Tidsstempelbasert synkronisering i distribuerte, objektorienterte systemer.

Christer Hoel

Institutt for informatikk,
Universitetet i Oslo

15. mai 1995

Forord

Uansett hvor mange ganger en veileder eller andre påpeker hvor viktig det er å sette av *god* tid til avslutningsarbeidet med en hovedfagsoppgave blir de siste ukene før innlevering en opplevelse man ikke glemmer. Finlesning og feilretting har mange ganger vært nære ved å knekke en stakkars student. Leting etter, og lesing av kildelitteratur tar mye tid i starten. Etter at den prosessen er et tilbakelagt stadium tror man gjerne at nå er slitet over og det er *bare å skrive*. Den gang ei. Det er da slitet starter.

En stor takk til min veileder Georg Ræder som har vært dyktig til å lede meg inn på riktig spor igjen når mine egne ideer og min egen entusiasme for disse mildt sagt har vært ufruktbare. Takker også for hans velvilje til å stille opp når jeg har hatt behov for det.

Arbeidet har alt i alt gitt ganske mye. Man lærer seg, i det minste i avslutningsfasen av arbeidet, at planlegging ikke nødvendigvis er et onde. Tilfellene har vært mange der jeg har sittet med en utgave av oppgaven hvor samme ting er omhandlet flere ganger og kanskje med ulike kommantarer og sprikende konklusjoner. Etter intens jobbing den siste tiden er forhåpentligvis dette rettet opp.

Litt vemodig blir man jo også nå når mange års slit av skolebenker snart er over. Skal man glede seg til det som kommer eller burde man egentlig bli en evig student? Hadde ikke hatt noe imot det siste, for studentlivet representerer stort sett en behagelig tilværelse. Man er sin egen herre, noe som også i perioder kan straffe seg. Spesielt dersom herren er et utpreget B-menneske og synes det er fryktelig morsomt å spille gitar hele dagen. «*Skrive ? Nei det haster ikke, det er lenge igjen til levering*», trodde man litt for lenge samtidig som man prøvde å skrive en sang som det aldri ble noe av. Men morsomt har det vært!

Jeg har allerede rettet en takk til Georg. Av andre som må takkes er internveileder Dag F. Langmyhr. Vi har ikke sett så mye til hverandre, men takker for gjennomlesing av oppgaven i avslutningsfasen. De kommentarer og rettledninger som kom fra hans side har vært nyttige og avgjørende for at jeg nå leverer denne oppgaven og ikke venter til neste korsvei. Videre en stor takk til mine foreldre Anne Berit og Kjell Erik, som har lettet studenttilværelsen betraktelig. Velvillig økonomisk støtte

fra dem har ført til at jeg også kan takke både Ringnes og Carlsberg for inspirasjon og mange lyspunkter i tilværelsen.

En takk også til Ola Petter Flem og Jon Martin Solaas for gjennomlesing av oppgaven med dertil nyttige kommentarer. De jobber begge med sine hovedfagsoppgaver, og jeg regner med å måtte betale min gjeld når innleveringstiden nærmer seg.

Til slutt en stor takk til Linda S. Aurland, det største lyspunktet i tilværelsen.

«Far out in the uncharted backwaters of the unfashionable end of the Western Spiral of the Galaxy lies a small unregarded yellow sun. Orbiting this at a distance of roughly ninety-eight million miles is an utterly insignificant little blue-green planet whose ape-descended life forms are so amazingly primitive that they still think digital watches are a pretty neat idea. . . »

- Douglas Adams, «The Hitch-hikers Guide to the Galaxy»

Innhold

Figurer

Kapittel 1

Innledning

I et distribuert, objektorientert system er behovet for ulike synkroniseringsmekanismer stort. I denne oppgaven ser man spesielt ser en på en metode som har vært benyttet til synkronisering innenfor distribuert simulering, der bruk av tidsstempel sammen med meldingsutveksling er sentrale bestanddeler i synkroniseringsmekanismen.

Oppgaven søker å gi svar på om denne modellen, eller varianter av denne, kan benyttes effektivt som parallellitetskontrollmekanisme i et distribuert, objektorientert system. Videre vil metoden samt alle utvidelser som blir omtalt forsøkt samlet i et klassehierarki. Dette gir oss en objektorientert modell av de ulike mekanismene og kan fungere som basis for en standardisert implementasjon av denne familien synkroniseringsmetoder.

1.1 Kort beskrivelse av oppgaven - resultater

Ved bruk av distribuerte systemer er det spesielt innenfor to områder det oppstår problemer. Generelt medfører bruk av slike systemer behov for mange ulike synkroniseringsmekanismer.

Objektene i systemet vil ha behov for å synkronisere seg med hverandre på mange ulike måter, avhengig av applikasjonen som modelleres. Det andre problemområdet er hvordan man skal legge strategier for feilhåndtering i slike systemer.

Denne oppgaven fokuserer på parallellitetskontroll[1] i distribuerte, objektorienterte systemer. Dette er bare en av mange nødvendige synkroniseringsmekanismer, men arbeidet i oppgaven er avgrenset til dette ene tema. Ulike former for feilhåndtering diskuteres ikke.

[1] For ordens skyld så er det engelske uttrykket her **concurrency control**.

Spesifikt vil en se på bruk av *Time Warp*-metoden[17] og varianter/utvidelser av denne. Metoden har sitt opphav fra distribuert simulering der det finnes mye vitenskapelig arbeid rundt metoden. Lite er derimot gjort når det gjelder bruk av denne metoden til parallellitetskontroll. Oppgaven vil gi et studium av hvordan TBS[2] kan benyttes til slik synkronisering i distribuerte, objektorienterte systemer og hvilke fordeler eller ulemper dette innebærer.

Tidsstempelbasert synkronisering baserer seg på kommunikasjon via meldinger mellom objektene i det objektorienterte, distribuerte systemet. Alle meldinger skal merkes med et *tidsstempel* som forteller om tidsoppfattelsen lokalt i avsenderobjektet. Mottakerobjektet benytter så denne tidsverdien til kontroll mot sin egen tidsoppfattelse. Ved en konflikt, der mottakerobjektet ligger foran i tid, vil dette gjøre en *tilbakerulling*. Det vil si at eksekvering vil gjenopptas et sted tilbake i historien til objektet. Hvor denne gjenopptagelsen av eksekveringen starter, avhenger av tidsstempelet i meldingen. Den tidligere sene melding vil nå kunne behandles som om den ankom tidsnok.

Bruk av TBS innenfor parallellitetskontroll virker relativt lovende. Som jeg skal vise treffer man på en litt annen type problemer enn man gjør innenfor distribuert simulering. Objektene vil her være konkurrerende isteden for samarbeidende. Det blir innført ulike former for tilpasninger av metoden, for å tilnærme den mer til parallellitetskontrollproblematikken. Av mekanismer jeg ikke har sett beskrevet i *Time Warp*-modeller i litteraturen kan nevnes innføring av totale ordninger gjennom en utvidelse av tidsstempelformatet.

Jeg forsøker i oppgaven å definere et klassehierarki som samler alle TBS-varianter som er omtalt. Dette kan så danne basis for et generellt grensesnitt for denne familien av synkroniseringsmetoder.

1.2 Motivasjon og problemstillinger

I denne oppgaven skal jeg forsøke å gi svar på om tidsstempelbasert synkronisering tilbyr en effektiv og tilfredsstillende løsningen på synkroniseringsproblemer i distribuerte, objektorienterte systemer, spesielt parallellitetskontroll.

Hvorfor er tidsstempelbasert synkronisering en fristende metode å benytte i slike systemer? Ved første øyekast ser denne metoden meget passende ut fordi den i seg

[2] TBS er en forkortelse for «tidsstempelbasert synkronisering». Dette er ment som en generell betegnelse på metoder som benytter seg av tidsstempler i meldinger som synkroniseringsmekanisme. Som oftest er dette utvidelser av den originale *Time Warp*-modellen[17]. I oppgaven blir TBS konsekvent brukt i denne sammenhengen. Der det spesielt henvises til den opprinnelige modellen benytter jeg det engelske *Time Warp*.

selv er ekstremt distribuert. Den baserer seg på at selvstendige objekter kommuniserer med hverandre ved hjelp av meldinger. Disse objektene er koblet samme *tidsmessig* på en løsest mulig måte. Det finnes typiske synkroniseringspunkter der to eller flere objekter krever dette, men ellers lever objektene uavhengig av hverandre med hensyn på tid og fremdrift. Dette passer godt sammen med situasjoner i distribuerte, objektorienterte systemer fordi man ville få kraftig reduksjon av effektiviteten i disse dersom man skulle kreve at alle objekter var strengt synkronisert med hensyn på fysisk tid.

Slik synkronisering er tradisjonelt sett utført ved hjelp av *lavnivå* metoder. Slik *lavnivå* synkronisering baserer seg som oftest på en eller annen form for låsing av de data som for øyeblikket skal aksesseres. Dette kan for eksempel implementeres ved hjelp av metoder som semaforer eller lignende.

Felles for alle tradisjonelle synkroniseringsmekanismer av denne typen er at de kan resultere i mye unødig venting. Vi kan si at alle disse metodene er pessimistiske av natur fordi de antar den verst tenkelige situasjonen og opererer så deretter. Det vil si at ved kritiske operasjoner der et objekt for eksempel skal skrive til en database vil databasen låses selv om det i ettertid viser seg at konflikter ikke oppsto.

