

Prossesser:

Prosess vs Tråd:

- Process krever context switch (her brukes CPU sykler for å overføre context til prosessen)
- Med tråder deler “de kjørbare enhetene”/trådene minneområde og annen global data som gjør at de slipper mye context switching. NB: De har fortsatt data lokalt for hver tråd.
- Endring fra ekskvering av en tråd til en annen er mye billigere enn fra en prosess til en annen.

Forking:

- Hvis fork() returnerer noe annet enn 0 betyr det at det er en child-prosess
- wait() venter på at noen av barneprosessene er ferdige med sin kjøring
- execve() redigerer kjøringen av prosessen til et annet program. Kode etter en execve() vil ikke utføres ettersom execve() ikke returnerer tilbake til der funksjonen ble kalt fra.
- Rekkefølgen et program utføres i ved bruk av fork() kan være litt tilfeldig (avhenger av scheduler).

CPU scheduling:

- Vi ønsker å schedule en prosess, tråd, jobb, e.l.
- Utfordring - mange ønsker tilgang til prosessen, dette håndterer scheduleren hvor det finnes mange forskjellige algoritmer. Prosesser har ofte også ulike prioriteringer.
- Dispatcher: litt synonymt med schedulerer. Scheduler velger hvem som får tilgang til prosessor, dispatcher utfører handlingen av å tildele dette.
- Scheduler må optimalisere CPU utilisering.
- Prosesser er ofte flaskehalsset av enten CPU(CPU bound) eller IO(IO bound). CPU bound betyr at prosessen gjør mye kalkulasjoner på CPU med lite ventetid for IO, mens IO bound er motsatt.
- Hvordan skal vi sortere et sett av prosesser, der noen er CPU bundet og noen er IO bundet?
 - Kjøre en type først og en annen etterpå er ueffektivt siden da er enten CPU eller disk ubrukt i lange perioder. (CPU ubrukt ved eksekvering av IO bundede prosesser, og disk ubrukt for eksekvering av CPU bundede prosesser)
 - En blandet rekkefølge av CPU bundede og IO bundede prosesser er foretrukket.

Algoritmer for scheduling:

- FIFO (first in first out): Kjør prosesser til de er ferdige i en kø
 - simpel
 - rettferdig?
 - veldig litt overhead
 - lang ventetid og avslutningstid
- SJF (shortest job first): Sorter prosesser etter kostnad og kjør dem i kø
 - simpel
 - bedre gjennomsnittlige tider enn FIFO
 - vanskelig å bestemme kostnad på forhånd
 - muligens lange avslutningstider og utsultning hvis nye korte jobber ankommer.
- RR (round robin): Dette er en variant av FIFO der vi deler opp prosesser i mindre tidsbiter (f.eks 10ms) sånn at vi unngår mange av problemene med FIFO.
 - alle programmer får lov til å kjøre (alle prosesser får tildelt tid med en gang)
 - mange context switches kan gjøre at dette tar lengre tid totalt
 - ingen prosesser kan bli "heldige" og bli ferdige fort

Hvordan bestemme lengden på tidsbiter for RR?

- Lange eller korte?
 - Dette avhenger av hvor mye tid prosesser trenger på CPU sammenlignet med disk.
 - F.eks ved tidsbit på 100ms med prosesser A,B og C der A og B kjører for evig og C løkker for evig med en alternering mellom 1ms på CPU og 10ms på disk, vil C sultes for tid på CPU (CPU bound). Dette gir oss dårlig disk utilization (5%)
 - Ved tidsbit på 1ms med de samme prosessene får vi mye høyere disk utilization (91%).
 - CPU bundede prosesser liker lange tidsbiter, IO bundede liker korte tidsbiter (siden de også avhenger av IO enheter og ikke bare CPU).

Egenskaper en scheduler må ta hensyn til:

- Behandle like oppgaver på like måter
- Ingen prosesser burde vente for evig
- Kort responstid
- Maksimere throughput - få gjort ferdig så mange oppgaver så fort som mulig
- Maksimere bruk av PCens ressurser (mellan 40-90% er vanlig) - balanse, alle komponenter skal benyttes
- Minimere overhead - tid brukt på andre ting enn direkte prosessering
- Forutsigbar tilgang til CPU
- Kernel - ressurshåndtering, prosessor utilization, throughput, rettferdighet
- Bruker - ønsker blant annet kort responstid og konsekventhet (PCen skal ikke kræsje, programmer skal ikke f.eks ta mye lengre tid å åpne en gang)

- Brukeren av systemet: Servere vil som regel ha lengre tidsbiter, brukersystemer vil ha kortere.

Scheduleringsklassifiseringer:

Dynamisk schedulering:

- Tilpasser seg ved kjøretid
- Fleksibel
- Flere beregninger - mer overhead

Statisk schedulering:

- Lager scheduleringsregler før kjøretid
- Genererer dispatching tabell for kjøretids-dispatcheren ved kompileringstid
- Trenger kunnskap om oppgaven før kompilering

Kan man avbryte en oppgave?

Preemptive schedulering:

- Kjøring av oppgaver kan avbrytes ("preempts") av prosesser med høyere prioritet
- Avbrutte prosesser fortsetter senere med samme tilstand
- Mer overhead
- Real-time prioriteres fremfor best effort.
 - Real-time prosesser vil alltid kjøres hvis de finnes
- Når skal vi tillate "preemption"?
 - Preemption points - scheduler vil sjekke om det finnes noen høyt prioriterte oppgaver den må slippe til - forutsigbart overhead - liten forsinkelse for håndtering av real-time prosesser
 - Immediate preemption - nødvendig for harde real-time systemer (brukersystemer?) - mer overhead siden man må ta vare på tilstanden til den avbrutte prosessen - ingen forsinkelse for håndtering av real-time prosesser

Non-preemptive:

- Oppgaver med høyere prioritet må vente på at oppgaven som prosesseres nå fullfører - kan også brukes uten prioritet
- Mindre overhead i forhold til context switcher
- Med prioritet har håndtering av real-time prosesser en liten forsinkelse.

I dag er dynamisk og preemptive scheduleringsalgoritmer vanligst, men alle brukes. Det fleste systemer har et slags form for prioritering.

Prioritetsscheduling:

- Scheduler lager flere køer for hver prioritet. Alle prosesser fra kø med høyest prioritet vil utføres før scheduler går over til neste kø
 - Fordel - rettferdighet (så lenge tildeling av prioriteter gjøres riktig)

- Ulempe - utsulting, noen prosesser må kanskje vente veldig lange - løsning: dynamiske prioriteter som endrer prioriteten på prosesser når de har ventet for lenge, men dette gir mer overhead.

Microsoft windows 2000: Forgrunnsaktiviteter gis lengre tidsbiter siden de trenger rask responsid

- 32 prioritetsnivåer med RR i hvert nivå
- Input og throughput orientert
- De 16 øverste nivåene: real-time prosesser med statiske prioriteter som kan kjøre for evig
- De 15 neste nivåene: prioritet kan økes eller synkes med 2 nivåer. CPU bundede prosesser får redusert prioritet. IO bundede prosesser får økt prioritet. Dette er for å øke interaktivitet med systemet.
- Det siste nivået er for veldig lavt prioriterte OS prosesser ### Windows 8/10: For det meste det samme, med 2 nye grupperinger som skiller mellom prosessprioritet og trådprioritet.
- Hver prosessklasse har 7 ekstra trådprioriteter.
- For å finne den endelige trådprioriteten for en prosess tar man hensyn til begge disse
- Dynamiske prioriteter for nivå 0-15
- Bruker kan også lage egen scheduling for hver applikasjon (user mode scheduling, UMS)
- Kan også gi garanti for multimedia prosesser (MMCSS)

Linux:

- 3 prioritetsklasser:
 - SCED_FIFO - prosesser kan kjøre for evig uten tidsbiter
 - SCED_RR - mykere versjon av SCED_FIFO, tidsbiter på 10ms(quantums)
 - SCED_OTHER - for vanlige brukerprosesser - 40 (nice-verdier) prioriteter med tidsbiter på 10ms(quantums)
- Tråder med høyest "goodness" kjører først.
 - For real-time prosesser under SCED_FIFO og SCED_RR: goodness = 1000 + prioritetsnivå
 - For timesharing (bruker prosesser) under SCED_OTHER: goodness = tid igjen for prosessen + prioritet
- Quantums resettes når ingen klare prosesser har quantums igjen (end of epoch)
- Denne løsningen var ikke helt bra - gikk over til CFS ### Nyere linux - Completely Fair Scheduler (CFS):
 - Bare en kø: Alle får sin rettferdige del av en tidsbit - F.eks. hvis du har en tidsbit på 10ms og to prosesser har samme prioritet deler de den tidsbiten opp i 5ms * 2
 - Hver prosess sorteres etter virtuel kjøretid, de med lavest får høyest prioritet
 - Bruker kun et rødsvart tre for køen av prosesser
 - Unngår å bruke mange separate køer schedulerer må søke gjennom.
 - For å finne neste prosess å kjøre går man bare lengst ned til venstre i treeet
 - Hvis 2 brukere køer opp prosesser vil hver bruker få 50% hver og så dele opp hver brukers tildelt prioritet på deres prosesser.

Når skal scheduler brukes?:

- Når en ny prosess lages
- Når en prosess er ferdig
- Når en prosess blokker
- Når en prosess avbrytes

- I preemptive systemer: ved preemption points.

Minne:

Utfordring:

- Hvordan skal vi alllokere minnet vi har?
- Hvordan skal vi finne minnet vi har tildelt?
- Hvordan bestemme hva som må ut når noe nytt skal inn (og vi ikke har plass)?
- Hvordan beskytte tildelt minne sånn at det ikke blir overskrevet?
- Hvordan forsikre at prosesser leser fra riktig minneområde?

Hierarkier:

- Forskjellige minne har forskjellige størrelser og hastigheter. Jo raskere jo mindre kapasitet. Det er store hopp i hastighet fra et nivå til et annet.
 - Registre
 - Cache (inneholder også flere nivåer)
 - Hovedminne (RAM)
 - Sekundærminne (disk)
 - Tertiærminne (USB)

Addressering:

- Hardware bruker *absolutt* addressering, dvs. spesielt reserverte minneområder. Dette er raskest, men veldig lite dynamisk.
 - For avlesing les absolutt byte f.eks 0x000000ff
- **Absolutt addressering virker ikke for software.** Man kan ikke garantere at en prosess alltid vil ha det samme tildelte minneområdet.
 - MEN, man kan bruke relativ addressering. Dvs. at hvis man vet startadressen til prosessen og en relativ adresse man kan legge til startadressen for å finne den absolutte fysiske addresen man trenger.
 - Større kostnad, mer overhead, men gir oss støtte for **dynamisk addressering**.
- Man partisjonerer minneområdet til en prosess opp i tre deler:
 - Kodesegment: Dette inneholder instruksjonene for å eksekvere programmet.
 - Datasegment: Dette inneholder globale og statiske variabler. Både uinitialiserte og initialiserte.
 - Heap: Minneområde satt av for dynamisk voksende minne - *malloc()*. Heapen vokser opp mot stacken.
 - Stack: Minneområde for parametre og variabler i en funksjon. Stacken vokser ned mot heapen. **- Her kan det oppstå problemer om de overflorer inn i hverandre.**
 - Prosesskontrollblokk (PCB): Systemdata, peker til starten av hvert segment (kodesegment, datasegment, stack).
 - På de aller høyeste minneaddressene: kommandolinjeparametre, omgivelsesvariabler. Struktur for å håndtere stacks med flere tråder.

Globalt minneoppsett for et system:

- Laveste minneaddresser for systeminformasjon.
- Neste er OS kjernen
- Resten settes av til vanlige prosesser (+ transiente prosesser fra OSet som kan byttes ut og avbrytes)

Minnehåndtering for multiprogrammering:

- Å ha alt i primærminne er umulig - må finne en balanse med sekundærminne
- Primitiv metode - *swapping*: Når en prosess startes lastes den inn fra sekundærminne inn i primærminne. Dette innebærer veldig mange kostbare operasjoner med å lese fra disk.
- Neste metode - *overlays*: Delegerer minnehåndtering til programmereren. Laster inn segmenter fra sekundær- inn i primær-minne når det trengs. Fortsatt ikke optimalt.
- Til slutt - *segmentering/paging*: Litt likt overlays, men lar kjernen håndtere det i stedet for programmereren.

Fikset partisjonering:

- Deler minne inn i statiske partisjoner ved oppstart av system.
- Enkelt å implementere og støtter swapping av prosesser.

Equal-size partisjoner (f.eks. alle partisjoner er på 8MB)

- Hvis et program krever mer minne enn størrelsen på en partisjon funker det ikke.
- Hvis en prosess krever veldig lite minne vil dette kaste bort mye minne (en type fragmentering)

Unequal-size partisjoner (f.eks. noen partisjoner på 2MB, noen på 4MB noen på 8MB, 16, osv):

- Støtter bedre en blanding av store og små prosesser
- Kan skape problemer hvis man har bare store eller bare små prosesser

Dynamisk partisjonering:

- Deler inn partisjoner under kjøretid
 - Når en prosess opprettes blir den tildelt det minnet den trenger.
- Skaper external fragmentering når prosesser fullføres og det er fritt minne løst i minne
 - Mulig løsning - *Compaction*: flytt alle kjørende prosesser til en del av minne og det ledige til en annen del av minne - gir en stor partisjon med fritt minne.
 - Tar tid og bruker ressurser - mer overhead.
 - Annen løsning - finne en bedre måte å allokere plass:
 - **First fit**: Starter fra starten og allokerer det første minneområdet som passer (oftest brukt av disse 3)
 - Rask og lite overhead
 - Gir ikke nødvendigvis best resultat

- **Next fit:** Litt samme som first fit, bare at den starter å lete fra der den allokerede minne sist i stedet for fra starten som i first fit.
 - Gir bedre spredning av minne.
- **Best fit:** Ser gjennom hele minneområdet og allokerer den minste partisjonen den finner som passer for prosessen.
 - Høres fristende ut
 - Mer overhead
 - Etter denne algoritmen har kjørt en stund sitter man ofte igjen med så mange små "ubrukelige" partisjoner at det ikke egentlig lønner seg.
- **Buddy system:** Blanding av fikset og dynamisk partisjonering. Når en ny prosess skal få tildelt minne finner man den minste partisjonen den har plass i. Så prøver man å partisjonere den partisjonen i 2 og sjekker om den prosessen fortsatt får plass. Dette gjør man helt til prosessen ikke får plass i partisjonen hvis man deler den i 2. Når en prosess er ferdig prøver man å merge prosessens minneområde med nabopartisjonene.
 - Gode resultat
 - Enda mer overhead
 - Hva hvis du f.eks. har en prosess som krever 513kB minne? Må man da allokerere en hel megabyte for denne? Løsning: ikke lagre alt kontinuerlig i minne.

Segmentering:

- Å kreve at minnet til en prosess må lagres kontinuerlig skaper mye fragmentering
 - Løsning: segmentering.
- Dette skaper mer overhead fordi man må kunne vite hvor en partisjon slutter og den neste begynner. Uten segmentering trenger du ikke vite hvor den slutter.
 - Ekstra oppslag for addressering for å ta vare på hvilken partisjon under den prosessen når man må søke opp.

Adresseoppslug med segmentering:

- Man må ha et **segment table** for startaddressene til hvert segment.
- Addressen er todelt
 - Først hvilket segment - øverste bitsene. Dette er en indeks i segment tabellen for å finne startaddressen til det segmentet man trenger.
 - Neste er offset fra starten av det segmentet.
- Når man har disse to kan man først lete opp startaddressen til segmentet man vil ha i segment tabellen. Så kan man legge til offsettet for å finne den absolute adressen man ønsker.

Paging:

- Med segmentering har vi overhead siden man tillater å ha segmenter med forskjellige størrelser så man må også ta vare på størrelsen på alle segmenter.
 - Løsning: **Paging**.
- Samme oppsett som segmentering, men med segmenter med faste størrelser.
 - Ingen ekstern fragmentering
 - Lite intern fragmentering, men dette kommer an på størrelsen på en page. (Hvis man må runde opp veldig mye kan dette skape intern fragmentering)
- Når man skal starte en prosess runder man opp til det laveste antall *pages* den prosessen trenger for å kjøre.
 - Kaster bort den lille delen minne man runder opp

- Hvis man skal starte en prosess med et litt fragmentert minne og den ikke har kontinuerlig plass fra det første ledige stedet i minne kan man legge inn starten av prosessen der og resten ved neste ledige plass (segmentert/paget).
- Man kan også kombinere segmentering og paging

Virtuelt minne:

- Hva hvis man skal starte en prosess når primærminne er fullt?
 - Løsning: Virtuelt minne - lagre prosessens data på disk.
- Da trenger vi en **page table**
- Man lager et virtuelt minneområde. Det virtuelle minneområdet består av en rekke pekere som kan peke både på minneområder i primærminne og på disk dersom nødvendig.
 - Hvis man har en peker til noe på disk må man finne plass til å flytte dette fra disk inn i minnet. Det innebærer å ta noe fra primærminnet og flytte det ut til disken for å så sette det fra disken inn i den nye åpne plassen.

Minneoppslag med virtuelt minne:

- Hvordan håndterer jeg at noe ligger på disk og noe i primærminnet?
 - Man skiller mellom virtuelle addresser og fysiske addresser.
 - Virtuell adresse: første bits for å indeksere page table, neste bits er offset fra denpagen.
 - Fysisk adresse: når du har indeksert page tabellen med de første bitsene i den virtuelle addressen sjekker du de tilsvarende bitsene på den indeksen. Når du har hentet startadressen til pagen sjekker du om **present biten** er 1. Hvis den er 1 kan du gjøre oppslaget som vanlig i primærminnet. Hvis den er 0 får du en **page fault**. Da må du hente dataen fra disk inn i minne først.

Page fault håndtering:

- Vi må gjøre en context switch.
 - Vi må lagre data i registre og annen volatil informasjon
- OS finner hvilken virtuell page som er forespurt
- OS sjekker om pagen er valid og om prosessen har tilgang.
- Så kan vi hente inn pagen fra disken
 - Vi må finne plass i minne: dvs. kaste ut noe fra minne for å lage plass til det som skal hentes fra disk. Hvis element valgt for å byttes ut er "dirty" (oppdatert fra det som er på disk), må man først skrive ut de forandringene for å ikke tape disse.
 - OS finner elementet fra disk og prøver å putte det inn i minnet.
 - Når dette er ferdig vil en "*disk interrupt*" eksekveres. Nå kan vi oppdatere page tabellen med elementet hentet fra disk sin nye posisjon i minnet, det innebærer å sette present bitet i page tabellen for elementet som genererte page faulten til 1.

Optimalisering av paging:

- Page tabeller kan bli ganske store og de kan være ganske kostbare å søke gjennom og man jobber ofte kun med et lite område i et addresseområde så du trenger ikke hele disse tabellene.
 - Løsning: **Multilevel paging**
- Multilevel paging innebærer nøsting av paging tabeller.

- For å kontrollere hvordan denne multilevel pagingen skal håndteres brukes det gjerne et kontroll register. Disse har forskjellig arkitektur avhengig av systemet.
- Hvis man vil søke opp et minneområde i en multilevel paging tabell må du først finne hvilken tabell du skal lete i. I den virtuelle minneadressen brukes de første n bitsene for å søke opp i hovedtabellen etter den nærmestepagen. I denne hovedtabellen har også hver page en tilknyttet present-bit så det kan også oppstå en page fault ved oppsök på denne page tabellen dersom denne er 0. I den nærmeste tabellen bruker du de neste bitsene fra den virtuelle minneadressen for å indeksere denne tabellen. Her kan det også hende at present-biten er 0 som igjen kan skape en page fault. Etter håndtering av denne eventuelle page faulten kan du bruke de resterende bitsene fra den virtuelle minneadressen som offsett inn i det endelige minneområdet til denne pagen.
 - Merk at multilevel paging kan gå dypere enn de 2 nivåene beskrevet her. Det innebærer en lengre virtuell minneadresse og flere page tabell oppsök.

Page replacement algoritmer:

- Hvordan skal vi velge hva vi skal bytte ut ved en page fault?
- Random, FIFO, not recently used, least recently used, osv
- Second chance: Versjon av fifo. Hvis et element aksesseres mer enn en gang settes et R bit. Ved en page fault vil det sjekkes om R bitet for det første elementet i pagen(?) er satt. Hvis det er satt vil elementet flyttes til starten av pagen med R bit satt til 0.
 - Ok algoritme, men trege ettersom listeoperasjoner er trege.
 - Løsning: **Clock** implementasjonen av second chance. Dette bruker en peker til det neste elementet. Pekeren oppdateres hver gang et element hentes fra pagen. Ved page fault vil vi sjekke om det pekeren (som nå peker på det første elementet i listen) peker på sin R bit er 0. Hvis den er 0 bytter vi ut det elementet der pekeren er. Hvis den er 1 setter vi R biten til 0 og går til neste element der vi sjekker R biten igjen.
 - Enda bedre: **Least Recently Used (LRU)**. Hvis vi ser på pagen som en lenket liste oppdaterer vi listen hver gang man henter et element fra en page ved å sette det elementet fremst i listen. Ved page fault bytter ut elementet det bakerste elementet ettersom det er det som var lengst siden var brukt. Denne algoritmen er mye mer kostbar, men er foretrukket siden den gjør at vi kan unngå dyre page faults.

Zero copy data paths:

- Kommunikasjon mellom flere komponenter, spesielt fra user-space ned til kernel-space er veldig dyrt. Da må man kopiere over masse data og undergå mange context switches.
 - Løsning: Zero copy data paths - lar flere komponenter og brukere dele samme minneområde ved bruk av pekere.

Applikasjonsspesifikke algoritmer:

- Disse algoritmene kan være lite effektive ved bruk i applikasjoner. F.eks hvis 2 personer skal spille av samme video to ganger. Løsning: applikasjonsspesifikke algoritmer

Least/most relevant for presentation:

- Brukt for multimedia applikasjoner. Laster inn elementer mest relevant for presentasjon fra disk, bytter ut minst relevante for presentasjon ved page faults.
- Denne algoritmen går ut fra at tilstanden til minne representerer en snapshot av en multimedia strøm der hver COPU (continues object presentation unit) har en

tilsvarende relevansverdi, f.eks. fra 0 til 1. Algoritmen tar også vare på to set med funksjoner, et for historien av multimedia allerede presentert og et for refererte elementer som er de som går inn i fremtiden.

- Relevansverdier kalkuleres ut ifra hvor nærmeste innholdet i et element er i forhold til det gjeldende presentasjonspunktet. Des desto lengre unna du er desto lavere relevansverdi har du. Relevansverdiene endrer seg også når presentasjonspunktet progresserer.
- Hver COPU kan også ha flere relevansverdier (f.eks. hvis det er flere klienter tilkoblet en server). Da må serveren beregne globale relevansverdier for alle COPUer som tar hensyn til alle tilkoblede klienter.
- Få page faults (færre disk aksesser)
- Interaktivt
- Kostbart å beregne relevansverdier for hver COPU, men fortsatt kanskje foretrukket framfor dyre disk aksesseringer.

Interval Caching:

- I motsetning til alle de andre algoritmene som har vært blokk-level caching, bruker denne strøm-avhengig caching.
- Hvis man for eksempel har 2 klienter tilkoblet en multimedia server kan man se på distansen mellom de 2 klientene for å beregne ut hvor relevant det er å beholde elementer i minne. På denne måten kan man gjenbruke elementer den ene klienten har brukt ved å cache det dersom den andre klienten er nærmere nok til at det virker gunstig.
- Hvordan kan vi velge intervallene vi ønsker å cache for?
 - Sorter etter lengden på intervallene og cache et og et intervall så lenge vi har mer minne.

Hvilken algoritme er best for video streaming?

- LRU: Koster mye og ikke veldig bra resultater
- LMRP: Koster mest og ikke veldig bra resultater
- IC: Koster lite og gir best resultater

Disk:

Disker:

- Disker er trege hvorfor bruke dem?
 - Har ikke nok primærminne.
 - Primærminne er ikke persistent (slettes når pc skrus av)
- Hva med SSDer?
 - Mer liknende minne - ingen mekaniske deler.
 - Mye raskere (mikrosek vs nanosek)
 - ~10x dyrere - sjeldent i servere
- Hva består en harddisk av?
 - **Platters:** Magnetisk materiale for å gi persistent lagring av bits
 - **Spindle:** Platene roterer rundt denne
 - **Tracks:** Alle plater deles inn i disse koncentriske sirklene
 - **Sectors:** Hver track deles inn i flere sektorer på en fast vinkel. Sektorer separeres av ikke magnetiske gaps som kan brukes for å identifisere starten på en ny sektor.

- **Cylinder:** En samling av plater (muligens brukt på begge sidene av platen) sine tracks som ligger samme avstand fra spindlen kalles et cylinder.
- **Disk head:** Flyttes fram og tilbake fra nærmest spindel til ytterste sektorer på kanten. Leser og endrer bits under den når platene roterer.
- Lagringskapasitet påvirkes av disse delene:
 - Hvor mange plater har vi? Brukes disse på begge sider?
 - Hvor mange tracks har vi på hver plate?
 - Hvor mange sektorer deler vi hver track inn i (gjennomsnittlig)? Hvor stor er en sektor?

Disk aksessering:

- Må posisjonere disk hode over tracks i riktig rekkefølge
 - **Seek time:** Først flytte diskhodet over riktig track. Hastigheten på dette avhenger av hvor mange spor man skal flytte seg over. For å beregne dette må man finne tiden for å starte og slutte søkingen + aksellerasjonen for å komme opp og ned i fart. Det er billigere per spor å flytte seg over mange spor, enn det er å flytte seg over få spor på grunn av aksellerasjonstiden. Dette tar vanligvis et par millisekunder.
 - **Rotational delay:** Vente til platen roterer nok til at det første bitet man skal lese ligger under diskhodet. Hastigheten på dette avhenger av hvor langt unna hodet, den blokken man skal lese fra er, i tillegg til hvor raskt (RPM) platene roterer. Vanligvis også noen millisekunder.
 - **Transfer time:** Vente til platen roterer nok til at alle de etterspurte bitsene har passert diskhodet. Hastigheten på dette avhenger av hvor mye man skal lese fra sporet(?) i tillegg til hvor raskt platene roterer og hvor tett dataene ligger. For å regne ut dette tar man mengden data man skal lese delt på *transfer rate*. Transfer rate finner man ved å ta mengden data per spor delt på tiden det tar for en rotasjon. Man kan også måtte endre spor å lese fra og da må man også ta hensyn til flytting av hodet.
 - **Other delays:** Venting i kø, venting på ressurser (ikke relevant, ofte lik 0)
- Tradeoffs: **Kapasitet vs hastighet**
- Forskjell mellom lese- og skriveoperasjoner:
 - Må alllokere plass
 - Må ofte verifiseres at det har blitt skrevet riktig informasjon - tar lengre tid
 - For modifikasjon kombinerer man lesing og skriving.

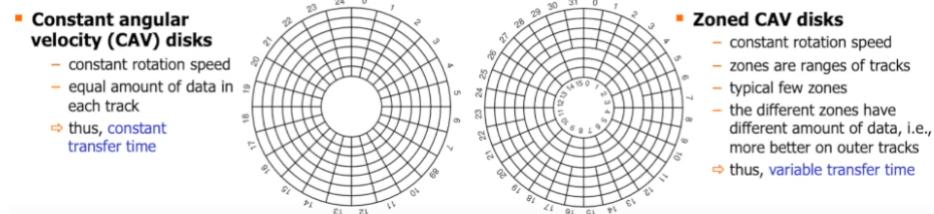
Optimalisering av disk aksessering:

- En treg algoritme med få disk aksesseringer vil antageligvis være bedre enn en rask algoritme med mange disk aksesseringer (se mer under Minne)
- Endre blokkstørrelse
- Endre diskscheduling
- Plassere data i fornuftige posisjoner sånn at data som sannsynligvis kommer til å hentes sammen ikke ligger langt fra hverandre.
- Cache ting i minnet
- Prefetch (f.eks. hvis vi ser på video og har sett frame 1 & 2 kommer vi antageligvis til å trenge frame 3 & 4)
- Kombinere diskar vha. RAID

Disk scheduling:

- Mye dyrere å avbryte en diskoperasjon enn prosess scheduling - Da dobler man søketiden, som er veldig dyrt.

