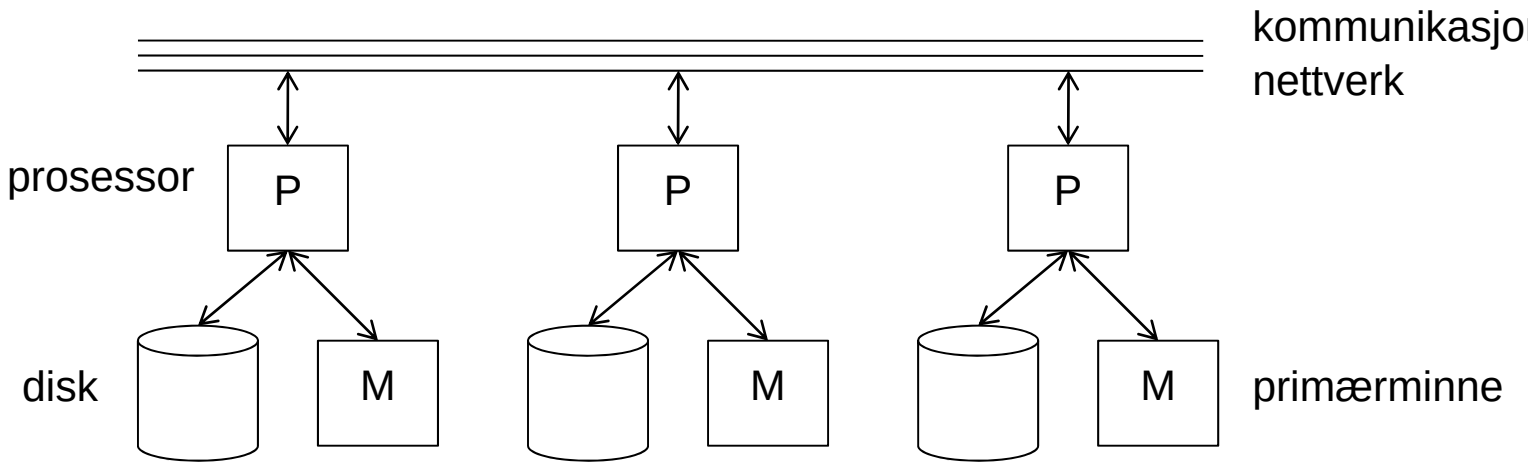
UNIVERSITETET
I OSLO

Parallelle og distribuerte databaser

Parallellberegninger

- **Database på én storskala parallellmaskin**:
  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

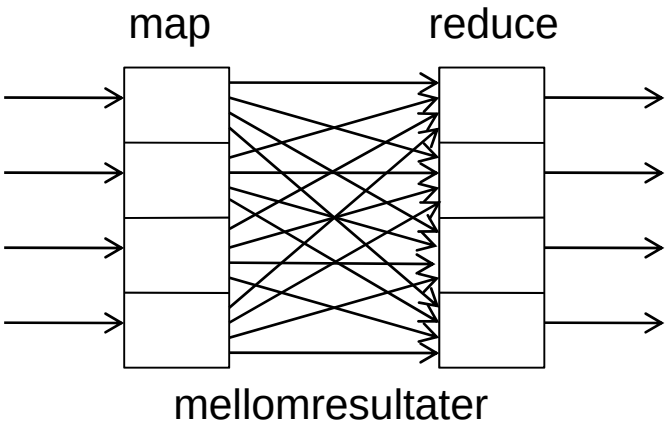
Parallelle algoritmer for shared-nothing-arkitekturen

- 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

- Rammeverk som tillater å enkelt beskrive/programmere databaseprosesser og utføre et høyt antall av dem i parallell
  - Brukeren skriver kode for to funksjoner, *map* og *reduce*.
- Forutsetninger:
  - Massiv parallellmaskin
    - F.eks.: Samling av racks; dedikert kommunikasjonsnettverk mellom maskinene innen hvert rack, enkelt eksternt nettverk mellom rackene
  - 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- og reduce-funksjonene



- Map-funksjonen tar som input (k,v) der k er en nøkkel og v en verdi. Output (mellom-resultater) 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-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_k}] (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])
  - Virkemåte: Fjerner flerforekomster i [i_1, i_2, ..., i_n] 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

Distribusjon av 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
- Vi sier at 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