Dersom to objekter i et system ønsker å aksessere et annet dataobjekt samtidig, vil dette føre til at en av dem må vente til det andre er ferdig med sine operasjoner. Dersom man tenker seg et system der man ikke har muligheten til å forutsi om en transaksjon vil føre til endringer i objektet eller ikke, ser vi lett at mye unødig venting vil inntreffe. Dersom to objekter for eksempel forsøker å gjøre en leseoperasjon på en database uten påfølgende skriving, er det helt unødvendig at noen må vente.

I tidsstempelbasert synkronisering, slik det er definert gjennom den originale *Time Warp*-modellen [17] og varianter eller utvidelser av denne, gir man alle objektene i systemet frie tøyler til å gjøre de operasjoner de måtte ønske på en database, selv om disse operasjonene vil kollidere i tidsmessig forstand. Dersom man i ettertid ser at operasjoner førte til inkonsistens i databasen, vil de berørte objektene gjenopprette en gammel tilstand, for så å utføre sine operasjoner igjen. Generelt er det slik at situasjoner der man i tradisjonelle løsninger må kreve låsing, tilsvarer tilbakerullingssituasjoner i *Time Warp*-modeller.

Tidsstempelbasert synkronisering baserer seg på en *optimistisk* hypotese om at ingen operasjoner eller transaksjoner vil kollidere i tidsmessig forstand og dermed kunne gi oss en inkonsistent database som resultat. Problemet blir så å finne en implementasjon av denne metoden som i praksis gir oss en synkroniseringsmekanisme som i hvertfall ikke er merkbart dårligere i effektivitet enn de mer tradisjonelle synkroniseringsmekanismene. Eksemplet med databasen ovenfor indikerer hva som blir vinklingen i denne oppgaven. *Time Warp* er utviklet innenfor distribuert simulering for å synkronisere de samarbeidende objektene som deltar i simuleringen. Som eksemplet viser kan man også tenke seg metoden benyttet til å løse andre typer synkroniseringsproblemer. Hvilke aspekter som kan benyttes slik

de er og hva som må forandres, eventuelt fjernes, blir en viktig diskusjon.

1.2.1 Sentrale spørsmål

Noen av de sentrale spørsmål som melder seg og som søkes besvart i denne oppgaven er følgende:

- Gir tidsstempelbasert parallellitetskontroll av et distribuert, objektorientert system en tilfredsstillende løsning i praksis?

  Det finnes mange eksempler på bruk av tidsstempelbasert synkronisering i teoretiske modeller der det synes som om dette er en meget elegant og tilfredsstillende løsning. Det er faktisk slik at de fleste tidsstempelbaserte løsninger i distribuerte, objektorienterte systemer som finnes i litteraturen er rent teoretiske betrakninger rundt metoden som sådan. En interessant innfallsvinkel mot temaet er å nærme seg metoden fra en praktisk synsvinkel, noe denne oppgaven vil benytte seg av.

  I oppgaven gir jeg også noen eksempler der jeg benytter denne formen for synkronisering på noen klassisk problemer slik som for eksempel lese/skrive-problemet, og som vi skal se er det relativt enkelt å sette opp en teoretisk beskrivelse/modell som gir en god løsning på parallellitetskontrollproblemet.

- Hvordan påvirker en slik løsning effektiviteten i systemet i forhold til en mer tradisjonell løsning?

  Et av de kritiske punktene med hensyn til effektiviteten av et slikt system er hvor ofte eventuelle tilbakerullinger i systemet vil opptre, og hvor langt tilbake i tid dette vil bringe systemet. Det finnes et begrep innenfor problemområdet som skal gi oss et mål på hvor langt systemet har kommet i sin eksekvering, målt i et passende tidsbegrep. Begrepet kalles *GVT* (Global Virtual Time) og er ment å gi oss en nedre grense for det tidspunktet det distribuerte systemet befinner seg ved. Fremdriften i systemet kan så beskrives gjennom dette begrepet. Vanskeligheten ligger i å finne en god og sikker måte å estimere denne tidsgrensen på, og å kunne bevise at denne grensen faktisk beveger seg forover i tid innenfor vårt definerte tidsbegrep.

  Et annet kritisk punkt med hensyn til effektivitet er administrasjonskostnadene som følger med metoden. Disse har sitt opphav fra flere ulike mekanismer og diskuteres der det faller naturlig.

- Kan metoden effektivt benyttes til parallellitetskontroll, og hvilke tilpasninger/endringer må i såfall gjøres ?

  Det finnes mange eksempler på implementasjon av metoden innenfor rammeverket distribuert simulering. Hovedmålet med denne oppgaven er å se

om metoden kan implementeres og benyttes som synkroniseringsmekanisme innenfor rammeverket parallellitetskontroll i distribuerte, objektorienterte systemer. Det vil bli presentert et forslag til hvordan en *standard* for et generelt *Time Warp*-objekt kan defineres. Dette gjøres ved hjelp av et objektorientert rammeverk for implementasjon av distribuerte objektorienterte løsninger.

Der det er naturlig vil jeg ta med resultater fra forskningsarbeid rundt bruk av *Time Warp*-modellen innenfor distribuert simulering. Spesielt dersom jeg føler at disse forslagene til forbedringer eller endringer har relevans for bruk av metodene innenfor parallellitetskontroll, eller har forbedret TBS-metodene genrelt.

- Hvordan er støtten i eksisterende programvare for å benytte seg av en slik form for synkronisering?

  Pr. idag finnes det endel spesifikasjoner av distribuerte, objektorienterte arkitekturer som ville være naturlige verktøy for en slik implementasjon. Alt som i praksis kreves for å kunne implementere parallellitetskontroll basert på *Time Warp* er en veldefinert objektmodell der alle objekter unikt kan identifisere ethvert annet objekt i systemet. Videre må alle objekter kunne sende meldinger til alle andre objekter og motta meldinger fra de samme objektene.

  Som sagt finnes det idag mange ulike systemer som kunne tenkes brukt til en slik implementasjon. To av kandidatene er kort omtalt nedenfor. Det sistnevnte systemet er valgt som basis for en kort diskusjon rundt implementasjon av metoden i et eksisterende verktøy. Motivasjonen for dette valget er ikke spesielt begrunnet i metoden som skal implementeres. Verktøyet har, som vi skal se senere, en tiltalende objektmodell og gode hjelpemidler under implementasjon. Videre er også valget begrunnet i at ANSAware systemet er installert hos Norsk Regnsentral og kunne tas ibruk som testverktøy. Valget av verktøy er altså tatt like mye på praktisk som teoretisk bakgrunn, men hvilket av disse to spesifikasjonene som benyttes til mitt enkel formål er ikke viktig.

  CORBA[3]: Et industri konsortium, OMG[4], arbeider med spesifikasjon av kommersielt tilgjengelige objektorienterte omgivelser. Konsortiet består av flere hundre medlemsorganisasjoner.

  Et resultat av dette arbeidet er CORBA-spesifikasjonen. Dette er en generell beskrivelse av hvordan de ulike bestanddelene i et slikt system skal konstrueres og hvordan interaksjonen mellom de ulike bestanddelene skal foregå. En implementasjon av parallellitetskontroll basert på *Time Warp* vil nok kunne gjøres innenfor CORBA-spesifikasjonen, men siden dette ikke er valgt som tema i denne oppgaven, vil dette ikke bli diskutert ytterligere.

[3] Common Object Request Broker Architechture.
[4] The Object Management Group.

ANSAware: ANSA[5] blir utviklet gjennom ISA[6]-prosjektet som i utgangspunktet bygger på det originale ANSA-prosjektet som startet i England i 1985.

Dette er kort fortalt spesifikasjon og implementasjon av en arkitektur som skal muliggjøre utviklingen av et distribuert, objektorientert system som for brukeren opererer som en enhet. Distribuert programmering og implementasjon av parallellitetskontroll basert på *Time Warp* innenfor dette rammeverket er tema i kapittel 8. Arbeidet i dette kapittelet bygger også på resultater og konklusjoner fra kapittel 7, der det diskuteres et forslag til hvordan en generell standard for en TBS-modell for parallellitetskontroll kan spesifiseres.

[5] Advanced Networked Systems Architechture.
[6] Integrated Systems Architechture.

Kapittel 2

Innføring i sentrale begreper

2.1 Introduksjon av tre sentrale temaer

Følgende tre temaer er sentrale i oppgaven, og jeg gir her en kort beskrivelse av disse på bakgrunn av litteratur/artikler[1] som jeg har benyttet som bakgrunnsstoff. De tre punktene er:

- Synkroniseringsmetoder
- Klokkebegrepet
- Objektorientering

2.1.1 Klokkebegrepet

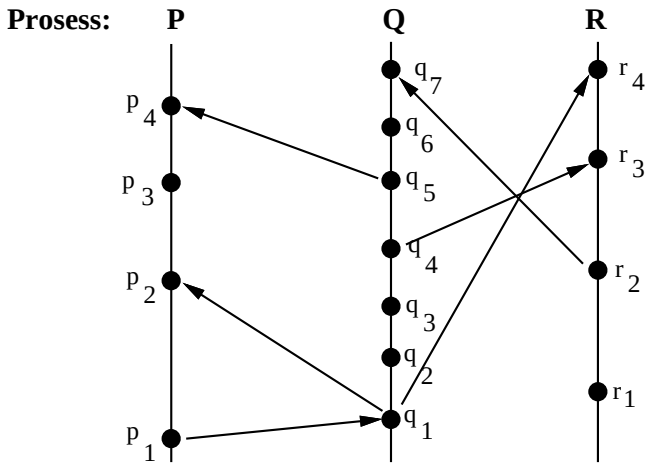
La oss her gi en presentasjon av hvordan Leslie Lamport[22] innfører begrepet logiske klokker i et distribuert system.

