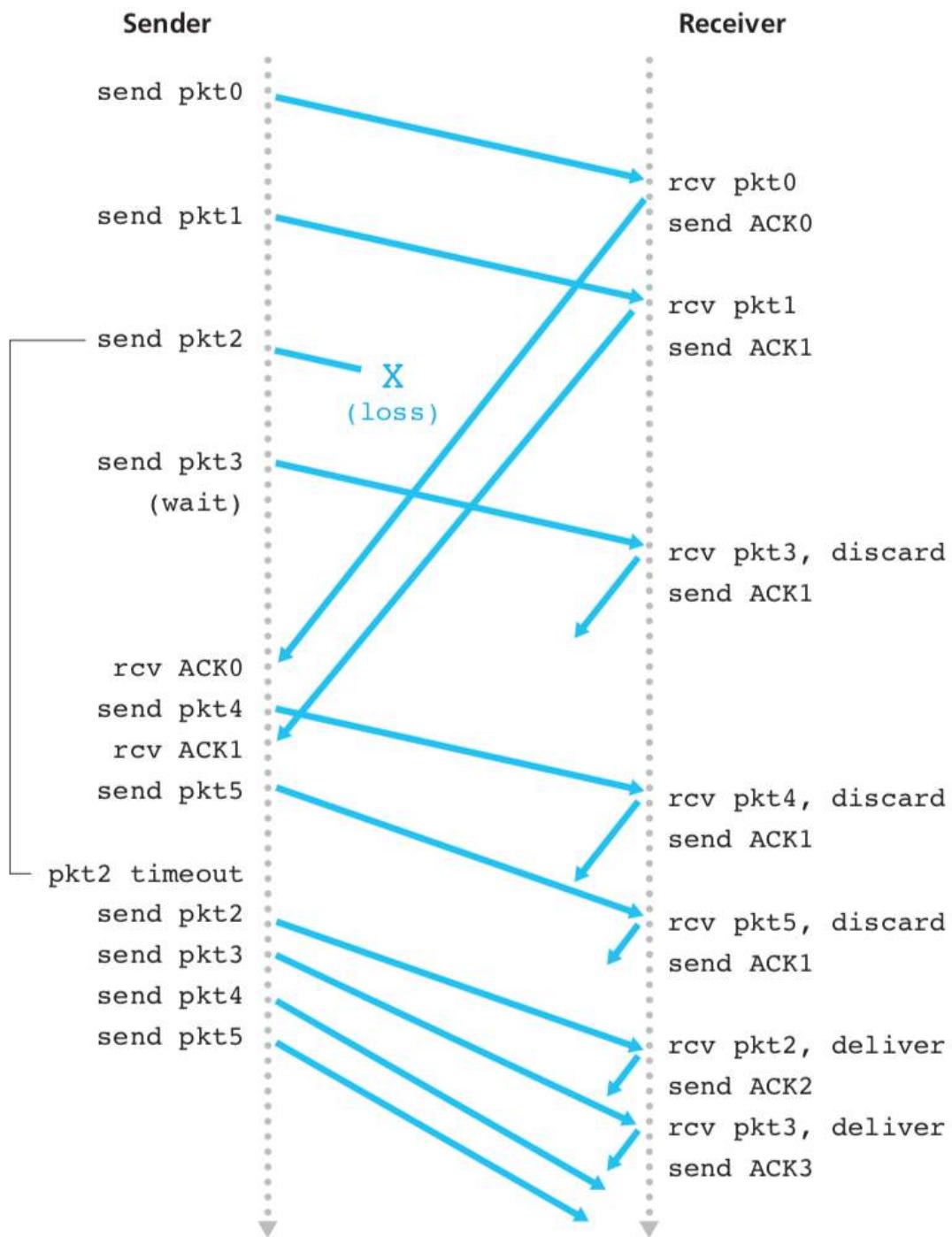
- *Påkallelse (invocation) ovenfra.* Når `rdt_send()` kalles ovenfra, må senderen først sjekke om vinduet er fullt, det vil si om det er  $N$  utestående, ubekreftede pakker. Dersom vinduet ikke er fullt, kan det sendes en pakke. Om det er fullt må senderen si ifra til det øvre laget, og indikere at vinduet er fullt.
- *Mottak av ACK.* Mottak av en ACK. I vår GBN-protokoll vil en bekreftelse for en pakke med sekvensnummer  $n$  bli tatt til å være en kumulativ bekreftelse, noe som indikerer at alle pakker med et sekvensnummer opp til og inklusive  $n$  er blitt mottatt korrekt på mottakeren.
- *En timeout-hendelse.* Protokollens navn, "Go-Back-N", som i stop-and-go-protokollen har GBN-protokolen er nedteller. Når en timeout skjer, vil senderen \*re-sende alle pakker som har blitt sendt og ikke er bekreftet.

Kummulativ acknowledgement er naturlig for GBN, da mottakeren kun godkjenner pakker når de kommer i riktig rekkefølge. Alle pakker som ikke kommer i rekkefølge vil bli forkastet.

Dersom pakke  $n$  ikke er mottatt og pakke  $n+1$  kommer, vil pakken forkastes, og mottakeren vil sende bekreftelse for pakke  $n-1$

### Selective Repeat (SR)

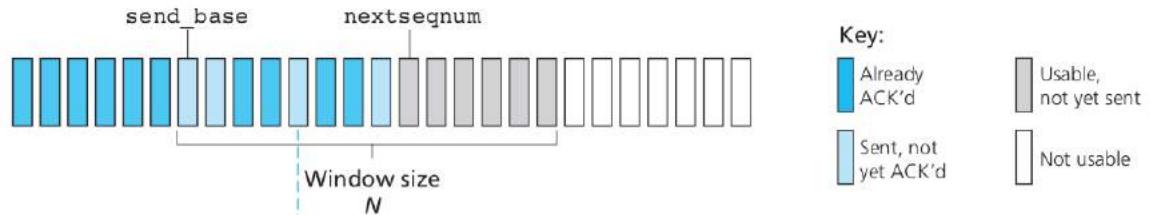
GBN-protokollen lar senderen potensielt "fylle pipelinen" med pakker. Men det er noen scenarioer der GBN har ytelsesfeil. Spesielt når vindustørrelsen og båndbreddeforsinkelsen er høye. En enkelt pakkefeil kan dermed sørge for at et stort antall pakker på sendes på nytt. Slik ser GBN ut dersom den må re-sende pakker.



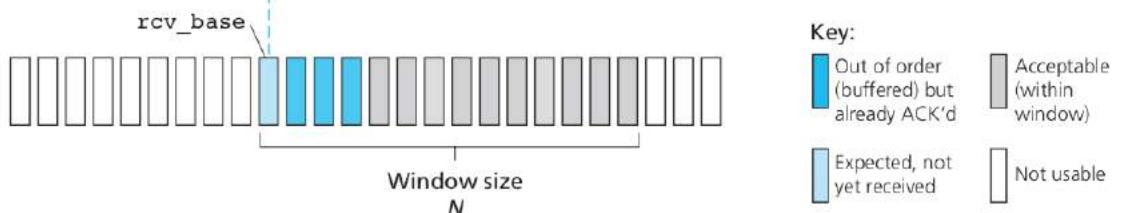
**Figure 3.22 ♦ Go-Back-N in operation**

Selective Repeat, som navnet hentyder, unngår re-sending av pakker som den tror har opplevd feil hos mottakeren.  
Mottakeren må godkjenne individuelt godkjente mottatte pakker

En vindustørrelse på  $N$  vil igjen bli brukt til å begrense antall utestående, ubekrefte pakker i pipelinen. Men i motsetning til GBN, har avsenderen allerede mottatt ACKs for noen av pakkene i vinduet.



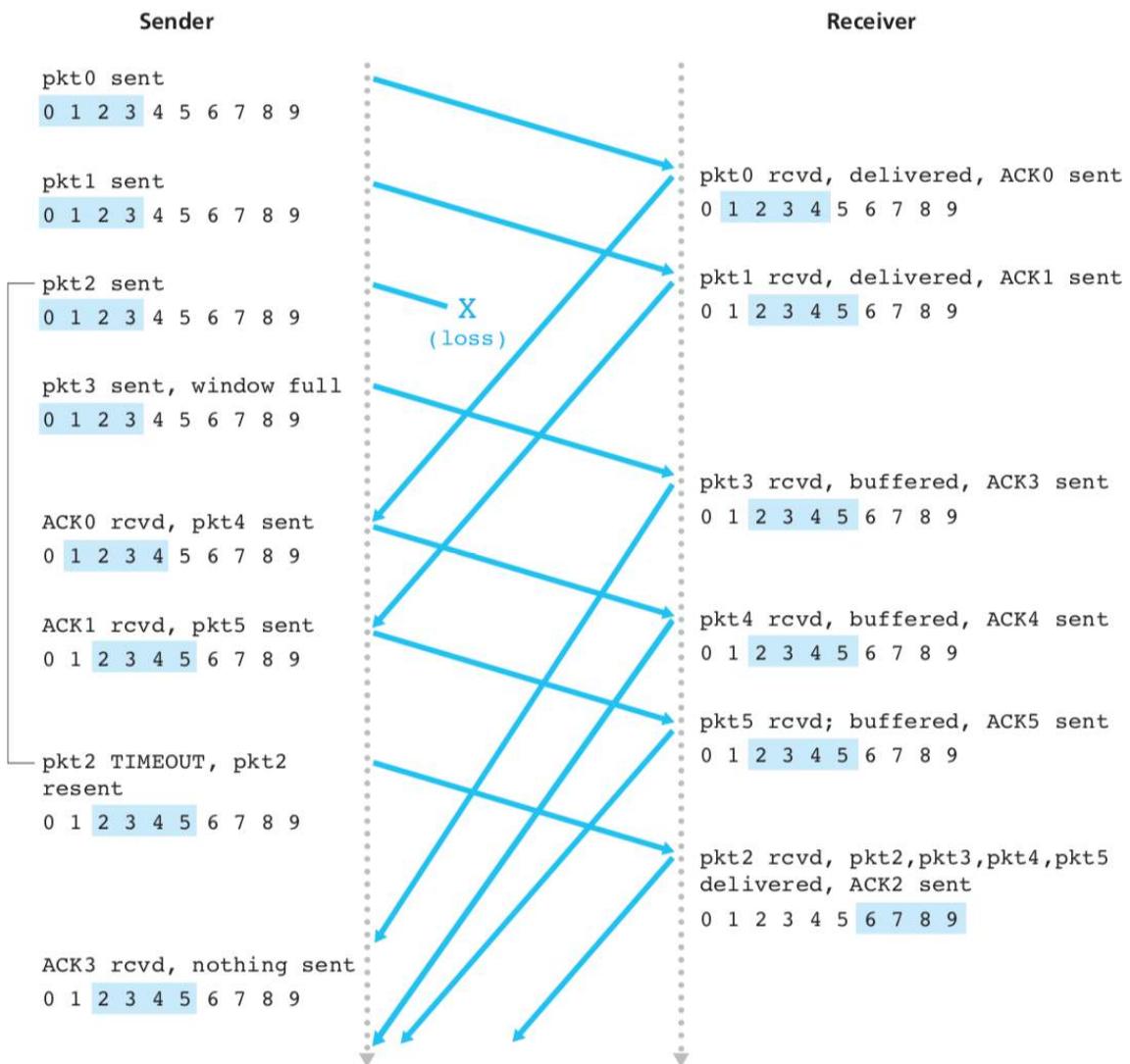
a. Sender view of sequence numbers



b. Receiver view of sequence numbers

**Figure 3.23** ♦ Selective-repeat (SR) sender and receiver views of sequence-number space

Pakker som ankommer i feil rekkefølge, vil bli buffret til alle manglende pakker (med lavere sekvensnummer) ankommer.



**Figure 3.26** ♦ SR operation

## Connection-Oriented Transport: TCP

### TCP-koblingen

TCP er sagt være **koblingsorientert** da en applikasjonsprosess kan sende data til en annen, må de to prosessene "handshake" med hverandre - som betyr å sende hverandre noen segmenter for å fastslå noen parametere for den kommende dataoverføringen. Begge parter må initialisere flere variabler for tilkoblingen.

Ettersom TCP-protokollen kun kjører på endesystemene og ikke i mellomliggende nettverkselementer (rutere m.m.), er de mellomliggende nettverkselementene helt uvitende om TCP-tilkoblingen - de ser datagrammer, ikke tilkoblinger.

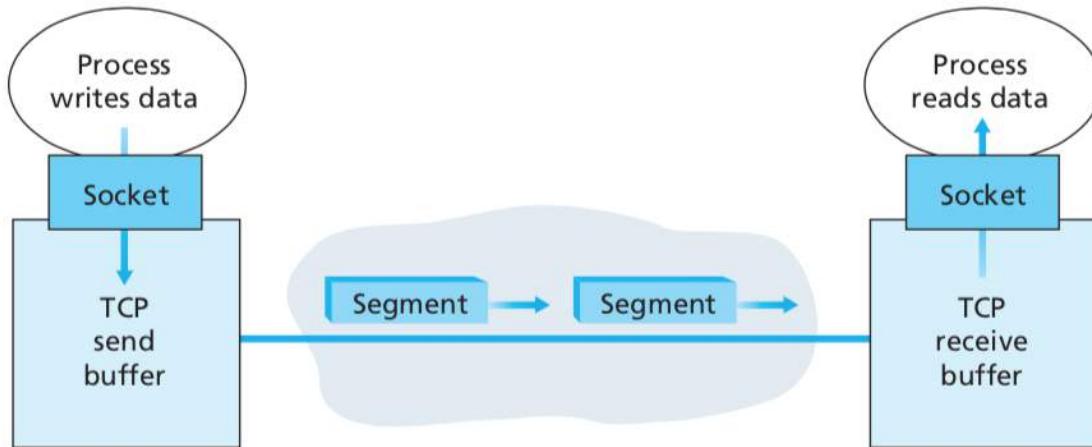
En TCP-kobling tilbyr en **fullstendig** (eng. *full-duplex*) tjeneste: Dersom det er en TCP-kobling mellom Prosess A hos en vert, og Prosess B hos en annen vert, da kan applikasjonlagsdata flyte fra A til B samtidig som applikasjonlagsdata kan flytte fra B til A.

En TCP-kobling er alltid **punkt-til-punkt**, det vil si mellom én sender og én mottaker.

Når en TCP-kobling er etablert kan de to applikasjonsprosessene begynne å sende data til hverandre. La oss se på det å sende data fra klientprosessen til serverprosessen:

- Klientprosessen overfører en datastrøm gjennom socketen sin
- Når dataene passerer gjennom døren, er dataene i hendene på at TCP kjører hos klienten. Som vist i Figur 3.28, styrer TCP disse dataene til koblingens **sende-buffer**, som er en av bufferne som settes til side under det første treveis håndtrykket.

- Fra tid til annen tar TCP noe data fra *sende-bufferen* og sender dataene til nettverkslaget.
  - Interessant er TCP-spesifikasjonen veldig avslappet tilbake når det gjelder å spesifisere når TCP faktisk skal sende bufferdata, og angir at TCP skal "sende dataene i segmenter på egen vilje."
- Maksimal mengde data som kan bli tatt av TCP og plassert i et segment er begrenset av **maksimal segmentstørrelse (MSS)**.
  - MSS er typisk satt ved å først se på den største datalinklags-rammen som kan bli sendt av klienten (såkalt **maximum transmission UNIT, MTU**).



**Figure 3.28** ♦ TCP send and receive buffers

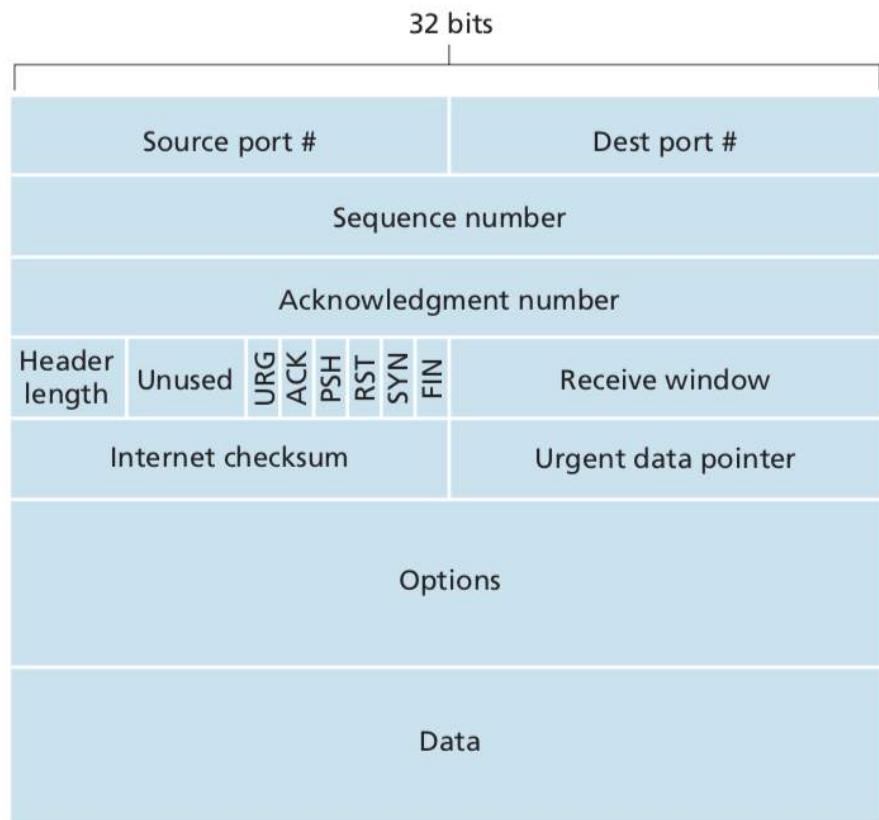
- TCP parrer hver klump med data med en TCP-header, og dermed former **TCP-segmenter**. Disse segmentete blir videre gitt til nettverkslaget, som lager IP-datagrammer.
- Når TCP mottar et segment fra det andre endesystemet, vil segmentets data bli plassert i TCP-koblingens **mottakerbuffer**, som vist i Figur 3.28 over.
  - Hver side av tilkoblingen har en *sende- og mottaker-buffer*.

TCP bruker ikke NAK!

#### TCP Segment Structure

TCP-segmentet består av header-felt og et data-felt. Sånn som UDP-segmentet har TCP **source og destination port numbers**, som blir brukt til multipleksing og demultipleksing, og også et **checksum-felt**. Et TCP-segment har også følgende felter:

- Den har et 32-bit **sekvensnummer-felt** og et 32-bit **acknowledgement number**, som blir brukt av TCP senderen og mottakeren for å implementere en pålitelig dataoverføringstjeneste.
- 16-bit **receive window-felt** som brukes til flytkontroll.
- 4-bit **header-lengt field** som spesifiserer lengden av TCP-headeren i 32-bit ord. Kan være variabel lengde, pga. TCP options-feltet.
- Det valgfrie og variabel-lengde **options field-feltet** brukes når sender og mottaker avtaler den maksimale segmentstørrelsen (MSS) eller for en vinduskaleringsfaktor eller for tidsstempeling.
- **Flag field** feltet består av 6 bits. **ACK**-biten indikerer om verdien i acknowledgement-feltet er gyldig. **RST**, **SYN**, og **FIN** bitene er brukt for koblingstabllering og ødeleggelse. **PSH**-biten indikerer at mottakeren skal sende dataen til det øvre laget umiddelbart. Til sist, **URG**-biten brukes for å indikere at det er data i segmentet som senderen har markert som *urgent / haster*. Pekeren til denne *urgente* dataen er i det 16-bits feltet **Urgent data pointer**.

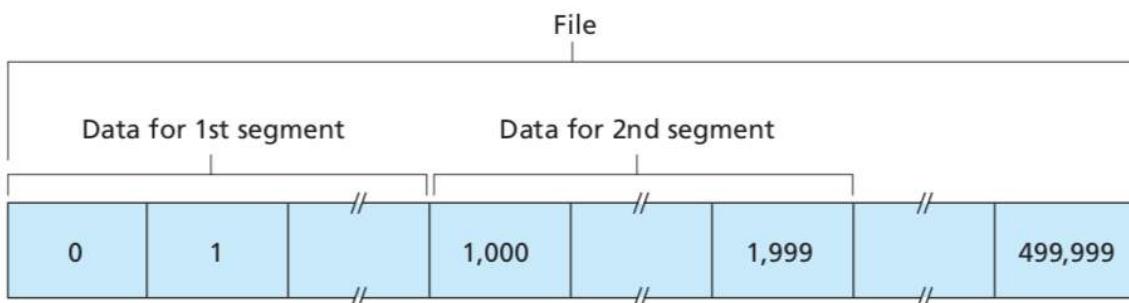


**Figure 3.29** ♦ TCP segment structure

#### Sequence Numbers and Acknowledgement Numbers

De to viktigste feltene i et TCP-segment er sekvensnummer-feltet og acknowledgementnummer-feltet. Disse er kritiske for å kunne tilby pålitelig dataoverføring.

Sekvensnummeret til et segment er bytestrøm-nummeret til den første byten i segmentet. For eksempel dersom man skal sende en fil på 500,000 bytes, og MSS er 1,000 bytes, og den første byte med datastrøm være nummerert med 0. Slik vil man da dele opp filen i TCP-sementer.

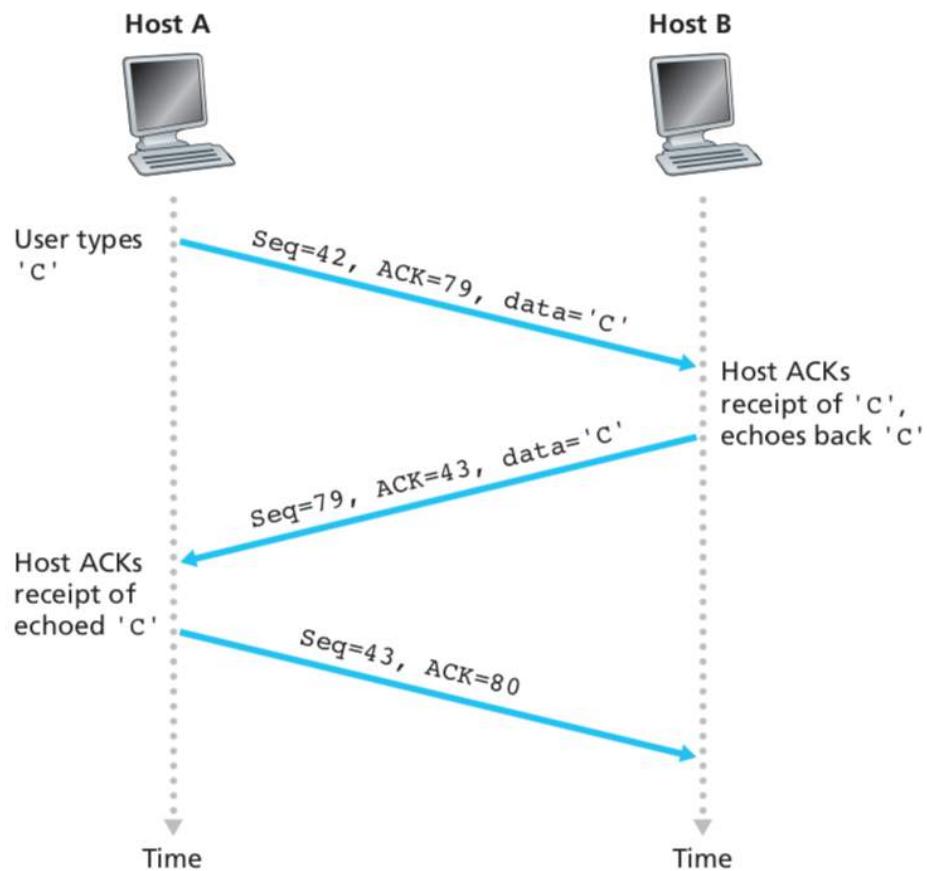


**Figure 3.30** ♦ Dividing file data into TCP segments

- Sekvensnummeret var bytestrøm-nummeret til første byte i dataen til segmentet. For første segment som sendes, vil sekvensnummeret være **0**. For andre segment vil det være **1000**, osv.
- Acknowledgementnummeret er sekvensnummeret til den neste byten som mottakeren forventer å få fra senderen.

Så når segment 1 med sekvensnummer 0 er mottatt vil mottakeren sende ACK 1000, da byte med bytestrømnummer 1000 er neste byte som forventes.

Telnet er en populær applikasjonlagsprotokoll som blir brukt for ekstern pålogging. Den kjører over TCP, og er designet for å jobber mellom ethvert par av verter. Telnet er en interaktiv applikasjon. Mange bruker nå idag SSH istedet for Telnet. Data sendt i Telnet er ikke kryptert og er derfor usikkert. Under er en figur av to verter som bruker Telnet, og vi ser at dataen blir ekkoet tilbake til klienten.



**Figure 3.31** ♦ Sequence and acknowledgment numbers for a simple Telnet application over TCP

#### Round-Trip Time Estimation and Timeout

TCP bruker en timeout/re-sendingmekanisme for å redde tapte/mistede segmenter. Timeouten skal klart være større enn koblingens **round-trip time (RTT)**, som er hvor lang tid det tar fra pakken er sendt til den er acknowledged.

##### Estimere Round-Trip Time:

Det er vanlig å lage starte med en *SampleRTT* på TCP-koblingen, og deretter kalle den som det hittil gjennomsnittet som blir *EstimatedRTT*, når man får en ny *SampleRTT*, vil TCP oppdatere *EstimatedRTT* med følgende formel:

- $\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}$

Den anbefalte verdien for  $\alpha$  er  $\alpha = 0.125$

Vi har også en variabel for variansen til RTT, som kalles *DevRTT*, som estimerer hvor mye *SampleRTT* typisk avviker fra *EstimatedRTT*

##### Sette og endre Timeout intervallet for re-sending

Gitt verdiene *EstimatedRTT* og *DevRTT* kan man regne ut hvilken verdi TCP sin timeout intervall skal være. Det er derfor vanlig å sette *TimeoutInterval* til å være:

- $\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$

En initiell *TimeoutInterval*-verdi på 1 sekund er anbefalt

Dersom et timeout finner sted, dobles verdien til *Timeout/Interval* for å unngå for tidlig timeout. Men så fort segmentet blir bekreftet, regnes *EstimatedRTT* ut på nytt med formelen over.

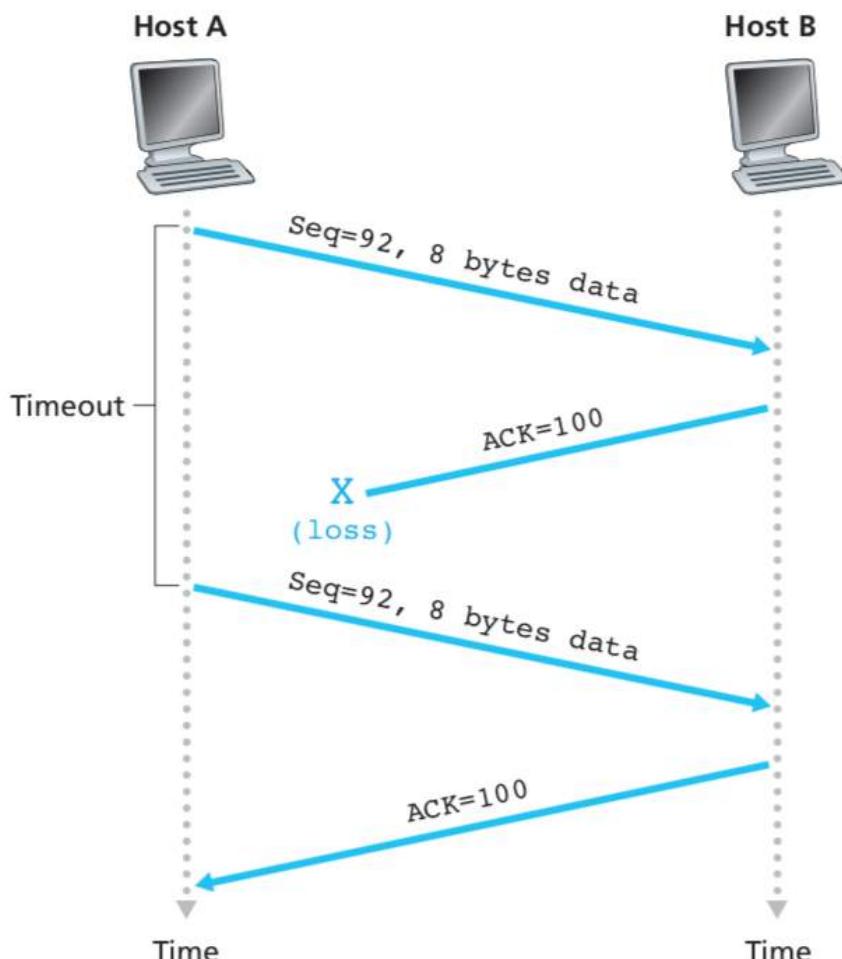
#### Pålitelig dataoverføring

Husk at Internettets nettverkstjeneste (IP) er upålitelig. IP gir ingen garanti for datagram-leveranse, garanterer ikke in-order leveranse og garanterer heller ikke integriteten til dataen i datagrammene.

TCA lager en pålitelig dataoverføringstjeneste på toppen av IP sin upålige, best-effort tjeneste. TCPs rdt sørger for at datastrømmen som prosessen leser ut fra TCP-mottaker-bufferen er ukorrup, uten hull, uten duplisering, og i riktig rekkefølge.

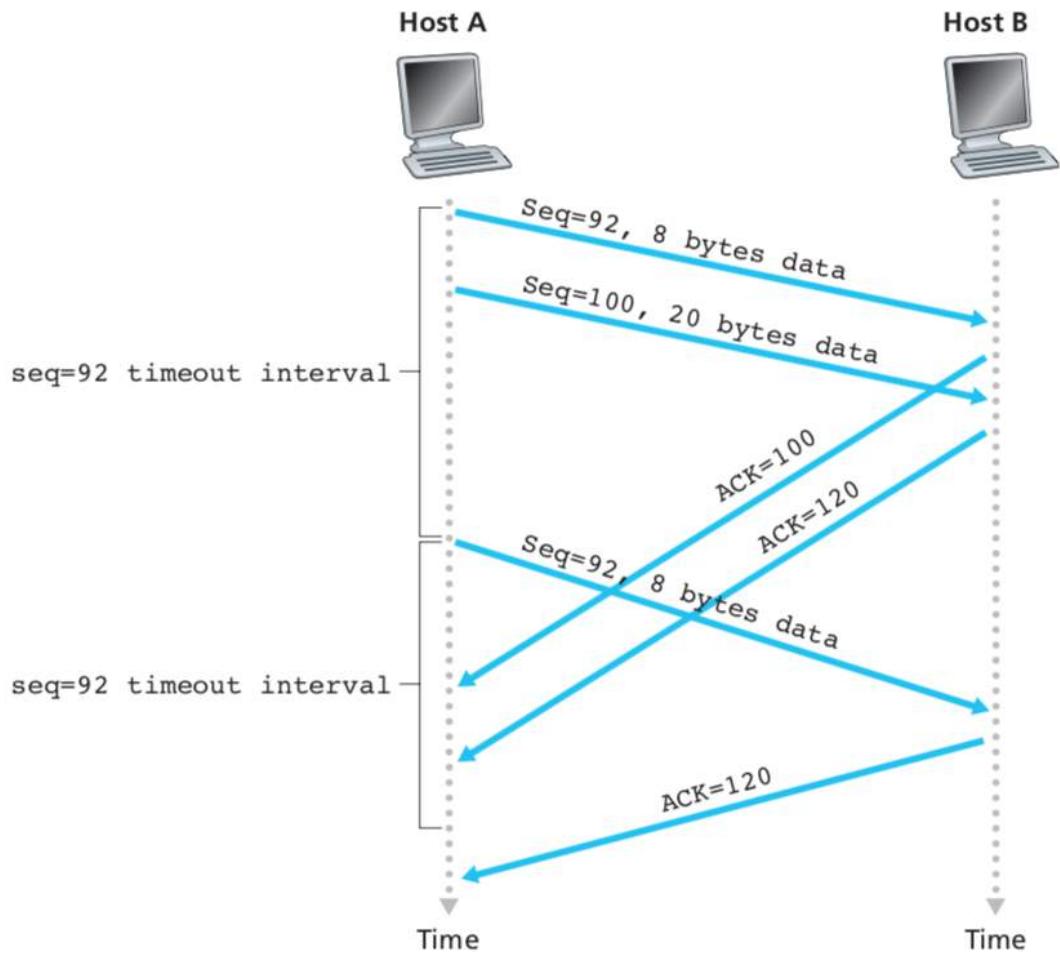
For å få til dette brukes mange av tingene som ble snakket om i forrige delkapittel. Tidligere har det vært diskutert om man skal ha en timer eller en per segment. Sistnevnte ville ført til stort overhead, og det anbefalte TCP timer-managmentet er å ha en *enkel* re-sendingstimer.

Et par scenarioer:



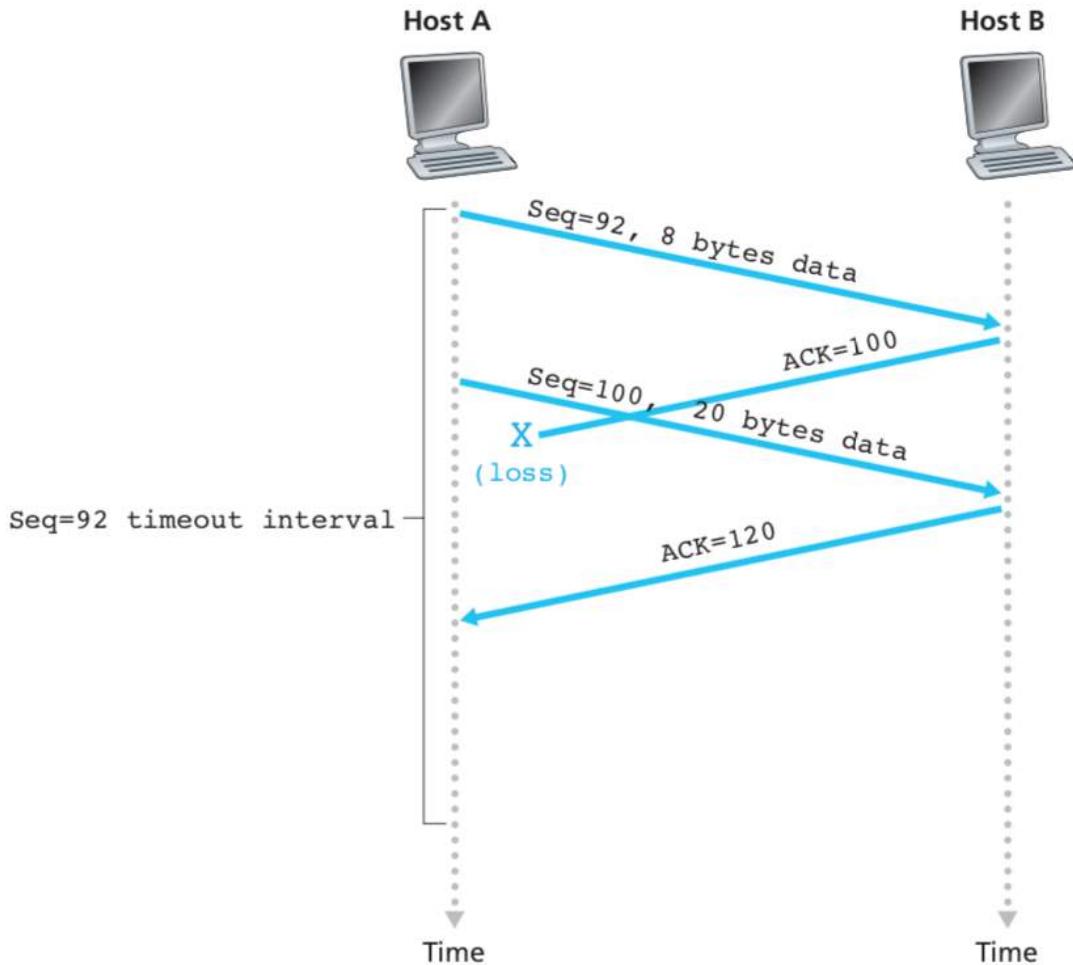
**Figure 3.34** ♦ Retransmission due to a lost acknowledgment

Her blir segment med sekvensnummer 92 sendt to ganger siden ACK-en ikke kom frem.



**Figure 3.35** ♦ Segment 100 not retransmitted

Her kommer ingen av ACK-ene frem før timeout, men andre ACK kommer før timeout til andre segment og blir ikke resendt.



**Figure 3.36** ♦ A cumulative acknowledgment avoids retransmission of the first segment

Her kommer ikke ACK=100 frem, men det gjør ACK=120, før timeout intervallet, og pga. kummulativ bekreftelse trenger ikke TCP-senderen og re-sende første segment.

**Timeout interval doubling:** Når en timeout skjer, så dobles som nevnt timeout intervallet hver gang timeout skjer og regnes ut på nytt når første ACK mottas. Dette er en begrenset form for congestion control.

**Fast Retransmit:** Ett av problemene med timeout-baserte re-sendinger er at timeoutperioden kan være relativt lang. Når et segment mistes tvinger denne lange timeout-perioden senderen til å forsinke/vente med å re-sende det tapte segmentet, og dermed øke ende-til-ende-forsinkelsen.

Heldigvis kan senderen ofte detektere tapte pakker lenge før timeout, pga noe som kalles **duplicate ACK-er**. En *duplicate ACK* er en ACK som re-bekrefter et segment som senderen allerede har mottatt en ACK for.

- En mottaker merker at den mangler et segment dersom den mottar et segment med sekvensnummer høyere enn forventet, og sender dermed en duplikat ACK for forrige mottatte pakke.
- Når sender mottar en duplikat ACK, vil den legge til antall ACKs mottatt for den pakken som mottakeren sier den bekrefter. Dersom denne verdien blir større enn 3, i dette tilfelle, vil TCP-senderen utføre en **fast retransmit**, og sender pakken *før* segmentets timer går ut.

#### Go-Back-N eller Selective Repeat?

Husk at TCP-Acknowledgements er kummulative og mottatt riktig mottatt men out-of-order segmenter er ikke individuelt ACK-et av mottakeren, og ligner kanskje gjerne på en GBN-type protokoll. Men det er mange forskjeller, som at TCP vil buffre riktig mottatt med out-of-order segmenter. Dersom en pakke  $n$  forsvinner, vil GBN re-sende alle pakkene  $n, \dots, N$  fra vinduet, mens TCP kun vil sende pakke  $n$ .

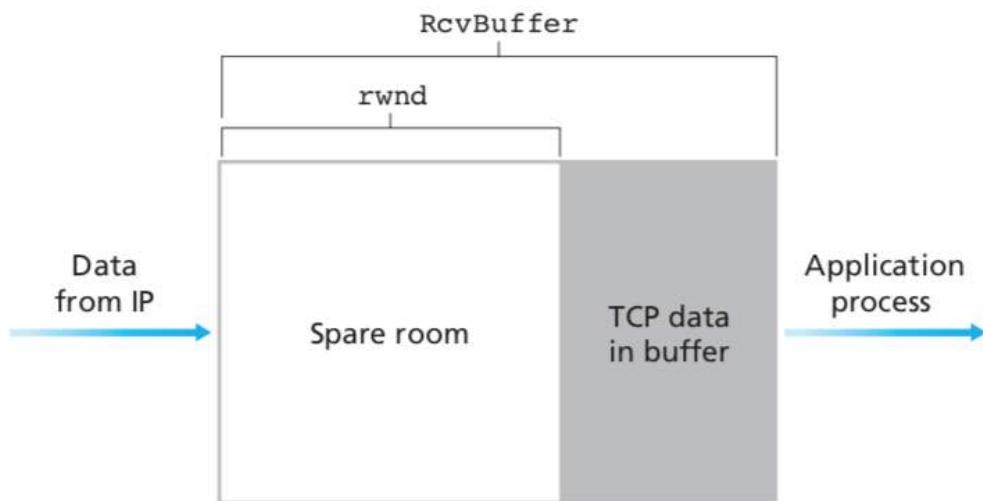
En foreslått modifisering av TCP, er den såkalte **selective acknowledgement**, som tillater TCP-mottakeren å bekrefte out-of-order segmenter, og TCP sammen med selective ACK vil se veldig ut som den generiske SR protokollen.

Men, TCPs feilsøkingsmekanisme er trolig best kategorisert som en hybrid av GBN og SR protokoller.

### Flytkontroll

Minner om at hver side av en TCP-tilkobling har en mottaker-buffer for koblingen. Dersom applikasjonen som leser data fra denne er relativt treg på å lese data, kan senderen lett *overflowe* denne bufferen ved å sende for mye data for fort.

TCP tilbyr en **flytkontroll-tjeneste** (eng. *flow control service*) ved å eliminere muligheten for at senderen overflorer mottakerens buffer. Dette gjøres ved at senderen vedlikeholder en variabel kalt **receive window**. Dette vinduet er brukt for å gi senderen en idé av hvor mye fri bufferplass som er ledig hos mottakeren. Begge vertene har hvert sitt *mottakervindu*.



**Figure 3.38** ♦ The receive window (*rwnd*) and the receive buffer (*RcvBuffer*)

- $rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]$ 
  - *rwnd* - ledig plass i buffer
  - *RcvBuffer* - total plass i buffer
  - *LastByteRcvd* - bytestrømnummeret på byten som ble sist mottatt fra nettverket
  - *LastByteRead* - bytestrømnummeret på byten som ble sist lest fra bufferen.

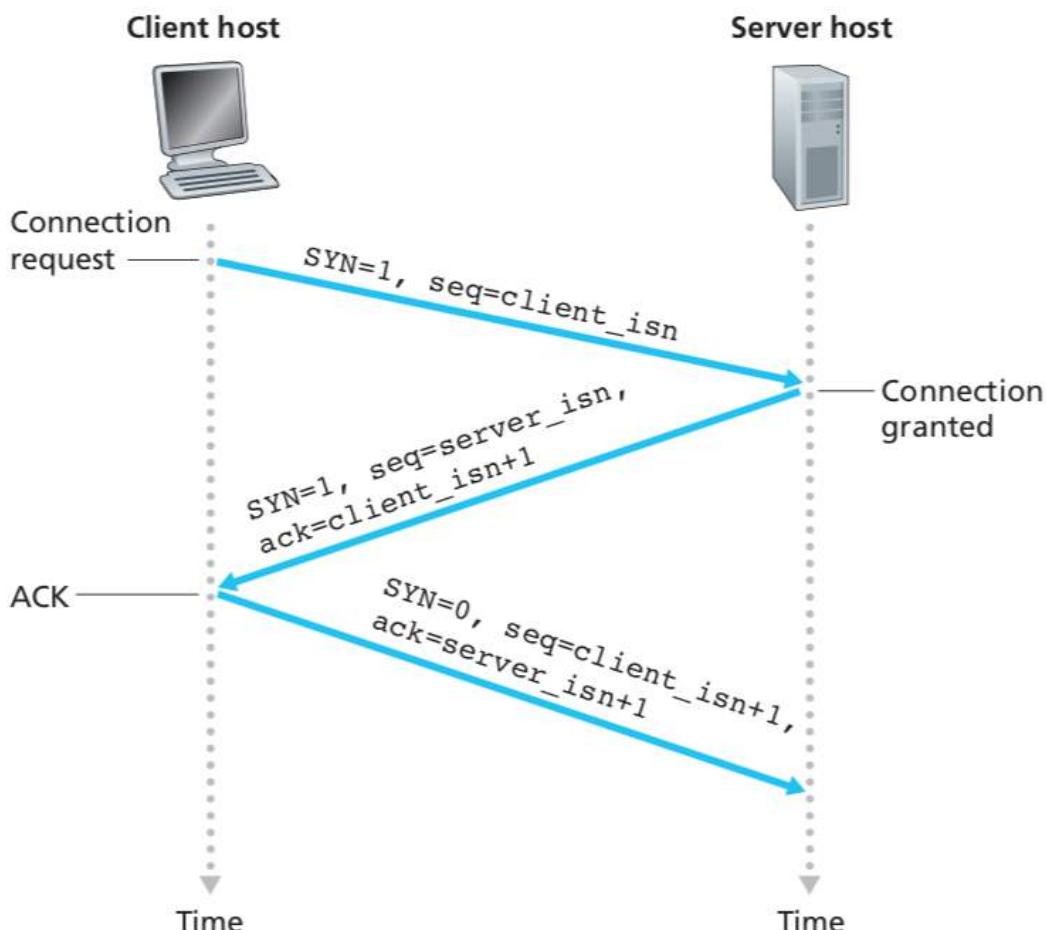
### TCP-tilkoblingsadministrasjon

Nå skal vi se på hvordan en TCP-tilkobling etableres og rives ned. Vi skal først se på hvordan man etablerer en TCP-tilkobling. Anta at en prosess på en vert (klient) ønsker å initialisere en tilkobling med en annen prosess på en annen vert (server). Applikasjonsprosessen hos klienten informerer klient TCP-en at den ønsker å etablere en tilkobling til en prosess på serveren. TCP-en hos klienten begynner så å etablere tilkoblingen med TCP-en hos serveren på følgende måte:

1. Klientsidens TCP sender et spesielt TCP-segment til serversidens TCP. Dette spesielle segmentet inneholder ingen applikasjonsdata, men av flagg-bitene i segmentets header, **SYN-biten**, er satt til 1. I tillegg velger klienten et tilfeldig start-sekvensnummer (*client\_isn*), og plasserer det i sekvensnummerfeltet på den initielle TCP SYN-segmentet, som sendes til serveren.
2. Når IP-datagrammet med TCP SYN-segmentet kommer hos serveren, ekstraherer serveren segmentet fra datagrammet, allokkerer TCP-buffere og variabler for koblingen. Den sender så et *connection-granted-segment* til client TCP-en. Den setter **SYN-flagget** til 1, og acknowledgemenumberet til å være *client\_isn + 1*. Serveren velger også sitt eget initielle sekvensnummer (*server\_isn*). *Connection-granted-segmentet* refereres til som et **SYNACK-segment**.

- Når klienten mottar SYNACK-segmentet, allokerer også klienten buffere og variabler til tilkoblingen. Klienten sender serveren så igjen et annet segment, som bekrefter SYNACK-segmentet. Setter acknowledgemenumberet til  $server\_isn + 1$  i TCP-headeren, og *SYN-biten* til 0, siden tilkoblingen er etablert.

For å kunne etablere tilkobling mellom to verter, må det sendes tre pakker, og av den grunn kalles denne etableringsprosedyren for en **three-way handshake**

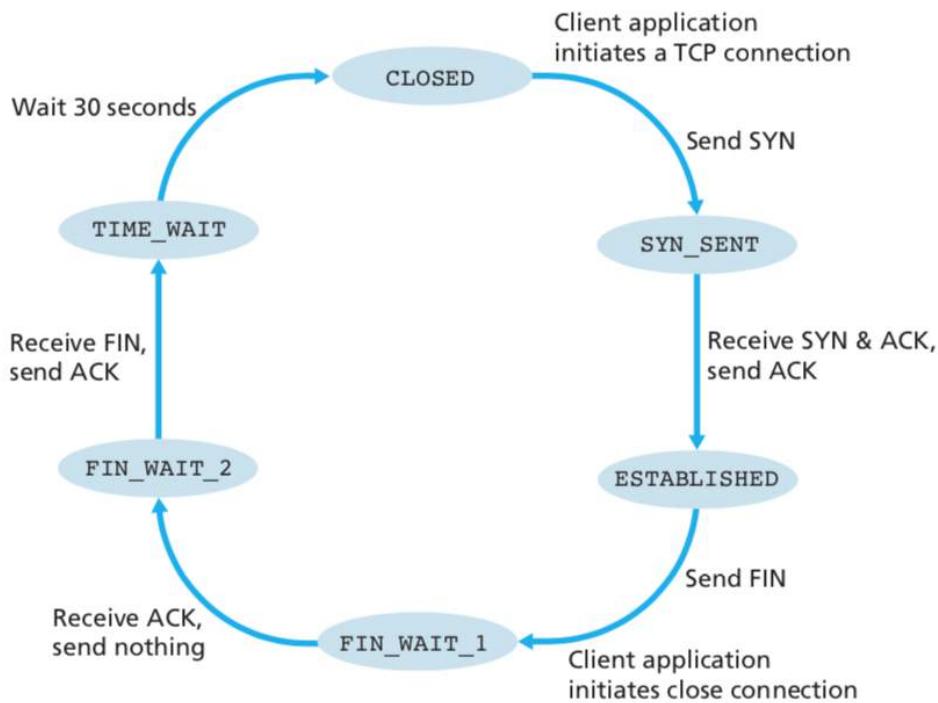


**Figure 3.39** ♦ TCP three-way handshake: segment exchange

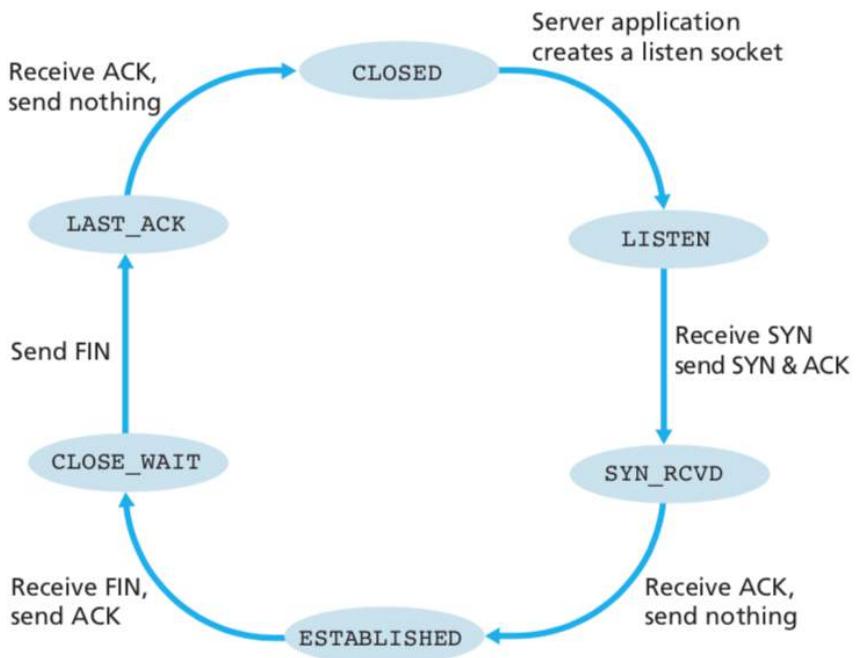
Dersom klientapplikasjonen ønsker å lukke koblingen (noter at server også kan lukke kobling). Da skjer følgende:

- Klienten sender TCP-segment med *FIN-bit* satt til 1. Klienten går i FIN\_WAIT\_1-tilstanden.
- Serveren bekrefter TCP-segmentet fra klienten.
- Når klienten får dette segmentet går den inn i neste tilstand, FIN\_WAIT\_2. Den venter på et segment fra serveren med *FIN-biten* satt til 1.
- Server sender TCP-segment med *FIN-bit* satt til 1. Klient acknowledge dette segmentet. Klienten går i TIME\_WAIT-tilstanden i tilfelle segmentet ikke kom frem.

Gjennom livet til en TCP-koblingen, har koblingen flere **TCP-tilstander**. Her er typiske tilstander en TCP-tilkobling har i levetiden sin:



**Figure 3.41** ♦ A typical sequence of TCP states visited by a client TCP



**Figure 3.42** ♦ A typical sequence of TCP states visited by a server-side TCP

### Congestion Control - Approaches to Congestion Control

Vi skal se på to store tilnæringer for congestion control, som er blitt tatt i bruk og diskutere spesifikke nettverksarkitekture og congestion-control protokoler som bruker disse metodene.

På det bredeste nivået kan vi skille mellom fremgangsmåter for congestion control ved om nettverkslaget gir noen eksplisitt hjelp til transportlaget for overbelastningsstyringsformål:

- *Ende-til-ende congestion control.* I en ende-til-ende tilnærming for congestion control, gir nettverkslaget *ingen eksplisitt støtte* til transportlaget for congestion-control-formål. Om det er overbelastning i nettverket eller ikke må utledes av endesystemene, som bare er basert på nettverksadferd (pakketap og forsinkelse).
  - For eksempel på TCP ta denne ende-til-ende-tilnærmingen, siden IP-laget ikke gir noen tilbakemelding til endesystemene angående nettverksbelastning. TCP-segmenttap (timeout eller 3 \* duplikat ACK) er brukt som indikasjon for nettverksoverbelastning (*eng. network congestion*).
- *Nettverksassistert congestion control.* Med nettverksassistert congestion control (*eng. network-assisted*), gir nettverklagskomponentene (dvs. rutere) eksplisitt tilbakemelding til senderen angående nettverksbelastningen (*congestion*)
  - Denne tilbakemeldingen kan være så enkel som en enkelt bit som indikerer overbelastning ved en link

I nettverksassistert congestion control, sier ruteren ifra på en av to måter. Enten sende en **choke-pakke**, som essensielt sier at den er opptatt, eller å legge på en indikator på pakken, slik at mottakeren kan si ifra til senderen om at overbelastningen/congestion

## TCP Congestion Control

TCP må bruke ende-til-ende congestion control istedet for netwerk-assisted congestion control, siden IP-laget ikke tilbyr noen eksplisitt feedback til endesystemene angående nettverksbelastning.

Tilnærmingen til TCP er å ha hver avsender begrense hastigheten som den sender trafikk til forbindelsen sin, som en funksjon av oppfattet nettverksbelastning.

Vi så i forrige delkapittel at begge sidene av TCP-koblingen hadde buffere, og flere variabler (*LastByteRead*, *rwnd* m.m.). TCP congestion-control-mekanismen hos senderen holder på en ekstra variabel, **congestion window**.

Overbelastningsvinduet / congestion window, betegnet *cwnd*, setter en begrensning på hastigheten der en TCP-avsender kan sende trafikk til nettverket. Spesifikt kan mengden av ubekrefte data ved en avsender ikke overstige minimum *cwnd* og *rwnd*, det vil si

- *LastByteSent – LastByteAcked min{cwnd, rwnd}*

For å ikke fokusere på flytkontroll, så antar vi at mottakervinduet er så stort at størrelsen dens kan ignoreres. Dermed er mengden av ubekrefte data ved avsenderen kun begrenset av *cwnd*.

DVed begynnelsen av hver RTT, tillater begrensningen at avsenderen sender *cwnd* bytes data inn i forbindelsen. På slutten av RTT mottar avsenderen bekreftelser for dataene. Dermed er avsenderens sendehastighet omtrent:

- *cwnd bytes /RTT sec*
  - Ved å **justere** verdien av *cwnd*, kan avsenderen derfor **justere** frekvensen der den sender data til sin forbindelse.

La oss se nærmere på hvordan en TCP-avsender oppfatter at det er overbelastning på veien mellom seg selv og destinasjonen. La oss definere en "tapshendelse" på en TCP-avsender som forekomsten av enten en *timeout* eller mottak av *tre dupliserte ACKer* fra mottakeren.

Ettersom TCP bruker acknowledgements for å trigge sin økning i "congestion window size", sier TCP å være **self-clocking**.

TCP følger følgende retningslinjer:

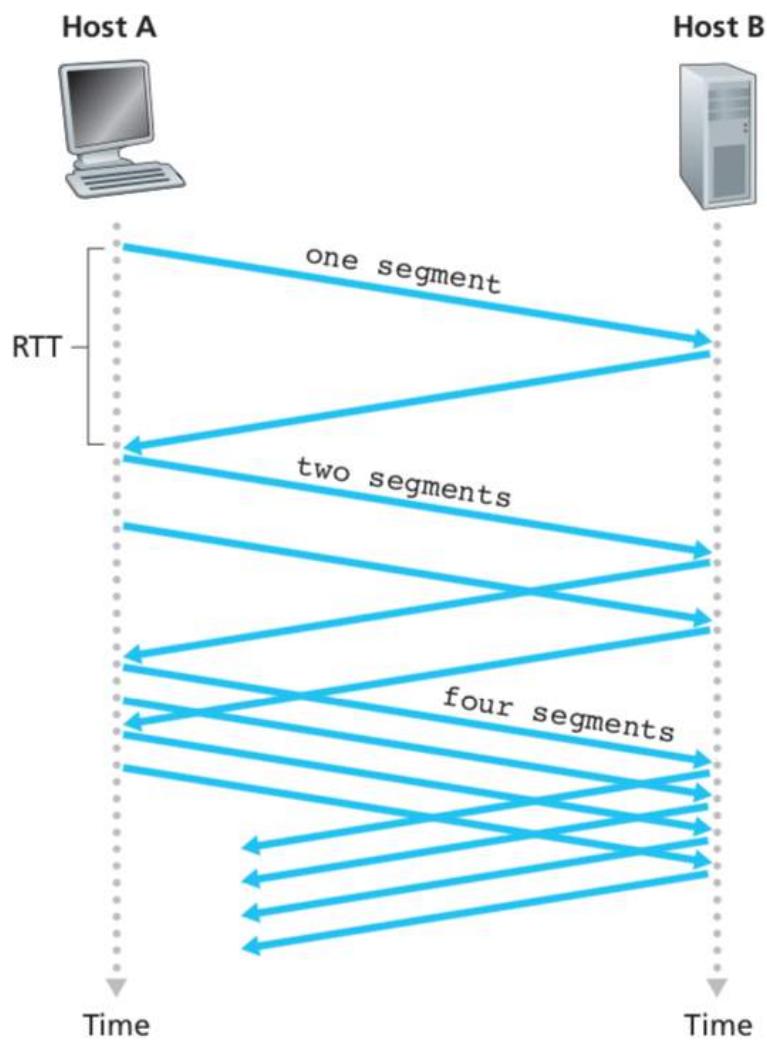
- *Et tapt segment innebærer overbelastning, og TCP-avsenderens hastighet bør derfor reduseres når et segment går tapt.* Husk at en time-out-hendelse eller kvittering av fire bekreftelser for et gitt segment (en original ACK og deretter tre dupliserte ACKs) tolkes som en implisitt "tapshendelse" -indikasjon av segmentet som følger fireganger ACKed-segment, utløser en re-sending av det tapte segmentet.
- *Et bekreftet segment indikerer at nettverket leverer avsenderens segmenter til mottakeren, og følgelig kan avsenderens hastighet økes når en ACK kommer før et tidligere ubekreftet segment.* Mottakelsen av bekreftelser er tatt som en implisitt indikasjon på at alt er bra - Segmenter blir vellykket levert fra avsender til mottaker, og nettverket er ikke overbelastet. Congestion window-et kan økes.

- **Båndbreddeundersøkelse (eng. Bandwidth probing)**. Gitt ACK-er som indikerer en overbelastningsfri kilde-til-destinasjon-vei og tapshendelser som indikerer en overbelastet vei, er TCPs strategi for å justere overføringshastigheten å øke frekvensen som følge av mottatte ACKs inntil det oppstår en tapshendelse, da blir overføringsrate redusert. Hver TCP-avsender virker på lokal informasjon asynkront fra andre TCP-sendere.

### TCP Congestion Control Algorithm

Algoritmen har 3 store komponenter: (1) slow start, (2) congestion avoidance og (3) fast recovery

1. **Slow start:** Når en TCP-kobling begynner, er verdien til  $cwnd$  initiativt en liten verdi på 1M MSS, som gir en sendingsrate på  $MSS/RTT$ . Deretter vil  $cwnd$  øke med 1 MSS for hvert bekrefte segment. Det vil se ut som figuren under.
  - Dersom det indikeres at en pakke er blitt mistet blir  $cwnd$  satt til 1 MSS, og det lagres en verdi  $ssthresh = cwnd/2$ .
  - Når  $cwnd = ssthresh$ , avslutter slow start, og TCP går inn i *congestion avoidance*-tilstanden
  - Dersom man finner 3 duplike ACKer, så utfører TCP en *fast retransmit* (kjapp re-sending), og går inn i *fast recovery*-tilstanden.



**Figure 3.51** ♦ TCP slow start

2. **Congestion avoidance:** Ved inngang til congestion avoidance-tilstanden er verdien av  $cwnd$  omtrent halvparten av verdien når congestion sist oppsto - congestion kan ligge rett rundt hjørnet! Så i stedet for å fordoble verdien av hver RTT, så øker TCP verdien av  $cwnd$  med bare 1 MSS hver RTT. Men når skal congestion-ens lineære økning (med 1 MSS per RTT) ende?