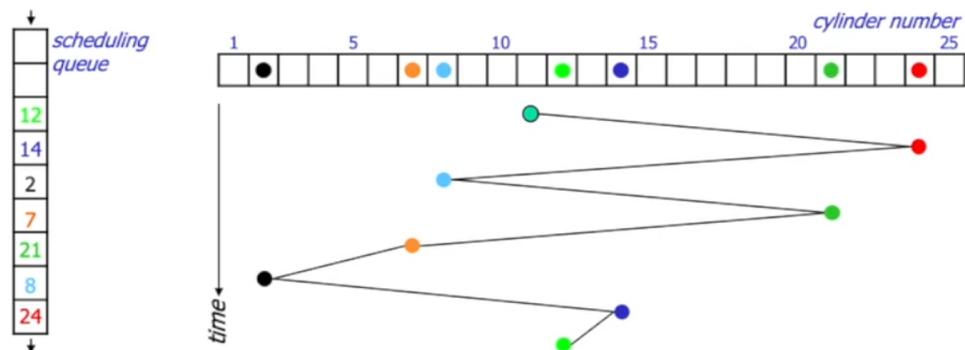
- Bedre å optimalisere rekkefølgen data hentes på for å minimere bevegelse på disken. OSet tar seg av dette.
- Generelle mål:
 - Kort responstid
 - Høy throughput
 - Rettferdighet
- Tradeoff: seek efficiency vs max response time
- Kompliserende faktorer:
 - OSet får ikke alltid riktig informasjon om disken. Her må diskkontrolleren må oversette.
 - Hvis noe blir fysisk ødelagt vil disken prøve å gjennoprette den ødelagte dataen. Dette er derfor noe plass settes av i tilfelle noe feil skjer. Når OSet sorterer forespørslene (se disk scheduling algoritmer) kan det hende denne sorteringen blir feil siden dataen kanskje ikke ligger der OSet tror.
 - De indre og ytre sylinderne har ikke faktisk like mye plass (omkretsen på de indre er mindre enn de ytre).
 - Her er det to muligheter: vanlig CAV disker (Constant Angular Velocity) og Zoned CAV disker. Med vanlig CAV flytter disken seg like raskt uansett hvor du skal lese fra og mengden data ytterst og innerst er den samme. Zoned CAV disker (det man bruker i dag) deler disken inn i soner basert på hvor langt unna spindlen du er. De ytre sporene har da mer kapasitet enn de indre.



Disk scheduling algoritmer:

- First come first serve: FIFO - lang søketid, kort responstid
incoming requests (in order of arrival, denoted by cylinder number):

12 14 2 7 21 8 24

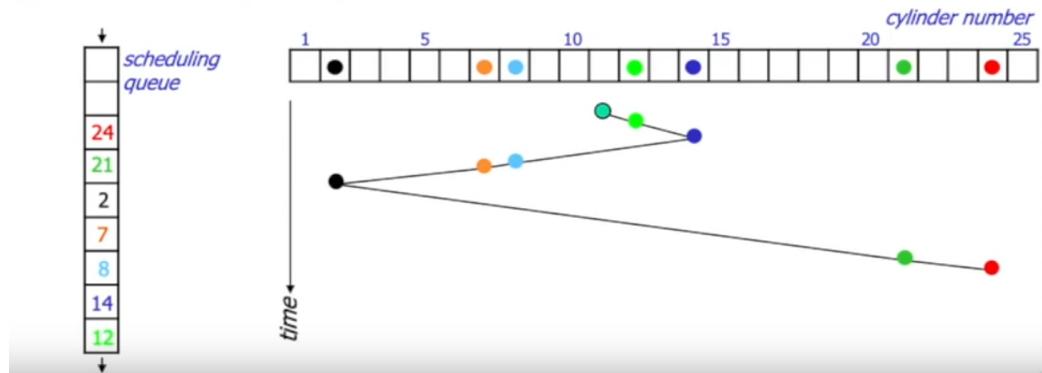


- Shortest seek time first: Sorter forespørsler etter distanse fra hodet - kort søketid, lengre responstid. Hvis det er mange forespørsler rundt hodet kan det føre til

utsultning for forespørsler lengre vekk fra hodet.

incoming requests (in order of arrival):

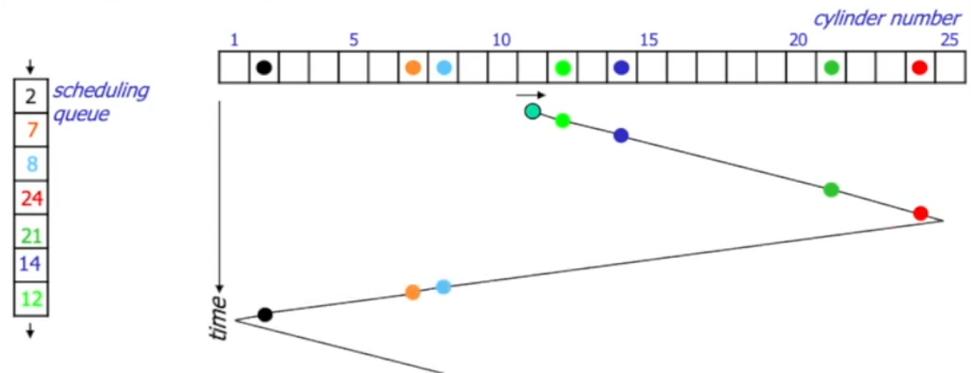
12 14 2 7 21 8 24



- SCAN (elevator): Kompromiss mellom FCFS og SSTF. Bidireksjonell. Først start med å gå i en retning. Sorter forespørsler etter det som er nærmest hodet, men alle forespørsler som ligger bak hodet legges bak de som ligger foran i køen av operasjoner.

incoming requests (in order of arrival):

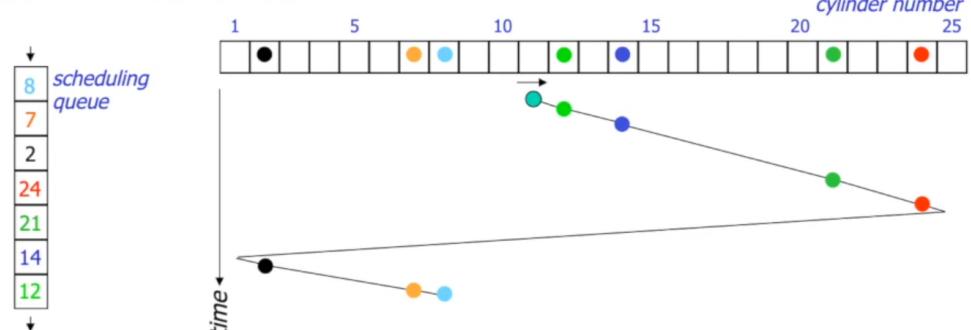
12 14 2 7 21 8 24



- C-SCAN (Circular SCAN): Optimalisering av SCAN. uni-directional - dette gjør at vi slipper å aksellerere når vi skal endre retning. Når vi kommer til slutten i den valgte retningen går vi heller videre tilbake til starten før vi fortsetter videre. Bedre i fleste tilfeller, men vanlig SCAN kan være raskere hvis det er få forespørsler.

incoming requests (in order of arrival):

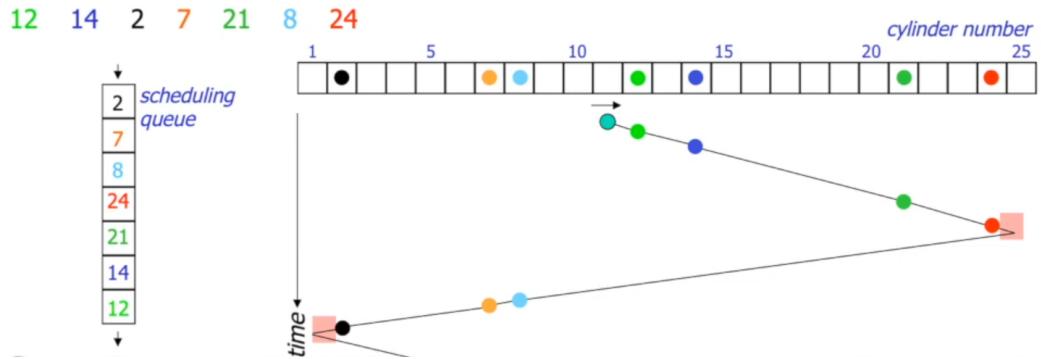
12 14 2 7 21 8 24



- Look og C-look: Optimalisering av SCAN og C-SCAN. I stedet for å gå mellom de helst ytterste og innerste sylinderne, går man mellom de ytterste- og innerste

forespurte sylinderne.

incoming requests (in order of arrival):

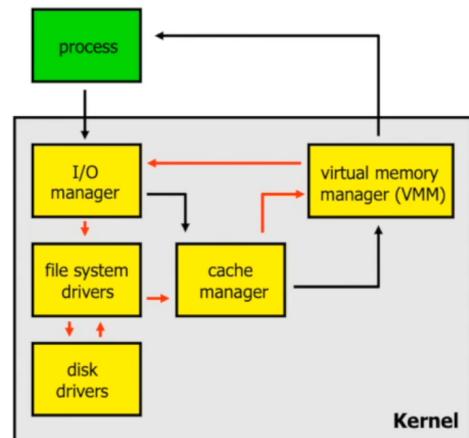


Moderne diskscheduling:

- Caching: Når OSet spør om å hente noe fra disk er det ikke sikkert man faktisk går helt til platene, siden diskene gjerne har et cache.
 - Måten dette gjøres på er at når man skal lese noe fra disken leser man hele sporet, og kanskje noen nabospor også, og kopierer dataen herfra inn i et minnebuffer der vi har satt av litt plass i minnet for caching. Hvis noen av denne cachede dataen blir forespurt senere slipper man da å gå helt ned på diskene igjen.
 - Hvor stort skal dette bufferet i minnet være?
 - Hvor mye data skal vi prefetche?
 - På Windows:

Buffer Caching: Windows (XP ++)

- An I/O manager performs caching
 - centralized facility to all components (not only file data)
- I/O request processing:
 1. I/O request from process
 2. I/O manager forwards to cache manager
 - in cache:
 3. cache manager locates and copies data to process buffer via VMM
 4. VMM notifies process
 - on disk:
 3. cache manager generates a page fault
 4. VMM makes a non-cached service request
 5. I/O manager makes request to file system
 6. file system forwards to disk
 7. disk finds data
 8. reads into cache
 9. cache manager copies data to process buffer via VMM
 10. VMM notifies process



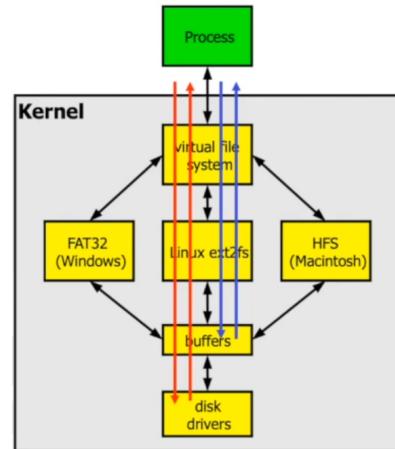
- På Linux:

Buffer Caching: Linux / Unix

- A **file system** performs caching
 - caches disk data (blocks) only
 - may hint on caching decisions
 - prefetching

- I/O requests processing:

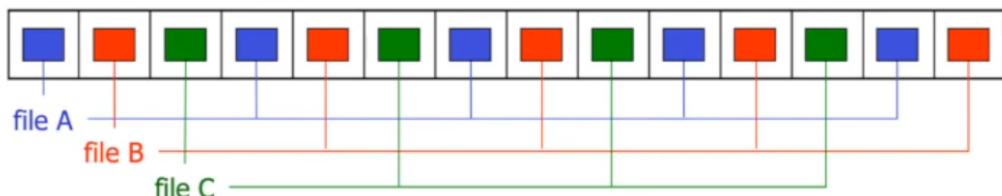
1. I/O request from process
2. virtual file system forwards to local file system
3. local file system finds requested block number
4. requests block from buffer cache
5. data located...
 - ... **in cache**:
 - return buffer memory address
 - ... **on disk**:
 - make request to disk driver
 - data is found on disk and transferred to buffer
 - return buffer memory address
6. file system copies data to process buffer
7. process is notified



- For caching strukturen brukes gjerne en LRU algoritme. I tillegg brukes en hash tabell der man kan hashe filnavn og blokknummer for raskere oppslag av den blokken. På den måten slipper du å søke gjennom hele listen.
- NOOP (FCFS med request merging, brukt i linux) merger sammen flere forespørsler for å spare tid. Denne algoritmen funker på ikke mekaniske disker siden de ikke har noe søketid.
- Deadline I/O
 - C-SCAN basert
 - 3 Køer: 1 sortert (elevator) kø, og 2 deadline køer (read/write)
 - Anticipatory er en versjon av denne algoritmen som venter litt for å prøve å slå sammen flere forespørsler.
- Completely Fair Queuing (CFQ): en kø for hver prosess der hver kø får en periodisk aksess. Hvor lenge denne perioden er avhenger av hvor mange brukere som er venter på køen. work-conserving: hvis en prosess ikke bruker tiden sin går man over til neste mens man er på disken.
 - Kan man optimalisere diskscheduling fra applikasjonen? - ja, OSet vet ikke alltid best.

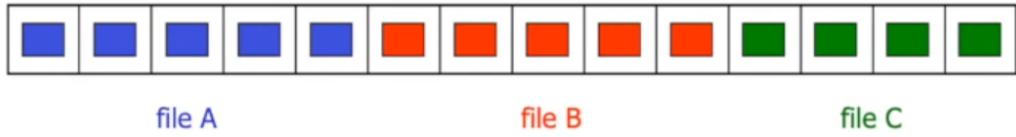
Plassering av data på disk:

- **Interleaved:** Minimal disk arm bevegelse hvis man har et forutsigbart mønster for forespørsler. Ingen gevinst hvis vi har et uforutsigbart mønster.

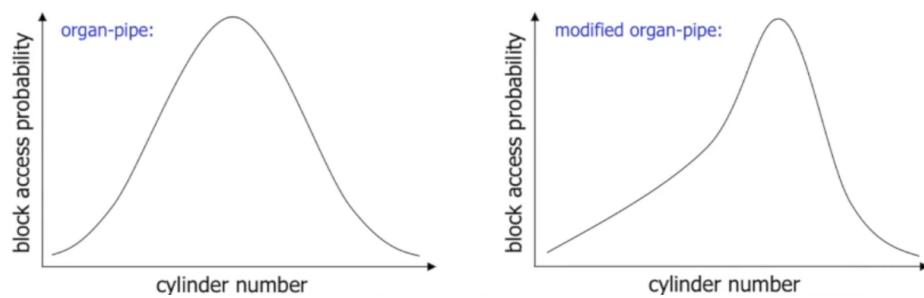


- **Contiguous:** Minimal disk arm bevegelse hvis man leser en hel fil. Ingen bevegelse av hodet når man leser fra én fil (ingen søketid, rotasjons forsinkelse). Vanligvis vil man lese fra andre filer i tillegg som gjør at vi kan få store søker fra

en fil til en annen.

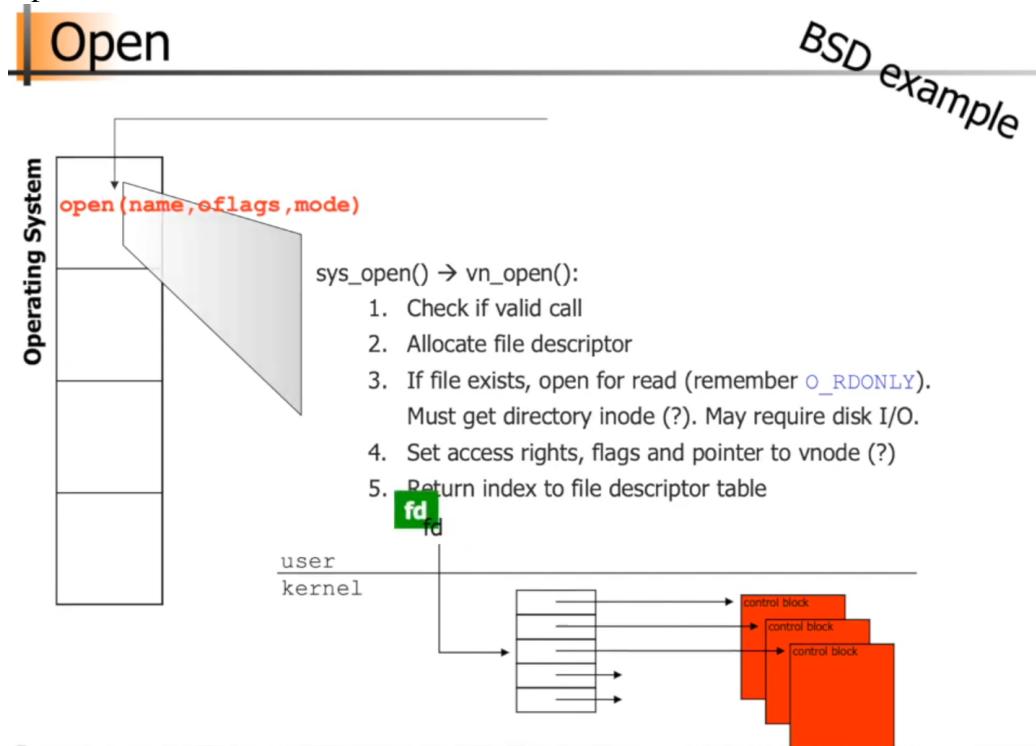


- **Organ-pipe:** Plasserer oftest forespurte blokker i midten av diskens sider det er der hodet i snitt trenger å gå kortest for å aksessere (bell-curve for hvor ofte blokker plasseres).
 - **Modified Organ-pipe:** Forskyver 'midten' litt lenger utover der det er plass til mer data og data kan leses raskere.

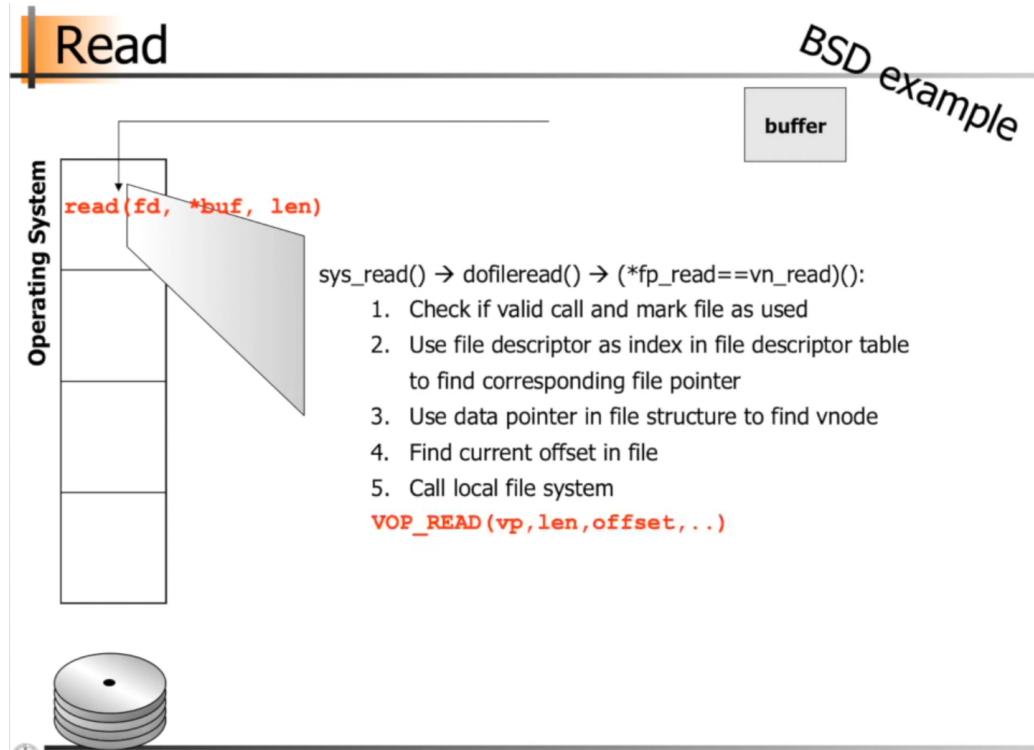


Filer:

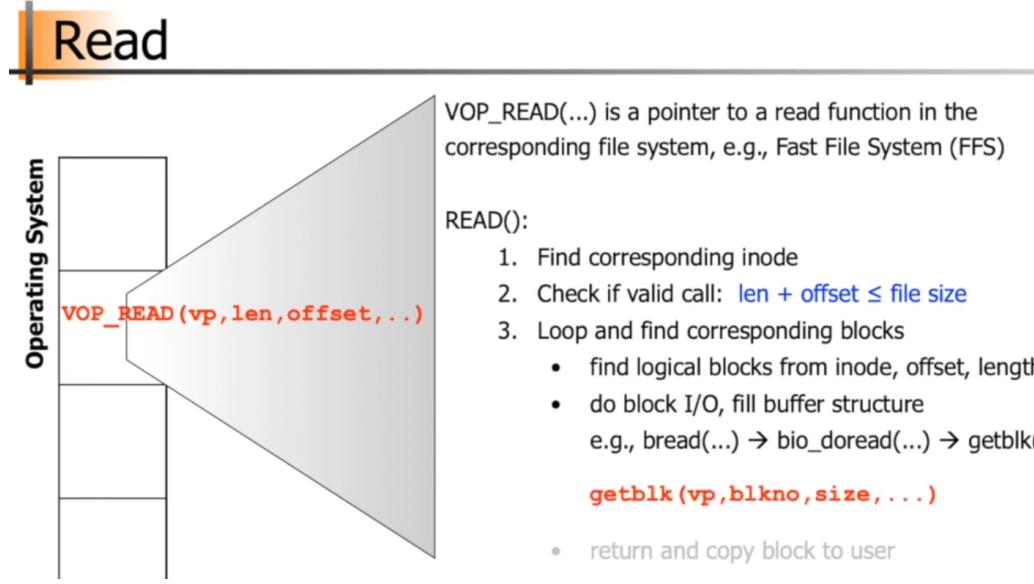
- En sekvens med bytes. I noen systemer tas det også hensyn til hva disse bytesene representerer når de leses, men vi ser på dem som **ustrukturerte filer**
- Filsystemet:
 - **Storage management:** allokerer plass, finne blokker, sortere sånn at de hører sammen.
 - **File management:** Hvem eier filen, hvilke tilganger har hvilke brukere?
- Open:



- Read:



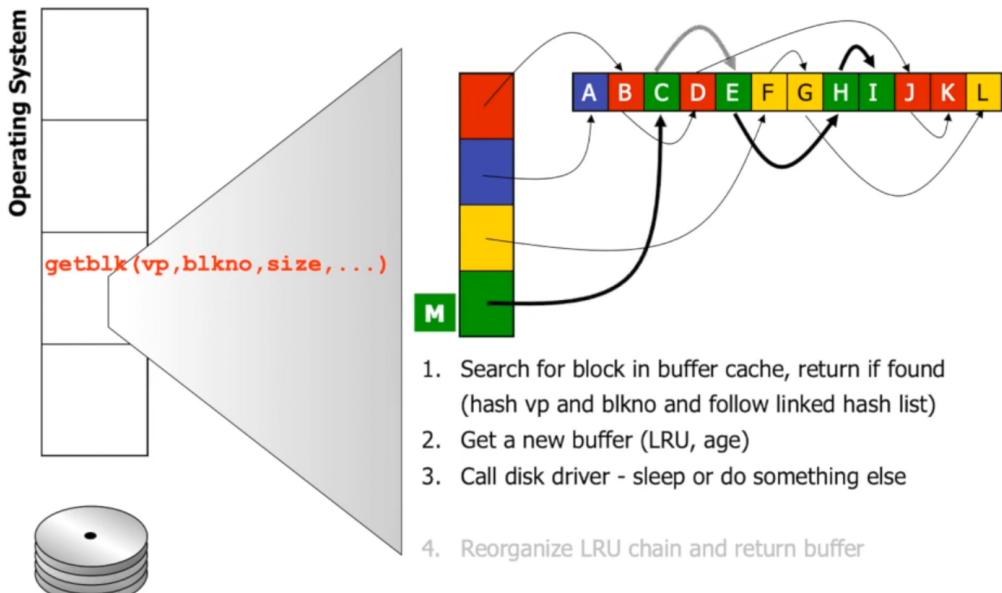
- VOP_READ(): Dette er en virtuell funksjonspeker. Dvs at det er en funksjon som peker på en annen funksjon avhengig av miljøet den kalles fra.



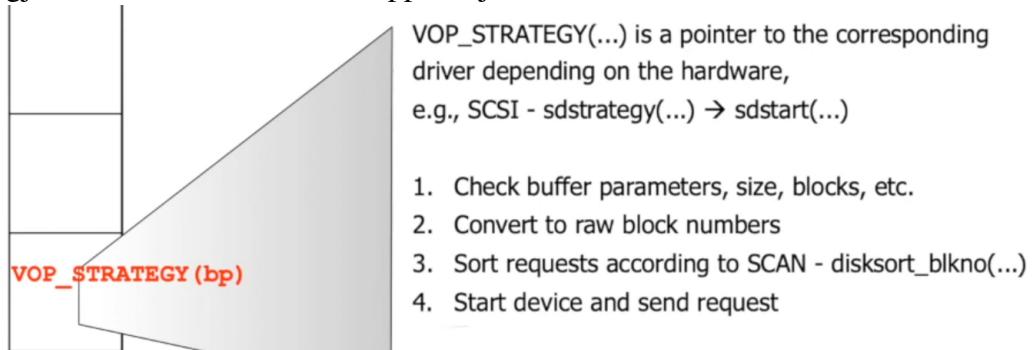
- getblk(): følg den lenkede hash listen. Hvis du ikke finner blokken i hash listen betyr det at vi ikke har blokken i cachet. Ifølge LRU bytter man da ut det least

recently used elementet (i bildet den gule L'en) og legger inn blokk i cache.

Read



- **VOP_STRATEGY:** Også en virtuell funksjonspeker. Avhenger av hva slags disk den skal lese fra. Den initierer endelig lesing fra selve disken. Etter dette er ferdig returnerer man tilbake til getblk() og oppdaterer hash listen, det innebærer å kaste ut det least recently used elementet og omorganisere lenkene i listen. Til slutt sender man en interrupt som sier at dataen ligger klar i buffercachen som gjør at man kan returnere til applikasjonen.



Filsystemet

- Hvordan finne riktige blokker på disken?
- **Chaining i media:**
 - Filsystemet har metadata om hver fil, metadataen er bare en peker til den første diskblokken. Alle andre diskblokker er lenket mellom hverandre som en lenket liste.
 - Fint for små filer eller sekvensiell lesing.
 - Dyrt å søke etter blokker midt i en fil (må gå gjennom alle foregående ledd i listen).
- **Chaining i et map:**
 - Filsystemet inneholder en tabell. Hvert element i tabellen inneholder en peker til en filblokk og en indeks til den neste pekeren til en annen filblokk. I

stedet for å søke gjennom alle filblokkene kan du søke gjennom mapet i stedet (billigere).

- FAT (File Allocation Table):
 - Boot sektor
 - 2x tabeller beskrevet ovenfor
 - Root dir
 - Andre dirs
- Tabellen kan fortsatt bli ganske stor å søke gjennom - løsning: **Table of Pointers**

- **Table of pointers:**

- Hver fil har en egen tabell med pekere på samme måte som i **Chaining i et map**.
- Inoder - hver fil/dir har en del metadata og pekere. ~12 pekere ~4KB for dataen i en fil - ikke nok for mange filtyper, derfor har inoder også *Single-, Double- og Triple- indirect* pekere som er pekere til egne diskblokker som holder på flere pekere til andre diskblokker.
 - Single indirect: inneholder en peker til en(eller flere?) diskblokk med pekere til andre diskblokker med data
 - Double indirect: inneholder en peker til en diskblokk med pekere til andre diskblokker som peker på diskblokker med data.
 - Triple indirect: inneholder en peker til en diskblokk med pekere til andre diskblokker som inneholder pekere til andre diskblokker med data.
 - Indireksjon fører til unødvendig mange diskoperasjoner - dyrt.

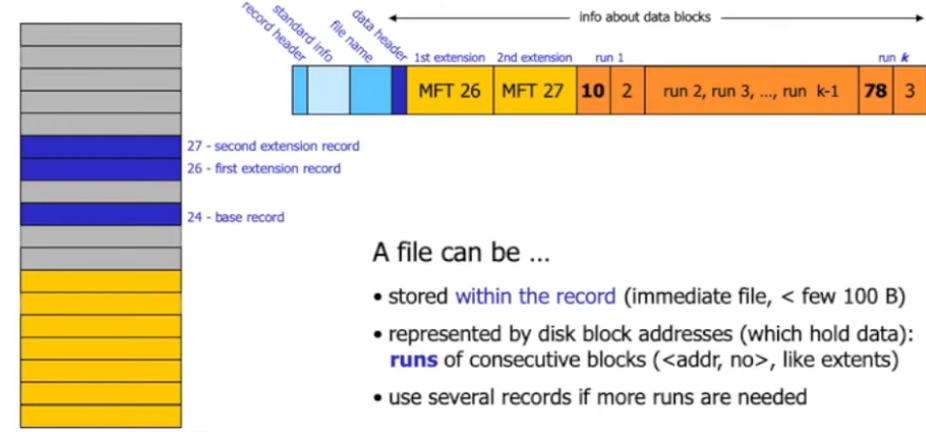
- **Extent-based allocation:**

- Samle filene så godt som mulig. Da kan vi optimalisere metadatastrukturen.
- Metadataen inneholder en peker til starten av filen i tillegg til en teller for hvor mange blokker filen inneholder.
- Raskere allokering
- Færre oppslag, kan lese data kontinuerlig (færre context switches?)
- Metadataen er mindre
- Brukt i XFS, JFS, EXT4, med en ny inode struktur.
- **EXT4:**
 - En inode inneholder et array av pekere.
 - Arrayen av pekere bruker de allokerete bytesene til å først beskrive en header (hva slags pekere er det, direkte eller indirekte), så de neste bytesene på å lage extents (pekere) som inneholder 4 felter.
 - Hvilket blokknr pekeren er
 - Den fysiske addressen på disk'en (delt opp i to variabler, 1x32 bit + 1x16 bit)
 - Hvor mange elementer ligger etter hverandre på disk'en.
 - Uten indireksjon kan man kun addressere $4 \times 128\text{MB} = 512\text{MB}$ med en extent. Løsning: definere i header om extenten er et HTREE (grunt og bredt for raske oppslag) av extents.
 - I dette tilfellet peker inoden på den fysiske addressen på disk til en blokk som inneholder blokkstørrelse (f.eks. 4KB) * størrelsen på en extent. Extentene på disk'en kan eventuelt også være flere pekere til andre extents som vil gi oss enda større addresserom i bytte mot hastighet.

