UNIVERSITETET
I OSLO

ifi

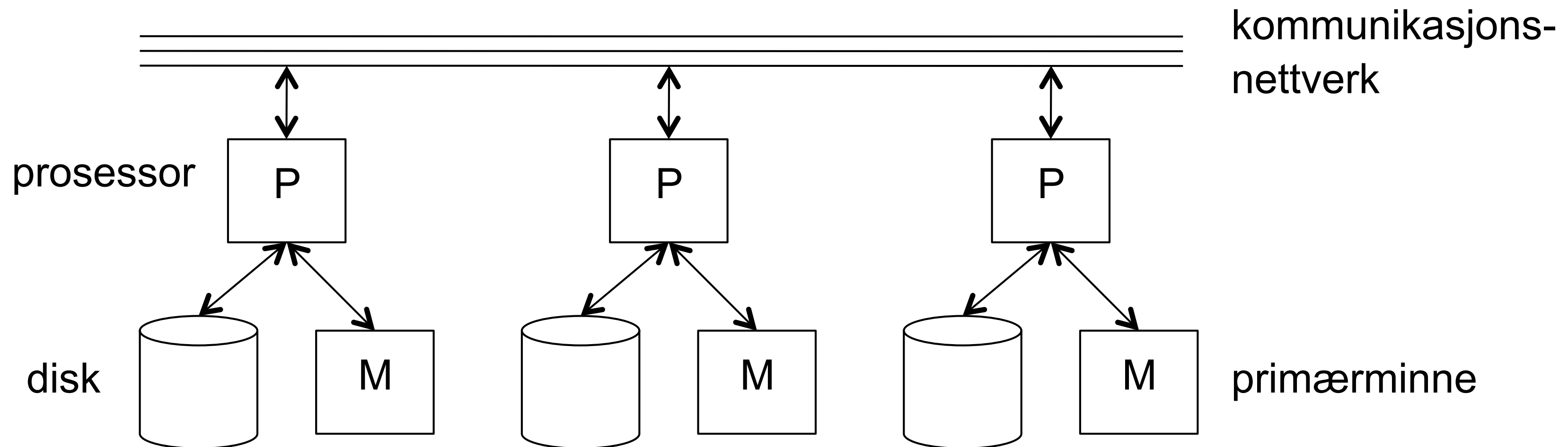
Parallelle og distribuerte databaser – del I

- Databaser på parallellmaskiner; map-reduce
- Distribuerte databaser
  - Distribusjonsmodeller (sharding, replikering)
  - Distribuerte transaksjoner: tofasecommit (2PC), distribuert låsing, vranglåshåndtering
  - CAP-teoremet

Parallellberegninger

- **Database på én storskala parallellmaskin/cluster**: Utnytter parallelliteten på dyre operasjoner, f.eks. join
- Modeller for parallellitet:
  - **Shared-memory-arkitektur**
    - Ett felles fysisk adresserom; hver prosessor har aksess til (deler av) primærminnet til de andre prosessorene
    - Hver prosessor har lokale disker
  - **Shared-disk-arkitektur**
    - Hver prosessor har lokalt primærminne
    - Hver prosessor har aksess til alle disker
  - **Shared-nothing-arkitektur**
    - Hver prosessor har lokalt primærminne og lokale disker
    - Er den arkitekturen som benyttes mest for databasesystemer

Shared-nothing-arkitekturen



- Hver prosessor har lokal cache, lokalt primærminne og lokale disker
- All kommunikasjon skjer ved meldingsutveksling over nettverket som forbinder prosessorene
  - Kostbart å sende data mellom prosessorer (betydelig overhead på hver enkelt melding)
  - Data bør om mulig bufres opp og sendes i større enheter

