Tid og koordinering

Foreleser: Olav Lysne

Bakgrunn

- Distribuerte koordineringsprotokoller
  - har ofte behov for en “hendte-før” relasjon mellom hendelser
    - *gjensidig utelukkelse* blandt en samling prosesser (som i NFS)
    - *replikering*: oppdatering på replikerte data må skje i samme rekkefølge i alle replika
- Logiske klokker
  - verktøy for å ordne distribuerte hendelser uten å vite nøyaktig når de inntraff
  - kan benyttes til å realisere “hendte-før” relasjonen
  - *fysiske klokker* kan ofte ikke benyttes til dette bl.a. fordi deres tidsoppløsning ikke alltid er tilstrekkelig, og fordi egenskapene til asynkron meldingsutveksling begrenser nøyaktigheten vi kan synkronisere klokkene med [Lamport 78]

Logiske klokker

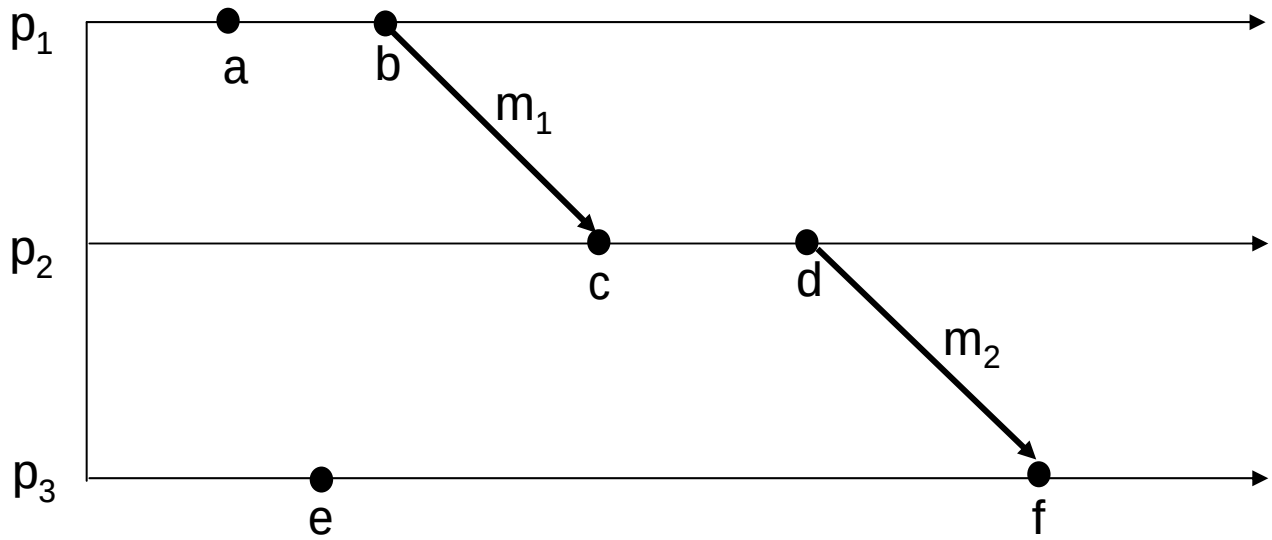
- Ide:
  - To hendelser vil alltid observeres i samme rekkefølge av uavhengige observatører hvis den ene hendelsen var årsaken til den andre
- Prinsipp
  - Hvis to hendelser skjer i samme prosess, da forekommer de i den rekkefølgen som prosessen observerte dem
  - Når en melding sendes mellom to prosesser, vil hendelsen "send meldingen" alltid skje før hendelsen "motta meldingen"
- *Hendte-før* relasjonen
  - fremkommer ved å generalisere de to relasjonene over

“Hendte-før” relasjonen

- Notasjon
  - $x \rightarrow^{p} y$: x hendte før y i prosessen p
  - $x \rightarrow y$: x hendte før y
- HF1
  - Hvis $\exists$ prosess p : $x \rightarrow^{p} y$, da er $x \rightarrow y$
- HF2
  - For enhver melding m : $send(m) \rightarrow rcv(m)$
- HF3
  - Hvis x, y og z er hendelser slik at $x \rightarrow y$ og $y \rightarrow z$, da er $x \rightarrow z$

"Hendte-før" relasjonen II

- Grafisk notasjon (diagram)



- Hendelser som ikke er relatert ved "hendte-før" relasjonen kalles *samtidige*: `a || e`

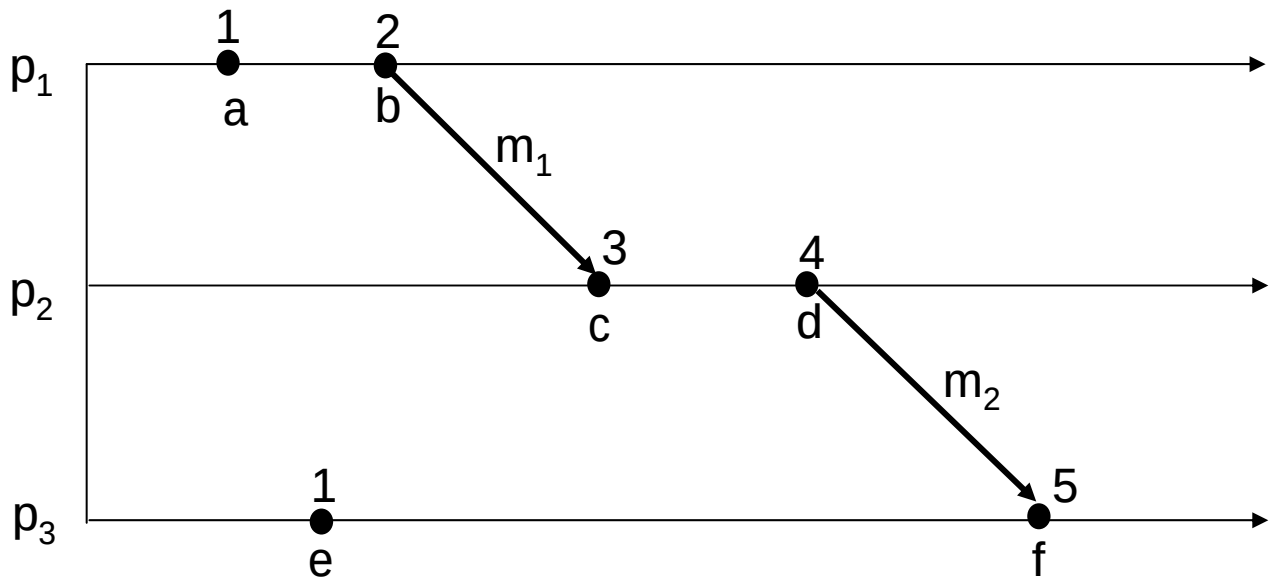
“Hendte-før” relasjonen III

- Ikke alle relasjoner mellom hendelser kan modelleres ved “hendte-før” relasjonen
  - f.eks. når årsaken til en hendelse er noe annet enn mottak av en melding
- Alle hendelser innen samme prosess vil være relatert gjennom “hendte-før” relasjonen, selv om det ikke er noen *virkelig* forbindelse mellom dem