2.1.1.1 Distribuerte systemer og partielle ordninger

Vi vil ofte si at en hendelse A skjedde før en hendelse B dersom A inntraff før B i fysisk tid. Dersom en slik definisjon skulle benyttes i et distribuert system måtte dette systemet inneholde *synkroniserte* klokker[2]. Men bruk av datamaskinens fysiske klokker for synkronisering i et distribuert system medfører endel problematikk fordi disse klokkene ikke er nøyaktige nok til enhver tid. Dette medfører at

[1] Se litteraturlista bakerst.

[2] Det vil si at alle klokkene i objektene vil på ethvert tidspunkt under eksekveringen vise samme verdi.



Figur 2.1: Et distribuert system med tre prosesser.

vi må undersøke forsinkelser hos disse klokkene for å kontrollere at dette ikke utgjør noen fare for *korrektheten* i et distribuert system. Dette lar seg gjøre, men for å distansere seg fra problematikken defineres her tidsrelasjonen i et distribuert system *uten* bruk av fysisk tid.

Det første man må gjøre er å gi en definisjon av et distribuert system og en relasjon mellom atomiske[3] operasjoner innenfor dette systemet. La oss tenke oss at et distribuert system består av et vilkårlig antall prosesser som hver er satt sammen av et endelig antall atomiske operasjoner som vi kaller *hendelser*.

Definisjon 2.1.1 *Et distribuert system består av et antall prosesser P_i, der hver prosess $P_i, i \in \{0, \ldots, n\}$ er sammensatt av et antall* hendelser *$p_k, k \in \{0, \ldots, m\}$, der det er definert en* partiell ordning *over hendelsene. Kommunikasjonen mellom de ulike prosessene P_i foregår ved at disse sender meldinger til hverandre. Vi sier at både å sende eller å motta en slik melding, i begge tilfeller utgjør en* hendelse.

Definisjon 2.1.2 *Relasjonen $\rightarrow$ benyttet på en samling av hendelser er den minste relasjonen som tilfredsstiller følgende krav:*

I Dersom a og b er hendelser i den samme prosessen og a kommer før b så gjelder $a \rightarrow b$

II Dersom a er en avsending av en melding m *og b er mottaket av den samme meldingen* m *så gjelder $a \rightarrow b$.*

[3]Uttrykket atomisk benyttes her for å klargjøre at operasjonen kan sees på som *en* operasjon i systemet med hensyn på en passe definert tidsrelasjon. Det vil si at operasjonen er udelelig innenfor den modellen vi jobber i. I en implementasjon kan det derimot være slik at en atomisk operasjon i tidsmessig forstand, vil bestå av mange operasjoner som for eksempel maskininstruksjoner.

III $a \rightarrow b \wedge b \rightarrow c \Rightarrow a \rightarrow c$

Lemma 2.1.1 *Relasjonen* $\rightarrow$ *er en* irrefleksiv, partiell ordning *på hendelser innenfor et distribuert system.*

La oss definere en relasjon som uttrykker «samtidighet» mellom to hendelser i et slik system.

Definisjon 2.1.3 *To forskjellig hendelser* a *og* b *sies å være samtidige, det vil si at relasjonen* $a \leftrightarrow b$ *gjelder, dersom det er slik at* $\neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$.

2.1.2 Synkroniseringsmetoder

Når jeg snakker om synkronisering innenfor et distribuert system er det viktig å være klar over at det med dette ikke menes at prosesser eller objekter skal være synkrone med hensyn på fysisk tid. En synkronisering mellom to hendelser vil som regel være en operasjon som sikrer at den innbyrdes rekkefølgen som to hendelser utføres i, kan sikres gjennom denne operasjonen. Dette betyr jo at de må ha en eller annen form for tidsbegrep som kan benyttes for å overholde en slik betingelse, men dette behøver ikke å inkludere bruk av klokker som viser reell tid.

De ulike metodene for synkronisering som blant annet er nevnt i kapittel 1.2 kan deles inn i to hovedgrupper:

- En samling av *pessimistiske metoder*. Herunder kommer alle de metodene som tradisjonelt er benyttet for synkronisering. Disse metodene baserer seg på en pessimistisk hypotese om at operasjoner/transaksjoner på data eller databaser i utgangspunktet er kritiske og krever at vi ordner disse hendelsene slik at mulige konflikter kan forhindres. Metoder som er av en slik pessimistisk art er for eksempel bruk av semaforer for låsing eller to-fase låsing.
- En samling av *optimistiske metoder*. Disse metodene er i teorien motsatte metoder til de pessimistiske når det gjelder antagelsene om systemet de skal synkronisere. Her antar vi at konfliktsituasjoner aldri vil inntreffe. Dette fungerer som en hovedregel. I de tilfeller denne hypotesen blir brutt, vil vi foreta en oppretting av systemet og for eksempel gjøre alle berørte operasjoner/transaksjoner på nytt.

Det er opplagt at hypotesen som antas i de optimistiske metodene ikke kan garanteres. Faktisk er det slik at dersom denne hypotesen alltid holder så ville det ikke være behov for noen form for synkronisering i det hele tatt. Tidsstempelbasert

synkronisering er en metode som sorterer under den siste av de to ovenfor nevnte. Når man i denne metoden får tilfeller der hypotesen ikke gjelder, vil systemet, eller de berørte objektene, utføre en tilbakerulling av seg selv i tidsmessing forstand. For forklaring av hvordan denne mekanismen implementeres i denne formen for synkronisering, se kapittel 3.1.2.

2.1.2.1 Synkronisering i distribuerte systemer

Synkronisering av et distribuert system vil, uformelt, bety at man ved hjelp av en passende metode sikrer at de ulike hendelsene starter og terminerer i en gitt rekkefølge. Det vil altså si at man foretar en eller annen form for ordning av rekkefølgen til de ulike hendelsene i systemet.

For å kunne sikre en slik rekkefølge av hendelser er man nødt til å spesifisere en binær relasjon som definerer en slik ordning. Denne ordningen kan så danne basis for synkronisering i systemet. Det er så relativt enkelt å innse at dette vil føre til at man må benytte seg av en passende definert *tidsrelasjon* for effektivt å kunne implementere en synkroniseringsmekanisme.

Generelt finnes det to ulike fremgangsmåter for å definere metoder for synkronisering i et distribuert system:

- Sentralisert synkronisering
- Distribuert synkronisering

Den distribuerte synkroniseringen inneholder ofte de samme mekanismer som den sentraliserte, bortsett fra at man der i tillegg distribuerer synkroniseringsmekanismen i seg selv.

Synkronisering kan oppnås gjennom bruk av en felles, global klokke. Dette vil i mange tilfeller bety datamaskinens egen klokke. De ulike prosessene i det distribuerte systemet kan så benytte denne klokken for å utføre sine oppgaver til rett tidspunkt. Dette vil fungere for en objektorientert applikasjon som er lukket i den forstand at den bare eksekverer på en maskin. I det øyeblikk man distribuerer applikasjonen og innfører kommunikasjon mellom applikasjonsobjekter eller kommunikasjon mot andre applikasjoner på andre maskiner oppstår problemer. To fysiske klokker på to ulike maskiner vil sjelden være synkrone nok i en slik sammenheng. Enda mer problematisk blir det når man distribuerer samme applikasjon ut på flere maskiner i et nettverk.

En annen løsning er å dedikere en klokke til hver prosess i det distribuerte systemet slik vi har sett gjennom innføring av logiske klokker. Teoretisk vil dette fungere

utmerket, men man vil i praksis møte endel problemer. Disse kan løses på ulike måter som vi skal se senere.

Et av de problemene man helt umiddelbart vil støte på er som sagt hvordan de ulike klokkene går i forhold til hverandre. Det vil være lite produktivt å synkronisere et system etter klokker som for eksempel ikke beveger seg fremover med samme hastighet. Dette vil være katastrofalt og lede til at man mister kontroll over synkroniseringen. Det man da altså ender opp med ved distribuert synkronisering er at man må ha mekanismer som kontrollerer de ulike klokkene mot hverandre. Dette blir altså synkronisering av de mekanismene (klokkene) som de ulike prosessene igjen benytter for synkronisering seg imellom.