Parallelle algoritmer for shared-nothing-arkitekturen

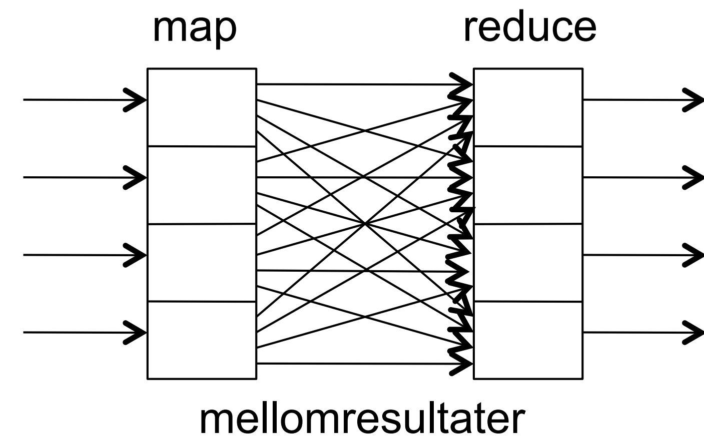
- Hovedprinsippet bak slike algoritmer er å fordele tuplene og arbeidsbelastningen mellom prosessorene. Tuplene fordeles slik at
  - kommunikasjonen minimeres
  - prosessorene kan utføre vanlige algoritmer lokalt på sine tupler
- Eksempler:
  - $\sigma_c(R)$: Fordel tuplene i R jevnt på alle diskene. Hver prosessor utfører σ_c lokalt på sine tupler
  - $R(X,Y) \bowtie S(Y,Z)$: Bruk en hashfunksjon h på Y til å fordele tuplene i R og S på p bøtter (der p er antall prosessorer). Tuplene i bøtte j sendes til prosessor j, som så kan utføre en lokal join på sine tupler

Map-reduce-rammeverket for parallellisering

- Programmeringsmodell
  - Designet for å utføre et høyt antall beregninger i parallell på maskinclustere (eller mer generelt på systemer bestående av et høyt antall maskiner knyttet sammen via et eller annet nettverk)
  - Enkelt å programmere
    - Brukeren skriver kode for to funksjoner, **map** og **reduce**
    - Rammeverket tar seg av parallellisering og distribuering av kode og data
- Arkitektur: Vanligvis shared-nothing-arkitektur
- Datalagring:
  - Filer, typisk svært store
  - Filene er oppdelt i chunks, hver chunk er replikert på tvers av disker for å overleve diskkræsj

Map

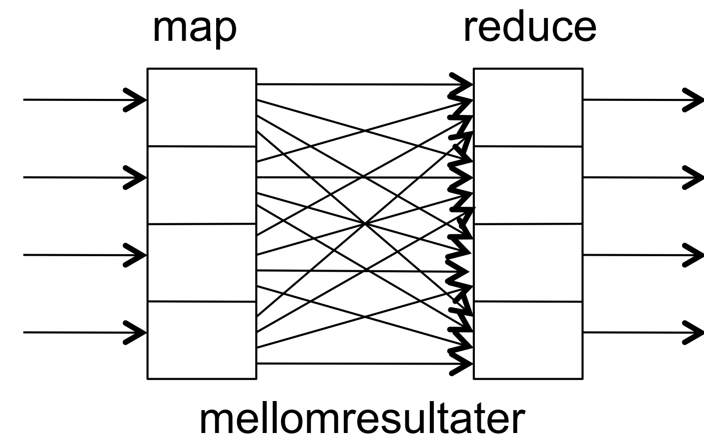
- **Map**-funksjonen tar som input par av formen (k,v) der k er en nøkkel og v en verdi.
- Output (**mellomresultater**) er en liste av par (h,w) der h er en nøkkel og w en verdi



- **Rammeverket** samler prinsipielt alle mellomresultatene for all input i par på formen (h, $[w_1, w_2, ..., w_{n_h}]$), ett for hver verdi av h

Reduce

- **Reduce**-funksjonen produserer for hvert par (h, [w_1, w_2, ..., w_{n_h}]) ett par (h,x) der x er reduksjonen av [w_1, w_2, ..., w_{n_h}] (x er ofte av samme type som w_i-ene)



- **Mastercontrolleren**
  – bestemmer hvor mange map- og reduce-prosesser som skal eksekveres, og på hvilke prosessorer
  – fordeler chunks av inputdata blant map-prosessene
  – bestemmer hvordan verdier i mellomresultatet skal pipelines mellom map- og reduce-prosessene

Eksempel: Beregning av invertert indeks

- Map:
  - Input: Par på formen (i,d) hvor i er en dokumentID og d et dokument.
  - Virkemåte: Scanner d tegn for tegn; for hvert ord w produseres output (w,i) hvor w brukes som nøkkel
- Reduce:
  - Input: Par på formen $(w, [i_1, i_2, ..., i_{n_w}])$
  - Virkemåte: Fjerner flerforekomster i $[i_1, i_2, ..., i_{n_w}]$ og sorterer etter stigende i_k