- **NTFS:**

- Hver partisjon har en **Master File Table** som er en linær sekvens av 1KB records. Hver record beskriver en dir eller en fil, med attributter og addressene.
- Første 16 bytes er reservert for metadata til filsystemet.
- Hver record har metadata og runs (samme som extents).
- Hvis en record ikke er nok for en fil lages ekstra records kalt extension records. Indeksene i MFTen for ext recordsene legges så inn i base

recorden.



A file can be ...

- stored [within the record](#) (immediate file, < few 100 B)
- represented by disk block addresses (which hold data): **runs** of consecutive blocks (<addr, no>, like extents)
- use several records if more runs are needed

Flere diskar:

Striping:

- Fordeler datablokker på flere diskar og lese fra alle diskene parallelt.
 - Raskere transfer rate enn ved en disk
 - Kan ikke betjene flere brukere samtidig
 - Hvis diskene har forskjellig hastighet kan man måtte vente på at den tregeste blir ferdig
- **Interleaving (Compound Striping):**
 - Ikke aksesser alle samtidig, men fordeler forespørslene over diskene og les i sekvens.
 - Lag disk grupper som kan brukes parallelt for flere brukere (en bruker per disk).
 - Hvis flere brukere prøver å aksessere samme disk samtidig har vi samme problem - løsning: staggered striping.
- **Staggered striping:**
 - Hvert element 'stripes' på tvers av diskene, dvs. at f.eks. kan du ha halvparten av et element på en disk og resten på en annen.
 - Flere brukere kan betjenes parallelt.
 - Mer effektive diskoperasjoner
 - Muligens raskere responsid.
 - Flere bruker kan fortsatt prøve å aksessere samme diskgruppe.
- **Redundant Array of Inexpensive Disks (RAID):** - **RAID 0:** stripet disk array uten redundans, bare for ytelse uten sikkerhet. - **RAID 1:** speiling, dvs. 2 kopier av det samme innholdet som gjør at vi kan lese fra 2 diskene samtidig. - **RAID 2&3:** lite brukt - er på bitnivå. - **RAID 4:** striping på alle utenom den ene diskene. Den siste diskene brukes som paritetsdisk. Det gjør at dersom den ene diskene går i stykker kan man gjenopprette fra paritetsdisken. - Skriveoperasjoner kommer til å gå til paritetsdisken for å oppdatere backupen, dette gjør at paritetsdisken blir en flaskehals. - **RAID 5:** løsning på problem med RAID 4 - i stedet for å ha en designert paritetsdisk fordeler vi pariteten på tvers av alle diskene. - **RAID 6:** Sett av to typer paritetsblokker for å håndtere 2 feil samtidig. - Man kan også lage kombinasjoner av disse f.eks. RAID 10 eller RAID 0+1

Hva med filsystemer over nettverk?:

- **Network Attached Storage (NAS):** diskene og filsystemet er på en fjern server.
- **Storage Area Network (SAN):** bare diskene er på en fjern server, filsystemet lagres lokalt.
- **Network File System (NFS):** Pakker funksjonskallet sendt fra applikasjonen og sender dem ned til kommunikasjonssystemet og sender det over til RPC

mottakeren som pakker funksjonallet opp og sender det videre til NFS mottakeren som kaller det lokale filsystemet og går ned på disken. Når dataen er lest inn pakkes det ned igjen og sender det tilbake til avsenderen av forespørselen.

Inter-Process Communication (IPC):

- Hvordan kommunisere imellom prosesser på samme maskin? Her kan man anvende mer effektive løsninger enn det som brukes for kommunikasjon over nettverk.
- Intro:

Message Passing

- What is message-passing for?
 - communication across address spaces and protection domains
- What we need (generic API):
 - send(dest, &msg)
 - recv(src, &msg)
- What should the "dest" and "src" be?
 - pid
 - file, e.g., a pipe
 - port: network address, etc
 - no dest: send to all
 - no src: receive any message
- What should "msg" be? We need ...
 - buffer to store the message
 - size ? ... for variable sized messages
 - type ? ... to distinguish between messages

- Indirekte kommunikasjon: Man forholder seg ofte til et mellomledd både avsender og mottakere sender til og leser fra (se på det som en mailbox).
- Mailboxes: Meldinger køet opp som FIFO for hver ID. Meldinger kan også gis typer. Hvis man ønsker å hente ut en spesiell type fra en kø må man ta hensyn til dette og passe på at man ikke henter ut feil type, må også oppdatere køen

dersom det forespurte elementet ikke faktisk var først i køen.

Mailboxes

- Mailboxes are implemented as message queues sorting messages according to FIFO
 - messages are stored as a sequence of bytes
 - system V IPC messages also have a type:

```
struct mymsg {  
    long mtype;  
    char mtext[...];  
}
```
 - get/create a message queue identifier: `Qid = msgget(key, flags)`
 - sending messages: `msgsnd(Qid, *mymsg, size, flags)`
 - receiving messages: `msgrcv(Qid, *mymsg, size, type, flags)`
 - control a shared segment: `msgctl(...)`

- Pipes:

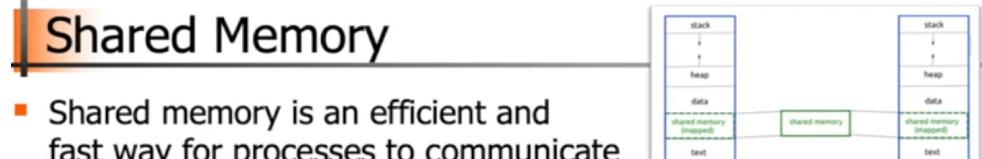
- Oppretter nye prosesser for hver kommando. | kaller på systemkallet pipe.
- Hvis vi ser på kommandoen `ls -l | more` vil systemkallet Pipe opprette et minneområde og 2 fildeskriptorer, en for lesing og en for skriving. Vi får også en inode for pipen. Denne inoden vil ha en peker til pipen, en lengde, en start, samt informasjon om lesere og skrivere til pipen. Så setter den i gang prosessen til `ls` som vil skrive ut outputen sin til det dedikerte minneområdet for pipen. Mens den skriver outputen sin vil den også endre på lengden i inoden til pipen. Hvis vi oppnår maksimal lengde før `ls` er ferdig vil vi måtte vente. Når `more` da skal begynne å lese ut fra pipen bruker den start pekeren i inoden og leser så mye som den klarer, eller helt fram til lengde variablen. Etter den er ferdig med å lese oppdaterer den start variabelen til der den glapp av og senker lengde variabelen med like mye som den leste.

- Mailboxes vs. Pipes:

Mailboxes vs. Pipes

- Are there any differences between a mailbox and a pipe?
 - Message types
 - mailboxes may have messages of different types
 - pipes do not have different types
 - Buffer
 - pipes – one or more pages storing messages contiguously
 - mailboxes – linked list of messages of different types
 - More than two processes
 - a pipe **often** (not in Linux) implies one sender and one receiver
 - many can use a mailbox

- Shared memory:
 - Når to brukere har hver sine addresserom oppretter de et fysisk minneområde som mappes inn i addresseromene til begge brukerne. Da kan begge brukerne både skrive til og lese fra dette minneområdet. Utfordring: overskriving av hverandres variabler.



- Shared memory is an efficient and fast way for processes to communicate
 - multiple processes can attach a segment of physical memory to their virtual address space
 - create a shared segment: `shmid = shmget(key, size, flags)`
 - attach a shared segment: `shmat(shmid, *shmaddr, flags)`
 - detach a shared segment: `shmdt(*shmaddr)`
 - control a shared segment: `shmctl(shmid, cmd, *buf)`
 - if more than one process can access a segment, an outside protocol or mechanism (like semaphores) should enforce consistency/avoid collisions
- Signals:
 - To måter å sende signaler: kill() og raise()
 - For å håndtere signaler: signal()
 - Man kan redefinere hva et signal skal gjøre ved å peke til en egendefinert funksjon ved fanging av signaler
`(signal(sig_nr, void (*func) ()))`. Når man gjør dette vil ikke default betydningen utføres, heller denne egendefinerte funksjonen.

Datakommunikasjon:

Nettverk strukturer:

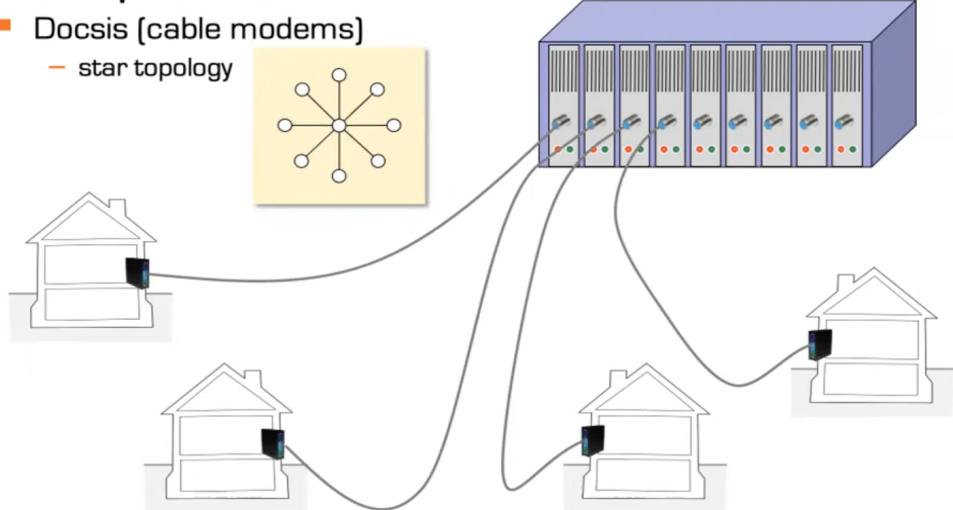
- Nettverk komponenter:
 - **Endesystem:** Noder som ikke frakter nettverkpakker videre til andre noder. F.eks. datamaskiner, gateleys, branndetektorer, osv.
 - **Intermediate system:** Noder som viderefører pakker til andre noder. F.eks. modem, ruter, switcher, web proxy, osv.
- Nettet kan ses på som en graf der det er kanter mellom noder. Noder vil være et endesystem eller et intermediate system og en kant vil være en kobling mellom to noder.

- **Point-to-point channels:**

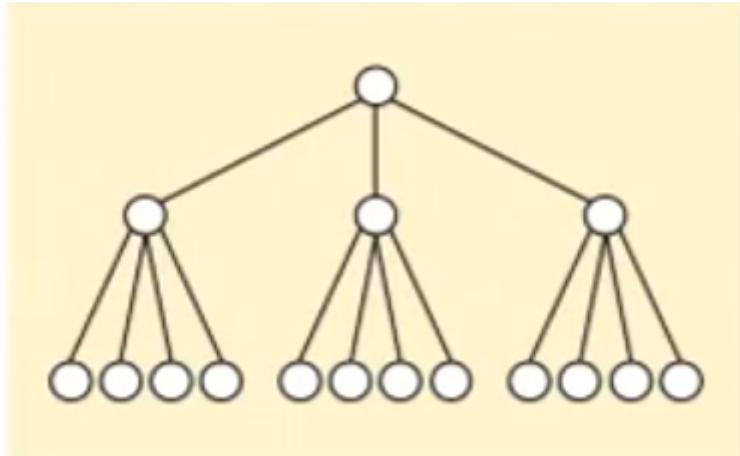
- Docsis: Et sentralt intermediate system som kobler sammen alle endesystemene i et nettverk. Stjerne topologi

Point-to-point channels

- Docsis [cable modems]
 - star topology



- Gigabit ethernet: Stjerne eller tre topologi.



- Andre strukturer:

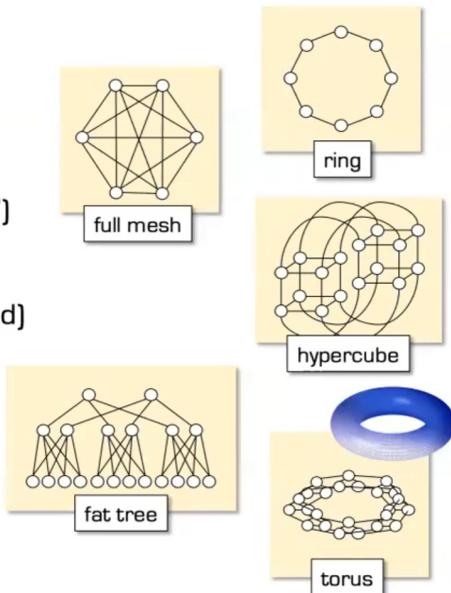
- Full mesh: alle noder er koblet mot hverandre - for mange kabler og tilkoblinger når det er mange noder.
- Hypercube: Mye brukt
- Torus: God skalerende topologi ettersom en node aldri trenger flere kanter dersom nettverket vokser.

- Ring: Ikke brukt

Network Structures

Point-to-point channels

- Docsis [cable modems]
 - star topology
- Gigabit Ethernet ("1GB Ethernet")
 - star or tree
- IEEE 802.5 "TokenRing" [outdated]
 - ring
- Some supercomputers use
 - full mesh
 - hypercube
 - torus
 - fat tree



- **Broadcasting kanaler:** Hvis en node sender kan alle andre høre på den. Problem: hvis to meldinger sendes samtidig kan de kollidere - dette må håndteres. Her kan man f.eks. sende et signal om at meldingen har blitt ødelagt og må sendes på nytt.
 - Kabel: Gammel ethernet brukte dette
 - Radio: Typisk bruk for dette i dag - trådløst. For å hindre kollisjoner: ikke sende med mindre mottakere har sagt at de vil høre etter.

Nettverkets oppgaver:

- Hvert par av intermediate systems må kunne veien til hverandre for å kunne videreføre data gjennom grafen.
- Nettverket må vite hvordan man skal finne veien fram til de riktige endesystemene. Her må de også finne veien til den riktige prosessen på det mottakende endesystemet.
- Nettverket må vite veien tilbake fra et endesystem til et annet
- Finne riktig måte å encode data slik at maskiner kan forstå hverandre (f.eks. hvis en little endian og big endian maskin kommuniserer)
- Må kunne finne alternativer dersom noe går galt. F.eks. hvis et IS ødelegges må det finne en alternativ sti gjennom grafen.
- Opprettholde sikkerhet og privatitet.
- Støtte høy traffik

Applikasjoners behov:

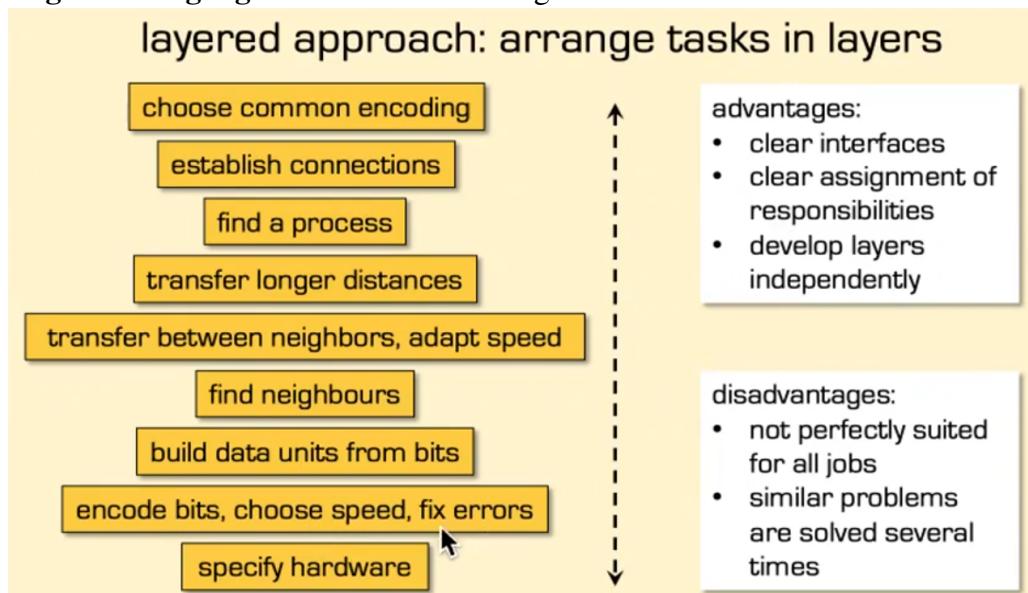
Filnedlasting:

- Unngå høy forsinkelse - Ikke kritisk (kan være irriterende men mindre forsinkelser er tolererbare)
- Støtte høy traffik - Ikke kritisk
- Håndtere nettverkproblemer - Kritisk - hvis man f.eks. skal overføre penger i banken må dette håndteres ordentlig om det skjer noe feil over nettet. #### Text-chat:
- Unngå høy forsinkelse - Ikke kritisk (kan være irriterende men mindre forsinkelser er tolererbare, mennesker klarer ikke skrive raskt nok for dette)
- Støtte høy traffik - Ikke kritisk

- Håndtere nettverkproblemer - Kritisk - Alle meldinger må komme fram til mottakeren ##### Streaming av media:
- Unngå høy forsinkelse - Noe er ok
- Støtte høy traffik - Kritisk
- Håndtere nettverkproblemer - Noe er ok (Hvis noen piksler mangler fra video er det tolererbart) ##### Haptisk interaksjon (f.eks. spill):
- Unngå høy forsinkelse - Kritisk (bruker må kunne reagere på hendelser fortløpende)
- Støtte høy traffik - Ukriktig (kommer an på tjenesten(??))
- Håndtere nettverkproblemer - Noe er ok (Hvis noe går tapt går det fint)

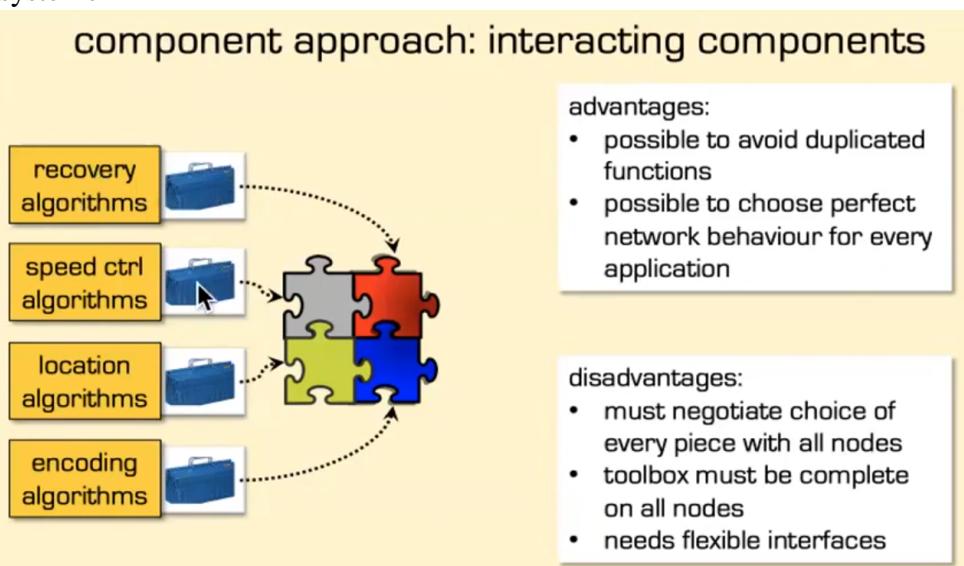
Strukturering av oppgaver:

- Lagdelt fremgangsmetode - Brukes i dag



- Komponent fremgangsmetode

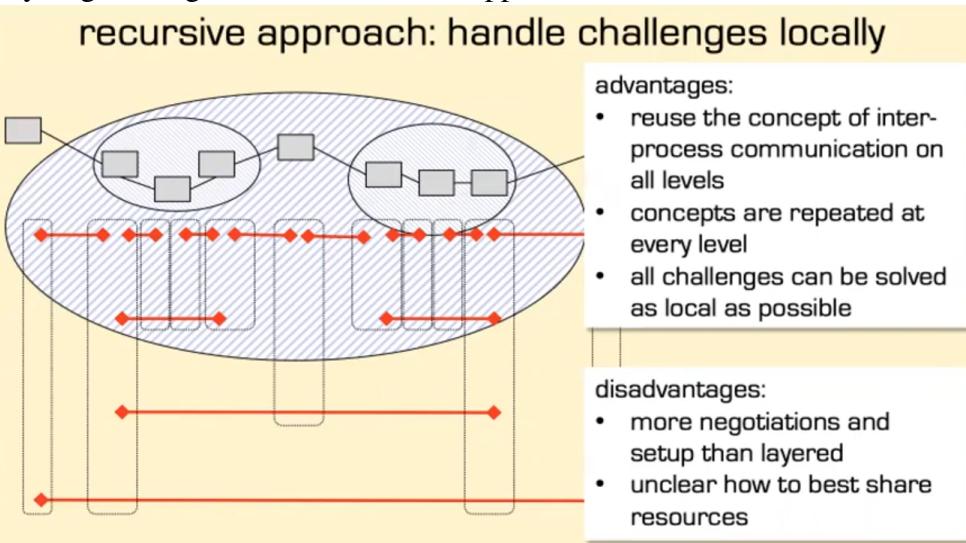
- Hvert lag kan dele komponenter - slipper å duplisere funksjoner
- Trenger veldig fleksible algoritmer for å kunne dele dem på tvers av lag
- Vansklig å bli enig om hva å velge over et større nettverk med mange ulike systemer



- Rekursiv fremgangsmetode

- Se på utfordringer lokalt først - hva trengs for å kommunisere med min direkte nabo?

- Mye signaleringsoverhead for å sette opp en sti



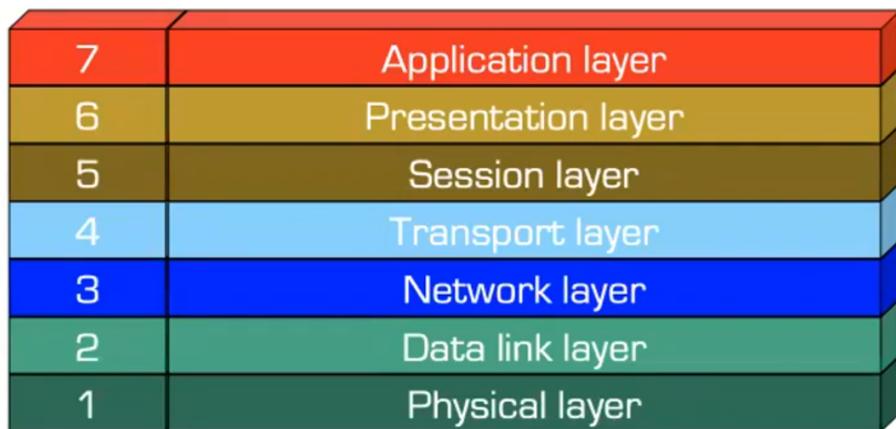
Layering model:

- To modeller: ISO OSI (Open Systems Interconnection) Reference Model og TCP/IP Reference Model Internet Architecture.
 - OSI er mer teoretisk, TCP/IP brukes i praksis

OSI:

ISO OSI (Open Systems Interconnection) Reference Model

- model for layered communication systems
- defines fundamental concepts and terminology
- defines 7 layers and their functionalities

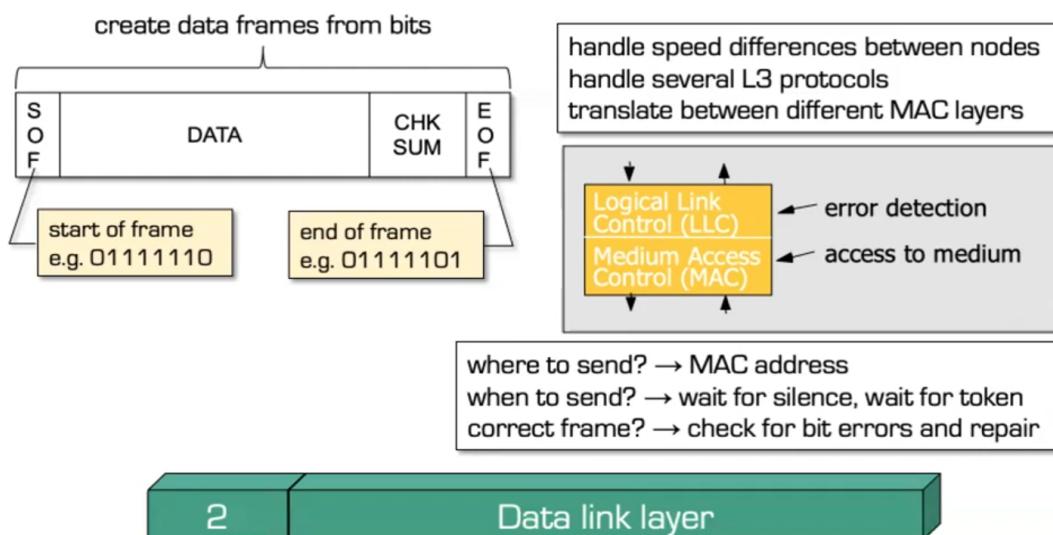


- Fysiske laget: Sende bits mellom to direkte naboer i et nettverk - kan være wifi/4g/kabel osv. - Hvordan koble nettverksadapterne
- Hvilke elektroniske kretser trengs (ikke med implementasjon) - Hva slags prosedyre trengs for å kommunisere med naboen (send melding ut for å spørre om naboen er klar for å motta pakker før den sender, enten over kabel eller med radiobølger) - Linklaget: Skaper større enheter ut av bitsene (bytes) gjør dem om til frames (rammer) - En ramme består av: en start og en slutt (markeres ved et kjent bitmønster), en mengde data, og en checksum som hjelper med oppdagelse av mulige

feil som kan oppstå under transport. - Ansvar deles opp i to underliggende lag Logical Link Control (LLC) (rettet mot lag 3, detekterer feil i rammer og muligens forkaster ved feil) og Medium Access Control (MAC) (Vi forholder oss mest til MAC, rettet mot lag 1, definerer hvordan en ramme ser ut). LLC muliggjør blant annet at vi kan ha et og samme internet for både trådfast og trådløst selv om de begge krever separate MAC lag, siden vi i dette tilfellet bare har et LCC lag. MAC laget finner nabo noden med sin MAC adresse, så bestemmer den om meldingen skal sendes enda eller om den skal vente på andre noder i nettet for å unngå kollisjon og bestemmer hvor stor rammene skal være.

Layer functions: data link layer

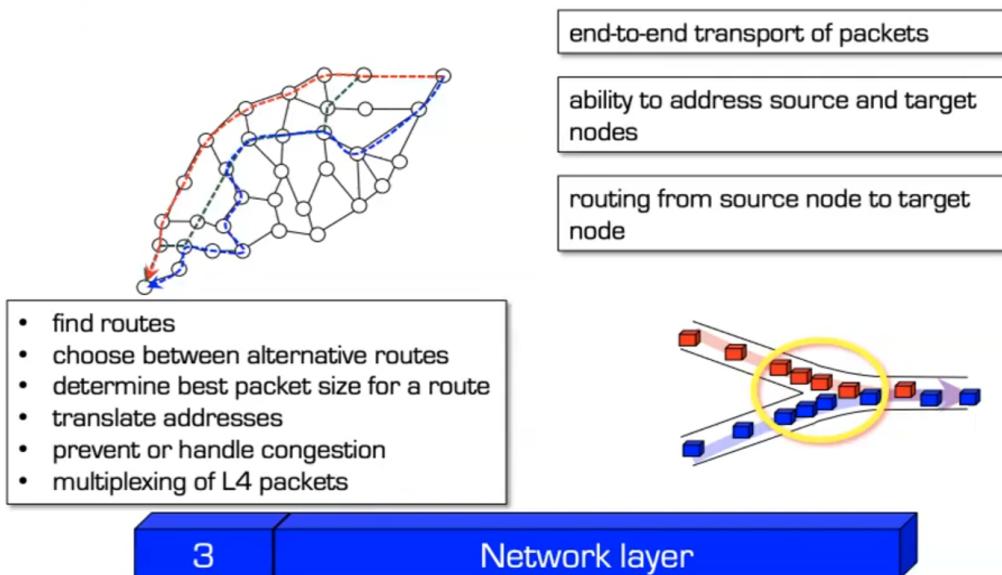
Responsibility: error-recovering frame stream, adjacent systems
Reliable data transfer between adjacent stations with frames



- Nettverklaget (IP): Har ansvar for å sende pakker uavhengig av hverandre fra et endesystem til et annet. Finner hvordan addressene ser ut og hvordan man finner en sti fra en adresse til en annen - dette kalles routing. Laget abstraherer nettverket til en graf sånn at den kan bruke algoritmer for å finne den *beste* stien avhengig av en rekke parametre. Hvis addressene ser annerledes ut må nettverklaget også oversette dette. Hvis for mange pakker bruker samme sti kan nettverklaget også fikse dette - i værste fall forkaste noen pakker, beste fall finne en annen sti. Nettverklaget har også ansvar for metning(skontroll?). Det vil si at dersom det er flere avsендere til samme mottaker som gjør at ikke alle pakker kan sendes i full hastighet må nettverklaget håndtere dette - lage et minnebuffer der pakkene kan ligge fram til de kan sendes? Forkaste noen pakker? Til slutt ansvar for å levere pakker opp til lag 4.