2.1.2.2 Logiske klokker

Vi innfører nå begrepet *logiske klokker*. Vi tar utgangspunkt i relasjonen $\rightarrow$ fra definisjon 2.1.2 samt at vi definerer en klokke Φ_j for hver prosess P_j som en funksjon som tilordner en hendelse p_{j_i} et heltall $\Phi_j(p)$. Det er et viktig poeng at logiske klokker ikke trenger å ha noen relasjon til fysisk tid. Det vi er interessert i er de abstrakte, relative relasjonene mellom hendelsene. Dette er nøyaktig klokkebetingelsen slik den er definert i [22]:

Definisjon 2.1.4 *For alle hendelser a og b så gjelder følgende:*
Dersom $a \rightarrow b$ så $\Phi(a) < \Phi(b)$

Legg her merke til at «samtidigheten» i definisjon 2.1.3 ikke innebærer følgende tankerekke:

$$\begin{array}{c} a \leftrightarrow b \\ \Downarrow \\ \Phi(a) >= \Phi(b) \wedge \Phi(b) >= \Phi(a) \Rightarrow \Phi(a) = \Phi(b) \end{array}$$

Dette ville innebære at to hendelser som er «samtidige» slik begrepet er definert for hendelser i et distribuert system, impliserer at de må inntreffe samtidig også målt i «logisk tid». Men dette er et urimelig krav til et slikt system, og vil heller ikke være tilfelle utfra definisjonene over.

Som vi senere skal se er det i mange tilfeller slik at en partiell ordning av hendelsene ikke er tilstrekkelig når vi skal se på TBS som parallellitetskontrollmekanisme. Vi vil oppnå en bedre løsning dersom vi kan definere en **total ordning** av hendelsene i systemet. For å få til dette benytter man i [22] en ordning av alle prosessene eller objektene i det distribuerte systemet:

Definisjon 2.1.5 *La $\prec$ betegne en tilfeldig **total** ordning av alle prosessene P_i i et distribuert system.*

Vi har nå en relasjon mellom de ulike prosessene i et system som hjelper oss til å definere en total ordning av alle hendelsene:

Definisjon 2.1.6 *Relasjonen $\Rightarrow$ gir en total ordning av alle hendelser i et distribuert system. Anta av vi har en hendelse* x *i prosess P_i og en hendelse* y *i prosess P_j. Da gjelder* x$\Rightarrow$y *hvis og bare hvis* ***enten*** $\Phi(\mathrm{x}) < \Phi(\mathrm{y})$ ***eller*** $\Phi(\mathrm{x}) = \Phi(\mathrm{y})$ *og $P_i \prec P_j$.*

Det skulle være lett å se at $\Rightarrow$ definerer en **total ordning**.

Eksempel 2.1.1 *La oss ta for oss situasjonen i et distribuert system slik det er vist i figur 2.1 på side 18. Anta at r_2 representerer en sending av en melding* m *fra prosess R og at q_7 representerer mottaket av den samme meldingen i prosess Q. Fra den definerte klokkerelasjonen $\rightarrow$ ovenfor vil disse ha en tid som pr. definisjon er slik at $\Phi(r_2) < \Phi(q_7)$. La oss nå si at $\Phi(r_2) = 4$ og at meldingen som sendes inneholder et* tidsstempel T *som i avsendingsøyeblikket settes lik den samme verdien. Vi legger nå et krav på alle prosesser som mottar en melding: Prosessen må sette sin egen klokke til en verdi som er større enn verdien av tidsstempelet og som er større eller lik nåværende verdi.*

Dermed sikrer vi altså gjennom bruk av et tidsstempel *at de to prosessene etter mottaket og dermed også etter avsending, har synkroniserte klokker i den forstand at klokkebetingelsen gitt i definisjon 2.1.4 er oppfylt.*

Denne enkle mekanismen kan så benyttes til synkronisering ved at vi krever at klokkebetingelsen gjelder mellom suksessive hendelser innenfor samme prosess og mellom to hendelser som representerer henholdsvis en meldingssending og et meldingsmottak av den samme meldingen i to ulike prosesser. Det vil si at meldinger kan komme for sent frem, eller en prosess kan ha avansert fortere enn *tillatt* dersom en melding med lavere tidsstempelverdi enn mottakerprosessens logiske klokke leses. I slike tilfeller må det gjøres en form for synkronisering mellom de to berørte prosesser slik at de igjen blir *enige* om tiden.

2.1.3 Objektorientering

Utviklingen innenfor datateknologi går med stor hastighet mot løsninger som baserer seg på utstrakt bruk av nettverk og samspill mellom ulike maskiner/applikasjoner gjennom disse nettverkene. De mulighetene som ligger i disse konseptene gjør at applikasjonene forandrer seg i forhold til sitt tradisjonelle utseende. Her eksekverer en applikasjon vanligvis som en prosess eller dersom det er flere prosesser, så går disse på den samme fysiske maskinen.

Gjennom de mulighetene og de enorme ressurser som tilbys oss som brukere gjennom stadig mer effektive og utbredte nettverk, har et begrep som *objektorientering*

blitt stadig viktigere. Dette konseptet kom først på banen gjennom programmeringsspråkene. Allerede på 1960/70 tallet kom objektorientert tankegang på banen gjennom utviklingen av for eksempel Simula ved Norsk Regnesentral/Institutt for Informatikk ved Universitetet i Oslo. Senere har man fått nyere programmeringsspråk med avansert støtte for objektorientert tankegang, slik som C++, BETA og Smalltalk. Disse konseptene begynner nå også å videreføres til å omfatte hele applikasjoner og programsystemer.

Man tenker seg, enkelt fortalt, at et program eller en applikasjon eksekverer på et ukjent antall maskiner som er knyttet sammen via et nettverk. Applikasjonene består av mange objekter som kan eksekvere på forskjellige maskiner og kommunisere med hverandre over nettverket. De ulike objektene skal sees på som *likeverdige* objekter som kommuniserer med hverandre på samtliges premisser.

I eksisterende applikasjoner og systemer har vi mer et klient/tjener forhold mellom objektene. Av slike systemer kan jeg for eksempel nevne X[4] og X11/Motif[5]. Her har vi et forhold mellom objektene der noen tilbyr en tjeneste som andre kan benytte seg av når det måtte passe. I en fullstendig objektorientert modell bør det være et krav at alle objektene som er en del av den samme applikasjonen eller det samme programmet har et forhold der alle objektene fungerer både som klienter og tjenere overfor hverandre. Noen hevder at en slik fullstendig objektorientering vil være et naturlig skritt videre fra dagens klient/tjener-teknologi (se [19]).

2.1.3.1 Objektorientering i programmeringsspråk

Som tidligere nevnt ble begrepet objektorientering først realisert gjennom implementasjon i ulike programmeringsspråk. Denne typen objektorientering er ofte benyttet for å innføre muligheten til å definere abstrakte datatyper. Disse abstrakte datatypene består som regel av en samlig programvariable og et sett med operasjoner som aksesserer de interne variablene. Man definerer seg en *mal* som beskriver hvordan en slik abstrakt datatype skal se ut, og så kan man lage seg realiserte instanser av typen ved hjelp av for eksempel et *create()*-kall eller liknende. Slike realiserte versjoner av en deklarasjon av en abstrakt datatype kalles for *objekter*. Selve definisjonen av hvordan objektet skal se ut kalles for en *klasse*.

Innføring av disse begrepene gir oss en helt ny måte å tenke på når vi skal programmere en applikasjon. Vi definerer oss egne datatyper og lager regler for hvordan disse skal aksesseres og så videre. En annen egenskap ved objektorienterte programmeringsspråk som er veldig behagelig er funksjonalitet rundt begrep som *subtyper* og *arving*. I teorien rundt programmeringsspråk ser vi ofte begrepene typer og subtyper brukt. Dersom vi for eksempel har to tilgjengelige typer i et språk

[4] X er implementert av Massachusetts Institute of Technology.
[5] X11/Motif is a registered trademark of the Open Software Foundation.

som heter **int** (heltall) og **Nat** (ikke-negative heltall) så er typen Nat en subtype av Int. På en nesten tilsvarende måte kan vi også deklarere oss subklasser i et objektorientert programmeringsspråk. Dersom vi deklarerer oss en klasse har vi muligheten til å benytte denne som en superklasse for en ny klassedeklarasjon. Vi sier at en subklasse arver alle egenskaper som er definert i superklassen. Dette er en underliggende funksjonalitet innenfor klasse/subklasse begrepet. I tillegg er det også slik at en subklasse kan definere sine egne interne variable og operasjoner i tillegg til de den arver fra sin superklasse, og superklassen kan holde noen av sine egenskaper skjult for subklassen.

Alle disse egenskapene summerer kort opp hva som menes med et objektorientert programmeringsspråk. For å formalisere det hele litt kan vi kikke på Peter Wegners definisjon av begrepet *objektorientert* [33] :

Definisjon 2.1.7 *objektorientering = objekter + klasser + arving*

I følge denne definisjonen er objektorientering en fellesbetegnelse på et system med tre ulike egenskaper. Dette er de samme egenskapene vi ser implementert i dagens objektorienterte programmeringsspråk. Objektorienteringen går først og fremst på deklarasjon av dataobjekter og bruken av disse som abstrakte datatyper. Det finnes også eksempler på objekter brukt i en noe videre forstand, slik man for eksempel kan implementere korutiner ved hjelp av klassebegrepet i Simula. Der har objektene en noe videre funksjon enn å opptre som instanser av abstrakte datatyper. Allikevel er det slik at disse objektene er deler av det samme programmet og ikke uten videre kan eksekvere uavhengig av hverandre på forskjellige maskiner eller prosessorer. De er deler av den samme applikasjonen og vil ha tilnærmet lik levetid i den forstand at dersom applikasjonen terminerer, så vil også alle instanser av de deklarerte klassene terminere.