Modell for logiske klokker

- Hver prosess p har sin egen logiske klokke C_p
  - monotont økende programvareteller
  - benyttes til å tidsmerke hendelser
- $C_p(a)$: tidsmerket til hendelsen a i prosess p
- Regler for logiske klokker
  - LK1:
    - C_p inkrementeres med 1 før hver hendelse skjer i prosess p
  - LK2:
    - Når en prosess p sender en melding m, vil m være bærer av verdien $t = C_p$
    - Når (m,t) mottas av en prosess q, beregner q $C_q := max(C_q, t)$, og anvender LK1 før hendelsen $rcv(m)$ tidsmerkes.

Eksempel logiske klokker



$x \rightarrow y \Rightarrow C(x) < C(y)$ (ikke ekvivalens!!)

Total ordning av hendelser

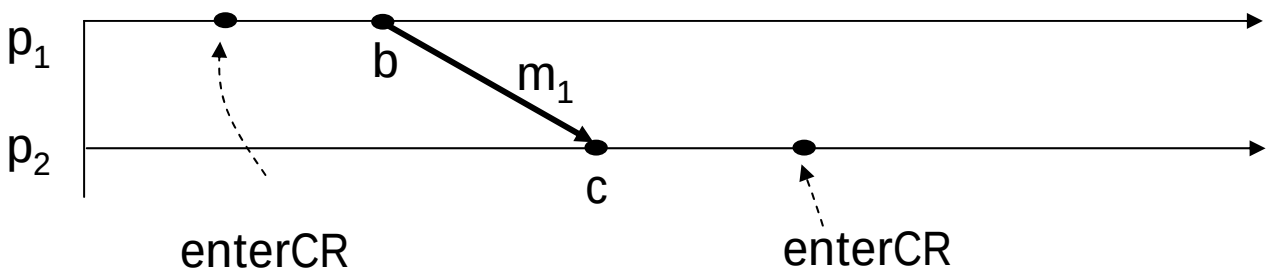
- Logiske klokker gir en partiell ordning av hendelser
  - par av hendelser generert av forskjellige prosesser kan ha identiske tidsmerker
- Utvidelse til total ordning
  - hver prosess har en (global) prosessidentifikator p
  - mengden av prosessidentifikatorer er totalt ordnet
- Globalt tidsstempel
  - a er en hendelse i prosess p_a med lokalt tidsstempel T_a
  - globalt logisk tidsstempel for a er (T_a, p_a)
- Ordning av globale tidsstempel
  - $(T_a, p_a) < (T_b, p_b)$ hviss enten $T_a < T_b$, eller $T_a = T_b$ og $p_a < p_b$

Behov for Distribuert koordinering

- deling av ressurser krever ofte gjensidig utelukkelse
- mange tjenere er ikke konstruert for å handtere synkronisert aksess til de ressurser de forvalter
  - deling av filer som i NFS
  - felles vindu for distribuerte prosesser
  - => behov for separat synkroniseringsmekanisme
- i DS: intet sentralt OS som tilbyr gjensidig utelukkelse
- => behov for separat, generisk mekanisme for distribuert gjensidig utelukkelse
- To tilnærminger til distribuert gjensidig utelukkelse
  - algoritme basert på sentral tjener
  - distribuert algoritme som benytter logiske klokker

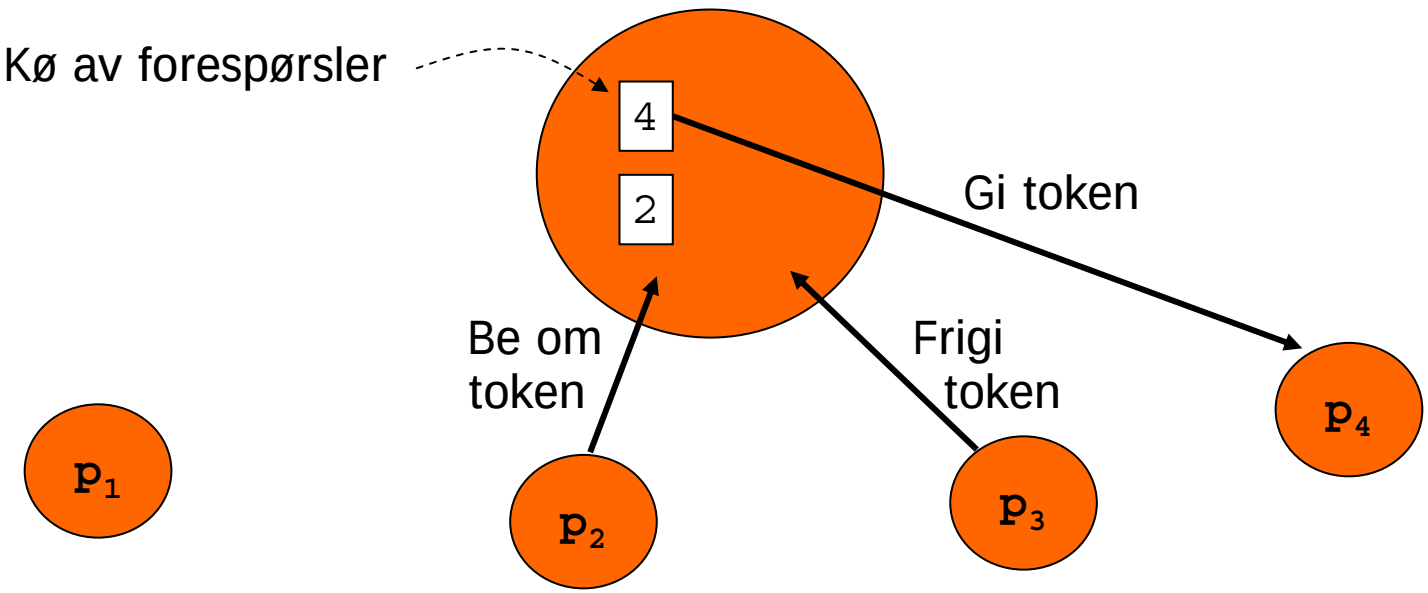
Krav til algoritme for gjensidig utelukkelse