Distribuerte databaser

- En database kalles **distribuert** hvis den er spredt over flere datamaskiner, kalt **noder** (sites), som er bundet sammen i et nettverk
- Hver node har sitt eget operativsystem og sitt eget DBMS
- Tre viktige formål med distribuerte databaser er:
  - større lagringskapasitet og raskere svartider
  - økt sikkerhet mot tap av data
  - økt tilgjengelighet av data for flere brukere
- Databasen kalles **distribusjonstransparent** hvis brukerne (applikasjonene) ikke merker noe til at databasen er distribuert (bortsett fra variasjon i svartidene)
  - I dag er det en selvfølge at en distribuert database er distribusjonstransparent

Distribusjonsmodeller

- **Sharding**:
  Hver node har ansvaret for et utsnitt (sin del) av dataelementene
- **Replikering**:
  Hvert dataelement er lagret på flere noder
- De to modellene er uavhengige:
  Et system kan bruke en eller begge teknikker

Distribusjon (sharding) av relasjonelle data

- Et eksempel på distribuerte data kan vi finne i en butikkjede hvor alle salg registreres av kassaapparatene og lagres lokalt på en datamaskin (node) i hver enkelt butikk

- Logisk sett har butikkjedens relasjonsdatabase én relasjon som inneholder alle salgsdata fra alle butikkene
- Salgsdataene er **horisontalt fragmentert** med ett fragment på hver node
- Hvis de horisontale fragmentene er disjunkte, er fragmenteringen **total**
- En relasjon er **vertikalt fragmentert** hvis ulike attributter er lagret på ulike noder
- Vertikale fragmenter må inneholde primærnøkkelen
- En vertikal fragmentering er **total** hvis ingen andre attributter enn primærnøkkelen ligger på flere noder

Replikerte data

- I en konsistent tilstand er replikatene like (de er eksakte kopier av hverandre)
- Hvis alle data er replikert til alle noder, har vi en **fullreplikert** database (**speildatabase**)
- Internt bruker DBMSet et replikeringsskjema som forteller hvilke data som ligger på hvilke noder
- Distribusjonstransparens medfører at applikasjonene ikke må ha kjennskap til replikeringsskjemaet og at de ikke har ansvar for å oppdatere replikatene
- Replikering er dyrt, men det gir økt lesehastighet og økt tilgjengelighet til data

Distribuerte transaksjoner

- Når optimalisereren og planleggeren skal lage fysiske eksekveringsplaner, må de ta hensyn til hvilke noder de ulike dataene ligger på (denne informasjonen finnes i replikeringsskjemaet som er kopiert til alle noder)
- Det er to hovedstrategier å velge mellom:
  - Kopier (på billigste måte) de data som trengs, til samme node og utfør eksekveringen der
  - Splitt eksekveringen opp i subtransaksjoner på de aktuelle nodene og gjør mest mulig eksekvering der dataene er (viktig for projeksjon og spesielt seleksjon)
- En god optimaliserer kombinerer de to strategiene for å minimalisere datatransmisjonen mellom nodene

Distribuert commit

- En transaksjon i en distribuert database kan oppdatere data på flere noder (spesielt må dataelementer som er endret, oppdateres på alle noder med replikater)
- Den noden der en transaksjon T initieres, kalles **startnoden** til T
- Planleggeren finner ut hvilke noder T trenger å aksessere, og starter en subtransaksjon på hver av disse (inklusive startnoden) for å gjøre Ts jobb lokalt
- For å oppnå global atomisitet må enten alle subtransaksjonene gjøre commit, eller alle må abortere
- Følgelig kan ikke T gjøre commit før alle subtransaksjonene har gjort det

Tofasecommit

- **Tofasecommit** (**2PC**) er en protokoll for å sikre atomisitet av distribuerte transaksjoner
- 2PC bygger på følgende forutsetninger:
  - To vilkårlige noder kan kommunisere med hverandre
  - Ingen node går ned permanent
  - Hver node sikrer atomisitet for sine lokale transaksjoner
  - Det finnes ingen global logg
  - Hver node logger sine operasjoner (inklusive 2PC-relaterte operasjoner)
- 2PC forutsetter at en av nodene utpekes til **koordinator**. Vanligvis er det startnoden som velges

2PC-protokollen – fase I

- Koordinatoren for en distribuert transaksjon T bestemmer seg for å gjøre commit
  - Koordinatoren skriver <T, Prepared> i loggen på sin node (og skriver loggen til disk)
  - Koordinatoren sender meldingen «***prepare T***» til alle noder som har subtransaksjoner av T
- Hver mottaker fortsetter eksekveringen til den vet om dens subtransaksjon T_i kan gjøre commit
  - Skriv nok i loggen til at det ved behov kan gjøres gjenoppretting på T_i
  - Hvis kan committe,
    - skriv <T, Prepared> i loggen
    - send meldingen «***prepared T***» til koordinatoren