- Dataelementer som er lagret på flere noder, kalles **replikerte**
- I en konsistent tilstand er replikerte data like
  (de er kopier av hverandre)
- Hvis alle data er replikert til alle noder, har vi en **fullreplikert** database (kalles også en **speildatabase**)
- Internt bruker DBMS 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 hastighet og økt tilgjengelighet til data

Distribuerte transaksjoner og queries

- 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 replikerte dataelementer som er endret, oppdateres på alle noder med replikater)
- Den noden der en transaksjon T initieres, kalles **startnoden** (eller **utgangsnoden**) 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 (2PC)

- 2PC er en protokoll for å sikre atomisitet av distribuerte transaksjoner
- 2PC bygger på følgende forutsetninger
  - Det finnes ingen global logg
  - Hver node logger sine operasjoner
    (inklusive meldinger den har sendt til andre noder,
    til bruk ved gjenoppretting etter nettverksfeil)
  - Hver node sikrer atomisitet for sine lokale transaksjoner
- 2PC forutsetter at en av nodene utpekes til koordinator
  Vanligvis, men ikke alltid, er det startnoden som velges

2PC-protokollen – fase I

- Koordinatoren for en distribuert transaksjon T bestemmer seg for å gjøre commit
- Koordinatoren skriver <Prepare T> i loggen på sin node
- Koordinatoren sender meldingen ***prepare T*** til alle noder som har subtransaksjoner av T
- Hver mottager fortsetter eksekveringen til den vet om dens subtransaksjon T_n kan gjøre commit
- Hvis ja,
  – skriv nok i loggen til at det kan gjøres redo på T_n
  – skriv <Ready T> i loggen og skriv loggen til disk
  – send meldingen ***ready T*** til koordinatoren
- Hvis nei,
  – skriv <Don't commit T> i loggen
  – send meldingen ***don't commit T*** til koordinatoren

2PC-protokollen – fase II

- Hvis koordinatoren har mottatt ***ready T*** fra alle nodene (subtransaksjonene)
  - skriver koordinatoren <Commit T> i sin logg og
  - sender ***commit T*** til alle andre involverte noder
- Hvis koordinatoren har mottatt ***don't commit T*** fra minst én node eller ikke alle har svart ved «timeout»
  - skriver koordinatoren <Abort T> i sin logg og
  - sender ***abort T*** til alle andre involverte noder
- En node som mottar ***commit T***, gjør commit på sin subtransaksjon og skriver <Commit T> i loggen sin
- En node som mottar ***abort T***, aborterer sin subtransaksjon og skriver < Abort T> i loggen sin

Feilhåndtering ved 2PC

- Hvis en ordinær node går ned, er det opplagt hva den skal gjøre med mindre dens siste loggpost er <Ready T>
  I så fall må den spørre en annen node om den skal gjøre commit T eller abort T
- Hvis koordinatoren går ned, velges en ny koordinator
- Med ett unntak kan den nye koordinatoren fullføre 2PC
- Unntaket er hvis alle nodene har <Ready T> som siste loggpost
  Da kan man ikke avgjøre om den opprinnelige koordinatoren har gjort commit eller abort
  Det er to mulige fortsettelser
  - Vente til koordinatoren kommer opp igjen
  - DBA griper inn og fatter en manuell avgjørelse

Låsing i distribuerte systemer

- Låsing av et replikert dataelement krever varsomhet:

  Anta T har leselås på en kopi A_1 av et dataelement A
  Anta U har skrivelås på en annen kopi 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 en av kopiene 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 avledes 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 en av kopiene som **primærkopi**
- Den fysiske låsen på primærkopien brukes som logisk lås på dataelementet
- Metoden reduserer faren for flaskehalser ved låsing
- Ved å velge kopier 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 kopier 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 kopier av A
  - en transaksjon får logisk skrivelås på A ved å ta skrivelås på minst x kopier 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 minst 3(n-1) meldinger og blir svært dyr
- Logisk leselås krever høyst 3 meldinger,
  og hvis det finnes en kopi på transaksjonens startnode, krever den ingen
- 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
- Eksempel:
  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 sier at det enkleste og beste i de fleste tilfeller er å bruke «timeout»:
  Transaksjoner som bruker for lang tid, rulles tilbake