- GU1:
  - Høyst en prosess kan eksekvere i en kritisk region av gangen
- GU2:
  - En prosess som ber om adgang til en kritisk region vil før eller siden få tillatelsen (så lenge enhver prosess som eksekverer i den kritiske regionen før eller siden forlater den)
- GU3 (valgfri):
  - Adgang til en kritisk region må skje i “hendte-før” rekkefølge



Sentral tjener algoritme

- Sentral tjener som gir adgang til kritisk region
- protokoll
  - enter() -- gå inn i kritisk region - blokker om nødvendig
    ..... -- aksesser delt ressurs i kritisk region
    exit() -- forlat kritisk region - andre prosesser kan nå gå inn



Evaluering
sentral tjener algoritme

- Er GU1 og GU2 oppfylt?
- Er GU3 oppfylt?
  - Hvordan sikre det?
- Svakheter sentral tjener?
  - Flaskehals for ytelse
  - Tjeneren kan feile
    - Kan velge ny tjener blant klientene
    - Krever distribuert valg-algoritme
    - Den nye tjeneren innhenter tilstanden til de øvrige klientene
    - Hvordan sikre at den gamle ordning (fra før feilen) bevares?
- Klient med token kan feile
  - hvordan sikre at tokenet blir tilgjengelig igjen?

Ringbasert algoritme

- Et token sendes rundt ringen i en retning.
- En prosess kan bare gå inn i kritisk region når den har tokenet.
- Når en prosess mottar et token og ikke trenger det sendes det videre.

Evaluering av ringbasert algoritme

- Feiltoleranse
  - Problematisk når en node går ned.
    - Koble sammen ringen igjen
    - Sikre at ringen inneholder akkurat ett token
- Ingen sentral flaskehals
  - Men meldinger sendes selv om ingen har behov for å gå inn i kritisk region.
- GU1 og GU2 er trivielt oppfylt, men GU3 er vanskelig.

Distribuert algoritme basert på logiske klokker

- Grunnleggende ide [Ricart & Agrawala, 1981]:
  - En prosess som ønsker å gå inn i en kritisk region, sender en multicast-melding (til de andre prosessene)
  - Prosessen kan kun gå inn i den kritiske regionen når de andre prosessene har svart på meldingen
  - Regler for når prosessene sender svaret, sikrer kravene GU1 - GU3
- Antagelser
  - Prosessene kjenner hverandres adresser
  - Sendte meldinger vil før eller siden leveres
  - Hver prosess har en logisk klokke og følger reglene LK1 og LK2.
  - Tidsstempler inkluderer prosessId: `<T,p>` (dvs. total ordning)
  - Hver prosess holder oversikt over sin tilstand mhp. token-eie
    - RELEASED, WANTED, HELD

Ricart & Agrawalas algoritme

Ved initialisering

state := RELEASED;

For å få tokenet

state:= WANTED;

Multicast forespørsel til alle prosessene

T := anropets tidsstempel;

wait until (antall svar mottat = (n-1));

state := HELD;

Ved mottak av en forespørsel $<T_i, p_i>$ i p_j $(i \neq j)$

if (state=HELD *or* (state=WANTED *and* $(T,p_j) < (T_i,p_i)$))

then

kø forespørsel fra p_i uten å svare

else

svar umiddelbart til p_i

end if

Ved frigivelse av token

state := RELEASED

svar på evt køet melding

Evaluering
Ricart & Agrawalas algoritme

- Er GU1 og GU2 oppfyllt?
- Er GU3 oppfyllt?
- Svakheter distribuert tjener?
  - Høyt antall meldinger for å få tokenet
    - 2(n-1) meldinger uten HW støtte for multicast
    - n meldinger med HW støtte for multicast
  - Dersom en prosess feiler, vil all progresjon stoppe

Oppsummering
distribuert gjensidig utelukkelse

- Er lite robuste overfor feil
  - kan fikses, men krever komplekse feilopprettingsprotokoller
- Er generelt komplekse
  - sentral tjener gir færrest meldingsutvekslinger, men kan bli en flaskehals
- Konklusjon:
  - av og til behov for separat tjeneste for distribuert gjensidig utelukkelse
  - likevel best at tjenere forvalter en ressurs også ved selv å tilby gjensidig utelukkelse (jfr. samtidighetskontroll)

Behov for distribuert valg algoritme

- I mange distribuerte algoritmer vil en av de deltagende prosessene spille en sentral koordinator-rolle
  - sentral tjener i distribuert gjensidig utelukkelse
  - koordinator i en distribuert transaksjon
- Dersom en slik prosess feiler, kan en av de gjenværende prosessene velges til å ta over den sentrale rolle
  - gir større feiltoleranse
- Hovedkrav til algoritme
  - valget må være entydig selv om flere prosesser deltar samtidig

“Bully” algoritmen

- [Silberschatz et al, 1993]
- Forutsetninger
  - prosessene kjenner hverandres identitet og adresser
  - mengden av prosessidentifikatorer er totalt ordnet
  - algoritmen velger deltakeren med størst identifikator som ny koordinator
- Meldingstyper
  - `election`: annoserer et valg
  - `answer`: sendes som svar på election meldingen
  - `coordinator`: annonserer identiteten til den nye koordinator