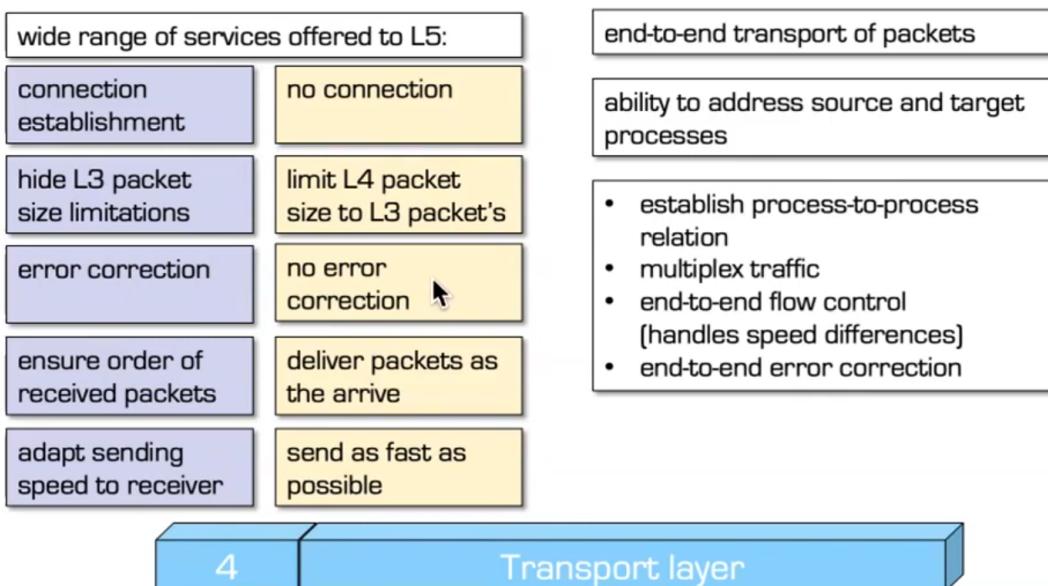
Layer functions: network layer

Responsibility: packet stream between end systems



- Transportlaget (TCP/UDP): Nettverkaget sender bare pakker mellom systemer - transportlaget skal så finne hvilke prosesser pakkene skal fram til. Legger til ekstra informasjon nødvendig for å finne prosessene når den sender pakker med nettverkaget. Skaper en relasjon mellom prosessene på ulike endesystemer. Dersom du har to prosesser som henter data fra samme endesystem må transportlaget håndtere hvilken prosess som skal motta hvilke pakker fra det andre endesystemet. På samme måte må transportlaget også tilby applikasjoner som sender flere strømmere med data til samme endesystem en måte å skille mellom formatet på de ulike dataene. Transportlaget håndterer også hastighetsforskjeller mellom avsender og mottaker. Dette kan gjøres ved å redusere hastigheten avsenderen kan sende pakker eller ved å forkaste pakker. Laget fikser også feil på endesystemer uten innflytelse på stien gjennom nettet.

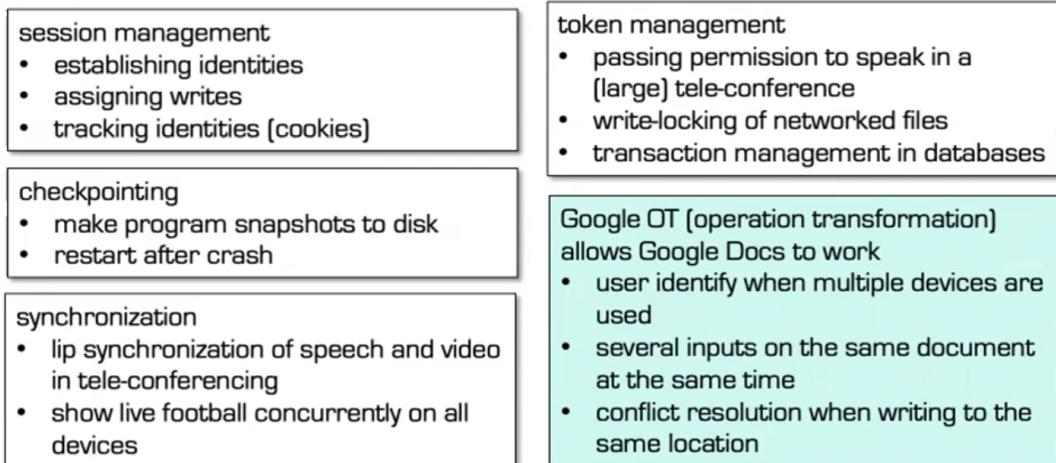
Responsibility: end-to-end message stream between processes



- Session layer: Skaper en strukturert dialog mellom prosessene på endesystemene. Sesjoner skal støttes over lengre perioder - også på tvers av ulike nettverk. Identifierer brukere og passer på hvilke tilganger brukerne har.

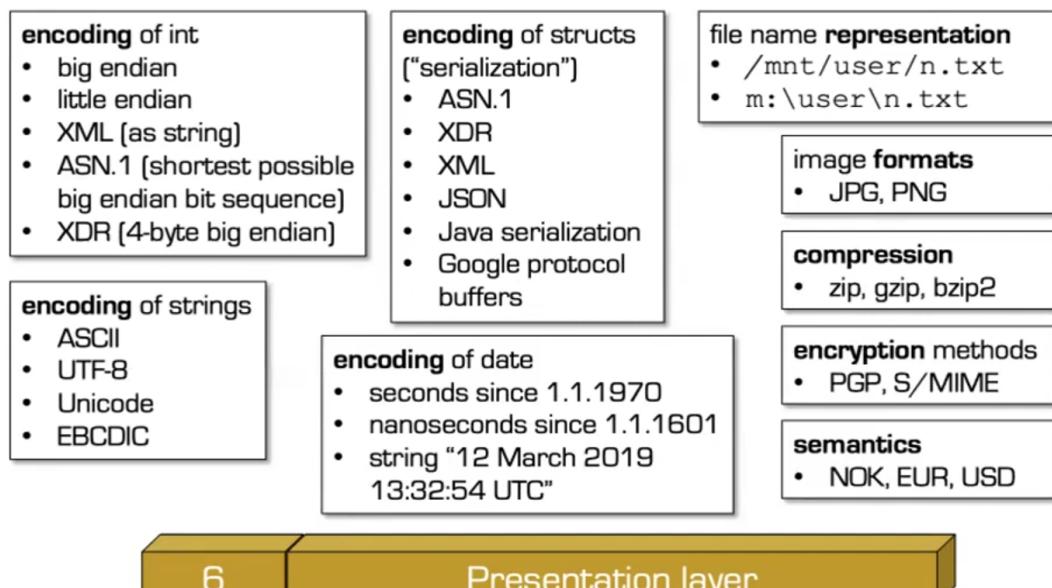
Responsibility: structured dialogue

support a "session" over a longer period



- Presentasjonslaget: Skal fikse ulikheter mellom endesystemer.

Responsibility: exchange of data (semantics!)



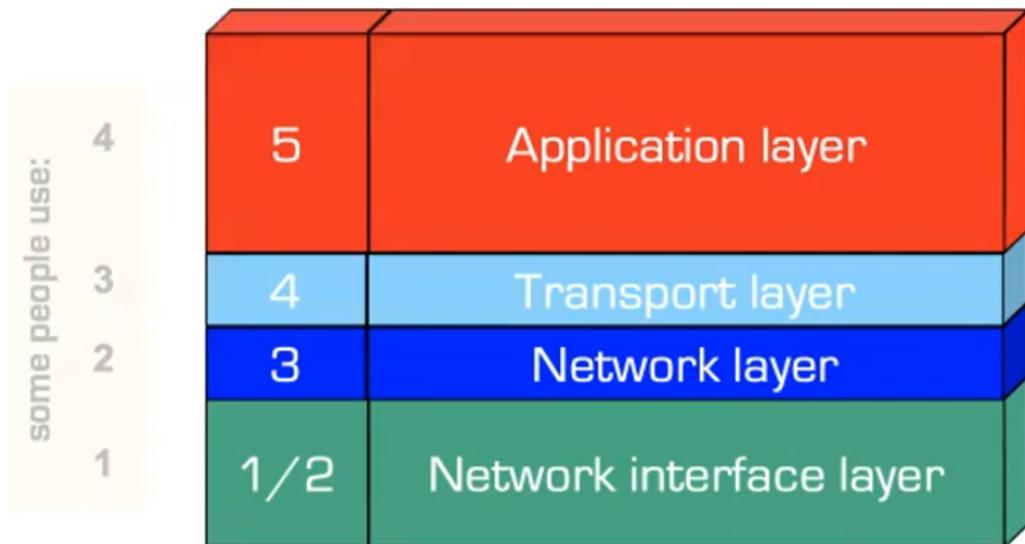
- Applikasjonslaget: Alt annet! Benytter seg av funksjonalitet fra lagene under.
Utvikleren kontrollerer hva som gjøres her.

- I mellom to endesystemer er hvert lag en *peer*. Det vil si at alle protokollene i hvert lag må passe på tvers av endesystemer. Fysisk vil kommunikasjon mellom endesystemer gå ned gjennom lagene for å sendes fysisk av en avsender, før det kan rekonstrueres gjennom de samme protokollene av mottakeren. Dersom det skal sendes gjennom et intermediate system vil ISet kun rekonstruere pakkene opp til nettverkaget ettersom det ikke trenger å vite noe mer for å kunne videresende relevant data.

TCP/IP modellen:

- Laget i konkurranse med OSI. Består kun av 5 lag - ingen session og presentasjonslag. I tillegg kombineres ofte lag 1 og 2 til et lag (Network

interface model).



- Nettverkslag og transport lag funker som i OSI, men applikasjonslaget må ta seg av oppgavene til presentasjonslaget og sesjonslaget.
- Det er kun lag 1-4 som er essensielle for å kunne kommunisere over internettet så disse vil forbli noenlunde det samme uansett. Disse lagene er også som regel definert i kjernen. Altså kan man ikke endre deres funksjonalitet direkte fra våre applikasjoner. De andre øverste 3 lagene er det utviklerne har kontroll over. Siden det er utviklerne som bestemmer hvordan dette skal implementeres er det ikke nødvendig å opprettholde standardiserte protokoller for disse lagene. Det er derfor TCP/IP modellen ikke har noen spesifikke retningslinjer for disse lagene.

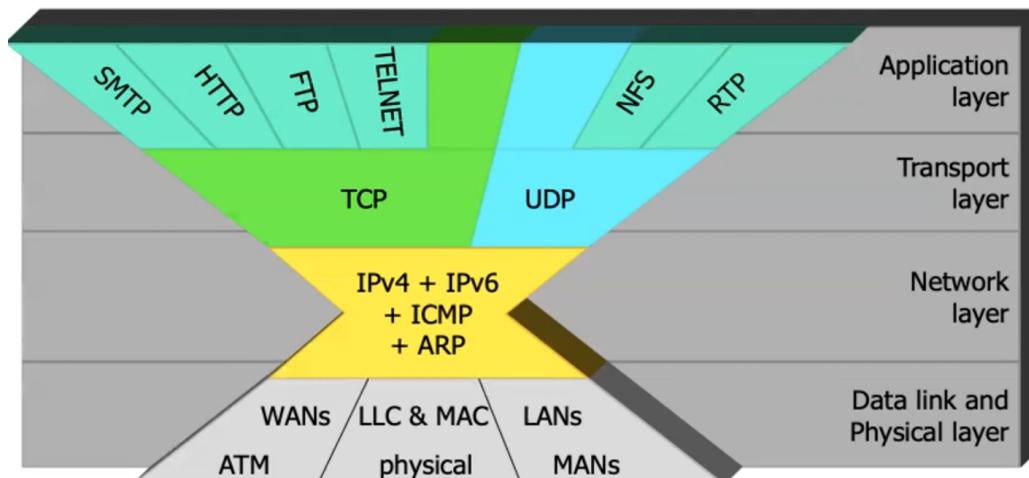
TCP/IP Layering considerations

Why no clear separation of upper layers?

- layers 1-4 are essential for co-existence on the Internet
 - e.g. different congestion control mechanisms on different hosts can lead to strong congestion
- session and presentation layer functions provide mostly application support

Layers 3 and 4 are not clearly separated

- transport protocol [TCP, UDP, others] and network protocol IP
- sometimes hard to draw a clear line where TCP ends and IP begins
- example:
 - Early Congestion Notification (ECN) capability is indicated on layer 3 and congestion is indicated on layer 3
 - sender is told about receiver's reception of congestion signal on layer 4



Nickname: "Hourglass Model"

Naming:

- **(N)-layer:** Abstraksjonsnivå for lag med definerte oppgaver tilbyr tjenester til laget over.
- **(N)-entity:** Implementasjonen (brukerne?) av protokollene definert for et lag. Aktive elementer på et lag. Kjørende kodebiter som benytter seg av maskinens ressurser. Entiteter som snakker med hverandre på samme lag er **peer entities**.
- **(N)-Service Access Point:** Et lag benytter seg av tjenestene til laget under og tilby tjenester til laget over. Hva slags tjenester et lag tilbyr er definert i denne NSAPen. Dette er en beskrivelse av disse tjenestene, interface, API. Et lag kan tilby flere NSAPer. F.eks. i lag 4 velger vi mellom to hovedprotokoller (TCP/UDP).
- **(N)-Protocol:** N-entiteter interagerer med hverandre gjennom en N-protocol. Strenge og entydige regler. Protokollen definerer hvilken rekkefølge meldinger må sendes i - en entitet vil først finde en forbindelse til en annen entitet på samme lag på en annen maskin, så kan de etablere et forhold for hvordan de kan kommunisere (f.eks. hvor mye data de kan overføre på en gang) før de til slutt kan kommunisere som vanlig. En protokoll må også definere hvordan pakker skal sendes og hvordan unntak skal håndteres (f.eks. en pakke når ikke fram).

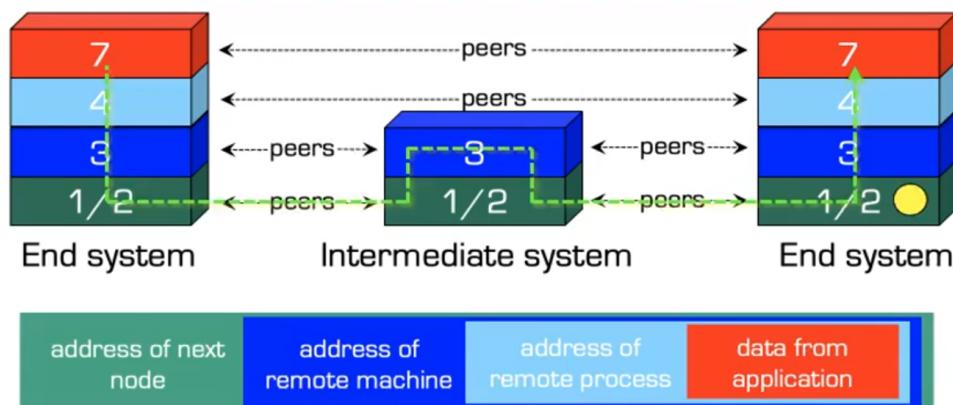
Definerer **ikke** hvilke tjenester som leveres til laget over eller hvordan disse tjenestene skal leveres.

- Protocol syntax: regler for formattering
- Protocol semantics: regler for håndtering av uforventede hendelser
 - Begge disse må være likt mellom kommuniserende entiteter
- PDU: Melding mellom to entiteter på samme lag som utfører en protokoll med hverandre
 - Mer spesifikke begreper:
 - Symbol: Lag 1
 - Frame: Lag 2
 - Packet: Lag 3
 - Message/datagram (segment?): Lag 4
 - Kombinasjon av andre begreper: Lag 5

Hvordan sende pakker:

- Sending av brev analogi: Man addreserer mottakeren og sender den gjennom flere ledd før den kommer fram. Hvert mellomledd pakker ut litt og litt av addresseinformasjonen (analogisk: pakker ut kontinent, så land, så adresse, så person).
- Hver gang data sendes fra nivå N til N-1 legger nivå N til sine N-protokoller (som en header, her kan den også legge inn en checksum bakerst for å forsikre at dataen er valid). Hver gang data sendes fra nivå N til nivå N+1 pakker nivå N ut dataen fra nivå N-1.

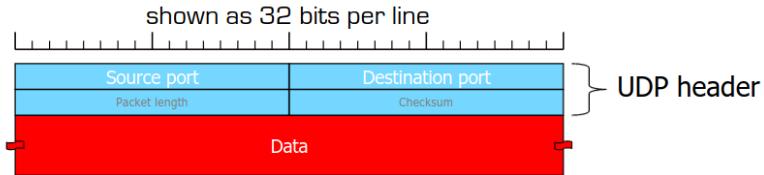
Data flow through the network



- Each sending N-entity at layer N adds N-protocol information
- ... which is important for its peer N-entity
- ... and the receiving N-entity removes it before passing the data to layer N+1
- Transport lag header (UDP): Bits sendes fra øverst til venstre ned til nederst til høyre. 64 bits/8 byte header. En port er en adresse en prosess fra lag 7 kan reservere for å bli identifisert etter mottak av en pakke fra den andre entiteten på lag 4 den skal sende til. Porten er internettets prosessadresse. Source port er prosessadressen til avsenderen. Destination port er prosessadressen til

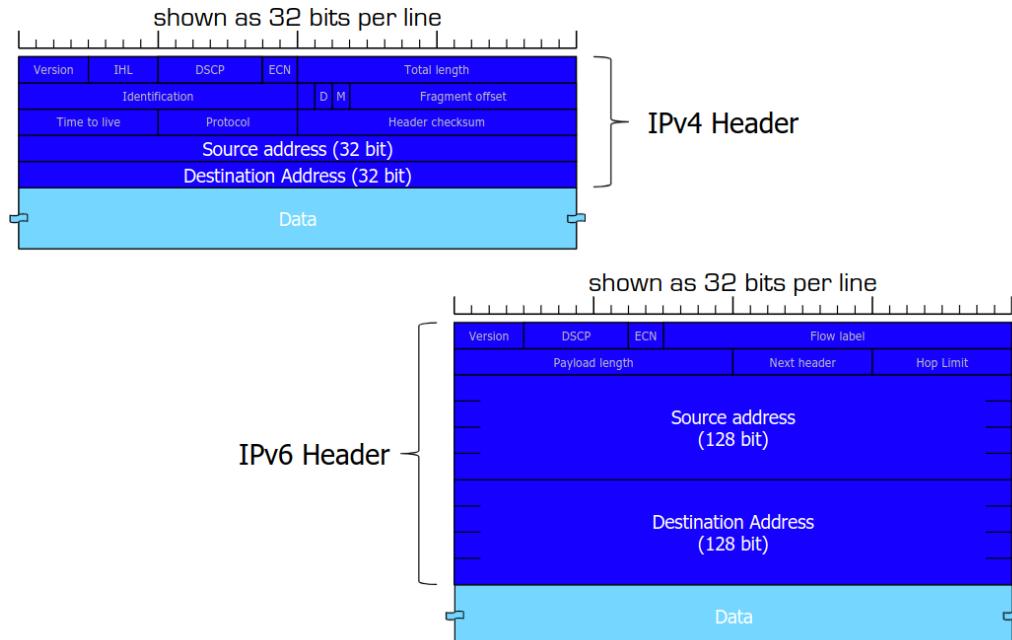
mottakerprosessen. IPv4 kan ignorere checksum delen av UDP headeren.

Transport layer header: UDP example



- **port**
- the term in Internet protocol for the address of a process on an end system
- the transport layer address
- Note: there are several transport layer protocols in the TCP/IP world, UDP is shown because it has the smallest header
- Network lag header (IPv4 og IPv6): Version: sier om pakken er IPv4 eller 6 - mer om disse headersene senere

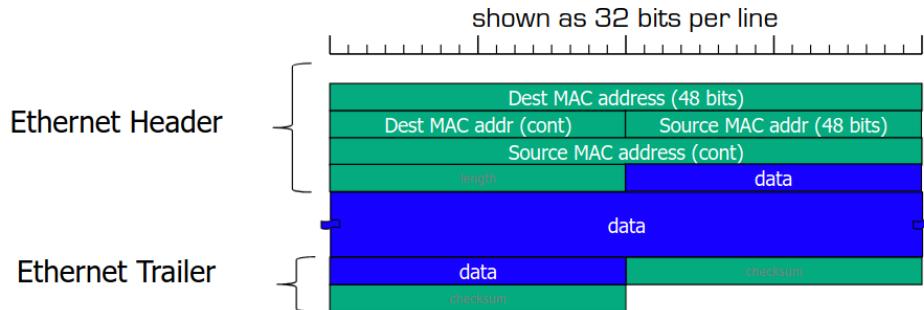
Network layer headers: IPv4 and IPv6



- Data link lag header (Ethernet): MAC addressene kan addressere både en IS eller et ES. Checksum for å sjekke at dataen har blitt overført riktig mellom

nabonodene.

Data link layer headers: Ethernet example



Network byte order:

Big vs Little Endian

■ Representing numbers

- the decimal number 36
- can be seen as $1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1$
- is identical to binary 100100
 - we prefer to think in whole bytes, and may write 00100100
- is identical to hexadecimal $24 = 2 \cdot 0x10 + 4 \cdot 0x1$
 - for clarity we write 0x24

- it is hard to transform directly from decimal to binary
but easy to transform from hexadecimal to binary

$$\begin{aligned} & 00100100 \\ \Leftrightarrow & 0010 : 0100 \\ \Leftrightarrow & 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 : 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1 \\ \Leftrightarrow & 2 : 4 \\ \Leftrightarrow & 0x24 \end{aligned}$$

sufficient to
think about
4 bits at a time!

Argument for Big Endian

■ compatible with western-world writing direction

- but when we use memory like this:

```
unsigned char byte[8];
byte[0] = 0x81;
for( int i=1; i<8; i++ ) byte[i] = 0;

unsigned char* ptr1 = (unsigned char*)&byte[0];
printf("%x\n", 1 + *ptr1);
unsigned short* ptr2 = (unsigned short*)&byte[0];
printf("%x\n", 1 + *ptr2);
unsigned int* ptr3 = (unsigned int*)&byte[0];
printf("%x\n", 1 + *ptr3);
unsigned long long* ptr4 = (unsigned long long*)&byte[0];
printf("%llx\n", 1 + *ptr4);
```

- Argument for big endian: Intuitiv skrivemåte for å skrive tall i vestlig skrivemåte der man skriver tall fra venstre til høyre - tall lengst til venstre har størst verdi. I en big endian maskin, hvis man putter endrer en `unsigned char byte[8]` sin

`byte[0]` med `+=1` endrer man byten lengst til høyre med en (se bilde).

Argument for Big Endian

- compatible with western-world writing direction

- but when we use memory like this:

```
unsigned char byte[8];
byte[0] = 0x81;
for( int i=1; i<8; i++ ) byte[i] = 0;

unsigned char* ptr1 = (unsigned char*)&byte[0];
printf("%x\n", 1 + *ptr1);
unsigned short* ptr2 = (unsigned short*)&byte[0];
printf("%x\n", 1 + *ptr2);
unsigned int* ptr3 = (unsigned int*)&byte[0];
printf("%x\n", 1 + *ptr3);
unsigned long long* ptr4 = (unsigned long long*)&byte[0];
printf("%llx\n", 1 + *ptr4);
```

Big Endian

82

8101

81000001

8100000000000001

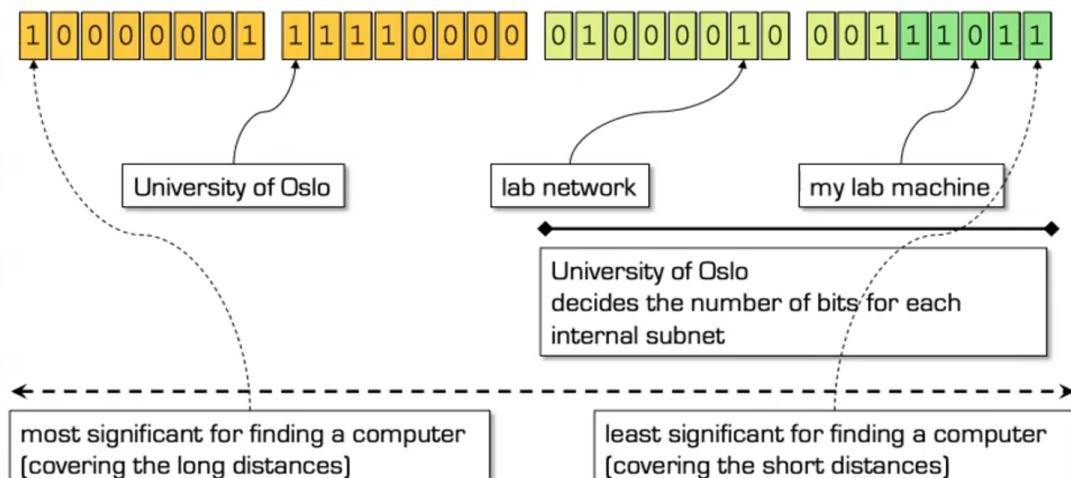
- I tillegg ønsker man når man kommuniserer over nettet å kunne sende de mest signifikante bitsene over nettet først siden da kan man allerede begynne å prosessere dataen før man har mottatt alt (analogi: telefonnummer - man kan se at telefonnummeret ligger i USA etter å ha lest 0d001... med big endian, mens hvis man hadde brukt little endian hadde man måttet lest hele telefonnummeret før man kan se landskoden). Denne analogien gjelder også for IP addreser ettersom de virker i stor grad på samme måte som telefonnummer (se bilde). OBS: pass på ved bruk av network byte order(big endian) på little endian maskiner ettersom dette kan skape bugs, bruk funksjoner for å konvertere byte order.