    Deretter: Vent på fase II
  - Hvis må abortere,
    - skriv <T, Abort> i loggen
    - send meldingen «***aborted T***» til koordinatoren

    Deretter: Kan rulle subtransaksjonen T_i tilbake umiddelbart

2PC-protokollen – fase II

- Hvis koordinatoren har mottatt «***prepared T***» fra alle nodene (subtransaksjonene), skriver koordinatoren <T, Commit> i sin logg og sender «***commit T***» til alle andre involverte noder.
  - En node som mottar «***commit T***», gjør commit på sin subtransaksjon og skriver <T, Commit> i loggen sin.
- Hvis koordinatoren har mottatt «***aborted T***» fra minst én node eller ikke alle har svart ved «timeout», skriver koordinatoren <T, Abort> i sin logg og sender «***abort T***» til alle andre involverte noder.
  - En node som mottar «***abort T***», skriver <T, Abort> i loggen sin og ruller tilbake sin subtransaksjon hvis ikke dette alt er gjort.

Initiering av protokollen

- Hvis koordinator bestemmer seg for å committe sin del av den distribuerte transaksjonen, starter den fase I og sender «***prepare T***» til alle.
- Hvis koordinator må rulle tilbake sin del av transaksjonen, kortslutter den protokollen ved å sende «***abort T***» til alle (da trengs bare siste del av fase II).
- Alternativt kan en hvilken som helst av de andre nodene initiere protokollen. Dette kan gjøres som følger:
  - Den første noden som kan committe, logger <T, Prepared> og sender «***beginCommit T***» til koordinator, som så starter fase I av protokollen. (Noden behøver da ikke å sende «***prepared T***» til koordinator siden dette er underforstått av «***beginCommit T***»-meldingen.)
  - Hvis en node må rulle tilbake sin del av transaksjonen, kan den kortslutte protokollen på vegne av koordinator. I såfall logger den <T, Abort> og sender «***abort T***» til alle.

Feilhåndtering ved 2PC - I

- Hvis en *ordinær* node går ned, fortsetter den protokollen når den kommer opp igjen.
- Det er opplagt hva den skal gjøre med mindre dens siste loggpost er <T, Prepared>.
- I så fall må den spørre koordinator eller en annen node om den skal gjøre commit eller abort.

Feilhåndtering ved 2PC - II

- Hvis *koordinatoren* går ned, velges en ny koordinator
- Den nye koordinatoren starter med å innhente status fra alle oppegående noder
- Med ett unntak kan den nye koordinatoren fullføre 2PC:
  - Unntaket er hvis alle nodene har <T, Prepared> som siste loggpost
  - Da kan man ikke avgjøre om den opprinnelige koordinatoren har forberedt commit eller abort
- Det er to mulige fortsettelser:
  - Vente til koordinatoren kommer opp igjen (tar lang tid)
  - DBA griper inn og fatter en manuell avgjørelse (er ikke nødvendigvis brukbart i praksis)

Blokkerende protokoller

- En protokoll er **blokkerende** hvis det at én strategisk node går ned på et kritisk tidspunkt, er tilstrekkelig til å få protokollen til å «henge».
- 2PC er blokkerende: Hvis koordinatoren går ned, kan protokollen bli hengende.
- Blokkering kan unngås ved å splitte opp fase II. **3PC - trefasecommit** - er en protokoll som gjør dette. (Vi tar ikke med beskrivelsen av 3PC her.)

Valg av ny koordinator

- Hvis koordinator går ned i 2PC eller 3PC, må det velges en ny koordinator. Men å velge en *entydig ny koordinator* er komplisert.
  - Å oppnå enighet om en ny koordinator kan sammenliknes med å oppnå enighet om commit, og er like komplisert å gjennomføre som commitprotokollen selv
  - Hvis valget av ny koordinator skal være en ikkeblokkerende protokoll, må man ta høyde for situasjoner der to eller flere noder tror de er den nye koordinatoren
  - I 3PC risikerer man inkonsistens (uenighet om transaksjonen er committet eller abortert) hvis flere noder tror de er ny koordinator
- **En løsning er Paxos commit** (foreleses i del 2).
  - Paxos commit er ikke-blokkerende – ingen enkeltnode kan få protokollen til å henge
  - Paxos commit er konsistent selv om flere noder tror de er koordinator