Peer-to-peer-systemer

- Et P2P-nettverk er et nettverk av maskiner som
  - er **autonome**: noder kan bli med i eller forlate nettverket etter eget forgodtbefinnende
  - er **løst koblet**: kommunikasjon via et universalnett (f.eks. Internett)
  - har **lik funksjonalitet**: ingen leder- eller kontrollnoder
  - **deler ressurser** med hverandre

Databaser på peer-to-peer-systemer

- En database i et P2P-nettverk må ivaretas kollektivt av nodene i nettverket. Krav til databasesystemet:
  - **distribuert**: data er fordelt på nodene i nettverket
  - **desentralisert**: alle nodene har samme administrative ansvar for databasen
  - **feiltolerant**: systemet må være pålitelig selv om noder feiler og selv om den mengden av noder som utgjør nettverket, kontinuerlig endres
  - **skalerbart**: systemet må være effektivt også for store nettverk

Distributed hashing

- **Distributed hash tables** (DHT) er en klasse av teknikker for å realisere **oppslagstjenester i P2P-nettverk**
- Anta at databasen består av skjemaet R(K,V) der K er primærnøkkel og V tilhørende verdi. En DHT består av
  - en **nøkkelpartisjonering**, dvs. en hashfunksjon h
    - For hvert tuppel (k,v) i R brukes h(k) til å beregne identiteten til den noden som har ansvaret for å lagre tuppelet (k,v)
  - et **overleggsnettverk** (**overlay network**), dvs. en logisk struktur som knytter sammen nodene i nettverket
    - Hver node trenger bare å ha kjennskap til en liten del av nettverket
    - Likevel kan noden som lagrer (k,v), nås fra en hvilken som helst annen node med et lite antall meldingsutvekslinger

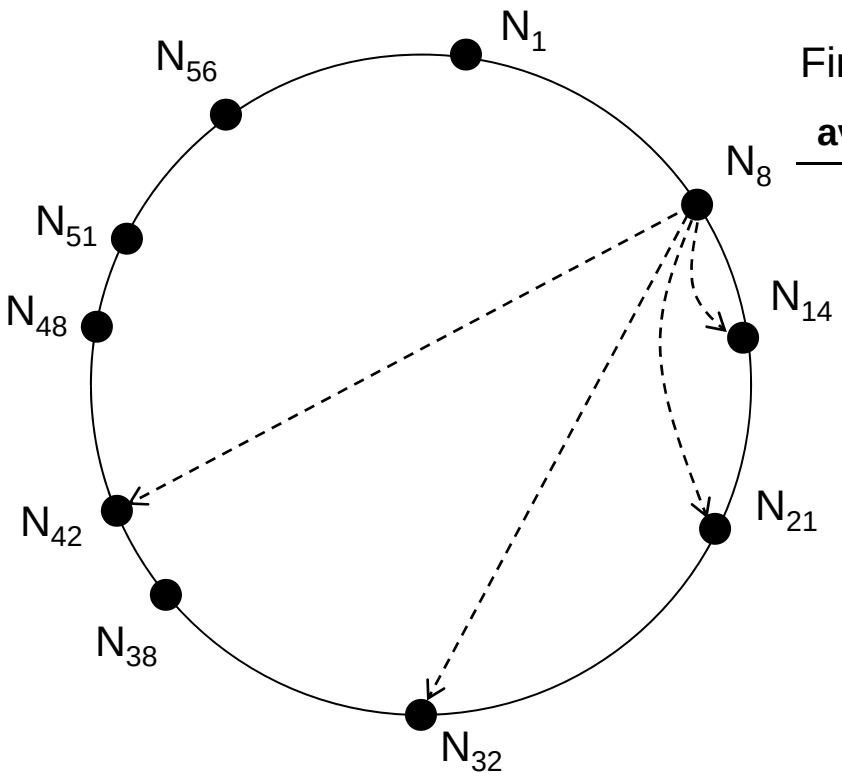
Eksempel på en DHT: Chord

- Overleggsnettverket består av
  - en **sirkel** som omfatter alle nodene i nettverket
  - strenger (**chords**) ”på tvers” i sirkelen
- Hashfunksjon:
  - $h(x) = x \bmod 2^m$ for et passende heltall m
  - maksimalt antall noder er 2^m