Bonus for Big Endian

- my lab machine in our lab network

129.240.66.59

0x81 F0 42 3B



- Argument for little endian: Type casting i minnet på little endian maskiner er mye mer effektivt siden man uavhengig av hvor mange bytes som er satt av for en variabel alltid kan aksessere de første n bytesene av variabelen på samme måte (f.eks. hvis

man har en long for tallet 0x81 og ønsker å konvertere dette til en short kan man bare ta de første 2 bytesene av longen og få det samme tallet)

Argument for Little Endian

■ easy to transform

- when we use memory like this:

```
unsigned char byte[8];
byte[0] = 0x81;
for( int i=1; i<8; i++ ) byte[i] = 0;

unsigned char* ptr1 = (unsigned char*)&byte[0];
printf("%x\n", 1 + *ptr1);
unsigned short* ptr2 = (unsigned short*)&byte[0];
printf("%x\n", 1 + *ptr2);
unsigned int* ptr3 = (unsigned int*)&byte[0];
printf("%x\n", 1 + *ptr3);
unsigned long long* ptr4 = (unsigned long long*)&byte[0];
printf("%llx\n", 1 + *ptr4);
```

81 00 00 00 00 00 00 00

Little Endian

82

82

82

82

■ cheap and easy to change the number of bytes used for an integer value

harder for the human mind
but faster to process

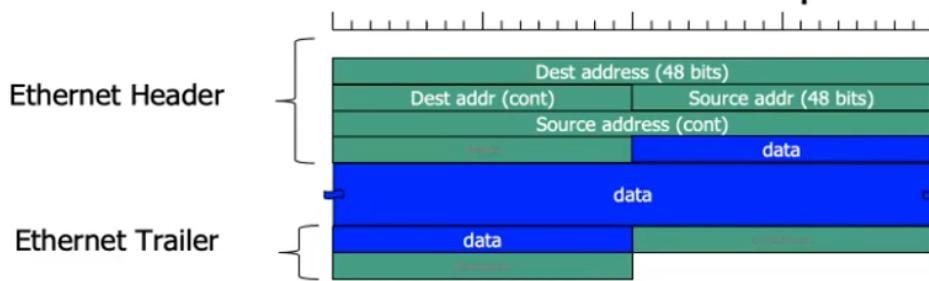


Addressering med MAC:

- MAC addresser brukes for kommunikasjon mellom to direkte naboer. Dette kan brukes for maskiner koblet til samme nettverk. For kommunikasjon med maskiner på andre nettverk brukes IP addresser. MAC addresser lagres ofte også i maskiners cache for å kunne kommunisere raskere.
- MAC addresser er ikke nødvendig i et true point-to-point nettverk. Dette er fordi hver gang lag 2 sender en pakke mha. tjenester fra lag 1 vil pakken gå rett til den fysiske forbindelsen som kommer til å sende bits til nabomaskinen og lag 2 addressen er ikke nødvendig fordi man kjenner identiteten til pakken kommer til å sendes til. Til tross for dette kan man fortsatt bruke MAC addressen i slike tilfeller siden de kan brukes for å identifisere maskiner over lengre perioder hvis de f.eks. kobler seg fra nettet og så kobler seg på igjen. Da slipper man å måtte oppdatere informasjonen om hvordan kommunisere med den noden på andre maskiner.
- Men for ekte broadcast nettverk der en maskin sender og andre lytter er det nødvendig å kjenne addressen til maskinen man ønsker å nå. Da kan senderen legge inn mottakeradressen sånn at nodene på nettverket skjønner hvem som skal lese hvilke pakker.
- Det siste intermediate systemet som skal videresende pakken til et endesystem må også kunne finde riktig endesystem. Da må vi oversette fra IP adresse (lag 3) til Netadapter adresse (lag 2, f.eks. ethernet adresse). Dette må også gjøres for hver overgang fra et system til et annet, da må vi også ta hensyn til ulike datalinklagsprotokoller underveis som kan ha ulike hastigheter, underspesifikasjoner og lag2 addresseringsstiler (Wifi og ethernet har faktisk samme her, men DSL har forskjellig). Derfor kan vi ikke sende med informasjon om MAC addresser i pakkene på lag 3/4, dette må finnes ut lokalt.
- Noen maskiner kan omskrive sin MAC adresse + virtuelle maskiners MAC addresser er forskjellig fra maskinen de kjører på så den globalt unike IDen er

ikke perfekt.

Address resolution: Ethernet example



■ MAC address structure

- Ethernet and WiFi are L2 layers using "EUI-48":
Extended Unique Identifier with 48 bits
- 6 bytes, written like this: f2 : 18 : 98 : 3a : b8 : 97
- to recognize easily that the text is supposed to mean a MAC address

■ Ethernet MAC addresses should be globally unique

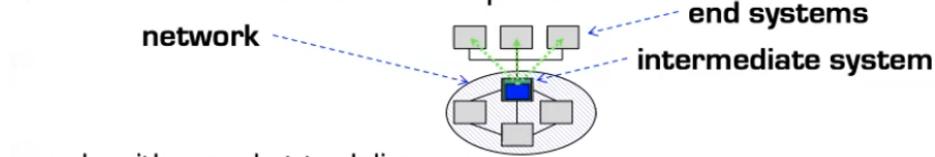
• Hvordan oversette MAC til IP?

- **Direct mapping:** Legge IP addressene fra lag 3 inn i MAC addressen. Siden IPv4 er 32 bits og MAC addreser er 48 bits kan man ta hele IP addressen og legge det inn i MAC addressen, de første 16 bitsene i MAC addressen kan da være en prefix for lokale maskiner på det nettet. Da kan maskiner finne ut hvem pakken er ment til basert på IP addressen(?). Denne metoden virker ikke over internett siden IP addressen vil være for endesystemet, ikke den neste noden pakken skal sendes til.
- **Mapping table:** Hver node har en tabell for IP addresser og MAC addresser som kan oppdateres og slås opp i for å se hvilke MAC addresser passer til IP addressen fra lag 3 headeren(?) før pakken kan sendes videre. Det finnes to måter å generere denne tabellen:
 - Manuelt vedlikehold: Mye arbeid men ikke urealistisk.
 - Hver gang en ny maskin kobles til internettet broadcaster de sin MAC og IP adresse sånn at alle mottakere kan oppdatere sine tabeller. OBS: maskiner som kobles til senere vil ikke ha noen informasjon om andre maskiner på nettverket.
- **Address resolution protocol (ARP):** Hver maskin på nettet vedlikeholder en cache over sine direkte naboer som oversetter maskiners MAC addresser til IP addresser. Cachen benytter en timeout som gjør at dersom en maskin ikke har vært kontaktet over en gitt timeout sletter vi den fra cachen. Når en node ønsker å nå en gitt IP adresse i et nettverk mha. ARP vil avsenderen broadcaste *ARP requesten* over nettet som forespør den tilsvarende MAC addressen til IP addressen. Hvis noen svarer med sin MAC adresse oppdaterer avsenderen og mottakeren sine caches med hverandres data for lettere kommunikasjon i fremtiden, frem til forbindelsen timeouter. Andre

noder ignorerer forespørsele.

Address Resolution

3rd idea: address resolution protocol



node with a packet to deliver:

```

if a local cache contains IP address ↔ MAC address
    send packet & update cache removal timeout
else
    send broadcast to all stations
    "Who has IP address?"
    if one node responds
        add IP address ↔ MAC address mapping to cache
        set timeout for removal from cache to some minutes
        send packet
    else
        drop packet

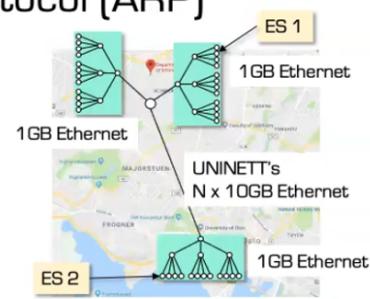
```

- Hva med kommunikasjon over større nettverk, da vil det jo forekomme enorme mengder ARP requests?

- Da kan vi bruke **proxy ARP**. Dette går ut på at vi deler opp nettverket i mindre nettverk. Hvert nettverk vil da ha en router som kjenner til de andre nettverkene som cacher et mapping table. Når noen broadcaster en ARP request og ingen maskiner på det lokale nettverket svarer vil det gå til routeren som sjekker om den allerede har den forespurte MAC adressen. Hvis den ikke har det vil den sende en forespørsel til de andre nettverkene. Hvis noen av de andre nettverkene svarer vil den oppdatere cachen sin sånn at neste gang noen broadcaster en ARP request slipper routeren å videreføre denne forespørselen til de andre nettverkene.
- Proxy ARP er mulig men lite brukt i dag. I stedet anser vi bare hvert mindre nettverk som et separat nettverk som er koblet sammen gjennom en router. Når et endesystem da skal sende en melding bruker den IP addressen til mottakeren og sender pakken til routeren som kobler oss til det andre nettet som inneholder mottakernoden.

Address Resolution Protocol (ARP)

- End system not directly available by broadcast
- Example: ES 1 to ES 2
 - ARP would not receive a response
 - Ethernet broadcast is not rerouted over a router
- L2 solution: proxy ARP
 - the local router knows all remote networks with their respective routers
 - responds to local ARP
 - local ES 1 sends data for ES 2 always to the local router, this router forwards the data (by interpreting the IP address contained in the data)
- L3 solution: remote network address is known
 - local ES 1 sends data to the appropriate remote router
 - local router forwards packets



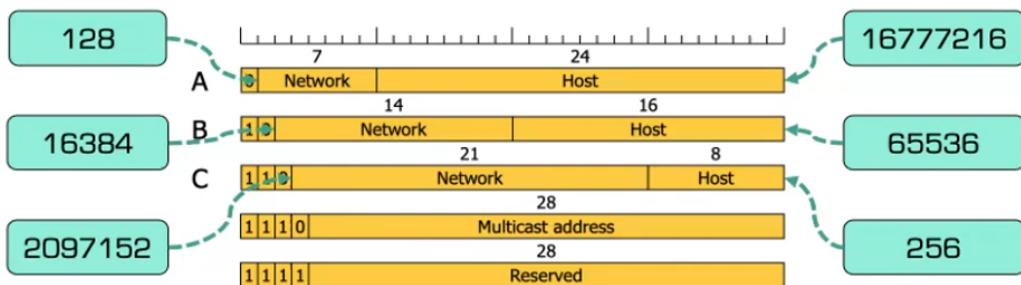
- **Reverse Address Resolution Protocol (RARP):** I stedet for å forespørre MAC addresser basert på sin IP adresse kan man forespørre IP addreser basert på sin MAC adresse. Når et system kobler seg på internettet med RARP broadcaster den en *RARP request* for å finne sin IP adresse. Da vil *RARP serveren* utdele en IP adresse så lenge den er tilgjengelig på LAN. Dette brukes heller ikke noe særlig i dag.

Addressering med IP:

- Internettet består av subnetverk som deles inn i 5 klasser basert på de første bitsene i IP addressen. Disse nettverkene er også koblet til hverandre.

Internet Addresses and Internet Subnetworks

- Original global addressing concept for the Internet
 - For addressing end systems and intermediate systems
 - each network interface (not ES) has its own unique address
 - 5 classes



- ICANN (Internet Corporation for Assigned Numbers and Names)
 - manages network numbers
 - delegates parts of the address space to regional authorities
- En nettverkadresse definerer hvem som tilhører hvilket nettverk, ikke hvordan nettverk er sammenkoblet. For å finne stien fra et nettverk til et annet *Routing*.
- Hvordan strukturere nettverk?
 - Gammeldags metode: Ha et stort nettverk hvor adresering til nabøer gjøres på lag 2, f.eks. med proxy ARP. Men denne metoden holder ofte ikke siden nettverk ofte vokser utover den tildelte nettverksklassen.
 - Alternativt: Dele opp nettverket i flere mindre deler som ligger under det større nettverket og bruke *routing* for å addressere noder internt. Men hvordan kan man skille de mindre nettverkene fra det store? Fjerne klasseindelingsbetydningen til den første biten i addressen. I stedet lager vi en lokal indikator for hvor mange bits vi bruker til nettverk/host. For å gjøre dette deler vi opp bitsene tidligere brukt for host og deler den opp i en seksjon for undernettverk og en mindre andel bits for host. Da må vi også finne en måte å bestemme hvilke bits som definerer nettverket/subnetverket og hvilke som definerer hosten på nettverket - for dette bruker vi en subnet maske som er et tall med n antall 1 bits på rad i starten. Da kan vi gjøre en bitvis & operasjon mellom addressen og masken for å få ut subnet addressen. Hvis man router til subnet addressen kommer pakken fram til nettverksinngangen og man kan da se på resten av bitsene i addressen for å velge en maskin under subnettet. Routeren kan da bruke en algoritme for å gjøre 3 sjekker:
 - Hvis & operasjon mellom subnet maske og nettverksdelen av addressen er forskjellige betyr det at det er en pakke ment for et annet nettverk
 - Hvis & operasjon mellom subnet maske og nettverksdelen av addressen er like og & operasjon mellom subnet maske og subnetdelen av addressen er like er det en pakke som skal til det lokale systemet.

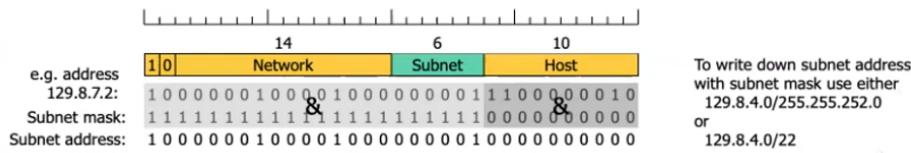
- Hvis & operasjon mellom subnet maske og nettverksdelen er like, men & operasjon mellom subnet maske og subnetdelen av addressen er ulike er det en pakke til et annet subnetverk.

Internet Address and Internet Subnetworks



■ Idea

- local decision for subdividing host share into subnetwork portion and end system portion



- Use "subnet mask" to distinguish network and subnet part from host part

- Computers inside the network 129.8.4.0/22 can make 3 checks

- Algorithm in router (by masking bits: AND between address and subnet mask):
 - packet to another network (& with Network mask are different)
 - packet to local end system (& with Network and Subnet masks are the same)
 - packet to other subnetwork (& with Network mask is the same, but Subnet is not)

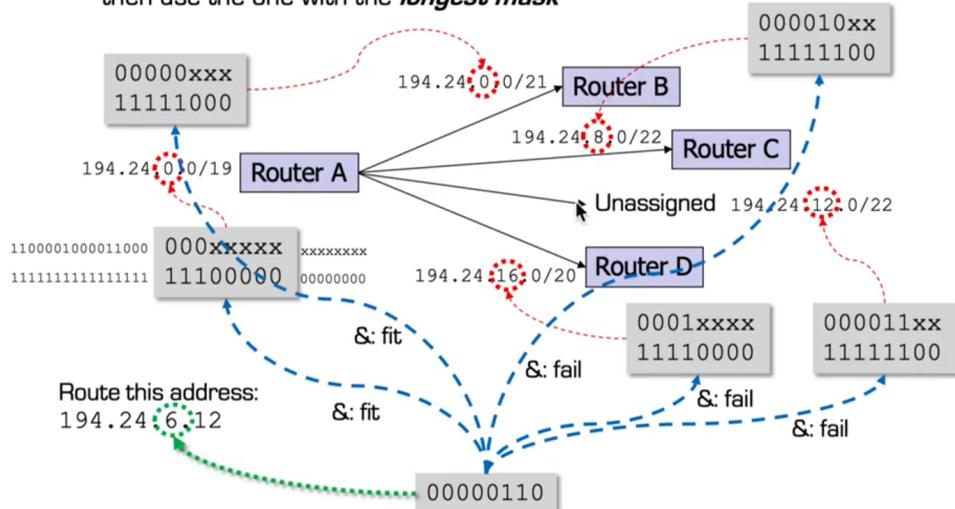
Classless InterDomain Routing (CIDR):

- Subnetverk er ikke bra nok
- Klasse A nettverk med 16 millioner addreser for mye i mange tilfeller
- Klasse C nettverk med 256 addreser for lite i mange tilfeller
- De fleste organisasjoner er intereserte i klasse B nettverk, men de er det ikke nok av (16384)
- Derfor kan vi i stedet ta ut deler av et
- I stedet for å ha et stort nettverk med en organisasjon som forvalter det kan vi ta noen av det nettets subnetverk og skille dem ut for å selge dem separat fra dets tidligere nettverk. For å identifisere disse nettverkene kan vi fortsatt bruke den samme subnetmasken. Antall mulige hosts under en adresse er lik $2^{\text{antall 0 bytes i subnet masken}}$ dette kan brukes for å fordele nettverk med passende størrelser.
 - Med denne løsningen kan du ha flere nettverk med samme bits i starten, men med forskjellige subnet og subnetmasker. For å finne hvilket nettverk en pakke skal til i slike tilfeller bruker man **longest mask** prinsippet. Det vil si at da sendes pakkene til det nettverket med lengst subnetmaske som fortsatt returnerer en identisk streng etter & operasjon.
 - For å bestemme hvilket nettverk en adresse tilhører må vi da gjøre & operasjoner med addressen og alle nettverk/subnettverk under ansvarsområdet til det originale nettet de splittet seg fra. I bildet under passer addressen til Router A, Router B, IKKE Router C (den femte biten i den tredje byten i addressen til Router C er 1, i addressen vi skal route må den være 0), IKKE unassigned og IKKE Router D. Da må vi velge den

routeren addressen passer til som har lengst maske som er Router B (21bits).

CIDR: Classless InterDomain Routing

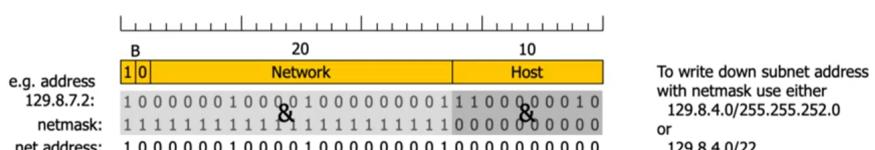
- Rule of the longest match
 - if several entries with different subnet mask length may match then use the one with the **longest mask**



- Netmaske separerer nettverk delen av addressen fra host delen av addresser.
 - For å få nettverk addressen fra en IP adresse med cidr kan man ta & operasjon mellom IP addressen og netmasken.
 - Hvordan skal endesystem bestemme om de skal sende pakker direkte til andre endesystem eller om de skal sendes til routeren, og hvordan bestemmer en router om den skal sende til en maskin den har ansvar for/en annen router? - Ta destination adresse og gjør en & operasjon med netmasken fra sitt eget nettverk interface.
 - For endesystemer: Hvis resultatet av dette er likt subnet addressen kan man sende pakken til et lokalt endesystem (kanskje først ARP forespørsel for å finne systemet og så sende). Hvis ikke likt - send til nettverkets router.
 - For routere: Spiller ikke noe rolle om pakken er lokal. Hvis & operasjon er lik subnet adresse - send direkte til endesystem (kanskje først oppslag i ARP cache/ARP forespørsel så sende til MAC adresse). Hvis ikke likt oppslag i routing tabell så send til annen router.

CIDR: Classless InterDomain Routing

- Idea
 - local decision for subdividing host share into network portion and end system portion



- Use "netmask" to distinguish network part from host part
- Routing with 3 levels of hierarchy
 - **end system**: compute "dst addr & netmask == subnet addr"
 - TRUE: packet to local end system [perhaps ARP, then deliver packet]
 - FALSE: packet to another network [send to this network's router]
 - **router**: compute "dst addr & netmask == subnet addr"
 - TRUE: packet to local end system [perhaps ARP, then deliver packet]
 - FALSE: packet to another network [look up in routing table, send to other router]

IPv6:

- Ingen får permanente addresser, de må lånes ut og byttes regelmessig. En fiks for dette er **NAT**, men den er ikke ideell siden det innebærer at vi ikke bruker en én til én oversettelse mellom IP addresser og maskiner.

IP Version 6 (IPv6)

- Motivation for IPv6: problems with IPv4
 - Too few addresses
 - Bad support for QoS
 - Bad support for mobility
 - Many other shortcomings ...
- Mål med IPv6:
 - Øke skalerbarhet
 - Fikse begrensninger med ipv4
 - Integrere sikkerhet i ipv6
 - ipv4 pleide å bruke 3 bits for å indikere noen flagg for å effektivisere kommunikasjon, men dette måtte reinterpretert for andre mål. Med ipv6 skulle dette komme tilbake med traffikklasser og flow labels (identifisere om pakken var en del av en strøm eller separate enheter)
 - Multicasting: Sende en pakke fra en maskin til flere mottakere samtidig, pakken kan kopieres i nettet sånn at avsenderen slipper å sende flere ganger.
 - Mobility: Addresser skal kunne flytte seg fra et nettverk til et annet uten å måtte forandre routing tabeller over hele verden.
 - Kunne eksistere i samspill med ipv4

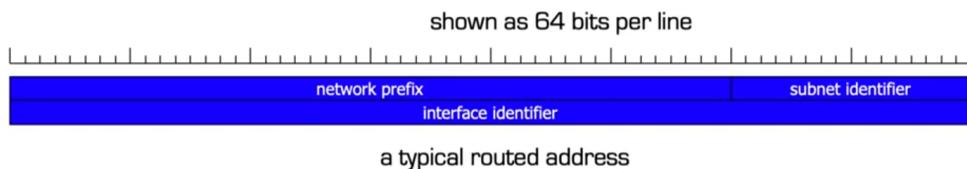
IPv6 Objectives

- To support billions of end systems
 - longer addresses
 - To reduce routing tables
 - To simplify protocol processing
 - simplified header
 - To increase security
 - security integrated
 - To support real-time data traffic
 - flow label, traffic class
 - To provide multicasting
 - To support mobility (roaming)
 - To be open for change (future)
 - extension headers
 - To coexist with existing protocols
-
- The diagram illustrates the objectives of IPv6 by grouping them into three main categories, each enclosed in a brace:
- Scalability:** This category includes the first three objectives: "To support billions of end systems", "To reduce routing tables", and "To simplify protocol processing".
 - Addressing IPv4 limitations:** This category includes the next four objectives: "To increase security", "To support real-time data traffic", "To provide multicasting", and "To support mobility (roaming)".
 - Coexistence:** This category includes the last two objectives: "To be open for change (future)" and "To coexist with existing protocols".

- **Headers:**
 - IPv6 header er alltid 40 bytes lang. IPv4 uten noen options er 20 bytes, med options opp til 60 bytes. IPv6 trenger også options, men dette ligger i lag 4 dataen.
 - IPv4 støtter fragmentering av pakker fordi det var nyttig før, men i dag kan det føre til problemer når man har fragmenterte pakker en mottaker må prosessere siden hvis det oppstår en feil må man kanskje forkaste alle de andre pakkene også som er bortkastet båndbredde. IPv6 støtter ikke fragmentering. I stedet forhandler sender og mottaker seg fram til en maksimal lengde, men som i dag er mye lengre enn i gamle dager, som man så kan tilpasse pakkelengden etter. Hvis dette gjøres feil går pakkene tapt.
 - IPv4 brukte også checksums for å verifisere at pakker hadde blitt sendt riktig. Dette er forkastet i IPv6, gjøres heller i lag 2&4.
- IPv6 addresser: Ganske like IPv4 addresser. Ingen regler for hvordan network prefix, subnet identifier og interface identifier er delt opp, bruker også nettmasker for å identifisere dette.
 - For å komprimere IPv6 addresser kan man fjerne en stor blokk av nuller og erstatte dem med ::. Merk at man ikke kan gjøre dette flere steder i samme adresse ettersom vi da ikke hadde kunnet vite hvor mange nuller hver :: erstatter. Man kan også droppe ledende nuller (f.eks. ...:080a:: -> ...:80a::)

IPv6 addresses

- example of the IPv6 address spaces



- IPv6 addresses are written in sets of 2 bytes in hexadecimal notation
- 2a00:1450:400f:080a:0000:0000:0000:2004

- sets of zero can be compressed:
- 2a00:1450:400f:80a::2004

So, its netmask has 48 1-bits followed by 80 0-bits

- this address is part of the network

2a00:1450:400f::/48

which is known to be used by Google since 12/2018

- Hvordan tildele IPv6 addresser? noe med generering basert på MAC addresser ..?

Acquiring IPv6 addresses for mobile computers

1. getting a non-routable IPv6 address using auto-configuration
 - self-assigns an IPv6 address consisting of prefix FE80::0 followed by the interface identifier [RFC4291], which is created from the MAC address [RFC 8064]
 - before *using* the address, "Neighbour Solicitation" ICMP message must be sent to ensure the address is not in use yet [RFC4443] – variation of ARP Probe
2. DHCPv6 to ask for a routable address
 - requires auto-configured local address first
3. Mobile IPv6 to transfer routable home address to visited network
 - requires auto-configured local address first

Acquiring IPv6 addresses for mobile computers

- example IPv6 address of jordin.ifi.uio.no:
fe80::250:56ff:fe80:3f82
- which is an abbreviation for
fe80:0000:0000:0000:0250:56ff:fe80:3f82
- for us, so far mostly worthless because UiO does not route IPv6 anyway

Addressering i transportlaget:

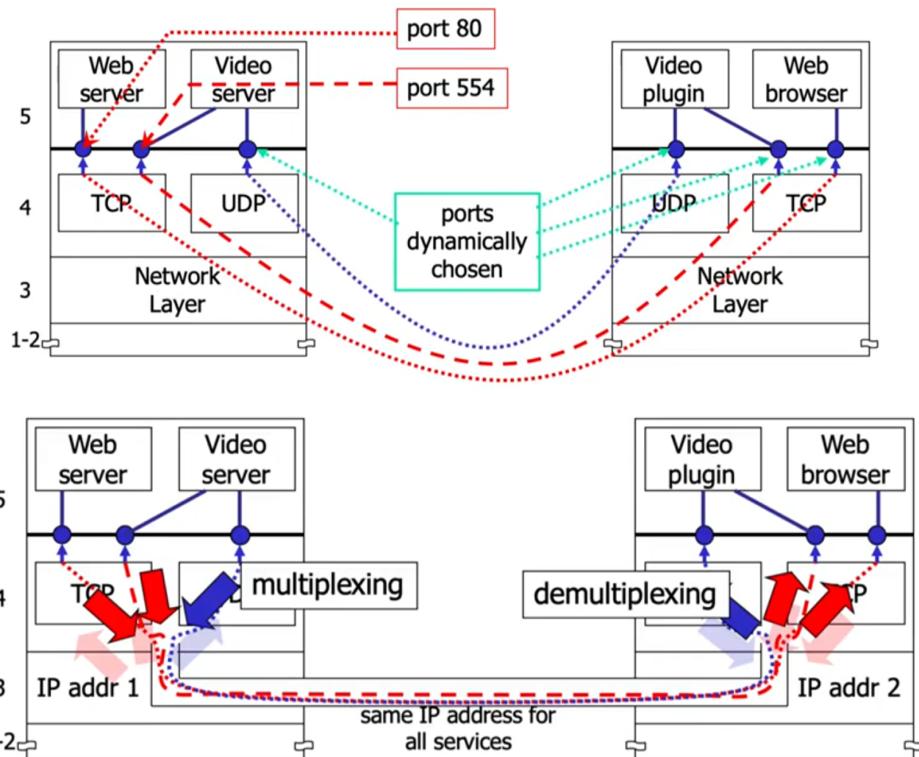
- Transportlagets oppgaver:
 - Addressere prosesser
 - Forvalte en forbindelse mellom applikasjoner (evt. sende uavhengige pakker (forbindelsesløst))
 - Overføre data mellom prosesser
 - Håndtering av feil
 - Pålitelig overføring av pakker
 - Flytkontroll sånn at applikasjonene ikke “drukner” i data
 - Metningskontroll sånn at ikke nettet får for mye data og pakker går tapt underveis
- Alle transportentiteter tilbyr et Service Access Point til applikasjonslaget. Applikasjonslaget er et interface som vi implementerer gjennom sockets. Addressen man tilbyr til applikasjonslaget defineres gjennom en port (16 bit integer).
- Kommunikasjon med andre prosesser går først gjennom nettverkslaget før det prosesseres gjennom transportlagsprotokollene (som regel TCP eller UDP) på den andre maskinen. Dataen som flyter mellom prosessene har flere navn: TCP - message, UDP - datagram, ISO: Transport Protocol Data Unit(TPDU), mer generelt kan vi si “pakker”.
- Hvis en applikasjon ønsker å addressere en mottaker-applikasjon kan man benytte seg av en Transport Service Access Point(TSAP) som inneholder informasjon om hva slags informasjon man må videresende til din lokale transportentitet for å kunne opprette en forbindelse.
- Applikasjoner trenger også vite hvilke ports de skal lytte på. For servere kan vi slå opp i faste lister over hva hvilke porter brukes for. Merk at mange av tjenestene i disse listene kun støtter en transportprotokoll (f.eks. SSH støtter kun TCP), så mange av portnumrene kan brukes for andre tjenester så lenge det bruker en annen transportprotokoll. For klienter er portene til serverne kjent. Når en klient skal koble seg til en server velger klienten en port dynamisk og

sender en `connect` forespørsel til serveren som inneholder denne dynamiskt valgte porten, da kjenner serveren porten til klienten.

- Eksempel - En web browser klient vil spille av en video fra en webside: Serverne venter på predefinerte porter for sine tjenester. Klienten velger dynamisk port (et tall over 1024) for sin browser og sender en `connect` melding over TCP som inneholder den valgte porten, da kjenner serveren også porten til browseren. Så vil klienten starte video pluginen for browseren som også trenger en kontrollkanal for å bekrefte forbindelse til video serveren (over TCP), denne velges også dynamiskt. Etter serveren har godtatt denne tilkoblingen og lagret porten til video pluginen til browseren, kan video pluginen og video serveren dynamisk velge en port for UDP kommunikasjon. De kan så sende en setup melding fram og tilbake for å lære hverandres porter som de til slutt kan bruke for selve videooverføringen over UDP. Nettverkslaget kan under denne prosessen overføre pakker fram og tilbake mellom klienten og serveren. Transportlaget har i oppgave å utføre multiplexing og demultiplexing over samme TSAP. Det vil si at transportlaget må sende og motta pakker over samme IP forbindelse og finne hvilke transportprotokoller hver pakke tilhører, hente ut portinformasjonen og videresende det til applikasjonslaget.

Multiplexing task of the Transport Layer

- Multiplexing and demultiplexing task of the transport layer
- Example: accessing a web page with video element
 - Three protocols used [minor simplification]
 - HTTP for web page
 - RTSP for video control
 - RTP for video data



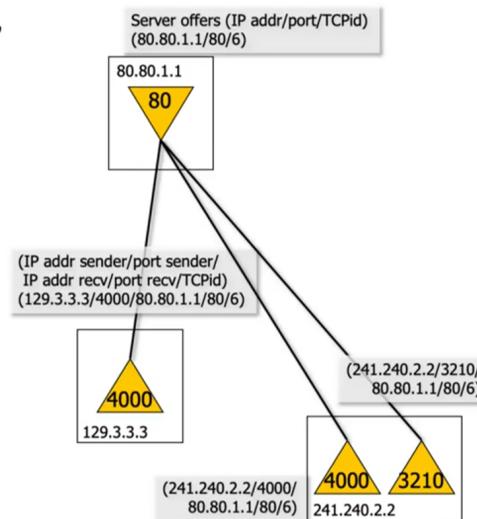
- ~64000 addreser for å tilby tjenester er ganske lite - hvordan utvide dette?
 - TCP:

- En TCP tjeneste tilbys gjennom en socket. En socket består av et *3-tuple* med 3 verdier: IP addressen til maskinen som tilbyr tjenesten, portnummeret for å finne fram til applikasjonen og en indikator om at socketen er for TCP.