“Bully” algoritmen II

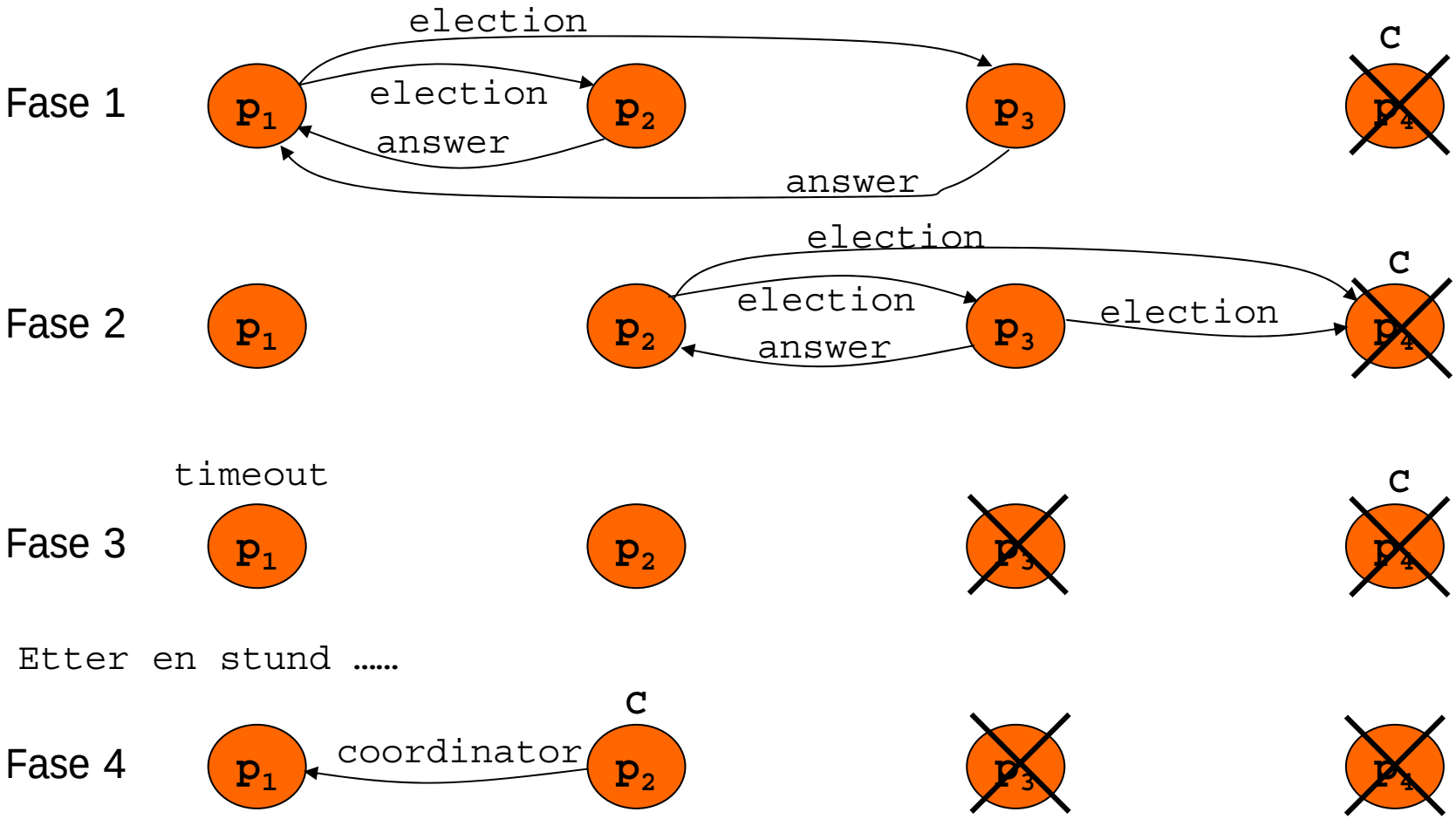
- Valgprosedyre
  - prosessen (som oppdager at koordinator har feilet) sender `election` meldingen til de prosessene som har høyere identitet
  - den venter så på `answer` meldingen en begrenset tid
  - Hvis ingen `answer` melding mottas, anser prosessen seg som ny koordinator og sender en `coordinator` melding til alle prosessene med lavere identitet
  - Dersom `answer` melding mottas, venter prosessen igjen en viss begrenset tid på `coordinator` meldingen. Hvis ingen mottas, starter den et nytt valg.

“Bully” algoritmen III

- Valgprosedyre (forts.)
  - Dersom en prosess mottar en `coordinator` melding, tar den vare på identiteten til koordinatoren (inneholdt i meldingen) og anser prosessen som ny koordinator
  - Dersom en prosess mottar en `election` melding, sender den tilbake en `answer` melding og begynner et nytt valg - med mindre prosessen ikke allerede har startet et valg
  - Når en prosess som har feilet, starter på nytt, begynner den et valg. Dersom den selv har høyeste identitet, avgjør den at den selv er koordinator og annonserer det, selv om en annen koordinator fungerer p.t.