Objektorientering i programmeringsspråk er derfor en typisk metode for å oppnå større grad av modularisering i programkoden. Metoden gir oss nye måter å løse programmeringsproblemer og algoritmedesignproblemer på, men dette er ting som i hovedsak kommer programmereren til gode. Det er faktisk slik at brukeren ikke er i stand til å avgjøre om en applikasjon er programmert ved hjelp av et objektorientert programmeringsspråk eller ikke.

2.1.3.2 Objektorientering i distribuerte systemer

Kan så definisjon 2.1.7 også benyttes når vi snakker om objektorientering i distribuerte systemer ? Vel, tankegangen og de målene vi ønsker å oppnå ved å innføre objektorientering er i allefall tildels sammenfallende enten vi snakker om programmeringsspråk eller distribuerte systemer. Med objektorientering i programmeringsspråk ønsker vi å oppnå en høyere grad av modularisering og strukturering av

programkode enn det som kanskje var mulig, eller ihvertfall praktisk, å få til med tradisjonelle andre og tredjegenerasjons språk. Dette målet er også noe vi ønsker å tilnærme oss med objektorientering i distribuerte systemer. Vi ønsker å splitte opp applikasjoner i selvstendige deler. Disse delene betegnes som objekter og skal kommunisere med hverandre gjennom et strengt definert grensesnitt. Denne oppdelingen gir også en merkbar effekt ut mot brukeren i motsetning til hva som er tilfellet for objektorientering i programmeringsspråk.

Et objektorientert system er dermed et mye videre begrep enn et objektorientert programmeringsspråk. I et objektorientert system kan for eksempel hvert objekt være en egen applikasjon som eksekverer som en egen selvstendig prosess. Disse objektene trenger ikke engang være programmert ved hjelp av et objektorientert programmeringsspråk. Objektene i et distribuert system vil være av en mer generell karakter enn objekter i et språk. Objektene i et system lever i teorien som selvstendige objekter som tilbyr tjenester til andre objekter i systemet. Objektene lever som et *speilbilde* av en definert mal. Dersom denne malen forandres, selv når objektet eksekverer eller lever, så er det mulig at disse forandringene gjenspeiles i de levende objektene. En slik funksjonalitet kan riktignok være problematisk å implementere, men er likefullt ønskelig og tilstede i ulike modeller. Dette er jo ikke generelt tilfellet for objekter i et programmeringsspråk. Disse kan riktig nok forandre *utseende* gjennom at de interne dataverdiene endrer seg, men hvis man først har deklarert et objekt så kan ikke denne deklarasjonen forandres etter oppstart av systemet. Et objektorientert programmeringsspråk er definisjonsmessig egentlig en spesiell instans av det videre begrepet objektorientert system. I praksis kan vi si at et OOPS[6] er valgt for å oppnå generelle mål, men visse mekanismer herfra er i praksis ikke så lurt eller lett i et OODS[7]. For eksempel er arvingsegenskapen fra OOPS et slikt eksempel. I praksis er altså ikke alltid modellen i definisjon 2.1.7 hensiktsmessig for et OODS, selv om den er teoretisk «riktig».

Dette betyr at definisjonen til Wegner ikke er *generell* nok for objektorientering i et distribuert, objektorientert system. I [19] angir forfatterne en definisjon av hvilke elementer som angir graden av objektorientering i et distribuert system:

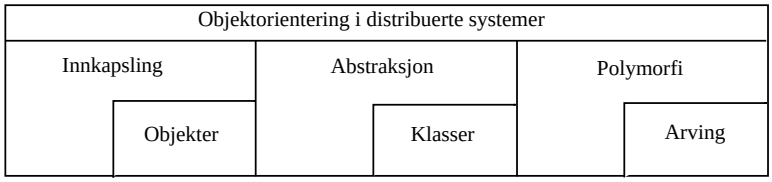
Definisjon 2.1.8 *objektorientering = innkapsling + abstraksjon + polymorfi*

Som de sier er dette en generalisering av Wegners definisjon (se [33]). La oss forklare betydningen av disse begrepene mer detaljert:

- *Innkapsling*. I hvilken grad objektene har kapslet inn og gjemt sine interne datastrukturer og operasjoner for andre objekter. Et objekt med høy grad av

[6] Dette er ment som en forkortelse for «objektorientert programmeringsspråk». Forkortelsen benyttes senere i oppgaven uten videre forklaringer.

[7] En forkortelse for «objektorientert, distribuert system». I likhet med OOPS benyttes også denne forkortelsen videre utover i oppgaven uten videre kommentarer.



Figur 2.2: Objektorientering i programmeringsspråk kontra distribuerte systemer.

innkapsling skal ha et veldefinert grensesnitt som gir aksess til de interne attributtene som verden utenfor objektet skal kunne ha kjennskap til. Disse verdiene skal bare kunne aksesseres gjennom dette grensesnittet og ingen andre attributter internt i objektet skal kunne aksesseres utenfra.

- *Abstraksjon.* Muligheten for å kunne gruppere sammenhørende entiteter med hensyn til felles egenskaper. Vi kan si at en klassedeklarasjon er en abstraksjon av de realiserte instansene av klassen.

- *Polymorfi.* Mulighet for overlapping og arvingsegenskaper. Dette innebærer for eksempel at en operasjon på en gitt abstrakt datatype også er gyldig for alle definerte subtyper. Som eksempel her kan nevnes Simulas klassebegrep der vi har arving av attributter og operasjoner til alle subklasser, dersom dette ikke eksplisitt forhindres i superklassen.

Legg merke til at kravet om et strengt definert grensesnitt inn mot objektet er gitt via *innkapsling*. Skjuling av data og interne ressurser gjøres ved at man definerer nettopp et slikt grensesnitt som interne attributter kan aksesseres gjennom. For et objektorientert, distribuert system kan vi si at samlingen av grensesnittene til alle objektene i systemet definerer systemets *samtaleunivers* eller *kommunikasjonsprotokoll*.

Kommunikasjonen mellom de ulike objektene foregår ved at objektene sender meldinger til hverandre. Via disse meldingene aksesserer et objekt et annet objekts grensesnitt for å utveksle informasjon. Legg merke til at disse meldingene ikke er det samme som en nettverksmelding. En melding som går på *applikasjonsnivå* har ingen logisk sammenheng med de meldingene som faktisk går mellom maskinene på det fysiske nettverket.

I [19] sier også forfatterne at grunnen til at man innfører en ny og *videre* definisjon av begrepet er fordi Wegners definisjon var for spesiell og kunne bare benyttes korrekt for objektorientering i programmeringsspråk. Dersom vi studerer definisjon 2.1.7 og 2.1.8 ser vi at de tre parametrene *objekter*, *klasser* og *arving* er instanser, eller spesialtilfeller av henholdsvis *innkapsling*, *abstraksjon* og *polymorfi*. Dette betyr at objektorientering og dermed spesielt begrepet objekt, har en vagere mening når vi snakker om objektorienterte systemer. I programmeringsspråkene er et objekt bare en realisert instans av en klassedeklarasjon, og forblir det gjennom hele sin

levetid. Det finnes ingen andre muligheter til forandring av et objekt enn å tilordne nye verdier til den interne datastrukturen i objektet. I et objektorientert system derimot, vil man kunne se egenskaper som at et levende, eksekverende objekt kan forandre seg over tid som resultat av læring eller kanskje at objektspesifikasjonen eller deklarasjonen av klassen forandres. Objektene selv kan også utføre handlinger som er utenfor en brukers kontroll som for eksempel å dele seg opp, via *replikasjon*, for å utføre tunge oppgaver. Alle slike egenskaper skal være skjult for brukeren.

Dette er egenskaper vi kjenner igjen dersom vi ser på eksisterende objektorienterte systemer. Tidligere nevnt er CORBA og ANSAware. Begge disse følger definisjon 2.1.8 av et objektorientert system. Den kanskje mest attraktive egenskapen ved objektorienterte systemer er måten man kan skjule interne strukturer og tilby tjenester til andre objekter gjennom et veldefinert grensesnitt. Implementasjonstekniske detaljer som egentlig er andre objekter uvedkomne vil heller ikke være synlig. Dette betyr igjen at et objekt kan identifiseres gjennom sin grensesnittspesifikasjon og brukere av dette grensesnittet trenger, eller skal, ikke kjenne til innholdet i objektene utover dette.

OODS fører imidlertid med seg noen problemer som vi ikke har i OOPS. I et OODS må man implementere ulike typer av synkroniseringsmekanismer mellom objektene, som man ikke trenger i OOPS. Objektene i et OODS har stor grad av selvstendighet og man kan ikke uten videre garantere enhetlig tidsoppfattelse og dermed parallellitet.

Ulike utgaver av et objekt kan være totalt forskjellig implementert uten at en bruker skal merke dette. Så lenge grensesnittspesifikasjonen for objektet er fulgt under implementasjonen er innmaten uinteressant for brukeren. Egenskapene ved et OODS gir oss dermed klasser som er uavhengige av implementasjonsverktøy og maskinarkitekturer. Dersom vi vet typen til en klasseinstans, eller et objekt, kan vi uten videre kommunisere med dette objektet gjennom det predefinerte grensesnittet.

Kapittel 3

Time Warp

3.1 Tidsstempelmodellen (Time Warp)

I [17] beskriver David R. Jefferson hvordan tidsstempelmetoden kan benyttes til å oppnå synkronisering ved hjelp av virtuell tid. Et distribuert system har en global virtuell klokke som tikker virtuell tid og som alltid beveger seg fremover. Den virtuelle tiden i et distribuert system defineres som et en-dimensjonalt koordinatsystem. Dette benyttes for å sikre fremdrift i eksekveringen samt til synkronisering mellom de ulike prosessene i systemet[1].