- TCPs congestion-algoritme oppfører seg det samme når en timeout oppstår: Setter verdien til  $cwnd$  og  $ssthresh$  og *slow start*.
- Ved tredobbelts duplikat ACK, så fortsettes det å sende segmenter, halverer  $cwnd$  og legger til 3 MSS for de tre ACK-ene. Går så inn i fast recovery\*.

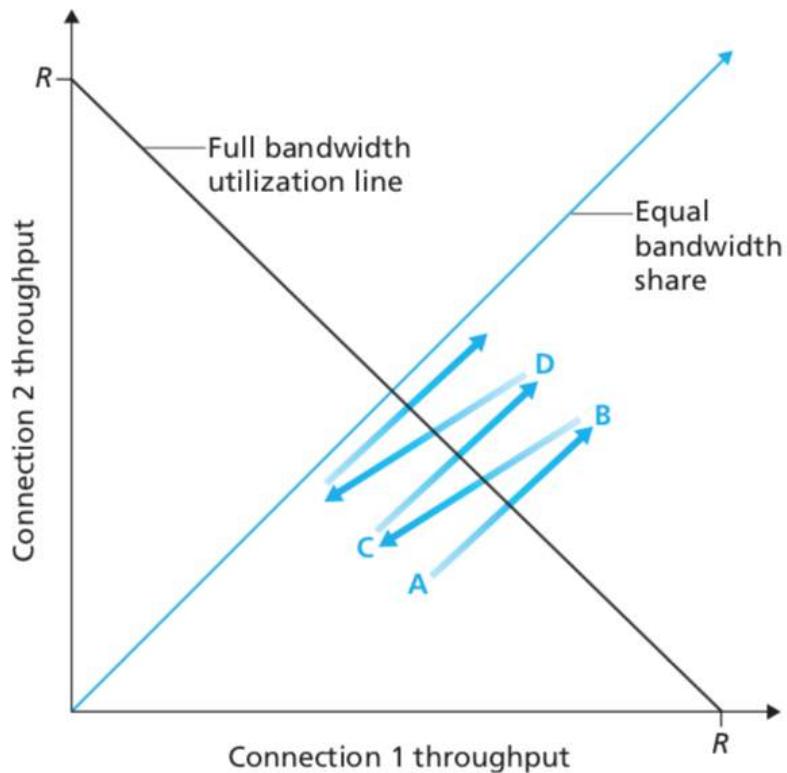
3. **Fast recovery:** Ved fast recovery økes verdien av  $cwnd$  med 1 MSS for hver duplikat ACK mottatt for det manglende segmentet som forårsaket at TCP kommer inn i fast recovery. Til slutt, når en ACK kommer for det manglende segmentet, går TCP inn i congestion-avoidance-tilstanden etter å ha deflatert  $cwnd$ .

- Hvis en **timeout** skjer, går over til slow-start-tilstand etter å utføre de samme handlingene som i slow start og congestion-avoidance: Verdien av  $cwnd$  er satt til 1 MSS, og verdien av  $ssthresh$  er satt til halvparten av verdien av  $cwnd$  da tapshendelsen skjedde.

TCP Congestion Control blir ofte referert til som en **additive-increase, multiplicative-decrease (AIMD)** form for congestion control.

### Fairness

Figuren under viser gjennomstrømingen til to connections gjennom en flaskehals. Punktene A, B, .. viser hvordan fordelingen av gjennomstrømning fordeler seg for hver endringene TCP Congestion Control gjør på congestion-vinduene deres.



**Figure 3.56** ♦ Throughput realized by TCP connections 1 and 2

### Fairness and UDP

Fra TCP sitt synspunkt er det urettferdig at multimedie-applikasjoner bruker UDP, da de kan sende så mye data de vil og i noen tilfeller miste pakker, istedet for å sette sendingsraten deres på et "fair" nivå, og hjelpe hverandre. Dermed er det mulig for UDP å overkjøre TCP-trafikk.

### Fairness and Parallel TCP Connections

Selvom vi kunne ha tvunget UDP-trafikken til å oppføre seg "fair", har vi ikke løst "fairness" problemet helt. Ingenting stopper TCP-baserte applikasjoner fra å kjøre parallele koblinger, som gir dem en større andel av båndbredden på linken.

# Kapittel 4 - The Network Layer

## Forwarding and routing

Rollen til nettverkslaget er enkelt - å flytte pakker fra den sendende verten til den mottakende verten. For å gjøre det så har nettverkslaget to viktige funksjoner:

- *Forwarding*. Når en pakke mottas ved en ruters input-linken, må ruteren flytte pakken til riktig utgangslink.
- *Routing*. Nettverkslaget må bestemme ruten eller stien tatt av pakkene mens de flyter mellom sender og mottaker. Algoritmene som bestemmer disse stiene er kalt **routing algorithms**.

*Forwarding* refererer til ruterens lokale handling for å overføre en pakke fra et inngangslinkgrensesnitt til riktig utgangslinkgrensesnitt.

*Routing* refererer til den nettverksbrede prosessen som bestemmer slutt-til-ende-banene som pakker tar fra kilde til destinasjon.

Hver ruter har et **forwarding table**. En ruters videresender en pakke ved å se på verdien til et header-felt hos den mottatte pakken, og sammenlikner med indeksene i ruterens forwarding table.

Når vi refererer til termet *packet switch* (nor. *pakkesvitsj*) mener vi en generell pakkesvitsje-enhet, som overfører en pakke fra input-link grensesnitt til output-link grensesnitt.

- Noen pakkesvitsjer, som kalles **link-layer switches**, baserer deres videresendingsbeslutninger på verdier i feltene til linklagsrammen (eng. *frame*), og refereres til som *link-layer (layer 2) devices*.
- Andre pakkesvitsjer, kalt **rutere**, baserer deres videresendingsbeslutninger på verdien i nettverklagsfeltet, og er dermed *network-layer (layer 3) devices*.

## Connection Setup

Vi sa nettopp at nettverkslaget har to viktige funksjoner, videresending og ruting. Men vi ser snart at i enkelte datanettverk er det faktisk en tredje viktig nettverkslagsfunksjon, nemlig **tilkoblingsoppsett** (eng. *connection setup*).

Husk at i TCP trengs det en treveis håndshake før data sendes. Dette gjør det mulig for avsenderen og mottakeren å sette opp nødvendig tilstandsinformasjon (for eksempel sekvensnummer og initiell flykontroll-vinduestørrelse).

På en analogisk måte vil noen nettverkslagsarkitekturen, for eksempel ATM, fram relay og MPLS, kreve at ruterne langs den valgte banen fra sender til mottaker "handshaker" med hverandre for å sette opp tilstander før nettverkslagdatapakker innenfor en gitt kilde-til-destinasjon-tilkoblingen kan begynne å strømme. I nettverkslaget kalles denne prosessen som tilkoblingsoppsett.

## Network Service Models

**Network service model** definerer egenskapene for ende-til-ende transport av pakker mellom sende- og mottaker-endesystemer.

La oss nå vurdere noen mulige tjenester som nettverkslaget kan gi, etter at transportlaget har overført en pakke til nettverkslaget:

- *Garantert levering*. Denne tjenesten garanterer at en pakke etter hvert vil ankomme destinasjonen sin.
- *Garantert levering med maks. forsinkelse*. Denne tjenesten garanterer ikke bare at pakken kommer frem, men levering innen et spesifisert vert-til-vert-forsinkelse.
- *In-order pakkelevering*. Denne tjenesten garanterer at pakkene kommer hos destinasjonen i rekkefølgen de ble sendt.
- *Garantert minimal båndbredde*. Så lenge en sender kan overføre pakker under koblingens bitrate, kan tjenesten garantere ingen tap av pakker og at de kommer frem innen et spesifisert vert-til-vert-delay.
- *Garantert maksimal jitter*. Denne tjenesten garanterer at tiden mellom overføringen av to pakker hos senderen er lik tidsforskjellen på mottakelsen av pakkene (eller mindre enn et spesifisert delay).
- *Sikkerhetstjenester*. Ved å bruke en hemmelig nøkkel, kun kjent av kilde- og desnitiasjonsvertene, kan man encrypte deler payloaden til datagrammet, og pakken er dermed konfidensiell mellom vertene.

| Network Architecture | Service Model | Bandwidth Guarantee      | No-Loss Guarantee | Ordering           | Timing         | Congestion Indication          |
|----------------------|---------------|--------------------------|-------------------|--------------------|----------------|--------------------------------|
| Internet             | Best Effort   | None                     | None              | Any order possible | Not maintained | None                           |
| ATM                  | CBR           | Guaranteed constant rate | Yes               | In order           | Maintained     | Congestion will not occur      |
| ATM                  | ABR           | Guaranteed minimum       | None              | In order           | Not maintained | Congestion indication provided |

**Table 4.1** ◆ Internet, ATM CBR, and ATM ABR service models

Internettets nettverkslag er en såkalt *best effort* tjeneste, det vil egentlig ikke si ingen tjeneste i det hele tatt. Med en best effort-tjeneste er ikke timingen mellom pakkene garantert, pakkenes rekkefølge ved ankommelse hos destinasjon eller heller ikke garantert, heller ikke om pakkene kommer frem i det hele tatt.

Av denne definisjonen så ville et nettverk som leverte ingen av pakkene til destinasjonen tilfredsstille definisjonen av en best effort-tjeneste.

To viktige ATM tjenestemodeller er *constant bit rate*- og *available bit rate*-tjenester:

- **Constant bit rate (CBR) ATM network service.** Dette var første ATM tjenestemodell som ble standardisert, og reflekterer telefonselskapene sin interesse i ATM. Målet med CBR tjeneste er å tilby en flyt av pakker (cells ATM-terminoloji) med et virtuelt rør hvis egenskaper er det samme som om en dedikert fastbåndbredde-overføringskobling eksisterte mellom sending og mottak av verter. Med CBR-tjenesten blir en strøm av ATM-cellene båret over nettverket på en slik måte at en celles ende-til-ende forsinkelse, variabiliteten i en celle ende-til-ende forsinkelse (det vil si jitteren) og brøkdelen av celler som er tapt eller levert sent, er alle garantert å være mindre enn angitte verdier.
- **Available bit rate (ABR) ATM network service.** Med Internett som tilbyr såkalt best-service-tjeneste, kan ATMs ABR best karakteriseres som en litt bedre enn best mulig service. Som med Internett-tjenestemodellen kan celler gå tapt under ABR-tjenesten. I motsetning til Internett, kan imidlertid ikke cellene ombestilles (selv om de kan gå tapt), og en minimums celleoverføringshastighet (MCR) garanteres til en tilkobling ved hjelp av ABR-tjenesten.

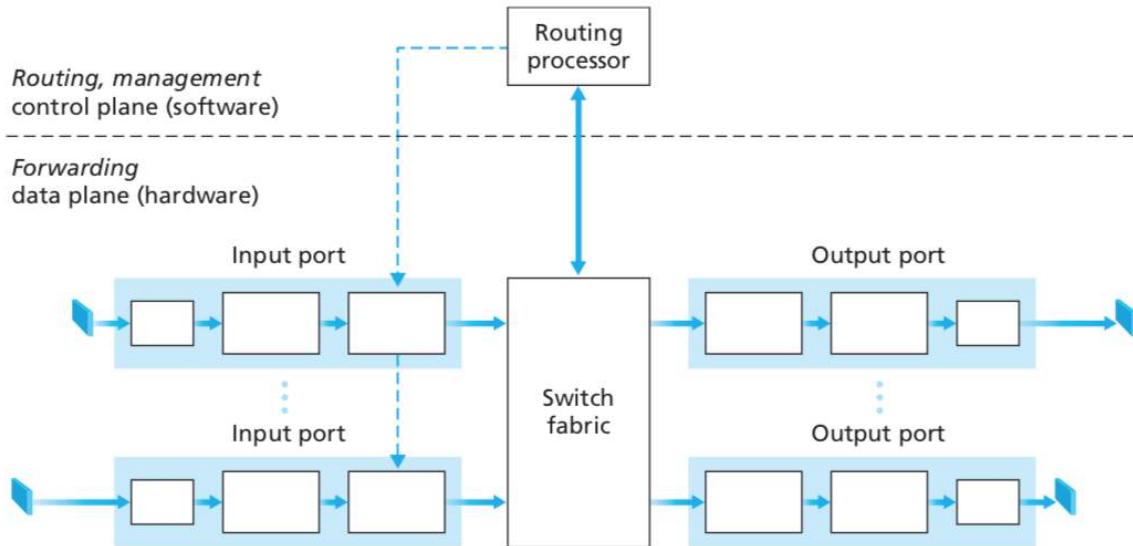
## What's Inside a Router

En ruter har fire komponenter som kan bli identifisert:

- *Input port:* En inngangsport utfører en rekke nøkkelfunksjoner. Den utfører den fysiske-lagsfunksjonen ved å avslutte en innkommende fysisk link ved en ruter; Dette vises i den venstre boksen til inngangsporten og den høyre boksen til utgangsporten i figur 4.6. En inngangsport utfører også link-layer-funksjoner som trengs for å samvirke med link-layer på den andre siden av innkommende linken. Dette er representert av mellomkassene i inngangs- og utgangsportene. Kanskje mest avgjørende, **oppslagsfunksjonen** utføres også ved inngangsporten; Dette vil skje i den høyre boksen til inngangsporten. Det er her at videresendingstabellen blir konsultert for å bestemme utgangsporten til ruten som en pakke vil bli videresendt via *switching fabric*-en. Kontrollpakker (for eksempel pakker som bærer rutingsprotokollinformasjon) videresendes fra en inngangsport til rutingsprosessoren.

Vær oppmerksom på at termen port her - som refererer til de fysiske inngangs- og utgangsrutergensesnittene - er tydelig forskjellig fra programvareportene som er knyttet til nettverksapplikasjoner og socketsene som er omtalt i kapittel 2 og 3.

- *Switching fabric:* Switching fabric-en kobler sammen ruterens inngangsporter til dens utgangsporter. Denne er kun inne i ruten - et nettverk i en nettverksruter.
- *Output port:* En utgangsport lagrer pakker mottatt fra switching fabric-en og sender disse pakkene til den utgående linken, ved å utføre nødvendige linklags- og fysisklags-funksjoner.
- *Routing processor:* Ruterprosessoren utfører ruter-protokoller, vedlikeholder *routing tables* og tilstandsinformasjon til tilkoblede linker, og regner ut *forwarding table*-et for ruten.



**Figure 4.6** ♦ Router architecture

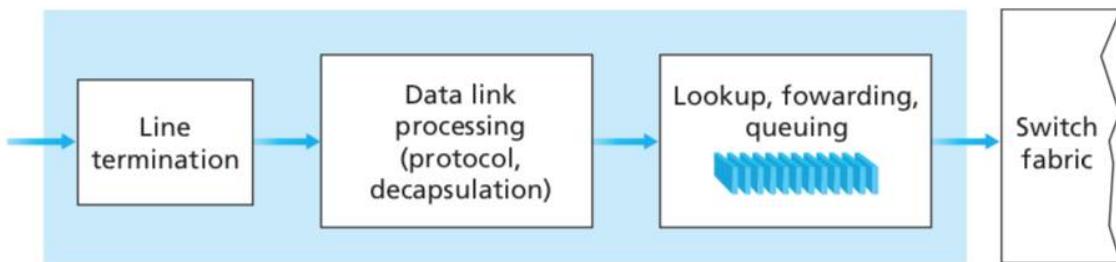
En ruters inngangsporter, utgangsporter og switchingfabric implementerer sammen forwarding-funksjonen. Disse forwarding-funksjonene er kollektivt referert til som **router forwarding plane**, som er implementert inn i maskinvaren. Forwarding-planet opererer på nanosekund-skaane, mens ruterens controlfunksjoner - som utfører ruterprotokollene, svarer koblinger, og utfører management-funksjoner, opererer på millisekund- eller sekund-skala. Disse **router control plane** funksjonene er vanligvis implementert inn i programvaren, og kjører på routing prosessoren.

#### Input-prosesserings

Som nevnt over, inngangsportens linjeterminering og link-lags prosessering implementerer det fysiske- og link-laget av den individuelle input-linken.

Inngangsporten skjekking (eng. *lookup*) er sentral for ruterens operasjoner - det er her ruteren bruker forwarding tabellen for å se opp hvilken utgangsport pakken skal bli forwardet til via switching fabricen.

Forwarding table-er blir utregnet og oppdatert av routing-prosessoren, med en "skygge-kopi" liggende hos hver inputport. Forwarding table-er blir kopiert fra ruterprosessoren til linjekortene over en separat buss indikert av den stiplede linjen på Figur 4.6 over. Med en skygge-kopi kan forwarding-beslutninger gjøres lokalt hos hver inngangsport, uten å spørre den centraliserte prosessoren på en per-pakke-basis. Dernes *slippes* en **centralisert flaskehals**.



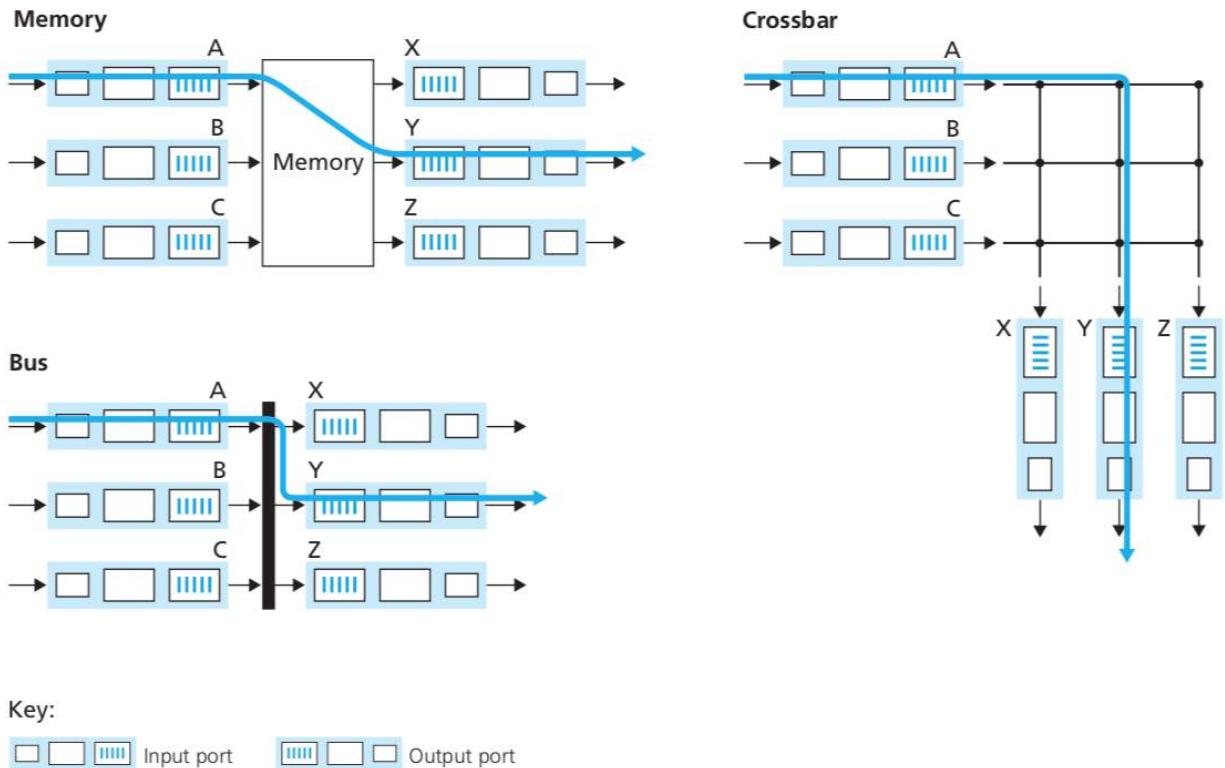
**Figure 4.7** ♦ Input port processing

Når en pakkes utgangsport har blitt bestemt via lookup-en, kan pakken bli sendt inn i switching fabric-en. I noen design vil pakker bli blokkert fra å gå inn i fabric-en dersom noen andre pakker fra andre inputporter bruker den. En blokket pakke blir satt i en kø hos inputporten, og som planlegges å gå inn i switching fabric-en senere.

Selvom denne *lookup-en* er den viktigste aksjonen ved inputportprosesserings, må mange andre handlinger bli gjort: (1) fysisk- og link-lagsprosesserings må skjer, (2) pakkens versjonsnummer, checksum og time-to-live-felt må sjekkes, og de to sistnevnte må overskrives, og (3) tellere brukt til nettverksadministrering (som antall IP datagram motatt) må oppdateres. Denne inngangsportprosesseringsen kan enkelt forklares med "match-and

## Switching

Switching fabric-en er hjertet til ruteren, det er gjennom dette stoffet at pakkene faktisk blir svitsjet (*altså forwardet*) fra inngangsport til utgangsport. Svitsjing kan bli utført på en rekke måter, vist i Figur 4.8.

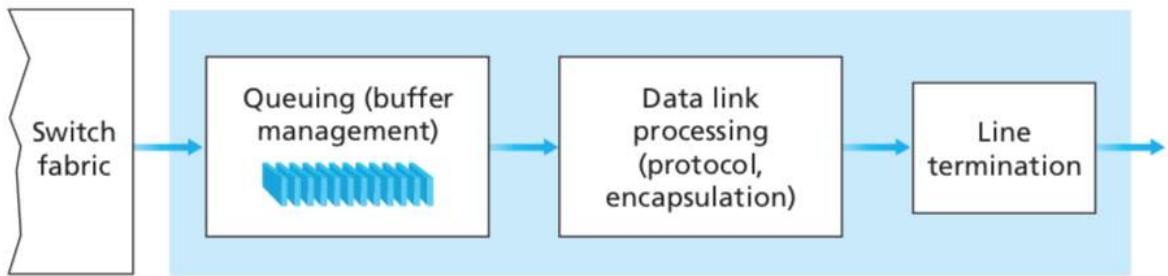


**Figure 4.8** ♦ Three switching techniques

- **Switching via memory.** De enkelste, tidligste ruterne var tradisjonelle datamaskiner der svitsjingen mellom portene ble gjort under direkte kontroll av CPU-en. Inngang- og utgangsportene fungerte som tradisjonelle I/O enheter i et tradisjonelt operativsystem.
  - Når en pakke ankom på inngangsporten, sendte det en *interrupt* til CPU-en som kopierte pakken over til prosessorminnet. Prosessoren hentet så destinasjonsadressen fra headeren, skjekket passende utgangsport og kopierte pakken til utgangsportens buffer. Dersom båndbredden til CPU-ens minne B, som er antall pakker i sekundet som kan leses, skrives eller sendes, så må den totale videresendingstrømmen (den totale hastigheten som pakker overføres fra inngangsporter til utgangsportene) være mindre enn  $B / 2$ .
- **Switching via a bus.** Ingangsporten overfører pakken direkte til utgangsporten over en *delt bus*, uten intervenering fra ruterprosessoren. Dette er vanligvis gjort ved å legge på en ekstra header-label som indikerer hvilken utgangsport den skal sendes fra. Pakken vil mottas av alle utgangsportene, men kun en port som matcher labelen, og beholder pakken. Label-en blir fjernet ved utgangsporten. Siden alle pakkene må over bussen, er ruterens svitsjing-hastighet begrenset av bussens hastighet.
- **Switching via an interconnection network.** En måte å overkomme båndbreddebegrensning av en enkel, delt buss er å bruke et litt mer sofistikert interkoblet nettverk. En *crossbar-switch* er en sammenkoblingsnettverk bestående av  $2N$  busser, som kobler sammen  $N$  inputporter med  $N$  utgangsporter, som vist over. Når en pakke skal fra port A til port Y, så vil switch-fabric-controlleren lukke krysspunktet ved krysset til bussene A og Y.

## Output Processing

Utgangsprosesering, vist i Figur 4.9 tar pakker om er lagret hos utgangsportens minne og sender dem over til utgangslinken. Dette inkluderer å selektere og de-queue pakker for seding, og utføre nødvendig link-lags- og fysisk-lags-overføringsfunksjoner



**Figure 4.9 ♦ Output port processing**

#### Where does Queueing Occur?

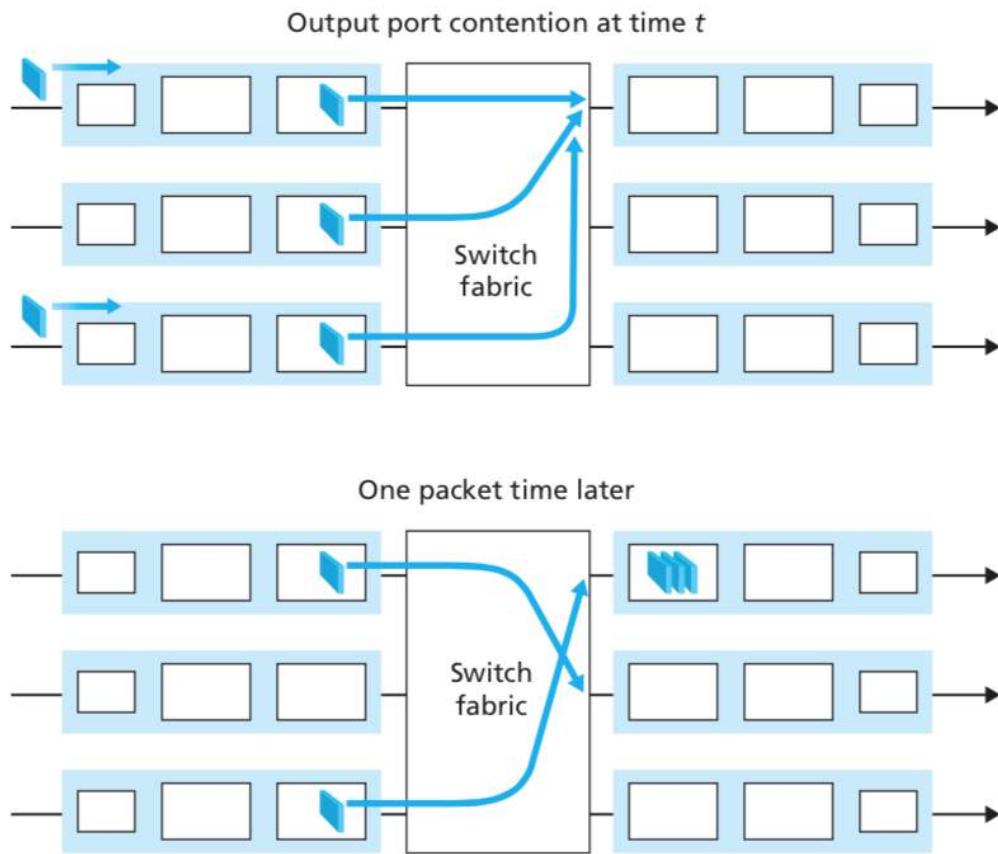
Som lett kan sees i Figur 3.8, er det klart at pakkekøer kan forme seg på både inngangsportene og utgangsportene - analogi kan være biler som kjører inn mot en rundkjøring. Plasseringen og omfanget av køen (enten ved inngangsportkøene eller i utgangskortkøene) vil avhenge av trafikkbelastningen, den relative hastigheten til koblingsmaterialet og linjens hastighet.

Dersom en kø blir veldig stor og det ikke er plass til flere pakker, vil det ikke være plass til å lagre innkommende pakker, og vi vil oppleve **packet loss**.

Vi sier at pakken ble "mistet i nettverket" eller "sluppet hos ruteren"

Det er i disse ruterkøende at pakker blir droppet eller mistet.

- Anta at inngang- og utgangslinjenes hastighetene (transmission rates) alle har en identisk overføringsrate på  $R_{line}$  pakker per sekund, og at det er  $N$  inngangs- og  $N$  utgangsporter.
- La oss definere switching fabric-ens overføringsrate  $R_{switch}$  som raten av pakker som kan bli overført fra inngangsport til utgangsport. Dersom  $R_{switch}$  er  $N$  ganger raskere enn  $R_{line}$ , da vil det bare oppstå ubetydelig kø på inngangsportene, men det kan oppstå kø hos utgangsportene dersom mange av paklene går til samme utgangsport.



**Figure 4.10** ♦ Output port queuing

Tommelregelen for bufferstørrelse var i mange år at mengde buffering ( $B$ ) skulle være lik gjennomsnitts round-trip time (RTT) ganger linkens kapasitet ( $C$ ).  $B = RTT \cdot C$

- En konsekvens med utgangsportkø er at en **packet scheduler** ved utgangsporten må velge en pakke blant de som er i køen for overføring. Denne seleksjonen kan gjøres enkelt, FIFO-scheduling, eller en mer sofistiskert måte, som *weighted fair queueing (WFQ)*, som deler utgangslinken fairly blandt de forskjellige ende-til-ende-tilkoblingene.
  - Packet scheduling spiller en viktig rolle for å tilby **quality-of-service guarantees**.
- Dersom det ikke er nok minne i bufferen til en innkommende pakken, må et valg tas om man enten skal droppe den innkommende pakken (kalles **drop-tail**) eller fjerne en eller flere av allerede kø-edde pakker, for å lage plass til den nyankommende pakken.
  - Vanlig er det vanlig å merke et header-felt om overbelastningssignal til senderen, før bufferen er full. Det finnes et antall pakke-dropping og -markeringspolicyer (kollektivt kjent som **active queue management (AQM) algoritmer**).

En av de mest studerte og impletterte AQM-algoritmenene er **Random Early Detection (RED)** algoritmen. Under RED lages et threshold-intervall  $[min_{th}, max_{th}]$ .

- Dersom den gjennomsnittlige kølengden er mindre enn  $min_{th}$ , vil pakken legges i køen.
- Dersom køen er full eller den gjennomsnittlige kølengden er større enn  $max_{th}$ , vil pakken markeres eller droppes.
- Dersom køen har en gjennomsnittskølengde i intervallet  $[min_{th}, max_{th}]$ , vil pakken markeres eller droppes med en sannsynlighet som er en typisk funksjon av min- og max-thresholdet.

Figur 4.11 viser et eksempel hvor to pakker (mørkt blå) på forsiden av inngangskøene er bestemt for samme øvre høyre utgangsport. Anta at switch-fabric velger å overføre pakken fra fronten av øvre venstre kø. I dette tilfellet må den mørkeblå pakken i nedre venstre kø vente. Men ikke bare må denne mørkeblå pakken vente, så også må den lyseblå pakken som står i køen bak den pakken i nedre venstre kø, selv om det ikke er noen tvil for midt-høyre utgangsporten (målet for den lyseblå pakke)n.

Dette fenomenet kalles **head-of-the-line-blokking (HOL)** i en svitsj med inngangskø.

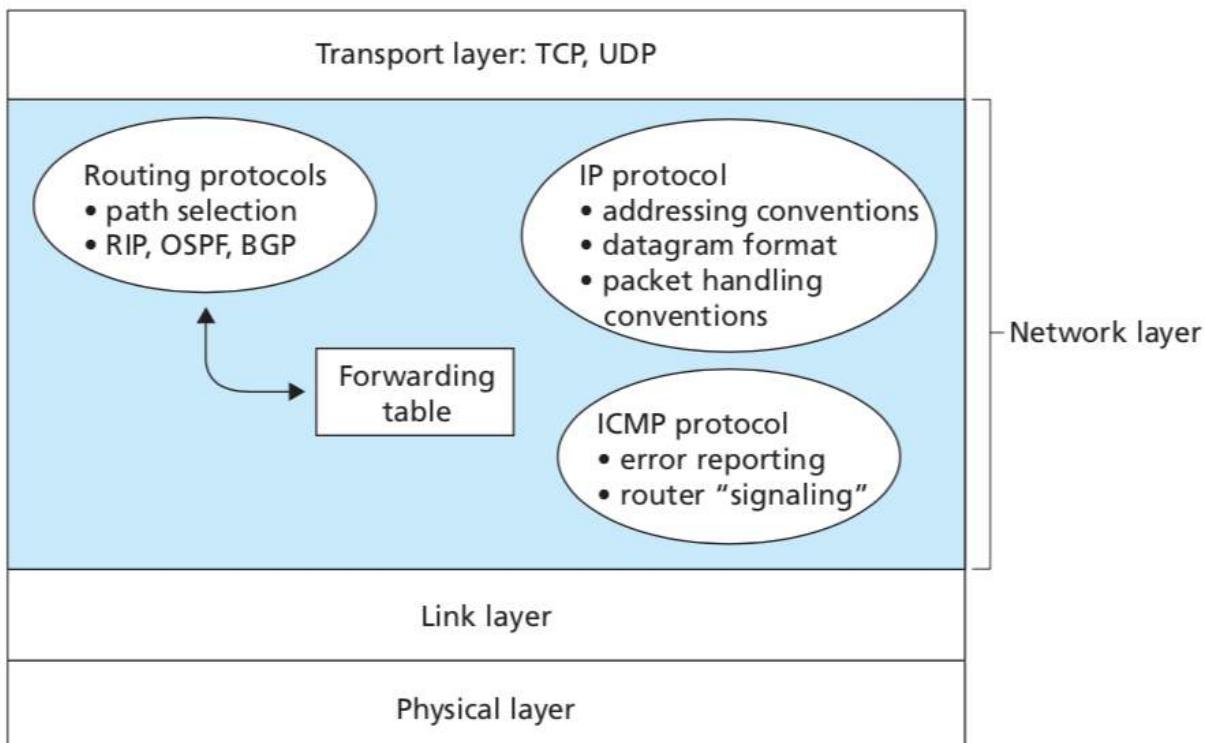
- En kø-pakke i en inngangskø må vente på overføring gjennom switching-fabric (selv om utgangsporten er ledig) fordi den er blokkert av en annen pakke forrerst på linjen.

### The Routing Control Plane

I vår diskusjon hittil og i Figur 4.6 har vi implisitt antatt at *routing control plane* fullt ut ligger i en ruter prosessor innenfor ruteren. Det nettverksbrede *routing control plane* er desentralisert med forskjellige deler (for eksempel av en rutingsalgoritme) som utføres hos forskjellige rutere som interagerer med hvandre ved å sende kontrollmeldinger til hverandre.

### The Internet Protocol (IP): Forwarding and Addressing in the Internet

Vi skal nå se at Internett adressering og forwarding er viktige komponenter til Internet Protocol (IP). Det er to versjoner av IP i bruk idag. Vi skal først se på den mye brukte IP protokoll versjon 4, som refereres til som IPv4. vi skal se på IP versjon 6 (IPv6), som er blitt foreslått å erstatte IPv4, mot slutten av seksjonen.

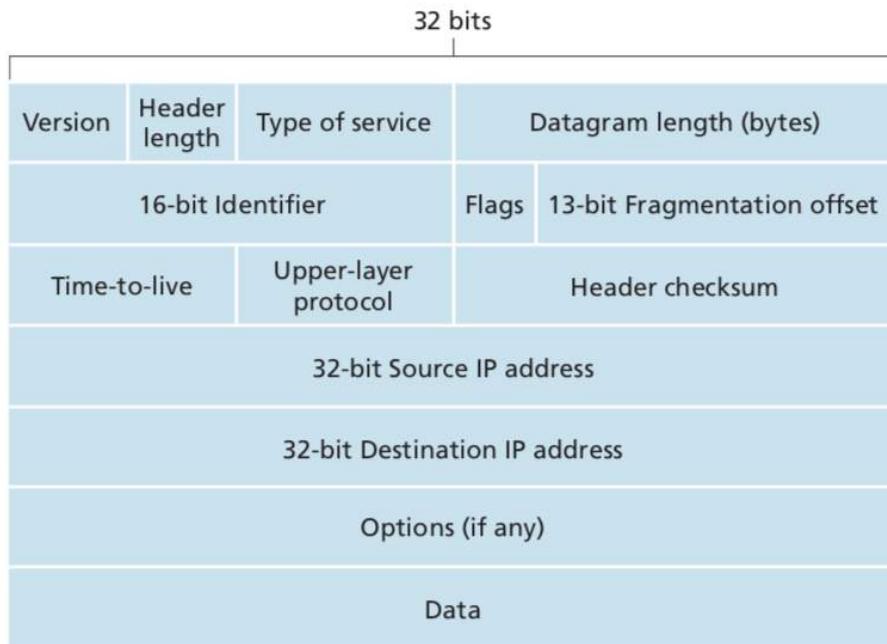


**Figure 4.12** ♦ A look inside the Internet's network layer

Vi ser at internettets nettverkslag har tre store komponenter. Den første komponenten er IP-protokollen. Den andre komponenet er ruter-komponenten, som bestemmer stien et datagram tar fra kilde til destinasjon. Den siste komponenten av nettverkslaget er et anlegg for å rapportere feil i datagrammer og svare på forespørsler om bestemt nettverkslagsinformasjon. Vi vil dekke internettets nettverkslagrings- og informasjonsrapporteringsprotokoll, ICMP (Internet Control Message Protocol) snart.

### Datagram Format

Husk at en nettverkslagspakke refereres til som et *datagram*.



**Figure 4.13** ♦ IPv4 datagram format

IPv4 datagram-formatet er vist i Figur 4.13. Nøkkelfeltene i TPv4-datagrammet er:

- **Version number**. Disse 4 bitene spesifiserer IP-protokoll versjonen til datagrammet. Forskjellige versjoner av IP burker forskjellige datagramformater.
- **Header length**. Fordi et IPv4-datagram kan inneholde et variabelt antall med options, er disse 4 bitene trengt for å bestemme hvor dataen i datagrammet begynner.
- **Type of service**. Type of service (TOS) bits er inkludert i headeren for å la forskjellige typer av IP datagrammer for å tillate forskjellige typer IP-datagrammer (f.eks. datagram som spesifikt krever low delay, high throughput eller reliability) for å skille de fra hverandre.
- **Datagram length**. Dette er den totale lengden til et IP-datagram (header pluss data). Dette feltet er 16 bits, og derfor er den teoretiske maks.størrelsen til et IP datagram 65,535 bytes - selvom de sjeldent er større en 1,500 bytes.
- **Identifier, flags, fragmentation offset**. Disse tre feltene har med den såkalte IP-fragmentasjonen, noe vi skal se på snart. IPv6 tilatter ikke fragmentering hos ruterne.
- **Time-to-live**. Time-to-live-feltet (TTL) er inkludert for å forsikre at datagrammer ikke sirkulerer for alltid pga f.eks. en loop i nettverket. Dersom TTL-feltet er 0, må pakken droppes.
- **Protocol**. Dette fletet brukes kun når IP-datagrammet kommer frem til destinasjonen. Verdien til dette feltet indikerer hvilken spesifikk transportprotokoll som skal ta hånd om dataen i datagrammet.
- **Header checksum**. Header-checksummen hjelper ruter i å finne bit-feil i et mottatt IP-datagram. Header-checksummen er regnet ut ved å behandle alle 16 byte bits ord i headeren som et tall og summe disse tallene sammen deretter å ta 1s komplement. Dersom man finner en feil pleier man å droppe datagrammet.
  - Noter at man har slike bit-error-skjekker både på transport og nettverkslaget. Hvorfor vi trenger å skjekke begge steder? Legg merke til at det kun er IP-headeren som blir checksummet ved IP-laget, mens TCP/UDP checksum regnes ut over hele TCP/UDP segmentet.
- **Source and destination IP addresses**. Når en kilde lager et datagram legger den til sin IP-adresse, og legger til mottakerens IP-adresse. Ofte bestemmer kilden destinasjonsadressen via en DNS-lookup.
- **Options**. Options-feltet lar en IP-header til å bli utvidet. Siden man har opitons-felt vil IP-datagrammer har variabel størrelse, og man må regne ut hvor dataen starter i datagrammet.
- **Data (payload)**. Til sist kommer vi til det viktigste feltet, nemlig dataen til datagrammet. Data-feltet til IP-datagrammet inneholder transportlags-segmentet (TCP / UDP) for å bli levert til destinasjonen. Datafeltet kan også holde andre typer data, som en ICMP melding.

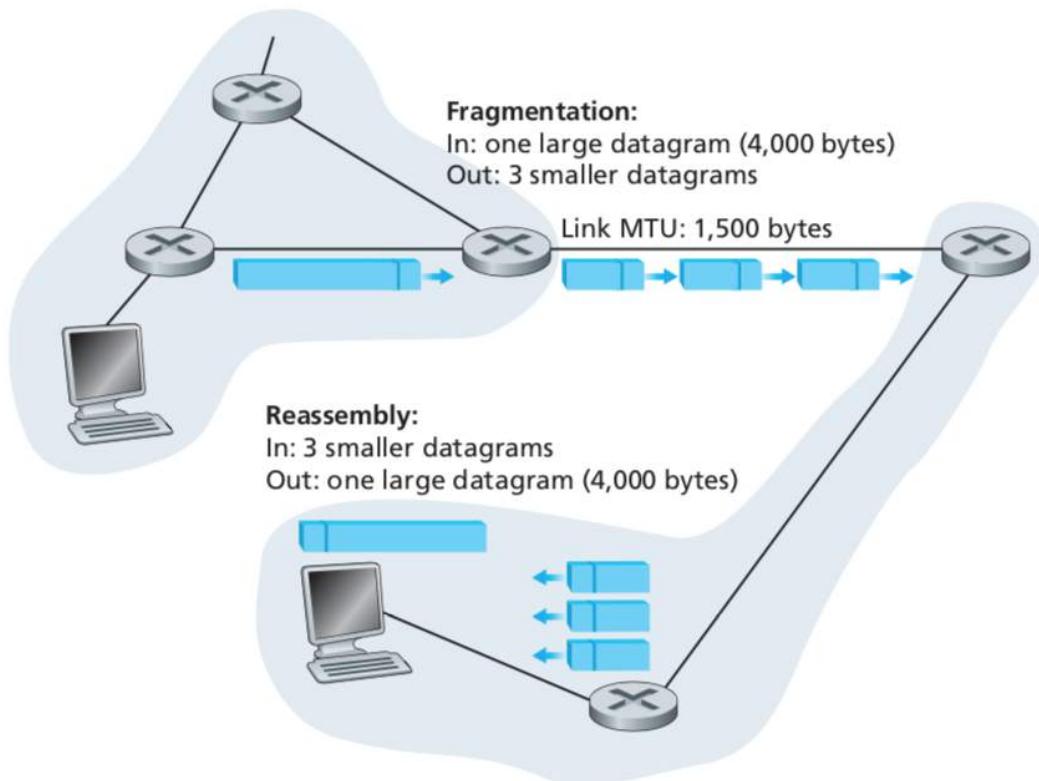
IP-datagrammet har totalt 20 bytes med header (antar ingen options). Dersom et datagram bærer et TCP-segment, da vil hvert datagram bære totalt 40 bytes med header (20 bytes IP-header pluss 20 bytes TCP-header) sammen med applikasjonsmeldingen.

## IP Datagram Fragmentation

I neste kapittel skal vi se at ikke alle link-layer-protokollen kan bære netverkslagspakker på samme størrelse. Noen protokoller kan bære store datagrammer, andre kan bare ta små. Den maksimale mengden data en **link-layer-frame** kan bære kalles \*maximum transmission unit (MTU). På grunn av at hvert IP-datagram er innkapslet i en linklagsramme for transport mellom en ruter til en annen, vil MTU-en til en linklagsprotokoll sette en streng grense på lengden til IP-datagrammet.

Dersom man mottar et IP-datagram fra en inngående link, og den utgående linken har en mindre MTU enn lengden til IP-datagrammet. Hva gjør man da?

Løsningen er å fragmentere dataen i IP-datagrammet til to mindre IP-datagram, og innkapsle hver av disse i hver sin linklagsramme. Hvert av disse mindre datagrammene er referert til som et **fragment**. Jobben om å defragmentere datagrammene er satt til å være hos endesystemene og ikke hos ruterne. For å kunne tillate verten å defragmentere så putter man *identifikasjon flag*, og *fragmentation offset* feltene i IP-datagramheaderen. Når et datagram lages så legger den sendende verten på et identifikasjonsnummer. For å forsikre seg om at alle pakker kommer frem, så vil den siste pakken ha flagget satt til 0, mens de andre har den satt til 1.



**Figure 4.14** ♦ IP fragmentation and reassembly

Dersom en eller flere av fragmentene ikke ankommer til destinasjonen vil det ukomplette fragmentet droppt og ikke sendt til transportlaget.

## IPv4 Addressing

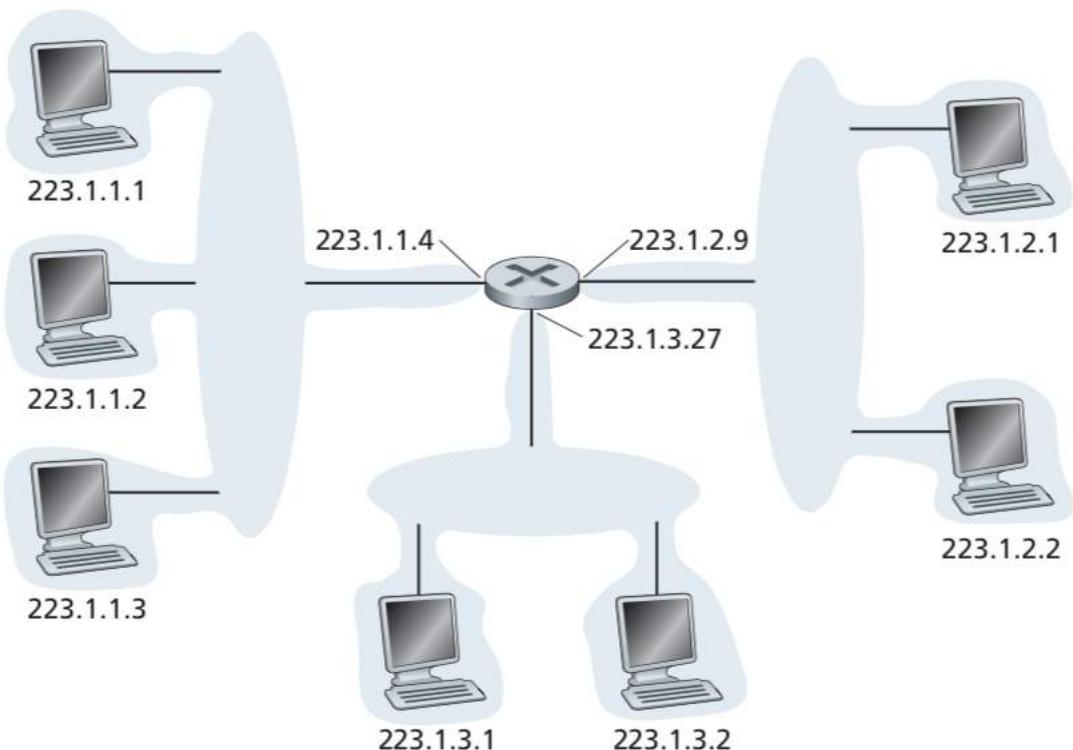
En vert har typisk en enkel link til nettverket - Når IP i verten ønsker å sende et datagram, gjør den det over denne linken. Grensen mellom verten og den fysiske-linken kalles et **interface**. Grensen mellom en ruter og en av dets linker kalles også et interface - en ruter har dog flere interfaceter, et for hver link.

Ettersom hver vert og ruter er kapable til å sende og motta IP-datagrammer, krever IP at hvert interface hos hver vert og ruter har sin egen IP-adresse. En IP-adresse er dermed teknisk assosiert med en interface, istedet for med verten eller ruten som inneholder interfacet.

Hver IP-adresse er 32-bits (4 bytes) lang. Det gir en total på  $2^{32}$  mulige IP-adresser (4 milliarder). Disse adressene er typisk skrevet i **dotted-decimal notation**, der hver byte i IP-adressen, i decimal, skiller fra hverandre med et punktum.

Adressen 193.32.216.9 i binær notasjon blir 11000001 00100000 11011000 00001001.

Hvert interface på hver eneste vert og ruter på internettet må ha en IP-adresse som er globalt unik. Disse adressene kan ikke bli valgt på en tilfeldig måte, men en del av interfacets IP-adresse er bestemt av subnettet som den er koblet på.



**Figure 4.15** ♦ Interface addresses and subnets

Figur 4.15 over gir et eksempel på IP-adressering og interfacer. I bildet er det en ruter (med tre interfaceter) som brukes til å sammenkoble seg med syv verter. Ta en titt på de tre vertene øverst til venstre. De har alle en IP-adresse på formen 223.1.1.xxx (som bestemmer de 24 bitene til venstre av IP-adressen).

De fire interfacene er altså sammenkoblet med hverandre av et nettverk som ikke har en ruter. Dette nettverket kan være sammenkoblet av Ethernet LAN. På IP-term, kalles denne typen for nettverk med sammenkoblede vert-interfacer og et ruter-interface for et **subnet** (Et subnet kalles også et *IP-nettverk*).

IP-adressering tilegner en adresse til dette subnettet: 223.1.1.0/24, der /24 notasjonen, kalt en **subnet mask**, indikerer at de 24 mest venstre bitene av de 32-bitene definerer subnettets adresse.

To determine the subnets, detach each interface from its host or router, creating islands of isolated networks, with interfaces terminating the end points of the isolated networks. Each of these isolated networks is called a subnet.

Internettets adressetildelingsstrategi er kjent som **Classless Interdomain Routing (CIDR)**. CIDR generaliserer begrepet subnet adressering. Som med subnetadressering er 32-biters IP-adresse delt inn i to deler og har igjen den stiplede desimalformen  $a.b.c.d / x$ , hvor x angir antall biter i den første delen av adressen.

De x mest signifikante biter av en adresse på skjemaet  $a.b.c.d / x$  utgjør nettverksdelen av IP-adressen, og blir ofte referert til som **prefiks** (eller *network prefix*) til adressen.

Før CIDR ble vedtatt, ble nettverksdelene av en IP-adresse begrenset til å være 8, 16 eller 24 bits i lengden, en adresseringsordning kjent som **classful addressing**, siden subnett med 8-, 16- og 24-biters subnett-adresser ble kjent som henholdsvis klasse A, B og C-nettverk.

IP-broadcast-adressen 255.255.255.255. Når en vert sender et datagram med destinasjonsadresse 255.255.255.255, blir meldingen levert til alle verter på samme subnett. Rutere videresender eventuelt meldingen til nærliggende subnett også (selv om de vanligvis ikke gjør det).

#### Obtaining a Block of Addresses

For å få en *blokk* med IP-adresser for bruk i en organisasjonens subnett, kan en nettverksadministrator først kontakte sin *ISP*, som vil gi adresser fra en større adresseblokk som allerede er blitt tildelt ISP. For eksempel kan Internett-leverandøren selv ha blitt tildelt adresseblokken 200.23.16.0/20. Internett-leverandøren kan i sin tur for eksempel dele sin adresseblokk inn i åtte like store sammenhengende adresseblokker og gi en av disse adresseblokkene til opptil åtte organisasjoner som støttes av denne Internett-leverandøren, som vist nedenfor. (Vi har understreket delnettelen av disse adressene for enkelhets skyld.)

|                |                |  |
|----------------|----------------|--|
| ISP's block    | 200.23.16.0/20 | <u>11001000</u> <u>00010111</u> <u>00010000</u> 00000000 |
| Organization 0 | 200.23.16.0/23 | <u>11001000</u> <u>00010111</u> <u>00010000</u> 00000000 |
| Organization 1 | 200.23.18.0/23 | <u>11001000</u> <u>00010111</u> <u>00010010</u> 00000000 |
| Organization 2 | 200.23.20.0/23 | <u>11001000</u> <u>00010111</u> <u>00010100</u> 00000000 |
| ...            | ...            | ...  |
| Organization 7 | 200.23.30.0/23 | <u>11001000</u> <u>00010111</u> <u>00011110</u> 00000000 |

#### Obtaining a Host Address: the Dynamic Host Configuration Protocol (DHCP)

Når en organisasjon har fått en blokk med adresser, kan de tildele individuelle IP-adresser til vert- og ruter-interfacer i organisasjonen sin. En systemadministrator vil typisk manuelt konfigurer IP-adressene inn i ruten.

Vertadressene kan også konfigureres manuelt, denne oppgaven blir ofte gjort med **Dynamic Host Configuration Protocol (DHCP)**. DHCP tillater en vert å få (tilordnet) en IP-adresse *automatisk*. En nettverksadministrator kan konfigurere DHCP slik at en gitt vert mottar samme IP-adresse hver gang den kobles til nettverket, eller en vert kan tilordnes en midlertidig IP-adresse som vil være *forskjellig* hver gang verten kobles til nettverket. I tillegg til vert-IP-adressetildeling tillater DHCP også at en vert lærer tilleggsinformasjon, for eksempel sin nettverksmaske, adressen til sin første hop-router (ofte kalt standard gateway) og adressen til den lokale DNS-serveren.

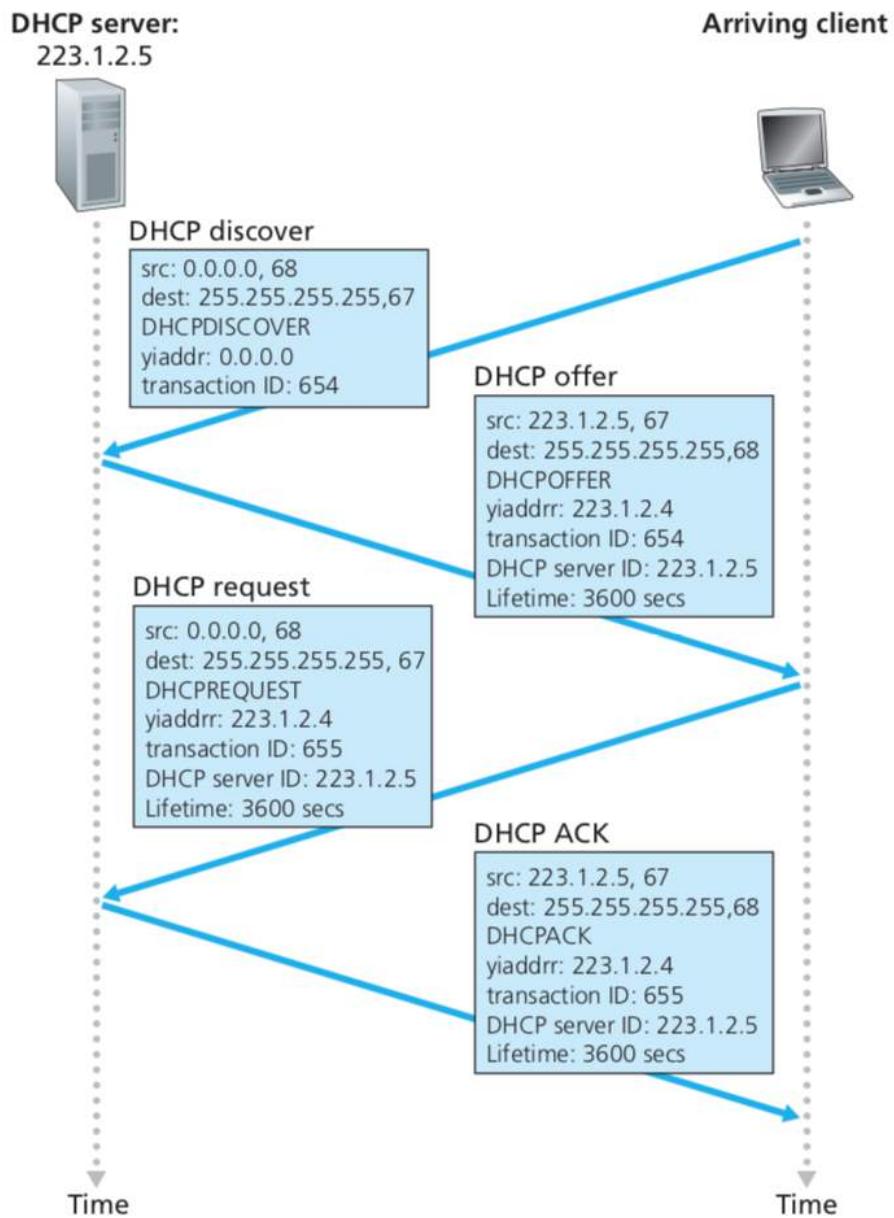
På grunn av DHCPs evne til å automatisere nettverksrelaterte aspekter ved å koble en vert til et nettverk, blir det ofte referert til som en plug-and-play-protokoll. Denne funksjonen gjør det veldig attraktivt for nettverksadministratoren som på annen måte må utføre disse oppgavene manuelt!

DCHP er bra i nettverk der det er mange verter som kommer og forlater nettverket ofte.

- DHCP er en klient-server protokoll. En klient er typisk en nyankommen vert som ønsker å få nettverkskonfigureringsinformasjon, som IP-adresse for seg selv. I det enkelste tilfellet har hvert subnett en egen DHCP-server.

Når en ny vert kommer til nettverket og ønsker å koble seg til, skjer det fire steg i DHCP klient-server interaksjonen:

1. *DHCP server discovery*. Klienten sender en **DHCP discover message** med en UDP pakke på port 67, til IP-adresse 255.255.255.255, og source (klientens) IP-adresse 0.0.0.0
2. *DHCP server offer*. Når DHCP-serveren mottar requesten, vil den svare klienten med en **DHCP offer message**, som sendes til 255.255.255.255, denne sendes til broadcast-adressen pga at det kan være flere DHCP-servere på subnettet. Hvert server offer-melding inneholder en transaksjons ID til det mottatte discover-meldingen, den foreslalte IP-adressen til klienten, nettverksmasken og en leasingtid for IP-adresser - hvor lang tid IP-adressen vil være gyldig for.
3. *DHCP request*. Den nye klienten kan velge mellom en eller flere server-offere og respondere til det valgte tilbudet med en \***DHCP request message**, og ekkoer tilbake konfigurasjonsparametrene fra den forrige meldingen.
4. *DHCP ACK*. Serveren respondere til DHCP request message-en med en **DHCP ACK message**, som bekrefter de forespurte parametrene.



**Figure 4.21** ♦ DHCP client-server interaction

#### Network Address Translation (NAT)

Vi vet nå at alle IP-kapable enheter trenger en IP-adresse. Dersom man har brukt opp alle IP-adressene i adresseblokken fra ISP-en sin, hvordan skal man da alllokere nye adresser? En tilnærming er med **network address translation (NAT)**.

Figur 4.22 viser driften av en NAT-aktivert ruter. Den NAT-aktiverte ruten, bosatt i hjemmet, har et grensesnitt som er en del av hjemmenettverket til høyre i figur 4.22. Adressering i hjemmenettverket er akkurat som vi har sett ovenfor – alle fire grensesnittene i hjemmenettverket har samme subnettadresse på 10.0.0 / 24. Adresseområdet 10.0.0.0/8 er en av tre deler av IP-adresseplassen som er reservert i for et privat nettverk eller et **rike** med private adresser, for eksempel hjemmenettverket i Figur 4.22.

Et rike med private adresser refererer til et nettverk hvis adresser bare har betydning for enheter innenfor det aktuelle nettverket.

For å se hvorfor dette er viktig, bør du vurdere det faktum at det er hundretusensvis av hjemmenettverk, og mange bruker samme adresserom, 10.0.0.0/24. Enheter innenfor et gitt hjemmenettverk kan sende pakker til hverandre ved hjelp av 10.0.0.0/24 adressering. Imidlertid kan pakker videresendt utover hjemmenettverket til det større globale Internett, tydeligvis ikke bruke disse adressene (som enten en kilde eller en destinasjonsadresse) fordi det er hundrevis av tusenvis av nettverk som bruker denne adresseblokken.

Det vil si at 10.0.0.0/24 adressene bare kan ha betydning innenfor det oppgitte hjemmenettverket. Men hvis private adresser bare har mening innenfor et gitt nettverk, hvordan håndteres adressering når pakkene sendes til eller mottas fra det globale Internett, hvor adressene er nødvendigvis unike? Svaret ligger i å forstå NAT:

- Den NAT-aktiverte ruten ser ikke ut som en rute for omverdenen. I stedet oppfører NAT-ruten seg til omverdenen som en enkelt enhet med en enkelt IP-adresse. I Figur 4.22 har all trafikk som forlater hjemme-ruten for det større Internett en kilde-IP-adresse på 138.76.29.7, og all trafikk som kommer inn i hjem-ruten må ha en destinasjonsadresse på 138.76.29.7.
- I hovedsak skjuler den NAT-aktiverte ruten detaljene i hjemmenettverket fra omverdenen. Kanskje du lure på hvor hjemmenettverkene får adressene deres, og hvor ruten får sin eneste IP-adresse. Ofte er svaret det samme - DHCP!
  - Ruten får adressen fra Internett-leverandørens DHCP-server, og ruten kjører en DHCP-server for å gi adresser til datamaskiner innenfor NAT-DHCP-ruter-styrte hjemmenettverkets adresseplass.