Eksempel

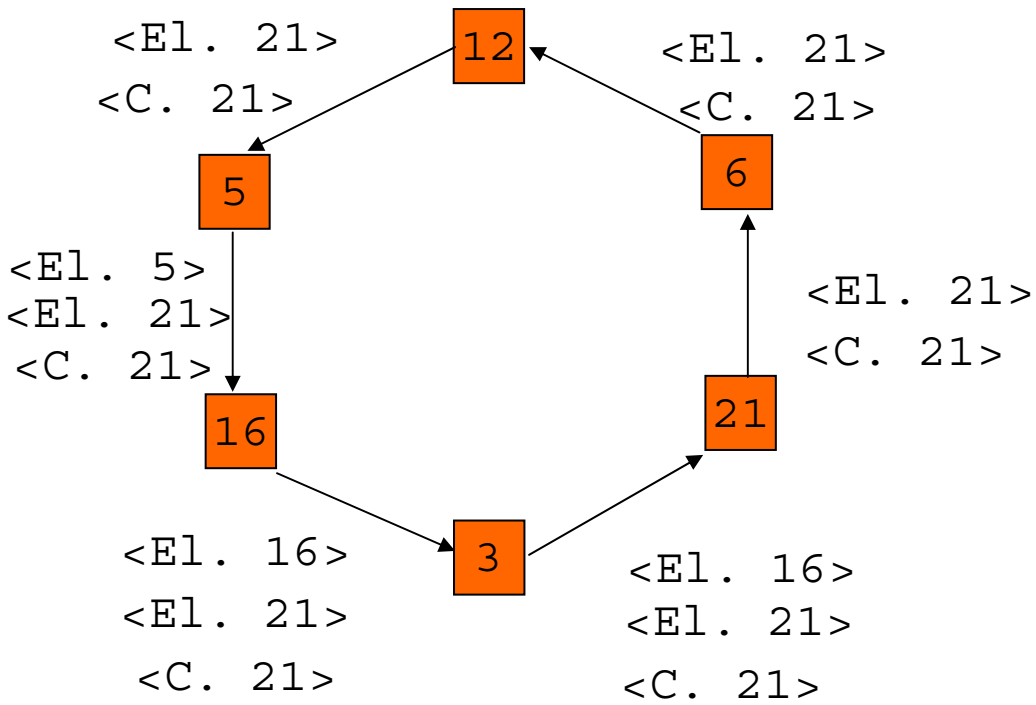
"bully" algoritmen



Evaluering
"bully" algoritmen

- Beste tilfelle: n-2 `coordinator` meldinger
  - skjer når prosessen med nest høyeste identitet oppdager at koordinator har feilet
- Værste tilfelle: $O(n^2)$ meldinger
  - skjer når prosessen med laveste identitet oppdager at koordinator har feilet
  - => (n-1) prosesser starter valg
- Ring-baserte algoritmer er mer effektive mhp antall meldinger

Ringbasert algoritme



Når en melding har gått hele veien rundt uten at prosessnummeret har endret seg, vil en prosess oppdage at den har høyest prosessnummer

Den må da informere de andre om dette gjennom en ny melding som går rundt.

…i tillegg vedlikeholdes en datastruktur for å håndtere flere valg som blir startet samtidig.

Lokale hendelser og tilstander

Historien (h) til en prosess modelleres som en sekvens av hendelser og påfølgende tilstander:

$$h_i = e_i^1 \leftrightarrow s_i^1, e_i^2 \leftrightarrow s_i^2, e_i^3 \leftrightarrow s_i^3, \cdots$$

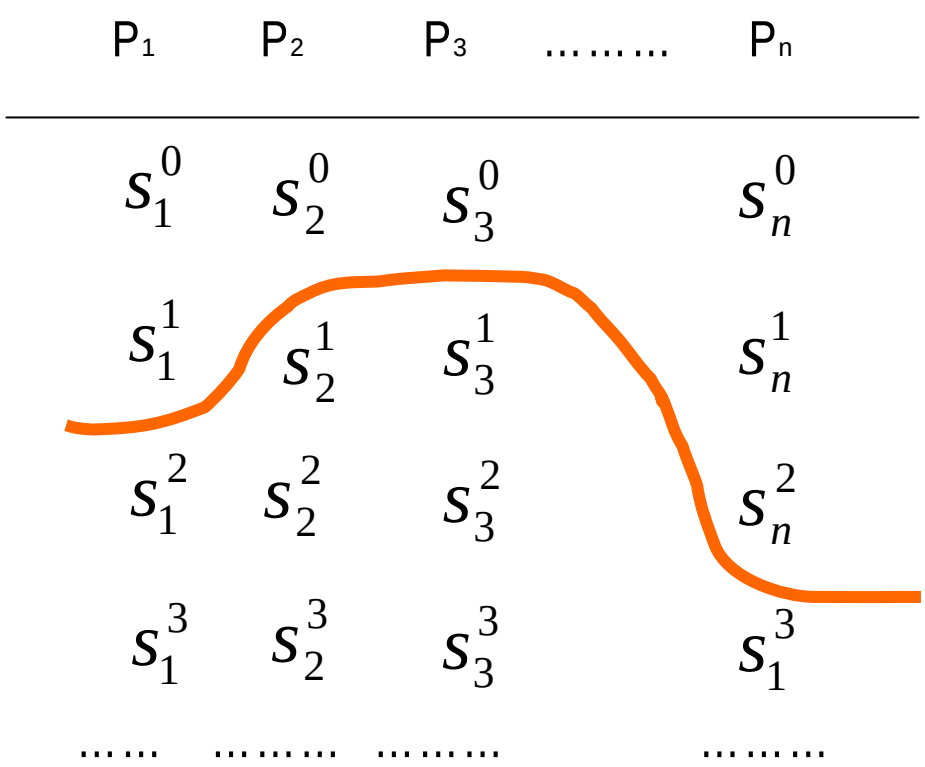
Noen ganger er vi bare interessert i hendelsene:

$$h_i = e_i^1, e_i^2, e_i^3, \cdots$$

Debugging – global tilstand

Foreleser: Olav Lysne

Global tilstand



…men det er jo ikke alle tilstander som er konsistente...

…hva hvis

$e_2^1 \rightarrow e_1^1$ ...?