Vi kan anta at koordinatsystemet består av positive heltall med en uendelig, øvre grense $+\infty$ der koordinatene er strengt ordnet etter relasjonen $<$[2]. En virtuell klokke vil alltid bevege seg forover i tid i den forstand at dersom en hendelse H_a inntreffer før en hendelse H_b, så gjelder $Virtuell\_tid(H_a) < Virtuell\_tid(H_b)$. Dette tilsvarer klokkebetingelsen, definert i definisjon 2.1.4. Dette globale koordinatsystemet representerer en *tenkt* global, virtuell klokke.

Imidlertid vil man i en implementasjon benytte seg av en lokal, virtuell klokke pr. prosess, der de ulike klokkene er *løst* synkronisert med den tenkte globale virtuelle klokken. Det er også slik at de lokale virtuelle klokkene kan bevege seg bakover i virtuell tid fordi *tidsstempelmekanismen* introduserer behov for *tilbakerulling* av eksekveringen i en prosess.

Det distribuerte systemet består av en mengde prosesser $\{P_1, \ldots, P_n\}$ som kommuniserer seg imellom ved hjelp av meldinger. Hver melding skal inneholde et *tidsstempel* som viser den virtuelle tiden til meldingen. Dette kan for eksempel være det virtuelle tidspunktet da meldingen ble sendt eller en egen parameter som

[1] Den originale artiklen snakker om *prosesser*. I dette kapittelet er derfor begrepet forsøkt brukt konsekvent. Merk at innenfor distribuerte systemer snakker vi gjerne om *objekter*. Betydningen av to begrepene er *i denne sammenhengen* den samme.

[2] Dette er den vanlige «mindre enn»-relasjonen slik vi kjenner den mellom heltall.

forteller når avsenderprosessen krever meldingen behandlet.
Hver prosess $P_i, i \in \{1, \ldots, n\}$ inneholder:

- en *innkø* der alle innkomne meldinger blir lagt i kø. Det er et krav til alle prosesser at denne innkøen skal være sortert etter meldingens *tidsstempel*. Dette sorteringskravet sikrer oss at alle meldinger blir behandlet i *riktig* rekkefølge i henhold til virtuell tid.
- en *utkø* der en «negativ» kopi av alle sendte meldinger skal lagres. Også denne køen må være sortert med hensyn på virtuell tid. Her kan man passelig benytte den virtuelle tiden der meldingen ble sendt. Meldingene som lagres i denne køen kalles **antimeldinger**.
- en *tilstandskø* der prosessen lagrer en kopi av alle tilstander den har hatt. Dette betyr en kopi av alle variable eller attributter, som kan forandre seg ved prosessering av en melding.

Anta nå at hvert objekt har en unik identifikator knyttet til seg, for eksempel et navn. Alle atomiske operasjoner utført i en prosess kan da knyttes sammen med et unikt navn[3] og et virtuell tidskoordinat. Anta at en atomisk operasjon finner sted i prosess *k* ved virtuell tid *t*. Vi betegner da denne operasjonen, samt alle sideeffekter av denne som (*k,v*). Vi kan altså unikt identifisere alle atomiske operasjoner[4] i systemet via disse to parametrene.

Hver melding som sendes skal inneholde minst de fire følgende verdier:

- *Avsenderidentifikator.* Dette skal være en identifikator som er unik for avsenderprosessen innenfor det distribuerte systemet.
- *Virtuell avsendertid.* Det virtuelle tidspunktet da meldingen ble sendt.
- *Mottakeridentifikator.* Den unike identifikatoren til mottakerprosessen. Akkurat som for avsenderidentifikatoren, må denne være unik innenfor det distribuerte systemet.
- *Virtuell mottakertid.* Den virtuelle tiden som avsenderen krever meldingen behandlet på.

Et distribuert system som benytter disse metodene for synkronisering må underlegges følgende semantiske regler:

[3] Dette navnet kan også defineres som en koordinat i et koordinatsystem sammen med den virtuelle tiden.

[4] I litteraturen benyttes de to begrepene «atomiske operasjoner» og «hendelser» om hverandre. I denne sammenhengen er begrepene identiske.

I Den virtuelle avsendertiden må være mindre enn den virtuelle mottakertiden innenfor samme melding, for alle meldinger i systemet.

II For alle hendelser $h_i, i \in \{1, \ldots, n\}$ gjelder at dersom h_i blir utført før h_{i+1}, det vil si $h_i \rightarrow h_{i+1}$, så skal $Virtuell\_tid(h_i) < Virtuell\_tid(h_{i+1})$ for alle $i \in \{1, \ldots, n\}$.

Begrepet *tidsstempel* defineres i *Time Warp* til en meldings *virtuelle mottakertid*. Som tidligere nevnt vil det i en implementasjon bli benyttet en lokal virtuell klokke for hver prosess i det distribuerte systemet. Dette er også tilfelle i den originale *Time Warp*-modellen.

De ulike prosessene vil som regel ha ulik oppfattelse av virtuell tid, men siden en prosess ikke kan se en annen prosess sin klokke vil dette ikke skape problemer for lokale operasjoner i prosessen. Den eneste oppfattelsen en enkelt prosess har av andre prosessers lokale virtuelle tid, er verdien på tidsstempelet i de meldingene den mottar. Som vi skal se er det også i slike tilfeller konflikter oppdages.

Det eneste kravet vi må legge på de ulike prosessene i systemet for å forhindre inkonsistens er en enkel regel for hvordan hver prosess skal oppdatere sin egen lokale virtuelle tid. Vi krever at den virtuelle klokken for alle prosesser skal forandre sin verdi *mellom hver hendelse* i prosessen. I tillegg må hver prosess sette verdien av sin egen klokke lik verdien av tidsstemplet i meldingen som ligger først i prosessens *innkø*, idet denne hentes inn og behandles. Tidsstempelverdien er, som nevnt, definert som den virtuelle mottakertiden i meldingen. Dette betyr at den lokale virtuelle tiden i en prosess må være lik denne i det øyeblikket meldingen leses inn og prosesseres. Dette gjøres ved en enkel tilordning. Den lokale virtuelle klokken settes lik tidsstempelverdien i meldingen som skal behandles.

Vi ser raskt at dersom en prosess forsøker å behandle en melding som har tidsstempelverdi lavere enn den lokale virtuelle tiden vil dette kunne gi inkonsistens i systemet. Her oppstår behovet for en synkroniseringsmekanisme. Prosessen skulle ha lest denne meldingen ved et tidligere virtuelt tidspunkt, og mekanismer som rydder opp i slike feil må implementeres. Mekanismen som benyttes i *Time Warp* er tilbakerulling. Enkelt fortalt annulerer prosessen alle sine operasjoner mellom tidsstempelet i den sene meldingen og prosessens lokale virtuelle tid. Deretter gjenopptas eksekveringen med den sene meldingen først i innkøen[5]. Tilbakerulling diskuteres mer i detalj i kapittel 3.1.2, men det er nødvendig å ha kjenskap til mekanismen før man leser neste kapittel.

[5] Meldingen er nå ikke lenger er sent ute, fordi den lokale virtuelle klokken settes ved tilbakerulling lik mottakertiden i den forsinkede meldingen.

3.1.1 Den globale kontrollmekanismen, global virtuell tid (GVT)

De lokale klokkene og tidsstempelmekanismen kan benyttes til synkronisering mellom prosessene. Mange andre viktige oppgaver løser de derimot ikke. Uten en global kontrollmekanisme kan vi ikke bevise progresjon i systemet med hensyn på virtuell tid. I tillegg er en slik global kontrollmekanisme nødvendig for administrasjon av minneforbruket.

Tilstandskøen som inneholder alle tidligere tilstander for prosessen kan forkortes ved innføring av begrepet *Global virtuell tid*, forkortet GVT. Dette representerer en nedre grense for hvor langt en prosess kan gjøre tilbakerulling med hensyn på virtuell tid. Denne verdien vil da også representere et måltall for hvor langt det distribuerte systemet som helhet har kommet i eksekveringen. Det vil derfor ikke være nødvendig å spare tilstander i tilstandskøen som har virtuell tid mindre enn GVT. På samme måte kan vi begrense det antall meldinger og antimeldinger vi trenger å lagre i henholdsvis innkøen og utkøen.

Innføringen av dette begrepet gir oss kontroll over endel problemer som står uløst ved bruk av lokale virtuelle klokker. For eksempel vil det generelt ikke være mulig å vise at det distribuerte systemet som helhet beveger seg fremover i tid, og dermed vil kunne avslutte eksekveringen, uten en slik global kontrollmekanisme. La meg definere begrepet GVT. Dette er identisk med Jeffersons definisjon i [17]:

Definisjon 3.1.1 *GVT ved et gitt tidspunkt r (reell tid) er den minimale verdien av:*

1. *alle virtuelle tider i alle prosesser ved r.*
2. *den virtuelle avsendertiden i alle meldinger som er sendt, men ennå ikke er prosessert ved r.*

Det er viktig at man gir en definisjon av GVT som er slik at ingen prosesser kan tenkes å forsøke en tilbakerulling av systemet til en tid $V_{t_i} < GVT$. Definisjonen over vil sikre dette gjennom krav 2 fordi tilbakerullinger nettopp inntreffer ved mottak av forsinkede meldinger.