Hvis alle datagrammer som kommer til NAT-ruten fra WAN, har **samme** destinasjons-IP-adresse, hvordan kjenner ruten den **interne verten** som den skal videresende et gitt datagram?

Trikset er å bruke et **NAT-oversettelsestabell** på NAT-ruten, og å inkludere *portnumre* samt *IP-adresser* i tabelloppføringene. Når en intern vert sender en pakke sendes det med intern IP-adresse samt et vilkårlig portnummer samt destinasjonsadresse. Hos ruten så lagres IP-adressen fra LAN-siden og WAN-siden og det interne portnummeret, samt et vilkårlig kildeportnummer for ruten. Dette lagres i oversettelsestabellen.

Når datagrammet fra destinasjonsadressen ankommer NAT-ruten, så vil ruten indekserer ruten NAT-oversettelsestabellen ved hjelp av destinasjonens IP-adresse og destinasjonsportnummer for å oppnå riktig IP-adresse (10.0.0.1) og destinasjonsportnummer (3345).

Har vært diskusjon rundt NAT-oversettelse da flere mener at portnummere skal brukes for å adressere prosesser, ikke adressere verter.

- Problem ved P2P, dersom Vert A ønsker å etablere TCP-tilkobling til Vert B, fungerer ikke dette, da Vert B ikke kan oppføre seg som en server og akseptere TCP-tilkobligner.
  - Mulig å unngå dette med **connection reversal / NAT traversal** ved at Vert A, spør Vert B gjennom en Vert C, som ikke er bak en NAT, om å initiere en TCP-tilkobling direkte tilbake til Vert A.

## UPnP

**NAT-traversal** leveres i økende grad av **Universal Plug and Play (UPnP)**, som er en protokoll som gjør at en vert kan oppdage og konfigurere et nærliggende NAT. UPnP krever at både verten og NATen er UPnP-kompatible. Med UPnP kan et program som kjører i en vert, be om en NAT-kartlegging mellom dens (private IP-adresse, privat portnummer) og (offentlig IP-adresse, offentlig portnummer) for noen forespurtt offentlig portnummer.

Hvis NAT aksepterer forespørselen og oppretter kartlegging, kan nodene fra utsiden initiere TCP-tilkoblinger til (offentlig IP-adresse, offentlig portnummer).

Videre lar UPnP programmet vite verdien av (offentlig IP-adresse, offentlig portnummer), slik at søknaden kan annonseres den til omverdenen.

Oppsummert lar UPnP eksterne verter å initiere kommunikasjonsøkter til NAT-edde verter, ved hjelp av enten TCP eller UDP. NAT har lenge vært en *nemesis* for P2P applikasjoner - UPnP, som gir en effektiv og robust NAT-traversal løsning, kan være deres frelser.

## Internet Control Message Protocol (ICMP)

ICMP brukes av verter og rutere til å kommunisere nettverkslagsinformasjon til hverandre. Den mest typiske bruken av ICMP er for *feilrapportering*. For eksempel, når du kjører en Telnet-, FTP- eller HTTP-økt, kan det hende du har oppdaget en feilmelding, for eksempel "Destination network unreachable".

Denne meldingen hadde sin opprinnelse i ICMP. På et tidspunkt kunne en IP-ruten ikke finne en vei til verten spesifisert i Telnet-, FTP- eller HTTP-applikasjonen. Denne ruten skapte og sendte en *type-3* ICMP melding til verten din som angir feilen.

ICMP er ofte ansett som en del av IP, men arkitektonisk ligger den like over IP, da ICMP-meldinger er ført inne i IP-datagrammer. Det vil si at ICMP-meldinger blir båret som IP-nytelast, akkurat som TCP- eller UDP-segmenter blir båret som IP-nytelast.

På samme måte, når en vert mottar et IP datagram med ICMP spesifisert som den øverste lagprotokollen, demultiplekse datagrammets innhold til ICMP, akkurat som det ville demultiplekse datagrammets innhold til TCP eller UDP.

- ICMP-meldinger har en *type* og et *kodefelt*, og inneholder overskriften og de første 8 byte av IP-datagrammet som forårsaket at ICMP-meldingen genereres i utgangspunktet (slik at avsenderen kan bestemme datagrammet som forårsaket feilen).
- Utvalgte ICMP-meldingstyper er vist i Figur 4.23. Merk at ICMP-meldinger ikke bare brukes til signalfeilforhold.
- Det velkjente pingprogrammet sender en *ICMP type 8 kode 0* melding til den angitte verden. Destinasjonsverden, ser ekkoforesporselen, sender tilbake en *type 0 kode 0 ICMP-ekko-svar*.

**Traceroute:** Sender ICMP-meldinger i vanlige datagrammer med UDP-segmenter, med usannsynlige UDP portnumre. Det første datagrammet har TTL på 1, neste har 2, tredje har 3, osv. Kilden starter også timeren for hver av datagrammene. Når det  $n$ -te datagrammet kommer til den  $n$ -te ruten, overerer den  $n$ -te ruten at TTL-en til datagrammet har gått ut, og sender en ICMP-warning melding til verden. Slik kan Traceroute karlegge antall og identiteten til ruterne mellom seg og destinasjonen.

## IPv6

På 1990 begynte de å se på en etterfølger til IPv4-protokollen - i hovedsak fordi de 32-bit IP-adressene begynte å bli brukt opp. Det ble så utviklet en ny IP protokoll, IPv6.

### IPv6 Datagram Format

Formatet til IPv6-datagrammet er vist i Figur 4.24 under. De viktigste endringene i IPv6 er tydelige i datagramformatet:

- *Expanded addressing capabilities.* IPv6 øker størrelsen på IP-adresser fra 32 til 128 bits. Dette gjør at verden ikke kommer til å gå tom for IP-adresser. Finnes også nå **anyone address**, som lar et datagram å bli levert til enhver av en gruppe av verter.
- *A streamlined 40-byted header.* Noen IPv4 felt har blitt droppet eller gjort valgfrie. Den resulterende 40-byte fast-lengde headeren tillater raskere prosessering av IP-datagrammet.
- *Flow labeling and priority.* IPv6 har en unnvikende definisjon av *flow (nor. flow)*. Dette tillater merking av pakker som tilhører bestemte strømmer som avsenderen ønsker spesiell håndtering for, for eksempel en ikke-standardkvalitet-tjeneste eller sanntids-tjeneste. For eksempel kan lyd og videooverføring sannsynligvis bli behandlet som en strømning. På den annen side kan de mer tradisjonelle applikasjonene, for eksempel filoverføring og e-post, ikke behandles som strømmer.

Følgende felt er definert i IPv6:

- *Version.* Dette 4-bit-feltet identifiserer IP-versjonsnummeret, IPv6 bærer en verdi av 6 i dette feltet.
- *Traffic class.* 8-bit felt som er nesten likt TOS-feltet hos IPv4
- *Flow label.* 20-bit felt som identifiserer flow-en til datagrammet.
- *Payload length.* 16-bit verdi som gir antall bytes som følger etter headeren.
- *Next header.* Feltet identifiserer protokollen som datafeltet i datagrammet skal bli levert til (f.eks. UDP eller TCP).
- *Hop limit.* Innholdet i dette feltet blir dekrementert med en hos hver rute som videresender datagrammet. Dersom hoplimit-en blir 0, vil datagrammet bli kastet.
- *Source and destination addresses.* IPv6-adressene til kilde og destinasjon, to felt på 128-bit.
- *Data.* Dette er payload-delen av IPv6 datagrammet.

Felt som har forsvunnet fra det vi så i IPv4:

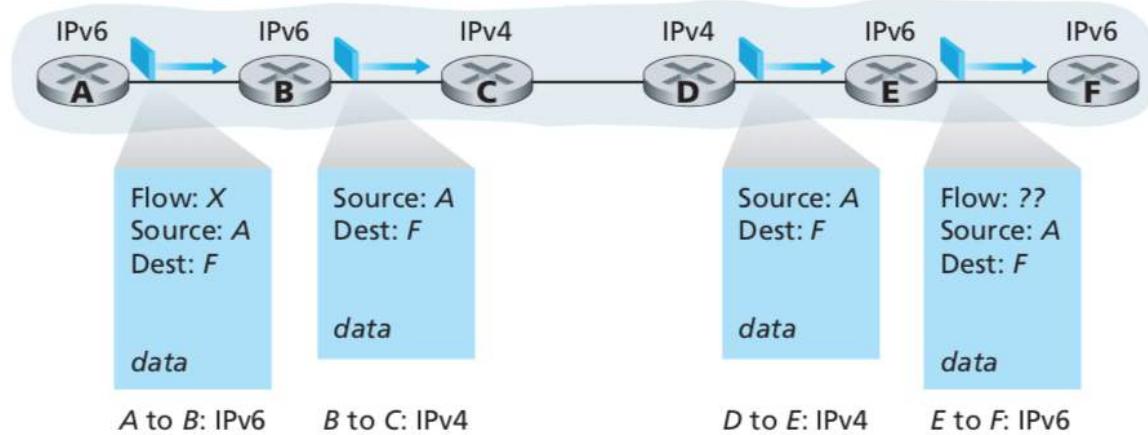
- *Fragmentation/Reassembly:* IPv6 tillater ikke fragmentering eller refragmentering hos mellomliggende rutere, dette kan kun bli gjort hos kilde eller destinasjon. Dersom en pakke kommer til en utgående link som har mindre kapasitet enn størrelsen til datagrammet, blir pakken droppet, og en ICMP-error-melding blir sendt tilbake.

- *Header checksum*: I IPv4, måtte checksummen bli regnet ut på nytt for hver ruter, pga den endrende verdien TTL. Dette tok tid, og som med fragmentering, var dette en for dyr operasjon i IPv4.
- *Options*. Et options-felt er ikke lenger en del av IP-headeren. Men er ikke borte, kan bruke *next headers*-feltet.

Overgangen fra IPv4 til IPv6:

Alternativer:

- Et alternativ ville være å erklære en flaggdag - en gitt tid og dato da alle Internett-maskiner ville bli slått av og oppgradert fra IPv4 til IPv6.
- Sannsynligvis den enkleste måten å introdusere IPv6-kompatible noder på, er en **dual-stack**-tilnærming, hvor IPv6-noder også har en fullstendig IPv4-implementering.
  - En slik knute, referert til som en IPv6 / IPv4 knutepunkt i RFC 4213, har evnen til å sende og motta både IPv4 og IPv6 datagrammer. Når du samarbeider med en IPv4-node, kan en IPv6 / IPv4-node bruke IPv4 datagrammer. Når du samarbeider med en IPv6-node, kan den snakke IPv6.
  - IPv6 / IPv4-noder må ha både IPv6- og IPv4-adresser
  - Bruke DNS for å bestemme om noder er kompatible med IPv4 eller IPv6.
  - *Problem*: Når et IPv6-datagram blir konvertert over til et IPv4-datagram, vil feltene hos IPv6-datagrammet som ikke har noen motpart i IPv4 forsvinne.
- Et alternativ til dual stack-tilnærmingen er kjent som **tunneling**. Tunneling kan løse problemet som er nevnt ovenfor. Den grunnleggende ideen bak tunneling er følgende. Anta at to IPv6-noder vil interopere ved hjelp av IPv6 datagrammer, men er koblet til hverandre med IPv4-rutere. Vi refererer til det settet med IPv4-rutere mellom to IPv6-rutere som en **tunnel**, som illustrert i figur 4.26. Ved tunneling tar IPv6-noden på senderens side av tunnelen hele IPv6 datagrammet og setter det i **data** (payload)-feltet til et IPv4 datagram, og sender det til den andre IPv6-ruten.

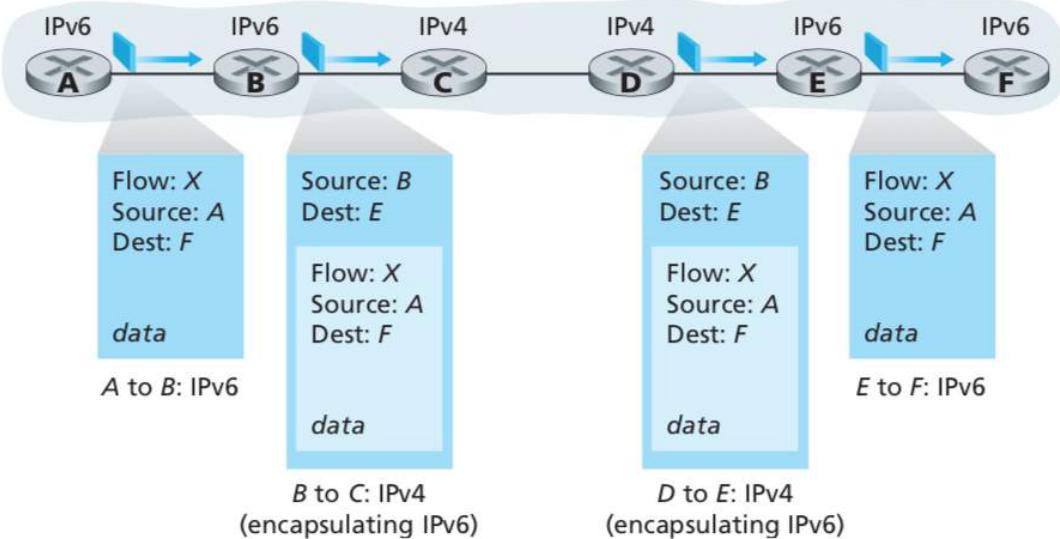


**Figure 4.25** ♦ A dual-stack approach

## Logical view



## Physical view



**Figure 4.26 ♦ Tunneling**

En viktig leksjon som vi kan lære av IPv6-opplevelsen er at det er enormt vanskelig å endre protokollene til nettverkslag

### Brief Introduction to IP Security

Med sikkerhet som en stor bekymring i dag, har Internettforskere flyttet videre til å designe nye protokoller for nettverkslag som gir en rekke sikkerhetstjenester.

En av disse protokollene er *IPsec*, en av de mest populære sikre nettverkslagprotokollene og også distribuert i Virtual Private Networks (VPN). Selv om IPsec og dets kryptografiske underlag er dekket i detalj i kapittel 8, gir vi en kort introduksjon på høyt nivå til IPsec-tjenester i denne delen.

IPsec er designet for å være bakoverkompatibel med IPv4 og IPv6. Spesielt, for å høste fordelene med IPsec, trenger vi ikke å erstatte protokollstabler i alle rutere og verter på Internett. Hvis du for eksempel bruker transportmodusen (en av to IPsec "mode"-ene), hvis to verter vil kommunisere sikkert, må IPsec bare være tilgjengelig i de to vertene. Alle andre rutere og verter kan fortsette å kjøre "vanilla IPv4".

Vanilla = Straight out of the box

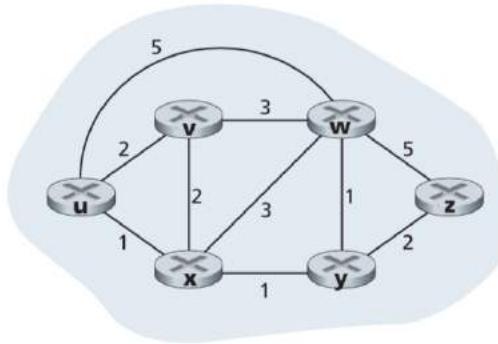
Vi skal nå se på IPsec sin transportmodus. På sendingssiden sender transportlaget et segment til IPsec. IPsec krypterer deretter segmentet, legger til flere sikkerhetsfelt i segmentet, og inkapsulerer den resulterende nyttelasten i et vanlig IP datagram. (Det er faktisk litt mer komplisert enn dette, som vi ser i kapittel 8.) Senderverten sender deretter datagrammet til Internett, som overfører det til mottakerverten. Der, dekrypterer IPsec segmentet og sender det ukrypterte segmentet til transportlaget.

Tjenestene som tilbys av en IPsec-økt inkluderer:

- *Cryptographic agreement*. Kommuniserende verter kan bli enige om crypto-algoritmer og nøkler
- *Encrypting of IP datagram payloads*. IPsec krypterer payloaden til datagrammet.
- *Data integrity*. IPsec tillater mottakende vert å verifisere at datagrammet ikke har blitt modifisert.
- *Origin authentication*. Mottakende vert kan være sikker på at kilde IP-adressen i datagrammet er den faktiske kilden til datagrammet.

## Routing Algorithms

Typisk er en vert koblet direkte til en ruter, den **defaulde ruteren** for verten (også kalt **first-hop ruter** til verten). Når en vert sender en pakke, vil pakken sendes til sin defaultruter. Vi refererer til defaultruteren til kilden som **source router**, og for destinasjonskilden **destination router**.



**Figure 4.27** ♦ Abstract graph model of a computer network

Figuren over viser en graf  $G = (V, E)$ , som har vektede kanter, og representerer et datanettverk. En routing algoritme skal identifisere den minst kostbare stien fra kilder og destinasjoner. Stien fra A til B som har minst total kost, kalles en **least-cost path**. En sti fra A til B med færrest mulig linker kalles **shortest path**. Dersom alle kanter i en graf har samme kost, vil least-cost path også være shortest path.

- En **global routing algoritme** regner ut least-cost pathen fra en kilde til destinasjon ved å bruke komplett, global kunnskap om nettverket. Algoritmen tar tilkoblingene mellom alle noder og alle cost-ene som input. Her kreves det at algoritmen på en eller annen måte kan få tak i all denne informasjonen før den utfører. Algoritmer med global tilstandsinformasjon refereres ofte til som **link-state (LS) algoritmer**.
- I en **desentralisert routing algoritme** gjøres kalkuleringen av least-cost path på en iterativ, distribuert metode. Ingen node har komplett informasjon om kostnader til alle nettverkslinker. Istedet begynner hver node med kun kunnskapen av kosten til sine egne direkte tilkoblede linker. Da, gjennom en iterativ prosess av kalkulering og forveksling av informasjon med nabonodene sine, kan en node etterhver kalkulere den least-cost pathen til destinasjonsnoden eller settet med destinasjoner. Den desentraliserte routing algoritmen som er pensum i TTM4100 er kalt en **distance-vector (DV) algoritme**, siden hver node vedlikeholder en vektor av estimatorer for costs (avstander) til alle de andre nodene i nettverket.

En annen måte å klassifisere routing algoritmer er om de er statiske og dynamiske routing algoritmer. Statiske endrer seg sjeldent over tid, mens dynamiske endrer routing paths når nettverkets trafikkmengder eller topologi endres.

En tredje måte er om algoritmen er load-insensitive eller load-sensitive. I algoritmer som er **load-sensitive** varierer costen til linkene dynamisk for å reflektere nivået av overbelastning på linken.

I dag er de fleste av internettets routing algoritmer **load-insensitive**, da linkens kostnad ikke eksplisitt reflekterer nåværende eller nylig nivå av overbelastning.

## Kapittel 5 - The Link Layer: Links, Access Networks, and LANs

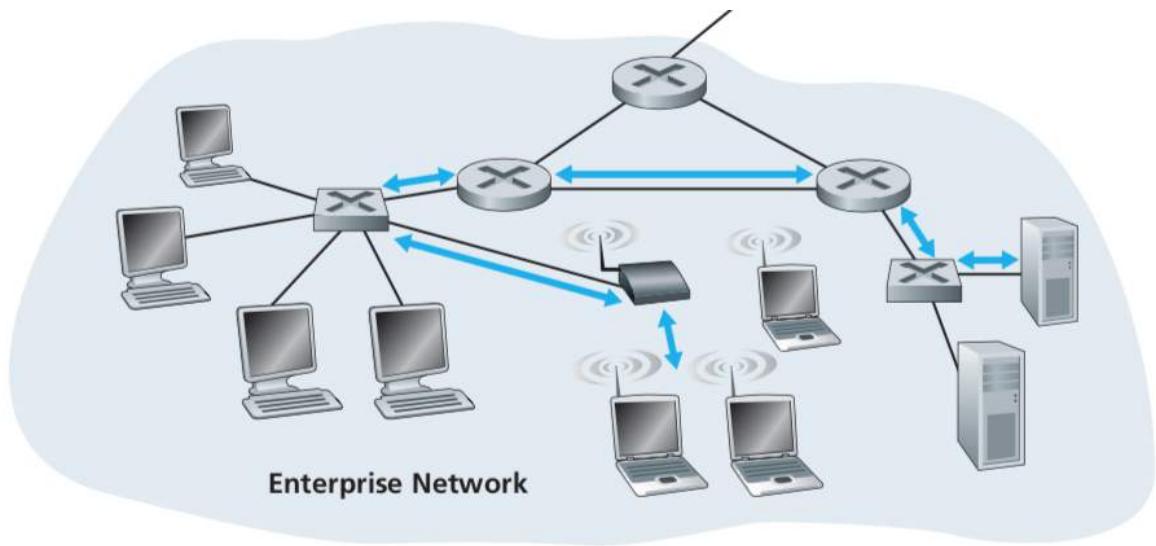
### Introduction to the Link Layer

Når vi skal diskutere linklaget, skal vi se at det to forskjellige typer med linklagskanaler. Den første typen er *broadcast channels*, som kobler sammen verter i trådløse LANs, satellittnettverk og hybrid fiber-coaxial cable (HFC) aksessnettverk. Den andre typen av linklags-channels er punkt-til-punkt kommunikasjonslinker, som gjerne finnes mellom to rutere koblet på en langdistansekobling.

I dette kapittelet vil jeg referere til enhver enhet som kjører en linklags-protokoll (layer 2) som en **node**. Noder inkluderer verter, rutere, svitsjer og WiFi-aksesspunkter. Jeg vil referere kommunikasjonskanaler som som sammenkobler to nabonoder langs en kommunisjonssti som en **kobling** (eng. **link**)

Se på eksempelet under i Figur 5.1, tenk at et datagram blir sendt fra den ene trådløse verten til en av serverene. Datagrammet vil da gå igjennom seks koblinger: en WiFi-link mellom den sendende verten og WiFi-aksesspunktet, en Ethernet-link mellom aksesspunktet og linklags-switchen, en link mellom linklags-switchen og ruter, en link mellom de to ruterne, en Ethernet-link mellom ruter og linklags-switch, og til slutt en Ethernet-link mellom svitjen og serveren.

Over en gitt link / kobling vil den overførende noden innkapsle datagrappet i en **linklagsframe**, og sender **framen** inn i linken.



**Figure 5.1** ♦ Six link-layer hops between wireless host and server

### The Services Provided by the Link Layer

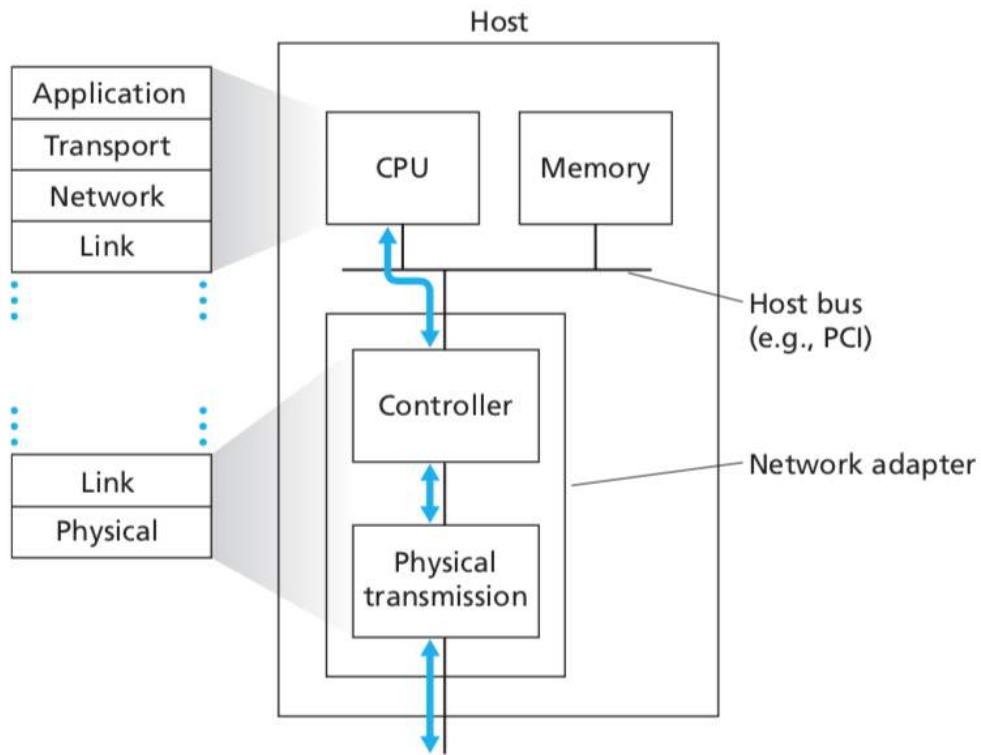
Mulige tjenester som kan bli tilbuddt av linklagsprotokoller inkluderer:

- *Framing*. Nesten alle linklagsprotokoller innkapsler hver nettverklags-datagram inn i en linklagsframe før overføring over en link. En frame består av et datafelt, der datagrammet blir innsatt, og et antall header-felt.
- *Link access*. En medium access control (MAC) protocol angir reglene for en frame som skal bli overført på linken. For punkt-til-punkt-linker som kun har en sender på den ene enden av linken og en enkelt mottaker på den andre siden av linken, er MAC protokollen enkel - senderen kan sende en frame når linken ikke er opptatt.
- *Reliable delivery*. Når en linklagsprotokoll tilbyr pålitelig leveringstjeneste, garanterer den å bevege hvert nettverksdatagram over linken uten feil. På samme måte som transportlags pålitelig leveringstjeneste, en linklags pålitelig leveringstjeneste kan bli oppnådd med acknowledgements og re-sendinger. En pålitelig leveringstjeneste for koblinger er ofte brukt for koblinger som er utsatt for høye feilrater, for eksempel en trådløs kobling. Rdt på linklagsprotokoller er ansett å være en unødvendig overhead, og derfor er det mange kablede-protokoller som ikke tilbyr pålitelig leveringstjeneste.
- *Error detection and correction*. Linklagsmaskinvaren i en mottakede kan feilaktig bestemme at en bit i en ramme er null når den ble overført som en, og omvendt. Slike bitfeil innføres ved signaldemping og elektromagnetisk støy. Fordi det ikke er behov for å videresende et datagram som har en feil, gir mange linklagsprotokoller en mekanisme for å oppdage slike bitfeil.

### Where Is the Link Layer Implemented

For det meste, er linklaget implementert i en **nettverksadapter** (eng. **network adapter**), også kjent som et **network interface card (NIC)**. I hjertet nettverksadapteren er linklagskontrolleren, vanligvis en enkel, spesial-tilfelle chip som implementerer mange av linklagstjenestene (framing, link access, error detection, m.m.). Dermed er mye av en linklagets kontrollerfunksjonalitet implementert i maskinvare.

Figur 5.2 viser en typisk vertarkitektur:



**Figure 5.2** ♦ Network adapter: its relationship to other host components and to protocol stack functionality

- På den sendende siden tar kontrolleren et datagram som har blitt laget og lagret i vertens minne av høyere lag i protokollstakken, innkapsler datagrammet i en linklagsframe, og snder frammen inn i kommunikasjonskoblingen, etter link-access-protokollen.
- På den mottakende siden, en kontroller mottar en hel frame, ekstraherer nettverklags-datagrammet.

Dersom linklaget utfører feildeteksjon, er det sendekontrolleren som setter *feildetekteringsbitene* i frame-koden, og det er mottakskontrollen som utfører *feilsøking*.

Som bilde over viser, så ser vi at mesteparten av linklaget er implementert i maskinvare, men også at deler av linklaget er implementert i programvare som kjører på vertens CPU.

Dette er stedet i protokollstakken hvor programvare møter maskinvare

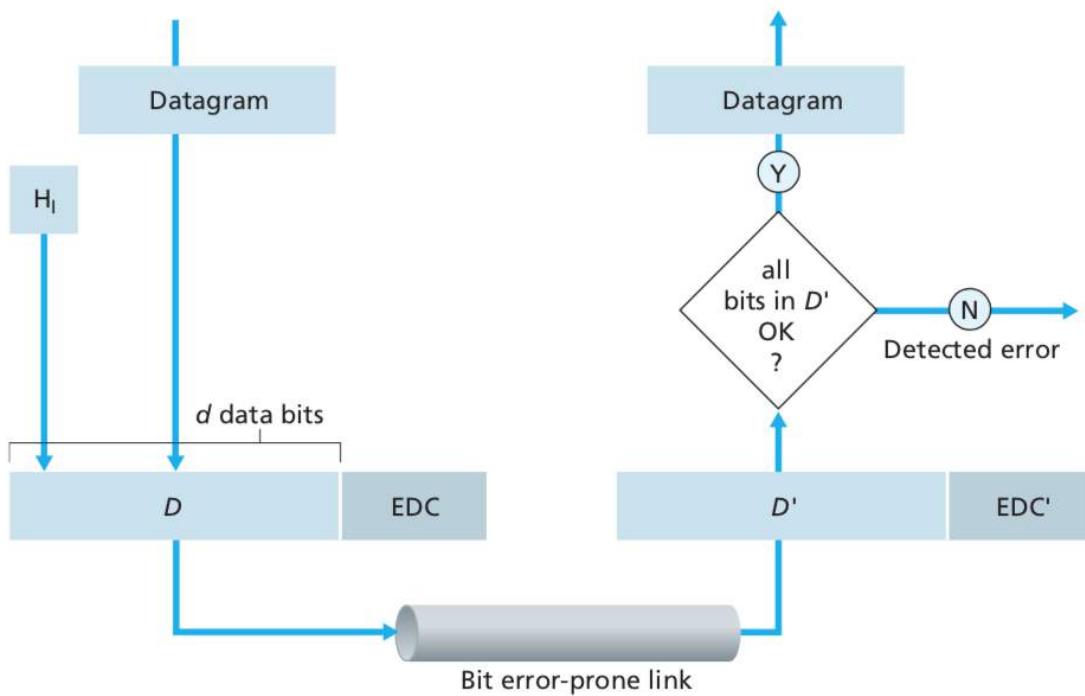
### Error-Detection and -Correction Techniques

I forrige avsnitt, noterte vi at **bit-level error detection og correction** - finne og rette på korrupte bits i linklagsframe-er sendt mellom to nabonoder - er to tjenester som leveres av linklaget.

Figur 5.3 illustrerer innstillingen for studien vår. Ved sendingskoden blir data,  $D$ , som skal beskyttes mot bitfeil, forsterket med feildetekterings- og korrigerbels (EDC - Error Detection Correction). Vanligvis inkluderer dataene som skal beskyttes, ikke bare datagrammet som sendes ned fra nettverkslaget for overføring over lenken, men også adresseringsinformasjon, sekvensnumre og andre felt i koblingsrammeoverskriften.

Både  $D$  og EDC sendes til mottakskoden i et link-nivå ramme. Ved mottakskoden mottas en sekvens av biter,  $D'$  og  $EDC'$ . Vær oppmerksom på at  $D'$  og  $EDC'$  kan avvike fra den opprinnelige  $D$  og  $EDC$  som et resultat av i-overføring-bitflips.

Mottakerends oppgave blir da å sjekke om  $D'$  er den samme som originalen  $D$ .



**Figure 5.3** ♦ Error-detection and -correction scenario

Selv med bruken av feildeteksjonbits (EDC) kan det fortsatt være uoppdagede bit-feil

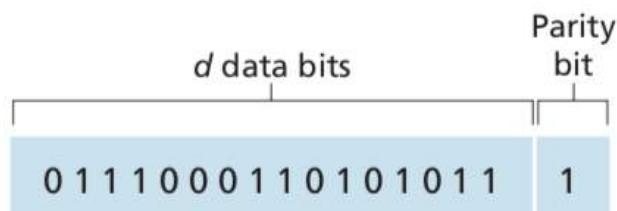
Vi skal nå se på tre teknikker for å finne feil i overført data – paritetssjekker (illustre basisideén bak feildeteksjon og korrekjon), checksumming metoder (brukes typisk mer i transportlaget) og cyclic redundancy checks (CRC – som typisk brukes i linklaget, i en adapter).

#### Parity Checks

Den enkelste formen for feildeteksjon er å bruke en enkelt **paritetsbit**. Anta at dataen,  $D$ , som ble sendt i Figur 5.4, har  $d$  bits.

I en *partalls-paritetsskjema*, vil senderen inkludere en ekstra bit og velger den verdi slik at antallet 1-ere i de  $d + 1$  bitene (data + paritetsbit) er partall. For *oddetalls-paritetsskjemaer*, er paritetsbiten bestemt slik at det er oddetall antall 1-ere.

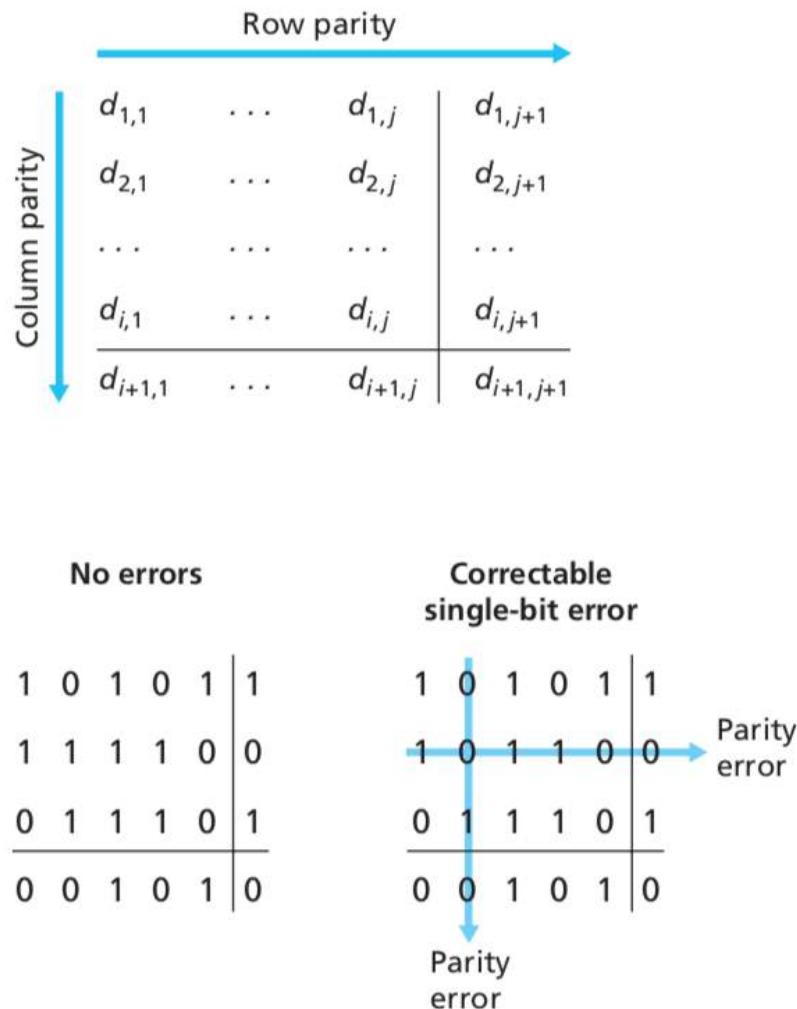
Mottakeren må kun telle antall 1-ere i de mottatte  $d + 1$  bitene. Dersom det er oddetall 1-ere er funnet i et partalls-paritetsskjema, vet mottakeren at minst en bit-feil har funnet sted (eller mer presist, et *odd*e antall bitfeil).



**Figure 5.4** ♦ One-bit even parity

Dersom man antar at det veldig lav sannsynlighet for bit-feil og at dersom ett finner sted er det lite sannsynlig at et annet også finner sted i samme datagram, er en-bit-paritetkjekker nok. Dersom dette ikke er tilfelle må man bruke **to-dimensjonal paritetkjemaer**. Her blir de  $d$  bitene i  $D$  delt inn i  $i$  rader og  $j$  kolonner. En paritetsverdi er regnet ut for hver rad og for hver kolonne. Dette resulterer i  $j + i + 1$  paritetsbits utgjør linklagsrammene feil-deteksjonsbits. Med rader og kolonner så kan mottaker ikke bare finne ut at det har skjedd en bitfeil, men også bruke rad- og kolonnenummeret for å rette opp bitfeilen.

To-dimensjonal paritet kan også *finne* enhver kombinasjon av to feil, men ikke *rette* de.



**Figure 5.5** ♦ Two-dimensional even parity

- Mottakerens evne til både å detektere og rette feil er kjent som **forward error correction (FEC)**

#### Checksumming Methods

I checksum-metoder, blir de  $d$  bitene med data i Figur 5.4 behandlet som en sekvens av  $k$ -bit tall. En enkel checksum-metode er å enkelt summere disse  $k$ -bit tallene og bruke denne resulterende summen som feildeteksjonbitsene.

- Internet checksum-en** er basert på denne metoden - bytes med data er behandlet som 16-bit tall, og summert. 1s komplementet av denne summen former Internettets checksum og blir bært i segmentets header. Som diskutert i Kap. 3 sjekker mottakeren checksummen ved å ta 1s komplementet av summen av den mottatte dataen (inkludert checksum-en), og sjekker om alle bitsene i resultatet er 1. Dersom en av bitsene er 0 indikerer det feil.
- I TCP og UDP protokollene, er Internett checksum-en regnet ut over alle felt (header og data)
- I IP er checksummen utregnet kun over IP-headeren (Siden UDP- eller TCP-segmentet har sin egen checksum)

Checksumming krever relativt lite pakke overhead. For eksempel krever checksummene i TCP og UDP kun 16 bits. Imidlertid gir de relativt svak beskyttelse mot feil sammenlignet med *cyclic redundancy check*, som diskuteres nedenfor, og som ofte brukes i linklaget.

## Cyclic Redundancy Check (CRC)

En feildeteksjonsteknikk brukt over hele dagens datamaskin nettverk er basert på **cyclic redundancy check (CRC) codes**. CRC koden er også kjent som **polynomial codes**, siden det er mulig å se bitstrengen som skal sendes som et polynom, der koeffisienter er 0- og 1-verdiene i bitstrengen, med operasjoner på bitstrengen tolket som polynom aritmetikk.

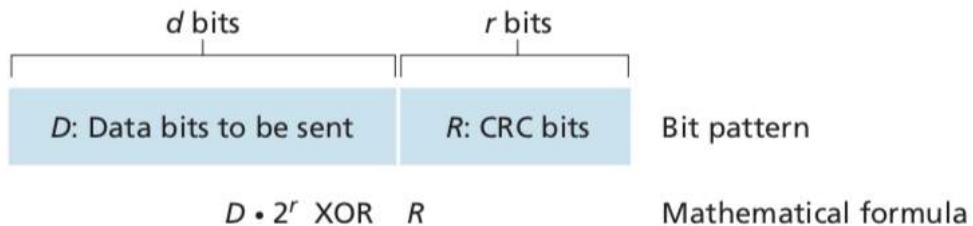
CRC koder opererer som følgende. Vi har den  $d$ -bit biten med data,  $D$ , som skal sendes mellom to noder. Senderen og mottakeren må først bli enige om et  $r + 1$  bit mønster, kjet som en **generator**, som blir betegnet som  $G$ . Vi krever dog at det mest signifikante (mest venstre) biten i  $G$  er 1.

Nøkkelideen bak CRC koder er vist i Figur 5.6. For et gitt stykke data,  $D$ , vil senderen velge  $r$  tilleggsbits,  $R$ , og legge dem til  $D$  slik at det resulterende  $d + r$  bitmønsteret (binært tall) er nøyaktig delelig med  $G$  (det vil si ingen rest) ved bruk av modulo-2 aritmetisk.

Feilsøkingsprosessen med CRC er så enkel:

- Mottakeren deler de mottatte bitene  $d + r$  med  $G$ . Hvis resten er *ikke-null*, vet mottakeren at en *feil* har oppstått; ellers blir dataene akseptert som *riktige*.

Alle CRC-kalkuleringer er gjort i modulo-2 aritmetikk uten carries i addisjon eller løn i subtraksjon. Dette betyr at addisjon og subtrasjon er identiske, og begge er ekvivalente med \*bitwise exclusive-or (XOR) operanden)

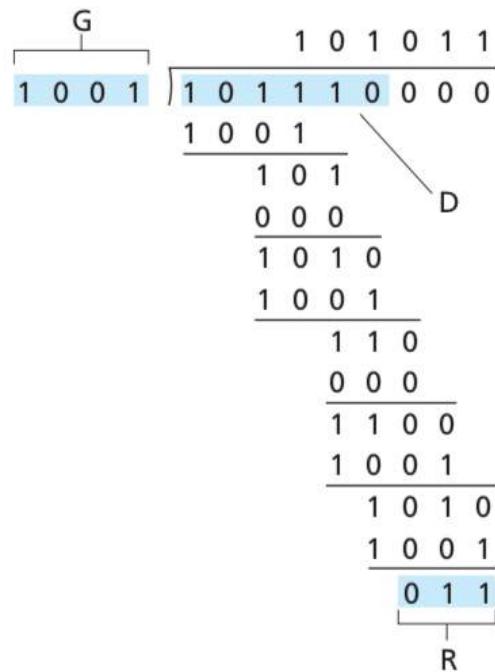


**Figure 5.6 ♦ CRC**

Internasjonale standarder har blitt definert for 8-, 12-, 16-, og 32-bit generatorer,  $G$

This equation tells us that if we divide  $D \cdot 2^r$  by  $G$ , the value of the remainder is precisely  $R$ . In other words, we can calculate  $R$  as

$$R = \text{remainder} \frac{D \cdot 2^r}{G}$$



**Figure 5.7** ♦ A sample CRC calculation

### Multiple Access Links and Protocols

Fra introduksjonen husker vi at det finnes to typer nettverkskoblinger: **point-to-point link** - en sendernode og en mottakernode, og **broadcast link** - flere sendende noder og flere mottakende, hver sendte pakke blir sendt til alle noder.

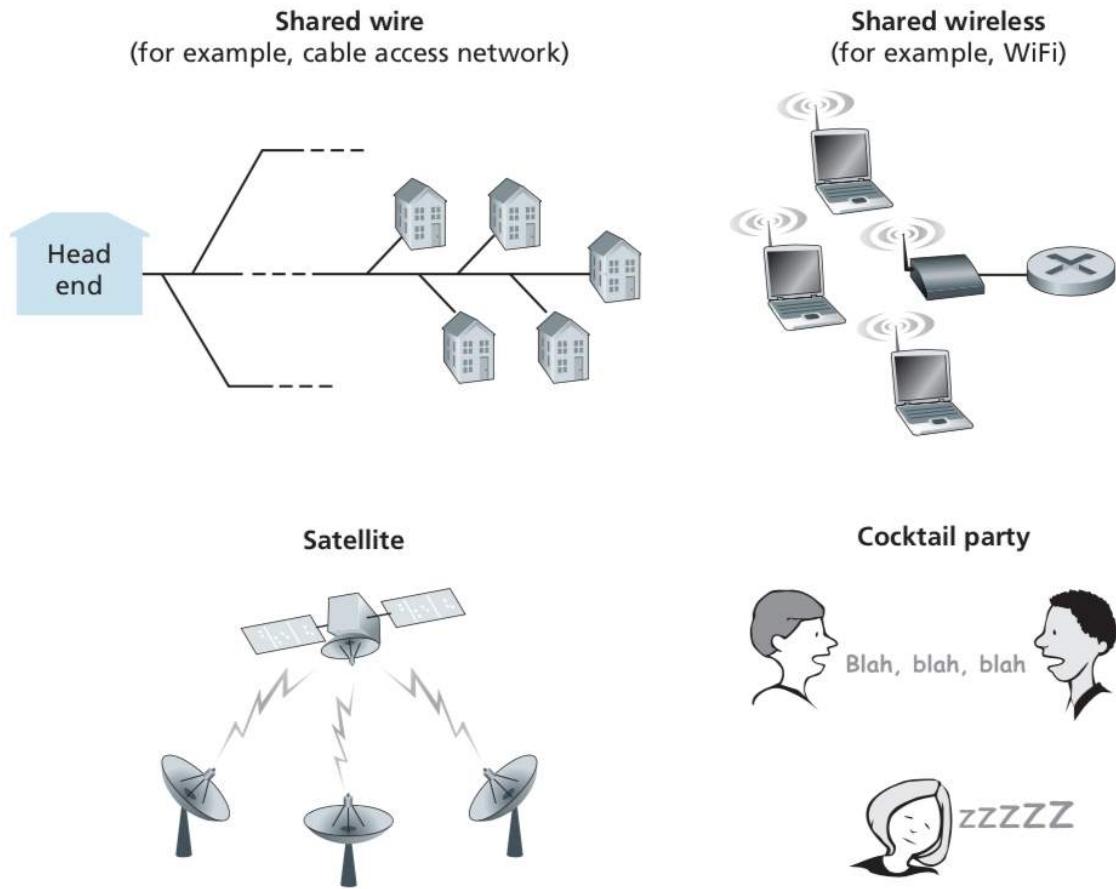
Vi skal se på et problem som er sentralt i linklaget: hvordan koordinere aksessen til flere sendende og mottakende noder i en delt broadcast channel

Broadcast-channels er ofte brukt i LANs, nettverk som er geografisk konsentrent i en bygning, universitet eller lignende.

For at flere noder skal kunne snakke sammen må man ha visse protokoller, og disse er bestemt av såkalte **multiple access protocols**.

Ettersom alle noder er kapable for å sende frames, kan flere enn to noder sende frames samtidig. Når dette skjer, vil alle nogene motta flere frames samtidig - de sendte rammene vil da **kollidere** hos alle mottakerne. Typisk når det er en kollisjon skjer, kan ingen av de mottakende nogene få noe ut av de kolliderte rammene.

For å sikre at kringkastingskanalen utfører nytlig arbeid når flere noder er aktive, er det nødvendig å koordinere overføringen av de aktive nogene på en eller annen måte. Denne koordineringsjobben er ansvaret for tilgangsprotokollen.



**Figure 5.8** ♦ Various multiple access channels

Vi klassifiserer *multiple access protocol*-er i tre categorier: **channel partitioning protocols**, **random access protocols**, og **taking-turns protocols**.

Vi kan konkludere at en multiple access protocol for en broadcast channel med rate  $R$  bits per sekund, burde ha følgende ønskede egenskaper:

1. Når en node har data å sende, har den noden en overgangshastighet på  $R$  bps.
2. Når  $M$  noder har data som skal sendes, har hver av disse node en gjennomstrømning på  $R / M$  bps. Dette trenger nødvendigvis ikke å være at hver av de  $M$  nodene alltid har en øyeblikkelig hastighet på  $R / M$ , men heller at hver node skulle ha en gjennomsnittlig overføringshastighet på  $R / M$  over noe passende definert intervall av tiden.
3. Protokollen er desentralisert, dvs. det er ingen masternode som representerer et enkelt feilpunkt for nettverket.
4. Protokollen er enkel, så den er billig å implementere.

### Channel Partitioning Protocols

Minner om fra Kap. 1 at **time-division multiplexing (TDM)** og **frequency-division multiplexing (FDM)** er to teknikker som brukes for å partisjonere en broadcast channels båndbredde blant alle noder som bruker kanalen.

Anta at en **kanal** (eng. *channels*) støtter  $N$  noder og at overføringsraten til kanalen er  $R$  bps. TDM deler tid inn i **time frames** (nor. *tidsramme*) og deler hver tidsramme inn i  $N$  **time slots** (nor. *tidsluker*).

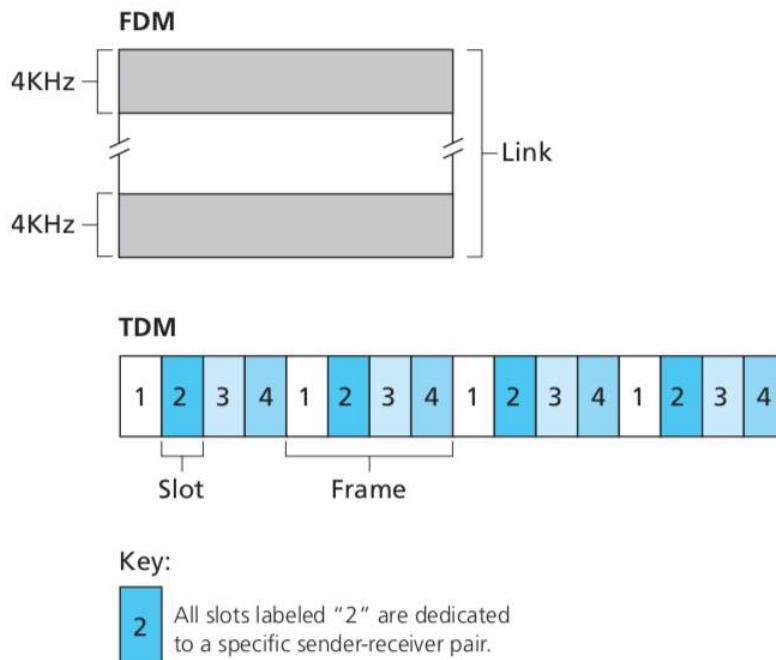
Når en node har en linklagsframe å sende, så overfører den framens bits under en tildelt tidsluke i den roterende TDM-rammen. Typisk er tidsluke-størrelsene bestemt slik at en enkelt pakke kan bli overført under en tidsluke.

Figur 5.9 under viser en enkel fire-noders TDM-eksempel. Et som en samtale som går i sirkel, der hver person får prate i en gitt tid, så den neste personen, når alle personene har fått snakket, så begynner mønsteret på nytt.

- TDM er tiltrekkende fordi det eliminerer kollisjoner og er helt fair. Alle noder får dedikert en overføringsrate  $R/N$  bps under hver tidsramme.

- Likevel er det noen negative sider: Hver node er begrenset av en øvre gjennomsnittlig overføringsrate på  $R/N$  bps, selv når det kun er en node som vil sende pakker, og hver node er nødt til å vente på sin tur.

Mens TDM deler inn broadcast channelen i tid, deler FDM Rbps-kanalen i forskjellige frekvenser (hver med en båndbredde på  $R/N$ ) og tilordner hver frekvens til en av  $N$  nodene. FDM skaper dermed  $N$  mindre kanaler med  $R / N$  bps ut av den enkle, større  $R$  bps-kanalen. FDM deler både fordelene og ulempene til TDM. Det unngår kollisjoner og deler båndbredden ganske mellom de  $N$  nodene. Imidlertid har FDM også en viktig ulempe som med TDM - en node er begrenset til en båndbredde på  $R / N$ , selv når det er den eneste noden med pakker som skal sendes.



**Figure 5.9** ♦ A four-node TDM and FDM example

En tredje kanalpartisjoneringsprotokoll er **code division multiple access (CDMA)**. Mens TDM og FDM tildeler tidsluker og frekvenser til henholdsvis nodene, tilordner CDMA en forskjellig kode til hver node. Hver node bruker deretter sin unike kode for å kode de databitene den sender. Hvis kodene er valgt nøyne, har CDMA-nettverk den fantastiske egenskapen at forskjellige noder kan overføre samtidig, og likevel har de respektive mottakere riktig mottatt avsenderens kodede databit (forutsatt at mottakeren vet avsenderens kode) til tross for forstyrrende overføringer av andre noder.

### Random Access Protocols

Den andre klassen av multiple access protokoller er **random access protokoller**. I en random access-protocol overfører en overføringsnode alltid ved full hastighet av kanalen, nemlig  $R$  bps. Når det er en kollisjon, sender hver node involvert i kollisjonen gjentatte ganger sin ramme (det vil si pakke) til rammen kommer gjennom uten kollisjon. Men når en node opplever en kollisjon, sender den ikke nødvendigvis straks rammen tilbake. I stedet venter det en *tilfeldig* forsinkelse før re-sending av rammen.

Hver node involvert i en kollisjon velger uavhengige tilfeldige forsinkelser. Fordi tilfeldige forsinkelser er valgt uavhengig, er det mulig at en av nodene vil velge en forsinkelse som er tilstrekkelig mindre enn forsinkelsene til de andre kolliderende nodene, og vil derfor kunne snike sin ramme inn i kanalen uten kollisjon.

Det finnes dusinvis av tilfeldige tilgangsprotokoller som er beskrevet i litteraturen. I denne delen beskriver vi noen av de mest brukte random access-protokollene - *ALOHA*-protokollene og *CSMA*-protokollene (carrier sense multiple access). Ethernet er en populær og distribuert CSMA-protokoll.

### Slotted ALOHA

I beskrivelsen av slotted ALOHA, antar vi følgende:

- Alle linklagsrammer består av nøyaktig  $L$  bits.

- Tiden er delt inn i luke av  $L/R$  sekunder - atslå tiden det tar å overfør en ramme.
- Noder starter å overføre rammen kun i begynnelsen av en luke.
- Nodene er synkroniserte slik at hver node ved når luken begynner.
- Dersom to eller flere rammer kolliderer i en luke, da vil alle nodene detektere kollisjonen før luken ender.

La  $p$  være en sannsynlighet (eng. *probability*), altså et tall mellom 0 og 1. Operasjonen til slotted ALOHA ved hver node er enkel:

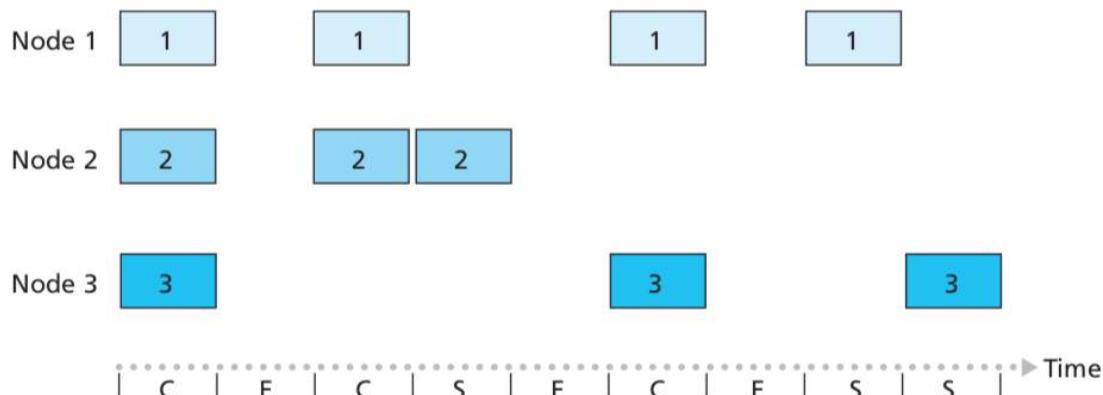
- Når en node har en ny ramme å sende, venter den til begynnelsen av neste luke og overfører hele rammen i luken.
- Dersom det ikke er en kollisjon har noden suksessfullt overført rammen, og den trenger ikke å tenke på å re-sende rammen. (Noden kan nå forberede en ny ramme for overføring, hvis den har det)
- Dersom det er en kollisjon, detekterer noden kollisjonen før luken er ferdig. Da vil noden re-sende rammen i den etterfølgende luken med en sannsynlighet  $p$ , helt til rammen er sendt uten en kollisjon

Sannsyneligheten for at pakken blir re-sendt er  $p$ , og da er implisitt sannsyneligheten for at den ikke blir sendt  $(1 - p)$

Slotted ALOHA ser ut til å ha mange fordeler. I motsetning til channel partitioning, tillater slotted ALOHA en node å sende kontinuerlig med full hastighet,  $R$ , når noden er den eneste aktive noden. (En node sies å være aktiv hvis den har rammer for å sende.)

Slotted ALOHA er også svært *desentralisert*, fordi hver node oppdager kollisjoner og bestemmer seg selv når den skal sendes om igjen. (Slotted ALOHA krever imidlertid at lukene skal synkroniseres i nodene, om kort vi skal diskutere en unslotted versjon av ALOHA-protokollen, så vel som CSMA-protokoller, ingen av dem krever slik synkronisering.) Slått ALOHA er også en ekstrem enkel protokoll.

- En luke der nøyaktig én node overfører, sies å være en **vellykket luke**. **Effektiviteten** til en slotted multiple access protokoll er definert til å være den langsiktige brøkdelen av vellykkede luke i tilfellet når det er et stort antall aktive noder, som hver har et stort antall rammer å sende.



#### Key:

- C = Collision slot
- E = Empty slot
- S = Successful slot

**Figure 5.10** ♦ Nodes 1, 2, and 3 collide in the first slot. Node 2 finally succeeds in the fourth slot, node 1 in the eighth slot, and node 3 in the ninth slot

- Sannsynligheten for at en gitt node har en suksess er  $p(1 - p)^{N-1}$
- Fordi det er  $N$  noder, er sannsynligheten for at noen av  $N$  noder har en suksess  $Np(1 - p)^{N-1}$ .
- Med litt utregning får man at den effektive overføringsraten til kanalen ikke er  $R$  bps, men kun  $0.37 \cdot R$  bps (Det er kun 37%)

Slotted ALOHA protokollen krever at alle noder synkroniserer overføringene deres til å starte på begynnelsen av hver luke. Den første ALOHA-protokollen var faktisk en uslotted, fullstendig desentralisert protokoll. I ren ALOHA, når en ramme først ankommer (det vil si at et nettverkslagsdatagram sendes ned fra nettverkslaget ved sendingsnoden), overfører noden straks rammen i sin helhet inn i kringkastingskanalen (*eng broadcast channel*). Hvis en overført ramme opplever en kollisjon med en eller flere andre sendinger, vil noden så umiddelbart (etter fullstendig sending av sin kolliderte ramme) sende rammen tilbake med sannsynligheten  $p$ . Ellers venter noden for en rammeoverføringstid. Etter denne venturen overfører den deretter rammen med sannsynligheten  $p$ , eller venter (gjenværende tomgang) for en annen rammetid med sannsynlighet  $(1 - p)$ .

- Sannsynligheten for at ingen andre noder sender i intervallet når sendingsnoden sender er  $(1 - p)^{N-1}$
- Sannsynligheten for en suksessfull overføring (og at ingen andre sender i nodens sendeintervall) blir da  $p(1 - p)^{2(N-1)}$ .
- Dette gir en maksimum effektivitet på un  $1/(2e)$ , som nøyaktig er halvparten av max. effekten til slotted ALOHA. Det er prisen å betale for en fullt desentralisert ALOHA protokoll.

### Carrier Sense Multiple Access (CSMA)

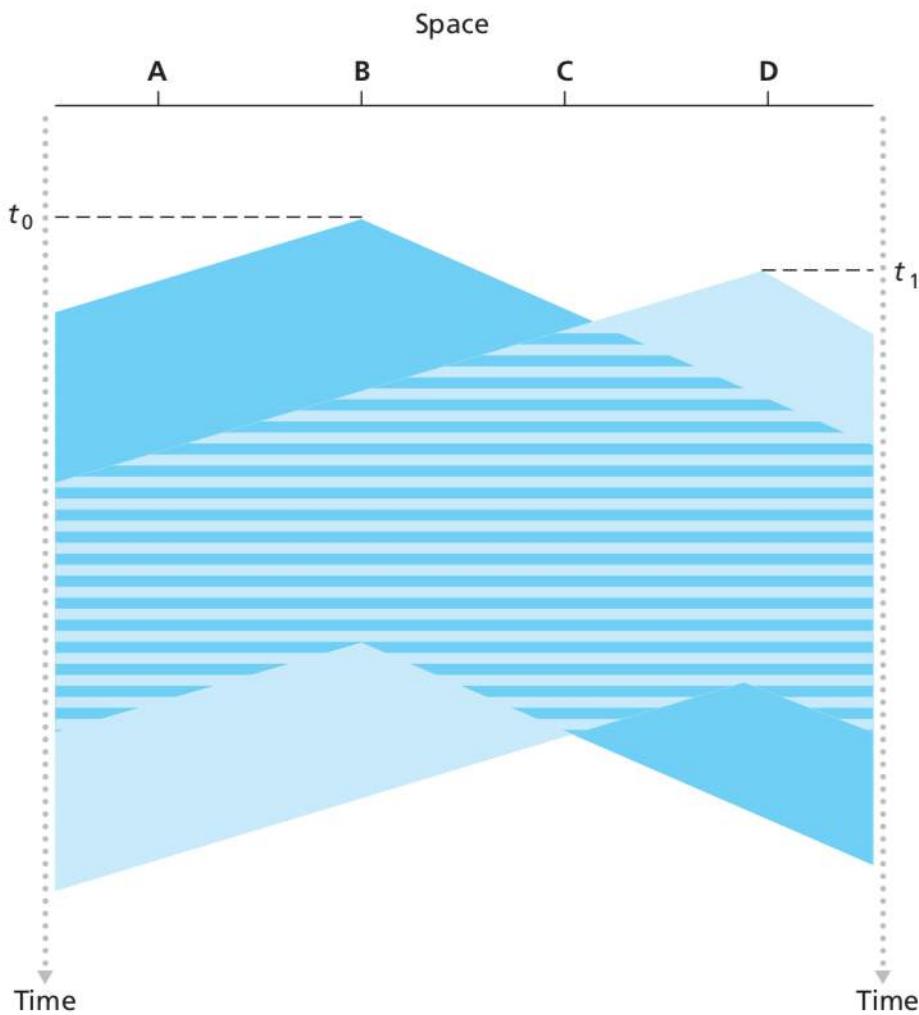
I både slotted og ren ALOHA blir en nodes beslutning om å overføre uavhengig av aktiviteten til de andre noder som er festet til kringkastingskanalen. Spesielt legger ikke en node merke til om andre noder holder på å overføre når den begynner å overføre, eller stopper å overføre når en annen node begynner å blande seg med overføringen.