Overleggsnettverket i Chord

- En node med ID w plasseres i sirkelen i posisjon h(w)
  - Vi antar at alle noder har en ID og at disse er entydige
- Hver node har en peker til sin forgjenger og etterfølger i sirkelen
- I tillegg har hver node en **fingertabell** med peker til de nodene som har posisjon j+1, j+2, j+4, ..., $j+2^{m-1}$ der j er nodens egen posisjon i sirkelen
  - Hvis det ikke er noen node i posisjon $j+2^i$ for en i, inneholder fingertabellen den noden med urviseren som er nærmest denne posisjonen. (Posisjonene beregnes modulo 2^m.)
  - Fingertabellen har lengde m og tar derfor ikke stor plass selv for store nettverk (dvs. stor m)

Eksempel: m=6



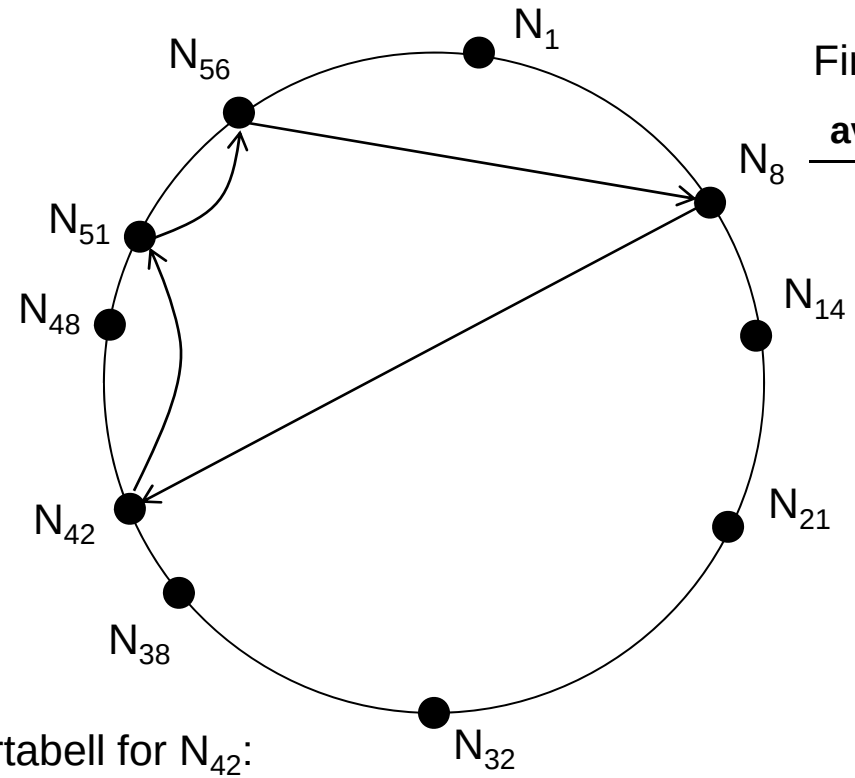
Fingertabell for N_8:

avstand	1	2	4	8	16	32
node	N_{14}	N_{14}	N_{14}	N_{21}	N_{32}	N_{42}

Datahåndtering i Chord

- Et tuppel (k,v) i tabellen R(K,V) lagres på noden som har posisjon h(k).
  - Hvis det ikke er noen node i posisjon h(k), lagres tuppelet på den noden som er nærmest denne posisjonen (med urviseren).
- Hvordan finne et tuppel i nettverket: Anta at node N_i ønsker å finne v for en gitt k, men ikke har tuppelet lagret lokalt. Beregn j=h(k). Sett $N_c = N_i$.
  - La N_s være etterfølgeren til N_c. Hvis $c < j \leq s$, ligger tuppelet på N_s. Send en melding til N_s og be den om å sende tuppelet med nøkkel k direkte til N_i.
  - Ellers: N_c slår opp i fingertabellen sin og finner den noden N_h der h er størst mulig, men mindre enn j. Tuppelet (k,v) må befinne seg i en posisjon lenger ut i sirkelen enn h. N_c sender en melding til N_h og ber den om å overta rollen som N_c. Gjenta fra forrige strekpunkt.
- Det trengs maksimalt m+1 meldinger:
  m forespørselsmeldinger + 1 melding med resultatet.

Eksempel:
N_8 ønsker å finne (k,v) der h(k)=52



Fingertabell for N_8:

avstand	1	2	4	8	16	32
node	N_{14}	N_{14}	N_{14}	N_{21}	N_{32}	N_{42}

Fingertabell for N_{42}:

avstand	1	2	4	8	16	32
node	N_{48}	N_{48}	N_{48}	N_{51}	N_1	N_{14}

Tuppelet (k,v) ligger på den noden som ligger nærmest posisjon 52 (regnet med urviseren), dvs. node N_{56}. Men N_8 kjenner ikke til node N_{56}.