Låsing i distribuerte systemer

- Låsing av et replikat krever varsomhet:

  Anta at T har leselås på et replikat A_1 av et dataelement A.
  Anta at U har skrivelås på et annet replikat A_2 av A.
  Da kan U oppdatere A_2, men ikke A_1.
  Resultatet blir en inkonsistent database.
- Med replikerte data må vi skille mellom to typer låsing:
  - låsing av et logisk dataelement A (global lås)
  - fysisk låsing av et av replikatene av A (lokal lås)
- Reglene for logiske lese- og skrivelåser er de samme som de som gjelder for vanlige låser i en ikke-distribuert database
- Logiske låser er fiktive – de må implementeres ved hjelp av de fysiske

Sentralisert låsing

- Den enkleste måten å implementere logiske låser på, er å utnevne en av nodene til **låsesjef**
- Låsesjefen håndterer alle ønsker om logiske låser og bruker sin egen låstabell som logisk låstabell
- Det er to viktige svakheter ved et slikt sentralisert låsesystem:
  - låsesjefen blir fort en flaskehals ved stor trafikk
  - systemet er svært sårbart; hvis låsesjefen går ned, får ingen satt eller hevet noen lås
- Kostnaden er minst tre meldinger for hver lås som settes:
  - en melding til låsesjefen for å be om en lås
  - en svarmelding som innvilger låsen
  - en melding til låsesjefen for å frigi låsen

Primærkopilåsing

- Primærkopilåsing er en annen type sentralisert låssystem
- I stedet for en felles låsesjef velger vi for hvert logisk dataelement ut et av replikatene som **primærkopi**
- Den fysiske låsen på primærkopien brukes som logisk lås på dataelementet
- Metoden reduserer faren for flaskehalser ved låsing
- Ved å velge replikater som ofte blir brukt, til primærkopier, reduseres antall meldinger ved håndtering av låser

Avledede logiske låser

- Metoden går ut på at en transaksjon får en logisk lås ved å låse et tilstrekkelig antall av replikatene
- Mer presist:
  Anta at databasen har n replikater av et dataelement A
  Velg to tall s og x slik at $2x > n$ og $s+x > n$
  - En transaksjon får logisk leselås på A ved å ta leselås på minst s replikater av A
  - En transaksjon får logisk skrivelås på A ved å ta skrivelås på minst x replikater av A

  Det er maksimalt n låser til utdeling:
  - At $2x > n$ medfører at to transaksjoner ikke begge kan ha logisk skrivelås på A
  - At $s+x > n$ betyr at to transaksjoner ikke samtidig kan ha henholdsvis logisk leselås og logisk skrivelås på A

Leselås-én, skrivelås-alle

- Dette oppnår vi ved å velge s = 1 og x = n
- Logisk skrivelås krever i verste fall minst 3(n-1) meldinger og blir svært dyr
- Logisk leselås krever høyst 3 meldinger,
  og hvis det finnes et replikat på transaksjonens startnode, krever den ingen meldinger
- Metoden egner seg der skrivetransaksjoner er sjeldne
- Eksempel:
  Elektronisk bibliotek der nodene har kopi av ofte leste dokumenter

Majoritetslåser

- Dette oppnår vi ved å velge $s = x = \lceil (n+1)/2 \rceil$
- Logisk skrivelås kan ikke bli billigere enn dette
- Men det at logisk leselås også krever omtrent 3n/2 meldinger, virker svært dyrt
- I systemer som tilbyr kringkasting av meldinger, blir kostnaden lavere
- Fordelen er at metoden er robust mot nettverksfeil
- <u>Eksempel</u>:
  Dersom en nettverksfeil deler databasen i to, kan den delen som inneholder flertallet av nodene, fortsette som om intet var hendt.
  I minoritetsdelen kan ingen få så mye som en leselås

Distribuert vranglås

- Faren for vranglås i et distribuert låsesystem er stor
- Det finnes mange varianter av Vent-på-grafer som kan forhindre distribuert vranglås
- Erfaring tilsier at det enkleste og beste i de fleste tilfeller er å bruke «timeout»:
  Transaksjoner som bruker for lang tid, rulles tilbake

CAP-teoremet

CAP-teoremet: I et distribuert system er det ikke mulig å til enhver tid tilby samtlige av følgende tre garantier:

- **C**onsistency – en forespørsel vil på et gitt tidspunkt gi samme svar/respons uansett hvilken node den eksekveres på
- **A**vailability – hver forespørsel resulterer i en respons
- **P**artition tolerance – systemet fortsetter å være virksomt selv om meldinger kan gå tapt (deler av nettverket feiler) eller deler av nodene feiler/går ned

CAP-teoremet utdypet – I

- Med **consistency** menes det som dekkes av A+C i ACID, dvs. at en enkelt forespørsel+respons skal utføres atomært og bevare konsistens.