Som mennesker har vi menneskelige protokoller som tillater oss ikke bare å oppføre seg med mer sårbarhet, men også å redusere mengden tid som "kolliderer" med hverandre i samtale og dermed øke mengden data vi bytter i samtalene våre. Spesielt er det to viktige regler for høflig menneskelig samtale:

- *Listen before speaking.* Dersom noen snakker, venter man til de er ferdige. I nettverksverdenen kaller vi dette **carrier sensing** - en node hører på kanalen før den overfører. Dersom en ramme fra en annen node er i ferd med å bli overført inn i kanalen venter den til er et lite mellomrom uten overføringer.
- *If someone else begins talking at the same time, stop talking.* I nettverksverdenen kaller vi dette **collision detection** - en overførende node hører på kanalen når den overfører. Dersom den merker at en annen node overfører en konflikt-ramme, stopper noden overføringen og venter en tilfeldig stund før den begynner å denne *sense-and-transmit-when-idle* sykelen.

Disse to reglene er kjennetegnet i familien til **carrier sense multiple access (CSMA)** og CSMA med kollisjonsdeteksjon (**CSMA / CD**) protokoller.

Ettersom alle noder utfører carrier-sensing, vil det forekomme kollisjoner? Svaret på dette kan bli illustrert i et space-time diagram. Figur 5.12 viser et space-time diagram med fire noder (A, B, C, D) koblet til en broadcast bus. Den horisontale aksen representerer posisjonen til hver node i rommet, den vertikale aksen representerer tiden.



**Figure 5.12** ♦ Space-time diagram of two CSMA nodes with colliding transmissions

- Ved tid  $t_0$ , merker node B at kanalaen kjører på tomgang, og at ingen andre noder overfører. Node B begynner å overføre, med bits som forplanter seg begge veier i broadcastmediumet.
- Den nedgående forplantningen av B's bits i figur 5.12 med økende tid indikerer at en ikke-null tid er nødvendig for at B bits faktisk propagerer (om enn i nærheten av lysets hastighet) langs broadcastmediet.
- På tidspunktet  $t_1$  ( $t_1 > t_0$ ) har noden D en ramme som skal sendes. Selv om node B for tiden overfører ved tid  $t_1$ , har ikke bitene som overføres av B ennå ikke kommet til D, og dermed registrerer D at *kanalen er tom* ved  $t_1$ .
- I samsvar med CSMA-protokollen begynner D derfor å sende sin ramme. Kort tid senere begynner Bs overføring å forstyrre Ds overføring ved D. Fra figur 5.12 er det tydelig at ende-til-ende **kanalforplantningsforsinkelsen** til en kringkastingskanal - tiden det tar for et signal å forplante seg fra en av nodene til en annen - vil spille en viktig rolle når det gjelder å bestemme ytelsen.

Jo lengre denne forplantningsforsinkelsen, desto større sjanse for at en carrier-sensing node ikke er i stand til å fornemme en overføring som allerede har begynt på en annen node i nettverket.

#### #####Carrier Sense Multiple Access with Collision Detection (CSMA/CD)

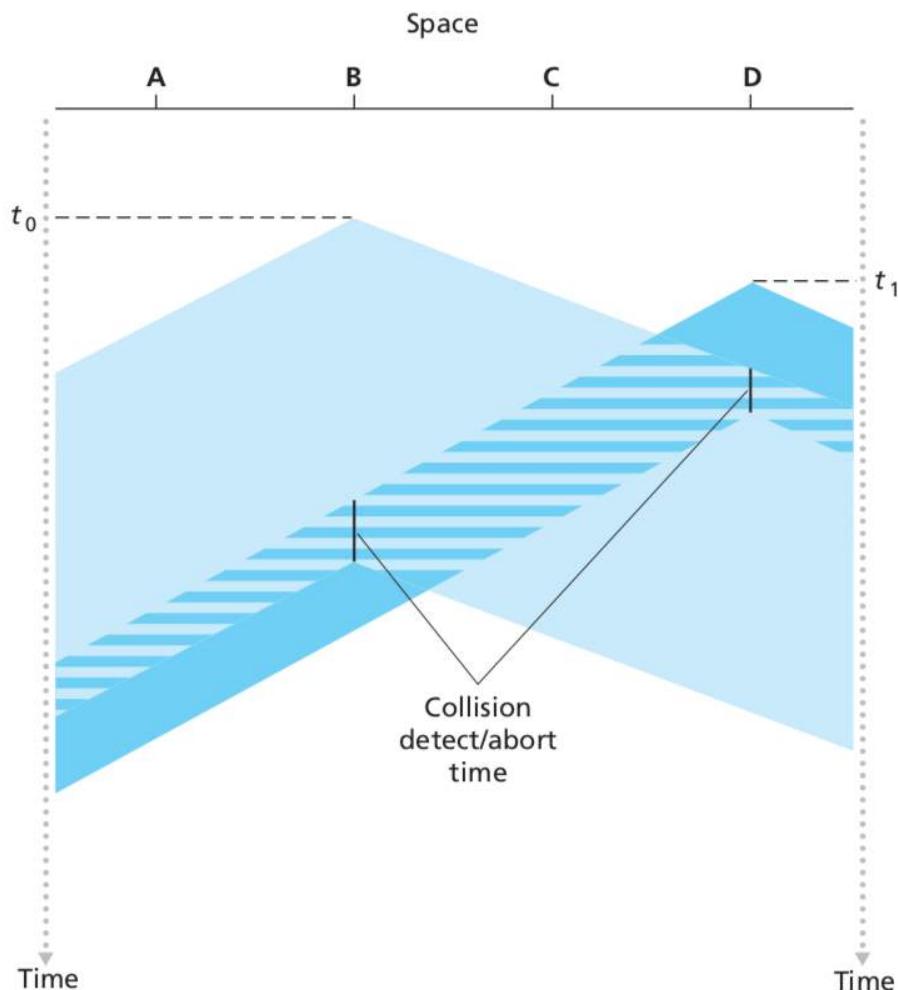
I Figur 5.12, er det ingen av noe de utfører kollisjonsdeteksjon (CD), både B og D fortsetter å overføre rammene sine, selv om en kollisjon har skjedd.

Når en node utfører kollisjonsdeteksjon, stopper den overføringen så snart den oppdager en kollisjon. Figur 5.13 viser det samme scenariet som i Figur 5.12, bortsett fra at de to noderne avbryter overføringen kort tid etter detektering av en kollisjon.

Det er klart at å legge til kollisjonsdeteksjon i en protokoll hjelper protokollutførelsen ved ikke å overføre en ubruklig, skadet ramme.

Før vi analyserer CSMA / CD-protokollen, la oss nå oppsummere operasjonen fra perspektivet til en adapter (i en knutepunkt) knyttet til en kringkastingskanal:

1. Adapteren får et datagram fra nettverkslaget, forbereder en linklagsramme, og setter rammen i adapterbufferen.
2. Hvis adapteren registrerer at kanalen er ledig (ingen signal-energi), begynner den å overføre rammen. Hvis adapteren derimot oppdager at kanalen er opptatt, venter den til den er ledig og begynner deretter å overføre rammen.
3. Under overføringen overvåker adapteren for tilstedeværelse av signalenergi som kommer fra andre adapttere ved hjelp av broadcastkanalen.
4. Hvis adapteren sender hele rammen uten å detektere signal energi fra andre adapttere, er adapteren ferdig med rammen. Hvis adapteren derimot oppdager signal energi fra andre adapttere mens den overføres, avbryter den overføringen (det vil si at den slutter å sende sin ramme).
5. Etter avbrudd venter adapteren en tilfeldig tid og går deretter tilbake til trinn 2.



**Figure 5.13 ♦** CSMA with collision detection

For å forhindre at kolliderende rammen, kolliderer flere ganger etter hverandre brukes den *\*binary exponential backoff* algoritme, som gjør at en ramme som har opplevd  $n$  kollisjoner velger en verdi av  $K$  fra  $\{0,1,\dots, 2^n-1\}$ .

$$\text{Efficiency} = \frac{1}{1 + 5d_{\text{prop}}/d_{\text{trans}}}$$

Husk at to ønskelige egenskaper til en tilgangsprotokoll er (1) når bare en node er aktiv, har den aktive noden en gjennomstrømning på R bps, og (2) når M-noder er aktive, har hver aktiv knute en gjennomstrømning på nesten R / M bps. ALOHA- og CSMA-protokollene har denne første egenskapen, men ikke den andre.

Dette har motivert forskere til å skape en annen klasse protokoller - **taking turns protocols**. Som med random access protokoller er det dusinvis av taking-turnsprotokoller. Vi diskuterer to av de viktigste protokollene her.

- Den første er **polling protocol**. Pollingprotokollen (*nor. spørreprotokollen*) krever at en av nodene skal betegnes som en hovednode. Masternoden spør hver av knutepunktene i en rundgang, først node 1, så node 2 osv. Masternoden sender først en melding til node 1 og sier at den (node 1) kan sende opp til et maksimum antall rammer. Etter at noden 1 sender noen rammer, forteller hovednoden node 2 at den kan sende opp til maksimalt antall rammer. (Hovednoden kan bestemme når en node er ferdig med å sende sine rammer ved å observere mangelen på et signal på kanalen.) Prosedyren fortsetter på denne måten, idet hovednoden *poller* hver node på en sykisk måte.
  - Eliminerer kollisjoner og tomme slots, som har vært et problem hos random access protokollene. Gir også *høyere effektivitet*. Men den introduserer en *polling delay*.
  - Problemer: Dersom kun en node aktiv, må den spørre alle de andre nodene mellom hver gang den aktive noden har sendt maksimum antall frames. Polling delay - tiden det tar å sende en melding til en node. Dersom masternoden feiler, vil hele kanalen være ubruklig.
- Den andre taking-turns protokollen er **token-passing protocol**. I denne protokollen er det ingen master node. En mindre, spesial-ramme, kalt en **token** blir forvekslet mellom nodene i et bestemt mønster (f.eks. sykisk). Når en node mottar en token, beholder den tokenen kun dersom den har en ramme å sende, hvis ikke sender den noden umiddelbart videre til neste node. Dersom den vil sende beholder den tokenen, sender maksimalt antall med rammer, og sender tokenen videre.
  - Dersom en node kræsjer, kan hele kanalen kræsje.

### DOCSIS: The Link-Layer Protocol for Cable Internet Access

I de tre foregående delene har vi lært om tre klasser med flere accessprotokoller: *channel-partitioning* protokoller, *random access* protokoller, and *taking-turns* protokoller. Et kabelnettverk vil gjøre det til en utmerket case-studie her, da vi finner aspekter av hver av disse tre klassene med flere tilgangsprotokoller med kabelaksessnettet!

Husk fra Kap. 1 at et kabelaksessnettverk typisk kobler sammen flere tusen boligkabelmodemer til et kabelmodemtermineringsystem (CMTS) hos kabelnettverkhodeenden. **The Data-Over-Cable Service Interface Specifications (DOCSIS)** spesifiserer kabedata nettverksarkitekturen og dets protokoller. DOCSIS bruker FDM for å dele nedstrøms (CMTS til modem) og oppstrøm (modem til CMTS) nettverksegmenter inn i flere frekvenskanaler. Hver oppstrøms- og nedstrømskanal er en broadcastkanal.

- Rammer sendt fra CMTS langs nedstrømskanalen går til alle modemene, men ettersom det kun er en enkel CMTS som sender langs broadcastkanalen er det *ingen multiple access problem*.
- Rammer sendt oppstrøms fra modemene, er mer interessant da det er flere kabelmodemer (sendere) og en mottaker langs oppstrømskanalen, og kollisjoner kan potensielt oppstå.

Som illustrert i Figur 5.14, er hver oppstrømskanal delt opp i intervaller i tid (TDM-ish) bestående av en sekvens mini-luker, som kabelmodemene kan sende til CMTS-en. Hvem som skal sende bestemmer CMTS-en ved å sende en såkalt MAP-melding langs nedstrømskanalen for å spesifisere hvilket kabelmodem som skal få sende i hvilken mini-luke, for intervallet spesifisert i meldingen.

- Hvordan vet CMTS-en hvilke modemer som har data å sende i utgangspunktet. Dette er løst ved at kabelmodemene sender mini-luke-request-frames til CMTS.-en.
- Et kabelmodem kan verken merke at oppstrømskanalen er opptatt eller detektere en eventuell kollisjon. Men den merker kollisjonen dersom den ikke får en bekrefteelse for rammen i neste nedstrømskontrollmelding.
- Når en kollisjon oppstår bruker modemet binær eksponentiell backoff.

Et kabeltilkoblingsnettverk tjener som et fantastisk eksempel på flere tilgangsprogrammer i action - *FDM*, *TDM*, random access, og sentralt allokerede tidsluker innenfor et nettverk.

## Switched Local Area Networks

Etter å ha dekket broadcast-ettverk og multiaksessprotokoller, skal vi nå se på swithed-lokalnettverk. Figur 5.15 viser et switched-lokalnettverk som kobler sammen tre avdelinger, to servere og en ruter med 4 svitsjer. Ettersom disse svitsjene operererer på linklaget, svitsjer de linklagsrammer, og ser ikke på nettverksadresser (IP-adresser), og bruker heller ikke routing-algoirmer for å bestemme stien. Istedet for å bruke IP-adresser bruker de heller link-lagsadresser, som vi skal se på nå.

### Link-Layer Addressing and ARP

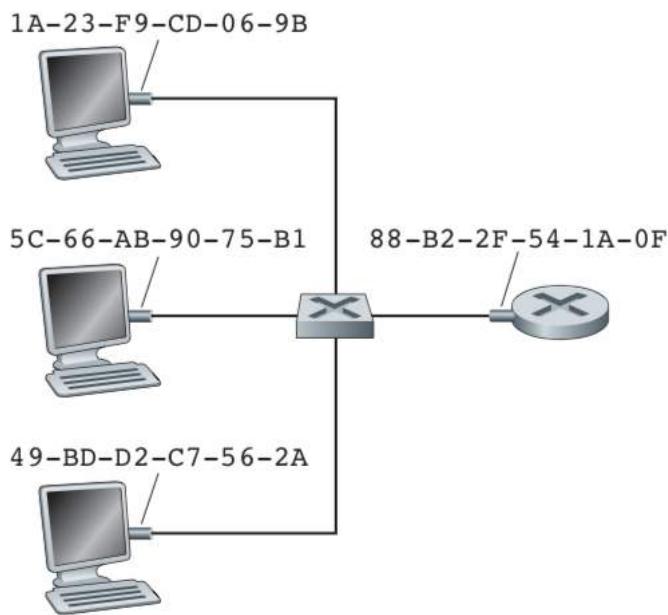
Verter og rutere har linklagsadresser. Men hvorfor trenger verter og rutere både linklags- og nettverksadresser? Vi skal se syntaksen og funksjonen til linklagsadresser, og se hvorfor begge er viktige. Vi skal også se på *Address Resolution Protocol (ARP)*, som kan oversette IP-adresser til linklagsadresser.

#### MAC Adresser

Det er faktisk ikke selve ruterne og vertene som har linklagsadresser, men deres adapttere, dvs. nettverksinterfacene, som har linklagsadresser.

En vert eller ruter i et multiple network-interfaces vil dermed ha flere linklagsadresser assosiert med den. Samme som at den har flere IP-adresser. Det er viktig å notere at linklagssvitsjer ikke har linklagsadresser assosiert med interfacene sine. En linklagsadresse er variert kalt en **LAN adresse**, en **fysisk adresse**, eller en **MAC adresse**. Siden MAC adresse er det som er mest brukt, er det det jeg bruker fremover.

For de fleste LANer er MAC adressen 6 bytes lang, som gir  $2^{48}$  mulige MAC-adresser. De er representert som hexadecimalnotasjon, der hver byte er skilt.



**Figure 5.16** ♦ Each interface connected to a LAN has a unique MAC address

Selvom det var meningen av MAC-adressen skulle være permanente er det nå mulig å endre en adapters MAC-adresse gjennom programvare. Ingen adapttere har samme adresse. Men hvordan passer to selskaper i forskjellige land på å asse på å bruke andre MAC-adresser enn andre land? Svaret er at det er IEEE som vedlikeholder MAC-adresserommet. Et selskap som skal lage adapttere, kjøper en klump med adresserom bestående av  $2^{24}$  adresser for en nominell sum. IEEE allokerer et adresserom på e av  $2^{24}$  adresser ved å bestemme de første 24 bitene av MAC-adressene, og lar selskapet bestemme de unike kombinasjonene av de siste 24 bitene for hvert adapter.

En adapters MAC-adresse har en flat struktur, og endrer seg ikke ut i fra hvor adapteren går. En PC med Ethernet-interface har samme MAC-adresse utansett hvor PC-en går. En mobil med et 802.11 interface har samme MAC-adresse, uansett hvor mobilen er, kan sammenliknes med personnummer. Dette er en stor forskjell fra IP-adresser. Men på samme måte som en person kan finne det nyttig å både ha et personnummer og en postadresse, kan det være nyttig for en vert- eller ruter-interface å ha både MAC adresse og IP-adresse.

Når et adapter ønsker å sende en ramme til en destinasjonsadapter, vil det sendende adapteret legge inn destinasjonsadapterets MAC-adresse inn i rammen og deretter sende rammen inni LAN-et.

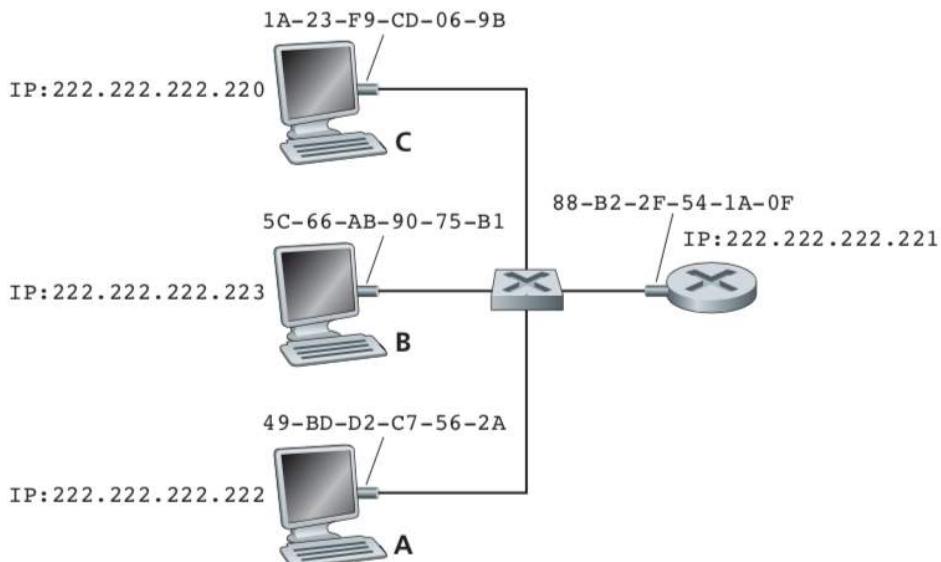
Når et adapter mottar en ramme, vil den skjekke om destinasjons-MAC-adressen er den samme som adapterets.

Noen ganger vil et adapter at alle andre adaptere skal få pakken, da kan man bruke MAC-adressen FF-FF-FF-FF-FF-FF. Som er en spesiell **MAC broadcast adresse**.

### Address Resolution Protocol (ARP)

Ettersom det er både nettverklagsadresser (f.eks. IP-adreser) og linklagsadresser (MAC-adresser), trengs det å kunne oversette mellom dem. For internettet er dette jobben til **Address Resolution Protocol (ARP)**.

For å forstå behovet for en protokoll som ARP, bør du vurdere nettverket som vises i figur 5.17. I dette enkle eksempelet har hver vert og ruten en enkelt IP-adresse og en enkelt MAC-adresse. Som vanlig vises IP-adresser med stiplede desimaler og MAC-adresser vises i heksadesimal notasjon. I forbindelse med denne diskusjonen antar vi i denne delen at bryteren sender alle rammer; det vil si når en bryter mottar en ramme på ett grensesnitt, videresender rammen på alle sine andre grensesnitt. I neste avsnitt vil vi gi en mer nøyaktig utredning av hvordan brytere opererer. Anta nå at verden med IP-adresse 222.222.222.220 vil sende et IP datagram til vert 222.222.222.222. I dette eksemplet er både kilden og destinasjonen i samme delnett. For å sende et datagram må kilden gi sin adapter ikke bare IP-datagrammet, men også MAC-adressen for destinasjonen 222.222.222.222. Sendingsadapteren vil da konstruere en linklagsramme som inneholder destinasjonens MAC-adresse og sende rammen til LAN.



**Figure 5.17** ♦ Each interface on a LAN has an IP address and a MAC address

Det viktige spørsmålet som er adressert i denne delen er, hvordan bestemmer den sendende verden MAC-adressen til mottakerverten med IP-adresse 222.222.222.222? Som du kanskje har gjettet, bruker den ARP. En ARP-modul i sendingsverten tar en hvilken som helst IP-adresse på samme LAN som inngang, og returnerer den tilhørende MAC-adressen.

- I eksempelet over senderverten 222.222.222.220 sin ARP-modulen IP-adressen 222.222.222.222, og ARP-modulen returnerer den tilsvarende MAC-adressen 49-BD-D2-C7-56-2A.

Så vi ser at ARP oversetter en IP-adresse til en MAC-adresse. På mange måter er det analogt med *DNS*, som oversetter vertsnavn til IP-adresser. En **viktig forskjell** mellom de to beslutningstakere er imidlertid at DNS oversetter vertsnavn for verter hvor som helst på Internett, mens ARP løser IP-adresser bare for verter og rutergrensesnitt på samme delnett. Hvis en node i California skulle forsøke å bruke ARP for å løse IP-adressen til en node i Mississippi, ville ARP returnere med en feil.

Nå som vi har forklart hva ARP gjør, la oss se på hvordan det fungerer. Hver vert og ruten har en ARP-tabell i sitt minne, som inneholder mappings av IP-adresser til MAC-adresser. Figur 5.18 viser hva et **ARP-table** i verden 222.222.222.220 kan se ut. ARP-tabellen inneholder også en time-to-live-verdi (TTL), som indikerer når hver kartlegging vil bli slettet fra tabellen.

En node har nødvendigvis ikke mappingen for alle andre noder i subnettet.

En typisk utløpstid for en oppføring er 20 minutter fra når en oppføring er plassert i et ARP-bord.

| IP Address      | MAC Address       | TTL      |
|-----------------|-------------------|----------|
| 222.222.222.221 | 88-B2-2F-54-1A-0F | 13:45:00 |
| 222.222.222.223 | 5C-66-AB-90-75-B1 | 13:52:00 |

**Figure 5.18** ♦ A possible ARP table in 222.222.222.220

Men dersom en node ønsker å sende en pakke til en IP-adresse, og må finne MAC-adressen til denne IP-adressen. Svaret er enkelt om noden har destinasjonsnoden i ARP-tabellen. Men dersom den ikke er der, da må senderen bruke ARP-protokollen for å finne denne adressen. Da lager senderen en spesiell \*ARP pakke. En ARP-pakke har flere felt, inkludert senderens og mottakerens IP- og MAC-adresser. Både query- og response-meldinger har samme format.

Sendernoden vil da sende en ARP-query pakke til MAC broadcast adressen, nemlig FF-FF-FF-FF-FF-FF. Adapteret innkapsler pakken i en linklagsramme og sender den inn i subnettet. Rammen som inneholder ARP-melingen blir mottatt hos alle de andre adapterene i subnettet, som sender det til ARP-modulene sine. Alle sjekker om IP-adressene i ARP-tabellen sin matchet med destinasjons IP-adressen i ARP-pakken. Den som har en match, sender tilbake en ARP-melding med den ønskede mappingen til den forspørrende noden. Den mottakende noden oppdaterer ARP-tabellen sin, og sender til adressen som sto i responsmeldingen.

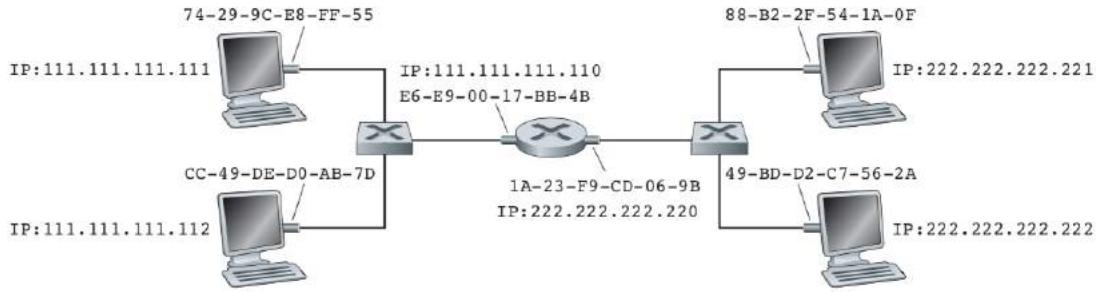
- Er plug-and-play, trenger ingen konfigurering av en systemadministrator
- Dersom en enhet forsvinner fra subnettet, slettes den etterhvert fra ARP-tabellene i subnettet.

Kan være diskusjon om hva slags protokoll ARP er, er linklayer-protokoll. Men kan sees på som en protokoll som både tilhører linklayer og transportlaget.

#### Sende et datagram utenfor subnettet

Det burde nå være klart hvordan ARP opererer når en vert vil sende et datagram til en vert *på samme subnet*. Men la oss se på en mer komplisert situasjon, når en vert i et subnet ønsker å sende et nettverksdatagram til en vert *utenfor subnettet*.

Det flere ting å notere seg med figuren under. Hver vert har nøyaktiv en IP-adresse og ett adapter. Men som vi vet fra Kap 4, har en rute en IP-adresse for *hvert* av sine interfaceter. For hvert ruterinterface er det også en ARP-modul (i ruta) og et adapter. I figuren har ruta 2 interfaceter, 2 adaptorer, 2 ARP-moduler og henholdsvis 2 MAC-adresser. Noter også at Subnett 1 har nettverksadressen 111.111.111./24 og at Subnett 2 har nettverksadressen 222.222.222/24, og dermed har alle interfacene i subnett 1 adresser på formen 111.111.111.xxx, og i subnett 2 har interfacene 222.222.222.xxx.



**Figure 5.19** • Two subnets interconnected by a router

Dersom en vert i Subnet 1 ønsker å sende et datagram til en vert i Subnet 2, må den sendende verten sende datagrammet til adapteret sitt, men den sendende verten må også indikere MAC-adressen til den mottakende adapteren. Som vi ser i figuren må datagrammet først til ruteren for å komme seg til en mottakervert i subnet 2. Men hvordan får senderverten MAC-adressen til ruteren, med ARP selvfølgelig (Samme som over). Når senderverten sender rammen, vil ruteren ta den imot, se at den er til seg og sende den til nettverkslaget til ruteren. Da var pakken fremme hos ruteren.

Vi må fortsatt flytte datagrammet fra ruteren til destinasjonsverten. Ruteren må nå bestemme riktig interface som datagrammet skal videresendes til. Som diskutert i kapittel 4, gjøres dette ved å konsultere et forwardingtable i ruteren. Forwardingtabellen forteller ruteren at datagrammet skal videresendes via ruteren grensesnitt 222.222.222.220. Dette grensesnittet passerer da datagrammet til adapteren, som innkapsler datagrammet i en ny ramme og sender rammen til delnett 2. Denne gangen er måladressens MAC-adresse i virkeligheten MAC-adressen til det endelige målet. Og hvordan får ruteren denne destinasjons MAC-adressen? Fra ARP, selvfølgelig!

## Ethernet

Ethernet har i hovedsak tatt over det kablede LAN-markedet. Ble oppfunnet på 1970-tallet, og har fortsatt å utvikle seg og vokse og har fortsatt sin dominerende posisjon.

I dag er Ethernet langt den mest utbredte kablede LAN-teknologien, og det er sannsynlig å forbli det i overskuelig fremtid. Man kan si at Ethernet har vært i lokalnettverk som Internett har vært på globalt nettverk.

Det er mange grunner til Ethernet suksess. For det første var Ethernet det første distribuerte høyhastighetsnettverket.

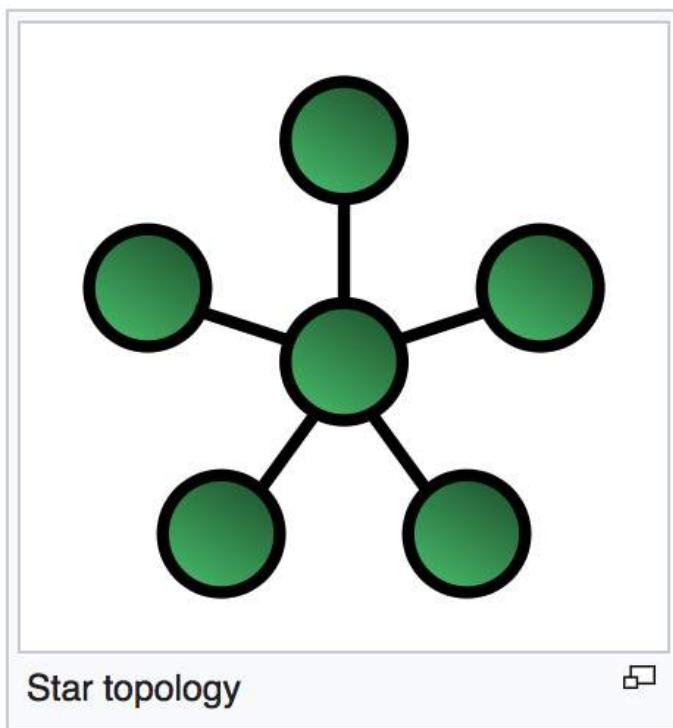
Fordi den ble distribuert tidlig, ble nettverksadministratorer fortrolige med Ethernet - dets underverk og dens kjennskap - og var motvillige til å bytte over til andre LAN-teknologier når de kom på scenen. For det andre var token-ring, FDDI og ATM mer komplekse og dyrere enn Ethernet, noe som ytterligere frarådde nettverksadministratorer fra å bytte over.

- Det originale Ethernet LAN brukte en coax-bus som sammenkoblet nodene. Ethernet med bustop
  - Ethernet med en busstopologi er et *broadcast LAN* - alle overførte rammer reiser til og behandles av alle adaptere som er koblet til bussen.
  - Husk at vi dekket Ethernets CSMA / CD-tilgangsprotokoll med binær eksponensiell backoff i tidlige avsnitt.

Ved slutten av 1990-tallet hadde de fleste bedrifter og universiteter erstattet sine LAN med Ethernet-installasjoner ved hjelp av en *hubbasert stjernetopologi*. I en slik installasjon er vertene (og ruterne) direkte forbundet med en *hub* med *twisted pair copper wire*. En **hub** er en fysisklags-enhet som virker på individuelle bits i stedet for frames. Når en bit, som representerer en null eller en, kommer fra ett interface, gjenoppretter huben bit-en, øker energistyrken dens og overfører biten til alle de andre interfacene.

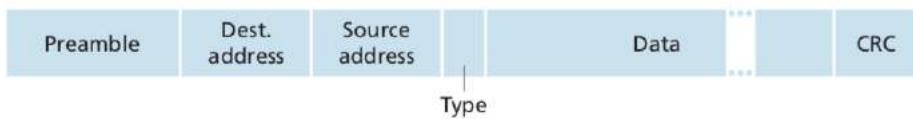
- Derfor er Ethernet med en hubbasert stjernetopologi også et broadcast LAN. Når et hub mottar en bit fra en av dens interfacene, sender den en kopi ut på alle sine andre interfacene.

Dersom en hub mottar rammer fra to forskjellige interfacene samtidig, oppstår en kollisjon, og nodene som skapte rammene må sende dem på nytt.



I begynnelsen av 2000-tallet opplevde Ethernet enda en stor evolusjonær endring. Ethernet installasjoner fortsatte å bruke en stjerne topologi, men huben i sentrum ble erstattet med en **switch**. Vi vil undersøke switchet Ethernet i dybden senere i dette kapittelet. For nå nevner vi bare at en svitsj ikke bare er "kollisjonsløs", men er også en ekte store-and-forward pakkesvitsj; men i motsetning til rutere, som opererer opp gjennom lag 3, virker en svitsj kun opp gjennom lag 2.

#### Ethernet Frame Structure



**Figure 5.20** ◆ Ethernet frame structure

Vi kan lære mye av Ethernet av å se på Ethernet-framen i figuren over. Vi noterer at selvom vår Ethernet-ramme bærer et IP-datagram, kan det også bære andre nettverkspakker i payloaden.

La det sendende adatperet, adapter A, ha MAC-adressen AA-AA-AA-AA-AA-AA og det mottakende adapteret, adapter B, ha MAC-adressen BB-BB-BB-BB-BB-BB. Det sendende adapteret innkapsler IP-datagrammet i en Ethernet-ramme, og sender rammen til det fysiske laget. Det mottakende adapteret tar imot rammen fra det fysiske laget, ekstraherer IP-datagrammet og sender datagrammet til nettverkslaget. I denne konteksten la oss se på de 6 feltene i Ethernet-rammen i Figur 5.20:

- *Data field (46 til 1,500 bytes)*. Dette feltet har IP-datagrammet. Maximum transmission unit (MTU) til Ethernet er 1,500 bytes. Dette betyr at dersom et IP-datagram er større en 1,500 bytes, må verter fragmentere datagrammet som vi snakket om i Kap 4. Minimumstørrelsen til datafeltet er 46 byte. Dette betyr at dersom et datagram er mindre enn dette må man "fylle" resten av plassene slik at det blir 46 bytes. Bruker length-fieldet i IP-datagramme til å fjerne *fyll*.
- *Destination address (6 bytes)*. Dette feltet inneholder MAC-adressen til destinasjonsadapteret. I eksempelet over blir det BB-BB-BB-BB-BB-BB.
- *Source address (6 bytes)*. Dette feltet inneholder MAC-adressen til det sendende adapteret, som i eksempelet blir AA-AA-AA-AA-AA-AA.
- *Type field (2 bytes)*. Dette type-feltet tillater Ethernet å multiplekse nettverkspakker. For å forstå dette, må vi huske at verter kan bruke andre nettverkspakker i tillegg til IP. Faktisk kan en gitt vert støtte flere nettverkspakker ved hjelp av forskjellige protokoller for forskjellige applikasjoner. Av denne grunn, når Ethernet-rammen kommer til adapter B, må adapter B vite hvilken nettverkspakk den skal passere (det vil si demultiplekse) innholdet i datafeltet.

- IP og andre protokoller for nettverkslaget har hver sitt eget standardiserte typenummer. I tillegg har ARP-protokollen (diskutert i forrige avsnitt) sitt eget typenummer, og hvis den ankommerrammen inneholder en ARP-pakke (dvs. har et typefelt på 0806 heksadesimale), vil ARP-pakken bli demultiplexert opp til ARP-prokot.
- *Cyclic redundancy check (CRC) (4 bytes)*. Som diskutert tidligere, er meningen til CRC-feltet å tillate det mottakende adapteret, adapter B, å detektere feil i rammen.
- *Preamble (8 bytes)*. Ethernet-rammen begynner med et 8-byte preamble-felt (*nor. innledning el. forord*). Hvert av de 7 første bytene har en verdi på 10101010; den siste byten har verdien 10101011. De første 7 bytene av preamble-en skal "vekke" mottaker-adapteret, og synkronisere klokkena dens til avsenderens klokke. Hvorfor skal klokkena være synkroniserende? Husk at adapter A har som mål å overføre rammen til 10 Mbps, 100 Mbps eller 1 Gbps, avhengig av typen Ethernet LAN. De to siste bitene (to etterfølgende 1-ere) varsler mottakeren at de "viktige greiene" kommer nå.

Alle Ethernet-teknologiene gir **connectionless** tjeneste til nettverkslaget. Det vil si når adapter A ønsker å sende datagram til adapter B, innkapsler adapter A datagrammet i en Ethernet-ramme og sender rammen inn i LAN-en uten *handshaking* med adapter B. Denne tilkoblingsløse tjenesten på lag 2 er analog (ligner på visse aspekter) med IPs layer 3 datagramtjeneste og UDPs layer 4 connectionless tjeneste.

Ethernet-teknologier gir en **upålitelig** service til nettverkslaget. Spesielt når adapter B mottar en ramme fra adapter A, kjører den rammen gjennom en CRC-sjekk, men *verken* sender en bekrefteelse når en ramme passerer CRC-sjekken eller sender en negativ bekrefteelse når en ramme mislykkes CRC-sjekken. Når en ramme mislykkes i CRC-kontrollen, kasserer adapter B bare rammen. Adapter A har således ingen anelse om dens overførte ramme nådde adapter B og passerte CRC-sjekken.

- Denne mangelen på pålitelig transport (ved linklaget) bidrar til å gjøre Ethernet enkelt og billig. Men det betyr også at strømmen av datagrammer som sendes til nettverkslaget, kan ha hull.
- Hvis det er hull på grunn av forkastede Ethernet-rammer, ser applikasjonen ved vert B også hullene? Som vi lærte i kapittel 3, avhenger dette av om applikasjonen bruker UDP eller TCP.
  - Hvis programmet bruker **UDP**, vil programmet i verten B faktisk se hullene i dataene.
  - På den andre siden, hvis applikasjonen bruker **TCP**, vil ikke TCP i vert B bekrefte dataene i forkastede rammer, noe som forårsaker at TCP i vert A skal sendes på nytt. Merk at når TCP sender data tilbake, vil dataene til slutt gå tilbake til Ethernet-adapteren der den ble kassert. På denne måten sender Ethernet således data videre, selv om Ethernet ikke vet om det overfører et helt nytt datagram med helt nye data, eller et datagram som inneholder data som allerede er overført minst en gang.

### **Ethernet Technologies**

I vår diskusjon ovenfor har jeg henvist til Ethernet som om det var en enkelt protokollstandard. Men faktisk kommer Ethernet i mange forskjellige typer, med noe forvirrende akronymer som 10BASE-T, 10BASE-2, 100BASE-T, 1000BASE-LX, og 10GBASE-T.

Disse og mange andre Ethernet teknologier har blitt standardisert over årene av IEEE 802.3 CSMA/CD (Ethernet) arbeidsgruppen. Selvom disse akronymene viser veldig kompliserte, er det en orden. Akronymene refererer til hastigheten til standarden: 10, 100, 1000, eller 10G, for 10 Megabit(/sec), 100 Megabit, Gigabit eller 10 Gigabit Ethernet.

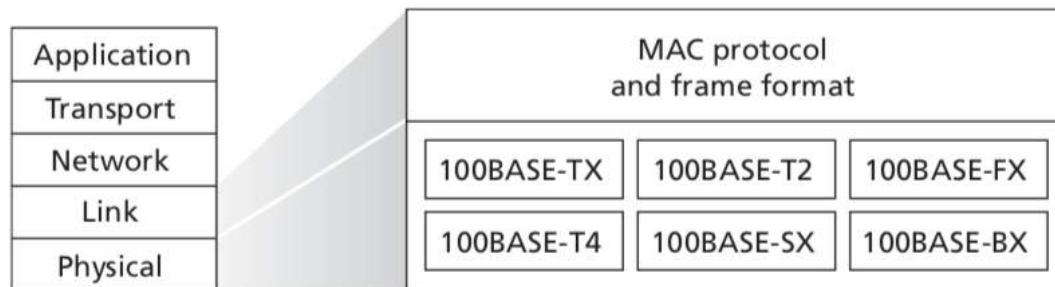
"BASE" refererer til baseband Ethernet, noe som betyr at det *fysiske mediet* bare bærer Ethernet-trafikk; nesten alle 802.3-standardene er for baseband Ethernet. Den siste delen av akronymet refererer til selve det fysiske mediet; Ethernet er både en linklag- og en fysisklagspesifikasjon, og bæres over en rekke fysiske medier, inkludert koaksialkabel, kobbertråd og fiber. Vanligvis refererer en "T" til tvunnde kobberledninger.

- Historisk sett ble et Ethernet oppfattet som et segment av koaksialkabel. De tidlige 10BASE-2 og 10BASE-5 standardene angir 10 Mbps Ethernet over to typer koaksialkabel, hver begrenset i lengde til 500 meter.
  - Lengre avstander kan oppnås ved å bruke en **repeater** - en fysisk lag-enhet som mottar et signal på inputsiden, og regenererer signalet på outputsiden.
  - En koaksialkabel samsvarer pent med vårt syn på Ethernet som et broadcasting-medium. Alle rammer som overføres av ett grensesnitt, mottas ved andre grensesnitt, og Ethernets CDMA/CD-protokoll løser pent multiaksessproblemet.

Ethernet i dag er veldig annerledes den originale bus-topologi-designet med koaksialkabel.

I de fleste installasjoner i dag, er noder tilkoblet til en svitsj gjennom et punkt-til-punkt-segment av tvunnet kobberkabel eller fiber-optiske kabler.

I midten av 1990-tallet ble Ethernet standardisert til 100 Mbps, 10 ganger raskere enn 10 Mbps Ethernet. Den opprinnelige Ethernet MAC-protokollen og rammeformatet ble beholdt, men høyrehastighets fysiske lag ble definert for kobbertråd (100BASE-T) og fiber (100BASE-FX, 100BASE-SX, 100BASE-BX). Figur 5.21 viser disse forskjellige standardene og den vanlige Ethernet MAC-protokollen og rammestilen.



**Figure 5.21** ♦ 100 Mbps Ethernet standards: a common link layer, different physical layers

100 Mbps Ethernet er begrenset til en 100 meter avstand over tvunnet kobberkabel og til flere kilometer over fiber, noe som gjør at Ethernet-brytere i forskjellige bygninger kan kobles sammen.

Gigabit Ethernet er en utvidelse av de svært vellykkede 10 Mbps og 100 Mbps Ethernet-standarder. Gigabit Ethernet har en rå datahastighet på 1000 Mbps, og har full kompatibilitet med den store installerte basen av Ethernet-utstyr. Standarden for Gigabit Ethernet, referert til som IEEE 802.3z, gjør følgende:

- Bruker standart Ethernet ramme-format (Figur 5.20), og er baklengskompatibel med 10BASE-T og 100BASE-T teknologi. Dette tillater enkel intergrering av Gigabit Ethernet med eksisterende installerte baser av Ethernetutstyr.
- Tillater punkt-til-punkt-koblinger i tillegg til delte kringkastingskanaler. Punkt-til-punkt koblinger brukes svitser, mens kringkastingskanaler bruker huber, som nevt tidligere. I Gigabit Ethernet jargon er hubs kalt *buffered distributors*.
- Bruker CSMA/CD for delte kringkastingskanaler - maksimumavstanden mellom noder er strengt begrenset pga maksimal effektivitet.
- Tillater full-duplex / fullstendig toveis-operering på 1,000 Mbps i begge retninger for punkt-til-punkt-kanaler.

Kunne i utgangspunktet kun kjøre over optisk fiber, men idag kan den også kjøre over kategori 5 UTP-kabling.

Oppsummering av Ethernet:

- I starten med bustopologier og hub-basert stjerne topologier, var Ethernet klart en kringkastingskanal, der rammekollisjoner fant sted.
- For å håndtere dette inkluderte Ethernet standarden CSMA/CD-protokollen
- Det er kommet svitsj-basert Ethernet som bruker store-and-forward pakkesvitsjing.
- Moderne switcher er full-duplex, slik at en svitsj og en node kan sende rammer til hverandre samtidig uten innblanding. Med andre ord er svitsj-basert Ethernet LAN uten kollisjoner, og trenger faktisk eg ikke en MAC-protokoll.

### Link-Layer Switches

Rollen til en svitsj er å motta inkommende koblinglagsrammer, og videresende dem inn i utgående koblinger. Vi skal se på denne vi videresendingsfunksjonen, og vi kommer til å se at svitsjen i seg selv er **gjenomsiktig** for vertene og ruterne, det vil si at en vert/ruter adresserer rammer til en annen vert/ruter, uten å vite om en svitsj vil motta rammen og videresende den.

Raten til rammer som ankommer til en av bryterens utgangsinterfacer, kan overskride linkkapasiteten til dette grensesnittet. For å imøtekommе dette problemet, må svitsj utgangsinterfacer ha buffere, på omtrent samme måte som ruterens utgangsinterface har buffere for datagrammer. La oss nå se nærmere på hvordan svitsjer fungerer.

## Forwarding and Filtering

**Filtering** er svitsjfunksjonen som bestemmer om en ramme skal videresendt til et interface, eller bli forkastet. **Forwarding** er svitsjfunksjonen som bestemmer hvilke interfacene som en ramme skal bli sendt i, og videresende rammen til de interfacene.

Svitsjfiltrering og forwarding er gjort med et **switch table**. Denne svitsjtabellen inneholder rader for noen av, men nødvendigvis ikke alle, vertene og ruterne i et LAN. En rad i svitsjtabellen består av (1) en MAC-adresse, (2) switchinterfacet som leder til MAC-adressen, og (3) tiden når raden ble innsatt i tabellen. Figuren under viser en svitsjtabell:

| Address           | Interface | Time |
|-------------------|-----------|------|
| 62-FE-F7-11-89-A3 | 1         | 9:32 |
| 7C-BA-B2-B4-91-10 | 3         | 9:36 |
| ....              | ....      | .... |

**Figure 5.22** ♦ Portion of a switch table for the uppermost switch in Figure 5.15

Selv om denne beskrivelsen av rammeoverføring kan høres ut som vår diskusjon av datagram-videresending i Kap. 4, ser vi snart at det er viktige forskjeller. En viktig **forskjell** er at svitsjer videresender pakker basert på *MAC-adresser i stedet for på IP-adresser*. Vi vil også se at en svitsjtabell er konstruert på en helt annen måte enn en routers forwardingtabell.

For å forstå hvordan byte filtrering og videresending fungerer, anta en ramme med destinasjonsadresse *DD:DD:DD:DD:DD:DD* kommer til bryteren på grensesnitt *x*. Bryteren indekserer sitt bord med MAC-adressen *DD:DD:DD:DD:DD:DD*. Det er tre mulige tilfeller:

- Det er ingen oppføring i tabellen for *DD:DD:DD:DD:DD:DD*. I dette tilfellet videresender svitsjen kopier av rammen til utgangsbufferne foran alle grensesnitt bortsett fra grensesnitt *x*. Med andre ord, hvis det ikke er noen oppføring for destinasjonsadressen, kringkaster svitsjen rammen.
- Det er en oppføring i tabellen som forbinder *DD:DD:DD:DD:DD:DD* med interfacet *x*. I dette tilfellet kommer rammen fra et LAN-segment som inneholder adapter *DD:DD:DD:DD:DD:DD*. Dermed er det ikke nødvendig å videresende rammen til noen av de andre grensesnittene, utfører svitsjen filtreringsfunksjonen ved å forkaste rammen.
- Det er en oppføring i tabellen som forbinder *DD:DD:DD:DD:DD:DD* med interfaces *y ≠ x*. I dette tilfellet må rammen sendes til LAN-segmentet som er festet til grensesnitt *y*. Bryteren utfører sin videresendingsfunksjon ved å sette rammen i en utgangsbuffer som går foran grensesnittet *y*.

## Self-Learning

En svitsj har den fantastiske egenskapen av at tabellen dens bygges automatisk, dynamisk og autonomisk - uten noen innvending. Med andre ord er svitsjer **self-learning**. Denne egenskapen er oppnådd følgende:

1. Svitsjtabellen er i utgangspunktet tom.
2. For hver inkommende ramme på et interface, lagrer svitsjen (1) MAC-adressen til rammens *source address field*, (2) interfacet som rammen ble sendt fra, og (3) nåværende tidspunkt, i svitsjtabellen. Dersom enhver vert i LAN-et etterhvert skulle ha en ramme, ville alle vertene ha havnet i tabellen.
3. Svitsjen sletter en adresse i tabellen dersom ingen rammer er mottatt fra den adressen som kilde-adresse, etter en viss periode (**aging time**). På denne måten, dersom en PC blir ertattet av den andre PC-en (med et annen adapter), vil MAC-adressen fra den første PC-en etterhvert forsvinne fra tabellen

Svitsjer er **plug-and-play devices**, ettersom det ikke kreves noen inngrep fra en nettverksadministrator eller bruker. Svitsjer er også **full-duplex**, som gjør at den kan motta og sende samtidigt.

## Properties of Link-Layer Switching

Etter å ha beskrevet den grunnleggende operasjonen av en linklagsbryter, la oss nå vurdere funksjonene og egenskapene deres. Vi kan identifisere flere fordeler ved å bruke brytere, istedet for broadcastkoblinger som busser eller hubbaserte stjernetopologier:

- *Elimination of collisions.* I en LAN bygget av svitsjer (uten hubs), er det ingen båndbredde mistet på grunn av kollisjoner. Som med en ruter, er den maksimume gjennomstrømningen til en svitsj summen av alle interface-ratene. Dermed gir brytere en betydelig ytelsesforbedring over LAN med kringkastingskoblinger.
- *Heterogenous links.* Ettersom en svitsj isolerer en link fra en annen, kan de forskjellige koblingene i LAN-et operere på forskjellige hastigheter og forskjellige mediumer.
- *Management*, I tillegg for å tilby ekstra sikkerhet, vil en svitsj også lette på nettverksadministrasjon. Hvis en adapter for eksempel feiler og kontinuerlig sender Ethernet-rammer (kalt en *jabbering-adapter*), kan en svitsj oppdage problemet og internt koble fra den malfunksjonelle adapteren. Svitsjer samler også statistikk over båndbredd bruk, kollisjonssatser og trafikktyper, og gjør denne informasjonen tilgjengelig for nettverksadministratoren.

#### Sniffing a Switched LAN: Switch Poisoning

Antar i eksempelet et *switched LAN*. Dersom en vert A ønsker å sende en pakke til vert B, gjennom en svitsj, og B er i svitsj-tabellen vil pakken sendes direkte, og *kun*, til vert B. Dersom en vert C, kjører en sniffer, vil ikke C være i stand til å sniffe A-to-B-rammen.

Mens dersom vert B ikke er i svitsjtabellen, vil svitsjen kringkaste rammen, og C vil kunne sniffe pakken.

Et vellkjent angrep mot en svitsj, er kalt **switch poisoning**, der man sender et tonn med pakker til en svitsj, med mange forskjellige tulle kilde-MAC-adresser, og dermed fylle svitsjtabellen, og dermed ikke lar det være noen plass til andre MAC-adresser med ekte verter.

Dette forårsaker at svitsjen vil kringkaste de fleste rammer, slik at de kan bli plukket opp av en sniffer.

#### Switches Versus Routers

Som vi har lært i Kap. 4, er rutere store-and-forward pakkesvitsjer som videresender pakker som bruker nettverkslagsadresser (f.eks. IP-adresser). Selv om en svitsj også er en store-and-forward pakkesvitsj, er den fundamentalt forskjellig fra en ruter ved at den videresender pakker ved hjelp av MAC-adresser. Mens ruteren er en lag-3-pakkesvitsj, er svitjen en lag-2 pakkesvitsj.

Selv om brytere og rutere er fundamentalt forskjellige, må nettverksadministratorer ofte velge mellom dem når du installerer en samtrafikk. Gitt at både svitser og rutere er kandidater for sammenkoblingsenheter, hva er fordelene og ulemperne ved de to tilnærmingene?

#### Pros and Cons - Svitser:

Først vurdere fordeler og ulemper ved svitser. Som nevnt ovenfor er svitser plug-and-play, en egenskap er elsket av alle overarbeidede nettverksadministratorer i verden. Svitser kan også ha relativt høye filtrerings- og videresendingshastigheter - som vist i figur 5.24, må brytere bare behandle rammer opp gjennom lag 2, mens rutere må behandle datagrammer opp gjennom lag 3.

På den andre siden, for å hindre sykler av kringkastingsrammer, er den aktive topologien til et switchet nettverk begrenset til et spennetre. Et stort switchet nettverk ville også kreve store ARP-tabeller i verter og rutere og ville generere betydelig ARP-trafikk og behandling. Videre er brytere utsatt for kringkastingsstormer. Hvis en vert går ut av kontroll og overfører en endeløs strøm av Ethernet-sendingsrammer, vil bryterne videresende alle disse rammene, slik at hele nettverket faller sammen.

#### Pros and Cons - Rutere:

Nå vurdere fordeler og ulemper med rutere. Fordi nettverksadressering ofte er hierarkisk (og ikke flatt, som MAC-adressering), kjører ikke pakker normalt ikke via rutere, selv når nettverket har overflødige stier. (Imidlertid kan pakker sykle når rutetabeller er feilkonfigurerte, men som vi lærte i kapittel 4, bruker IP et spesielt datagramheaderfelt for å begrense syklingen (TTL)).

Pakker er således ikke begrenset til et spennetre og kan bruke den beste veien mellom kilde og destinasjon. Fordi rutere ikke har spennbegrensningen, har de fått internett til å bli bygget med en rik topologi som blant annet inneholder flere aktive koblinger mellom Europa og Nord-Amerika.

En annen egenskap av rutere er at de gir brannmurbeskyttelse mot lag-2 kringkastede stormer. Kanskje den *viktigste ulempen* med rutere er at de *ikke* er *plug-and-play* og vertene som kobler til dem, trenger at deres IP-adresser skal konfigureres. Også rutere har ofte en *større prosesseringstid* per pakke enn svitser, fordi de må behandle seg gjennom lag 3-feltene.

### Så, når skal man bruke hva?

Gitt at både svitser og rutere har sine fordeler og ulemper (som oppsummert i tabell 5.1 under), når skal et institusjonelt nettverk (for eksempel et universitetskampusnett eller et bedriftsnettsted) bruke brytere, og når skal det bruke rutere?

Vanligvis små nettverk bestående av noen få hundre verter, har noen få LAN segmenter. Brytere er nok for disse små nettverkene, da de lokaliserer trafikk og øker gjennomstrømning uten å kreve noen konfigurasjon av IP-adresser.

Men større nettverk bestående av tusenvis av verter inkluderer vanligvis rutere i nettverket (i tillegg til brytere). Ruterne gir en mer robust isolasjon av trafikk, kontroll kringkastingsstormer, og bruker mer "intelligente" ruter blant vertene i nettverket.

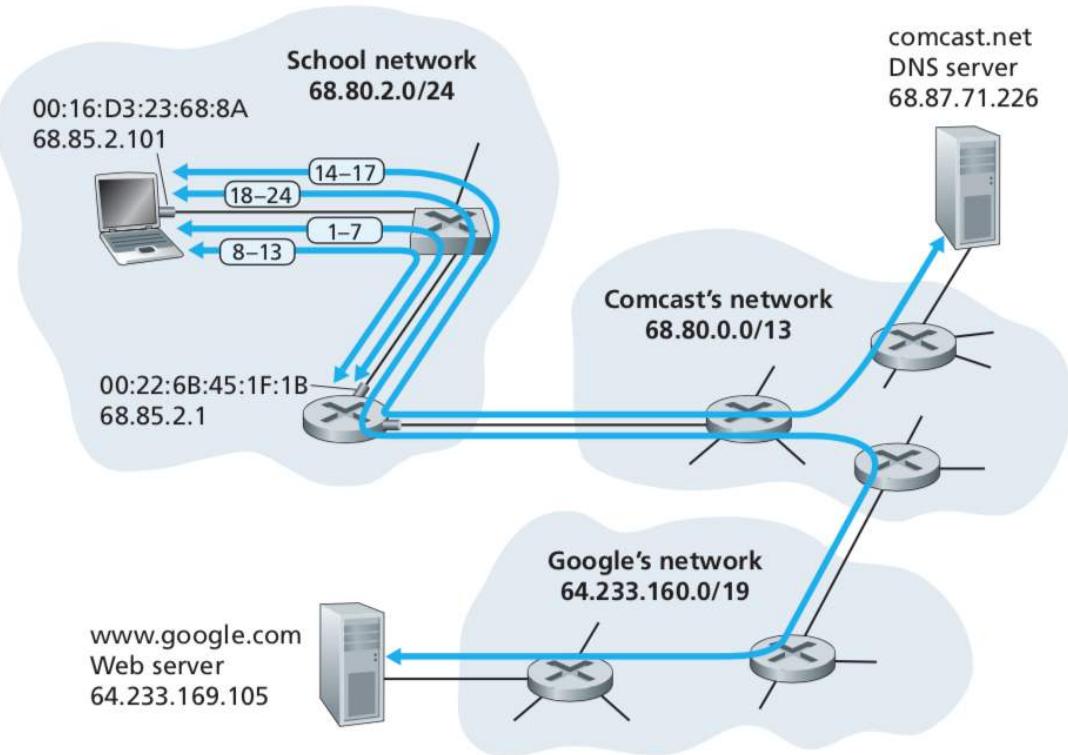
|                   | Hubs | Routers | Switches |
|-------------------|------|---------|----------|
| Traffic isolation | No   | Yes     | Yes      |
| Plug and play     | Yes  | No      | Yes      |
| Optimal routing   | No   | Yes     | No       |

**Table 5.1** ♦ Comparison of the typical features of popular interconnection devices

### Retrospective: A Day in the Life of a Web Page Request

Nå som vi har sett på koblingslaget i dette kapitellet, og nettverk, transport, og applikasjonslaget, er veien nedover protokollstakken komplett.

Vi skal nå se på det store bildet, og dette gjør vi ved å se på den enkelste forespørselen: nedlasting av en webside.



**Figure 5.32** ♦ A day in the life of a Web page request: network setting and actions

#### Getting Started: DHCP, UDP, IP and Ethernet

La oss anta at Bob starter opp datamaskinen sin, kobler til en Ethernet kalbel tilkoblet skolens Ethernet svitsj, som igjen er koblet til skolens ruter, som vist i Figur 5.32. Skolens ruter er koblet til en ISP, som i dette tilfellet er comcast.net. I dette tilfelle tilbyr comcast.net DNS-tjeneste for skolen, og DNS-serveren ligger på Comcast nettverket istedet for skolens nettverk. Vi antar at det kjører en DHCP-server på ruteren.