- En TCP tilkobling består av et *5-tuple* av 5 verdier: IP addressen til sender og mottaker, portaddressen til sender og mottaker og en indikator om at socketen er for TCP.
- Med disse identifikatorene har vi vesentlig flere muligheter for å identifisere TCP tilkoblinger ettersom en tilkobling er unik så lenge én av disse faktorene er forskjellige.

Connection – Addressing

- **TCP service** obtained via service endpoints on sender and receiver
 - Typically socket
 - Socket number consists of a *3-tuple*
 - IP address of host and
 - 16-bit local number (port)
 - TCP protocol identifier
- Transport Service Access Point
 - Port
- **TCP connection** is clearly defined by a *5-tuple* consisting of
 - IP address of sender and receiver
 - Port address of sender and receiver
 - TCP protocol identifier
- Applications can use the same local ports for several TCP connections
 - if the remote side is different



Metningskontroll og flytkontroll:

- End to end principle: Endesystemer vet best hva de trenger så nettet burde blande seg inn så lite som mulig. F.eks. hvor mange pakker man skal sende per tidsenhet er en del av ansvarsområdet til transportlaget.
- To delutfordringer: Avsender sender for raskt for mottakeren (flytkontroll/flow control) eller avsender sender for raskt for nettverket (metningskontroll/congestion control). Begge fører til packet loss, men krever to separate algoritmer for å løses.
- Endesystem ønsker å øke sin egen ytelse, dvs. sende data til mottakeren så raskt og effektivt som mulig, uten hensyn til hva dette resulterer i for andre endesystemer/strømmer. Nettverket ønsker å øke total ytelse, uten å egentlig bry seg om ytelsen til individuelle endesystemer. Det vil si at nettverket gjerne skaper ulemper for noen endesystemer til fordel for andre. Dette er et problem som må løses av en uavhengig tredjepart - gjerne transportlagsimplementasjonen i kjernen.

- Tjenester som tilbys av lag 4 protokoller: Multihoming - kan sende til flere mottakere samtidig

	UDP	DCCP	TCP	SCTP
Connection-oriented service		X	X	X
Connectionless service	X			
Ordered			X	X
Partially Ordered				X
Unordered	X	X		X
Reliable			X	X
Partially Reliable				X
Unreliable	X	X		X
With congestion control		X	X	X
Without congestion control	X			
Multicast support	X	X		
Multihoming support				X

Flytkontroll:

- Kan gjøres på lag 2 og lag 4.
- Stop and wait: Enklest. Sender venter på ACK fra mottaker før den sender neste pakke. Men dette er ikke nok, siden pakker kan gå tapt. Her må man fikse to problemer. 1: fikse tapte pakker, 2: skille mellom tapte pakker fra sender (data) og mottaker (ACK).
 - For å fikse tapte pakker kan vi starte en timer hos senderen. Hvis timeren går ut og den ikke har mottatt en ACK kan vi sende pakken på nytt og start ny timer. Hvis vi mottar ACK etter vi har sendt en pakke kan vi gå videre til neste.
 - For å skille mellom tapt data og tapte ACK pakker kan vi legge inn et sekvensnummer som er enten 0 eller 1. Sendere starter med å sende en pakke med sekvensnummer 0. Hvis senderen ikke mottar noe ACK innen sin timeout vil den sende den samme pakken på nytt også med sekvensnummer 0. Hvis mottakeren mottar denne pakken og den ikke har sett den før (data pakken gikk tapt) send ACK med sekvensnummer 0, hvis mottakeren har sett denne pakken før kan den gjette at ACK pakken gikk tapt og sender også tilbake ACK med sekvensnummer 0. Da vil mottakeren

forvente at neste pakke er med sekvensnummer 1.

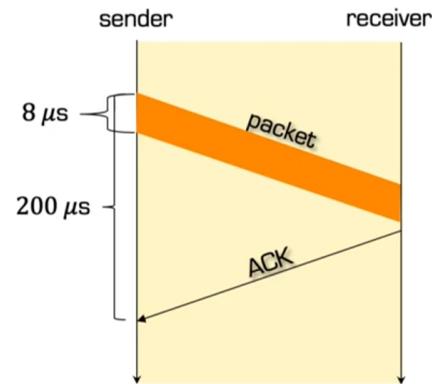
Flow control: Stop-and-Wait

Stop-and-Wait

- sender can never send new packet before ACK or timeout
- connection is idle most of the time
- poor **throughput**

Gigabit Ethernet channel

- transmission rate: 1 Gbps
- roundtrip delay 200 μ s [2 * 0.1 ms]
- packet size: 8000 bit
- in comparison
→ ACK is short and negligible



this means

- sending takes 8000 bit / 1.000.000.000 bps = 8 μ s
- sender is blocked for 192 μ s of 200 μ s
- Channel utilization 4%

- Sliding window: Pipelining, send flere pakker og acks samtidig.
 - To vinduer, et for sender (pakker som er sent men ikke acknowledged (in flight)) og et for mottaker (pakker som kan acknowledges)
 - Hvordan vi plasserer dataen i mottakerbufferet bestemmes ut ifra et sekvensnummer som sendes med hver pakke. Annerledes fra stop and wait må dette sekvensnummeret være litt større siden det gjerne skal leses inn i fler enn 2 posisjoner samtidig. Dette betyr at vi trenger en større header.
 - For senderen:
 - Algoritmen må ta vare på to variabler, lower bound og upper bound. Lower bound er det minste sekvensnummeret som ikke er acknowledged, upper bound er det neste sekvensnummeret som skal sendes. Hvis LB = UB er ingenting in flight og sending er idle. I tillegg lagrer vi en konstant for maksimal window size som sier hvor mange pakker vi kan ha in flight om gangen.
 - Senderen MÅ også ta vare på alle pakker som ikke har blitt ACKet for å kunne retransmitte ved feil
 - Når en ny pakke skal sendes må vi øke UB (modulo maks sekvensnummer+1).
 - Når vi mottar en ACK må vi øke LB (modulo maks sekvensnummer+1).
 - For mottakeren:
 - Mottakerbufferet trenger ikke å ta vare på pakker den allerede har lest, de kan sendes rett til applikasjonen før de sender ACK. Mottakeren burde hvertfall ikke ta vare på flere pakker enn window size.
 - UB starter som LB + window size.
 - LB starter som 0
 - Hvis LB == UB er bufferet fullt og vi har ikke klart å sende ut ACKer.
 - Hvis UB == (LB + window size) % maks sekvensnummer+1 har vi ingenting å ACKe og plass til window size nye pakker.
 - Når vi mottar en pakke må vi øke LB (modulo maks sekvensnummer+1), send ACK og øk UB (modulo maks sekvensnummer+1). ACK sendes bare hvis pakken er identifisert som korrekt og pakken kan transmitemmes på riktig måte til applikasjonen. Det vil si at pakken er i riktig rekkefølge og vanligvis at den ikke har noen bit errorer.

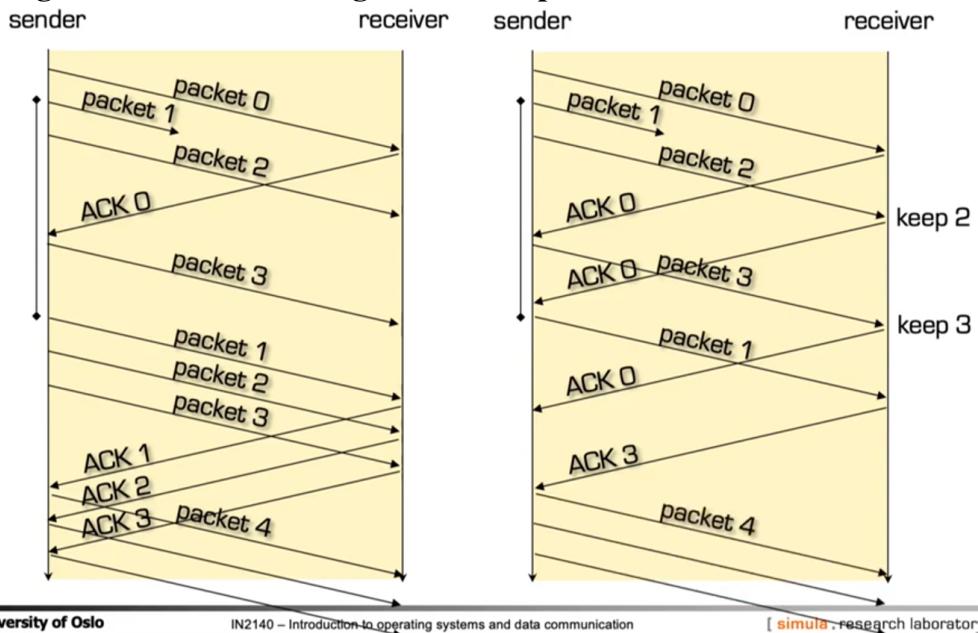
Cumulative ACK:

- Man trenger ikke sende ACK pakker for hver individuelle pakke man leser som mottaker, det holder å sende ACK for den nyeste pakken du har lest. Da forstår senderen at det innebærer at de andre pakkene også har blitt lest.
- ? Hvis f.eks. pakker 1,2,3 og 5 sendes og ACKes på riktig måte, men 4 går tapt, hopper vi over dette og går på sendersiden over til å sende pakke 6 som neste pakke.
 - Uten cumulative ACK i dette tilfellet: vent på timeout for pakke 4 og 5 før man retransmitter. ?

Retransmission med sliding window:

- To varianter:
- Mottakerens buffer har kun plass til den neste forventede pakken:
 - Hvis f.eks. pakke 0 og 2 sendes riktig, men pakke 1 går tapt vil mottakeren godta pakke 0, videresende det til applikasjonen og svare med ACK, men forkaste pakke 2 siden den forventet at neste pakke skulle være pakke 1. Når senderen mottar ACK for pakke 0 vil den prøve å sende pakker etter pakke 2, men disse vil alle forkastes. Når timeouten for pakke 1 slår inn vil senderen igjen sende pakke 1 som mottakeren så godtar og svarer med ACK. Etter dette kan kommunikasjon fortsette som vanlig.
 - Denne varianten kalles **Go-Back-N**. Forkortet: når en pakke feiler, forkast alle pakker etter den, gå tilbake til den pakken og send alt etter på nytt. Fordeler med dette er at mottakeren slipper å ha et buffer for å gjøre beregninger før den leverer ting til applikasjonen. Det er ofte tilfellet at dersom et av lagene under mister en pakke er det sannsynlig at den mister flere på rad, og Go back N vil fikse dette fort. Ulempen er at andre vi forkaster pakker som har blitt sendt på riktig måte.
- Mottakeren har et buffer på samme størrelse som vinduet sånn at den kan motta en pakke per vinduselement:
 - Hvis f.eks pakke 0 og 2 sendes riktig, men pakke 1 går tapt vil mottakeren godta pakke 0, videresende det til applikasjonen og svare med ACK. Den lagrer også pakke 2 i bufferet og svarer med ACK for denne, men den kan ikke sende det opp til applikasjonen enda siden den er i feil rekkefølge. Merk at ACK som sendes for pakke 2 sendes med sekvensnummeret til den siste pakken som kom i riktig rekkefølge (pakke 0). Når senderen mottar ACK for pakke 0 og 2 vil den fortsette å sende pakker opp til window size som mottakeren lagrer i bufferet sitt. Når timeouten for pakke 1 går ut vil den sende denne pakken på nytt og mottakeren kan da svare med en cumulative ACK for alle pakkene den har i bufferet (ACK 3) og sende pakkene i bufferet opp til applikasjonen.
 - Denne varianten kalles **Selective Repeat**. Denne fører generelt til at det er mindre trafikk og færre pakker som må retransmittes siden man bare retransmitter de pakkene som utløste en timeout fremfor å retransmitte alle pakker sendt etter en timeout ble utløst som i **Go back n**. Dette fører til bedre utnytelse av båndbredden, i bytte mot mulighet for flere timeouts hvis nettet mister flere pakker på rad. Dette fører til bedre utnytelse av båndbredden, i bytte mot mulighet for flere timeouts hvis nettet mister flere pakker på rad. **Derfor må man vite litt om de underliggende lagene for å**

velge mellom Go back N og Selective Repeat.



- Sammenheng mellom vindu størrelse og antall mulig sekvensnummer.

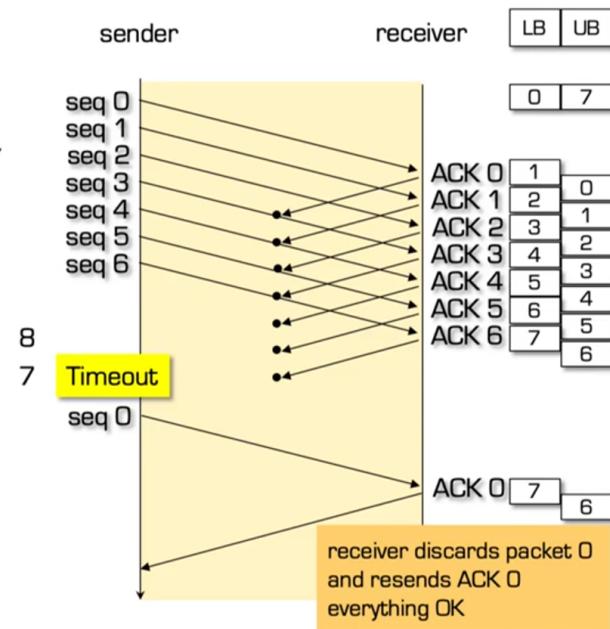
Sliding Window: Go-Back-N

Correlation between

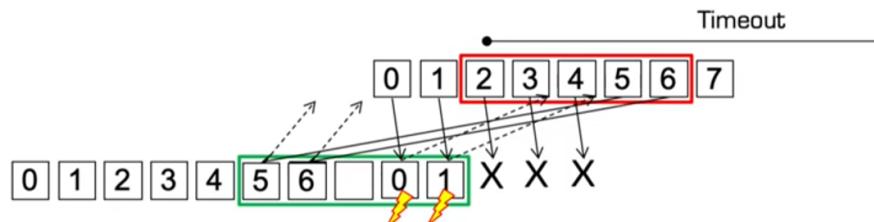
- window size and
- number of possible sequence numbers
- at least max. window size strictly less than range of sequence numbers

Example for correct window size:

- amount of sequence numbers
- window size
- all ACKs lost



Sliding Window: Selective Repeat



Correlation between

- window size and
- number of possible sequence numbers
- max. window size <= 1/2 range of sequence numbers

Example for **incorrect** window size:

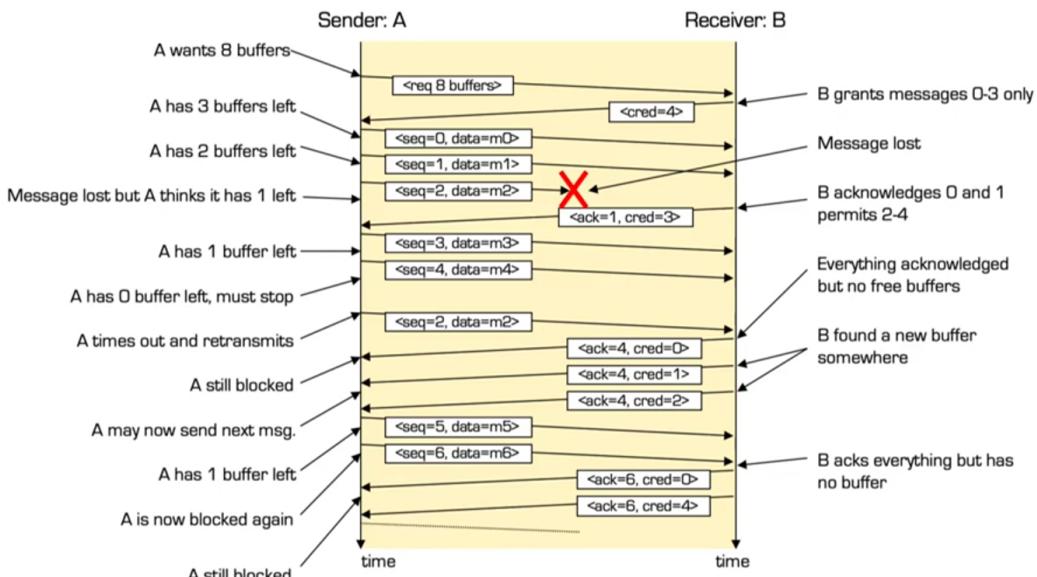
- amount of sequence numbers 8
- window size 5
- all ACKs are lost, and the packet that has been lost last is the first one to arrive at the receiver again

- Hvordan velge mellom Go back n eller Selective repeat?

- Burst errorer er ikke så vanlig i moderne nettverk - kanskje hvis nettverket er i et område med mye støy.
- Selective Repeat må bruke flere bits, og kan bare ha et vindu halvparten så stort som maksimal sekvensnummer, mens go back n kan ha så mange pakker in flight som maksimal sekvensnummer - 1. Dvs. at for å ha like mange pakker in flight må selective repeat ha flere bits i en header enn go back n, men dette er nok ikke et stort problem.

Credit mechanism:

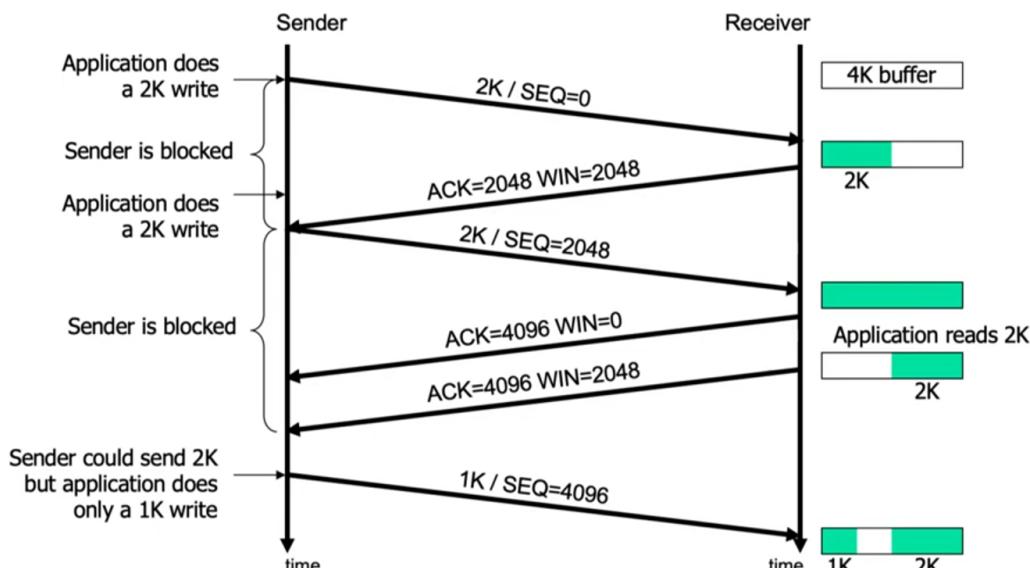
- Sender forespør et visst antall buffere sånn at den kan sende pakker etter egen evne.
- Mottaker reserverer så mange buffere som maskinen dens tillater.
- Mottaker sender tilbake ACK som inneholder hvor mange buffere den kan allokere for senderen (credit felt) i tillegg til sekvensnummeret.
- Dette lar systemene tilpasse seg dynamisk til situasjonen. Hvis mottakeren har masse ledig kapasitet kan den allokere flere buffere for sine forbindelser, og hvis den har mange forbindelser og lite plass kan den dele dette opp mellom hver forbindelse. For forbindelser med høy throughput (typ videoavspilling) kan den allokere mange buffere, mens for forbindelser med lav throughput (typ SSH) kan den allokere få buffere.
- Eksempel: Sender A forespør 8 buffere, men får bare 4 fra mottaker B.



- TCP bruker credit mechanism i sin header under Window feltet. Denne lar mottakeren fortelle senderen hvor mye plass den har på sitt receive buffer. TCP teller dette i bytes framfor pakker.

- Flytkontroll i TCP:

TCP Flow Control

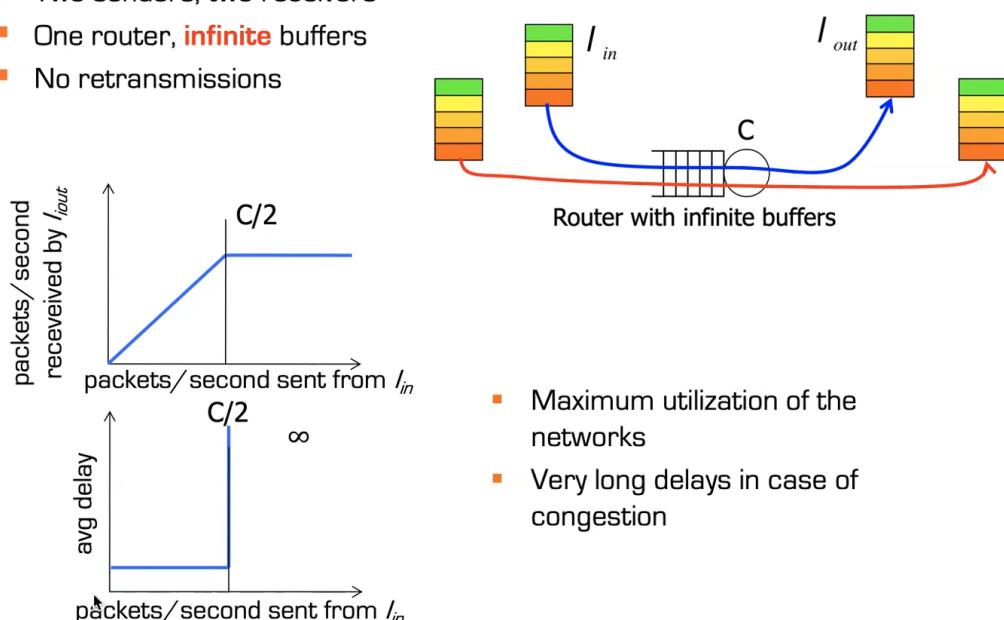


Metningskontroll:

- Metningskontroll skal finne ut hvordan man best sender data over nettet, hvilken senderate skal man bruke fra en sender til en mottaker og over alle mellomnodene og hvor flaskehalsen ligger.
- *Persistent congestion*: Router er mettet over en lengre periode, excessive traffic offered.
- *Transient (forbigående) congestion*: Korte perioder der noder i nettet er overlastet. Ofte er dette pga. *burstiness* - traffik sendes mye på en gang og lite andre perioder.
- Ofte en ruter som er problematisk (flaskehals).
- En router har gjerne et visst antall inngående forbindelser og utgående forbindelser i tillegg til noe minne. Den deler da minnet sitt over hver utgående forbindelse.

Reasons for congestion

- Two senders, two receivers
- One router, **infinite** buffers
- No retransmissions

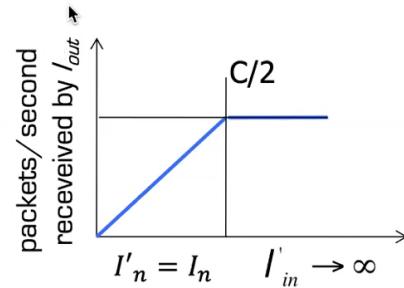
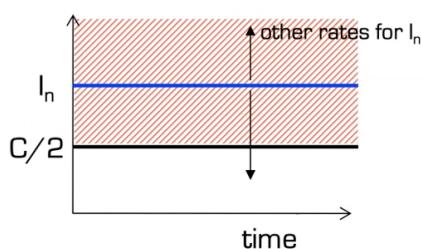


- Maximum utilization of the networks
- Very long delays in case of congestion

- Med endelige køer (buffere) vil vi tape pakker etter køen fylles. Latensen er konstant fram til køen fylles, når køen fylles vil latensen øke til et maksimalpunkt og forbli der fram til køen tømmes. Men vi ønsker ikke at routerne alltid skal ha fulle køer siden det vil føre til mye latens.
- Noen data MÅ retransmittes og vi kan ikke godta pakketap for disse (f.eks filnedlastning og operativsystemoppdatering). En løsning for dette er å bare sende pakker flere ganger for å øke sannsynligheten for at minst en av dem kommer fram på riktig måte. Ellers må vi oppdage når pakker går tapt og retransmitte dem.

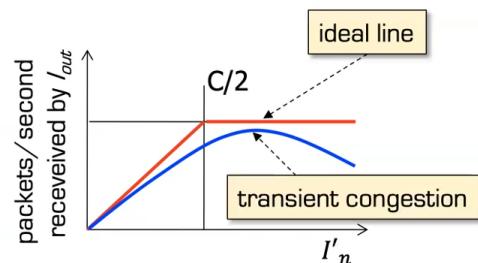
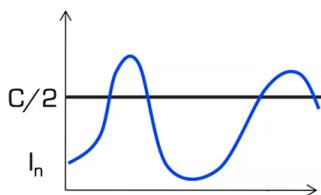
- Retransmission means that I_{out} receives all data that I_{in} sends
- When there is loss, retransmission is happening, and $I'_{in} > I_{out}$
- Retransmission of delayed (but not lost) packets increases I'_{in} above the perfect value, without increasing I_{out}
- “Cost of congestion”:
 - More work (retransmissions) for a desired throughput
 - Useless retransmissions, some links transmit several copies of the same packet

Constant bitrate (non-bursty) traffic



- Retransmission means that I_{out} receives all data that I_{in} sends
- When there is loss, retransmission is happening, and $I'_{in} > I_{out}$
- Retransmission of delayed (but not lost) packets increases I'_{in} above the perfect value, without increasing I_{out}
- “Cost of congestion”:
 - More work (retransmissions) for a desired throughput
 - Useless retransmissions, some links transmit several copies of the same packet

Bursty traffic



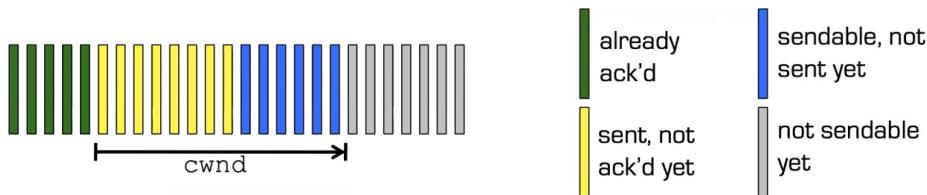
- Historisk: første idéer om metningskontroll antok at hver gang en pakke gikk tapt var dette på grunn av metning - kanskje ikke veldig god antagelse i dag.
- TCP metningskontroll: lite trafikk - øk sendingsrate, mye traffik - senk sendingsrate. For å gjette seg fram til om vi har mye traffik - sjekk om vi taper mange pakker.
 - Additive-increase, multiplicative-decrease (AIMD). Dette går ut på at så lenge det ikke er problemer med overføring av data øk antall pakker sendt om gangen med 1, dette gjøres fram til en pakke ikke kommer fram - da halverer vi antall pakker sendt om gangen.
 - Slow start. For at det ikke skal ta for lang tid å sende pakker over nettverk med høy kapasitet (1 pakke, 2 pakker, 3 pakker...) kan vi doble antall pakker sendt hver gang i starten (1 pakke, 2 pakker, 4 pakker, 8 pakker...). Denne doblingen gjøres fram til vi når et ssthresh(hold), etter det vil den gå tilbake til AIMD. ssthresh har historisk vært 65kb men denne verdien avhenger av algoritmen brukt.
 - Reaction to timeout events - for å skjønne om ting går galt.