Konsistente tilstander

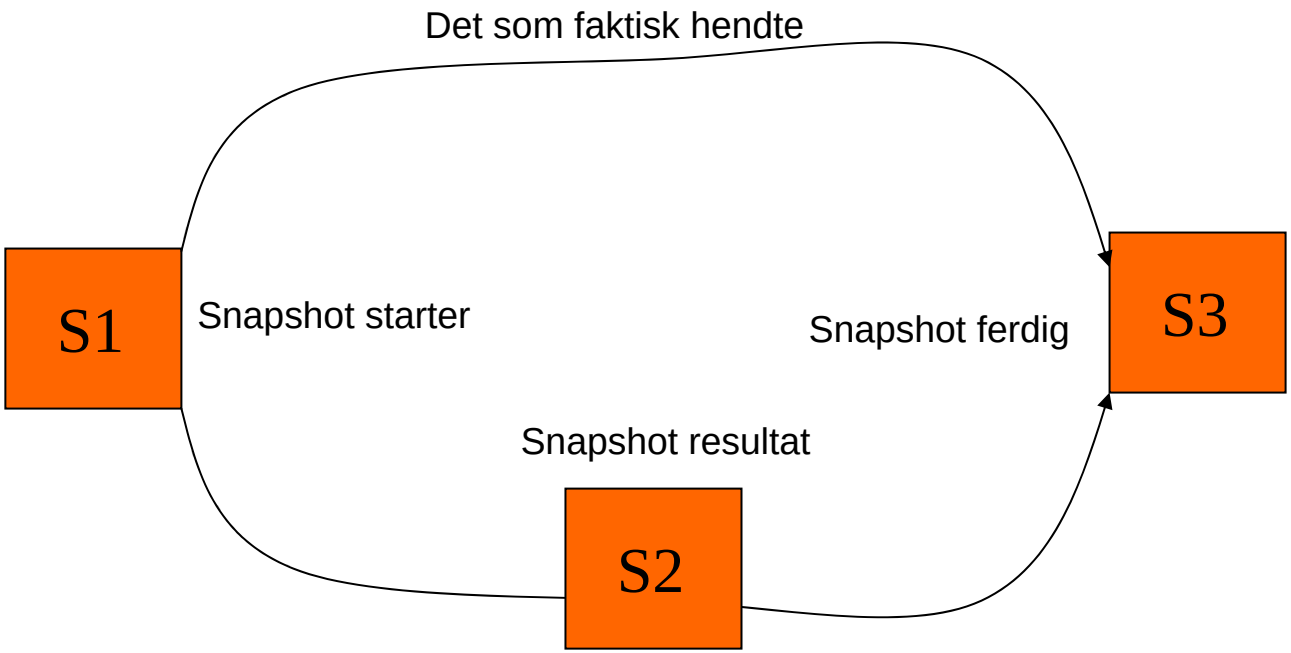
En tilstand C er konsistent dersom

$$e \in C \wedge (f \rightarrow e) \Rightarrow f \in C$$

Vi konsentrerer oss bare om konsistente tilstander!

Snapshot algoritmen

- Foreslått av Leslie Lamport i 1985
- Finner en konsistent global tilstand som *kan* ha forekommet

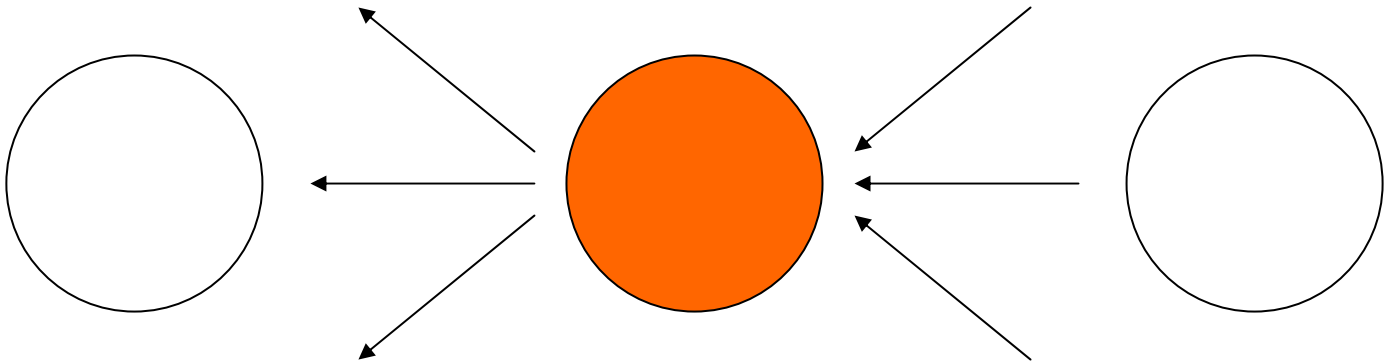


Snapshot II

- Antakelser:
  - Hverken kanaler eller nettverk feiler.
  - Kanaler opererer etter FIFO prinsippet
  - Nettverket av kanaler er komplett, alle prosesser kan nå alle andre prosesser med meldinger gjennom en sekvens av kanaler.
  - Alle prosesser kan initiere et ”snapshot”.
  - Prosessene fortsetter med vanlig prosessering mens snapshotet tas.

Prosessmodell

- Hver prosess deler kanalene sine opp i to sett, de utgående og de innkommende.



Prosessenes ansvar

- Algoritmen bygger på enkel graftraversering
  - En prosess tar initiativ til logging, logger sin egen tilstand, og sender en markørmelding ut på alle sine output kanaler.
- Hver prosess har ansvar for
  - å logge egen tilstand,
  - å logge tilstanden til alle sine innkommende kanaler,
  - sende markøren videre.
- ...men hvordan kan vi vite at dette gir en konsistent tilstand??

Intuisjon

- Hver prosess må sørge for at de som befinner seg i andre enden av en output link logger sin egen tilstand før de får meldinger over linken.
  - De må sende markør umiddelbart etter at de har logget sin egen tilstand.
- Legg merke til at alle prosesser...
  - sender markører over alle sine output linker.
  - mottar markører over alle sine input linker.
  - Når den mottar en markør på en input link vet den at senderen har logget sin tilstand.
- Dersom en prosess mottar en melding fra en input link etter at den har logget sin egen tilstand, men før senderen har logget sin, må prosessen late som om meldingen ikke har kommet.

Prosedyre ved initiering av snapshot

P logfører sin tilstand.

P sender markør over alle output linker.