Når Bob først kobler PCen sin til nettverket, kan han ikke gjøre noe uten en IP-adresse. Den første nettverksrelaterte oppgaven tatt av Bob sin datamaskin er å kjøre DHCP protokollen for å få en IP-adresse, fra den lokale DHCP-serveren:

- Operativsystemet i Bobs PC lager en **DHCP request melding** og putter den inn i **UDP-segment** med destinasjonsport 67 (DHCP server), og kildeport 68 (DHCP klient). UDP-segmentet innkapsles i et **IP-datagram** med en kringkastings IP-adresse (255.255.255.255) og en kilde IP-adresse på 0.0.0.0, siden Bobs PC ikke har noen IP-adresse enda.
- IP-datagrammet som inneholder DHCP-req-meldingen er så plassert i en **Ethernet frame**. Ethernet rammen har en destinasjons MAC-adresse på FF:FF:FF:FF:FF:FF, slik at rammen blir sendt til alle enhetene koblet til svitsjen (og forhåpentligvis en DHCP server) - Rammens kilde MAC-adresse er den av Bobs bærbar PC, 00:16:D3:23:68:8A.
- Den kringkastede Ethernetrammen er den første rammen sendt av Bobs PC til Ethernetsvitsjen. Svitjen kringkaster den innkommende rammen til alle utgangsportene, inkludert den koblet til ruteren.
- Ruten mottar rammen, inneholdende DHCP-requesten på interfacet med MAC-adresse 00:22:6B:45:1F:1B, og IP-datagrammet blir ekstrahert fra Ethernetrammen. Datagrammets kringkastede IP-destinasjonsadresse indikerer at dette IP-datagrammet skal behandles av øvrelagsprotokoller ved denne noden, slik at datagrammets nyttelast (et UDP-segment) dermed blir **demultiplekset** opp til UDP, og DHCP-forespelsen meldingen blir hentet fra UDP-segmentet. DHCP-serveren har nå DHCP-forespørsel meldingen.
- La oss anta at DHCP-serveren kan allokerere IP-adresser i **CIDR** (Classless Inter-Domain Routing) block 68.85.2.0/24. La oss anta at DHCP serveren allokerer adresse 68.85.2.101 til Bobs PC. DHCP-serveren alger så en **DHCP ACK melding** inneholdene denne IP-adressen, i tillegg til IP-adressen til DNS-serveren (68.87.71.226), IP-adressen til standard gateway ruten (68.85.2.1), og subnettmasken (68.85.2.0/24). DHCP meldingen puttes i et UDP-segment, som puttes inn i IP-datagram, som igjen puttes inn i en Ethernetramme. Ethernetrammen har en kilde-MAC-adresse til ruterens interface til hjemmenettverket (00:22:6B:45:1F:1B) og en destinasjons-MAC-adresse til Bobs datamasking (00:16:D3:23:68:8A).

6. Ethernetrammen inneholdene DHCP ACK-en blir sent av ruteren til svitsjen. Siden svitsjen er \*self-learning og har tidligere mottat en Ethernetramme (DHCP request) fra Bobs PC, vet svitsjen at den skal videre sende rammen adressert til 00:16:D3:23:68:8A kun til den eneste utgangsporten går til Bobs PC.
7. Bobs PC mottar Ethernet rammen, inneholdne DHCP ACK-en, ekstraherer IP-dagagrammet fra Ethernetrammen, ekstraherer UDP-segmentet fra IP-dagagrammet og ekstraherer DHCP ACK meldingen fra UDP-segmentet. Bobs DHCP-klient lagrer så dens IP-adresse, IP-adressen til dens DNS-server. Den installerer også adresen til den standarde gateway serveren inn i dens **IP forwarding table**. Bobs PC vil nå kunne sende alle dagammer med destinasjonsadresser utenfor dens subnett 68.85.2.0/24, til den standarde gateway ruteren. På dette tidspunktet har Bobs laptop initialisert nettverkskomponenter og er klar til å begynne å behandle webside-henting. (Noter at det er kun de to siste stegene beskrevet i Kap. 4 om DHCP som er nødvendige)

### Still Getting Started: DNS and ARP

Når Bob skriver inn URL-en for [www.google.com](http://www.google.com) inn i nettleseren sin, begynner en lag kjede av hendelser som etterhvert resulterer i at Google sin hjemmeside blir vist i nettleseren hans. Bobs nettleser begynner prosessen ved å lage en **TCP-socket** som vil bli brukt til å sende **HTTP-requesten** til [www.google.com](http://www.google.com). For å kunne lage en socket, må Bobs PC vite IP-adressen til [www.google.com](http://www.google.com). Det er **DNS protokollen** som tilbyr navn-til-IP-adresse oversettingstjenesten.

1. Operativsystemet på Bobs PC lager en **DNS query melding**, og putter strengen "[www.google.com](http://www.google.com)" i spørsmål-seksjonen i DNS-meldingen. Denne DNS-meldingen plasseres i et UDP-segment med destinasjonsport 53 (DNS-server). UDP-segmentet plasseres i et IP-dagagram med destinasjons IP-adressen 68.87.71.226 (DNS-adressen returnert i DHCP ACK-meldingen) og en kilde IP-adresse på 68.85.2.101.
2. Bobs datamasking plasserer så dagagrammet i en Ethernetramme. Denne rammen blir sendt (addresert, i linklaget) til gatewayruteren. Men selvom Bobs datamaskin vet IP-adressen til skolens gateway ruter (68.82.2.1) via DHCP ACK-meldingen, vet den ikke gateway ruterens MAC-adresse. For å finne MAC-adressen til ruteren, må Bobs datamaskin bruke **ARP protokollen**.
3. Bobs datamaskin lager en **ARP spørre-melding** med en mål-IP-adresse på 68.85.2.1 (gateway ruteren), plasserer ARP-meldingen i en Ethernet-ramme med kringkastings destinasjonsadresse (FF:FF:FF:FF:FF:FF) og sender Ethernetrammen til svitsjen, som leverer rammen til alle tilkoblede enheter, i tillegg til gatewayruteren.
4. Gatewayruteren mottar rammen med ARP-spørremeldingen på interfacet til skolens nettverk, og finner mål-IP-adressen til 68.85.2.1 i ARP-meldingen og matcher denne IP-adressen med sitt interface. Gatewayruteren forbereder et **ARP reply**, indikerer sin MAC-adresse (00:22:6B:45:1F:1B) korresponderer med mål-IP-adressen. Plasserer ARP-replymeldingen i en Ethernetramme, og sender den til svitsjen som leverer rammen hos Bobs datamaskin.
5. Bobs datamaskin mottar rammen med ARP-replymeldingen, og ekstraherer MAC-adressen til gatewayruteren fra rammen.
6. Bobs datamaskin kan endelig adressere Eternetrammen med DNS-spørringen til gatewayruterens MAC-adresse. Han sender IP-dagagrammet med IP-destinasjonsadressen til DNS-serveren, i rammen med destinasjonsadresse hos gatewayruteren, til svitsjen, som leverer rammen hos gatewayruteren.

### Still Getting Started: Intra-Domain Routing to the DNS Server

1. Gatewayruteren mottar rammen og ekstraherer IP-dagagrammet inneholdene DNS-spørringen. Ruteren slår opp destinasjonsadressen til dagagrammet (68.87.71.226) og bestemmer fra forwardingtabelen sitt at dagagrammet skal bli set til den vestre ruteren i Comcast nettverket i Figur 5.32 over. IP-dagagrammet legges i en linklagsramme passende for linken som går mellom skolens ruter og Comcasts venstre ruter, og rammen blir sendt over denne koblingen.
2. Ruteren mest til venstre i Comcast sitt nettverk mottar rammen, ekstraherer IP-dagagrammet, eksaminerer datagrammets destinasjonsadresse, og bestemmer den utgående interfacet som dagagrammet skal bli videresendt til DNS-serveren til, ut fra dens forwarding table, som har blitt fyllt av Comcasts intra-domain protocol (som *RIP*) i tillegg til Internets inter-domain protocol, **BGP**.
3. Til slutt kommer IP-dagagrammet med DNS-spørringen til DNS-serveren. DNS-serveren ekstraherer DNS-spørringen, slår opp på navnet [www.google.com](http://www.google.com) i DNS databasen sin, og finner **DNS resource recorden** som inneholder IP-adressen (64.233.16.105) for [www.google.com](http://www.google.com) (antatt at den er cachet i DNS-serveren). DNS-serveren lager en **DNS reply melding** inneholdene vertsnavn-til-IP-adressemappingen, og plasser DNS-replymeldingen i et UDP-segment, og segmentet i et IP-dagagram adressert til Bobs datamaskin. Dette dagagrammet vil bli videresendt tilbake gjennom Comcasts nettverk til skolens ruter og fra der, gjennom Ethernetsvitsjen til Bobs datamaskin.
4. Bobs datamaskin ekstraherer UP-adressen til serveren [www.google.com](http://www.google.com) fra DNS-meldingen. Til slutt, er Bobs datamaskin endelig klar for å ta kontakt med [www.google.com](http://www.google.com) serveren.

### Web Client-Server Interaction: TCP and HTTP

- Nå som Bobs datamaskin har IP-adressen til [www.google.com](http://www.google.com), kan den lage en **TCP-socket** som vil bli brukt til å sende **HTTP GET** meldingen. Når Bob lager en TCP socket, vil TCP-en i Bobs datamaskin først utføre en **three-way handshake** med TCP-en hos [www.google.com](http://www.google.com). Bobs datamaskin lager et **TCP SYN** segment med destinasjonsport 80 (for HTTP), plasserer TCP-segmentet i et IP-datagram med destinasjons-IP-adressen 64.233.169.105 ([www.google.com](http://www.google.com)), plasserer datagrammet i en ramme med destinasjons MAC-adresse 00:22:6B:45:1F:1B (gatewayruteren) og sender rammen til svitsjen.
- Ruteren i skolens nettverk, Comcasts nettverk og Googles nettverk videresender datagrammet med TCP SYN til [www.google.com](http://www.google.com), ved å bruke forwardingtablene i hver ruter.
- Til slutt kommer datagrammet med TCP SYN frem til [www.google.com](http://www.google.com). TCP SYN-meldingen blir ekstrahert fra datagrammet, demultiplexet til velkomstsocketen med port 80. En connection socket blir laget for TCP-tilkoblingen mellom Google HTTP-serveren og Bobs datamaskin. Et **TCP SYN ACK**-segment blir generert, plassert i et datagram og adressert til Bobs datamaskin, plassert i en linklagsramme passende for koblingen som kobler sammen [www.google.com](http://www.google.com) til sin første hop-ruter.
- Datagrammet med TCP SYNACK-segmentet blir videresendt gjennom Google, Comcast, skolenettverket, og til slutt hos Ethernethkortet på Bobs datamaskin. Datagrammet blir demultiplexet til TCP-socketen laget i steg 1, som går inn i sin tilkoblede tilstand.
- Med socketen på Bobs PC klar for å sende bytes til [www.google.com](http://www.google.com), lager Bobs nettleseer en HTTP GET-melding med URL-en som ønskes. HTTP GET-meldingen blir skrevet inn i socketen, men GET-meldingen som blir payloaden til et TCP-segment. TCP-segmentet blir plassert i et datagram, og sendt og levert hos [www.google.com](http://www.google.com)\* etter stegene 1-3 over.
- HTTP-serveren hos [www.google.com](http://www.google.com) leser HTTP GET-meldingen fra TCP-socketen, lager en **HTTP response** melding, plasserer den forespurte nettsidens innhold i bodyen til HTTP response-meldingen, og sender den inn i TCP-socketen.
- Datagrammet med HTTP responsmeldingen blir videresendt gjennom Google, Comcast og skolenettverekene, og ankommer Bobs datamaskin. Bobs nettleseerprogram leser HTTP Responsmeldingen fra socketen, ekstraherer html-en for nettsiden fra bodyen til HTTP Resposnen, og viser endelig nettsiden.

## Kapittel 6 - Wireless and Mobile Networks

---

To viktige ord å kunne se forskjellen på er trådløs og mobilitet – for det vil kunne la oss bedre isolere, identifisere og mestre nøkkelkonseptene i hvert område.

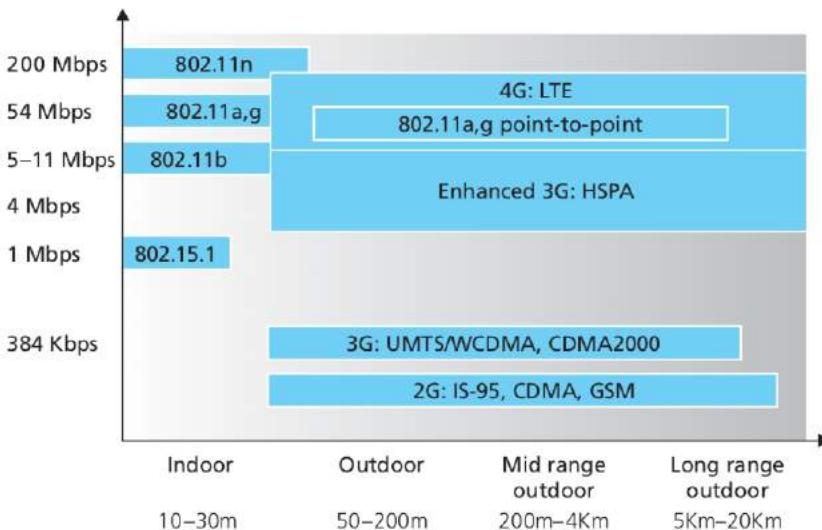
Noter at det er mange nettverksmiljøer der nodene er *trådløse*, men ikke *mobile* (f.eks. trådløse hjemme eller kontornettverk med stasjonære arbeidsstasjoner og store skjermer, og at det er begrensede former for mobilitet som ikke krever trådløse koblinger (for eksempel en arbeidstaker som bruker en kabelt bærbar PC hjemme, slår av den bærbare datamaskinen, stasjoner til jobb og kobler den bærbare til selskapets kablede nettverk)).

Selvfølgelig er mange av de mest spennende nettverksmiljøene de der brukerne er både trådløse og mobile, for eksempel et scenario der en mobilbruker i en bil opprettholder en Voice-over-IP-samtale og flere pågående TCP-tilkoblinger mens mens han raser ned motorveien til 160 kilometer i timen. Det er her, i krysset mellom trådløs og mobilitet, at vi finner de mest interessante tekniske utfordringene!

### Introduction

Vi kan begynne med å holde diskusjonen generell nok til å dekke en rekke med nettverk, inkludert både trådløs LANs som IEEE 802.11 og mobilnett slik som et 3G nettverk. Vi kan identifisere følgende elementer i et trådløst nettverk:

- Wireless hosts*. Som i kablete nettverk er verter endesystemer som kjører applikasjoner En **trådløs vert** kan være en datamaskin, smarttelefon, eller en iPad. Vertene selv kan være mobile, eller ikke.
- Wireless links*. En vert kobler seg til en basestasjon (definert under) eller til andre trådløse vertern gjennom en **trådløs kommunikasjonskobling/link**. Forskjellige trådløse koblingsteknologier har forskjellige overføringsrater, og kan overføre over forskjellige avstander. Figur 6.2 viser to nøkkelegenskaper (*dekningsområde* og *koblingsrate*) av de mer populære trådløse nettverksstandardene. (Figuren er bare ment å gi en grov ide om disse egenskapene).



**Figure 6.2** ♦ Link characteristics of selected wireless network standards

- **Base station.** **Basestasjonen** er nøkkeldelen for den trådløse infrastrukturen. Ulik en trådløs vert og en trådløs kobling, er basestasjonen ingen åpenbar motstykke til kabelt nettverk. En basestasjon er ansvarlig for sending og mottak av data (for eksempel pakker) til og fra en trådløs vert som er tilknyttet den basestasjonen. En basestasjon vil ofte være ansvarlig for å koordinere overføringen av flere trådløse verter som den er tilknyttet. Når vi sier at en trådløs vert er "tilknyttet" en basestasjon, mener vi at (1) verten befinner seg innenfor basestasjonens trådløse kommunikasjonsdistanse, og (2) verten bruker den basestasjonen for å videresende data mellom den (verten) og det større nettverket. **Celltowers** (nor. teletårn) i mobilnett og aksesspunktspunkter i 802.11 trådløse LAN er eksempler på basestasjoner.

Verter tilknyttet en basestasjon refereres ofte til som opererende i *infrastrukturmodus*, siden alle tradisjonelle nettverkstjenester (for eksempel adressetildeling og ruting) er gitt av nettverket som en vert er koblet til via basestasjonen. I **ad hoc-nettverk** har trådløse verter ingen slik infrastruktur som å koble til. I mangel av slik infrastruktur må vertene selv sørge for tjenester som ruting, adressetildeling, DNS-lignende navneoversettelse og mer.

Når en mobil (*adj.*) vert beveger seg utenfor rekkevidden til en basestasjon og inn i rekkevidden til en annen, vil den endre sin tilknytning i det større nettverket (dvs. bytte basestasjon som det er tilknyttet) en prosess referert til som **handoff**.

- Slik mobilitet gir mange utfordrende spørsmål. Hvis en vert kan flytte seg, hvordan finner man mobilens vert nå i nettverket, slik at data kan videresendes til den mobile verten? Hvordan håndteres adressering, gitt at en vert kan være på en av mange mulige steder? Hvis verten beveger seg under en TCP-tilkobling eller telefonsamtale, hvordan blir data rutet slik at forbindelsen fortsetter uavbrutt?
- **Network infrastructure.** Dette er det større nettverket som en trådløs vert kan ønske å kommunisere med.

Etter å ha diskutert "brikkene" i et trådløst nettverk, bemerker vi at disse brikkene kan kombineres på mange forskjellige måter for å danne forskjellige typer trådløse nettverk. På høyeste nivå kan vi klassifisere trådløse nettverk i henhold til to kriterier: (1) om en pakke i det trådløse nettverket krysser nøyaktig en **trådløs hop** eller **flere trådløse hops**, og (2) om det er *infrastruktur* slik som en basestasjon i nettverket:

- **Single-hop, infrastructure-based.** Disse nettverkene har en basestasjon som er koblet til et større kabelt nettverk (for eksempel Internettet). Videre er all kommunikasjon mellom denne basestasjonen og en trådløs vert over en enkelt trådløs hop. 802.11-nettverkene du bruker i klasserommet, på kaféen eller biblioteket, og 3G-mobilnettverk, faller alle i denne kategorien.
- **Single-hop, infrastructure-less.** I disse nettverkene er det ingen basestasjon som tilkoblet til det trådløse nettverket. Men som vi vil se, kan en av nodene i dette enkelthopp-nettverket imidlertid koordinere overføringene til de andre noder. Bluetooth-nettverk og 802.11-nettverk i ad hoc-modus er enkelthopp, infrastrukturfrie nettverk.

- *Multi-hop, infrastructure-based.* I disse nettverkene er det en basestasjon til stede som er kabelt til det større nettverket. Noen trådløse noder kan imidlertid måtte videresende kommunikasjonen via andre trådløse noder for å kunne kommunisere via basestasjonen. Noen trådløse sensornettverk og såkalte **trådløse nettverksnett** (eng. **wireless mesh networks**) faller i denne kategorien.
- *Multi-hop, infrastructure-less.* Det er ingen basestasjon i disse nettverkene, og noder må kanskje videresende meldinger blant flere andre noder for å nå en destinasjon. Noder kan også være mobile, med tilkoblingsmuligheter mellom noder - en klasse av nettverk kjent som **mobile ad hoc-nettverk (MANETs)**. Hvis de mobile nogene er kjøretøy, er nettverket et **kjøretøy ad hoc-nettverk (VANET)**.

I dette kapittelet vil vi for det meste begrense oss til *single-hop-nettverk*, og deretter for det meste til *infrastrukturbaserte nettverk*.

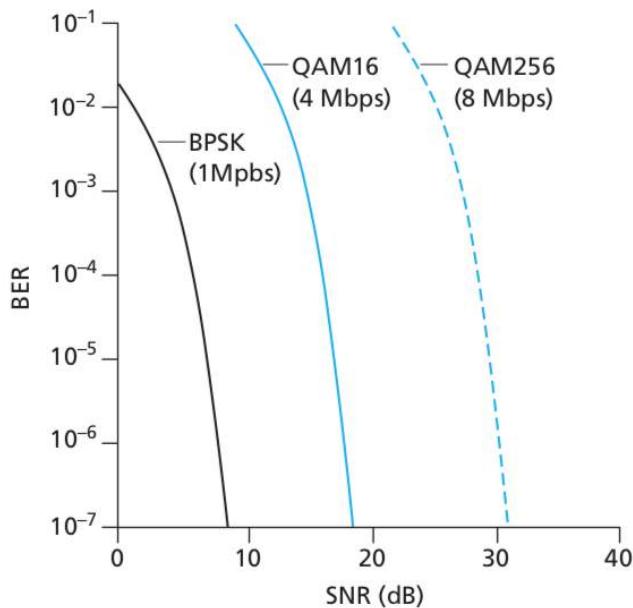
## Wireless Links and Network Characteristics

La oss begynne å se på et enkelt kabelt nettverk, et hjemmenettverk, med en kabelt Ethernet svits som kobler sammen vertene. Dersom vi erstatter det kablede Ethernettet med et trådløs 802.11 nettverk, et trådløs nettverksinterface ville erstattet vertens Ethernet-interface, men det ville ikke vært nødvendig med noen endringer på nettverkslaget eller over. Dette antyder at vi fokuserer vår oppmerksomhet på *linklaget* når vi ser etter viktige forskjeller mellom kablede og trådløse nettverk. Faktisk kan vi finne en rekke forskjeller mellom kablede og trådløse koblinger:

- *Decreasing signal strength.* Elektromagnetisk stråling demper når den passerer gjennom materiell (for eksempel et radiosignal som går gjennom en vegg). Selv i ledig plass vil signalet spre seg, noe som resulterer i redusert signalstyrke (noen ganger referert til som **path loss**) når avstanden mellom avsender og mottaker øker.
- *Interference from other sources.* Radiokilder som sender på samme frekvensbånd vil forstyrre hverandre. For eksempel overfører 2,4 GHz trådløse telefoner og 802.11b trådløse LAN på samme frekvensbånd. Dermed kan 802.11b trådløs LAN-brukeren som snakker på en 2,4 GHz trådløs telefon forvente at verken nettverket eller telefonen vil fungere spesielt godt. I tillegg kan det komme interferens fra andre overførende kilder, elektromagnetisme i området (f.eks. nær motor, mikrobølgeovn).
- *Multipath propagation.* **Multipath propagation** oppstår når deler av den elektromagnetiske bølgen reflekterer gjjenstander og bakken, og tar baner av forskjellige lengder mellom en avsender og en mottaker. Dette resulterer i uskarphet av mottatt signal ved mottakeren. Det å flytte objekter mellom avsender og mottaker kan føre til at multipath propagation endres over tid.

La oss se på en vert som mottar et trådløst signal - verten mottar et elektromagnetisk signal som er en kombinasjon av en nedgradert form av det originale signalet sendt fra senderen og bakgrunnsstøy fra miljøet rundt. **Signal-to-noise ratioen (SNR)** er et relativt mål på styrken til det mottatte signalet og denne støyen. SNR er vanligvis målt i enheter av desibel (dB). SNR målt i dB, er 20 ganger forholdet mellom 10-base-logaritmen av amplituden til det mottatte signalet til lydens amplitude. For våre formål trenger vi bare å vite at en større SNR gjør det lettere for mottakeren å trekke ut det overførte signalet fra bakgrunnsstø.

Figur 6.3 under, viser **bit-error raten (BER)** - som grovt sett er sannsynligheten for at en overført bit som blir mottatt har error hos mottakeren - versus SNR for tre forskjellige modelleringsteknikker for koding av informasjon for overføring.



**Figure 6.3 ♦ Bit error rate, transmission rate, and SNR**

Figur 6.3 illustrerer flere fysiklag-karakteristikker som er viktige for å forstå høyrelags trådløse kommunikasjonsprotokoller:

- For en gitt moduleringsskjema, desto høyere SNR, desto lavere BER. Siden en sender kan øke SNR-en ved å øke overføringskraften, kan senderen minske sannsynligheten for at en ramme som blir mottatt ved å øke overføringskraften.
- For en gitt SNR, en moduleringsteknikk med høyere bit-overføringsrate vil ha høyere BER. I figuren har BPSK (1 Mbps) mindre BER enn QAM16 (4 Mbps), grunner overføringsraten.
- Dynamisk utvelgelse av fysiklagsmodulasjonsteknikken kan brukes til å tilpasse modulasjonsteknikken til kanalforhold. SNR (og dermed BER) kan endres som følge av mobilitet eller på grunn av endringer i miljøet. Adaptiv modulering og koding brukes i celledatasystemer og i 802.11 WiFi og 3G-celedatametene som vi skal se på snart. Dette tillater for eksempel valget av en modulasjonsteknikk som gir den høyeste overføringshastigheten mulig underlagt en begrensning på BER, for gitt kanalegenskaper.

En høyere og tidsvarierende bit-error rate er ikke det eneste forskjellen mellom kablede og trådløse koblinger. **Det skjulte terminalproblemet** - at fysiske objekter, f.eks. fjell, kan hindre overføringer - og **fading** - signalet blir svakere, og kan skje udetektbare kollisjoner - gjør at multiaccess i et trådløst nettverk betydelig mer komplekst enn i et kabelt nettverk.

## WiFi: 802.11 Wireless LANs

Selvom mange teknologier og standarder for trådløse LANs ble utviklet på 1990-tallet, er det en spesifikk klasse av standarder som ble vinneren: det **IEEE 802.11 trådløse LAN-et**, også kjent som **WiFi**. Vi skal se på 802.11 trådløse LANs, og se deres rammestruktur, medium aksessprotokoller, og internetworking av 802.11 LANs med kablet Ethernet LANs.

Det er flere 802.11 standarder for trådløst LAN, inkludert 802.11b, 802.11a, 802.11g. Tabellen 6.1 viser hovedtrekkene ved disse standardene.

- De bruker alle samme mediums-aksessprotokoll, CSMA/CD (Carrier Sense Multi Access / Collision Detection).
- Alle har samme rammestruktur for linklagsrammene.
- Alle tre standarder har muligheter for redusere overføringsraten for å kunne sende over større avstander.
- Alle standardene tillater både "infrastrukturmodus" og "ad hoc modus"

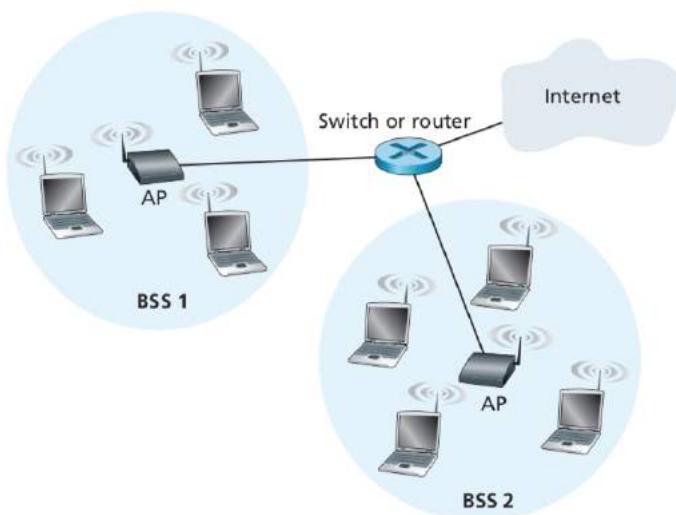
| Standard | Frequency Range (United States) | Data Rate     |
|----------|---------------------------------|---------------|
| 802.11b  | 2.4–2.485 GHz                   | up to 11 Mbps |
| 802.11a  | 5.1–5.8 GHz                     | up to 54 Mbps |
| 802.11g  | 2.4–2.485 GHz                   | up to 54 Mbps |

**Table 6.1** ♦ Summary of IEEE 802.11 standards

En relativt ny WiFi-standard, 802.11n (2012) bruker multiple input multiple output (*MIMO*) antenner; med f.eks. to eller flere antenner på den sendende siden, og to eller flere antenner på den mottakende siden som overfører/mottar forskjellige signaler. Avhengig av moduleringskjemaet, kan overføringshastigheter på flere hundre megbits per sekund oppnås med 802.11n.

### The 802.11 Architecture

Figur 6.7 illustrerer hovedkomponentene i 802.11 trådløse LAN arkitekturen. Fundamentalbyggeklossen til 802.11 arkitekturen er **basic service settet (BSS)**. En BSS inneholder en eller flere trådløse stasjoner og en sentral **basestasjon**, kjent som et **aksesspunkt (AP)** i 802.11 jargon. Figur 6.7 viser AP-en til hver av de to BSS-ene som kobler seg til en sammenkoblingsenhet (slik som en svitsj eller ruter) som igjen leder til internettet.



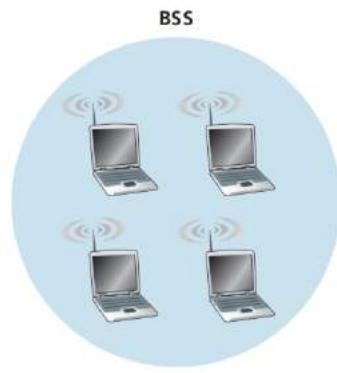
**Figure 6.7** ♦ IEEE 802.11 LAN architecture

I et typisk hjemmenettverk er det en AP og en ruter (typisk integrert sammens som en enhet) som kobler BSS-en til internettet.

Som med Ethernet-enheter har 802.11 trådløse stasjoner en 6-byte MAC-adresse som er lagret i firmwaren til stasjonens adapter. Hver AP har også en MAC-adresse for sit trådløse interface.

Trådløse LANs som distribuerer AP-er, kalles ofte **infrastruktur trådløse LAN**, med "infrastrukturen" som AP-ene, sammen med den kablede Ethernet-infrastrukturen som forbinder AP og en ruter.

Figur 6.8 viser at IEEE 802.11 stasjoner også kan gruppere dem selv sammen for å forme et ad hoc nettverk - et nettverk uten sentral kontroll og uten noen tilkobling til den "ytre verden".



**Figure 6.8** ♦ An IEEE 802.11 ad hoc network

#### Channels and Association

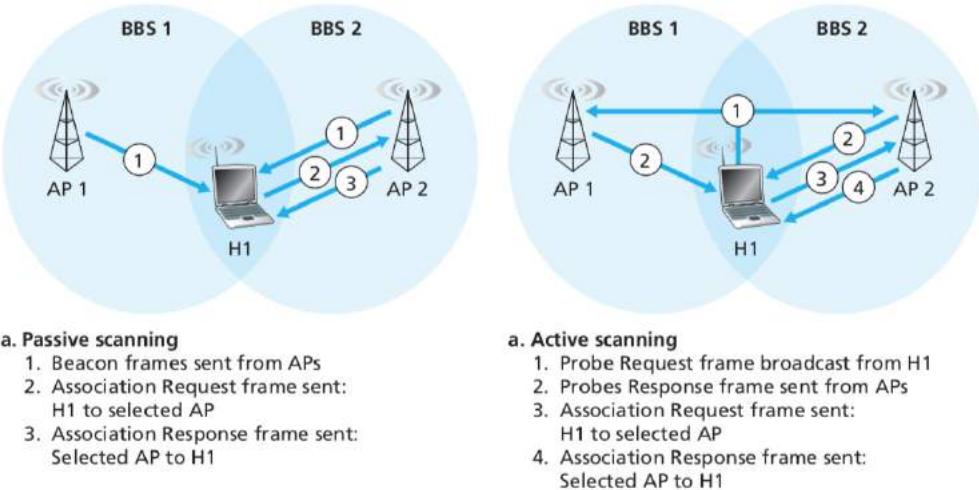
Når en nettverksadministrator installerer en AP, vil administratoren tilegne ett en- eller to-ords **Service Set Identifier (SSID)** til aksesspunktet. Når man ser på *tilgjengelige nettverk* på PCen så er det SSID-ene til alle AP-ene i området man ser. Administratoren må også velge kanalnummeret til AP-en. Fra Figur 6.1 husker vi at 802.11 opererer på et frekvensområde på 2.4 GHz til 2.485 GHz. Innenfor dette 85 MHz båndet har vi 11 delvis overlappende kanaler, som man må velge mellom. Med 11 Mbps for 802.11 på dette frekvensområdet, så vil en nettverksadministratør kunne oppnå 33 Mbps, ved å installere tre 802.11b AP-er på samme sted, med tilegnede kanaler 1,6, og 11 til AP-ene, og koble disse sammen med en svitsj.

En **WiFi-jungel** er en fysisk plassering der en trådløs stasjon mottar et tilstrekkelig sterkt signal fra to eller flere AP-er. For eksempel, i mange kaféer i New York City, kan en trådløs stasjon hente et signal fra mange nærliggende AP-er. En av AP-ene kan administreres av kafeen, mens de andre AP-ene kan være i boliger i nærheten av kafeen. Hver av disse AP-ene vil trolig være lokalisert i et annet IP-undernett og ville ha blitt tildelt en kanal, uavhengig av hverandre.

Anta nå at du kommer inn i en slik WiFi-jungel med din bærbar datamaskin, søker trådløs internetttilgang. Anta at det er fem AP-er i WiFi-jungelen. For å få tilgang til Internett, må din trådløse stasjon være med i et av delnettene, og må derfor **knytte (eng. associate)** seg til nøyaktig én av AP-ene. Tilknytning betyr at den trådløse stasjonen oppretter en virtuell ledning mellom seg selv og AP-en. Spesielt vil kun den tilknyttede AP-en sende datarammer til din trådløse stasjon, og den trådløse stasjonen vil sende datarammer til Internett bare gjennom den tilhørende AP-en. Men hvordan forbinder din trådløse stasjon med en bestemt AP? Og mer fundamentalt, hvordan vet din trådløse stasjon hvilke AP-er, hvis noen, er der ute i jungelen?

- 802.11-standarden krever at en AP regelmessig sender **beacon frames**, som hver inneholder APs SSID og MAC-adresse. Din trådløse stasjon, som vet at APer sender ut beaconrammer, skanner de 11 kanalene, søker beaconrammer fra noen AP'er som kan være der ute. Etter at du har lært om tilgjengelige APer fra *beacon frames*, velger du (eller din trådløse vert) en av AP-ene for tilknytning.
- Det er ingen bestemt algoritme for å bestemme hvilken av de tilgjengelige AP-ene som man skal knytte seg til. Typisk er det verten som velger AP-en som leverer de beacon-rammene med høyest signal.

Prosessens av å skanne kanaler og høre etter beaconrammer er kjent som **passiv skanning** (Figur 6.9a). En trådløs vert kan også utføre **aktiv skanning**, ved å kringkaste en *sonderamme (eng. probe frame)* som blir mottatt av alle AP-ene innenfor vertens rekkevidde, som vist i figur 6.9b. AP-er reagerer på sondeforespørselsrammen med en sondresponsramme. Den trådløse verten kan deretter velge AP som skal tilknyttes blant de responterende AP-ene.



**Figure 6.9** ♦ Active and passive scanning for access points

Etter å ha valgt en AP som man skal knytte seg med, vil den trådløse verten sende en assosiasjons-request-ramme til AP-en, og AP-en responderer med en assosiasjons-response-ramme. Noter at denne andre request-response handshaken er nødvendig med aktiv skanning, da AP-en som svarer på probe-requesten ikke vet hvilke av de (potensielt mange) responderende AP-ene verten vil ønske å koble seg til.

For å kunne etablere assosiasjon til en bestemt AP, kan det hende at den trådløse stasjonen er nødt til autentisere seg selv. En av måtene, som brukes av mange selskaper, er å gi tilgang basert på en stasjons MAC-adresse. En annen måte som brukt på cafeer eller lignende er å bruke brukernavn og passord.

### The 802.11 MAC Protocol

Ettersom flere stasjoner vil kunne ønske å overføre datarammer på samme tid, over samme kanal, trengs det en multiaksessprotokoll for å koordinere disse overføringene. En **stasjon**, er verken en trådløs stasjon eller en AP.

- Som diskutert i kapittel 5 er det generelt tre klasser med multiaksessprotokoller: channel partitioning (inkludert CDMA), random access og taking turns. Inspirert av den store suksessen til Ethernet og protokollen for random access, valgte designerne av 802.11 en random access-protokoll for 802.11 trådløse LAN.
- Denne tilfeldige tilgangsprotokollen refereres til som **CSMA med kollisjonsungåelse**, eller mer kortfattet som **CSMA/CA**. Som med Ethernet's CSMA / CD, står "CSMA" i CSMA / CA for "carrier sense multiple access", noe som betyr at hver stasjon sjekker kanalen før sending, og lar være å sende når kanalen oppfattes optatt. Selv om både Ethernet og 802.11 bruker carrier-sensing random access, har de to MAC-protokollene viktige forskjeller.

For det første, i stedet for å bruke kollisjonsdeteksjon, bruker 802.11 *kollisjonsunngåelsesteknikker*. For det andre, på grunn av de relativt høye bitfeilratene for trådløse kanaler, bruker 802.11 (i motsetning til Ethernet) en linklagsbekreftelse/-overføring (Automatic ReQuest - ARQ) -skjema. Vi vil beskrive 802.11s kollisjonsunngåelse og linklagsbekreftelseskjemaer nedenfor.

Husk fra Kap. 5 at med Ethernet's kollisionsdetekteringsalgoritme, lytter en Ethernet-stasjon til kanalen som den overfører over. Dersom den registrerer at en annen stasjon også overfører, stopper den sin overføring og forsøker å overføre igjen etter å ha ventet en liten, tilfeldig tid. I motsetning til 802.3 Ethernet-protokollen implementerer 802.11 MAC-protokollen ikke kollisjonsdeteksjon. Det er to viktige årsaker til dette:

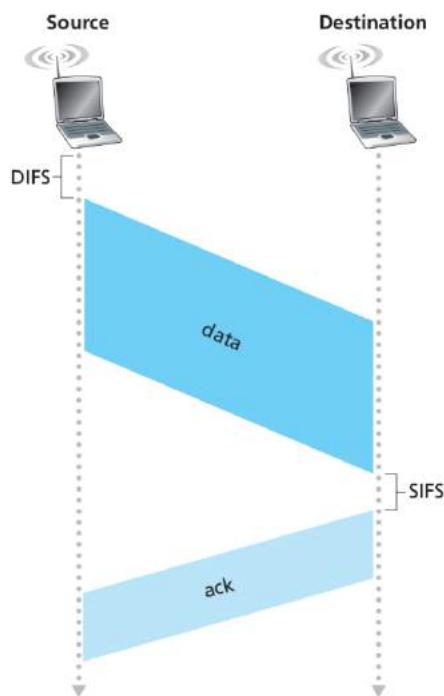
- Muligheten for å detektere kollisjoner avhenger av muligheten for å sende (stasjonens signal) og motta (for å bestemme om en annen stasjon også sender).
  - På grunn av at astyrkett til det mottatte signalet er typisk veldig lite i forhold til det sendte signalet, er det veldig dyrt å lage maskinvare som kan detektere kollisjoner.
- Selv om adapteren kunne ha overført og lyttet samtidig (og antagelig stoppe overføringen når den registrerer en opptatt kanal), vil adapteren fortsatt ikke kunne oppdage alle kollisjoner på grunn av det skjulte terminalproblemet og fading.

Fordi 802.11-trådløse LAN ikke bruker kollisjonsdeteksjon, så blir det slik at når en stasjon begynner å overføre en ramme, sender den rammen i sin helhet. Som det kan forventes, kan overføring av hele rammer (spesielt lange rammer) når kollisjoner er utbredt, senke ytelsen til en multiaksessprotokoll. For å redusere sannsynligheten for kollisjoner, bruker 802.11 flere kollisjonsunngåelsesteknikker, som vi snart skal diskutere.

La oss undersøke 802.11s linklags acknowledgment-skjema. Som vi snakket om i stad er det ikke sikkert at en ramme kommer frem til destinasjonen sin. For å håndtere denne betydelige risikoen for feil, bruker 802.11 MAC-protokollen koblingslagsbekreftelser. Som vist i Figur 6.10, når mottaksstasjonen mottar en ramme som passerer CRC, venter den en kort tidsperiode kjent som **Short Inter-Frame Spacing (SIFS)** og sender deretter tilbake en bekreftelsesramme. Hvis sendestasjonen ikke mottar en bekreftelse innen en gitt tid, forutsetter det at en feil har oppstått og re-sender rammen, ved hjelp av CSMA / CA-protokollen for å få tilgang til kanalen. Hvis en bekreftelse ikke mottas etter et bestemt antall resendinger, gir senderstasjonen opp og kasserer rammen.

Etter å ha diskutert hvordan 802.11 bruker linklagsbekreftelser, er vi nå i stand til å beskrive 802.11 CSMA/CA-protokollen. Anta at en stasjon (trådløs stasjon eller en AP) har en ramme klar for overføring:

1. Dersom stasjonen merker at kanalen er på tomgang, sender den rammen sin etter en kort periode kjent som **Distributed Inter-Frame Space (DIFS)**, se Figur 6.10.
2. Hvis ikke, vil stasjonen bestemme en tilfeldig backoff-verdi ved å bruke binær eksponentiell backoff, og telle den denne verdien når kanalen oppfattes å være ledig. Når kanalen oppfattes opptatt, vil tellerverdien forblie den samme.
3. Når nedtelleren når null, vil stasjonen overføre hele rammen, og venter for en bekreftelse.
4. Dersom en acknowledgement blir mottatt, vet den sendende stasjonen at rammen har blitt mottatt korrekt. Dersom den har en annen ramme å sende begynner CSMA/CA-protokollen på steg 2. Dersom det ikke kommer noen acknowledgement, vil den overførende stasjonen gå tilbake til backoff-fasen i steg 2, med et tilfeldig tall valgt fra et større intervall.



**Figure 6.10** ♦ 802.11 uses link-layer acknowledgments

Husk at under Ethernets CSMA / CD, flere tilgangsprotokoller (Seksjon 5.3.2), begynner en stasjon å overføre så snart kanalen blir oppdaget ledig. Med CSMA / CA, venter stasjonen imidlertid fra å sende mens den teller ned, selv når den registrerer at kanalen er tomgang. Hvorfor tar CSMA / CD og CDMA / CA slike forskjellige tilnærminger her?

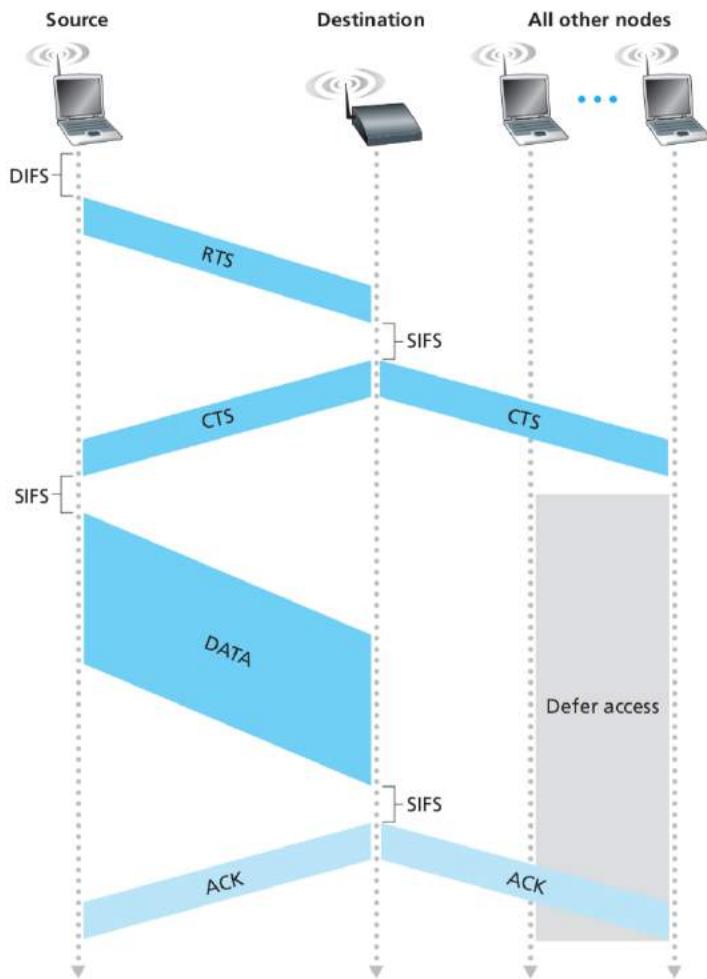
- Dersom to stasjoner ønsker å sende en ramme, og en tredje stasjon overfører en ramme. Vil begge stasjonene sende over pakkene sine når den tredje stasjonen er ferdig med overføre. Som vil resultere i en kollisjon. Men målet med 802.11 er å unngå kollisjoner når det er mulig. Når de to stasjonene oppdager at kanalen er opptatt, går de inn i random backoff, og dersom de velger forskjellige backoff-verdier, vil den ene først starte/vinne, og den andre sin nedteller fryses, og forsetter når den andre stasjonen er ferdig. Kollisjoner kan fortsatt skje da det kan være skjulte terminaler.

## Dealing with Hidden Terminals: RTS and CTS

802.11 MAC-protokollen inneholder også en nifty (men valgfri) reservasjonssystem som hjelper til med å unngå kollisjoner selv i nærvær av skjulte terminaler. La oss undersøke denne ordningen i sammenheng med figur 6.11, som viser to trådløse stasjoner og ett tilgangspunkt. Begge trådløse stasjonene ligger innenfor AP-området (hvis dekning vises som en skyggelagt sirkel) og begge har tilknytning til AP. På grunn av fading er signalområdet for trådløse stasjonene imidlertid begrenset til interiøret i de skraverte kretsene vist i figur 6.11. Dermed er hver av de trådløse stasjonene skjult fra den andre, selv om ingen er skjult fra AP.

Dersom H1 skulle sende en pakke, og halvveis gjennom denne overføringen, så sender også H2 (som ikke registrerer H1s overføring), da vil det skje en kollisjon. Kanalen vil derfor være forkastet under hele H1s og H2s overføring.

For å unngå dette så tilatter IEEE 802.11 protokollen stasjonen å bruke en kort **Request to Send (RTS)** kontrollramme, og en kort **Clear to Send (CTS)** kontrollramme for å reservere aksesen til kanalen. Når en sender ønsker å sende en dataramme, kan den først sende en RTS ramme til AP-en, og indikerer den totale tiden nødvendig for å overføre datarammen og ACK-rammen. Når AP-en mottar denne RST-rammen, responderer den ved å kringkaste en CTS-ramme. Denne CTS-rammen har to funksjoner: Å gi sender eksplisitt tillatelse for å sende, og for å instruere andre stasjoner til å ikke sende under den reserverte tiden.



**Figure 6.12** ♦ Collision avoidance using the RTS and CTS frames

Bruken av RTS og CTS rammer kan øke ytelsen på to måter:

- Det skjulte stasjonsproblemet blir redusert, siden en lang data-ramme bare overføres etter at kanalen er reservert.
- Fordi RTS- og CTS-rammene er korte, varer en kollisjon med en RTS- eller CTS-ramme bare i løpet av den korte RTS- eller CTS-rammen. Når RTS- og CTS-rammene er riktig overført, bør følgende DATA- og ACK-rammer overføres uten kollisjoner.

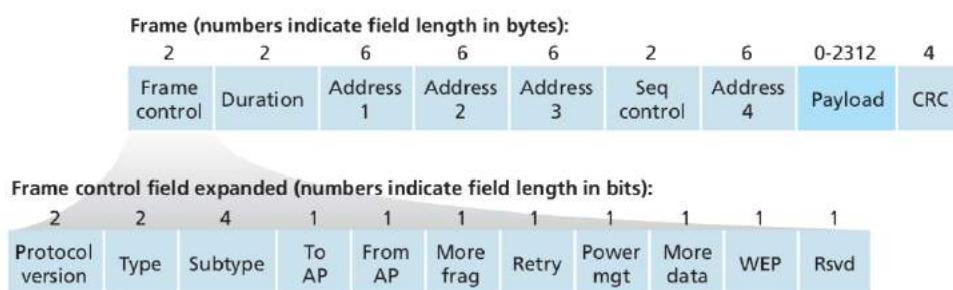
Selvom RTS/CTS kan hjelpe å redusere kollisjoner introduserer den også forsinkelser og bruker kanalressursser.

Derfor brukes RTS/CTS kun for å reservere kanalen for overføring av lange data-rammer.

Vår diskusjon har hittil kun fokusert på 802.11 på en multiaksess måte. Det er verdt å nevne at dersom to noder har retningsantenner, som de peker mot hverandre, og kjører 802.11 protokollen over det som essensielt er en punkt-til-punkt-kobling. Gitt den lave prisen på 802.11-maskinvare, kan bruk av retningsantenner og en økt overføringseffekt tillate 802.11 å bli brukt som et rimelig middel for å gi trådløse punkt-til-punkt-tilkoblinger over ti kilometer unna.

### The IEEE 802.11 Frame

Selvom 802.11 rammen deler mange likheter med en Ethernetramme, så har den også et antall felt som er spesifikk for sin bruk for trådløse koblinger. 802.11 rammen er vist i Figur 6.13. Nummerene over hvert felt i rammen representerer lengden til feltet i *bytes*, tallet over subfeltene er i *bits*.



**Figure 6.13** ♦ The 802.11 frame

#### Payload and CRC field:

I hjertet til rammen, er payloaden, som typisk består av et IP-datagram eller en ARP-pakke. Selvom feltet er tillatt å være så langt som 2,312 bytes, er det typisk mindre enn 1,500 bytes. Som med en Ethernet-ramme, så inkluderer en 802.11-ramme en 32.bit cyclic redundancy check (CRC) slik at mottaker kan bitfeil. Som vi har sett er det mye mer vanlig med bitfeil i trådløse LAN enn i kablede LAN, så her er CRC enda mer nyttefullt

#### Address field:

802.11-rammen har *fire* adressefelt, der hver av dem inneholder en 6-byte MAC-adresse. Men hvorfor fire felt? La oss først se på de tre første adressefeltene:

- Adresse 2 er MAC-adressen til stasjonen som sender rammen.
- Adresse 1 er MAC-adressen til den trådløse stasjonen som skal motta rammen.
- Adresse 3 er MAC-adressen til ruter-interfacet. Denne adressen brukes når svitsjen (AP-en) skal lage en Ethernet-ramme og sende til ruten med interface med MAC-adresse lik adresse 3.

#### Sequence Number, Duration, and Frame Control Fields:

Når en stasjon mottar en ramme fra en annen stasjon og sender tilbake en acknowledgement, kan denne ACK-en gå tapt kan det hende man å sende flere kopier av en gitt ramme. Bruken av sekvensnummere tillater mottaker å skille mellom nye pakker og re-sendinger av gamle pakker. Sekvensnummer-feltet har derfor samme funksjon her på linkaget som i transportlaget i Kap. 3.

Husk at 802.11-protokollen tillater en overføringsstasjon å reservere kanalen i en tidsperiode som inkluderer tiden for å overføre datastammen og tiden til å overføre en bekrefteelse. Denne varighetsverdien er inkludert i rammens *duration-felt* (både for datarammer og for RTS- og CTS-rammene).

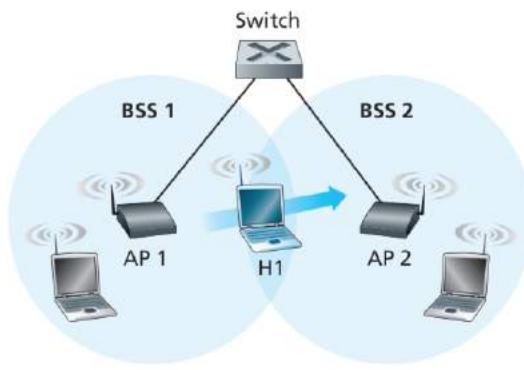
Som vist i figur 6.13, inneholder rammekontrollfeltet mange delfelter. Vi sier bare noen få ord om noen av de viktigste delområdene; Feltypene og undertypefeltene brukes til å skille sammen assosiasjoner, RTS, CTS, ACK og datarammer. Feltene til og fra brukes til å definere betydningen av de forskjellige adressefeltene. (Disse betydningen endres avhengig av om ad hoc eller infrastrukturmodus brukes, og i tilfelle infrastrukturmodus, om en trådløs stasjon eller en AP sender rammen.) Endelig angir WEP-feltet om kryptering blir brukt eller ikke.

#### Mobility in the Same IP Subnet

For å kunne øke den fysiske rekkevidden til et trådløst LAN, bruker selskaper og universiteter ofte flere BSS innenfor samme IP-subnet. Dette øker naturligvis spørsmålet om mobilitet blant BSS-er - hvordan beveger trådløse stasjoner sømløst fra en BSS til en annen mens du opprettholder pågående TCP-økt?

Som vi ser i dette underavsnittet, kan mobilitet håndteres på en relativt enkel måte når BSS-ene er en del av subnettet. Når stasjoner flytter mellom subnett, vil det være behov for mer sofistikerte mobilitetshåndteringsprotokoller.

- La oss se på et eksempel på mobilitet mellom BSS-er i samme subnett. Figur 6.15 viser to sammenkoblede BSS-er med en vert, H1, som beveger seg fra BSS1 til BSS2. Fordi i dette eksempelet er sammenkoblingsenheten som forbinder de to BSS-ene ikke en ruter, og dermed tilhører alle stasjonene i de to BSS-ene, inkludert AP, samme IP-subnett.
- Når H1 beveger seg fra BSS1 til BSS2, kan det dermed beholde sin IP-adresse og alle dens pågående TCP-tilkoblinger. Hvis samtrafikkinnretningen var en ruter, ville H1 måtte ha skaffet seg en ny IP-adresse i subnettverket til den den beveget seg. Denne adresseendringen ville forstyrre eventuelle pågående TCP-tilkoblinger på H1.



**Figure 6.15** • Mobility in the same subnet

Men hva skjer når H1 beveger seg fra BSS1 til BSS2?

- Da H1 vandrer bort fra AP1, oppdager H1 et svekkelsessignal fra AP1 og begynner å skanne etter et sterkere signal. H1 mottar beaconrammer fra AP2 (som i mange institusjoner og universitetsinnstillinger vil ha samme SSID som AP1). H1 disassocierer deretter med AP1 og forbinder med AP2, mens du beholder sin IP-adresse og opprettholder sine pågående TCP-økter.
- Dette tar opp håndoff-problemet fra verts- og AP-visningspunktet. Men hva med svitjsen i figur 6.15? Hvordan vet det at verten har flyttet fra en AP til en annen? Som du kanskje husker fra kapittel 5, er svitjser "self learning" og bygger automatisk forwardingtabellene sine. Denne selvlærerende funksjonen håndterer flyttinger av og til. Svitjser ble imidlertid ikke designet for å støtte svært mobile brukere som vil opprettholde TCP-tilkoblinger mens de beveger seg mellom BSS.
- Husk at før flyttingen har svitjsen en oppføring i videresendingstabellen som parrer H1s MAC-adresse med det utgående svitsj-interfacet som går til H1. Dersom H1 først er i BSS1, blir et datagram adressert til H1 rettet til H1 via AP1. Når H1 binder assosierer seg med BSS2, må rammene dens imidlertid rettes til AP2.
- En løsning er for AP2 å sende en **kringkasting Ethernet-ramme med H1s kildeadresse** til svitjsen like etter den nye foreningen. Når svitjsen mottar rammen, oppdaterer den forwardingtabellen, slik at H1 nås via AP2.

## Advanced Features in 802.11

Vi gjør oss ferdig med 802.11 med en kort diskusjon om to avanserte egenskaper som finnes i 802.11-nettverk. Som vi vil se, er disse egenskapene ikke helt angitt i 802.11-standarden, men gjøres mulig av mekanismer spesifisert i standarden.

### 802.11 Rate Adaption

Som vi se er det forskjellige moduleringsteknikker (med forskjellige overføringsrater som de tilbyr), som passer til forskjellige SNR-scenarioer.

Tenk for eksempel en mobil (*adj.*) 802.11-bruker som i utgangspunktet er 20 meter fra basestasjonen, med et høyt signal-støyforhold. På grunn av den høye SNR-en, kan brukeren kommunisere med basestasjonen ved hjelp av en fysisk lagmodulasjonsteknikk som gir høye overføringshastigheter, samtidig som man opprettholder en lav BER. Dette er en lykkelig bruker! Anta nå at brukeren blir mobil, går bort fra basestasjonen, med SNR faller når avstanden fra basestasjonen øker. I dette tilfellet, hvis moduleringsteknikken som brukes i 802.11-protokollen som opererer mellom basestasjonen og brukeren, ikke endres, vil BER bli uakseptabelt høy da SNR reduseres, og til slutt vil ingen sendte rammer bli mottatt riktig.

Av denne grunn har noen 802.11 implementeringer en **rate adaptjons evne** som adaptivt velger den underliggende fysiske lag moduleringsteknikken som skal brukes, basert på nåværende eller nylige kanalegenskaper.

- Hvis en node sender to bilder på rad uten å motta en bekrefteelse (en implisitt indikasjon på bitfeil på kanalen), faller overføringshastigheten tilbake til neste lavere frekvens. Hvis 10 rammer på rad er bekreftet, eller hvis en tidtaker som sporer tiden siden siste fallback utløper, øker overføringshastigheten til neste høyere hastighet.
- 802.11 rate adaption, og TCP congestion control, ligner det lille barnet som kontinuerlig maser foreldrene sine for mer og mer godteri til foreldrene endelig sier "Nok!" Og barnet rygger av (bare for å prøve igjen senere etter at forholdene forhåpentligvis har blitt bedre!). Det er også foreslått en rekke andre ordninger for å forbedre denne grunnleggende automatiske kursjusteringsordningen.

#### Power Management

Strøm er en verdifull ressurs i mobile enheter, og 802.11-standarden gir dermed strømstyringsfunksjoner som tillater 802.11 noder å minimere mengden tid som deres sense-, transmit- og receive-funksjoner og andre kretser må være på. 802.11 strømstyring fungerer som følger. En node kan eksplisitt skifte mellom sove og våkne tilstander. En node indikerer til AP-en at den skal gå i dvale ved å sette strømstyringsbiten i toppteksten til en 802.11-ramme til 1. En tidsur i noden settes deretter til å våkne noden like før AP er planlagt å sende sin beacon-ramme. Siden AP vet fra den angitte strømstyringsbiten at noden skal sove, vet den at den ikke skal sende noen rammer til den noden, og vil buffere rammer som er bestemt for sovende vert for senere overføring. En knute vil våkne like før AP sender en fyrramme og går raskt inn i fullt aktiv tilstand.

## Cellular Internet Access

I det forrige avsnittet undersøkte vi hvordan en Internett-vert kan få tilgang til Internett når de befinner seg i et WiFi-hotspot, det vil si når det er i nærheten av et 802.11-tilgangspunkt. Men de fleste WiFi-hotspots har et lite dekningsområde på mellom 10 og 100 meter i diameter. Hva gjør vi da når vi har et desperat behov for trådløs internettgang, og vi kan ikke få tilgang til en WiFi-hotspot?

Gitt at mobiltelefoni nå er allestedsnærværende i mange områder over hele verden, er en naturlig strategi å utvide mobilnett, slik at de ikke bare støtter taletelefoni, men også trådløs internettgang. Ideelt sett vil denne Internett-tilgangen være med en rimelig høy hastighet og vil sørge for sømløs mobilitet, som lar brukere beholde TCP-tilkoblingene sine når de er ute å reiser.

## An Overview of Cellular Network Architecture

I vår beskrivelse av mobilnettverksarkitektur i denne delen, adopterer vi terminologien til *Global System for Mobile Communications (GSM)* standarder. På 1980-tallet anerkjente europeerne behovet for et pan-Europeisk digitalt mobiltelefonielt system som ville erstatte de mange inkompatible analoge celletelefonisystemene, som fører til GSM-standarden. Europeere distribuerte GSM-teknologi med stor suksess i begynnelsen av 1990-tallet, og siden da har GSM vokst til å være sjefen av mobiltelefon-verden, med over 80% av alle mobilabonnenter verden over bruker GSM.

Når man snakker om mobilnettverk, klassifiserer man teknologien gjerne i en av flere "generasjoner". Den første generasjonen var designet primært for samtale-trafikk.