- Algoritmer brukt i dag: TCP New Reno, TCP Cubic, TCP PRR, TCP BBR, TCP Prague

TCP Congestion Control

Basic terms:

- congestion window (cwnd)**
 - largest amount of data [of a connection] that can be in the network at a time
- maximum segment size [MSS]**
 - largest number of bytes that a TCP entity sends at once
 - always* in IPv6 and *usually* in IPv4: largest payload that fits into an IP packet behind the TCP header
 - TCP header size is minimal 20 bytes
 - with options maximal 60 bytes
- End-to-end control [no support from the network layer]
- Send rate is limited by the size of a congestion window, cwnd [counted in bytes]

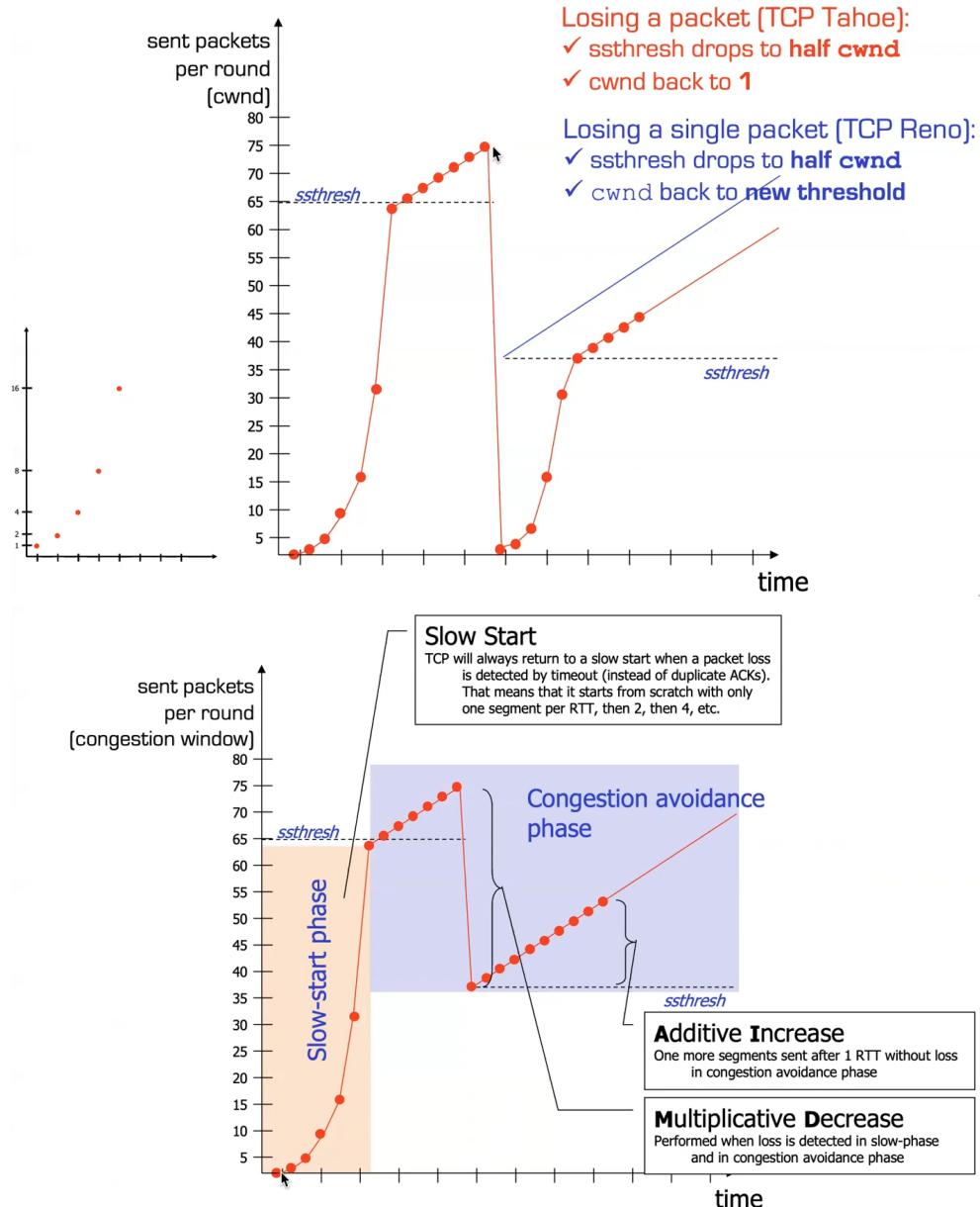


- cwnd bytes can be sent in each RTT:

$$\text{sending rate} = \frac{\text{cwnd}}{\text{RTT}}$$

- very often the permitted bytes are sent in $\frac{\text{cwnd}}{\text{MSS}}$ packets

Historically, initial ssthresh was 65 K, now in Linux it starts with ∞



- Fairness: TCP ønsker at når N TCP strømmer deler en flaskehals skal hver TCP strøm motta en N'te-del av flaskehalsens båndbredde. Mer realistisk: Når N TCP strømmer med samme RTT og loss rate deler en flaskehals og de er uendelig lange skal hver TCP strøm motta en N'te-del av flaskehalsens båndbredde.

User Datagram protocol (UDP):

- Upålitelig, forbindelsesløs, meldingsorientert

UDP is mostly IP with short transport header

- De-/multiplexing
- Source and destination port
- Ports allow for dispatching of messages to receiver process

- Checksum er ikke nødvendig i IPv4, men fullstendig fjernet i IPv6. Derfor gjøres checksumming nå i UDP.
 - I TCP/IP verden: for å kunne addressere en mottaker trenger man source og destination addressene siden man må finne mottakerprosessen på maskinen.

Derfor må UDP checksummen sjekke ting utover transportpakken og sjekke noen felt fra IP. Dette bryter lagdelingsprinsippet, men er nødvendig.

- UDP checksumming går også utover å sjekke felt fra UDP delen av pakken. Den sjekker også noen felt fra IP for å forsikre seg

UDP checksum includes

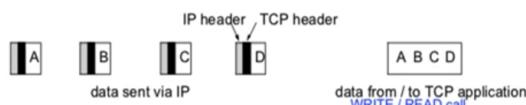
1. UDP header [checksum field initially set to 0]
2. Data
3. Pseudoheader
 - Part of IP header
 - source IP address
 - destination IP address
 - Protocol
 - length of [UDP] data
 - Allows to detect misdelivered UDP messages

Source address		
Destination Address		
00000000	Protocol=17	UDP segment length

- Source port er ikke nødvendig i UDP. Dette går fint så lenge vi ikke forventer noe svar.
- UDP brukes for:
 - Enkle client-server interaksjoner som f.eks. klient sender én pakke til server og forventer én pakke respons.
 - Kommunikasjon der lav latens er viktigere enn packet loss og duplication(?): video konferanser, gaming, IP telefon
 - Domain Name Service, Network Time Protocol, Simple Network Management Protocol, Bootstrap Protocol (BOOTP, broadcasting over nettet), Trivial File Transfer Protocol, Network File System, Real-time Transport Protocol.

Transmission Control Protocol (TCP):

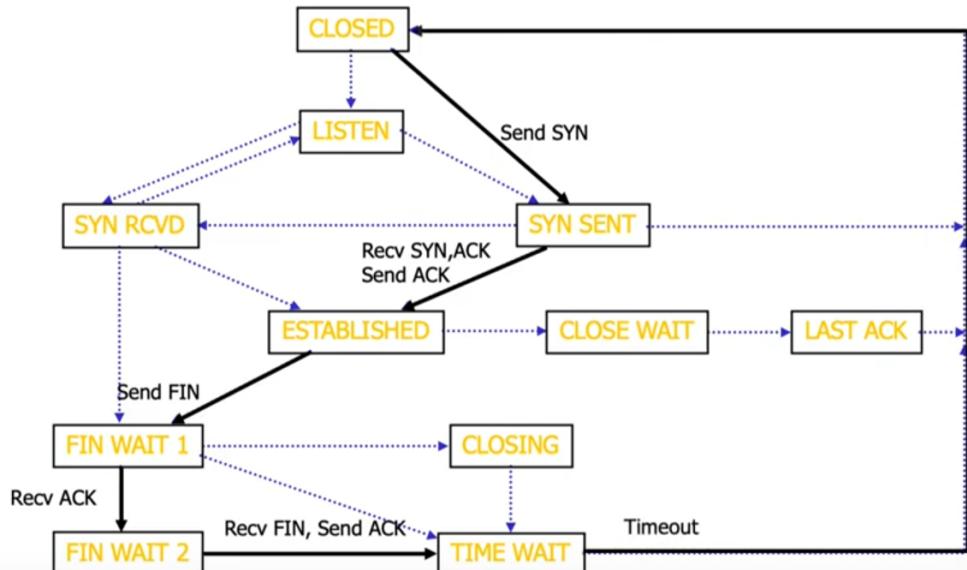
- Pålitelig
- Ende til ende: leverer til endesystemer, multiplexing og demultiplexing for å komme fram til rett applikasjon. Demultiplexing oppnår man ved å se på destination port feltet i TCP headeren.
- Byte stream: (ustrukturert) datastrøm-orientert. Går ut fra at applikasjonen ikke bryr seg om hvordan dataen er delt opp. TCP støtter ikke gruppering av data - dette er opp til applikasjonen for håndtering, men dataen kommer fram i riktig rekkefølge.
- Over en upålitelig nettverk tjeneste.
 - **Buffered data transmission**
 - Byte stream not message stream: message boundaries are not preserved
 - no way for receiver to detect the unit(s) in which data were written



- For transmission the sequential data stream is
 - Divided into segments
 - Delayed if necessary [to collect data]
- Tilbyr feilhåndtering: duplikater, feil rekkefølge, feil pakker
- Forbindelsesorientert: Virtuell forbindelse ettersom nettverklaget ikke har evne til å levere forbindelser til TCP. TCP bryr seg ikke om hvordan routingen mellom endesystemer foregår. TCP forbindelser står bare på endesystemene. Dette krever også en etableringsfase der tilstanden på hvert endepunkt (**OBS: ikke nødvendigvis endesystem siden det kan være på samme system**) opprettes. Etter en forbindelse er opprettet kan begge sider sende meldinger - håndtering av dette er opp til applikasjonen.
- TCP tilstandsmaskin: Brukes på klientsiden (for å holde styr på opprettelse av forbindelsen) og serversiden (for å holde styr på venting og opprettelse av

forbindelsen). Tilstandsmaskinen starter i CLOSED. Overføring er kun tillatt hvis begge endepunkter er i tilstanden ESTABLISHED.

- Opprettelse for klient: Oppretter forbindelse med `socket()` - tilstand CLOSED. Først sender den gjennom `connect()` en SYN pakke uten data for å synkronisere maskinene mellom endepunktene - tilstand SYN SENT. Når klien ten mottar en pakke fra serveren markert som SYN og ACK vet den serveren er klar, da sender klien ten ACK til serveren - tilstand ESTABLISHED. Hvis den ikke mottar noe svar fra serveren vil `connect()` returnere en feil til applikasjonen - tilstand CLOSED. Når klien ten er ferdig med forbindelsen kaller applikasjonen `close()` og sender den en FIN pakke til serveren - tilstand FIN WAIT 1. Når serveren mottar FIN pakken fra klien ten svarer den med en ACK - tilstand FIN WAIT 2. Når klien ten mottar ACK fra serveren sender den tilbake en ACK - tilstand TIME WAIT. For å forsikre seg om at ingen pakker sendes gjennom denne gamle forbindelsen venter vi en viss periode (1-3 min) før vi lukker forbindelsen - tilstand CLOSED. TCP client
- Opprettelse for server: Starter med å gå fra CLOSED til LISTEN (funksjonskall `socket()`, `bind()`, `listen()`) der den venter på SYN pakker fra klien ter. Når serveren mottar en SYN pakke oppretter man tilstanden for en forbindelse og svare med pakke markert som SYN og ACK til klien - tilstand SYN RCVD. Hvis serveren mottar en ACK fra klien ten går den inn i tilstanden ESTABLISHED. Hvis klien ten avslutter forbindelsen mottar serveren en FIN pakke som serveren svarer med en ACK - tilstand CLOSE WAIT. Så kan serveren sende tilbake en FIN pakke - tilstand LAST ACK før den avslutter forbindelsen med å gå direkte tilbake til CLOSED.



- For å identifisere forbindelser ser man på de 4 feltene til en socket (source addr., dest. addr., source port, dest. port.).
- Ulemper sammenlignet med UDP: Bruker flere ressurser både for klien og server. Trenger buffering både for sender og mottaker for å kunne gjenoverføre data med feil. Trenger timer for å bestemme når den må gjenoverføre pakker (hvis de har gått tapt), for å kunne etablere en forbindelse (markere feil ved oppsett av forbindelse) og for å identifisere feil underveis i en forbindelse. Krever tid for connect og disconnect (pakker må sendes over nettet 3 ganger).
- Brukes for: Web (HTTP), filoverføring (SCP, FTP), Interative terminal (SSH, telnet), E-mail (STMP, IMAP).

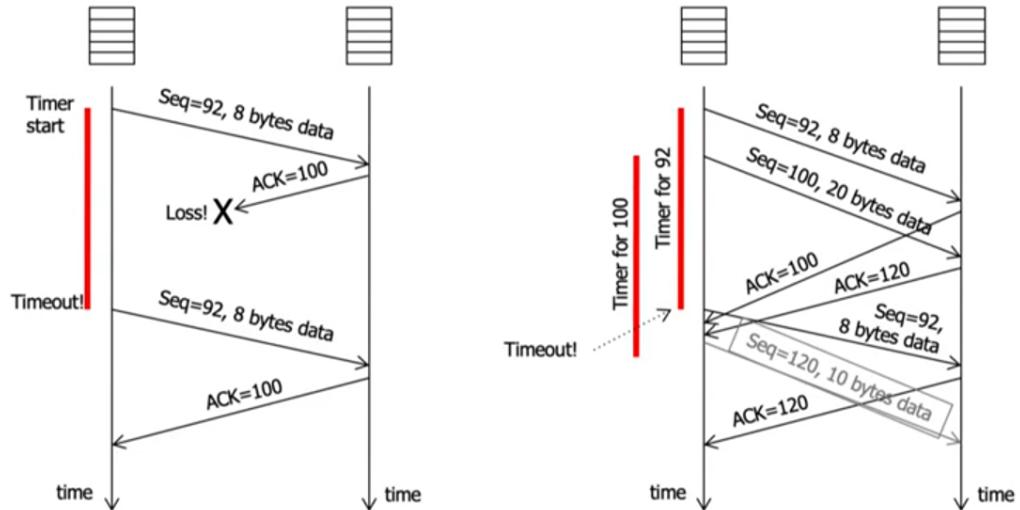
TCP begreper:

- Båndbredde: Klassisk definisjon: antall signaler per sekund målt i hertz. Ny definisjon (informasjonsrate, brukt i lag 3-5): antall bits overført korrekt per sekund.
 - **Nominal båndbredde:** Maksimal båndbredde mulig å overføre i bits/sek. på en link.
 - **Tilgjengelig båndbredde:** Båndbredden en sender klarer å sende gjennom et nett, når det allerede er traffik på nettet.
 - **Bottleneck båndbredde (viktig i metningskontroll):** Minste nominale båndbredde over en sti fra en sender gjennom et nett (den tregeste mellomnoden på vei mellom to endesystemer).
 - **Throughput:** Mengden bits som kommer fram til mottakeren / båndbredden sett av mottakeren.
 - **Goodput:** Båndbredden den mottakende applikasjonen opplever. Alle meningsfulle bytes transportlaget klarer å videresende opp til applikasjonen. Kan være lik throughput hvis det ikke er noen problemer under overføring av data.
- Delay:
 - **Propagation delay:** Tid for å overføre en bit mellom to endesystem
 - **Latens:** Propagation delay + msg_length / bottleneck bandwitch + queuing delay. Dette er kun et estimat av latensen, f.eks. kan prosesseringens delay også være relevant.
 - **Jitter:** Endringer i delay. Kritisk for real-time applikasjoner ettersom vi ikke vil at f.eks. når noen snakker kommer det siste ordet i en setning fram før det første.
 - **End-to-end delay:** Tiden det tar for en melding å sendes fra et endepunkt til det mottas av et annet endepunkt. 2 typer ende til ende delay: for applikasjonslaget går all prosessering av data på lagene under applikasjonslaget under ende til ende delay. For transportlaget refererer dette til tiden det tar for en pakke å komme fra en transportentitet til en annen, kun pakkene som kommer fram, ikke pakkene som må gjenoverføres.
 - **Round-trip time (RTT):** Tiden for å overføre en bit fram og tilbake mellom to endesystem.

TCP pålitelighet:

- Må detektere om bytes som mottas allerede finnes hos mottakeren, i så fall forkast dem. Det er tre alternativer for mottakelse av data. Enten kan det være data man allerede har lest, data som er ny og kan leveres rett til applikasjonen, eller om det er data som allerede er kommet fram men som manglet noe vi nå har mottatt. For å finne ut av dette bruker vi et felt som heter `piggyback acknowledgement`. Denne inneholder bytenummeret fra pakkestarten som er den neste mottakeren forventer. Hvis mottakeren tar imot en forventet pakke med data som kommer i riktig rekkefølge vil piggyback acknowledgement inneholde bytenummeret til den byten etter den siste byten den har mottatt. Hvis mottakeren mottar en pakke som inneholder data med et sekvensnummer som ligger etter et hull av data som ikke har kommet frem vil TCP sende tilbake en ACK til senderen markert med et flagg i headeren. Pakken TCP sender til senderen inneholder byten(sevensnummeret?) til den første byten den mangler. Siden alle disse pakkene kan gå tapt startes det en timer hver gang en pakke sendes. Hvis senderen av pakken ikke mottar noe ACK svar for pakken de sendte kan de anta at pakken ikke har blitt lest og må i så fall sende pakken på nytt. Grunnen til at senderen av en pakke ikke mottar en ACK kan også være at ACK pakken gikk tapt, for dette må vi håndtere muligheter for gjentatt lesing av samme pakke dersom noen sender en pakke 2 (eller flere) ganger siden de aldri

leste noen ACK.



Timeout retransmission

In modern TCP implementations,
a lot is done to suppress spurious retransmissions.

The reason is that timeouts have another meaning: congestion

- Hvordan beregne retransmission timer? Må være lengre enn RTT. Hvis den er for kort vil det forekomme mange spurious retransmissions. Hvis den er for lang vil senderen reagere tregt og flytkontroll bufferet fylles opp. Løsning: estimated RTT.

Estimated RTT

- Use a SampleRTT: Measure time from segment sending and ACK received
- Compute EstimatedRTT as a floating average
- Increase the EstimatedRTT with a safety margin that is big when SampleRTT changes a lot
 - (because fewer timeouts are considered better)

$$x = 0.1$$

$$\text{Deviation} = (1 - x) * \text{Deviation} + x * |\text{SampleRTT} - \text{EstimatedRTT}|$$

$$\text{EstimatedRTT} = (1 - x) * \text{EstimatedRTT} + x * \text{SampleRTT}$$

$$\text{Timeout} = 3 * \text{EstimatedRTT} + 4 * \text{Deviation}$$

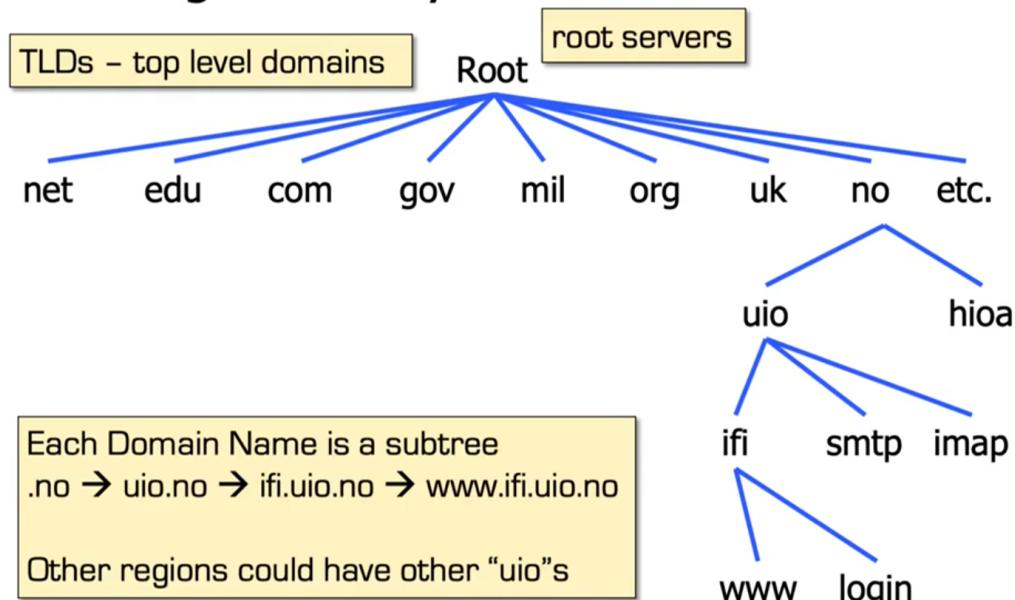
Spurious retransmission

Addressering i applikasjonslaget (DNS):

- For å koble seg på en ekstern maskin: SSH/telnet(gammel). Denne bruker IP addresser med mindre noe annet er spesifisert i `/etc/hosts` (eller DNS server forbinder andre navn med IP addresser). I denne filen kan du assosiere IP addresser med mer lesebare navn. For å forvalte slike assosiasjoner på en større skala bruker vi DNS (Domain Name System).
- DNS benytter et hierarkisk namespace, f.eks. `.com` → `google.com` → `mail.google.com`.
- Vi kan se på DNS som en distribuert database siden det ikke er én sentral server som tar var på disse forbindelsene.
- DNS bruker både UDP og TCP på port 53 der den lytter etter alle innkommende IP addresser.
- DNS servere må bruke TCP for å forsikre at oppdateringer kommer fra kilder vi stoler på.

- DNS klienter får ikke bruke TCP. Dette er for å redusere server load, men er et sikkerhetsproblem ettersom det er lett å lure klienter til å ta i mot misvisende UDP pakker der man kan utføre man in the middle angrep.
- Naming hierarki: 13 root servers i verden. Root servere har ansvar for å oppsløse oppslag til toppleveldomenene (.com, .gov, .no). Andre organisasjoner må så prøve å legge seg under et passende top level domene (f.eks. bedrifter under .com, universiteter under .edu, organisasjoner under .org, osv). Noder i dette treet har en tendens til å være veldig brede og grunne. Dette er fordi navn burde være korte og lette å huske. Internet council for assigned numbers and names (ICANN) har ansvar for root domenet og fordeler ansvar for underdomener som .no, .edu, .com. UNINETT har ansvar for .no. UIO har ansvar for .uio. UIO har også ansvar for .ifi. Organisasjoner kan velge om de vil fordele forvaltningen av domener eller beholde det sentralt.

Naming Hierarchy



- For forvaltere av et subtre har de ansvar for å besvare alle oppslag for navn under det subtre, også om navnene eksisterer eller ikke. Det betyr ikke at alle DNS servere inneholder alle navn under det subtre. Men de må vite hvem de kan videresende forespørselen til for å slå opp alle navnene. F.eks. har .no ansvar for å slå opp alle navn under .no, men ikke å lagre alle disse navnene. Alle forvaltere må også ha minst 2 servere i tilfelle krasj. Alle DNS servere må også kunne addressene til root servere. Root servere kjenner alle top level domener og alle serverne som betjener disse domenene.
- ICANN administrerer de 13 root serverne. 6 av disse er *anycasted*.

Oppslag i DNS:

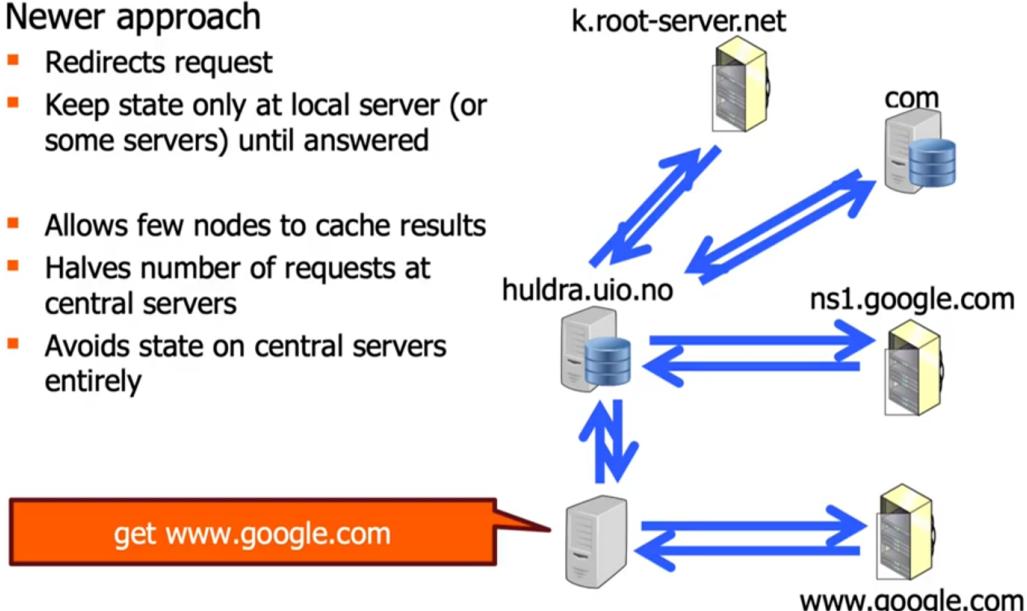
- Rekursiv DNS query (gammel): Hver server på vei fra forespørrende klient til serveren som har svaret på queryen beholder forbindelsen/tilstanden opp mens oppslaget prosesseres. Dette tillater serverne å cache svaret i det det returneres tilbake gjennom kjeden av servere. Men dette er en ganske dyr fremgangsmetode siden det bruker minne å opprettholde denne tilstanden.
- Iterated DNS query (ny): Omdirigerer oppslag tilbake til lokal DNS server for å unngå lagring av tilstand mens oppslaget prosesseres på andre servere. Da kan vi bare cache resultatet i den lokale serveren og ikke de andre som omdirigerte

oppslaget.

Iterated DNS Query

Newer approach

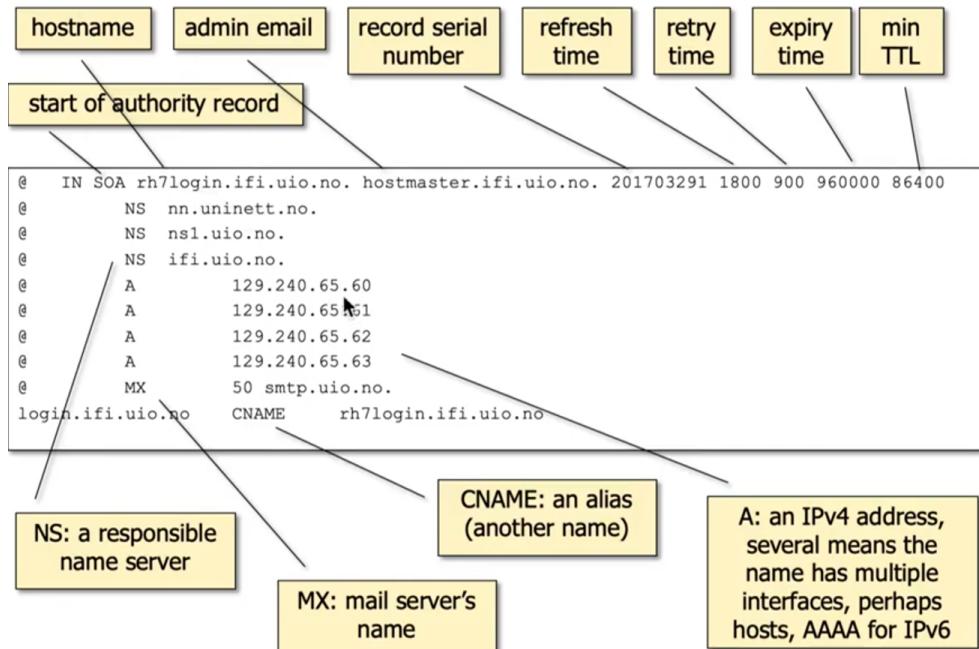
- Redirects request
- Keep state only at local server (or some servers) until answered
- Allows few nodes to cache results
- Halves number of requests at central servers
- Avoids state on central servers entirely



- Konflikt - Caching vs. Freshness: Når en server oppdaterer sin informasjon, en annen server har cachet den gamle informasjonen og en klient søker opp nettsiden og får informasjonen fra serveren med cachet informasjon vil denne cachete informasjonen være utdatert. Løsning: Når en server cacher informasjon lagres dette kun en viss periode (mellan 5 min og 72t).
- Maskiner kan ha flere aliser (flere navn for en samme IP adresse), og et navn kan mappes til flere maskiner (et navn til flere IP addreser). Dette kan for eksempel brukes i *Content Delivery Networks (CDN)*. Siden DNS tillater *zoning* (servere gir forskjellig svar avhengig av vår lokasjon) kan vi basert på dette bestemme hvilken server vi skal bruke for et gitt navn. Et problem med dette er at DNS oppslag ikke alltid kan brukes for å identifisere forespørrelens faktiske lokasjon pga. external resolvers. F.eks. vil alle chrome DNS oppslag komme fra addressen 8.8.8.8 som er eid av google og dersom man prøver å redirigere til en server nærmere denne addressen vil det sannsynligvis ikke gi best resultat. På den andre siden kan dette være en fordel siden det gjør oppslag mer anonyme. Dette problemet kan ikke løses med DNS, men bedrifter kan gjøre private avtaler for å omgå dette problemet.
- Siden vi kun cacher informasjon om nettsider i en viss periode kan vi i løpet av denne perioden bestemme om det er nødvendig å bytte hvilke servere vi ønsker å benytte for fremtidige oppslag for nettsiden. For eksempel hvis det i løpet av et "cachings interval" gjøres mange oppslag til en spesifikk server kan vi ved neste gang den lokale DNS serveren må oppdatere cachen omdirigere til en annen server som opplever mindre belastning.
- DNS records:
 - Serial number: oppdateres hver gang noe endres.
 - Informasjon om hvor lenge en cache skal ta vare på data. 1800 - må oppdatere cachet etter 30 min. 900 - hvis et navneoppslag feiler må man vente 15 minutter før man kan prøve igjen. 960000 - etter 11 dager med gjentatte feilende forsøk på et navn skal det slettes fra recorden. 86400 - Minste tiden et navn må beholdes i cache før man har lov å spørre igjen eller

oppdateringen spres til andre web servere.