Et negativt aspekt ved å måtte vedlikeholde[6] denne definisjonen er det globale aspektet som innføres i en ellers så distribuert modell. I praksis er det tilnærmet umulig å finne den teoretisk korrekte verdien av GVT. Dette måtte bety at hele systemet måtte stoppe opp og alle meldinger på vei til sin mottaker måtte lokaliseres og inspiseres. Heldigvis er det slik at det rekker med en mer operasjonell definisjon av GVT. Dette er egentlig bare et estimat, men vil allikevel gjøre nytten.

[6] Les regne ut. Man må ha en algoritme av global karakter for å kunne regne ut GVT etter den teoretiske definisjonen.

Definisjon 3.1.2 *Operasjonell GVT, forkortet $\mathcal{OGVT}$, er et estimat av GVT som alltid oppfyller egenskapen $\mathcal{OGVT} \leq \mathcal{GVT}$.*

Det finnes algoritmer som beregner en slik $\mathcal{OGVT}$. Disse er relativt raske og enkle. Den originale algoritmen som nevnes i [17] vil eksekveres på **O(d)** tid, der d er forsinkelsen som inntreffer ved å sende en *broadcast* til alle prosesser i systemet. Den originale algoritmen for å regne ut en slik operasjonell GVT finnes i [30, 8]. Algoritmen beskrives i kapittel 3.3.2 og en skisse av koden er gjengitt i tillegg A.1.1.

I ulike modeller skal vi senere se at det kan være fornuftig å forandre algoritmen som estimerer GVT slik at den passer bedre sammen med det konkrete eksemplets natur. Merk at definisjon 3.1.1 av teoretisk GVT fortsatt gjelder for *Time Warp*. Det vil fortsatt være slik at GVT skal være et ytre mål på hvor langt det distribuerte systemet har kommet i sin eksekvering. Denne verdien vil også alltid representere en nedre grense for hvor langt systemet kan gjøre tilbakerulling. I praksis vil dette bety at all informasjon som vedrører prosesshistorien[7] kan fjernes og minne frigjøres dersom denne informasjonen gjelder for tilstander som er eldre enn estimatet av GVT. I og med at definisjonen av GVT kan forandres noe fra implementasjon til implementasjon[8] er det slik at en eventuell algoritme som skal estimere GVT vil variere i tråd med denne definisjonen.

I og med at ingen prosesser kan rulle tilbake under denne grensen, vil også GVT være det siste virtuelle tidspunktet hvor det distribuerte systemet garantert er konsistent. Vi kan i praksis ikke stole på noe informasjon som ligger i systemet, eller som vi henter ut, dersom denne informasjonen er laget på basis av data som er nyere enn GVT. Vi risikerer da at en eller flere prosesser gjør en tilbakerulling. Når prosessen/prosessene vi hentet våre data fra igjen befinner seg ved samme virtuelle tidspunkt, vil disse verdiene kanskje ha en helt annen verdi. Faktisk kan en si at det er sannsynlig at de har en annen verdi fordi tilbakerullinger gjøres dersom en prosess har gått glipp av en melding. Den har altså mistet informasjon på veien. Spesielt vil dette gjelde dersom våre data stammer fra en prosess som ligger langt fremme i virtuell tid i forhold til resten av systemet.

3.1.1.1 Administrasjon av minne - fossilfjerning

Bruk av begrepet GVT gir oss i tillegg muligheten for å administrere minneforbruket i systemet. Vi vet nå at ingen prosesser kan gjøre en tilbakerulling i tid slik at gamle, køede meldinger med mottakertid mindre enn GVT blir benyttet eller sendt ut i systemet. Dette gjelder både for vanlig meldinger i innkøen, antimeldinger i utkøen samt tilstander i tilstandskøen med virtuell tid mindre enn GVT. Følgelig kan

[7] Denne informasjonen skal være lagret i prosessens tilstandskø.

[8] Snakker da mer generelt om TBS. *Time Warp*-metoden spesielt har en gang for alle definert GVT innenfor sitt rammeverk.

disse fjernes fra sine respektive køer. Mekanismen kalles i [17] og all senere litteratur jeg har sett for **fossil collection**. Som oversettelse av begrepet videre i oppgaven benyttes *fossilfjerning*. La meg definere hva som menes med en fossil melding:

Definisjon 3.1.3 *En fossil melding er en melding $\oplus m_i$ som er køet i fortidsdelen av en prosess sin innkø der mottakertiden t_m er slik at $t_m < GVT$.*

En fossil antimelding er en melding $\ominus m_i$ som er køet i en prosess sin utkø der mottakertiden t_m er slik at $t_m < GVT$.

Slik fjerning av fossile meldinger kan iverksettes umiddelbart etter en GVT-algoritme har funnet et nytt estimat. Hvor ofte fossilfjerning skal gjennomføres er selvfølgelig helt avhengig av faktorer som hyppigheten til eksekvering av GVT-algoritmen samt antall meldinger som sendes kontra tilgjengelig minne. Fossilfjerning kan bare gjøres en gang pr. prosess mellom hver gang GVT-algoritmen har eksekvert. Dette er selvfølgelig fordi to fossilfjerninger i samme prosess med samme GVT-estimat ikke ville gi noen effekt den andre gangen.

I praksis kan det bli slik at man tvinges til å øke hyppigheten av GVT-estimering for å kunne frigjøre minne, iallefall i systemer hvor det sendes store mengder meldinger. Minnetilgjengelighet kan altså påvirke effektiviteten i et slikt system i negativ retning dersom den er lav.

3.1.2 Tilbakerulling og antimeldinger

La oss nå ta opp tråden fra kapittel 2.1.2.2 og 3.1 og se på hva begrepet tilbakerulling[9] innebærer. Se på følgende situasjon i et tenkt objektorientert, distribuert system med et passende antall prosesser $\{P_1, \ldots, P_N\}$ for en fast $N > 1$: En prosess $P_j, j \in \{1, \ldots, N\}$ mottar en melding med tidsstempelverdi t_m. Anta nå at prosessens egen virtuelle klokke viser tiden $V_{P_j} = v$ der $t_m < v$. Meldingen som har kommet inn skulle ha vært prosessert på et tidligere virtuelt tidspunkt i eksekveringen. Det er derfor nødvendig å forta en tilbakerulling i prosess P_j.

Dette gjøres ved at prosess P_j sender ut alle antimeldinger den har lagret i sin *utkø* som har avsenderverdi større enn t_m. Alle slike antimeldinger skal sendes til den prosessen som opprinnelig mottok den tilsvarende meldingen. Dersom det eksisterer en melding og en tilsvarende antimelding i den samme køen skal disse umiddelbart oppheve hverandre, det vil si at begge forsvinner fra køen. Dersom vi angir en melding med $\oplus m_j$ og den tilsvarende antimeldingen som $\ominus m_k$ kan vi sette opp følgende teorem om forholdet melom de to meldingstypene i systemet:

[9] Dette er en direkte oversettelse av uttrykket **rollback** i den engelske litteraturen.

Teorem 3.1.1 *$\mathcal{M}_t = \sum_j \oplus m_j + \sum_k \ominus m_k = 0$ der $\mathcal{M}_t$ angir den totale meldingssummen for hele det distribuerte systemet ved $GVT = t$.*

Før jeg setter opp beviset trenger jeg å påpeke antagelsen fra tidligere om at meldingsutvekslingen er sikker, det vil si at ingen meldinger går tapt.

Bevis 3.1.1 *Beviset gjøres ved at man kikker på de tilfeller der meldinger lages eller ødelegges. Vi har tre ulike tilfeller å kontrollere:*

- *En prosess lager en melding $\oplus m$ for utsendelse til en annen prosess.*

  Vi vet pr. definisjon at dette innebærer at en negativ kopi $\ominus m$ lages samtidig og legges inn i denne prosessens utkø. Vi har da fortsatt at $\mathcal{M}_t = 0$. □

- *En antimelding sendes ut fra en prosess grunnet tilbakerulling. Vi vet at denne meldingen vil komme frem til mottakerprosessen innen endelig tid. Der vil den forsvinne sammen, og samtidig, med sin tilsvarende* positive *melding. Dette viser at vi fortsatt har situasjonen $\mathcal{M}_t = 0$.* □

- *Meldinger forsvinner fra system ved **fossilfjerning**. Anta at* Time Warp *systemet starter **fossilfjerning** for alle meldinger i systemet som har et tidsstempel lavere enn en gitt t. Vi vet da at alle positive meldinger i systemet har en tilsvarende negativ melding liggende i avsenderprosessens utkø. Men denne meldingen vil også bli gjenstand for fjerning, siden den har samme tidsstempelverdi som sin tilsvarende positive melding. Hele system vil under tiden det tar å gjøre en slik opprydding stå stille med hensyn på virtuell tid i alle prosesser. Dermed sikrer vi oss at $\mathcal{M}_t = 0$, for alle t.* □

Det å annulere en tidligere sendt melding, består altså bare av å sende den tilsvarende antimeldingen. Dersom en prosess mottar en antimelding køes denne på vanlig måte i prosessens innkø. Dersom den tilsvarende positive meldingen er køet, men ennå ikke behandlet, vil de to meldingene *oppheve* hverandre og forsvinne fra systemet. Dersom den tilsvarende positive meldingen allerede er behandlet vil antimeldingen føre til en tilbakerulling i prosessen fordi den har lavere tidsstempelverdi enn prosessens lokale virtuelle tid. Tidsstempelverdien i antimeldingen er jo nettopp lik tidsstempelverdien i den tilsvarende positive meldingen. Etter at tilbakerullingen er utført vil nå både den negative og den positive meldingen befinne seg i innkøen som ubehandlede meldinger og forsvinne. Den siste muligheten er at den negative meldingen ankommer før den tilsvarende positive. I såfall skal den negative meldingen bare køes som vanlig. Vi kan også tillate at den negative meldingen blir prosessert av prosessen. Dette kan da for eksempel gjøre en nulloperasjon. Resten blir analogt til om den positive meldingen hadde kommet frem først.