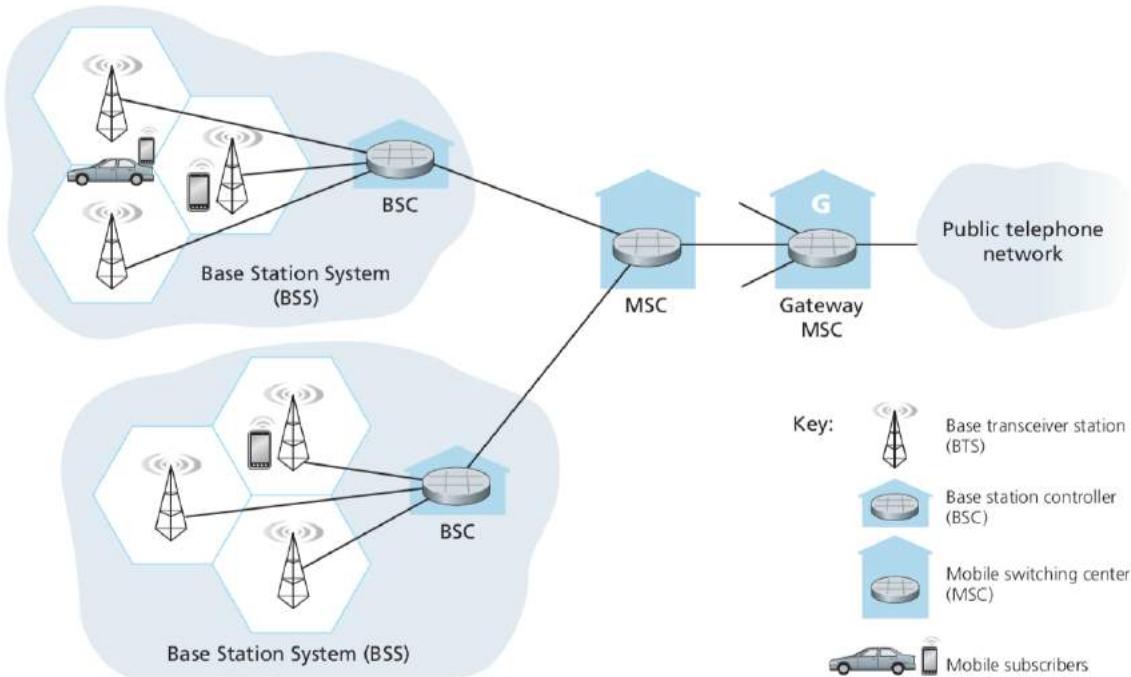
- Første generasjon (1G) systemer var analoge med FDMA systemer designet ekslusivt for voice-only kommunikasjon. Disse systemene finnes ikke idag, og har blitt erstattet av digitale 2G systemer.
- Det originale 2G systemet ble også designet som en voice-tjeneste, men ble senere utvidet (2.5G) til å støtte data (f.eks. internettet) i tillegg til voice-tjeneste.
- 3G-systemene som er i dag, støtter også tale og data, men med stadig større vekt på datakompetanse og høyhastighets radio-aksesslinker.

### Cellular Network Architecture, 2G: Voice Connections to the Telephone Network

Begrepet *cellular* refererer til det faktum at regionen som dekkes av et mobilnettet er delt opp i et antall geografiske dekningsområder, kjent som celler, vist som heksagoner på venstre side av figur 6.18.

Som med 802.11 WiFi-standarden som vi studerte i avsnitt 6.3.1, har GSM sin egen spesifikke terminologi. Hver celle inneholder en **base transceiver station (BTS)** som overfører signaler til og mottar signaler fra de mobile stasjonene i sin celle. Dekningsområdet for en celle avhenger av mange faktorer, inkludert overføringsfaktoren til *BTS*, overføringseffekten til brukerenhetene, hindrende bygninger i cellen og høyden på basestasjonantenna.

Selv om figur 6.18 viser hver celle som inneholder en basestasjonssender som ligger midt i cellen, plasserer mange systemer i dag BTS-er i hjørner hvor tre celler krysser, slik at en enkelt BTS med retningsantenner kan betjene tre celler.



**Figure 6.18** ♦ Components of the GSM 2G cellular network architecture

GSM-standarden for 2G-cellesystemer bruker kombinert FDM / TDM for luftgrensesnittet. Husk at med ren *FDM* er kanalen delt inn i en rekke frekvensbånd med hvert bånd som er viet til en samtale. Husk også at med ren *TDM*, er tiden delt inn i rammer med hver ramme videre partisjonert i slots og hver samtale blir tildelt bruken av en bestemt slot i den roterende rammen. I kombinert FDM / TDM-system blir kanalen delt inn i en rekke frekvensundergrupper. I hvert underfrekvensbånd blir tiden delt inn i rammer og spor.

- GSM-nettverkets **basestasjonskontroller (BSC)** vil typisk betjene flere titalls basestasjoner. BSCs rolle er å allokkere BTS-radiokanaler til mobilabonnenter, utføre paging - dvs. finne cellen der mobilen befinner seg - og utføre overføring av mobilbrukere.
- **Mobile switching center-et (MSC)** spiller en sentral roll i brukerautentisering og *accounting* (f.eks. bestemme om en mobil skal få kunne koble seg til mobilnettverket), samtaletablering, samtale-teardown, og handoff (overføring av mobilbrukere)

### 3G Cellular Data Networks: Extending the Internet to Cellular Subscribers

Nå som vi er i farta, ønsker vi også å lese epost, bruke internettet, få lokasjonsavhengige tjenester (Maps og TripAdvisor) eller stremme video. For å gjøre dette må telefonen kjøre på full TCP/IP-protokol stakk (inkludert fysisk, kobling, nettverk, transport og applikasjonslagene), og koble seg på Internettet via mobilnettverket.

Nedenfor fokuserer vi på UMTS (Universal Mobile Telecommunications Service) 3G-standarder utviklet av 3. Generation Partnership-prosjektet

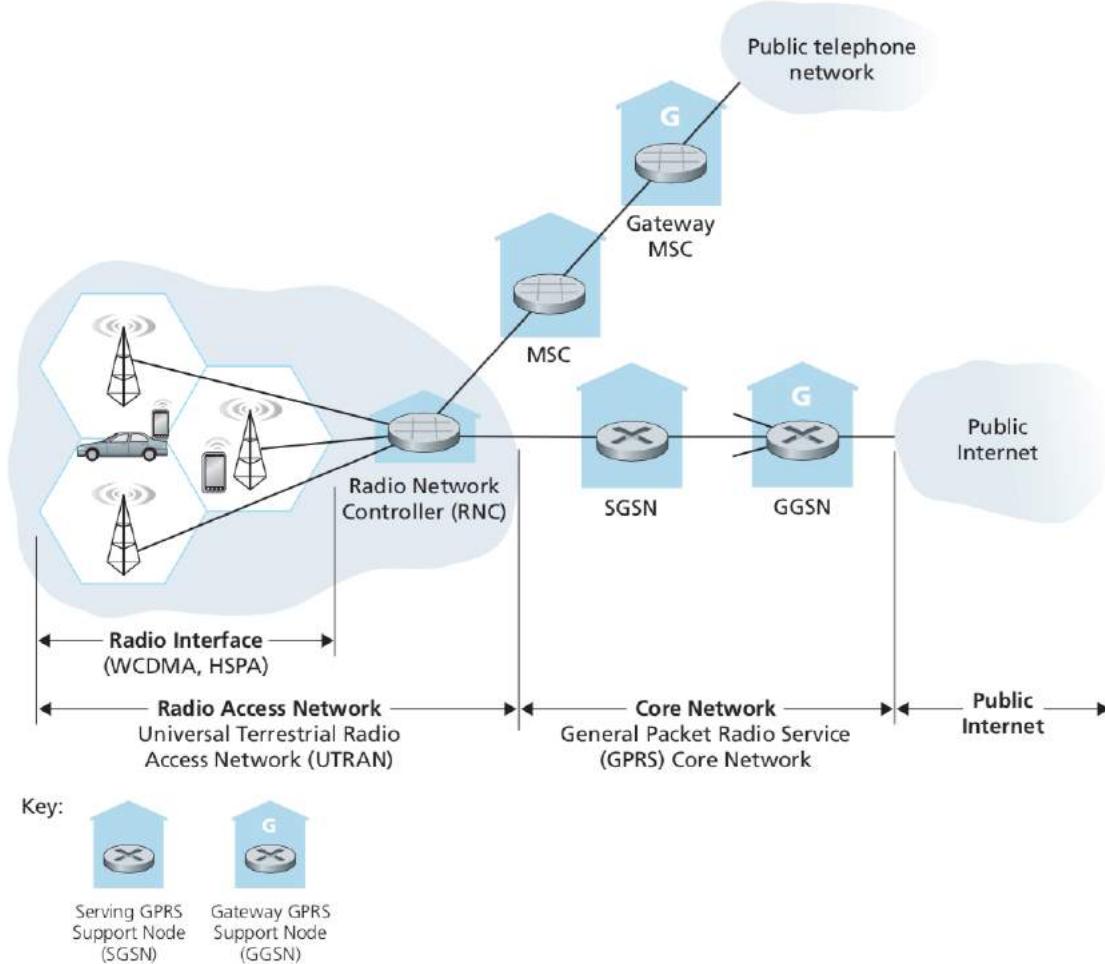
#### 3G Core Network:

3G-core-mobildatanettverket forbinder radioaksessnettverk til det offentlige Internettet. Kjernenettverket samarbeider med komponenter i det eksisterende mobilnettverket (spesielt MSC). På grunn av den betydelige mengden eksisterende infrastruktur i det eksisterende mobilnettverket, er tilnærmingen tatt av designere av 3G-datatjenester klar: \*la det eksisterende kjerne GSM-mobilnettet være uberørt, og legg til flere parallelle datafunksjoner i parallel til det eksisterende mobilnettverket. \*

Alternativet - Integrering av nye datatjenester direkte inn i kjernen i det eksisterende mobilnettverket - ville ha hevet de samme utfordringene som oppstod i Kap.4, hvor vi diskuterte integrering av nye (IPv6) og arv (IPv4) teknologier på Internett.

Det er to typer noder i 3G-kjernenettverket: **Serving GPRS Support Nodes (SGSNs)** og **Gateway GPRS Support Nodes (GGSNs)**. (GPRS står for Generalized Packet Radio Service, en tidlig datatjeneste i 2G-nettverk, her diskuteres den utviklede versjonen av GPRS i 3G-nettverk).

- En SGSN er ansvarlig for å levere datagrammer til/fra de mobile node i radioaksessnettet som SGSN er tilknyttet. SGSN samhandler med mobilkommunikasjonsnettverkets MSC for dette området, og gir brukerautorisasjon og overføring, opprettholder lokasjons(celle)informasjon om aktive mobile node og utfører datagram-videresending mellom mobile node i radionettverket og en GGSN.
- En GGSN fungerer som en gateway, som forbinder flere SGSNs med det større Internett. En GGSN er således det siste stykket 3G-infrastruktur som et datagram med opprinnelse på en mobil node møter før den kommer inn i det større Internett. Til omverdenen ser GGSN ut som hvilken som helst annen gatewayrouter; Mobilheten til 3G-node i GGSNs nettverk er skjult fra omverdenen bak GGSN.



**Figure 6.19** • 3G system architecture

#### 3G Radio Access Network: The Wireless Edge:

3G radioaksessnettverket er det trådløse first-hop nettverket som vi ser som en 3G-bruker. **Radio Network Controlleren (RNC)** kontrollerer typisk flere "cell base transceiver (en enhet som kan motta og sende) stations" likt basestasjonene vi så i 2G-systemer. Hver celles trådløse kobling opererer mellom mobile node og en base-tranceiver-stasjon, slik som i 2G-nettverk.

RNC-en sammenkobler både det kretskoblede mobilnettet via en MSC, og til det pakkesvitsjede Internettet via en SGSN. Selvom 3G-mobiltelefoni- og mobildatatjenester bruker forskjellige kjernenettverk, deler de et felst første/siste-hop radioaksessnettverk.

En betydelig endring i 3G UMTS (mer definert navn for 3G) over 2G-nettverk er at istedet for å bruke GSM sin FDMA/TDMA skjema, bruker UMTS en CDMA teknikk kalt Direct Sequence Spread Spectrum (DSSS) inni TDMA slots. Denne endringen krever et nytt 3G-mobilnettverk som fungerer parallelt med 2G BSS-radionettverket vist i figur 6.19.

## On to 4G: LTE

Med 3G systemene kjørende over hele verden, kommer 4G-systemene. **4G Long-Term Evolution (LTE)** standarden lagt fram av 3GPP har to viktige innovasjoner over 3G systemer:

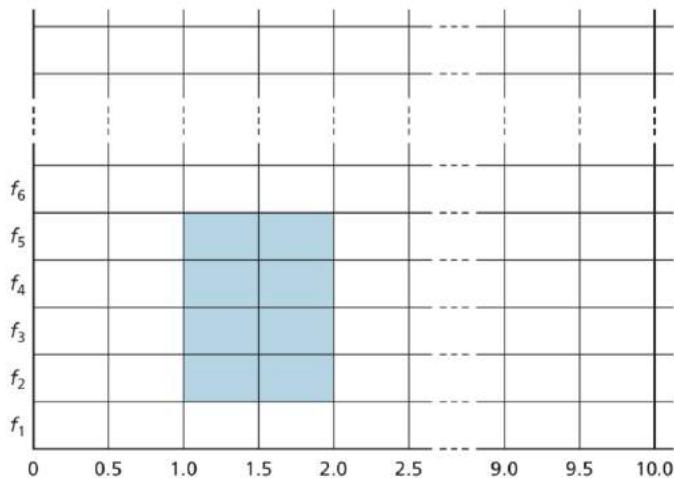
- **Evolved Packet Core (EPC).** EPC-en er forenklet all-IP kjernenettverk som forener det separate kretsomkoblede voice-telefonnettverket og det pakkedrevne mobilnettverket vist i figur 6.19. Det er et "all-IP-nettverk" ved at både tale og data vil bli levert i IP-datagrammer. Som vi har sett i Kap. 4 og senere i Kap. 7 er IPs "best effort" tjenestemodell ikke tilstrekkelig for de strenge ytelseskravene for *Voice-over-IP-trafikk (VoIP)* med mindre nettverksressurser klarer å unngå (istedet for å reagere på) overbelastning.

Derfor er en sentral oppgave for EPC å administrere/styre nettverksressurser til å levere denne høye kvaliteten på tjenesten. EPC gjør også en klar separasjon mellom nettverkskontroll- og brukerdataplanene, med mange av mobilitetsstøttefunksjonene. EPC tillater flere typer radio-tilgangsnettverk, inkludert eldre 2G- og 3G-radionettverk, for å knytte seg til kjernenettverket.

- **LTE Radio Access Network.** LTE bruker en kombinasjon av frekvensdelings-multipleksing og tidsdelings-multipleksing på nedstrømskanalen, kjent som ortogonal frekvensdelings-multipleksing (OFDM). I LTE er hver aktive mobile node allokeret en eller flere 0.5 ms tidsslots i en eller flere kanalfrekvenser. Ved å bli allokeret økende flere tidsslots (enten på samme eller forskjellige frekvenser) kan den mobile noden være i stand til å oppnå høyere overføringshastigheter.

Slot (re)allokering blant mobile noder kan bli gjort så ofte som hver millisekund. Forskjellige moduleringsskjemaer kan også bli brukt for å øke overføringshastigheten. LTE radionettverk bruker multiple-input, multiple output (MIMO) antenner. Den maksimale dataraten for en LTE-bruker er 100 Mbps nedstrøms og 50 Mbps oppstrøms, når man bruker 20 MHz båndbredde.

Den spesielle allokeringen av tidsluker til mobile noder er ikke bestemt av LTE-standarden. I stedet vil avgjørelsen av hvilke mobile noder vil få lov til å overføre i et gitt tidsluke på en gitt frekvens bestemmes av *schedule algorithm* som tilbys av LTE-utstyrleverandøren og/eller nettverksoperatøren. I tillegg kan brukerprioriteringer og kontrakte servisenivåer (for eksempel sølv, gull eller platina) brukes til å planlegge nedstrøms pakkeoverføringer. I tillegg til LTE-funksjonene beskrevet ovenfor, tillater LTE-Advanced nedstrømsbåndbredde på hundrevis av Mbps ved å allokerere aggregerte kanaler til en mobil node. En annen 4G trådløs teknologi er WiMax, som er fra Japan.



**Figure 6.20** • Twenty 0.5 ms slots organized into 10 ms frames at each frequency. An eight-slot allocation is shown shaded.

## Kapittel 7 - Multimedia Networking

Vi kan starte kapittelet med litt systematikken til multimediasplikasjoner nå i første avsnitt. Vi skal se at multimediasplikasjoner kan bli klassifisert som enten *streaming stored audio/video, conversational voice/video-over-IP*, eller *streaming live audio/video*. Vi skal se at hver av disse applikasjonsklassene har sine egne unike servicekrav som avviker vesentlig fra tradisjonelle elastiske applikasjoner som e-post, nettlesing og ekstern pålogging. I avsnitt 7.2, undersøker vi video streaming i detalj.

## Multimedia Networking Applications

Vi definerer en multmedianettverksapplikasjon som enhver nettverksapplikasjon som tilbyr lyd eller video.

### Properties of Video

Den kanskje viktigste egenskapen til video er dens **høye bit rate**. Videodistribusjon over internettet er typisk på 100kbps for lavkvalitetsvideokonferanse til over 3 Mbps for å stream høykvalitetsfilmer.

For å skjonne hvordan video-båndbreddekrav er i forhold til andre forskjellige Internettapplikasjoner. La oss anta at Frank er på Facebook og ser på et nytt bilde hvert 10 sekund, og at hvert bilde er 200 Kbytes (antar for enkelhetsskyld at 1Kbyte = 8000 bits). Antar også at Martha har hørt på mange MP3 sanger, med en rate på 128 kbps, og Victor som ser på video som har blitt kodet med en rate på 2 Mbps. I tabell 7.1 sammenliknes disse tre handlingene, der vi antar at hver session har vart i 67 minutter.

|                       | Bit rate | Bytes transferred in 67 min |
|-----------------------|----------|-----------------------------|
| <b>Facebook Frank</b> | 160 kbps | 80 Mbytes                   |
| <b>Martha Music</b>   | 128 kbps | 64 Mbytes                   |
| <b>Victor Video</b>   | 2 Mbps   | 1 Gbyte                     |

**Table 7.1** • Comparison of bitrate requirements of three Internet applications

Vi ser at videostreaming bruker desidert mest båndbredde, med en bitrate på mer enn ti ganger mer enn musikkapplikasjonen. Gitt populærheten til video og dens høye bitrate er det ikke rart at Cisco tipper at streaming og lagring av video vil ta bruke 90% av den globale internetttrafikken.

En annen viktig egenskap til video er at den kan bli komprimert, og dermed bytte kvalitet med bitrate. En video er en sekvens av bilder, og blir vist med en konstant rate, f.eks. 24 eller 30 bilder i sekundet. En ukomprimert, digitalt kodet bilde består av en liste med piksler. Det er to typer av redundans i video, som kan bli utnyttet av **video komprimering**.

- *Spatial redundancy* (nor. romlig redundans) er redundansen innenfor et gitt bilde. Intuitivt, et bilde som for det meste består av hvite plasser har en høy grad av redundans og kan komprimeres effektivt uten å gi en vesentlig oppussing av bildekvaliteten.
- *Temporal redundancy* (nor. tidsredudans) reflekterer repitisjon fra et bilde til det etterfølgende bilde. Dersom et bilde og det etterfølgende bildet er identiske, er det ikke noe poeng å re-kode det etterfølgende bildet. Da er det mer effektivt å indikere i encodingen at det etterfølgende bildet er det samme.

Vi kan også bruke kompresjon for å lage *flere versjoner*\* av samme video, alle på forskjellige kvalitetsnivåer. For eksempel kan vi bruke kompresjon for å lage tre versjoner av samme video, med ratene 300 kbps, 1 Mbps og 3 Mbps, så kan man la brukere bestemme hvilken versjon de vil se med den tilgjengelige båndbredden.

### Properties of Audio

Digital lyd har veldig mye lavere båndbreddekrav enn video. Digital lyd har sine egne unike egenskaper som må bli tatt hensyn til når man designer multedianettverksapplikasjoner. For å forstå disse egenskapene, la oss se på hvordan man analog lyd til digital lyd:

- Det ananloge lydsignalet blir samplet på en gitt rate, f.eks. 8,000 samples per sekund. Verdien til hver sample er et vilkårlig reellt tall.
- Hver av disse samplene blir avrudet til et endelig antall av verdier. Denne operasjonen kalles **kvantisering**. Tallet for slike endelige verdier, kalt kvantiseringss verdier, er typisk en opphøyning av to, for eksempel 256 kvantiseringss verdier.
- Hver av disse kvantiseringss verdier blir representert med et bestemt antall bits. For eksempel dersom det er 256 kvantiseringss verdier, da er hver verdi - og dermed hver sample - representert av en byte. Bitrepresentasjonene av alle samplene blir så sammenkoblet for å danne den digitale representasjonen av signalet. Som et eksempel, hvis et analogt lydsignal samples ved 8000 prøver per sekund, og hver prøve blir kvantisert og representert med 8 bits, vil det resulterende digitale signalet ha en hastighet på 64.000 bits per sekund. Ved avspilling kan lyden dekodes tilbake til analogt signal.

Den vanlige enkodingsteknikken som akkurat ble beskrevet kalles **pulse code modulation (PCM)**. Stemmeenkoding bruker ofte PCM, med en samplingrate på 8,000 per sekund og 8 bit per sample, som gir en rate på 64kbps. CD bruker også PCM, med en samplingsrate på 44,100 samples i sekundet med 16 bits per sample, som gir 705,6 kbps for mono og 1.411 Mbps for stereo.

PCM brukes sjeldent på internettet. En vanlig kompresjonsteknikk for nær CD-kvalitets steromeusikk er **MPEG 1 layer 3** eller også kjent som **MP3**. MP3 enkodere kan komprimere til mange forskjellige rater; 128 kbps er den vanligste og produserer liten lyddegradering. En relatert standart er **Advanced Audio Coding (AAC)** som er laget av Apple.

### Types of Multimedia Network Applications

Vi deler multimedia-applikasjoner inn i tre kategorier: *(i) streaming stored audio/video, (ii) conversational voice/video-over-IP og (iii) streaming live audio/video.*

#### Streaming Stored Audio and Video

Streaming lagret lyd (Spotify) er veldig lik streaming lagret video, selvom bitratene er typisk mye mindre. I denne klassen av applikasjoner er det underliggende mediumet forhåndsinnsplitt video, slik som en film, tv-serie osv. Disse forhåndsinnsplittede videoene blir plassert på servere, og brukere sender requests til serverene for å se videoer *on demand*.

Streaming stored video har tre viktige egenskaper:

- *Streaming*. I en streaming stored videoapplikasjon, vil klienten typisk begynne å spille videoen et par sekunder etter at den har begynt å motta video fra serveren. Dette betyr at klienten vil spille av fra et sted i videoen samtidig som den mottar senere deler fra serveren. Denne teknikken kalt **streaming** unngår å måtte laste ned hele videofilmen før man begynner å spille av.
- *Interactivity*. Fordi mediet er forhåndsinnsplitt, kan brukeren pause, reposisjonere, spole, øke hastighet på videoinnholdet.
- *Continous playout*. Når avspilling av videoen begynner, bør den fortsette i henhold til opprinnelig timing av opptaket. Derfor må data mottas fra serveren i tide for utspillingen hos klienten. Ellers opplever brukerne frysning av videoklipp (når klienten venter på de forsinkede rammene) eller rammen hopper over (når klienten hopper over forsinkede rammer).

Den aller viktigste ytelsesmålingen for streaming av video er gjennomsnittlig gjennomstrømning.

#### Conversational Voice- and Video-over-IP

Sanntidssamtaler over Internett blir ofte referert til som **Internett-telefoni**, siden det er, fra brukerens perspektiv, det som ligner på den tradisjonelle *kretskoblede telefontjenesten*. Det kalles også **Voice-over-IP (VoIP)**. Samtalevideo er likt, bortsett fra at den inkluderer video av deltakerene i tillegg til stemmene deres. I Kap. 2 så vi på et antall akser som applikasjonskrav kan bli klassifisert av. To av disse aksene - timinghensyn og toleranse av datatap - er veldig viktige for samtalelyd og -video.

- Timinghensyn er viktig fordi lyd- og videosamtaleapplikasjoner er veldig **forsinkelses-sensitive**. Dersom forsinkelsen på en samtale i en slik applikasjon blir for stor blir det nærmest umulig å holde en samtale.
- På den andre siden er samtalemultimediaapplikasjoner **taps-tolerante** (eng. *loss-tolerant*) - tap av og til fører til glitches i lyd/video-avstillingen, og disse tapene kan ofte bli delvis eller helt gjemt. Denne forsinkelsessensitiveten og tapstoleransen karakteristikkene er helt klart annerledes enn de elastiske data-applikasjonene slik som nettsurfing, epost og lignende.

#### Streaming Live Audio and Video

Denne tredje klassen av applikasjoner ligner på tradisjonell kringkastingsradio og fjernsyn, bortsett fra at overføring foregår over Internett. Disse programmene tillater en bruker å motta en *live* radio eller tv-overføring, for eksempel en sanntids sportsbegivenhet eller en pågående nyhetshendelse som sendes fra et hvilket som helst hjørne av verden. I dag sender tusenvis av radio- og fjernsynsstasjoner over hele verden kringkasting av innhold over Internett.

Multicast-distribusjonen blir som oftest utført i dag via applikasjonslags multicast (ved hjelp av P2P-nettverk eller CDN) eller gjennom flere separate unicast strømmer. Siden et event er live, kan vi tillate noe forsinkelser fra hendelsen til en bruker spiller den av på skjermen, rundt 10 sekunder vil være tilatt.

#### Streaming Stored Video

Streaming-videosystemer kan bli klassifisert i tre kategorier: **UDP streaming**, **HTTP streaming** og **adaptiv HTTP streaming**. Selvom alle disse brukes i praksis, bruker majoriteten av dagens systemer *HTTP streaming* og *adaptiv HTTP streaming*.

En felles egenskap av disse tre formene er den omfattende bruken av applikasjonsbuffer på llientsiden for å redusere effektene av varierende ende-til-ende forsinkelser og varierende mengder tilgjengelig båndbredde mellom server og klient.

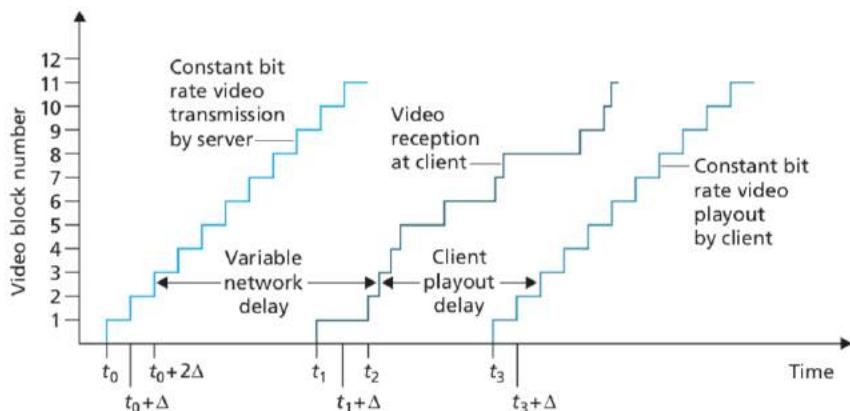
For streaming vide (lagret og live) kan brukerne vanligvis tolererer en liten oppsummering på flere sekunder mellom klienten ber om en video, til klienten spiller den av.

Derfor, når videoen kommer til klienten, behøver klienten ikke umiddelbart å starte videoen, men kan istedet bygge opp en reserve av video i en applikasjonsbuffer. Når klienten har bygd opp en reserve på flere sekunder med buffret-men-ikke-avspilt-video, kan klienten spille av videoen.

Det er to viktige fordeler som tilbys av en slik **klientbuffering**. For det første kan klientsidebuffering absorbere variasjoner i server-til-klientforsinkelse. Hvis et bestemt stykke videodata forsinkes, så lenge det kommer før reserven av mottatt-men-ikke-spilt video er oppbrukt, vil denne lange forsinkelsen ikke bli lagt merke til. For det andre, hvis server-til-klient båndbredde kort faller under videokonsentrasjonshastigheten, kan en bruker fortsette å nytte kontinuerlig avspilling, igjen så lenge klientprogrambufferen ikke blir helt tom.

Figur 7.1 under illustrerer klient-side buffering. I dette eksempelet er videoen kodet i en viss bitrate, og dermed skal hver videoblokk med videorammer spilles over samme tidsrom,  $\Delta$ . Serveren sender første videoblokk på  $t_0$ , andre blokk på  $t_0 + \Delta$ , og tredje blokk på  $t_0 + 2\Delta$ , osv.

Når videoen skal avspilles må alle hver videoblokk bli avspilt i tidsrommet  $\Delta$ . Dersom klienten skulle ha spilt av videoen med en gang første blokk ankom, ville ikke klienten rukket å få blokk 2 før blokk 1 var ferdig. Istedet venter klienten til  $t_3$  med å starte å spille av.



**Figure 7.1** ♦ Client playout delay in video streaming

## UDP Streaming

Vi skal kjapt se på UDP-streaming. Med UDP-streaming overfører serveren video med en hastighet som samsvarer med klientens video-consumption rate ved å klokke ut videoens klumper over UDP i en jevn hastighet.

For eksempel, hvis videoforbruket er 2 Mbps og hver UDP-pakke bærer 8 000 bits av video, vil serveren sende en UDP-pakke inn i socketen hvert  $(8000 \text{ bits}) / (2 \text{ Mbps}) = 4 \text{ msec}$ . Som vi lærte i kapittel 3, fordi UDP ikke bruker en overbelastningsstyringsmekanisme, kan serveren skyve pakker inn i nettverket ved forbrukstakten av videoen uten ratekontrollsbegrensningene i TCP. UDP-streaming bruker vanligvis en liten klient-sidebuffer, stor nok til å holde mindre enn et sekund med video.

Før du overfører videoblokene til UDP, vil serveren inkapslere videobrikkene i transportpakker som er spesielt utviklet for å transportere lyd og video, ved hjelp av *Real-Time Transport Protocol (RTP)* eller en lignende ordning.

- En annen kjennetegn ved UDP-streaming er at i tillegg til server-til-klient videostrømmen, opprettholder klienten og serveren i parallel også en separat kontrollforbindelse der klienten sender kommandoer om tilstandssendringer (for eksempel pause, play, reposisjon, og så videre).

Har flere negative trekk:

- For det første, på grunn av den uforutsigbare og varierende mengden tilgjengelig båndbredde mellom server og klient, kan UDP-streaming med konstant hastighet ikke gi kontinuerlig avspilling. For eksempel, vurder scenariet der videokonsentrasjonshastigheten er 1 Mbps, og den tilgjengelige båndbredden til server-til-klienten er vanligvis mer enn 1 Mbps, men noen få minutter faller den tilgjengelige båndbredden under 1 Mbps i flere sekunder. I et slikt scenario vil et UDP-streaming-system som overfører video med en konstant hastighet på 1 Mbps over RTP / UDP, sannsynligvis gi en dårlig brukeropplevelse, med frysing eller hoppet over rammer snart etter at tilgjengelig båndbredde faller under 1 Mbps.
- Den andre ulykken ved UDP-streaming er at det krever en media-kontrollserver, for eksempel en RTSP-server, for å prosessere interaktivitetsforspørslar fra klient til server og å spore klienttilstand (f.eks. klientens avspillingspunkt i videoen, om videoen blir pauset eller spilt, og så videre) for hver pågående klientsøkt. Dette øker den totale kostnaden og kompleksiteten i å distribuere et stort videoprosesssystem.
- Den tredje ulykken er at mange brannmurer er konfigurert til å blokkere UDP-trafikk, slik at brukerne bak disse brannmurene ikke mottar UDP-video.

## HTTP Streaming

I HTTP streaming, så blir videoen bare lagret på en HTTP-server som en ordinær fil med en spesifikk URL. Når en bruker ønsker å se videoen, må klienten etablere en TCP-tilkobling til serveren, gjøre en HTTP GET-request for den URL-en. Serveren sender så videofilen i en HTTP Response-melding, så fort som mulig, dvs. så fort som TCP congestion control og flow control tillater.

P klientsiden blir bytene fanget i klientapplikasjonens buffer. Når et bestemt antall bytes er ankommet, kan klienten spille av videoen. Klienten henter periodeisk videoramme fra bufferen, dekomprimerer rammene, og viser dem på brukerens skjerm.

Som vi lærte i Kap. 3 kan overføring av filer over TCP gi stor variasjon på server-til-klient-overføringsrate, pga TCP congestion control og flow control. Derfor er det ikke uvanlig at overføringsraten varierer på en saw-tooth måte (opp-og-ned). I tillegg kan pakker bli veldig forsinket pga TCPs re-sendingsmekanisme. Dette kan unngås med klientbufring og prefetching (skal se på i neste avsnitt).

Bruken av HTTP over TCP tillater videoen å traversere brannmurer og NATs mye lettere. Streaming over HTTP har heller ikke bruk for en mediakontroll-server, slik som en RTSP, som gjør det billigere. På grunn av disse fordelene bruker de fleste video-strømmetjenestene idag, inkludert YouTube og Netflix, HTTP streaming over TCP som sin underliggende strømmeprotokoll.

### Prefetching Video

For streaming av *lagret/stored* video, kan klienten forsøke å laste ned videoen med en høyere rate enn forbruksraten, og dermed **prefetching** (nor. forhåndshenter) videoramme som skal bli brukt i fremtiden. Denne prefetchede videoen er naturligvis lagret i klientapplikasjonens buffer.

Slik prefetching forekommer naturlig med TCP streaming, siden TCPs congestion avoidance vil forsøke å bruke all tilgjengelig båndbredde mellom server og klient.

For å få litt innsikt i prefetching, la oss ta en titt på et enkelt eksempel. Anta at *videoforbruket* er 1 Mbps, men nettverket kan levere videoen fra server til klient med en konstant hastighet på 1,5 Mbps. Da vil klienten ikke bare kunne spille av videoen med en veldig liten spillforsinkelse, men vil også kunne øke *mengden buffede videodata med 500 Kbits hvert sekund*. På denne måten, hvis klienten i fremtiden mottar data med en hastighet på mindre enn 1 Mbps i en kort tidsperiode, vil klienten kunne fortsette å gi kontinuerlig avspilling på grunn av reserver i bufferen.

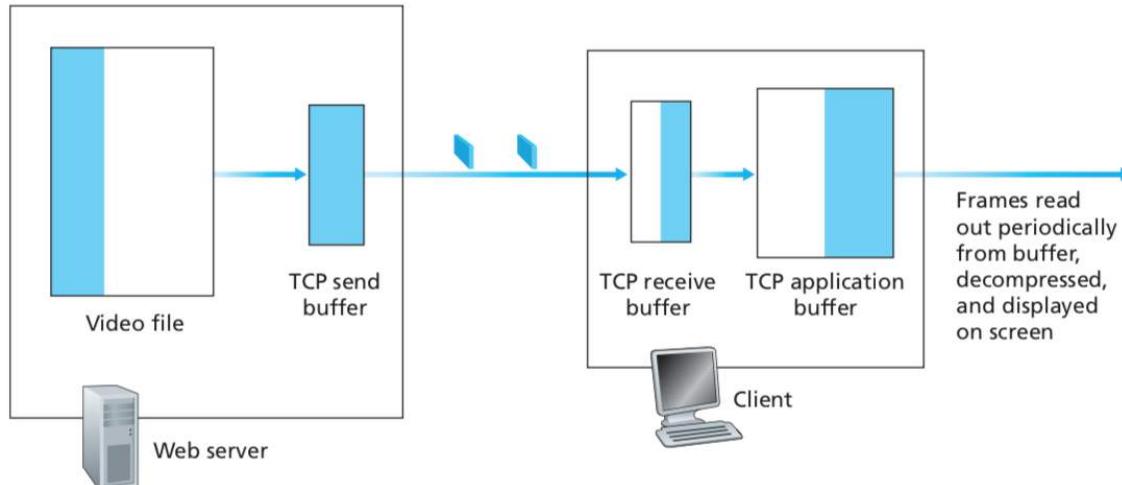
### Client Application Buffer and TCP

Figur 7.2 illustrerer interaksjonen mellom klient og server for HTTP-streaming. På serversiden, vil delen av videofil som er hvit allerede ha blitt sendt til server-socketen, mens den blå delen er det som gjenstår å bli sendt. Etter å ha blitt sendt gjennom "socket døren", vil bytene bli plassert i TCP-sendbufferen før det blir sendt til internettet.

På klientsiden, leser klientapplikasjonen (media player) lese bytes fra TCP-mottakerbufferen (gjennom sin klientsocket), og plassere bytene i klientapplikasjonens buffer. Samtidig henter klientapplikasjonen periodisk videoramme fra klientapplikasjonsbufferen, dekomprimerer rammene og viser dem på brukerens skjerm.

Dersom applikasjonsbufferen er større enn filstørrelsen vil det være som å laste ned en fil over HTTP - klienten henter kun filen så fort som mulig fra serveren. Men hva vil skje dersom brukeren pauser videoen under streamingprosessen. Under pauseperioden, blir ikke bits fjernet fra bufferen, og etterhvert vil bufferen bli full, dersom bufferens størrelse er mindre enn filstørrelsen. Når applikasjonsbufferen blir full, vil ikke bytes kunne flyttes fra TCP-mottakerbufferen. Når klientens TCP-mottakerbuffer blir full, kan ikke bytes lenger fjernes fra serverens TCP-sendebuffer, slik at den også blir full. Når TCP-sendebufferen blir full, kan serveren ikke sende flere byte til kontakten. Hvis brukeren midlertidig stopper videoen, kan serveren derfor bli tvunget til å slutte å overføre, i så fall vil serveren bli blokkert til brukeren gjenopptar videoen.

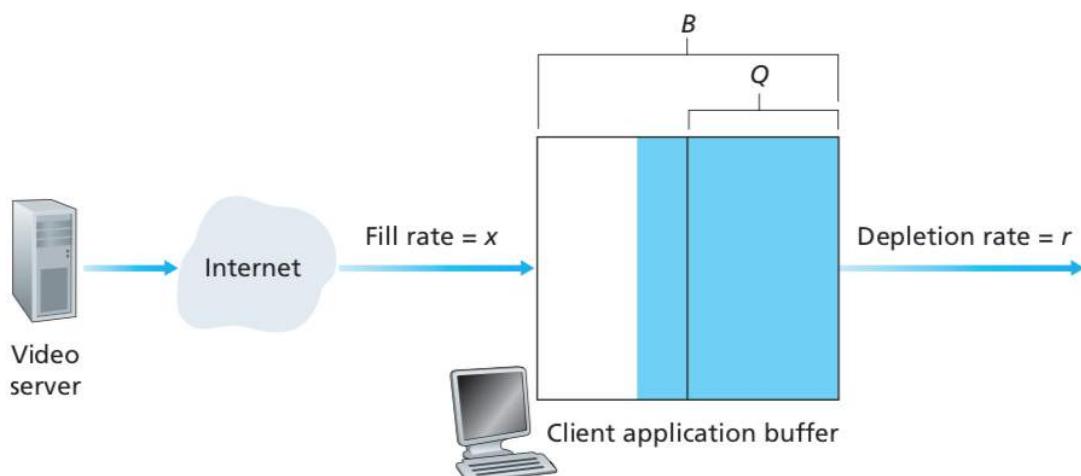
Dersom man har en full buffer, vil ikke serveren kunne sende kjappere enn klientens videoforbruksrate. Derfor pålegger en full klientprogrambuffer indirekte en grense på frekvensen som video kan sendes fra server til klient når den streames over HTTP.



**Figure 7.2** ♦ Streaming stored video over HTTP/TCP

#### Analysis of Video Streaming

Enkelte enkle modeller vil gi mer innsikt i innledende avspillingsforsinkelse og frysing på grunn av applikasjonsbufferuttømming. Som vist i figur 7.3, la  $B$  betegne størrelsen (i biter) av klientens applikasjonsbuffer, og la  $Q$  angi antall biter som må buffes før klientprogrammet starter avspilling. (Selvfølgelig,  $Q < B$ .) La  $r$  angi videoforbrukshastigheten - hvor raskt klienten trekker biter ut av klientprogrambufferen under avspilling. Så, for eksempel hvis videoens rammeoppsætning er 30 bilder/sek, og hver (komprimerte) ramme er 100,000 biter, deretter  $r = 3$  Mbps.



**Figure 7.3** ♦ Analysis of client-side buffering for video streaming

La oss anta at serveren sender bits med konstant hastighet  $x$  når klientens buffer ikke er full. Anta at applikasjonens buffer er tom ved  $t = 0$ . Hva er tiden  $t = t_p$  når avspillingen begynner? Hva er tiden  $t = t_f$  når klient-applikasjonsbufferen blir full?

- La oss først bestemme  $t_p$ , det er tiden når  $Q$  bits har kommet i applikasjonsbufferen. Det ankommer  $x$  bps. Det betyr at den initielle bufringsforsinkelsen  $t_p = Q/x$ .
- La oss nå bestemme  $t_f$ , når klientapplikasjonsbufferen blir full. Vi observerer at dersom  $x < r$  vil bufferen aldri bli full, nærmere bestemt vil den bli tom. Dersom  $x > r$ , vil bufferen kunne bli full, da det ankommer flere bits enn det klientapplikasjonen fjerner fra bufferen. Da vil bufferen fylles fra  $Q$  til  $B$  med en rate på  $x - r$  bits. Det gir formelen:  $t_f = Q/x + (B-Q)/(x-r)$ .

### Early Termination and Reposition the Video

HTTP streamingsystemer bruker ofte HTTP-byte-range headeren i HTTP GET-meldingen, som spesifiserer den spesifikke utvalget av bytes kunden ønsker å få fra den ønskede videoen. Dette er spesielt viktig når brukeren ønsker å reposisjonere seg, dvs. hoppe, til et senere punkt i videoen. Da må brukeren indikere den nye posisjonen, ved å sende en ny HTTP request.

Dersom en bruker ønsker å avslutte å se på en video tidlig vil det være prefetched-men-ikke-sett data overført av serveren - som vil være bortkastede nettverksbåndbredde og serverressursser. Dersom man spoler forbi eller terminerer videoen tidlig, vil man forkaste allerede nedlastet video, som er dette bortkastet bruk av båndbredde og ressursser, og derfor har mange applikasjonsbuffere begresende buffere, for bespare seg på slike bortkastede rammer.

### Adaptiv Streaming and DASH

Selvom HTTP streaming, beskrevet som over er brukt mye i praksis, har den en stor mangel. Alle klienter mottar samme koding av videoen, selvom det kan være store variasjoner på tilgjengelig båndbredde for klienten, både på tvers av forskjellige klienter, men også over tid for samme klient. Dette har ledet til utviklingen av en ny type HTTP-basert streaming kalt **Dynamic Adaptive Streaming over HTTP (DASH)**. I DASH blir video kodet til forskjellige versjoner, der hver versjon har forskjellige bitrater og dermed kvalitetsnivåer. Klienten velger dynamisk deler av videosegmenter på noen få sekunder fra forskjellige versjoner. Når det mengden tilgjengelig båndbredde er høy, velger klienten høybit-rate versjonen, og når det er lite tilgjengelig båndbredde velger klienten dårligere kvalitet.

Tillater brukere med forskjellige internettaksessrater å se streamme videoer med forskjellige koderater. Med DASH er hver videoversjon lagret i HTTP-serveren, hver med en forskjellig URL. HTTP-serveren har også en **manifest-fil**, som oppgir URL-en for hver av versjonene sammen med dems bitrate. Klienten ber først om manifestfilen og lærer om de forskjellige versjonene. Klienten velger så en chuck med videodata fra en versjon, regner ut fått båndbredde som kjører en *rate determination algorithm* som velger neste chuck å forespørre.

Ved dynamisk oppsyn på tilgjengelig båndbredde, klientbuffernivå, og endre på overføringsraten med versjonsvitsjing, oppnår som oftest DASH kontinuerlig avspilling med beste kvalitetsnivå tilgjengelig. Det lagres også separat mange forskjellige versjoner av lyden, som også har sin egen URL.

- Med disse implementasjonene kan brukeren velge chuncks med både videodata og lyddata i forskjellige kvaliteter.

### Content Distribution Networks

For et internettvideoselskap er kanskje den mest "rett frem"-løsningen for å tilby videotransport er å kjøpe et enkelt massivt datasenter, lagre alle dens videoer i datasenteret og strømme videoer direkte fra datasenteret til klienter over hele verden. Men det er tre store problemer med denne fremgangsmåten.

- Først dersom en klient er langt unna datasenteret, vil server-til-klient-pakker krysse mange kommunikasjonskoblinger og vil nok gå igjennom mange ISP-er, med noen ISP-er på forskjellige kontinenter. Dersom en av disse koblingene tilbyr en gjennomstrømning som er lavere enn videoforbrukerraten, vil ende-til-ende gjennomstrømningen være mindre enn forbrukerraten, og vil resultere i irriterende fryseforsinkelser for brukeren. Sannsyneligheten for noe sånt øker med antall koblinger fra server til klient.
- Den andre negative egenaskapen er at en populær video som gjerne vil bli sendt mange ganger over samme kommunikasjonslink. Ikke bare vil det være bortkastet båndbredde, vil også internettvideoselskapet måtte betale ISP-en for å sende de samme bytesene igjen og igjen.
- Et tredje problem er at denne løsningen er at et enkelt datasenter representerer et enkelt punkt for feil - dersom datasenteret eller koblingene går ned, vil det ikke være mulig å distribuere noen videostrømmer.

For å ta hånd om dette problemet bruker de aller fleste store videostrømmetjenester **Content Distribution Networks (CDNs)**. En CDN passer på servere på flere geografiske distribuerte områder, lagrer kopier av videoer (og andre webinnhold) i sine servere, og forsøker å sende hver brukerforespørsel til en CDN-lokasjon som vil tilby den beste brukeropplevelsen. Det kan være en **privat CDN**, som er eid av selskapet som distribuerer selv, eller en **third-party CDN**, som distribuerer innhold på vegne av flere content-providers.

CDNs har typisk en av to forskjellige server plasseringsfilosofier:

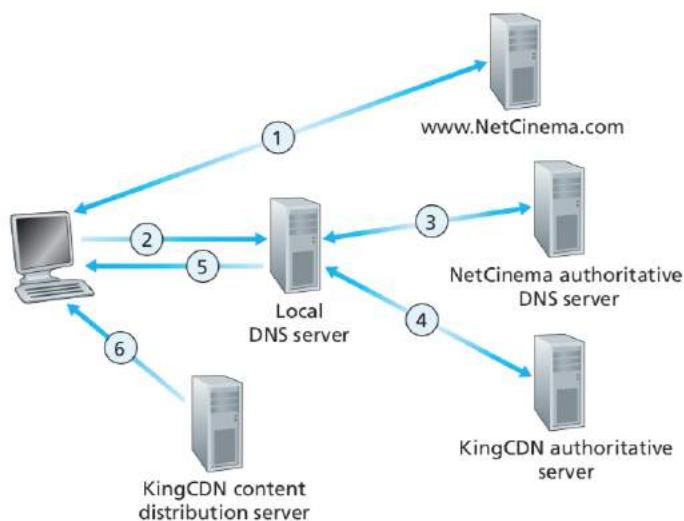
- **Enter Deep.** En av filosofiene er på *enter deep* inn i aksessnettverkene til ISP-ene, ved å utplassere servercluster i aksess ISP-er over hele verden. Målet for denne fremgangsmåten er å komme nærmere endebrukerne, derfor ved å forbedre bruker-oppfattet forsinkelse og gjennomstrømning, ved å redusere antall koblinger mellom brukeren og CDN-clusteret der den henter data. På grunn av dette distribuerte designet, vil oppgaven om å styre og vedlikeholde clusterne vanskelig.
- **Bring Home.** En annen designfilosofi er å *bring the ISPs home*, ved å bygge store clusters med et mindre antall (f.eks. titall) av nøkkellokasjoner og koble sammen disse clusterne med et privat high-speed-nettverk. I Stedet for å gå inn i aksess ISP-ene, vil disse CDNene typisk plassere hver cluster på en lokasjon som er nær PoPs (Points of Presence) av mange tier-1 ISP-er. Sammenliknet med enter deep-designmetoden vil denne bring-home designet resultere i mindre vedlikehold og administrering, men på bekostning av høyere forsinkelse og lavere gjennomstrømning til brukerne.

Når clusterne er på plass, vil CDN-en replikere innhold på dens clustere. CDN-en ønsker nødvendigvis ikke å kopiere alle videoer i hvert cluster, siden noen er sjeldent sett, og er populære kun i noen land. Når en bruker ber om en video fra et cluster som ikke har videoen, henter clusteret videoen fra sentralrepoet eller et annet cluster, og lagrer samtidig en kopi av videoen samtidig som den streamer videoen til klienten. Når clusteret senere blir fult fjerner den videoen som ikke blir forespurt ofte.

#### ##### CDN Operation

Når har vi sett på de to største måtene å utplassere en CDN. La oss nå se på hvordan en CDN opererer. Når en nettleser blir bedt om å hente en spesifikk video (bestemt av URL, må CDN-en ta fange opp requesten, slik at den kan (1) bestemme et passende CDN servercluster for klienten på det tidspunktet, og (2) omdirigere klientens forespørsel til en server i det clusteret.

De fleste CDN-er tar fordel av DNS for å fange og omdirigere requests. La oss se på et enkelt eksempel på hvordan DNS typisk er involvert. La oss se for oss en content-provider NetCinema, bruker en tredjeparts CDN-selskap KingCDN, for distribuere sine videoer til kunder. La oss nå se på de 6 stegene i Figur 7.4 under:



**Figure 7.4** ♦ DNS redirects a user's request to a CDN server

1. Bruker besøker en nettside hos NetCinema
2. Brukeren trykker seg inn på linken <http://video.netcinema.com/6Y7B23V>, brukerens vert sender en DNS-spørring for video.netcinema.com.
3. Brukerens lokale DNS-server (LDNS) videresender DNS-spørringen til en autorativ DNS-server, som ser strengen "video" i vertsnavnet. For å "overlate" DNS-spørringen til KingCDN, i stedet for å returnere en IP-adresse, returnerer den autoriserte DNS-tjeneren NetCinema til LDNS et vertsnavn i KingCDNs domene, for eksempel a1105.kingcdn.com.

4. Herfra går DNS-spørringen inn i KingCDNs private DNS-infrastruktur. Brukerens LDNS sender deretter en ny spørring, nå for a1105.kingcdn.com, og KingCDNs DNS-system returnerer til slutt IP-adressene til en KingCDN innholdsserver til LDNS. Det er således her, innenfor KingCDNs DNS-system, at CDN-serveren som kunden vil motta innholdet til, er spesifisert.
5. LDNS-en videresender IP-adressen til den innholdsleverende CDN-noden til verten.
6. Når brukeren får IP-adressen til KingCDN sin innholdsserver, lager den en direkte TCP-tilkobling med serveren på den IP-adressen og gjør en HTTP GET-request på videoen.

#### Cluster Selection Strategies

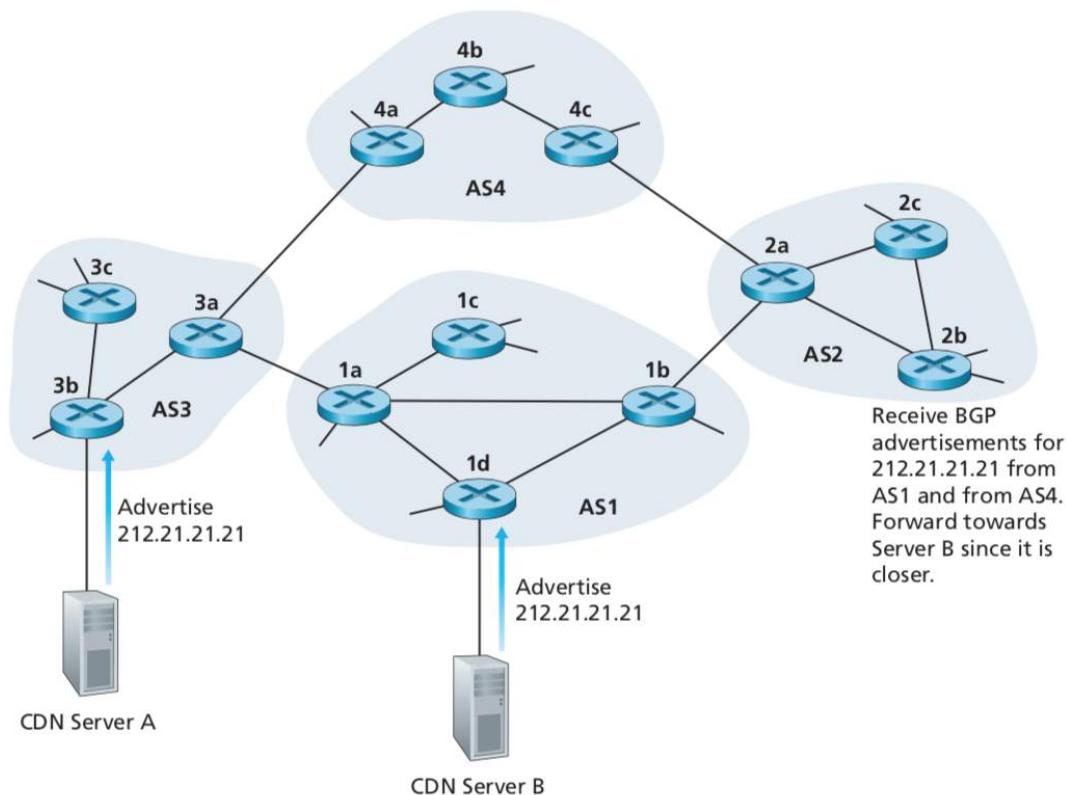
Ved kjernen av enhver CDN utplassering er en **cluster selection strategy**, en mekanisme for å dynamisk omdirigere klienter til et servercluster eller et datasenter inni CDN-en.

En enkel strategi er å tilordne klienten til klyngen som er **nærmest geografisk**. Ved å bruke kommersielle geografiske databaser (som Quova og Max-Mind), blir hver LDNS IP-adresse kartlagt til en geografisk plassering.

For å kunne bestemme det beste clusteret for en klient basert på de nåværende trafikkforholdene, kan CDN i stedet utføre periodiske **samtidsmålinger** av forsinkelse og taptøyte mellom clustere og klienter. For eksempel kan en CDN ha hver av sine clustere periodisk sende probes (for eksempel pingmeldinger eller DNS-spørringer) til alle LDNSene rundt om i verden. En ulempe ved denne tilnærmingen er at mange LDNSer er konfigurert til ikke å svare på slike probes.

Kan se på mellrommet mellom SYNACK og ACK-en i handshaken mellom klient og server, istedet for å sende ekstra trafikk for å undersøke forholdene. Dette krever dog å omdirigere klienten til andre clustere med tiden for å se hvor det er best.

En helt annen tilnærming for å matche klienter med CDN-servere er å bruke **IP anycast**. Ideen bak IP anycast er å få ruterne i Internettet til å føre klientens pakker til det "nærmeste" clusteret, som bestemt av BGP. Spesifikt, som vist i figur 7.5, i løpet av konfigurasjonen av IP-anycast, tilordner CDN-selskapet samme IP-adresse til hver av sine klynger, og bruker *standard BGP* for å annonse denne IP-adressen fra hver av de forskjellige clusterlokasjonene. Når en BGP-rute mottar flere ruteannonser for samme IP-adresse, behandler den disse annonsene som å gi forskjellige veier til samme fysiske plassering (når annonsene faktisk er for forskjellige veier til *forskjellige* fysiske steder). Etter standard operasjonsprosedyrer vil BGP-ruten velge den "beste" (for eksempel nærmeste, bestemt av AS-hop-teller) ruten til IP-adressen i henhold til den lokale rutevalgsmekanismen.



**Figure 7.5** ♦ Using IP anycast to route clients to closest CDN cluster

## Case Studies: Netflix, YouTube and Kankan

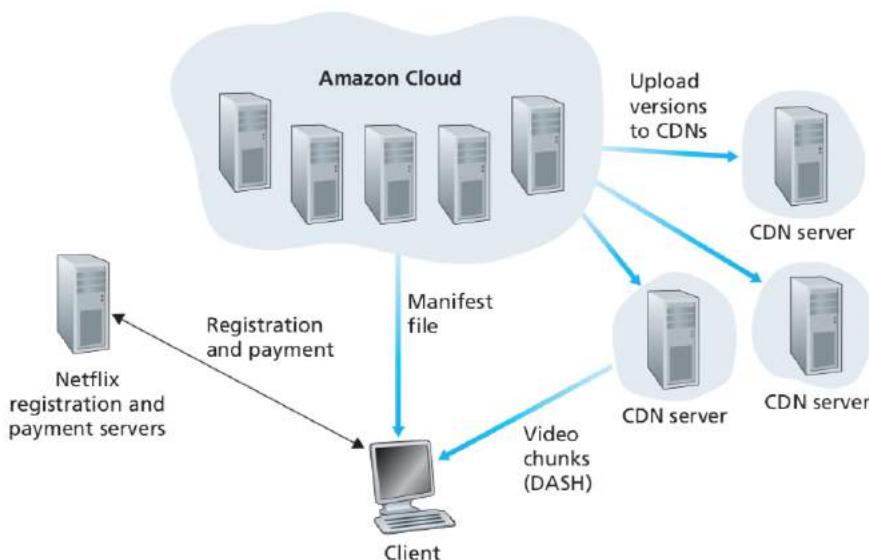
### Netflix

Genererer idag rundt 80% av all nedlastingstrafikk i USA. Som vi skal se bruker Netflix mange teknikker som vi har sett i dette kapitlet, inkludert videodistribuering med CDN (flere CDN-er faktisk) og adaptiv streaming over HTTP.

Figur 7.6 viser den generelle arkitekturen til Netflix-plattformen. Den har fire hovedkomponenter: registrering og betalingstjenester, Amazonskyen, flere CDN-leverandører, og klienter. I sin egen maskinvare-infrastruktur håndterer Netflix registrering og betalingsservere. Netflix kjører sin online-tjnesneste på virtuelle maskiner på Amazon Cloud. Noen av funksjonene som skjer på Amazon cloud inkluderer:

- *Content ingestion.* Før netflix kan levere filmer til kunder, må de først legge inn og prosessere filmene. Netflix får en studioversjon og opplaster dem til vertene i Amazon-skyen.
- *Content processing.* Maskinene på Amazon cloud lager mange formater av hver film, passende for en rekke klientvideospillere kjørende på datamaskin, mobil eller spillkonsoller. En forskjellig versjon blir laget for hver av disse formatene, med forskjellige bitrater, som tillater adaptiv streaming over HTTP med DASH.
- *Uploading versions to the CDNs.* Når alle versjonene av en film har blitt laget, må vertene i Amazon cloud-en laste opp versjonene til CDN-ene.

For å levere filmer til kunder \*on demand, bruker Netflix mye CDN-teknologi. Netflix brukte i 2012 tre tredjeparts CDN-selskap samtidig.



**Figure 7.6** • Netflix video streaming platform

### YouTube

Med rundt en halv milliard videoer i biblioteket sitt og en halv milliard avspillinger per dag (2011), er YouTube (eid av Google) den største videodelingssiden i verden. Som med Netflix, bruker YouTube mye CDN-teknologi for å distribuere filmene sine. Ulikt Netflix, bruker ikke Google tredjeparts CDN-er, men bruker istedet sine egen private CDN for å distribuere YouTube-videoer. Google har installert serverclustere på mange hundre forskjellige steder. Google bruker DNS for å omdirigere en kunde-forespørsel til et spesifikt cluster.

For mesteparten av tiden er Google sin cluster-strategi å omdirigere klienten til klusteret der RTT-en mellom klienten og clusteret er minst. Men noen ganger må en klient videresendes til et cluster lengre unna for å balansere trafikken på clusterene.

Dersom et cluster for en request for en fil som den ikke besitter, så kan den videresende klienten istedet for å hente filen, ved å bruke en såkalt HTTP redirect-melding.

YouTube bruker HTTP-streaming, og lager ofte et lite antall forskjellige versjoner for en film, hver med en forskjellig bitrate og korresponderende kvalitetsnivå. I 2011, brukte YouTube ikke adaptive streaming (som DASH), men istedet krevde at brukeren manuelt valgte versjon. For å spare båndbredde og serverressurser som ville bli bortkastet ved reposisjonering eller tidlig avslutning, bruker YouTube HTTP byte-range-forespørselen for å begrense strømmen av overførte data etter at en målmengde video er forhåndshentet.

## Kankan

Netflix og YouTube må ikke bare betale for servermaskinvare, men må også for båndbredden til serverne brukt for å distribuere videoene. Gitt skalaene på disse tjenestene blir et slikt "klient-server"-utpllassering ekstermt kostbart.

Vi konkluderer dette avsnittet med å snakke om en annen måte å tilby video *on demand* over internettet på en større skala, som tillater en signifikant lavere kontnad på båndbredde og infrastruktur. Denne måten bruker P2P-leveranse istedet for klient-server(via CDN-er)-leveranse. P2P-videoleveranse har blitt brukt med stor suksess av flere selskaper i Kina, inkludert Kankan (eid av Xunlei), og fler. Kankan, som i 2011 ledet P2P-basert video-on-demand-levering i Kina, med over 20 millioner unike brukere hver måned.