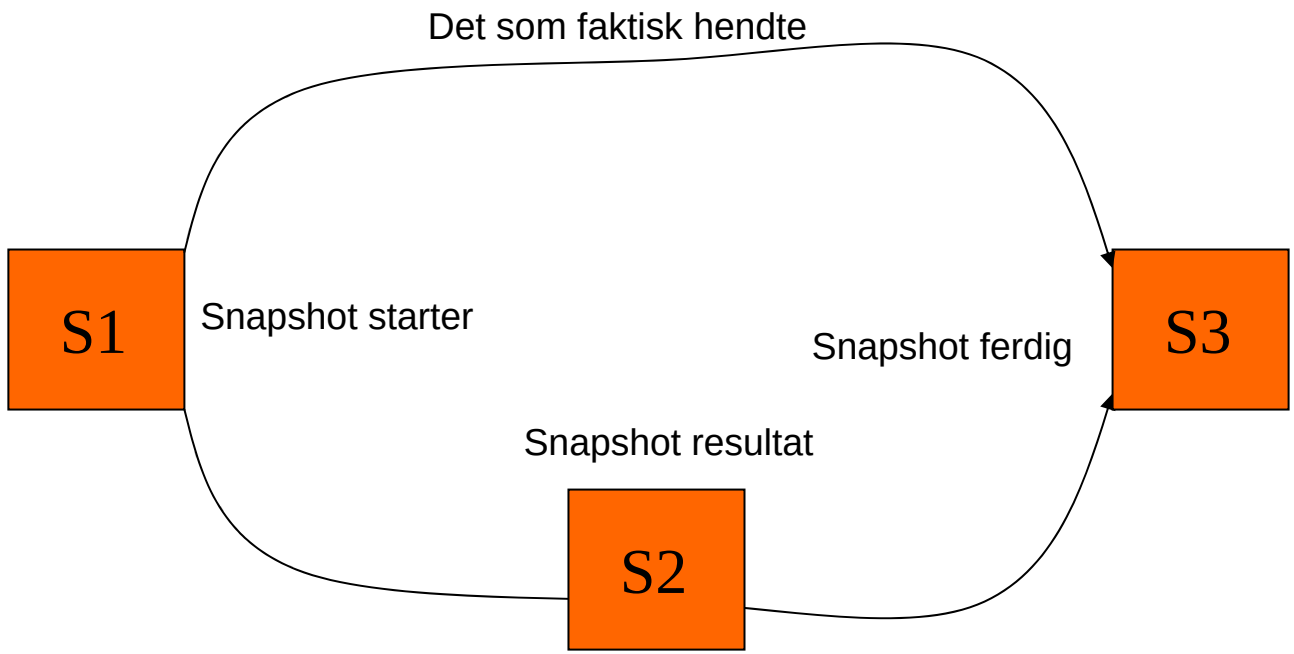
Starter med registrering av innkomne meldinger på alle input kanaler

Prosedyre ved mottak av markør

```
Når P mottar markør over kanal c
  IF P ikke har  logført sin tilstand
     P logfører sin tilstand.
     P videresender markør over alle output linker.
     Logfører at input kanal c er tom
     Starter med registrering av innkomne
        meldinger på alle andre input kanaler
  ELSE
     P logfører tilstanden til c:
           alle de meldingene som er
           mottatt på c siden P logførte
           sin egen tilstand, sies å være
           på vei over kanalen
  END
```

La oss løfte blikket

- Det vi skulle oppnå var...



- Hvordan kan vi vite at vi har fått det til?

Bevisbyrde

- Den registrerte tilstanden er konsistent.
  - Dersom $x \rightarrow y$, og y hendte i p før p logget sin tilstand, så må x ha hendt i q før q logget sin tilstand.
- Tilstand S2 må være nåbar fra S1.
- Tilstand S3 må være nåbar fra S2.
- Bevisene gjennomgås ikke
  MEN DE ER PENSUM

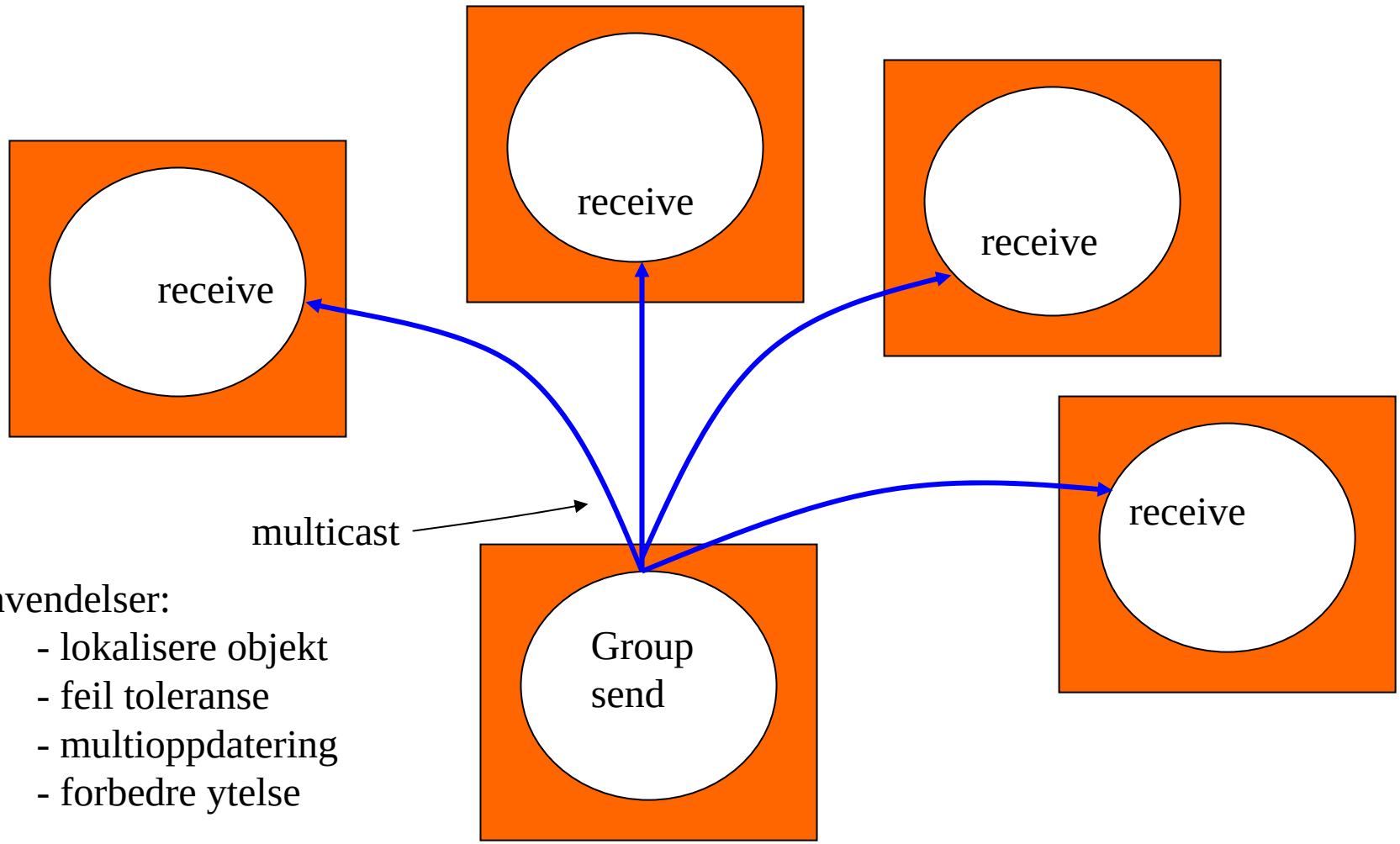
Snapshots begrensninger

- Anta at vi ønsker å få vite hvorvidt noe *virkelig* fant sted.
  - Dette kalles ***definitely Φ***
  - I et kjernekraftverk: var alle ventiler åpne samtidig på noe tidspunkt?
- ...eller hvorvidt noe *kan* ha skjedd
  - Dette kalles ***possibly Φ***
  - I et system som styres av et token som sendes rundt -- kan det tenkes at det på et tidspunkt kan være to token?
- Snapshot algoritmen gir *én* tilstand som *kan* ha skjedd. Vi kan derfor vanskelig benytte den til å svare på spørsmålene over.