1. Nærmeste foran posisjon 52 i fingertabellen til N_8 er N_{42}. N_8 sender en forespørsel til N_{42}.
2. Nærmeste foran posisjon 52 i fingertabellen til N_{42} er N_{51}. N_{42} sender en forespørsel til N_{51}.
3. Siden etterfølgeren til N_{51} er N_{56}, må tuppelet befinne seg der. N_{51} ber N_{56} sende tuppelet til N_8.

Mobile ad-hoc-nettverk

- Mobile ad-hoc-nettverk (MANETs) er trådløse nettverk som settes opp mellom noder som kommer i kommunikasjonsavstand
- MANETs er karakterisert ved
  - hyppige endringer i den underliggende topologien
    - noder forflytter seg
    - naboskap endres
    - nettverk partisjoneres og smelter sammen
  - kommunikasjon skjer via radiobåndet
    - stor feilrate (kollisjoner av meldinger i eteren)
    - lav båndbredde
  - hver nettverkspartisjon består av i høyden 50-100 noder
    - mer enn dette er ikke praktisk håndterbart (pga. feilraten)

Databaser i mobile nettverk

- Krav til databasesystemet: Som for P2P-nettverk, men i tillegg er det utfordringer knyttet til nettverkskapasitet og hyppige endringer i nettverkstopologien

Eksempel: MIDAS Dataspace

- EU-prosjekt 2006-2009 med åtte internasjonale partnere.
  - Fra Ifi: Forskningsgruppen DMMS (Distribuerte multimediasystemer)
- Formål: Mellomvare som støtter utvikling av applikasjoner for MANETs
- Applikasjonsscenarier:
  Redningsaksjoner og store sportsarrangementer
- Prototyp som er testet ut i emuleringsomgivelse og på små håndholdte enheter (PDAer)
- Proof-of-concept-applikasjoner:
  - Det 32. internasjonale sykkelcrossløpet i Gieten, Nederland, 2007
  - De Nijmegen Vierdaagse, 2008
    (firedagers turmarsj i Nijmegen, Nederland)

Datahåndtering i MIDAS I

- Dataelementer er replikert på tvers av noder for at noder skal beholde sin funksjonalitet ved nettverkspartisjoner
  - noder kan fortsette å jobbe på sin lokale kopi
- Replikater må holdes konsistente (de skal representere samme logiske enhet)
- Antall replikater må balanseres mot økt nettverkstrafikk
  - oppdateringer av et dataelement krever meldingsutveksling mellom alle replikatene

Datahåndtering i MIDAS II

- Konsistens på tvers av nettverkspartisjoner kan ikke garanteres
  - noder i forskjellige nettverkspartisjoner kan ikke kommunisere
  - replikater i forskjellige nettverkspartisjoner kan derfor utvikle seg ulikt
- Når to partisjoner smelter sammen, må konsistens gjenopprettes
- Løsning: Versjonering – ingen sletting eller endring av gamle data
  - to replikater samordnes ved å utveksle alle versjoner av dataelementet
  - applikasjoner må selv velge hvordan semantiske konflikter skal løses (hvilken versjon av et dataelement som er ”den rette”)
  - selv med versjonering (ingen data slettes) forblir datamengden overkommelig i de gitte applikasjonsscenariene

Oppslagstjenester i MIDAS

- Hver node vedlikeholder en oversikt (**metadata**) over hvilke replikater som befinner seg på hvilke noder
  - Når nye replikater opprettes, propageres metadata om dette til alle noder i nettverkspartisjonen
    - Radiobølger betyr at meldinger kringkastes (alle i lytteavstand lytter på alle meldinger som sendes). Dette utnyttes i valg av algoritmer for metadatapropagering
    - Størrelsen på total mengde metadata forblir liten i applikasjonsscenariene
  - Når to partisjoner smelter sammen, utveksles metadata
  - En node vil ikke nødvendigvis ha en perfekt global oversikt fordi det tar tid før endringer i metadata er propagert til alle
    - Propagering av metadata i en nettverkspartisjon krever flere meldinger når ikke alle noder er i direkte kommunikasjonsavstand til hverandre (**multihop**)