DNS Record

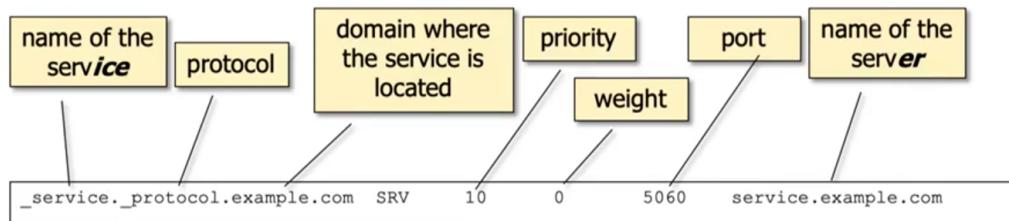


- Multicast DNS - måte for maskiner å dele tjenester over lag 2 (ethernet/wifi)

mDNS

A way of discovering services by announcing them with IP multicast

- RFC 6762 (2013): multicast DNS
- records announce services (as well as link-local hostnames that are invisible outside the current multicast domain, like mymac.local)
- records are never authoritative and mDNS can never redirect or recurse



Example from my machine:

```
_ssh._tcp.example.com SRV 10 0 22 lx-193-157-212-9.uio.no
```

Routing and switching (lag 3):

- Nettverkslagets hovedoppgave er å tilby tjenester til transportlaget (enten forbindelsesløst eller forbindelsesorientert)
- Routing: hvilken vei skal vi velge for å sende pakker? Må bestemmes kontinuerlig gjennom en pakkes reise.
- Metningskontroll: Enkel løsning brukes - hvis det er for stor trafikk på en node, bare forkast noen pakker. Ellers kan metningskontroll utføres på lag 4.

- Quality Of Service: Garantere til lag 4 at en pakke ankommer en mottaker innen en viss tid. Garantere at et visst antall pakker av en viss størrelse ankommer innen viss tid. Garantere at rekkefølgen til pakkene ankommer i samme rekkefølge som de sendes (gjøres gjerne i lag 4 i stedet).
- I IP headers:
 - Flow label: Kan brukes for å identifisere at pakken skal være forbindelsesorientert (denne brukes lite i dag). Alternativt kan man bare bruke destination address for å sende pakker forbindelsesløst. Ut ifra denne addressen kan man beregne seg fram til den beste neste noden å sende pakken til på vei mot mottakeren.
 - Mange andre felt i header brukes for å forsikre QoS krav - DSCP (Differentiate Services Code Point - for å klassifisere en traffikkklasse som kan bruke ressurser i mellomnodene reserver til den klassen (øke prioritet på pakkene)), ECN (hjelpemiddel for å gjøre det lettere å håndtere metning på lag 4)
- Hva må en mellomnode vite? Mellomnoden må kjenne nettverkstopologien for sitt nærområde - kjenne sine subnett og nabonett. Må kunne sende til et endesystem hvis de har ansvar for det. Hvis den skal støtte QoS må den kjenne til regnekraft/minne og kapasitet for overførsel til nabonoder. Må også kjenne lasten på sitt system for å kunne kommunisere med forbindelsesorienterte tjenester på lag 4 hvor rask kommunikasjon de kan forvente. Etter dette kan en mellomnode videresende pakker enten gjennom *routing* eller *switching*.
- Terminologi:
 - Routing og switching har mye til felles - man har en mellomnode som skal finne ut hvilket interface den skal bruke for å videresende pakken (så lenge det ikke skal sendes til et endesystem på det lokale nettverket).
 - Vanligvis: Switching på lag 2, routing på lag 3 - i dag noen ganger switching på lag 3, men ikke routing på lag 2.
 - I bildet under: Identifier er felt som f.eks flow label. Det er ikke alltid tydelig om det er snakk om routing eller switching.
 - In packet-based networks like the Internet, we call something ...
 - ... **routing**, when an IS reads a destination address from an arriving packet, computes which of its direct neighbors is best suited for reaching that destination, and sends the packet to the neighbor.
 - ... **switching**, when an IS reads an identifier from an arriving packet, looks it up in a pre-filled mapping table that translates the identifier to a direct neighbor, and sends the packet to the neighbor.

Circuit switching (linjesvitsjing):

- Basert på fysiske elektriske forbindelser. Forbindelser mellom switching sentre på vei mellom endesystemer. Forbindelsene mellom switching sentre kan manipuleres. Dette er ting som telefonsentre der mennesker manuelt plugget kabler mellom den som ringer til den som blir ringt - da oppstår en fysisk elektrisk kobling mellom disse to. Dette kan også automatiseres basert på input-telefonnummeret. I dag: Wavelength Division Multiplexing (WDM). Koblingspunkter der lysfrekvenser representerer en deltaker i kommunikasjon. Samme som gamle switchboards, bare mindre og optiskt.
- Egenskaper:
 - Tar lang tid å etablere en forbindelse mellom endesystemer. Dette må man også vente på før man kan sende pakker.
 - Ressurser kan gå tapt hvis deltakere ikke har noe å sende - ingen andre endesystemer kan bruke samme kabel/frekvens.
 - Ingenting kan stoppe kommunikasjon etter den er etablert (med mindre uhell med kabel eller lignende).

Packet Switching (pakkesvitshing):

- Antar at hver pakke må routes på hver mellomnode mellom hvert endesystem.
- Det er ingen forhåndsdefinert rute for kommunikasjon, når det finnes flere veier velger mellomnoden hvilken. Noen ganger kan mellomnoder velge feil vei.
- Dette brukes i internettet.
- Egenskaper:
 - Kan ikke reservere ressurser for pakker siden de sendes uavhengig av hverandre og mellomnoder kan velge forskjellige veier for dem. I stedet kan man legge til tilstander på mellomnodene som sjekker mottakeradressen til pakken og prioriterer spesifikke veier basert på addressen. Da kan dedikere båndbredde for pakker til samme adresse.
 - Mulighet for metning hvis mange pakker skal i samme retning/til samme endesystem.
 - God bruk av ressurser (potensielt 100%) når mange meldinger sendes samtidig.

Circuit Switching vs Packet Switching:

- Etablering av forbindelse kan ta lang tid med circuit switching - etterpå er båndbredde reservert for denne forbindelsen uten fare for metning. Negativt at mye av kapasiteten til nettet kan være ubrukt hvis endesystemer kommuniserer når forbindelsen er opprettet uten å ha noe å kommunisere. For pakkesvitjing slipper vi etableringsfasen og allokerer ikke båndbredde. Alle kan sende pakker til hvem de vil når de vil - dette kan føre til metning, men vi oppnår også god bruk av ressurser
- Tid for overføring av data med circuit switching er konstant siden de alltid går samme vei, med packet switching kan dette variere basert på hvilken vei pakker ender opp med å gå.

Virtual circuit switching:

- Ligner på circuit switching, men baserer seg på pakker.
- Hvordan: Etabler en sti fra sender til mottaker som skal brukes så lenge forbindelsen varer. Tilstandsinformasjon lagres på alle mellomnoder og endesystem som garanterer at nodene har tilstrekkelige ressurser for å utføre kommunikasjonen. Når en mellomnode skal videresende en pakke slipper den å kalkulere hvilken nabo den må sende pakken til. Den kan heller sjekke identifier i pakken for å finne hvilken nabo den skal sende pakken til i en tabell etablert i etableringsfasen - mye mer effektivt.
- Siden man kjenner stien,ressursene tilgjengelige og hvilke forbindelser som er etablert, kan man basert på informasjonen om pakkene tilby QoS (f.eks. definere gjennomsnittlig tid for å overføre pakker og worst case tid for å overføre pakker) og vedlikeholde rekkefølge på pakkene.
- I IPv6 kan man bruke flow label for identifikasjon. Denne er kortere og mottas før destination address i headeren.

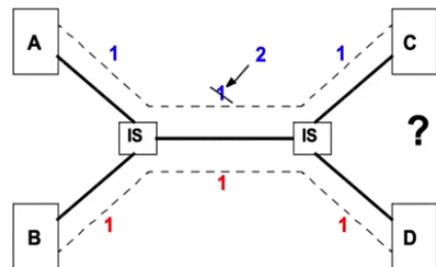
Mer Virtual Circuit Switching:

- I etableringsfasen:
 - Velg en vei
 - Bestem en VC identifier for hele denne circuiten
 - Alle IS må lagre informasjonen om veien med det gitte VC identifier
 - Nettverket må reservere ressurser nødvendige for forbindelsen

- I data transfer phase:
 - Alle pakker følger samme vei og inneholder VC identifier (ingen addresseinformasjon, denne glemmes etter veien er funnet i etableringsfasen)
 - Mellomnode bruker lagret vei for å bestemme neste node å sende til
 - Kanskje oppdatere VC identifier i pakken
 - VC identifier er ikke globalt unike. Valget av en VC identifier gjøres uavhengig på endesystemer, så når to endesystemer bruker en VC identifier betyr ikke det at dersom en mellomnode leser denne VC identifieren vet den hvilken vei å sende pakken til. Den må også ta hensyn til hvem som sender pakken. Når en mellomnode leser VC identifier fra en pakke og ser at dersom den videresender pakken til den neste naboen vil ikke naboen være i stand til å skille mellom pakker fra forskjellige sendere, må den mellomnoden omskrive VC identifieren til en av forbindelsene.

Implementation Virtual Circuit

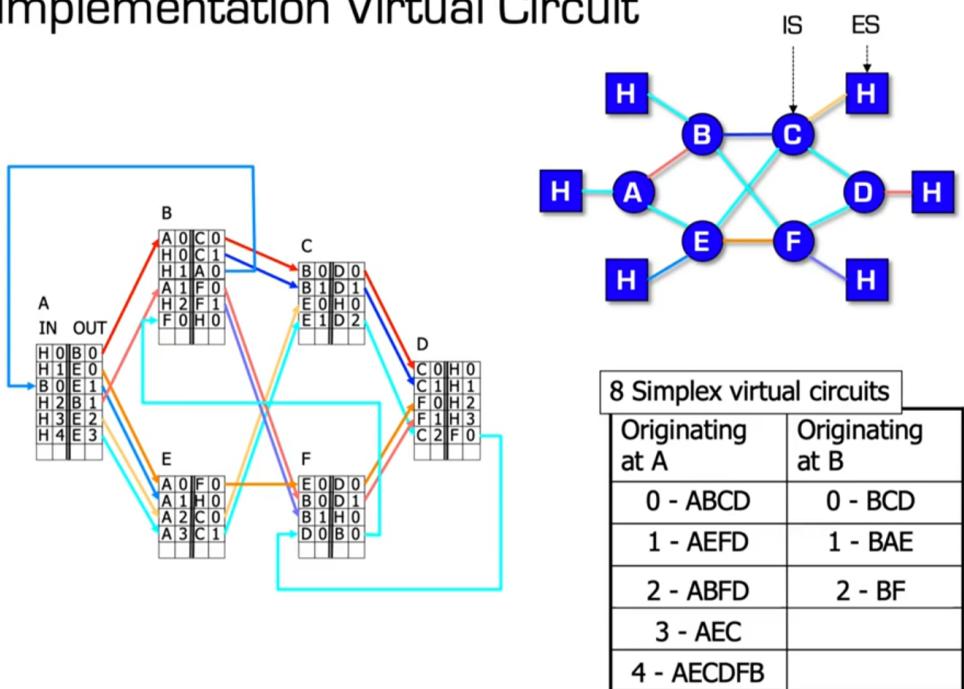
- ES allocates VC number independently [no negotiation]
- Problem: the same VC identifiers may be allocated to different paths



- Solution: allocate VC numbers for virtual circuit *segments*
 - IS differentiates between incoming and outgoing VC-number
 - IS receives incoming VC number in CONNECT message from previous node
 - IS creates outgoing VC number [unique between IS and successor (IS)]
 - IS sends outgoing VC number in CONNECT message to next node
- Disconnect phase:
 - Alle mellomnoder frigjør reserverte ressurser
 - Slett informasjon om den VC fra sine tabeller
- Implementasjon:
 - Veivalg (routing) er uavhengig av implementasjonen til VC.

- Se på IN og sjekk om noden pakken kommer fra allerede har brukt den samme VC identifieren. Hvis ja endre den

Implementation Virtual Circuit



Virtual circuit vs packet switching:

- Virtual circuits trenger fortsatt en etableringsfase som kan ta tid, dette slipper vi med packet switching.
- I PS er det lett å endre på routingen dersom noe går feil ettersom det aldri blir definert noen foretrukket vei på forhånd. Dette er ikke like lett med VC, da vil hele forbindelsen kollapse dersom en mellomnode feiler. For å løse dette kan man signalere senderen, mottakeren og alle mellomnoder på veien sånn at de kan frigjøre ressurser for den tidligere forbindelsen sånn at man kan sette opp en ny forbindelse som igjen krever vesentlig oppsett.
- I VC brukes kortere addresser for å finne neste nabo å sende pakke til - lavt overhead gjennom data transfer phase. Sammenlignet med PS som bruker lengre addresser for å addressere endesystemet direkte, som i tillegg gjør at pakker er større.
- Virtual circuit switching kan reservere båndbredde som lar lag 3 kommunisere til lag 4 en sannsynlighet for packet loss og hvor lang tid den kan forvente å måtte vente for å motta pakker (tilbyr QoS), dette kan vi ikke gjøre med packet switching.
- Når vi reserverer båndbredde med virtual circuit switching kan dette begrense kommunikasjonen mellom endesystemer. Dette lar oss garantere stabil kommunikasjon, i bytte mot lavere utilisering av båndbredde sammenlignet med packet switching.
- Hverken av løsningene tilbyr konstant tid for overføring av data. For VC varierer denne tiden fortsatt mellom den gjennomsnittlige-, maksimale-, og litt under gjennomsnittlige- ventetiden.
- Med VC går alle pakker samme vei, mens i PS kan pakker mellom to like endesystemer gå forskjellige veier.
- VC krever mer minne i mellomnoder for å ta vare på tabeller for hver forbindelse. Her må den lagre både den naboen pakken kommer fra, VC identifieren den har brukt og hvilken neste nabo å videresende pakken til + muligens en oppdatert VC identifier + informasjon om hvilke ressurser som er allokkert for den pakken.

Message switching:

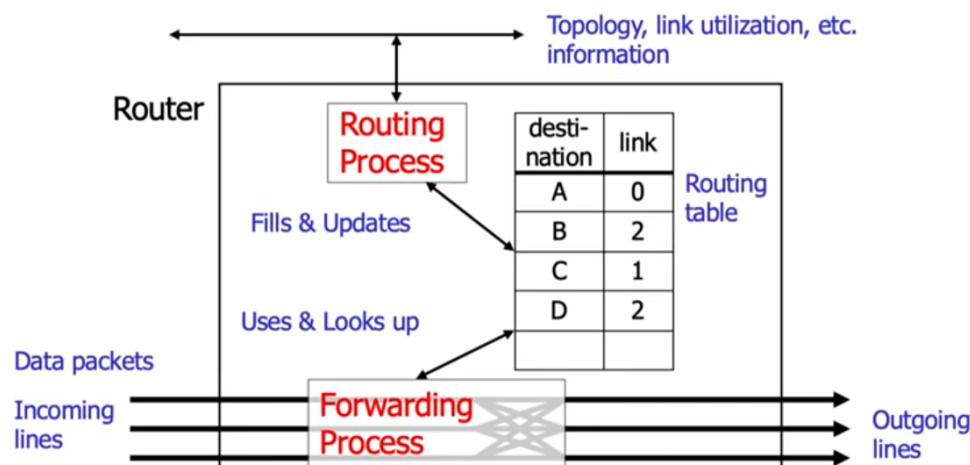
- Brukes sjeldent på lag 3, noen ganger lag 5.
- Hvordan: Antar at all data er delt opp i meldinger. Melding sendes fra endesystem til mellomnode, der mellomnoden samler lag 2 frames fram til hele meldingen fra sender til mellomnoden er overført. Når mellomnoden har mottatt hele meldingen videresender den dette til neste mellomnode med samme prosess. Dette innebærer at meldinger mellomlagres i mellomnoder. Dette lar oss korrigere feil før vi videresender data til neste mellomnode.
- Denne metoden kan ta veldig lang tid
- I dag brukes dette blant annet for NASA deep space network. Mars rover videresender lagrer data lokalt frem til den er innenfor rekkevidden til en satellitt som går rundt Mars. Når den kommer innen rekkevidde overfører den data mha. message switching for å forsikre sikker overførsel av data. Til slutt videresender satellitten dataen til jorda når mulig.
- Egenskaper:
 - Voldsomt minnebruk på mellomnoder (kanskje til og med utover primærminnet)
 - Veldig tregt
 - Garanterer at meldinger sendes riktig
 - Noder kan brukes til sine fulle kapasiteter over lengre perioder

Routing:

- Routingalgoritmer jobber med grafer.
- IPv6: for forbindelsesorienterte tjenester - route label i IP header, for forbindelsesløse tjenester - endesystemets adresse i destination address.
- I virtual circuits må man bare route en gang (evt på nytt når feil oppstår), mens med pakkesvitsjing gjøres routing for hver pakke.
- Routingalgoritmen må definere hvilken utgående linje en innkommende pakke skal sendes gjennom, eventuelt om pakken skal videresendes til den lokale maskinens lag 4.
- Routing og forwarding: vi skiller mellom prosessen om å velge hvilken linje vi skal sende pakker gjennom (routing) og prosessen om å videresende pakker (forwarding).

■ Distinction can be made

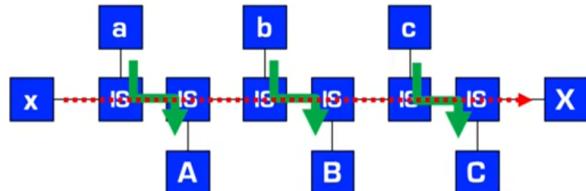
- Routing: makes decision which route to use
- Forwarding: what happens when a packet arrives



- Egenskaper til gode routingalgoritmer: Simplicity - minimer last på ISer, Robustness - Compensere for IS og link feil og håndtere endring i topologi og traffik, Staiblity - consistent resultater (pakker skal ta omtrent samme tid å

overføre mellom to endesystemer), Fairness - noen pakker skal ikke bli sendt over høyt belastede stier mens andre blir sendt over lavt belastede stier, Optimality - nettet skal utnyttes perfekt.

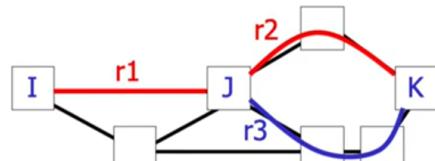
- Konflikter: Fairness vs optimization.
 - Often conflicting: fairness and optimization
 - Some different optimization criteria
 - average packet delay
 - total throughput
 - individual delay
 - conflict
 - Example:
 - communication among $a \rightarrow A$, $b \rightarrow B$, $c \rightarrow C$ uses full capacity of horizontal line
 - optimized throughput, but
 - no fairness for $x \rightarrow X$ – Tradeoff between fairness and optimization
 - Therefore often
 - hop minimization per packet
 - it tends to reduce delays and decreases required bandwidth
 - also tends to increase throughput
- Klassifisering av routingalgoritmer: Ikke adaptive algoritmer og adaptive algoritmer.
- Ikke adaptive algoritmer: Aktuell tilstand av nettet tas ikke hensyn til i algoritmen. Dvs. last på rutenettet, avstand mellom noder, antall hopp.



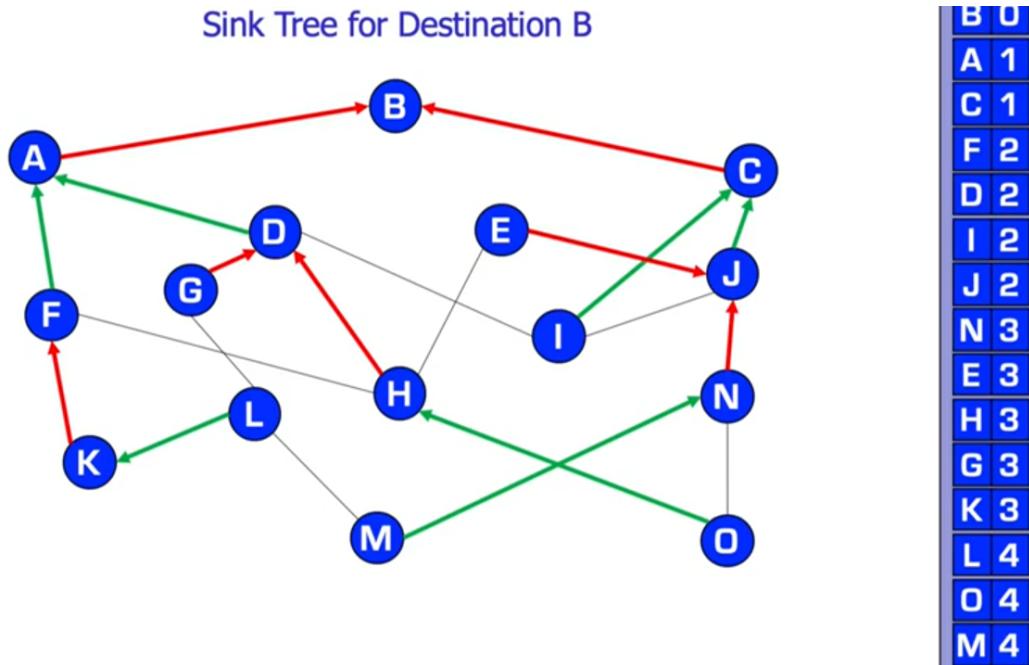
- Class: Non-adaptive Algorithms
 - Current network state not taken into consideration
 - Assume average values
 - All routes are defined off-line before the network is put into operation
 - No change during operation [static routing]
 - With knowledge of the overall topology
 - Spanning tree
 - Flow-based routing
 - Without knowledge of the overall topology
 - Flooding

- Class: Adaptive Algorithms
 - Decisions are based on current network state
 - Measurements / estimates of the topology and the traffic volume
 - Further sub-classification into
 - Centralized algorithms
 - Isolated algorithms
 - Distributed algorithms
 - Starting idea: using a route has a *cost*
 - number of hops, delay, ...
 - General statement about optimal routes
 - if router J is on the optimal path from router I to router K
 - then the optimal path from router J to router K uses the same route
 - Idea of the proof
 - best route from I to K is like this:
 - r1: from I to J, then
 - r2: from J to K
 - then r2 is also the best route from J to K
 - if better route r3 from J to K would exist
 - then concatenation of r1 and r3 would improve route from I to K
 - Set of optimal routes
 - from all sources
 - to a given destination

form a tree rooted at the destination: **Sink Tree**



- Sink tree bruker minimalitetsprinsipp for å definere korteste vei fra en node til rotten.



- Sink trær har ingen løkker. Ikke nødvendigvis unikt - det finnes flere sink trær som løser det samme problemet.

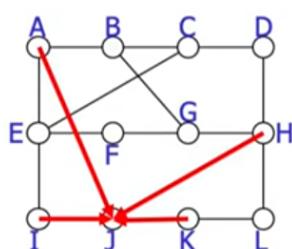
Link state routing:

- IS-IS (intermediate system)
- Open Shortest Path First (OSPF)
- Prinsipp: hvert IS måler avstanden til sine nabover, så sprer den sine målinger over nettet og beregner den ideelle routen fra seg selv til alle andre.
- Fremgangsmetode: distribute med flooding.
 1. Determine the address of adjacent IS
 2. Measure the "distance" [delay, ...] to neighbouring IS
 3. Organize the local link state information in a packet
 4. Distribute the information to all IS
 5. Calculate the route based on the information of all IS
- 1 fase: hent informasjon om nabonoder og lag graf basert på dette. Hvis en ny node kobles til nettet sender den en HELLO melding over alle sine lag 2 kanaler, den nye nodens nabonoder må så svare med sin egne addresse.
- 2 fase: måle avstand i latens mellom alle nabonoder. Sjekk hvor lang tid det tar å sende en pakke. Da må vi også måle queueing delay og vi må bestemme om vi vil måle forsinkelsen dersom pakken lå på starten av køen til ISet eller på slutten.
- 3 fase: organisere informasjonen som en link state pakke. Link state pakker består av: Sekvensnummer - brukes for å forsikre seg at pakkene ikke går i løkke når det floodes gjennom nettet (hvis en node leser et sekvensnummer den allerede har sett, forkaster den den kopien og sender den første pakken). Age - settes opp for å kunne slette gammel informasjon om noder (som kanskje ikke lenger fins). De andre feltene er de nyeste målingene av distanser til andre nabonoder. Her trenger det ikke være symmetriske veier fra og til en node, f.eks. kan den beste veien fra A til C være gjennom B, mens den beste veien fra C til A kan være gjennom D. Dette går fint, men vil kanskje endres ved nye state updates.
- 4 fase: distribuer den lokale informasjonen til alle ISer.

- 5 fase: compute nye router. Konstruer graf over nettverk og beregn Dijkstras shortest path for å finne korteste vei mellom to noder.

Distance Vector routing:

- Unngår kalkulering av dijkstra, gjør kun tabelloppsslag.
- Hvert IS har en tabell (vektor) som inneholder den beste kjente avstanden til alle andre noder i nettet og hvilken linje som må brukes for å komme seg dit. Disse tabellene oppdateres ved å utveksle informasjon mellom sine direkte naboyer (ingen flooding) som gjør dette mer skalerbart. Ingen noder trenger å kjenne hele topologien til nettet, men må kjenne sine naboyer.
 - Each IS
 - maintains routing table with one entry per router in the subnet
 - is assumed to know the distances to each neighbor
 - sends list with estimated distances to each destination *periodically* to its neighbors
 - X receives list $E[Z]$ from neighbor Y
 - Distance X to Y: e
 - Distance Y to Z: $E[Z]$
 - Distance X to Z via Y: $E[Z] + e$
 - IS computes new routing table from the received lists containing
 - Destination IS
 - Preferred outgoing path
 - Distance

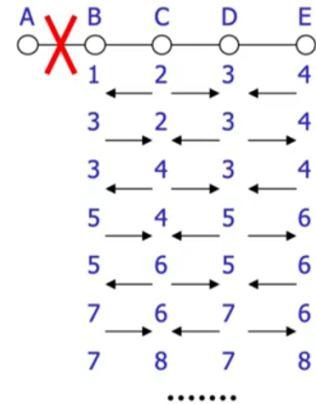


	A	I	H	K	line
A	0	24	20	21	8 A
B	12	36	31	28	20 A
C	25	18	19	36	28 I
D	40	27	8	24	20 H
E	14	7	30	22	17 I
F	23	20	19	40	30 I
G	18	31	6	31	18 H
H	17	20	0	19	12 H
I	21	0	14	22	10 I
J	9	11	7	10	0 -
K	24	22	22	0	6 K
L	29	33	9	9	15 K

delay JA JI JH JK
8 10 12 6

- Previous routing table will not be taken into account
 - Reaction to deteriorations

- Distance vector routing er dårlig på å oppdage feil.
 - Slow distribution of information about new long paths [with many hops]
 - “Count to Infinity” problem of DVR
- Example: deterioration
 - Here: connection destroyed
 - A was previously known, but is now detached
 - The values are derived from [incorrect] connections of distant IS
- Comment
 - Limit "infinite" to a finite value, depending on the metrics, e.g.
 - 'infinite' = maximum path length+1



• For å fikse count to infinity problem: Split horizon algorithm. Prinsipp: generelt publiser distansen til alle nabøer, men hvis naboen vi skal sende til rapporterer at routen har feilet vil vi i stedet for å påstå at vi kan komme dit best gjennom den noden, oppdatere informasjon om at den beste avstanden til den noden ikke lenger er kjent. Distansen mellom to noder settes til uendelig i neste oppdatering hvis noden som til nå trodde den var den beste veien rapporterer at den nå ikke vet en vei til den noden.

- Variant: ‘Split Horizon Algorithm’
- Objective: improve the “count to infinity” problem
- Principle
 - In general, to publicize the “distance” to each neighbour
 - If neighbor Y exists on the reported route, X reports the response “false” to Y
 - distance X [via Y] according to arbitrary i: ∞

- Example: deterioration [connection destroyed]
 - B to C: A = ∞ [real], C to B: A = ∞ [because B is on path to A], ...
- But: still poor, depending on topology, example
 - Connection CD is removed
 - A receives “false information” via B
 - B receives “false information” via A
 - Slow distribution [just as before]