- Med **availability** menes at enhver forespørsel til en ikke-feilet node skal resultere i en respons. (Den opprinnelige formuleringen av CAP-teoremet sier at «så godt som alle» forespørsler skal motta en respons, for i praksis kan man ikke garantere 100% tilgjengelighet.)

CAP-teoremet utdypet – II

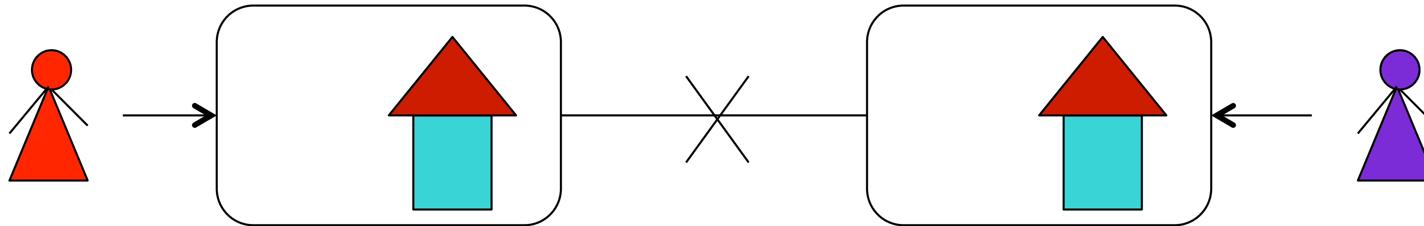
- CAP-teoremet ble opprinnelig formulert med tanke på web service-systemer, men ideene er gyldige uansett hvilket formål det distribuerte systemet har
- En populær, men overforenklet, fremstilling av CAP-teoremet sier at man i ethvert distribuert system må velge «to av tre», dvs. CA, CP eller AP.
- Merk at å velge CA, dvs. velge bort P'en i CAP, ikke er en opsjon i et tjenesteytende distribuert system, i den forstand at man må designe slike systemer med tanke på at systemet skal fortsette å virke selv om deler av nettverket feiler.

CAP-teoremet utdypet – III

Når en nettverkspartisjon oppstår, må man velge mellom konsistens og tilgjengelighet, man kan ikke tilby begge.

- CP: Hvis man velger konsistens fremfor tilgjengelighet, kan operative noder måtte avslå å yte tjenester (dvs. forespørsler må kanskje besvares med «prøv igjen senere» eller liknende)
- AP: Hvis man velger tilgjengelighet fremfor konsistens, kan det være at når nettverket igjen virker som det skal, er det inkonsistenser i systemet der systemet ikke selv kan rydde opp i/bli kvitt alle inkonsistensene

Eksempel: Bestilling av hotellrom



To personer ønsker å bestille samme rom i samme hotell for samme tidsrom. Bestillingssystemet er distribuert over to noder med replikater på begge. Nettverket mellom nodene er nede.

Alternativer:

- CP: Ingen rom kan bestilles (hver av forespørslene besvares med «ikke tilgjengelig/prøv senere» selv om hver av nodene er fullt operativ).
- AP: Begge bestillingene går igjennom (inkonsistens/overbooking). Inkonsistensen rettes når personene ankommer hotellet (ved at hotellet har et ekstra rom som er avsatt til å håndtere overbooking).

CAP-teoremet i praksis

- I praksis er det ikke et enten-eller, men en glidende overgang/tradeoff mellom de to alternativene.
  - Man kan f.eks. velge forskjellig tilnærming (CP/AP) for forskjellige funksjoner/operasjoner
  - Når det ikke er nettverkspartisjoner, bør systemet ha som mål å oppfylle alle de tre CAP-garantiene
- I praksis er det dessuten en tradeoff mellom konsistens og responstid.
  - Jo mer konsistens man krever, desto flere noder må involveres i arbeidet med å garantere dette, og jo lenger tid tar det å respondere på en forespørsel
  - Denne formen for tradeoff gjelder selv om det ikke er noen nettverkspartisjoner