På et høyt nivå er P2P-videostreaming vedlig likt BitTorrent-filnedlasting. Når en peer ønsker å se en film, kontakter den en tracker for å se at andre peers i systemet har en kopi av den filmen. Når peeren så spør om deler av filmen i parallell fra andre peers som har filen. Ulik fra BitTorret, er at spøringer er gjort for deler som skal bli spilt i nær fremtid, for å sikre kontinuerlig avspilling.

Kankan-designet bruker en *tracker* og sin egen DHT (distributed hash table) for å følge med på innhold. Kankan bruker UDP når det er mulig, og leder til stort forbruk av UDP-trafikk i Kina.

## Voice-over-IP

Sanntidssamtaler over internettet blir ofte referert til som **internettlefoni**, det er også kjent som **Voice-over-IP (VoIP)**. Vanlig video er likt på mange måter med VoIP, bortsett fra at det inkluderer video av deltakerne, i tillegg til stemmene deres.

### Limitations of the Best-Effort IP Service

IP tilbyr "best-effort"-tjeneste. Dette betyr at IP-protokollen, gjør så godt den kan for å flytte hvert datagram fra kilde til destinasjon så fort som mulig, men lover ingenting, ikke at det kommer frem i det hele tatt heller.

Mangelen på slike garantier er en stor utfordring for designet av sanntidssamtaleapplikasjoner, som er svært sensitive for pakkeforsinkelse, jitter og tap. I denne delen dekkes flere måter som kan forbedre ytelsen til VoIP over et best-effort-nettverk.

I en applikasjon der senderen genererer 8,000 bytes per sekund, og sender disse i en klump med bytes hvert 20 msec. Hver pakke kommer seg til mottakeren med konstant ende-til-ende-forsinkelse, når pakken ankommer hos mottakeren periodisk hvert 20 msec. I et slikt miljø kan mottaker spille av hver chunck så fort den ankommer. Uheldigvis er det slik at pakker kan bli borte og de fleste pakker ikke vil ha samme ende-til-ende forsinkelse. Av denne grunn må mottakeren være forsiktig når det gjelder å bestemme (1) når man skal spille av en chunck, og (2) hva man skal gjøre med en manglende chunck.

### Packet Loss

La oss se for oss et UDP-datagram som har blitt laget av VoIP-applikasjonen. UDP-segmentet er innkapslet i et IP-datagram. Når datagrammer går igjennom nettverket, går det igjennom ruterbuffere (dvs. kør). Det er mulig at en av disse bufferene på veien fra senderen til mottakeren er full, og i dette tilfellet kan det ankommende datagrammet bli forkastet.

Tap kan bli eliminert ved å sende pakker over TCP (som tilbyr pålitelig dataoverføring) istedet for UDP. Imidlertid er retransmisjonsteknikker ofte ansett som uakseptable for sanntidstale-applikasjoner som VoIP, fordi de øker ende-til-ende-forsinkelsen. Videre, på grunn av TCP-overbelastningsstyring, kan pakktap føre til en reduksjon av TCP-senderens overføringshastighet til en hastighet som er lavere enn mottakerens dreneringshastighet, og muligens fører til buffersutting. Av disse grunner kjører de fleste eksisterende VoIP-applikasjoner over UDP som standard. Skype bruker UDP med mindre en bruker står bak en NAT eller brannmur som blokkerer UDP-segmenter (i hvilket tilfelle TCP brukes).

Men det å miste pakker er ikke nødvendigvis så katastrofalt som man kanskje tror. Faktisk kan pakkefallsrater mellom 1 og 20 prosent tolereres, avhengig av hvordan lyden blir kodet og overført, og om hvordan tapet er skjult ved mottakeren. For eksempel kan feilkorrigering (FEC) bidra til å skjule pakketap. Vi ser nedenfor at med FEC overføres redundante opplysninger sammen med den opprinnelige informasjonen, slik at noen av de tapte originaldataene kan gjenvinnes fra den overflødige informasjonen.

Likevel, hvis en eller flere av koblingene mellom avsender og mottaker er svært overbelastet, og pakketap overstiger 10 til 20 prosent (for eksempel på en trådløs kobling), så er det egentlig ingenting som kan gjøres for å oppnå akseptabel lydkvalitet.

Klart har best-service-tjenesten sine begrensninger.

#### End-to-Ende Delay

**Ende-til-ende-forsinkelsen** er akkumulering av overføring, prosessering og køforsinkelser i rutere, forplantningsforsinkelser i koblinger; og forsinkelser i endesystems prosesseringsforsinkelser.

For sanntidstalapplikasjoner, som VoIP, er ende-til-ende-forsinkelser mindre enn 150 msec ikke oppfattet av et menneske. Men forsinkelser mellom 1590 og 400 msec kan være akseptable med ikke ideelle, og forsinkelser over 400 msec kan være et hinder for lydsamtaler. Pakker som forsinkes med mer enn terskelen, blir dermed effektivt tapt.

Mottakersiden av en VoIP-applikasjon vil vanligvis droppe eventuelle pakker som er mer forsinket enn en bestemt terskel (eng. *threshold*), f.eks. mer enn 400 msec. Pakker som forsinkes med mer enn thresholden blir sett på som tapt.

#### Packet Jitter

En viktig komponent i ende-til-ende forsinkelser er variasjonen på køforsinkelsene som en pakke opplever i nettverksruterne. På grunn av disse varierende forsinkelsene, kan tiden fra når en pakke genereres ved kilden til den mottas hos mottakeren variere fra pakke til pakke, som vist i figur 7.1. Dette fenomenet kalles *\*jitter*.

Dersom mottakeren ignorerer tilstedeværelsen av jitter og spiller ut chuncks så snart de kommer, kan den resulterende lydkvaliteten lett bli uforståelig for mottakeren. Heldigvis kan jitter ofte bli fjernet ved å bruke **sekvensnummere**, **timestamps** og **avspillingsdelay**.

#### Removing Jitter at the Receiver for Audio

For vår VoIP-applikasjon, hvor pakker genereres periodisk, burde mottaker forsøke å spille av chuncksene periodisk, med nærvær av tilfeldig nettverksjitter. Dette er typisk gjort ved å å kombinere følgende to mekanismer:

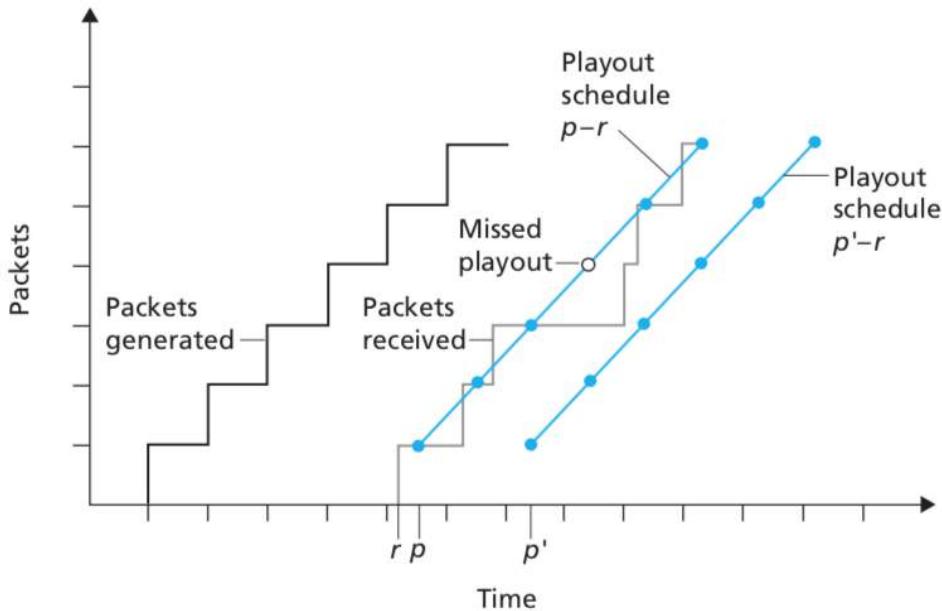
- *Forhåndsfører hver bit med en tidsstempel.* Avsenderen stempler hver chunck med tiden da chunck ble generert.
- *Delaying playout of chuncks at the receiver.* Som vi så i Figur 7.1, må avspillingsforsinkelsen av de mottakte lydbitene være lang nok til at de fleste av pakkene ankommer før de er planlagt for avspilling. Denne avspillingsforsinkelsen kan bli endret under avspillingen av lyden, eller være bestemt for hele avspillingen.

#### Fixed Playout Delay

Med den bestemte-forsinkelse-strategien, vil mottaker prøve å spille av hver bit eksakt  $q$  msec etter at biten ble generert. Så dersom en bit ble tidsstemplert hos senderen hos tid  $t$ , skal biten bli avspilt på tid  $t + q$ . Pakker som ankommer etter den planlagte avspillingstiden blir kastet og ansett som tapte.

Men hva er et godt valg for  $q$ ? VoIP, kan støtte forsinkelser opp til 400 msec, men kan få bedre samtaler med mindre forsinkelser. Men dersom  $q$  er veeldig mye mindre enn 400 msec, vil det være flere pakker som ikke ankommer innen den planlagte avspillingstiden sin. Avviket mellom avspillingsforsinkelsen og tapstapet er illustrert i figur 7.7. Der den første pakken mottas på tiden  $r$ .

For den første avspillingsplanen, er den faste innledende avspillingsforsinkelsen satt til  $p - r$ . Med denne tidsplanen kommer den fjerde pakken ikke fram til den planlagte avspillingen, og mottakeren anser det gått tapt. For den andre avspillingsplanen, er den faste innledende utkoblingsforsinkelsen satt til  $p' - r$ . For denne tidsplanen kommer alle pakker før de planlagte avspillingstidene, og det er derfor ikke noe tap.



**Figure 7.7** ♦ Packet loss for different fixed playout delays

#### Adaptive Playout Delay

Forrige eksempel demonstrerte en viktig forsinkelses-tap forhandlingen som skjer når man bruker en avspillingsstrategi med bestemt avspillingsforsinkelse. Ved å gjøre den første avspillingsforsinkelsen litt større, vil de fleste pakker ankomme før deadlinene sine, og det vil være neglisjerbar tap.

Den naturlige måten å håndtere denne avhandlingen er å estimere nettverksforsinkelsen og variansen av nettverksforsinkelsen, og for å justere avspillingsforsinkelsen tilsvarende i begynnelsen av hver snakkespurt. Her vil da talestillehet enten bli komprimert eller forlenget, utifra endringen i forsinkelse.

Beskriver nå en generisk algoritme som mottakeren kan bruke for å adaptivt justere avspillingsforsinkelser. La  $t_i$  = tidsstempelen til den  $i$ te pakken.  $r_i$  = tiden pakke  $i$  ankommer hos mottakeren, og  $p_i$  = tiden pakke  $i$  blir avspilt hos mottaker.

Ende-til-ende-forsinkelsen til den  $i$ te pakken er  $r_i - t_i$ . På grunn av nettverksjitter, vil denne forsinkelsen variere fra pakke til pakke. La  $d_i$  betegne gjennomsnitts nettverksforsinkelsen når den  $i$ te pakken er mottatt.

$$d_i = (1 - u) d_{i-1} + u (r_i - t_i)$$

, hvor  $u$  er en bestemt konstant f.eks.  $u = 0.01$ . La  $v_i$  betegne det estimerte gjennomsnittsaviket fra gjennomsnittsforsinkelsen.

$$v_i = (1 - u) v_{i-1} + u | r_i - t_i - d_i |$$

Estimatene for  $d_i$  og  $v_i$  er regnet ut for hver pakke som ankommer. Etter å ha regnet ut disse estimatene vil mottaren bruke følgende algoritme for avspillingen av pakkene. Dersom pakke  $i$  er første pakke i en snakkespurt, er avspillingstiden  $p_i$ :

$$p_i = t_i + d_i + Kv_i$$

, hvor  $K$  er en positiv konstant som brukes for å få de færreste av de ankommende pakkene skal bli tapte pga. sen ankommelse. Differansen mellom når en pakke ble sendt til den ble spilt i en spurt er utregnet som et offset, fra når den første pakken ble sendt til den ble avspilt. La

$$q_i = p_i - t_i$$

være tiden mellom fra første pakke i en snakkespurt ble laget til den ble avspilt. Dersom en pakke  $j$  også tilhører samme snakkespurt, vil avspillingstiden dens være

$$p_j = t_j + q_i$$

Algoritmen beskrevet gir perfekt mening antatt at en mottaker kan finne ut om en pakke er den første pakken i en snakkespurt. Dette kan gjøres ved å se på signalenergien.

### Recovering from Packet Loss

Skal nå beskrive noen skjemaer for å forsøke å beholde akseptabel lydkvalitet i nærvær av pakketap. Slike skjemaer er kalt **loss recovery schemes**. En pakke er tapt om den ikke kommer til mottakeren, eller om den kommer etter den planlagte avspillingstiden.

To typer tapforventningsordninger er **forward error correction (FEC)** og **interleaving**.

#### Forward Error Correction (FEC)

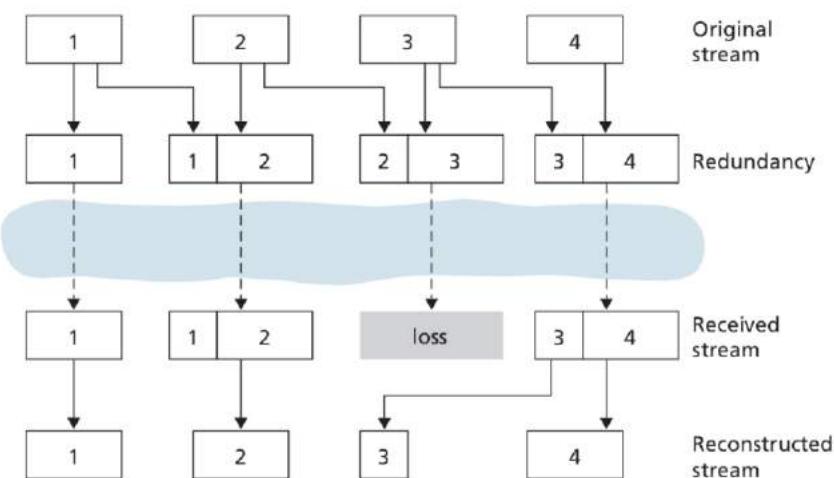
Hovedidén bak FEC er å legge til redundat informasjon i den original pakkestrømmen. For kostnaden av å marginalt øke overføringsraten, kan den redundante informasjonen brukes til å rekonstruere tilnærminger eller eksakte versjoner av noen av de tapte pakkene.

Den første FEC-mekanismen sender en redundat kodet chunk etter hver  $n$ 'te chunck. Den redundante chuncken blir laget ved å ta XOR på alle de  $n$  originale chunckene. På denne måten, dersom en av pakkene i en gruppe på  $n + 1$  pakker mistes, kan mottakeren rekonstruere den tapte pakken. Men dersom to eller flere pakkene i en gruppe blir mistet, kan ikke mottakeren rekonstruere pakkene. Ved å holde  $n + 1$ , gruppstørrelsen, liten, kan en stor andel av tapte pakkene bli rekonstruert når det ikke er mye tap.

Den andre FEC-mekanismen er å sende en lavere lydoppløsningsstrøm som den redundante informasjonen. For eksempel kan senderen opprette en nominell lydstrøm og en tilsvarende lavopløsning, lav-bitrates lydstrøm. (Den nominelle strømmen kan være en PCM-koding på 64 kbps, og den lavere-kvalitetsstrømmen kan være GSM-kodet på 13 kbps.)

Som vist i figur 7.8 konstruerer avsenderen den  $n$ te pakken ved å ta den neste delen fra den nominelle strømmen og legge til det  $(n - 1)$ 'te stykket fra den redundante strømmen. På denne måten, når det ikke er et sammenhengende pakketap, kan mottakeren skjule tapet ved å spille ut den lavbitede kodede klumpen som kommer med den etterfølgende pakken. På den annen side øker de ekstra bitsene overføringsbåndbredden og avspillingsforsinkelsen.

Dersom man ønsker å kunne dekke sammenhengende tap, så kan man legge til både  $(n-1)$ 'te og  $(n-2)$ 'te ledd i den redundante strømmen, men som igjen vil øke overføringsbåndbredden og avspillingsforsinkelsen.

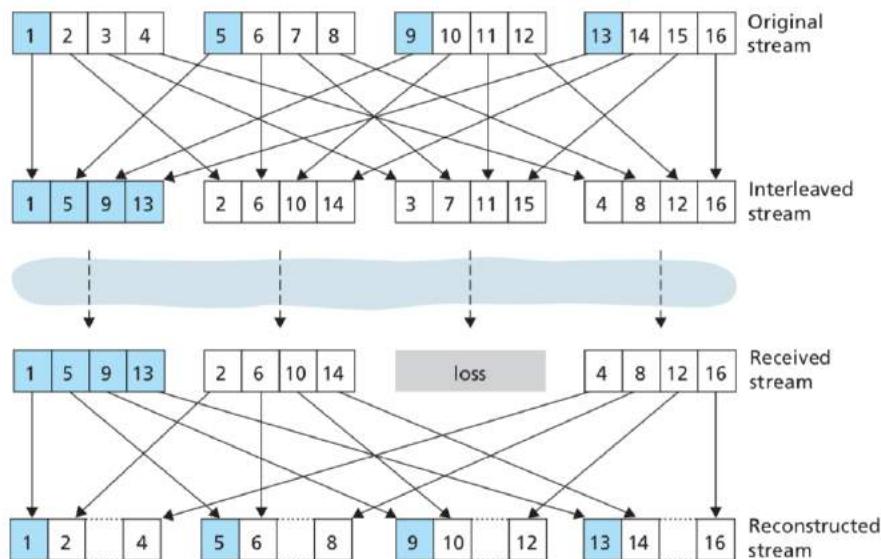


**Figure 7.8** ♦ Piggybacking lower-quality redundant information

Som et alternativ til redundtant overføring, kan en VoIP-applikasjon sende flettet (eng. interleaved) lyd. Som vist i Figur 7.9, kan senderen resekvensere lyddata før overføring, slik at opprinnelig tilstøtende enheter er separert med en viss avstand i den overførte strømmen.

Interleaving kan redusere effekten av pakketap. Dersom for eksempel enheter er 5 msec lange og chunks er 20 msec (det vil si fire enheter per brikke), så kan den første chunken inneholde enhetene 1, 5, 9 og 13; den andre klumpen kunne inneholde enhetene 2, 6, 10 og 14; og så videre.

- Figur 7.9 viser at tapet av en enkelt pakke fra en interleaved strøm resulterer i flere små hull i den rekonstruerte strømmen, i motsetning til det eneste store gapet som ville oppstå i en ikke-interleaved strøm.



**Figure 7.9** ♦ Sending interleaved audio

#### Error Concealment

Error concealment (nor. feilskjuling) skjemaer forsøker å produsere erstattninger for tapte pakker som er like originalen. Slike teknikker jobber for relativt små tapsrater (mindre enn 15%), og får små pakker (4-40 msec). Når taplengden nærmer lengden på et fonem (5-100 msec - minste lydlengde som kan endre meningen til et ord) kan disse teknikkene feile, siden hele fonemer kan bli mistet av lytteren.

Den enkleste formen for mottakerbasert gjenopprettning er **pakkerepetisjon** (eng. packet repetition). Pakkerepetisjon erstatter tapte pakker med kopier av pakkene som kom umiddelbart før tapet. Den har lav kompleksitet og fungerer rimelig bra. En annen form for mottakerbasert gjenopprettning er **interpolering**, som bruker lyd før og etter tapet til å interpolere en passende pakke for å dekke tapet. Interpolering utfører noe bedre enn pakkerepetisjon, men er betydelig mer beregningskomplisert.

#### Case Study: VoIP with Skype

Skype er en veldig populær VoIP-applikasjon med over 50 millioner aktive brukere. I tillegg til å tilby vert-til-vert VoIP-tjeneste, tilbyr også Skype vert-til-telefon-tjeneste, telefon-til-vert-tjeneste, og fler-parti vert-til-vert videokonferering.

Fordi Skype-protokollen er proprietær (har eierskap), og fordi alle Skypes kontroll- og mediepakker. For både tale og video har Skype-klientene mange forskjellige kodeker, som kan kode media til et bredt spektrum av rater og kvaliteter. F.eks. har videosporene for Skype har blitt målt til å være så lave som 30 kbps for en lavkvalitets økt på opptil nesten 1 Mbps for en høykvalitets økt.

Vanligvis er Skypes lydkvalitet bedre enn kvaliteten på "POTS" (vanlig gammel telefonservice) levert av telefonlinjesystemet. (Skype-kodeker sampler vanligvis lyd på 16.000 samples/sek eller høyere, som gir rikere toner enn POTS, som sampler på 8000 samples/sek.) Som standard sender Skype lyd- og videopakker over UDP. Kontrollpakker sendes imidlertid over TCP, og mediepakker sendes også over TCP når brannmurer blokkerer UDP-strømmer.

- Skype bruker FEC for tapgjenvinning for både tale- og videostrømmer sendt over UDP. Skype-klienten tilpasser også lyd- og videostrømmene den sender til nåværende nettverksforhold, ved å endre videokvalitet og FEC overhead.

Skype bruker P2P-teknikker på et antall innovative måter, som på en fin måte illustrerer hvordan P2P kan bli brukt i applikasjoner utover innholdsdistribusjon og fildeling.

- Som med direktemeldinger er vert-til-vert Internett-telefoni iboende P2P siden, i hjertet av applikasjonen, kommuniserer par av brukere (det vil si kolleger) i sanntid med hverandre. Men Skype har også P2P-teknikker for to andre viktige funksjoner, nemlig for brukerplassering og NAT-traversal.
- Som vist i Figur 7.10 er hver peer (vert) i Skype organisert i et hierarisk nettverk, med hver peer enten klassifisert som super-peer eller vanlig peer. Skype opprettholder en indeks som kartlegger Skype-brukernavn til nåværende IP-adresser (og portnumre). Denne indeksen er distribuert over super-peers. Når Alice ønsker å ringe til Bob, søker hennes Skype-klient den distribuerte indeksen for å avgjøre Bobs nåværende IP-adresse.
- P2P-teknikker er også brukt i Skype **relays**, som er nyttige for å etablere samtaler mellom verter på hjemmenettverk. Mange hjemmenettverkkonfigurasjoner tilbyr tilgang til internettet gjennom NATs. Husk at en NAT forhindrer en vert fra utenfor hjemmenettverket fra å starte en forbindelse til en vert i hjemmenettverket. Hvis begge Skype-oppringerne har NAT, er det et problem - ingen av vertene verken aksepteres et anrop initiert av den andre, noe som gjør en samtale tilsynelatende umulig. Den klare bruken av super-peers og reléer løser pent dette problemet.
  - Vertene bak NAT-ene får tildelt en super-peer, når de logger inn. Når en av vertene, vert A, ønsker å ringe den andre verten, vert B, vil vert A informere super-peeren sin om at den vil ringe vert B. Super-peeren kontakter super-peeren til vert B, som igjen informerer vert B om at vert A vil ta kontakt. De to super-peerne velger så en tredje ikke-NATet super-peer - relay-peeren - som skal være en releé mellom vert A og B. Vert A og vert Bs super-peers ber så sine verter om å initialisere en økt med releé. Dette er vist i figur 7.10.
  - Dersom det er flere verter som skal kommunisere sammen, bruker Skype en smart distribusjonsteknikk for å redusere den båndbreddeforbruket. Dersom det er  $N$  deltakere, så istedet for å sende  $N - 1$  lydstrømmer sender de alle strømmene til samtalesarteren, som kombinerer alle lydstrømmene til en felles lydstrøm som sendes til alle deltakerne via releén. For video, sendes alle  $N$  videostrømmene til et servercluster (i Estonia i 2011) som videresender de andre  $N - 1$  videostrømmene til hver deltaker.

Man lurer kanskje på hvorfor man alltid sender strømmene fra samme sted, enten innom servercluster eller samtalesarteren. I begge fremgangsmåtene vil alle de  $N(N-1)$  strømmene bli kollektivt mottatt hos de  $N$  deltakerne i samtalen.

Årsaken er at fordi oppstrøms koblingsbåndbredder er signifikant lavere enn nedstrøms båndbredder i de fleste aksesslinker, kan oppstrømslinkene kanskje ikke støtte  $N-1$ -strømmer med P2P-tilnærmingen.

I VoIP-systemer som Skype kan vert A og vert B sniffe hverandres IP-adresser, som kan brukes med geolokasjonstjenester for å finne IP-adressens plassering, og se på BitTorrent hva som har blitt lastet ned på IP-adressen.

## Protocols for Real-Time Conversational Applications

Sanntidstaleapplikasjoner, inkludert VoI og videokonferansing, er tiltrekende og veldig populært. Derfor jobber IETF og ITU hardt med å få ut standarder for denne klassen av applikasjoner. Med de passende standarder som er tilgjengelige for denne klassen applikasjoner, skaper uavhengige selskaper nye produkter som samarbeider med hverandre. I denne delen undersøker vi RTP for sanntidstaleapplikasjoner. Begge standarder nyter utbredt implementering i industriprodukter.

### RTP - Real-Time Transport Protocol

I forrige del så vi at sendersiden av en VoIP-applikasjon legger på header-felt til lyd-chunker før den sender de til transportlaget. Disse header-feltene inkluderer sekvensnummer og tidsstempeler. Ettersom de fleste multimedia networking applikasjoner bruker disse type felter, kan det være lurt med en standardisert pakkestruktur, med andre mulige nyttige felt. RTP, er en slik standard og kan bli brukt til å transportere vanlige formater som PCM, ACC og MP3 for lyd, og MPEG og H.263 for video.

#### RTP Basics

RTP kjører typisk over UDP. Den sendende siden innkapsler en mediachunk i en RTP-pakke, innkapsler denne i et UDP-segment, og gir det til IP. Mottakersiden ekstraherer RTP-pakken fra UDP-segmentet, ekstraherer mediaet fra RTP-pakken, og gir det til mediaplayeren for dekoding og rendering.

For eksempel, bruk bruken av RTP for å transportere stemme. Anta at kildelyden er PCM-kodet (det betyr, samplet, kvantisert og digitalisert) ved 64 kbps. Anta videre at applikasjonen samler de kodede dataene i 20 msec biter, det vil si 160 bytes i en chunk. Sendingssiden legger til hver chunk av lyddataene en RTP-header som inneholder typen lydkoding, et sekvensnummer og en tidsstempel. RTP-headeren er normalt 12 byte. Lydchunken sammen med RTP-headeren danner RTP-pakken.

Hvis en applikasjon inkorporerer RTP-i stedet for en proprietær ordning for å gi payload, sekvensnummer eller tidsstemplar, vil applikasjonen lettere kunne samhandle med andre nettverksbaserte multimedieapplikasjoner.

RTP tillater at hver kilde (for eksempel et kamera eller en mikrofon) tildeles sin egen uavhengige RTP-strøm av pakker. For eksempel, for en videokonferanse mellom to deltakere, kunne fire RTP-strømmer åpnes - to strømmer for overføring av lyden (en i hver retning) og to strømmer for overføring av videoen (igjen, en i hver retning). Men mange populære kodeteknikker - inkludert MPEG 1 og MPEG 2 - bundler lyden og videoen til en enkelt strøm under kodingsprosessen. Når lyden og videoen er samlet av koderen, blir det bare en RTP-strøm generert i hver retning.

RTP multicast-strømmer som tilhører sammen, for eksempel lyd- og videostrømmer som kommer fra flere avsendere i et videokonferanseprogram, tilhører en **RTP-økt**.

#### RTP Packet Header Fields

Som vist i Figur 7.11 er de firehoved RTP-headerfeltene payloadtype, sekvensnummer, tidsstempel og kildeidentifikasjonsfelt.

| Payload type | Sequence number | Timestamp | Synchronization source identifier | Miscellaneous fields |
|--------------|-----------------|-----------|-----------------------------------|----------------------|
|--------------|-----------------|-----------|-----------------------------------|----------------------|

**Figure 7.11** • RTP header fields

Payloadtype-feltet i RTP-pakken er 7 bits lang, som angider typen koding (f.eks. PCM, adaptiv delta modulering osv.) Dersom en sender endrerenkodingen underveis i en økt kan senderen informere mottakeren om dette ved å endre payloadtype-feltet. Tabell 7.2 og Tabell 7.3 i boken viser noen av lyd-payloadtypene og video-payloadtypene som støttes av RTP.

De andre viktige feltene er:

- *Sekvensnummerfelt*. Sekvensnummerfeltet er 16 bit langt. Sekvensnummeret inkrementerer med en for hver RTP-pakke sendt og kan brukes av mottakeren til å detektere pakketap og fikse pakkesekvensen. Dersom en mottaker ser at det er et hull i sekvensnumrene fra 86 til 89, da vet mottakeren at pakke 87 og 88 er borte. Da kan mottakeren prøve å skjule den tapte dataen.
- *Timestamp field*. Tidsstempel-feltet er 32 bit langt. Den reflekterer prøveuttaket for den første byten i RTP-datatpakken. Som vi så i forrige avsnitt, kan mottakeren bruke tidsstemplar for å fjerne pakkejitter introdusert i nettverket og å sørge for synkron avspilling på mottakeren. Klokkeslettet er avledd fra en sample-klokke hos avsenderen.
- *Synchronization source identifier (SSRC)*. SSRC feltet er 32 bits langt. Den identifiserer kilden til RTP-strømmen. Typisk har hver strøm i en RTP-økt en distinkt SSRC. SSRC-en er ikke IP-adressen til senderen, men istedet et tilfeldig nummer som kilden lager når strømmen startes. Sannsynligheten for at to strømmen får samme SSRC er veldig liten, og skulle det skje velger de to kildene en ny SSRC-verdi.

## Kapittel 8 - Security in Computer Networks

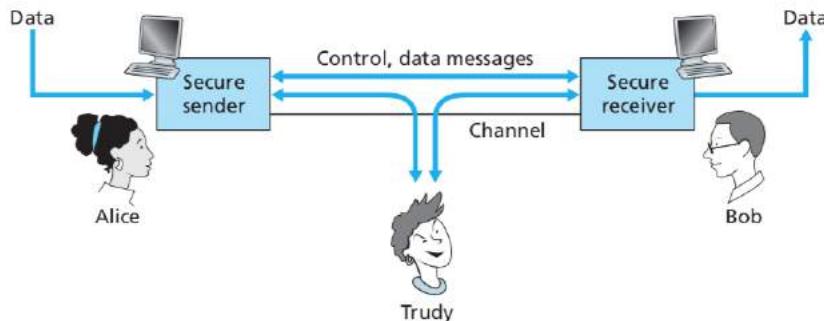
Vi skal nå se på hvordan man kan sikre nettverk fra internettangrep som malware attacks, denial of service, sniffing, source masquerading, og message modification og sletting. La oss introdusere Alice og Bob, to personer som vil kommunisere, og ønsker å gjøre det *sikkert*.

I første del av dette kapitlet skal vi dekke de grunnleggende kryptografiteknikkene som tillater kryptering av kommunikasjon, autentisering av partiet med hvem man kommuniserer og sikrer meldingsintegritet. I den andre delen av dette kapitlet skal vi undersøke hvordan de grunnleggende kryptografiprinsippene kan brukes til å lage sikre nettverksprotokoller. I tredje del av dette kapitlet skal vi vurdere operativ sikkerhet, som handler om å beskytte organisasjonsnettverk mot angrep.

## What Is Network Security?

Hva betyr det at noe er sikkert? Alice vil at kun Bob skal kunne forstå meldingen hun har sendt, selvom de kommuniserer over et *usikkert* medium hvor en intrenger (Trudy, the intruder) muligens fanger opp det som blir sendt fra Alice til Bob. Bob ønsker også at meldinger han mottar fra Alice, faktisk var fra Alice, og Alice ønsker at den personen hun kommuniserer med faktisk er Bob. Alice og Bob ønsker også at innholdet i meldingene deres ikke endret i overføringen. De ønsker også at de kan være garanert å kommunisere (at ingen nekter dem å kommunisere). Gitt disse vurderingene kan vi identifisere følgende ønskelige egenskaper for **sikker kommunikasjon**.

- *Confidentiality*. Kun sender og tiltenkt mottaker skal kunne forstå innholdet i overførte melding. Da "eavesdroppere" kan fange opp meldingene er det viktig at meldingen er kryptert slik at kun mottakeren kan forstå innholdet. Dette aspektet av konfidensialitet er det vanligste oppfattet som meningen med *sikker kommunikasjon*.
- *Message integrity*. Alice og Bob ønsker at innholdet i meldingen ikke blir endret på, enten av skadelige intensjoner eller uhell, i overføringen. Utvidelser til checksum-teknikker kan hjelpe på slik meldingsintegritet.
- *End-point authentication*. Både sender og mottaker skal kunne bekrefte identiteten sin til de andre deltakerene i kommunikasjonen. Når en bruker ønsker å få tilgang på en innboks, hvordan autentiserer mailserveren brukeren som den personen han eller hun sier at de er?
- *Operational security*. Nesten alle organisasjoner (selskaper, universiteter osv) i dag har nettverk som er koblet til et offentlige internettet. Disse internettene kan derfor potensielt bli komrommet. Angripere kan forsøke å plassere worms i vertene i nettverket, skaffe bedriftshemmeligheter, karlegge de interne nettverkskonfigurasjonene, og starte DoS-angrep.



**Figure 8.1** ♦ Sender, receiver, and intruder (Alice, Bob, and Trudy)

Etter å ha etablert hva vi mener med nettverkssikkerhet, kan vi se på nøyaktig hva slags informasjon som en inntrenger har tilgang til, og hva inntrengeren kan gjøre med det. Figur 8.1 viser et scenario der Alice, senderen, vil sende data til Bob, mottakeren. For å utveksle data sikkert, mens de møter kravene om konfidenzialitet, endepunkt autentisering og meldingsintegritet, vil Alice og Bob utveksle kontrollmeldinger og datameldinger (på samme måte som TCP-sendere og mottakere utveksler kotrollsegmenter og datasegmenter). Alle eller noen av disse meldingene blir vanligvis kryptert. Som diskutert i Kap. 1, kan en inntrenger potensielt utføre

- *eavesdropping* - sniffe eller opptak av kontroll- og data-meldinger på en kanal
- *modification, insertion, or deletion* av meldinger eller meldingsinnhold.

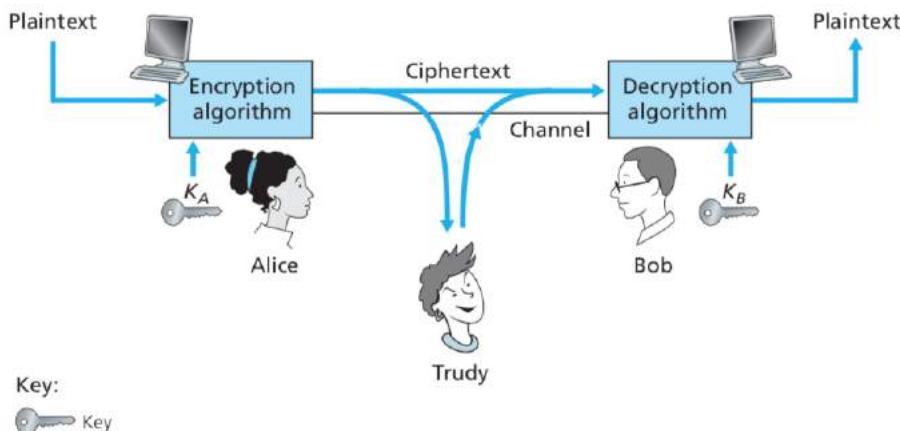
Som vi ser, med mindre de nødvendige mottiltak blir gjort, tillater dette inntrengerne å utføre et bredt spekter av sikkerhetsangrep: snooping på kommunikasjon (muligens å stjele passord og data), utgjøre seg for å være en annen, kapre en pågående økt, nekte tjenesten å legitimerte nettverksbrukere ved å overbelaste systemressurser, og så videre.

La oss nå se på kryptografi. Selvom bruken av kryptografi i å gi konfidenzialitet er selvsagt, ser vi snart at det også er sentralt for å gi kryptering til endepunktsautentisering og meldingsintegritet, som gjør kryptografi til en av hjørnestenene i nettverkssikkerhet.

## Principles of Cryptography

Kryptografiteknikker tillater en sender forkle/disguise data slik at en inntrerger ikke kan få noen informasjon fra den mottatte dataen. Mottakeren må selvfølgelig hente den originale dataen fra den krypterte dataen. Figur 8.2 viser noen viktige terminologier.

La oss anta at Alice vil sende Bob en melding. Alice sin melding i sin originale form (f.eks. "Bob, jeg elsker deg. Alice") er kjent som **plaintext** eller **klartekst**. Alice krypterer klartekst-meldingen med en **krypteringsalgoritme** slik at den krypterte meldingen, kjent som **ciphertekst**, ser uleselig ut for en inntrerger. I mange moderne kryptografiske systemer, inkludert de brukt av internettet, er selve krypteringsteknikken kjent, publisert og standardisert og tilgjengelig for alle.



**Figure 8.2** ♦ Cryptographic components

Klart, dersom alle kjenner metoden for kryptering, må det være noen hemmelig informasjon som hindrer en intrenger fra å dekryptere overført data. Det er her nøkler kommer inn. I Figur 8.2 gir Alice en **nøkkel**,  $K_A$ , en string med tall eller bokstaver som input til krypteringsalgoritmen. Krypteringsalgoritmen tar nøkkelen og klartekst-meldingen,  $m$ , som input og produserer ciphertekst som output. Notasjonen  $K_A(m)$  refererer til ciphertekst-formen av klarteksten  $m$ . Likt vil Bob gi en nøkkel,  $K_B$  til **dekrypteringsalgoritmen** som tar cipherteksten og gjør den om til klartekst. Så dersom Bob får en kryptert tekst  $K_A(m)$ , dekrypterer han den ved å kalkulere  $K_B(K_A(m)) = m$ .

- I et **symmetrisk nøkkelsystem** har både Alices og Bobs nøkler identiske og hemmelige.
- I et **offentlig nøkkelsystem** (eng. *public*) er par av nøkler brukt. Den ene nøkkelen er kjent av både Bob og Alice (ja, den er kjent for hele verden). Den andre nøkkelen er det kun kjent av enten Bob eller Alice (men ikke begge)

### Symmetric Key Cryptography

Før vi ser på de moderne krypteringsalgoritmene, kan vi se på en veldig gammel og enkel krypteringsalgoritme - kjent som **Caesar cipher**.

For engelsk tekst, fungerer Caesar-cipheren slik at den tar hver bokstav i klartekst-meldingen, og erstatter den med bokstaven som kommer  $k$  bokstaver senere (som tillater sykler, dvs.  $z$  blir fulgt av  $a$ ). For  $k = 3$ , da blir  $a$  i klartekst  $d$  i ciphertekst,  $b$  i klartekst blir  $e$  i ciphertekst osv osv. Det ville ikke ta lang tid å cracke denne koden, da det kun er 25 mulige verdier (i det engelske alfabetet).

En forbedring til Caesar-cipheren er **monoalphabetic cipher**, som erstatter en bokstav i alfabetet med en annen bokstav i alfabetet. Imidlertid, i stedet for å erstatte i henhold til et vanlig mønster (for eksempel, erstatning med en offset for  $k$  for alle bokstaver), kan et enhver bokstav erstattes av et hvilken som helst annen bokstav, så lenge hver bokstav har en unik ssubstitusjonsbokstav og vice versa. Substitusjonsregelen i figur 8.3 viser en mulig regel for koding av ren tekst. Her vil  $a$  i klartekst gi  $m$  i ciphertekst,  $b$  i klartekst gi  $n$  i ciphertekst. Kan lett løses med brute-force.

|                    |   |
|--------------------|---|
| Plaintext letter:  | a b c d e f g h i j k l m n o p q r s t u v w x y z |
| Ciphertext letter: | m n b v c x z a s d f g h j k l p o i u y t r e w q |

**Figure 8.3** ♦ A monoalphabetic cipher

Når man vurderer hvor lett det kan være for Trudy å bryte Bob og Alices kryptering, kan man skille tre forskjellige scenarioer, avhengig av hvilken informasjon innitrengeren har.

- *Ciphertext-only attack*. I noen tilfeller kan innitrengeren bare ha tilgang til den oppfangede cipherteksten, uten noen viss informasjon om innholdet i rentekstmeldingen. Vi har sett hvordan statistisk analyse kan hjelpe i et kryptert-teksteneste angrep på en krypteringsordning.
- *Known-plaintext attack*. Dersom Trudy på en måte vet at "bob" eller "alice" dukker opp i ciphertekst-meldingenDa kunne hun ha bestemt (plaintekst, ciphertekst) parringen for bokstavene *a*, *I*, *i*, *c*, *e*, *b* og o. Når en innitrenger vet noen av parringene, kaller vi dette et **known-plaintext attack** på krypteringsordningen.
- *Chosen-plaintext attack*. I et **chosen-plaintext attack**, er innitrengeren i stand til å velge plaintext-meldingen og få den tilhørende krypteringsformularen. For de enkle krypteringsalgoritmene vi har sett så langt, kan Trudy få Alice til å sende meldingen: "The quick brown fox jumps over the lazy dog", kunne hun helt bryte krypteringsskjemaet. Vi ser snart at for mer sofistikerte krypteringsteknikker betyr et chosen-plaintext-angrep ikke nødvendigvis at krypteringsteknikken kan brytes.

For fem hundre år siden, ble teknikker for å forbedre monoalfabetisk krypterings oppfunnet, kjent som **polyalphabetic encryption**. Ideen bak denne krypteringen er å bruke flere monoalfabetiske cipher, med et bestemt monoalfabetisk cipher for å kryptere en bokstav på en spesifikk posisjon i plaintext-meldingen. Et eksempel på en polyalfabetisk krypteringsskjema er vist i figur 8.4. Der det har blitt valgt to Caesar cipher,  $C_1$  og  $C_2$ , i det repeterende mønsteret  $C_1, C_2, C_2, C_1, C_2$ . Dvs. første bokstav bruker  $C_1$ , andre bruker  $C_2$ , tredje bruker  $C_2$ , fjerde bruker  $C_1$  og femte bruker  $C_2$ .

|                   |   |
|-------------------|---|
| Plaintext letter: | a b c d e f g h i j k l m n o p q r s t u v w x y z |
| $C_1(k = 5)$ :    | f g h i j k l m n o p q r s t u v w x y z a b c d e |
| $C_2(k = 19)$ :   | t u v w x y z a b c d e f g h i j k l m n o p q r s |

**Figure 8.4** • A polyalphabetic cipher using two Caesar ciphers

I dette eksemplet er krypterings- og dekrypteringsnøkkelen kunnskapen om de to Caesar-nøklene ( $k = 5, k = 19$ ) og mønsteret  $C_1, C_2, C_2, C_1, C_2$ .

### Block Ciphers

La oss nå se på litt mer moderne tider, og se på hvordan symmetrisk-nøkkel kryptering skjer idag. Det er to store klasser av symmetriske krypteringsteknikker: **stream ciphers** og **block ciphers**. Vi skal nå i dette avsnittet fokusere på blockcipher, som blir brukt i mange sikre internettprotokoller, som PGP (sikker epost), SSL (sikre TCP-tilkoblinger) og IPsec (for å sikre nettverkslagstransport).

I en blockcipher er meldingen som skal bli kryptert prosessert i blokker av  $k$  bits. For eksempel  $k = 64$ , da blir meldingen delt inn i 64-bit blokker, og hver blokk blir kryptert uavhengig av hverandre. Får å kryptere en blokk, bruker cipheren en-til-en mapping for å mappe de  $k$ -bit blokkene med klartekst til en  $k$ -bit blokk med ciphertekst.

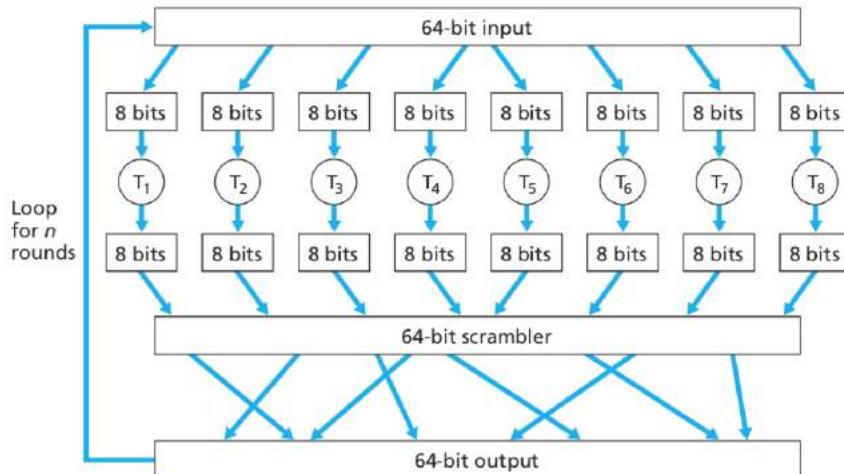
Så la oss se på et eksempel. La  $k = 3$ , slik at blokkcipheren mapper 3-bit input til 3-bit output. En mulig mapping er gitt i Tabell 8.1. Legg merke til at det er en en-til-en mapping. Blokkcipheren deler da altså opp plaintexten i deler på 3 bits. Du kan da bekrefte at meldingen 010110001111 blir kryptert til 101000111001.

For å brute force denne krypteringsskjemaet, kan kjapt bli gjort på en vanlig PC. Dersom  $k = 3$ , vil man ha  $2^3$  mulige inputs. Disse 8 inputtene kan bli permutert på  $8! = 40,320$  forskjellige måter. Det er ikke så mange kombinasjoner å gjøre på en datamaskin.

For å unngå brute-force-angrep, bruker blokkciphers vanligvis mye større blokker, som består av  $k = 64$  bits eller enda større. Merk at antall mulige mappings for en generell  $k$ -blokk-kryptering er  $2^{k!}$ . Som er astronomiske for selv moderate verdier av  $k$  (som  $k = 64$ ).

Selv om blokkcipheren, som ble beskrevet, med moderate verdier av  $k$  kan produsere robuste symmetriske nøkkeler, er de dessverre vanskelig å implementere. For  $k = 64$  og for en bestemt mapping, ville Alice og Bob måtte vedlikeholde et bord med 264 inngangsverdier, noe som er en uopprettelig oppgave. Videre, hvis Alice og Bob skulle bytte nøkler, måtte de hver for seg regenerere bordet.

I stedet bruker blokkciphrene vanligvis funksjoner som simulerer tilfeldig permuterte tabeller. Et eksempel av en slik funksjon for  $k = 64$  bits er vist i figur 8.5. Funksjonen bryter først en 64-bit blokk i 8 biter, med hver brikke som består av 8 bits. Hver 8-bits chunk behandles av en 8-bits til 8-bits tabell, som er av brukbar størrelse. For eksempel behandles den første klumpen ved tabellen betegnet av  $T_1$ . Deretter monteres de 8 utgangsstykke i en 64-biters blokk. Posisjonene til de 64 bitene i blokken blir deretter scramblet (permuttert) for å produsere en 64-bits utgang. Denne utgangen blir gitt tilbake til 64-bits inngangen, hvor en annen syklus begynner. Etter  $n$  slike sykluser, gir funksjonen en 64-biters blokk med ciphertekst. Formålet med rundene er å få hver inngangsbbit å påvirke de fleste (om ikke alle) de endelige utgangsbitene. (Hvis bare en runde ble brukt, ville en gitt inngangsbbit bare påvirke 8 av 64 utgangsbitene.) Nøkkelen for denne blokk-krypteringsalgoritmen ville være de åtte permutasjonstabellene (forutsatt at scramble-funksjonen er offentlig kjent).



**Figure 8.5** ♦ An example of a block cipher

### Cipher-Block Chaining

I nettverksapplikasjoner må vi gjerne kryptere lange meldinger (eller lange strømmer med data). Dersom vi bruker en blockcipher beskrevet over ved å dele opp meldingen i  $k$ -bit blokker og uavhengig kryptert hver blokk. Dersom to plaintext blokker er like, som f.eks. "HTTP/1.1", ville begge blokkene blitt kryptert til den samme cipherteksten.

For å fikse dette problemet kan vi legge til litt tilfeldighet i cipherteksten slik at to identiske plaintext-blokker produserer same ciphertekst blokk. Vi betegner block-cipherkrypteringsalgoritmen med nøkkelen  $S$  som  $K_S$ . Vi betegner  $m(i)$  den  $i$ te plaintext-blokk, og  $c(i)$  den  $i$ te ciphertekstblokken.

- Senderen lager en random  $k$ -bit tall  $r(i)$  for den  $i$ te blokken og kalkulerer  $c(i) = K_S(m(i) \oplus r(i))$ . Det regnes ut et nytt tilfeldig tall for hver blokk. Senderen sender  $c(1), r(1), c(2), r(2), c(3), r(3)$ , osv. Sideen mottakeren mottar  $c(i)$  og  $r(i)$  kan den kalkulere plaintext-blokk ved å regne ut  $m(i) = K_S(c(i)) \oplus r(i)$ .

Den stakkars leseren vil merke seg at innføring av tilfeldighet løser et problem, men skaper et annet: nemlig, Alice må overføre dobbelt så mange biter som før. Faktisk, for hver krypteringsbit må hun nå også sende en tilfeldig bit, dobling av nødvendig båndbredde. For å få det som vi vil, bruker blockciphers vanligvis en teknikk kalt **Cipher Block Chaining (CBC)**. Den grunnleggende ideen er å sende \*bare en tilfeldig verdi sammen med den aller første meldingen, og så har avsenderen og mottakeren bruk de beregnede cipherblokkene i stedet for det etterfølgende tilfeldige tallet. Spesifikt opererer CBC som følger:

1. Før man krypterer meldingen lager senderen en tilfeldig  $k$ -bit streng, kalt **Initialization Vector (IV)**. La oss betegne denne vektoren som  $c(0)$ . Senderen sender IV i klartekst til mottakeren.
2. For den første blokken regner senderen ut  $c(1) = K_S(m(1) \oplus c(0))$ . Senderen sender den krypterte blokken  $c(1)$  til mottakeren.
3. For den  $i$ te blokken genererer senderen den  $i$ te cipherblokken fra  $c(i) = K_S(m(i) \oplus c(i-1))$

Mottakeren får meldingen i klartekst ved å kalkulere  $m(i) = K_S(c(i)) \oplus c(i-1)$

### Public Key Encryption

Vi ser snart at offentlig-nøkkel kryptografisystemer også har flere fantastiske egenskaper som gjør dem nyttige ikke bare for kryptering, men også for autentisering og digitale signaturer.

Bruken av offentlig-nøkkel kryptografi er konseptuelt veldig enkelt. Anta at Alice ønsker å kommunisere med Bob. Som vist i Figur 8.6, istedet for at Bob og Alice deler en felles hemmelig nøkkel, så har Bob (mottakeren) to nøkler - en **offentlig** nøkkel som er tilgjengelig for alle i verden og en **privat** nøkkel som kun Bob vet om.

- Vi bruker notasjonen  $K_B^+$  og  $K_B^-$  for å referere til Bobs offentlige og private nøkler.
- For å kommunisere med Bob, henter Alice først Bobs offentlige nøkkel. Alice krypterer deretter hennes melding,  $m$ , til Bob ved hjelp av Bobs offentlige nøkkel og en kjent (for eksempel, standardisert) krypteringsalgoritme; det vil si, Alice beregner  $K_B^+(m)$ . Bob mottar Alices krypterte melding og bruker sin private nøkkel og en kjent (for eksempel standardisert) dekrypteringsalgoritme for å dekryptere Alices krypterte melding. Det vil si, Bob beregner  $K_B^-(K_B^+(m))$ .

Vi skal snart se at man få den offentlig og private nøkkelen slik at  $K_B^-(K_B^+(m)) = K_B^+(K_B^-(m)) = m$ .

Det er klart at hvis offentlig nøkkelskryptografi skal fungere, må nøkkelvalg og kryptering / dekryptering gjøres på en slik måte at det er umulig (eller nærmest umulig) for en inntrerger å enten avgjøre Bobs private nøkkel eller på annen måte dekryptere eller gjette Alices melding til Bob.

Et annet problem er at siden Bobs krypteringsnøkkel er offentlig, kan alle sende en kryptert melding til Bob, inkludert Alice eller noen som *hevder* å være Alice. I tilfelle der man har en enkelt felles hemmelig nøkkel, identifiserer det faktum at avsenderen kjenner den hemmelige nøkkelen implisitt hvem senderen er til mottakeren. I tilfelle av offentlig nøkkel kryptering, er dette imidlertid ikke lenger tilfelle, siden alle kan sende en kryptert melding til Bob ved hjelp av Bobs offentlige nøkkel. En digital signatur, et emne vi skal se på nå, er nødvendig for å binde en avsender til en melding.

## Message Integrity and Digital Signatures

I dette avsnittet skal vi se på en viktig del av kryptografi, som er å tilby **meldingsintegritet** - også kjent som meldingsautentisering. Sammen med meldingsintegritet, skal vi se på to relaterte temaer: digitale signaturer og endepunkts-autentisering.

Vi definerer meldingsintegritetsproblemet en gang til. Anta at Bob mottar en melding (kanskje kryptert eller i plaintekst) og han tror at meldingen har blitt sendt fra Alice. For å autentisere denne meldingen må Bob verifisere:

1. Meldingen ble faktisk sendt fra Alice
2. Meldingen ble ikke endret eller tullet med på sin vei til Bob.

## Cryptographic Hash Functions

En hashfunksjon tar inn et input  $m$ , regner ut en fikset lengde streng  $H(m)$ , kjent som en hash. Internet checksummen og CRC-er treffer denne definisjonen. En kryptografisk hashfunksjon krever for å ha følgende egenskap:

- Det er umulig å finne to forskjellige meldinger  $x$  og  $y$  slik at  $H(x) = H(y)$ .

Uformelt betyr denne egenskapen at det er beregningsmessig umulig for en inntrerger å erstatte en melding for en annen melding som er beskyttet av hash-funksjonen. Det vil si at hvis  $(m, H(m))$  er meldingen og hashen av meldingen "forge"/fikse innholdet i en annen melding,  $y$ , som har samme hashverdi som den opprinnelige meldingen.

- Dersom vi bruker vanlig checksum ser vi at "IOU100.99BOB" gir samme checksum som "IOU900.19BOB", og dette kan vi ikke ha noe av i en kryptografisk hashfunksjon.

MD5 funksjonen som i 2011 ble brukt mye, regner ut en 128-bit hash i en firestegs prosess bestående av en et paddingsteg (legge til en 1 og deretter nok 0-ere til å dekke et lengdekrav), et append-steg (legg til en 64-bit representasjon av meldingslengden før padding), en initialisering av en accumulator og en siste loopingsteg der meldingens 16-ords-blokker blir prosessert i fire runder.

En annen stor hashefunksjon brukt i 2011 er Secure Hash Algorithm (SHA-1). Denne algoritmen er basert på lignende prinsipper brukt i MD4. Outputlengden i SHA-1 er lengre enn MD5, og mer sikker. I dag brukes en nyere versjon som heter SHA-256, som er enda sikrere.

## Message Authentication Code

La oss nå gå tilbake problemet med meldingsintegritet. Nå som vi har skjønt hashfunksjoner, kan vi se på hvordan man utfører meldingsintegritet.

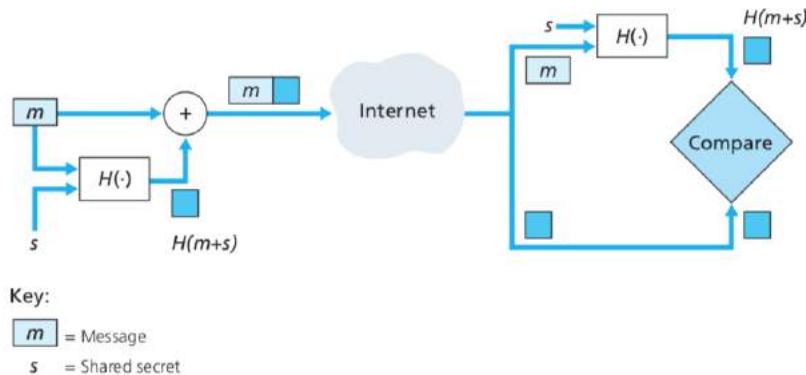
1. Alice lager en melding  $m$  utregner hashen  $H(m)$
2. Alice legger så til  $H(m)$  til meldingen  $m$ , altså lager en utvidet melding  $(m, H(m))$ , og sender denne til Bob.

3. Bob mottar den utvidede meldingen  $(m, h)$  og regner ut  $H(m)$ . Dersom  $H(m) = h$ , konkluderer Bob med at alt er bra.

For å utrøfne meldingsintegritet i tillegg til å bruke en kryptografisk hashfunksjon, trenger Alice og Bob en delt hemmelighet  $s$ . Denne delte hemmeligheten er ikke noe mer enn en streng med bits, kalt en **autentiserings nøkkel**. Ved å bruke denne delte hemmeligheten, kan meldingsintegritet utføres slik:

1. Alice lager melding  $m$ , setter sammen  $s$  med  $m$  for å opprette  $m + s$ , og beregner hashen  $H(m + s)$  (for eksempel med SHA-1).  $H(m + s)$  kalles **message authentication code-en (MAC)**.
2. Alice legger så til MAC-en til meldingen  $m$ , altså lager en utvidet melding  $(m, H(m + s))$ , og sender denne til Bob.
3. Bob mottar den utvidede meldingen  $(m, h)$ , og vet  $s$ . Han regner så ut MAC-en  $H(m + s)$ . Dersom  $H(m + s) = h$ , konkluderer Bob med at alt er bra.

Oppsummering av denne prosedyren finner du i figuren under.

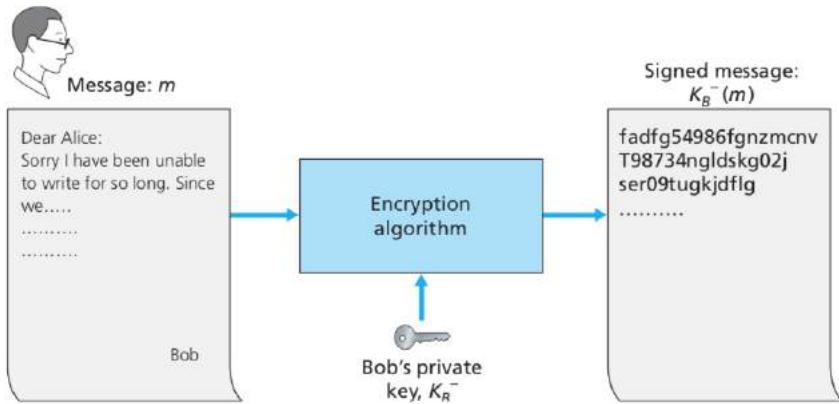


**Figure 8.9** ♦ Message authentication code (MAC)

## Digital Signatures

Vi signerer idag kontrakter, regninger, kvitteringer og brev i den ekte verden. Signaturen din atesterer at det faktisk er deg, og ingen andre, som bekrefter det du skriver under på. I en digital verden ønsker man ofte å indikere eieren eller skaperen av et dokument, eller å signere ens enighet til et dokuments innhold. En **digital signatur** er en kryptografisk teknikk for å oppnå disse kravene i den digitale verden. La oss nå nå se på hvordan vi kan designe en digital signatursskjema. Observer at når Bob signerer en meldingen, putter Bob noe i meldingen som er unik for han. Bob kan overveie å legge ved en MAC for signature, hvor MACen er laget ved å legge til nøkkelen hans (unik for han) til meldingen, får så å ta hashen. Men for at Alice skal kunne verifisere signaturen må hun også ha en kopi av nøkkelen, som gjør at nøkkelen ikke lenger er unik for Bob. Dermed kan ikke MAC-er brukes til digitale signaturer.

Husk at med offentlig-nøkkel kryptografi, har Bob både en offentlig og en privat nøkkel, der begge nøklene er unike for Bob. Anta at Bob ønsker å digital signere et dokument,  $m$ . Vi kan tenke på dokumentet som en fil eller en melding som Bob skal signere og sende. Som vist i Figur 8.10, for signere dokumentet, bruker Bob den private nøkkelen sin  $K_B^-$  for å regne ut  $K_B^-(m)$ . Det kan virke merkelig at Bob skal bruke sin private nøkkel, når vi akkurat så ble brukt for å dekryptere en melding (som hadde blitt kryptert med hans offentlige nøkkel), til å signere et dokument. Bobs digitale signatur av dokumentet er  $K_B^-(m)$ .