Gruppekommunikasjon

Olav Lysne og
Frank Eliassen

Gruppekommunikasjon



Anvendelser:

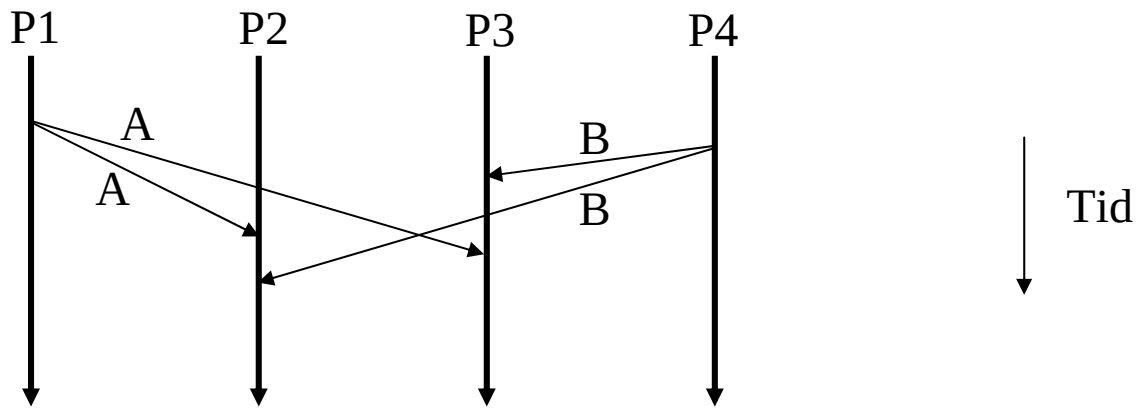
- lokalisere objekt
- feil toleranse
- multioppdatering
- forbedre ytelse

Gruppekommunikasjon II

- Atomisitet:
- ***Atomisk multicast***: En melding som overføres ved atomisk multicast mottas enten av alle prosesser som er medlem i den mottakende gruppe, eller meldingen blir ikke mottat av noen av dem.
- ***Pålitelig multicast***: En melding som overføres ved pålitelig multicast vil leveres til alle prosesser som er medlem i den mottakende gruppe, etter beste evne ("best effort"), men det gis ingen garantier.
- ***Upålitelig multicast***: En melding som overføres ved upålitelig multicast sendes kun en gang.

Gruppekommunikasjon III

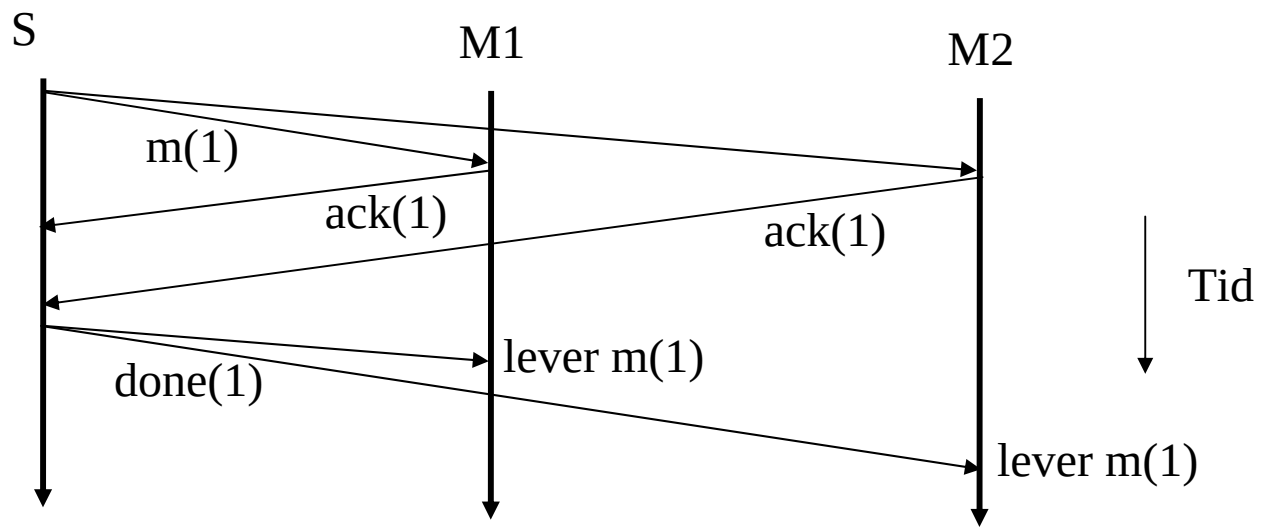
- Ordning:



- *Totalt ordnet multicast*: når flere meldinger overføres til en gruppe prosesser ved totalt ordnet multicast, vil alle meldingene mottas i samme rekkefølge av alle prosessene i gruppen.
  - Feiltoleranse basert på replikerte tjenester, kan realiseres ved totalt ordnet multicast

Implementasjon Gruppekommunikasjon

- Utnytte broadcast/multicast muligheter på nettlaget
- Pålitelighet: krever monitorering (av den som venter på svar)
- Pålitelig meldingsutveksling:



- Problem: Dårlig ytelse (mange meldinger)
- Effektiviseringer: Negative kvitteringer + meldingshistorier

Implementasjon Gruppekommunikasjon II

- Totalt ordnet atomisk multicast
  - Kan implementeres under antagelsen:
    Alle meldinger kan tilordnes en identifikator med en global felles ordning
- Mulige tilnærminger:
  - timestamps fra logiske eller fysiske klokker
  - sequencer prosess som alle meldinger sendes via
  - egen protokoll mellom medlemmene i en gruppe for å generere identifikatorer