**Figure 8.10** ♦ Creating a digital signature for a document

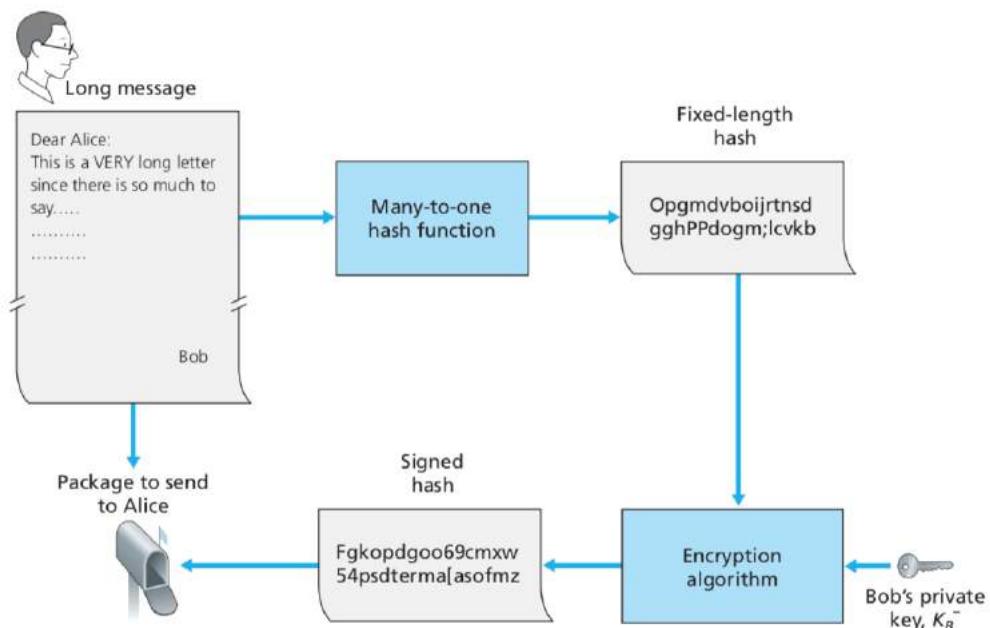
Men møter den digitale signaturen  $K_B^-(m)$  kravene våre om å være verifiserbar og ikke-forfalskbar? Anta at Alice har  $m$  og  $K_B^-(m)$ . Hun ønsker å bevise at Bob faktisk har signert dokumentet. Alice tar Bob sin offentlige nøkkelen  $K_B^+(K_B^-(m))$ , som gir  $m$ , og har dermed på følgende grunnlag for å bevise at Bob sendte meldingen:

- Den som signerte meldingen må ha den private nøkkelen,  $K_B^-$ , for å regne ut signaturen  $K_B^-(m)$ , slik at  $K_B^+(K_B^-(m)) = m$ .
- Den eneste personen som kjenner den private nøkkelen  $K_B^-$ , er Bob.

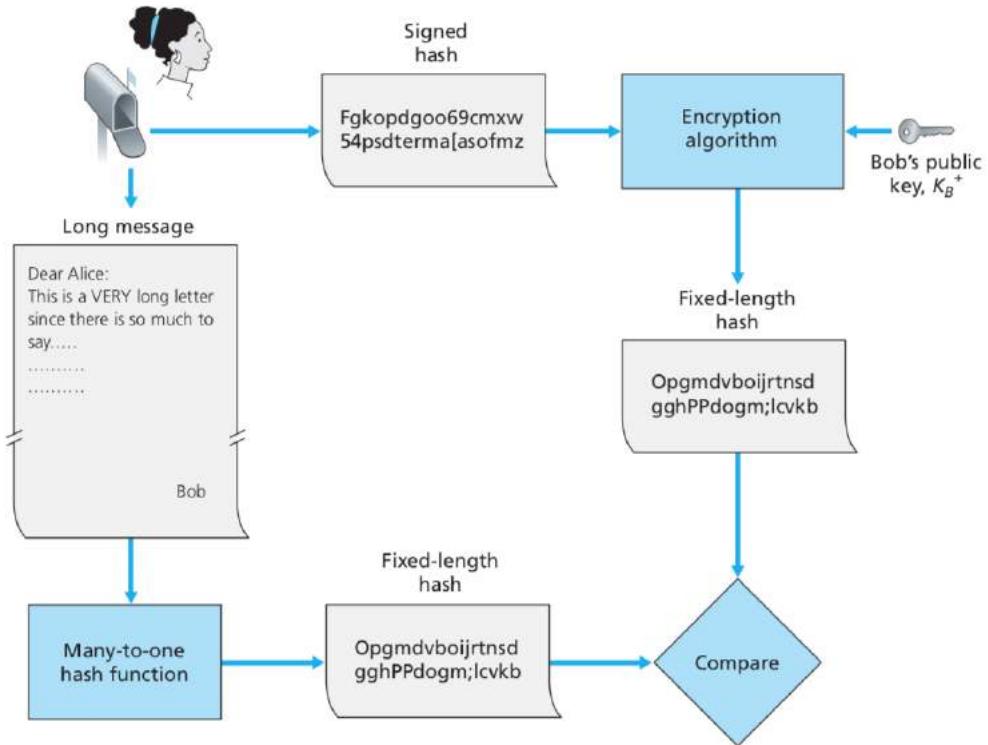
Det er også viktig å notere seg at dersom det originale dokumentet,  $m$ , blir modifisert til et alternativ,  $m'$ . Så vil ikke signaturen til Bob, lagd for  $m$ , gjelde for  $m'$ .

- En bekymring med å signere data ved kryptering er at kryptering og dekryptering er beregningsmessig dyrt. Gitt overheadene for kryptering og dekryptering kan signaturdata via fullstendig kryptering / dekryptering være overkill.
- En mer effektiv tilnærming er å introdusere hashfunksjoner i den digitale signaturen. Husk at en hashalgoritme tar en melding,  $m$ , med vilkårlig lengde og beregner et fast lengde "fingeravtrykk" av meldingen, betegnet med  $H(m)$ . Bob signerer hashen i stedet for selve meldingen, det vil si at Bob beregner  $K_B^- \cdot H(m)$ . Siden  $H(m)$  generelt er mye mindre enn den opprinnelige meldingen  $m$ , reduseres den beregningsmessige oppgaven som kreves for å skape den digitale signaturen betydelig.

Figur 8.11 gir en oppsummering av de operasjonsprosedyrene ved å lage en digital signatur.



**Figure 8.11** ♦ Sending a digitally signed message



**Figure 8.12** ♦ Verifying a signed message

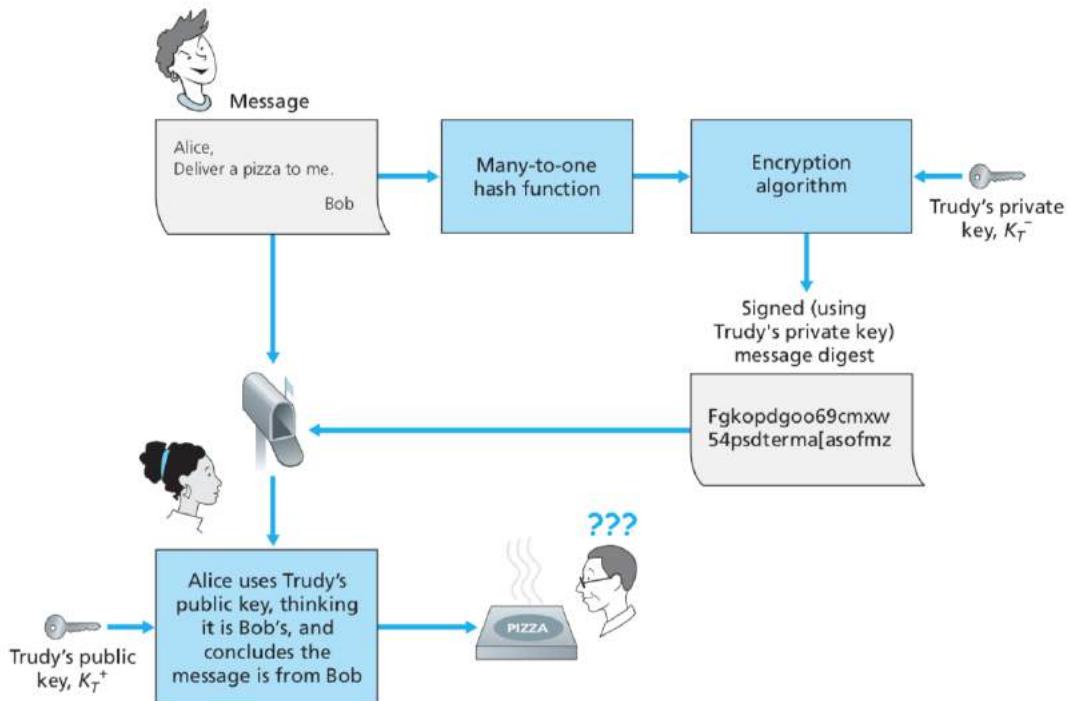
En digital signatur er en tyngre teknikk, enn MAC, da den krever en underliggende Public Key Infrastructure (PKI) med sertifiseringsautoriteter som beskrevet under.

#### Public Key Certification

En viktig anvendelse av digitale signaturer er **public key certification**, det vil si at en offentlig nøkkel tilhører en bestemt enhet. Offentlig-nøkkel-sertifisering brukes i mange populære sikre nettverksprotokoller, inkludert IPsec og SSL.

For å se på dette problemet kan vi se på en Internet-handel versjonen av den klassiske "pizza pranken". Alice jobber med pizzaleveranse og tar imot bestillinger over nett. Bob sender Alice en plaintekst-melding som inkluderer adressenhans og type pizza han ønsker. I denne meldingen inkluderer Bob også en digital signatur (det vil si en signert hash av den opprinnelige tekstmeldingen) for å bevise for Alice at han er den samme kilden til meldingen. For å verifisere meldingen bruker Alice Bob sin offentlige nøkkel og sjekker den digitale signaturen. Slik sjekker hun at det er Bob, og ikke en prankster, som har sendt ordren.

Dette er fint helt til Trudy kommer. Som vist i Figur 8.13 skal Trudy pranke Bob og Alice. Hun sender en melding til Alice, der hun sier at hun er Bob, gir Bobs hjemadresse, og bestiller en pizza. I meldingen legger hun også til hennes (Trudy's) offentlige nøkkel, og Alice antar naturlig at dette er Bobs offentlige nøkkel. Trudy legger også til en digital signatur med sin egen privat nøkkel. Når Alice mottar meldingen, bruker hun Trudys offentlige nøkkel (Tror det er Bobs) på den digitale signituren, og konkluderer med at klartekstmeldingen ble laget av Bob. Hun leverer så pizza hos Bob. What a prankster!

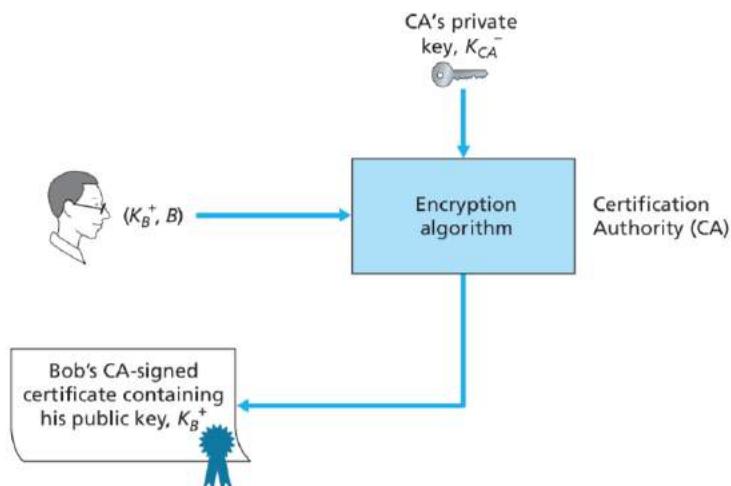


**Figure 8.13** ♦ Trudy masquerades as Bob using public key cryptography

Vi skal nå se på et eksempel at for at offentlig-nøkkel kryptografi skal være nyttig, må man kunne verifisere at man faktisk har den offentlige nøkkelen til en entitet (person, ruter, browser, osv.) til dem man kommuniserer med. For eksempel når Alice ønsker å kommunisere med Bob ved å bruke offentlig-nøkkel kryptografi, trenger hun å verifisere at den offentlige nøkkelen som tilsvarende skal være Bobs faktisk er Bobs.

Å knytte en offentlig nøkkel til en bestemt entitet er typisk gjort av en **Certification Authority (CA)**, som skal validere identiteten og gi ut sertifikater. En CA har følgende roller:

1. En CA verifiserer at en entitet er den som den sier at den er. Det er ingen bestemte metoder for hvordan sertifisering av en entitet skal gjøres. Altså kan man stole på identiteten knyttet til en offentlig nøkkel, kun i den grad man kan stole på en CA og dens identifikasjonsverifiseringsteknikker.
2. Når en CA verifiserer identiteten til entiteten, lager CA-en et **sertifikat** som binder den offentlige nøkkelen til entitetens identitet. Sertifikatet inneholder den offentlige nøkkelen og global unik identifikasjonsinformasjon om eieren til den offentlige nøkkelen (f.eks. et navn eller en IP-adresse.) Sertifikatet er så digitalt signert av CA-en.



**Figure 8.14** ♦ Bob has his public key certified by the CA

Vi skal nå se på hvordan verktøyene vi nå har sett på kan bli brukt for å tilby sikkerhet i internettet. Det er mulig å tilby sikkerhetstjenester til alle de fire øverste lagene i internettprotokollstakken. Når sikkerhet er tilbuddt av transportlaget, vil alle applikasjoner som bruker den protokollen kunne bruke sikkerhetstjenestene til transportprotokollen. Når sikkerhet er tilbuddt av nettverkslaget på en vert-til-vert-basis, vil alle transportlagssegmenter (og dermed alle applikasjonslagsdata) nytte sikkerhetstjenestene til nettverkslaget. Når sikkerhet tilbys på en linkbasis, vil all data i rammene reisenede over koblingen motta sikkerhetstjenestee til linken.

Du lurer kanskje på hvorfor sikkerhetsfunksjonalitet blir levert på mer enn ett lag på Internett. Ville det ikke bare være å gi sikkerhetsfunksjonaliteten på nettverkslaget og bli ferdig med det? Det er to svar på dette spørsmålet.

- For det første, selv om sikkerhet i nettverkslaget kan tilby "blanket coverage" (forsikring som dekker hus, men samtidig også møblene inni) ved å kryptere alle dataene i datagrammer (det vil si alle transportsegmentene) og ved å autentisere alle kilde-IP-adressene, kan det ikke gi brukernivå sikkerhet. For eksempel kan et handelsnettsted ikke stole på IP-lagssikkerhet for å godkjenne en kunde som kjøper varer på handelsstedet. Dermed er det behov for sikkerhetsfunksjonalitet ved høyere lag samt "blanket coverage" på lavere lag.
- For det andre er det generelt lettere å distribuere nye Internett-tjenester, inkludert sikkerhetstjenester, i de høyere lagene i protokollstakken.

Istedet for å vente på at sikkerhets skal bli implementert i nettverkslaget, som det nok er noen år til, vil mange applikasjonsutviklere bare gjøre det selv og introdusere sikkerhetsfunksjonalitet i applikasjonene sine. Et eksempel på dette er Pretty Good Privacy (PGP) som tilbyr sikker epost. Som kun trengte klient- og serverprogramkode var PGP en av de første sikkerhetsteknologiene som i stor grad ble brukt på Internett. Krevende kun klient- og serverprogramkode var PGP en av de første sikkerhetsteknologiene som i stor grad ble brukt på Internett.

### Secure E-Mail

Vi skal nå bruke kryptografiske prinsipper fra tidligere avsnitt for å lage et sikkert epost-system. Når vi nå designet et sikkert epostsystem, kan vi tenke oss eksempelet med kjærlighetsaffären mellom Alice og Bob. Se for deg at Alice vil sende en epost til Bob, og Trudy vil inntrenges.

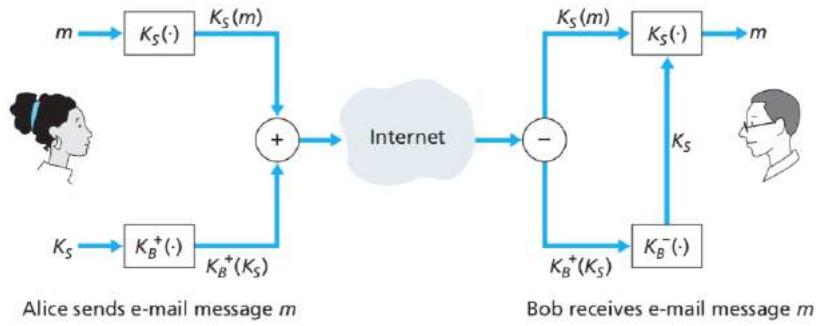
- Den første egenskapen vi vil ha **konfidensialitet** - både Alice og Bob ønsker at ingen (spesielt Trudy) skal kunne lese epostmeldingene de forveksler.
- Den andre egenskapen vi vil ha i det sikre epostsystemet vårt er **sender autentisering**. Spesielt, når Bob mottar meldingen "*I don't love you anymore. I never want to see you again. Formerly yours, Alice*", vil han naturligvis være sikker på at meldingen kom fra Alice, og ikke Trudy.
- En annen egenskap vi vil ha er **meldingsintegritet**, det vil si en forsikring på meldingen Alice sender ikke blir modifisert på veien til Bob.
- Til slutt vil vi at epostsystemet vårt skal tilby **receiver authentication**, det vil si at Alice vil være sikker at det er Bob som mottar meldingen hun sender, og ingen andre.

La oss se på det første problemet, konfidensialitet. Den mest rett frem måten å gi konfidensialitet for Alice er å kryptere meldingene sine med en symmetrisk-nøkkel kryptografi (som DES eller AES) og for Bob å dekryptere meldingen med nøkkelen. Problemet her er da å distribuere den felles nøkkelen uten at noen andre får tak i den. Derfor ser vi på en alternativ fremgangsmåte - offentlig-nøkkel kryptografi.

I offentlig-nøkkel tilnærmingen, gjør Bob den offentlige nøkkelen sin tilgjengelig, Alice krypterer meldingen sin med Bob sin offentlige nøkkel, og sender den krypterte meldingen til Bob sin epostadresse. Når Bob mottar meldingen kan han dekryptere den med sin private nøkkel. (Antatt at Alice for sikkert vet at hun har brukt Bob sin offentlige nøkkel). Et problem er fremdeles at offentlig nøkkelkryptering er relativt ueffektivt for lange meldinger.

For å overkomme dette effektivitetsproblemet, bruker vi en *session key*. Spesielt (1) velger Alice en tilfeldig symmetrisk session key,  $K_S$ , (2) krypterer meldingen sin,  $m$ , med den symmetriske nøkkelen, (3) krypterer den symmetriske nøkkelen med Bob sin offentlige nøkkel  $K_B^+$ , (4) legger sammen den krypterte meldingen og den krypterte nøkkelen i en pakke, og (5) sender denne til Bobs epostadresse. Stegene er illustrert i Figur 8.19.

Når Bob mottar pakken tar han (1) å bruke sin private nøkkel,  $K_B^-$  for å få den symmetriske nøkkelen  $K_S$ , og (2) bruker denne symmetriske nøkkelen for å dekryptere meldingen  $m$ .



**Figure 8.19** ♦ Alice used a symmetric session key,  $K_S$ , to send a secret e-mail to Bob

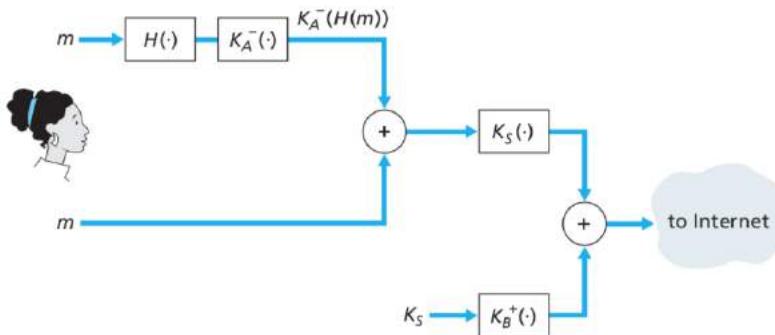
Etter å ha designet et sikkert epostsystem som tilbyr konfidensialitet, kan vi nå se på å designe et annet system som tilbyr både autentisering og meldingsintegritet. Vi antar, i et øyeblikk, at Alice og Bob ikke lenger bryr seg om konfidensialitet, og er kun bekymret for senderautentisering og meldingsintegritet. For å oppnå dette bruker vi digitale signaturer og meldingdigester. Spesifikt vil (1) Alice bruke en hashfunksjon  $H$  på meldingen sin  $m$ , for oppnå en meldingsdigest/hash  $H(m)$ , (2) signerer resultatet av hashfunksjonen med den private nøkkelen hennes,  $K_A^-$ , og lager en digital signatur, og (3) setter sammen den ukrypterte meldingen sammen med signaturen til en pakke, og (4) sender pakken til Bobs epostadresse.

Når Bob mottar pakken, vil han (1) bruke Alice sin offentlige nøkkel,  $K_A^+$ , på den signerte meldingsdigesten og (2) sammenligne resultatet med sin egen hash  $H(m)$ . Stegene er illustrert i Figur 8.20 under.



Som nevnt tidligere kan Bob, dersom de to resultatene er like, være sikker på at pakken kom fra Alice, og at den er uendret.

La oss nå se på å designe et epostsystem som tilbyr konfidensialitet, sender-autentisering, og meldingsintegritet. Dette kan bli gjort ved å kombinere prosedyrene i Figur 8.19 og 8.20. Alice lager første en innledende pakke, som i Figur 8.20, bestående av den originale meldingen sammen med en digitalt signert hash av meldingen. Hun behandler denne pakken som en melding i seg selv, og sender denne meldingen til Bob med stegene i Figur 8.19, dvs å hashe meldingen med en symmetrisk nøkkel, og sette dette resultatet sammen med den symmetriske nøkkelen kryptert med Bob sin offentlige nøkkel. Da får vi stegene vist i Figur 8.21 under.



**Figure 8.21** ♦ Alice uses symmetric key cryptography, public key cryptography, a hash function, and a digital signature to provide secrecy, sender authentication, and message integrity

Den sikre e-postdesignen som er over, gir trolig tilfredsstillende sikkerhet for de fleste e-postbrukere ved de fleste anledninger. Men det er fortsatt et viktig problem som gjenstår å bli adressert. Utformingen i Figur 8.21 krever at Alice oppnår Bobs offentlige nøkkel, og krever at Bob skal oppnå Alice's offentlige nøkkel. Fordelingen av disse offentlige nøklene er et ikke-trivielt problem.

Trudy kan fortsatt late som (*masquerade*) som hun er Bob, og gi Alice sin egen offentlige nøkkel.

En vanlig løsning for sikker distribuering av offentlige nøkler er å sertifisere de offentlige nøklene ved å bruke en CA.

## Securing TCP Connections: SSL

Vi skal nå gå ned et lag i protokollstakken og se på hvordan kryptografi kan forbedre TCP med sikkerhetstjenester, inkludert konfidensialitet, dataintegritet og endepunkts-autentikasjon. Denne forbedrede versjonen av TCP er kjent som **Secure Sockets Layer (SSL)**. En litt modifisert versjon av SSL versjon 3, kalt **Transport Layer Security (TLS)**, har blitt standardisert.

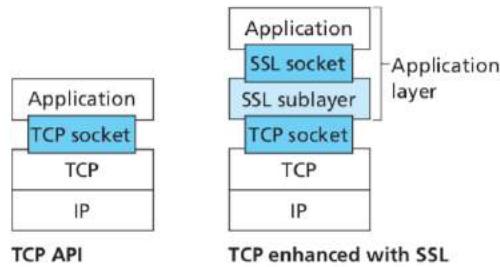
Siden utviklingen av SSL har den blitt utplassert på mange sider, og er støttet på alle populære nettlesere og webservere, og brukes essensielt av alle handelsnettsteder (inkludert Amazon, eBay, Yahoo!, osv.) Det brukes masse penger på SSL hvert år. Dersom du noen gang har kjøpt noe over internettet med bankkortet ditt, har kommunikasjonen mellom nettleseren og serveren for dette kjøpet nesten helt bestemt vært gjort over SSL.

Du kan se at du bruker SSL i nettleseren din dersom URL-en begynner på *https* istedet for *http*.

La oss ta for oss en kjøp på netthandelen, og se på hva man må ta hensyn til:

- Dersom ingen konfidensialitet (kryptering) blir brukt, kan en inntrænger fange Bobs ordre, og få tak i bankinformasjonen hans. En inntrænger kan da gjøre kjøp på Bobs vegne.
- Dersom ingen dataintegritet blir brukt, kan en inntrænger modifisere Bobs ordre, og få han til å kjøpe 10 ganger så mange enheter som han i utgangspunktet bestilte.
- Til slutt, dersom ingen server autentisering blir brukt, kan serveren vise Alice Incorporated sin kjente logo, når det faktisk er en side vedlikeholdt av Trudy, som har maskert seg som Alice Incorporated. Etter å ha mottatt Bobs ordre, kan Trudy ta Bob sine penger å stikke. Eller så kan Trudy kjøre et identitetstyveri ved å ta Bob sitt navn, adresse og bankkortnummer.

**SSL adresserer disse problemene ved å forbedre TCP med konfidensialitet, dataintegritet, server- og klientautentisering.** SSL blir ofte brukt for å tilby sikkerhet til transaksjoner som skal finne sted over HTTP. Men ettersom SSL sikrer TCP, kan det bli brukt av enhver applikasjon som kjører over TCP. SSL tilbyr en enkel Application Programmer Interface (API) med sockets, som ligner på TCPs API. Når en applikasjon ønsker å bruke SSL, må applikasjonen inkludere SSL klasser/biblioteker. Som vist i figur 8.24, selv om SSL teknisk sett ligger i applikasjonslaget, er det fra utviklerens perspektiv en transportprotokoll som gir TCPs tjenester forbedret med sikkerhetstjenester.



**Figure 8.24** • Although SSL technically resides in the application layer, from the developer's perspective it is a transport-layer protocol

## The Big Picture

Jeg starter med å beskrive en forenklet versjon av SSL, som lar oss se det store bildet i *hvorfor-en* og *hvordan-en* til SSL. Vi kaller denne *Nesten-SSL*. Nesten-SSL har tre faser: *handshake*, *key derivation* og *dataoverføring*. Vi skal nå se på disse tre fasene for kommunikasjonsøkter mellom en klient (Bob) og en server (Alice), der Alice har et private/offentlig nøkkelpar, og et sertifikat som binder hennes identitet til den offentlige nøkkelen henne.

### Handshake

Under handshakefasen, må Bob (a) etablere en TCP-tilkobling til Alice, (b) verifisere at Alice faktisk er Alice, og (c) sende Alice en "master secret key" som brukes av både Alice og Bob for å generere alle symmetriske nøkler de trenger for SSL-økten. Disse tre stegene vises under.

Noter at når TCP-tilkoblingen er etablert, sender Bob en hello melding til Alice. Alice responderer så med sertifikatet sitt, som inneholder den offentlige nøkkelen hennes. Bob genererer så en Master Secret (MS), krypterer MS-en med Alice sin offentlige nøkkel for å lage Encrypted Master Secret (EMS) og sender EMS-en til Alice. Alice dekrypterer EMS med den private nøkkelen sin for å få MS. Etter denne fasen ved både Bob og Alice (og ingen andre) master secret-en for SSL-økten.

#### Key Derivation

I prinsippet kan MS, som nå deles av Bob og Alice, brukes som den symmetriske økt nøkkelen for all etterfølgende kryptering og dataintegritetskontroll. Det er imidlertid generelt ansett sikrere for Alice og Bob å bruke forskjellige kryptografiske nøkler, og også å bruke forskjellige nøkler for kryptering og integritetskontroll. Dermed bruker både Alice og Bob MS til å generere fire nøkler:

- $E_B$  = øktens krypteringsnøkkelen for data sendt fra Bob til Alice
- $E_B$  = øktens MAC-nøkkelen for data sendt fra Bob til Alice
- $E_A$  = øktens krypteringsnøkkelen for data sendt fra Alice til Bob
- $M_A$  = øktens MAC-nøkkelen for data sendt fra Alice til Bob.

Alice og Bob genererer de fire nøklene fra MS. Dette kan bli gjort av å dele MS-en i fire nøkler (Men i ekte SSL, gjøres det litt mer komplisert).

#### Data Transfer

Nå som Alice og Bob deler de samme fire økt-nøklene ( $E_B$ ,  $E_B$ ,  $E_A$  og  $M_A$ ), kan de starte å sende sikret data til hverandre over TCP-tilkoblingen. SSL deler datastrømmen inn i *records*, legger til en MAC til hver record for integritetssjekking, og krypterer denne *record+MAC*-en. For å lage MAC-en, gir Bob record-dataen sammen med nøkkelen  $M_B$  inn i en hashfunksjon. For å kryptere pakken *record+MAC*, bruker Bob sin økt-krypteringsnøkkelen  $E_B$ . Den krypterte pakken sendes så over TCP for transport over nettet.

Vi er nå kommet en lang vei, men har fortsatt ikke sikret dataintegritet for hele meldingsstrømmen. Antatt at hvert TCP-segment innkapsles nøyaktig i en record, la oss se hvordan Alice prosesserer disse segmentene. La oss anta at Bob sender to segmenter til Alice, og at Trudy har tuklet med rekkefølgen til segmentene, TCP-sekvensnumrene til de to segmentene:

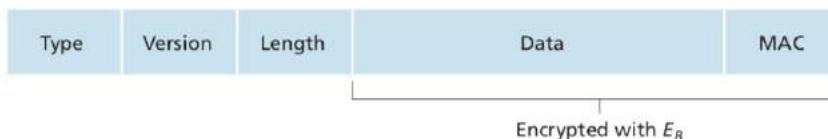
1. TCP kjørende hos Alice vil tro at alt er fint, og gi de to recordene til SSL-sublaget.
2. SSL hos Alice vil dekryptere de to recordene
3. SSL hos Alice vil bruke MAC-en i hver record for å verifisere dataintegriteten til hver av de to recordene.
4. SSL vil så gi den dekrypterte bytestrømmen av de to recordene til applikasjonslaget - men vil være i feil rekkefølge!

Et lignende scenario ville vært når Trudy fjerner et segment, eller erstatter et segment.

Løsningen på dette er å bruke sekvensnumre. SSL gjør det som følger. Bob vedligeholder en sekvensnummer-teller, som begynner på null, og inkrementerer for hver SSL-record han sender. Bob legger ikke ved et sekvensnummer i recorden selv, men når han regner ut MAC-en, inkluderer han sekvensnummere i utregningen av MAC-en. MAC-en er nå *dataen + MAC-nøkkelen  $K_B$  + det gjeldende sekvensnummeret*.

#### SSL Record

SSL-recorden er vist i Figur 8.26. Denne recorden består av et typefelt, lengdefelt, datafelt og MAC-felt. Noter at de første tre feltene er ukrypterte. Typefeltet indikerer om recorder er en handshake-melding eller en melding som inneholder applikasjonsdata. Den brukes også for å lukke en SSL-tilkobling. SSL hos den mottakende enden bruker lengdefeltet for å ekstrahere SSL-recordene fra den innkommende TCP-bytestrømmen. Versjonfeltet er selvforklarende.



**Figure 8.26** • Record format for SSL

#### A More Complete Picture

I forrige avsnitt så vi på nesten-SSL, nå skal vi se på hvordan SSL fungerer.

## SSL Handshake

SSL bestemmer ikke at Alice og Bob skal bruke en spesifikk symmetrisk nøkkel-algoritme, en spesifikk offentlig nøkkel-algoritme eller en spesifikk MAC. Istedet lar SSL Alice og Bob bli enige om den kryptografiske algoritmen på begynnelsen av SSL-økten under handshake-fasen. I tillegg, sender Alice og Bob hverandre engangsord til hverandre under handshaken, som brukes i lagingen av økt-nøklene ( $E_B$ ,  $E_B$ ,  $E_A$  og  $M_A$ ). Stegene i den ekte SSL-handshaken er som følgende:

1. Klienten sender en liste med kryptografiske algoritmer som den støtter, sammen med et klient-engangsord.
2. Fra listen velger serveren en symmetrisk algoritme, en offentlig nøkkel-algoritme (f.eks. RSA), og en MAC-algoritme. Den sender valgene sine tilbake til klienten, sammen med et sertifikat og et server-engangsord.
3. Klienten verifiserer sertifikatet, henter serverens offentlige nøkkel, genererer en Pre-Master Secret (PMS), krypterer PMS-en med serverens offentlige nøkkel, og sender den krypterte PMS-en til serveren.
4. Ved å bruke samme nøkkel-derivationsfunksjon (spesifisert av SSL-standarden) kan klienten og serveren uavhengig av hverandre lage Master Secret (MS) fra PMS-en og engangsordene. MS-en blir så delt opp i generere to krypterings- og to MAC-nøkler. Når den brukte symmetriske cipheren bruker CBC, da vil to Initialization Vectors (IVs) - en for hver side av tilkoblingen - fås fra MS-en. Herfra, er meldinger sendt mellom klienten og serveren er kryptert og autentisert.
5. Klienten sender en MAC av alle handshake-meldingene
6. Serveren sender en MAC av alle handshake-meldingene

De siste to stegene forsvarer handshaken fra å bli tuklet med. Listen i første meldingen kan bestå av både svake og sterke algoritmer, og dersom Trudy hadde slettet de sterke i listen, ville hun tvunget klienten til å velge en svak algoritme. Derfor sendes MAC-ene, så serveren kan se etter inkonsistens, samme gjelder klienten.

- De tilfeldige engangsordene brukes for at ikke alle SSL-økter skal være like. Dette gjøres pga. av dersom Trudy har sniffet hele handshaken mellom Bob og Alice, kunne hun bare ha sendt de samme pakkene dagen etterpå og utgitt seg for å være Bob - "*connection replay attack*". Men de tilfeldige engangsordene gjør at hver SSL-økt er forskjellig.

## Connection Closure

På et tidspunkt vil Bob eller Alice ønsker å avslutte SSL-økten. En tilnærming ville være å la Bob avslutte SSL-økten ved å bare avslutte den underliggende TCP-tilkoblingen, det vil si at Bob sender et TCP FIN-segment til Alice. Men en slik naiv design setter scenen for *trunkeringsangrepet*, hvor Trudy igjen kommer i midten av en pågående SSL-økt, og avslutter økten tidlig med et TCP FIN. Hvis Trudy skulle gjøre dette, ville Alice tro at hun fikk alle Bobs data da hun bare fikk en del av det.

Løsningen på dette problemet er å i typefeltet angi hvorvidt recorden har som oppgave å avslutte SSL-økten. (Selv om SSL-typen sendes i det åpne, er den autentisert ved mottakeren ved bruk av recordens MAC). Ved å inkludere et slikt felt, hvis Alice skulle motta en TCP FIN før den mottok en lukkende-SSL-record,, ville hun vite at noe muffings var på gang.

## Network-Layer Security: IPsec and Virtual Private Networks

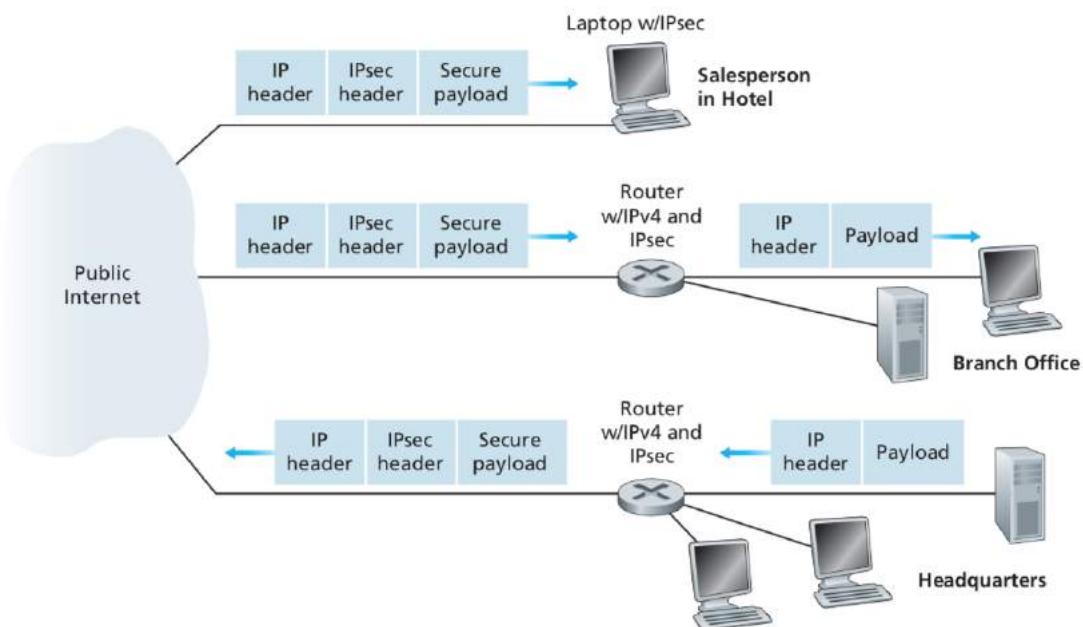
Som vi skal se bruker mange institusjoner IPsec for å lage **virtuelle private nettverk (VPNs)** som går over det offentlige nettverket. før vi ser på IPsec la oss se på hva det betyr å tilby konfidensialitet på nettverkslaget. Med nettverlagskonfidensialitet mellom et par nettverksentiteter, vil den sendende entiteten kryptere alle payloaden i alle datagrammer som skal bli sendt til den mottakende entiteten. Den krypterte payloaden kan være et TCP-segment, UDP-segment eller en ICMP-melding. Hvis en slik nettverkslagstjeneste var på plass, ville alle data sendt fra en enhet til den andre, inkludert e-post, websider, TCP-håndtrykkmeldinger og administrasjonsmeldinger (som ICMP og SNMP) - være skjult for ethvert tredjeparti som skulle ønske å sniffe nettverket. Av denne grunn er sies det nettverkslagsikkerhet gir "blanket coverage".

I tillegg til konfidensialitet kan en nettverkslags-sikkerhetsprotokoll potensielt tilby andre sikkerhetstjenester. For eksempel kan det gi kildeautentisering. En nettverlags-sikkerhetsprotokoll kan gi dataintegritet. En sikkerhetstjeneste for nettverkslag kan også gi *replay-attack*-forebygging, noe som betyr at Bob kunne ha oppdaget duplikatdatagrammer som en angriper setter inn. Vi vil snart se at IPsec faktisk tilbyr mekanismer, for alle disse sikkerhetstjenestene, det vil si konfidensialitet, kildeautentisering, dataintegritet, og *replay-attack*-forebygging.

## IPsec and Virtual Private Networks (VPNs)

En institusjon som strekker seg over flere geografiske områder ønsker ofte egne IP-nettverk, slik at dens vever og servere kan sende data til hverandre på en sikker og konfidensiell måte. For å oppnå dette, må institusjonen sette opp et stand-alone fysisk nettverk - inkludert rutere, koblinger og en DNS infrastruktur - som er helt separat fra det offentlige internettet. Et slikt disjunkt internett, dedikert til en spesiell institusjon kalles et **privat nettverk**. Ikke overraskende er et privat nettverk dyrt, og institusjonen må kjøpe, installere og vedlikeholde sin egen fysiske nettverksstruktur.

I stedet for å distribuere og vedlikeholde et privat nettverk, oppretter mange institusjoner i dag VPN-er over det eksisterende Internett-området. Med en **VPN** sendes institusjonens "interoffice" trafikk over det offentlige Internett i stedet for over et fysisk uavhengig nettverk. Men for å gi konfidensialitet, krypteres inter-office trafikken før den går inn på det offentlige Internett. Et enkelt eksempel på et VPN er vist i Figur 8.27.



**Figure 8.27** • Virtual Private Network (VPN)

Her består institusjonen av et hovedkvarter, et avdelingskontor og reisende salgspersoner som vanligvis har tilgang til Internett fra hotellrommene. (Det er bare én selger som er vist på figuren.)

I denne VPN, når to verter i hovedkvarteret sender IP datagrammer til hverandre, eller når to verter innenfor avdelingskontoret ønsker å kommunisere, bruker de god gammel vanilla IPv4 (dvs. uten IPsec-tjenester). Når to av institusjonens verter kommuniserer over en sti som går gjennom det offentlige Internett, krypteres trafikken før den kommer inn på Internett.

For å få en følelse av hvordan et VPN fungerer, la oss gå gjennom et enkelt eksempel i sammenheng med figur 8.27. Når en vert i hovedkvarteret sender et IP-datagram til en selger på et hotell, konverterer gateway-ruteren i hovedkvarteret vanilla-IPv4-datagrammet til et IPsec datagram og deretter videresender dette IPsec-datagrammet til Internettet. Dette IPsec-datagrammet har faktisk en tradisjonell IPv4-header, slik at ruterne i det offentlige Internett behandler datagrammet som om det var et vanlig IPv4-datagram - for dem er datagrammet et helt vanlig datagram.

Men, som vist Figur 8.27, inneholder payloaden til IPsec-datagrammet en IPsec-header, som brukes til IPsec-prosessering; Dessuten er nyttelastet til IPsec datagrammet kryptert. Når IPsec datagrammet kommer til selgerens bærbare, dekrypterer operativsystemet i den bærbare datamaskinen payloaden (og gir andre sikkerhetstjenester, for eksempel verifisering av dataintegritet) og sender den ukrypterte nyttelastet til den øvre-lag protokollen (for eksempel til TCP eller UDP).

Vi har nettopp gitt en oversikt over hvordan en institusjon kan bruke IPsec til å opprette en VPN.

## Securing Wireless LANs

Sikkerhetsspørsmålet i 802.11 har fått stor oppmerksomhet både i tekniske kretser og i media. Selv om det har vært betydelig diskusjon, har det vært liten debatt - det synes å være universell enighet om at den opprinnelige 802.11-spesifikasjonen inneholder en rekke alvorlige sikkerhetsfeil. Faktisk kan offentlig programvare bli lastet ned som utnytter disse hullene, slik at de som bruker sikkerhetsmekanismene til vanilla 802.11 er like åpne, for sikkerhetsangrep, som brukere som ikke bruker noen sikkerhetsfunksjoner i det hele tatt.

Vi skal nå se på sikkerhetsmekanismen kjent som **Wired Equivalent Privacy (WEP)**. Som navnet henter til, er det ment at WEP skal tilby et nivå av sikkerhet lignende som det vi finner i kablede nettverk. Vi skal snart se på sikkerhetshullene i WEP.

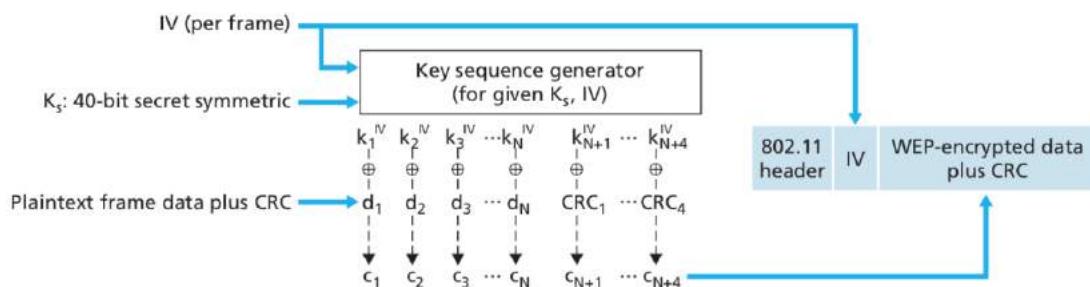
### Wired Equivalent Privacy (WEP)

IEEE 802.11 WEP-protokollen ble designet i 1999 for å tilby autentisering og datakryptering mellom en vert og et trådløst aksesspunkt (dvs. basestasjon) ved å bruke en symmetrisk delt nøkkels-tilnærming. WEP spesifiserer ikke en nøkkeladministreringsalgoritme, så det er antatt at verten og AP-en har på en måte blitt enige om en nøkkelen via en out-of-band-metode. Autentikasjonen skjer slik:

1. En trådløs vert foresprø autentisering av et aksesspunkt
2. Aksesspunktet responderer til autentiseringsforespørrelsen med et 128-byte engangsord.
3. Den trådløse verten krypterer engangsordet med den symmetriske nøkkelen som den deler med aksesspunktet.
4. Aksesspunktet dekrypterer vertens krypterte engangsord.

Dersom det dekrypterte engangsordet matcher engangsordet som originalt ble sendt, da er verten autentisert av aksesspunktet.

WEP-datakrypteringsalgoritmen er illustrert i Figur 8.30. En hemmelig 40-bit symmetrisk nøkkelen,  $K_S$ , er antatt å bære kjent av både verten og AP-en. I tillegg, en 24-bit IV (Initialization vector) er lagt på den 40-bit nøkkelen for å lage en 64-bit nøkkelen som skal krypteres i en enkelt ramme. IV-en vil endre seg fra en ramme til en annen, og dermed har hver ramme forskjellige 64-bit nøkler.



**Figure 8.30** • 802.11 WEP protocol

Kryptering skjer slik:

- En 4-byte CRC-verdi blir laget for datapayloaden. Databayloaden og de 4 CRC-bytene er så kryptert med RC4-stream cipheren. Det er nok å vite at RC4 algoritmen kun produserer en strøm med nøkkerverdier,  $k_1^{IV}, k_2^{IV}, k_3^{IV}, \dots$  som blir brukt for å kryptere dataen og CRC-verdiene i en ramme.
- Av praktiske grunner kan vi se for oss at operasjonene blir utført byte for byte. Krypteringen skjer ved å XOR-e den  $i$ te byten med data,  $d_i$ , med den  $i$ te nøkkelen,  $k_i^{IV}$ , i strømmen av nøkkerverdien generert av ( $K_S$ , IV)-paret for å produsere den  $i$ te byten med ciphertekst,  $c_i$ :

$$c_i = d_i \oplus k_i^{IV}$$

IV-verdien endres fra ramme til ramme, og er inkludert i *klartekst* i headeren til hver WEP-krypterte 802.11 ramme, som vist i Figur 8.30. Mottakeren bruker den hemmelige 40-bit symmetriske nøkkelen, legger til IV-en, og får den 64-bit nøkkelen og dekrypterer rammen:

$$d_i = c_i \oplus k_i^{IV}$$

- Riktig bruk av RC4 algoritmen krever at den samme 64-bit økkelen *aldrig* brukes mer enn en gang. Men husk at en WEP-nøkkelen endrer seg fra ramme til ramme. For en gitt  $K_S$ , betyr det at det kun er  $2^{24}$  unike nøkler. Dersom disse nøklene er valgt tilfeldig, er sannsynligheten for å ha valgt like IV-verdier (og dermed samme 64-bit nøkkelen) mer enn 99% etter kun 12,000 rammer. Med 1Kbyte rammer og en overføringsrate på 11Mbps, tar det kun et par sekunder for å sende 12,000 rammer.

- Dersom Trudy eavesdropper og ser at samme  $k_i^{IV}$  brukes flere ganger kan hun bruke de kjente verdiene for  $d_i$  og  $c_i$  (funnet med XOR som over) for å regne ut  $k_i^{IV}$ . Neste gang Trudy ser at samme IV blir brukt, vet hun nøkkelsekvensen  $1^{IV}, k_2^{IV}, k_3^{IV}, \dots$ , og kan dermed dekryptere den krypterte meldingen.

Det er også andre sikkerhetsbekymringer med WEP. Dersom visse svake nøkler i RC4, så vil det være mulig å angripe. WEP involverer også CRC bits. Dersom Trudy endrer meldingen, og regner om CRC-summen på nytt og bytter den ut med den gamle CRC-en vil mottakeren ta imot pakken og tro at alt er bra. \*Mangler meldingsintegritet.

## Operational Security: Firewalls and Intrusion Detection Systems

Når trafikk ankommer/forlater et nettverk blir sikkerhetsskjekket, logget, droppet og videresendt, blir det gjort av operasjonelle enheter kjent som firewalls/brannmurer, instrusion detection systems (IDSs), og instrusion prevention systems (IPSSs)

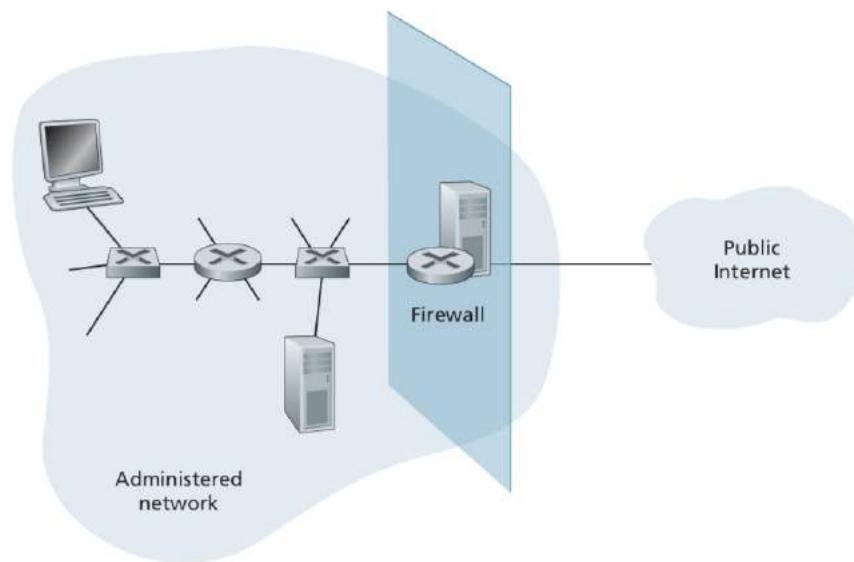
### Firewalls

En brannmur er en kombinasjon av maskinvare og programvare som isolererer en organisasjons interne nettverk fra internettet, som tillater noen pakker å passere og andre blokkeres. En brannvegg tillater en nettverksadministrator å kontrollere assessene mellom den ytre verden og ressursene i nettverket, ved å styre trafikkflyten til og fra disse ressursene. En brannmyr har tre mål:

- *All traffik fra utsiden til insiden, ig vice versa, passerer gjennom brannmuren.* Figur 8.33 viser en brannmur som sitter rett på grensen mellom administrerte nettverket og resten av Internettet.
- *Kun autorisert traffik, som definert av den lokale sikkerhetspolicyen, vil bli tillatt å passere.* Ettersom all trafikk går igjennom brannmuren, kan brannmuren begrense aksess til autorisert traffik.
- *Brannmuren selv må være immun mot penetrering.* Brannmuren selv er en enhet koblet til nettverket.

Cisco og Check Point er de to ledende brannmur-leverandørene i dag. Man kan også lett lage en brannmur (packet filter) fra en Linux box ved å bruke iptables (software fra Linux).

Brannmurer er klassifisert i tre kategorier: **traditional packet filters**, **stateful filters** og **application gateways**.



**Figure 8.33** ♦ Firewall placement between the administered network and the outside world

### Traditional Packet Filters

Som vist i figur 8.33 har en organisasjon typisk en gateway ruter som kobler det interne internettet til ISP-en. All traffik som forlater og ankommer det interne internettet passerer denne ruten, og det er i denne ruten hvor **pakke filtering** skjer. Et pakkefilter studerer hvert datagram i isolasjon, og bestemmer om datagrammet skal få lov å passere eller skal droppes. Filtreringsavgjørelser er typisk basert på:

- IP kilde eller destinasjonsadresse
- Protokoltype i IP-datagramfeltet: TCP, UDP, ICMP, OSPF, osv.

- TCP eller UDP filde og destinasjonsport
- TCP flagbits: SYN ACK, osv.
- ICMP meldingstype
- Forskjellige regler for datagrammer som ankommer eller forlater nettverket
- Forskjellige regler for de forskjellige ruterinterfacene.

En nettverksadministrator konfigurerer en brannmur basert på policyen til organisasjonen. Tabell 8.5 lister en rekke mulige policies som en organisasjon kan ha:

| Policy   | Firewall Setting  |
|--|---|
| No outside Web access.   | Drop all outgoing packets to any IP address, port 80                            |
| No incoming TCP connections, except those for organization's public Web server only. | Drop all incoming TCP SYN packets to any IP except 130.207.244.203, port 80     |
| Prevent Web-radios from eating up the available bandwidth.                           | Drop all incoming UDP packets—except DNS packets.                               |
| Prevent your network from being used for a smurf DoS attack.                         | Drop all ICMP ping packets going to a "broadcast" address (eg 130.207.255.255). |
| Prevent your network from being tracerouted  | Drop all outgoing ICMP TTL expired traffic                                      |

**Table 8.5** Policies and corresponding filtering rules for an organization's network 130.27/16 with Web server at 130.207.244.203

Brannmur-regler er implementert i ruterere med aksesskontrolllister, der hvert rutergrensesnitt har en egen liste.

| action | source address       | dest address         | protocol | source port | dest port | flag bit |
|--------|----------------------|----------------------|----------|-------------|-----------|----------|
| allow  | 222.22/16            | outside of 222.22/16 | TCP      | > 1023      | 80        | any      |
| allow  | outside of 222.22/16 | 222.22/16            | TCP      | 80          | > 1023    | ACK      |
| allow  | 222.22/16            | outside of 222.22/16 | UDP      | > 1023      | 53        | —        |
| allow  | outside of 222.22/16 | 222.22/16            | UDP      | 53          | > 1023    | —        |
| deny   | all                  | all                  | all      | all         | all       | all      |

**Table 8.6** An access control list for a router interface

### Stateful Packet Filters

I et tradisjonelt pakkefilter, gjøres filtreringsbeslutninger på hver pakke i isolasjon. Stateful filters tracker faktisk TCP-tilkoblinger, og bruker denne kunskaper til filtreringsbeslutninger.

For å forstå stateful-filtre, la vi undersøke tilgangskontrollisten i tabell 8.6. Selv om det er ganske restriktivt, tillater tilgangskontrollisten i tabell 8.6 likevel at enhver pakke som kommer fra utsiden med ACK = 1 og kildeport 80 for å komme gjennom filteret. Slike pakker kan brukes av angripere i forsøk på å krasje interne systemer med misdannede pakker, utføre DoS-angrep eller kartlegge internettet.

Den naive løsningen er å blokkere TCP ACK-pakker også, men en slik tilnærming vil hindre at organisasjonens interne brukere surfer på nettet. Stateful filtre løser dette problemet ved å spore alle pågående TCP-tilkoblinger i en tilkoblingstabell. Dette er mulig fordi brannmuren kan observere begynnelsen på en ny tilkobling ved å observere en treveishåndtrykk (SYN, SYNACK og ACK); og det kan observere slutten av en forbindelse når det ser en FIN-pakke for tilkoblingen. B

Brannmuren kan også (konservativt) anta at forbindelsen er over når den ikke har sett noen aktivitet over forbindelsen, for eksempel 60 sekunder.

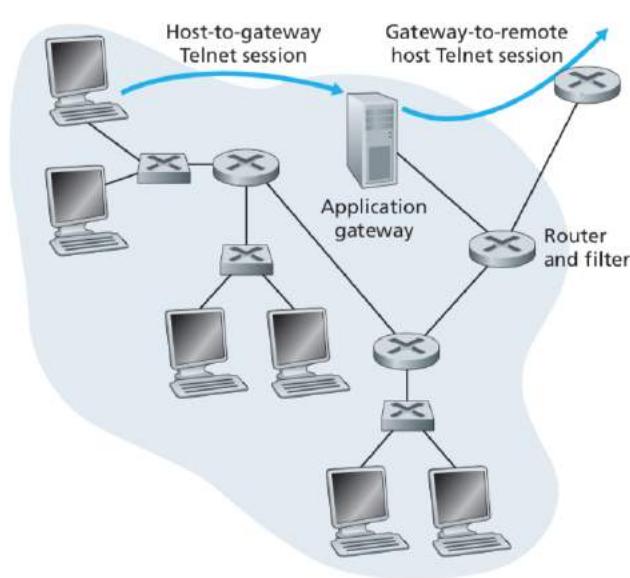
I tillegg inneholder stateful-filteret en ny kolonne, "check connection", i tilgangskontrolllisten, som vist i tabell 8.8. Merk at tabell 8.8 er identisk med tilgangskontrollisten i tabell 8.6, bortsett fra nå angir det at tilkoblingen skal kontrolleres for to av reglene.

| action | source address       | dest address         | protocol | source port | dest port | flag bit | check connexion |
|--------|----------------------|----------------------|----------|-------------|-----------|----------|-----------------|
| allow  | 222.22/16            | outside of 222.22/16 | TCP      | >1023       | 80        | any      |                 |
| allow  | outside of 222.22/16 | 222.22/16            | TCP      | 80          | >1023     | ACK      | X               |
| allow  | 222.22/16            | outside of 222.22/16 | UDP      | >1023       | 53        | —        |                 |
| allow  | outside of 222.22/16 | 222.22/16            | UDP      | 53          | >1023     | —        | X               |
| deny   | all                  | all                  | all      | all         | all       | all      |                 |

**Table 8.8** ♦ Access control list for stateful filter

### Application Gate

For å få et finere-nivås sikkerhet, må brannmurer kombinere pakkefiltrering med applikasjons gateways. Applikasjons gatewayer ser forbi IP/TCP/UDP headerne og gjør policy-beslutninger basert på applikasjonsdata. En **applikasjons gateway** er en applikasjonsspesifik server der all applikasjonsdata må igjennom. Flere applikasjonsgatewayer kan kjøre på samme vert, men hver gateway er en separat server med sine egne prosesser.



**Figure 8.34** ♦ Firewall consisting of an application gateway and a filter

For å få noen innsikt i applikasjons gatewayer, la oss designe en brannmur som tillater kun et begrenset sett av interne brukere til å Telnete til utsiden og forhindre alle eksterne klienter å Telnete inn.

En slik policy kan oppnås ved å implementere en kombinasjon av et pakkefilter (i en ruter) og en Telnet-applikasjonsgateway, som vist i figur 8.34.

- Ruterenes filter er konfigurert til å blokkere alle Telnet-tilkoblinger unntatt de som kommer fra IP-adressen til applikasjonsgatewayen. En slik filterkonfigurasjon tvinger alle utgående Telnet-tilkoblinger til å passere gjennom applikasjonsgatewayen.
- Tenk nå en intern bruker som ønsker å Telnete til omverdenen. Brukeren må først opprette en Telnet-økt med applikasjonsgatewayen. Et program som kjører i gatewayen, som lytter etter innkommende Telnet-økter, spør brukeren om brukernavn og passord. Når brukeren forsyner denne informasjonen, sjekker programgatewayen for å se om

brukeren har tillatelse til Telnet til omverdenen. Hvis ikke, blir Telnet-tilkoblingen fra den interne brukeren til gatewayen avsluttet av gatewayen. Hvis brukeren har tillatelse, så vil gatewayen:

- i. Spørre brukeren om verternavnet til den eksterne verten som brukeren vil koble til.
- ii. Sette opp en Telnet-økt mellom gatewayen og den eksterne verten
- iii. være en relé som reléer all data som kommer fra brukeren til den eksterne verten, og reléer til brukeren alle data som kommer fra den eksterne verten.

Dermed utfører Telnet-applikasjonsgatewayen ikke bare brukerautorisasjon, men fungerer også som en Telnet-server og en Telnet-klient, når den reléer informasjon mellom brukeren og den eksterne Telnet-serveren.

Merk at filteret tillater trinn 2 fordi gatewayen initierer Telnet-tilkoblingen til omverdenen.

Interne nettverk har ofte flere applikasjonsgateways, for eksempel gateways for Telnet, HTTP, FTP og e-post.

Applikasjonsgateways kommer ikke uten sine ulemper:

- For det første er det nødvendig med en annen applikasjonsgateway for hver applikasjon.
- For det andre er det en ytelsesstraff som skal betales, da alle data vil bli videresendt via gatewayen. Dette blir en bekymring, spesielt når flere brukere eller applikasjoner bruker samme gateway-maskin.
- Endelig må klientprogramvaren vite hvordan man skal kommunisere med gatewayen når brukeren gjør en forespørsel, og må vite hvordan å fortelle applikasjonsgatewayen hvilken ekstern server den skal kobles til.

Skrevet av Henrik Høiness

---