3.2 Aggressiv kontra lat kansellering

Tilbakerullingsmekanismen slik den originalt er foreslått innebærer såkalt *aggressiv kansellering*[10]. Metoden er aggressiv i den forstand at ved mottak av en melding som er for sent ute, vil den automatisk kansellere alle meldinger den selv har sendt, samt kansellere effekten av alle mottatte meldinger med høyere tidsstempelverdi enn den forsinkede meldingen.[11]

Definisjon 3.2.1 *I et* Time Warp *system som benytter seg av* ***aggressiv kansellering*** *vil alle antimeldinger med tidsstempelverdi høyere enn en forsinket melding øyeblikkelig tas ut av køen og sendes. Alle «utregninger» datert etter den forsinkede meldingen sitt tidsstempel vil bli ignorert. Det vil si at man gjenoppretter tilstanden fra virtuelt tidspunkt rett før mottakertiden i den forsinkede meldingen. Dette gjøres via objektets tilstandskø. Deretter restartes eksekveringen i objektet med den forsinkede meldingen som første melding i innkøen hos objektet.*

Ved bruk av denne fremgangsmåten er det lett å se at mange operasjoner som er utført vil bli kansellert, selv om de ikke er berørt av den sene ankomsten til en melding. En kan lett tenke seg at dette i mange tilfeller fører til unødig mye tilbakerulling og at man utfører de samme operasjonene to ganger, med nøyaktig samme resultat.

For å effektivisere metoden har det kommet forslag til en annen kanselleringsmetode, kalt *lat kansellering*[12] [23]. Denne metoden er langt «snillere» enn den originale metoden fordi den tar hensyn til hvorvidt operasjoner/meldingsavsendelser som i den originale metoden automatisk ville bli kansellert, er feil eller ikke. Dersom de er korrekte vil de bli beholdt og man slipper unødig tilbakerulling i andre objekter som en følge av unødvendig utsendelse av antimeldinger. La oss definere uttrykket:

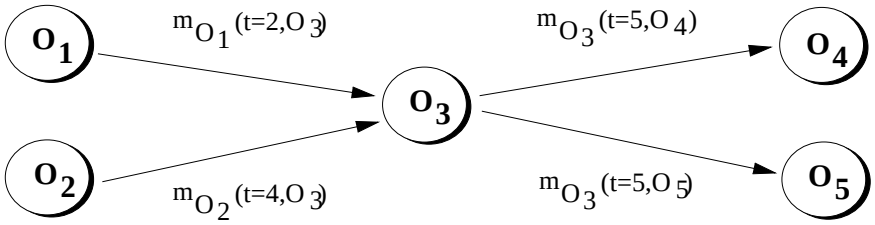
Definisjon 3.2.2 *I et* Time Warp*-system som benytter seg av* ***lat kansellering*** *vil man gjenoppta eksekveringen i objektet med den nye, forsinkede meldingen først i objektet innkø. De «gamle» operasjoner/antimeldinger vil ikke bli fjernet. En kopi av disse beholdes inntil objektet når samme tid som det hadde før den forsinkede meldingen ankom. De nye meldingene som genereres, sammenlignes så med de gamle versjonene i utkøens kopi. Dersom de er ulike sendes antimeldingen etterfulgt av den nye meldingen. Ved likhet kan man la de to nye meldingene oppheve hverandre uten at noen av dem køes eller sendes.*

Denne metoden vil, i alle fall i store systemer med mange objekter, kunne gi oss en effektivitetsøkning. Den ekstra administrasjonen som trengs for å sammenligne et

[10] «Aggressive cancellation» i den engelske litteraturen.

[11] Det vil si at innkøpekeren blir flyttet. Innkøen forblir den samme, men objektets oppfattelse av hvor langt det er kommet i traverseringen forandres.

[12] Ordlyden på engelsk er «lazy cancellation».



Figur 3.1: *Time Warp* modell med ulike kanselleringsstrategier

objekts kopi av tidligere utkø samt den nye versjonen av den samme køen vil være mindre enn den administrasjon som må gjøres ved alle tilbakerullingene som utsendelse av antimeldinger medfører. For å kunne hevde dette må man selvfølgelig kunne argumentere for at utsendelse av antimeldinger kan forhindres ofte nok til at det oppveier den ekstra administrasjonen som kreves.

La meg illustrere ulikhetene i funksjonaliteten mellom aggressiv og lat kansellering med et eksempel:

Eksempel 3.2.1 *Anta at vi har et system slik det er vist i figur 3.1. Anta så først at vi benytter oss av aggressiv kansellering.* O_3 *mottar* $m_{O_2}(t = 4, O_3)$ *og sender ut to meldinger* $m_{O_3}(t = 5, O_4)$ *og* $m_{O_3}(t = 5, O_5)$ *til henholdsvis* O_4 *og* O_5*. Anta så at* $m_{O_1}(t = 2, O_3)$ *ankommer* O_3*. Nå vil* O_3 *måtte rulle tilbake til* $t = 2$ *og sende ut antimeldinger for henholdsvis* $m_{O_3}(t = 5, O_4)$ *og* $m_{O_3}(t = 5, O_5)$ *slik at effekten for disse kan kanselleres hos mottaker objektene. Dette vil sannsynligvis forårsake en tilbakerulling i* O_4 *og/eller* O_5*. Anta så at* O_3 *gjenopptar eksekveringen, nå med* $m_{O_1}(t = 2, O_3)$ *først i innkøen. Etter prosessering av denne og etterpå* $m_{O_2}(t = 4, O_3)$ *sendes meldingene* $m'_{O_3}(t = 5, O_4)$ *og* $m'_{O_3}(t = 5, O_5)$*. Anta nå at behandlingen av* $m_{O_1}(t = 2, O_3)$ *ikke påvirker utsendelsen av de to meldingene sendt til* O_4 *og* O_5*, det vil si at* $m_{O_3}(t = 5, O_4) = m'_{O_3}(t = 5, O_4)$ *og at* $m_{O_3}(t = 5, O_5) = m'_{O_3}(t = 5, O_5)$ *og dermed var utsendelsen av de to antimeldingene som en følge av tilbakerullingen unødvendige i den forstand at de ikke påvirket sluttresultatet av eksekveringen.*

Slike «unødvendige» utsendelser av antimeldinger ved tilbakerulling blir forhindret ved bruk av ***lat kansellering****. Her ville man bare rullet tilbake* O_3 *og gjenopptatt eksekveringen med den forsinkede meldingen* $m_{O_1}(t = 2, O_3)$ *først i innkøen uten å sende ut antimeldinger for tidligere sendte meldinger* $m_{O_3}(t = 5, O_4)$ *og* $m_{O_3}(t = 5, O_5)$*. Når* O_3 *når* $t = 5$ *vil man så sammenligne antimeldinger fra første og andre utsendelse av meldinger til henholdsvis* O_4 *og* O_5*. Dersom de to er like i hvert tilfelle vil de nye antimeldingene forbli i køen og ikke sendt.*

Tilslutt vil disse forsvinne ved ***fossilfjerning*** *og eksekveringen vil ende opp med korrekt resultat.*

Et distribuert, objektorientert system som benytter *Time Warp* til synkronisering vil gjøre unødig arbeid gjennom unødig spredning av tilbakerullinger dersom den

originale metoden med aggressiv kansellering benyttes. Hvor stor en eventuell effektivitetsøkning vil være ved overgang til lat kansellering er vanskelig å uttale seg om på generell basis. Blant annet vil dette åpenbart avhenge av hyppigheten av tilbakerulling under eksekveringer i systemet. Dersom denne hyppigheten ikke er stor, er det ikke sikkert man veier opp kostnadene av noen unødvendige tilbakerullinger med den ekstra «overhead» som lat kansellering vil innebære. I [23] sies det at lat kansellering gir gode resultater ved lav «load» hos objektene som inngår i eksekveringen, men ingen generelle konklusjoner om hva som er best trekkes. Det kan altså synes som om hvilken løsning som bør velges skal overlates til hver enkelt implementasjon.

Merk at det vil være korrekt å la hvert enkelt objekt i en implementasjon bestemme sin egen kanselleringsstrategi. Dette er et interessant poeng fordi man da kan velge kanselleringsstrategi dynamisk innenfor hvert objekt i forhold til hva som gir høyest effektivitet. Empiriske tester kan indikere hvilken strategi som fungerer best for den spesielle applikasjonen som implementeres.

Dette er derimot ikke tema for dette kapittelet, men det vil bli diskutert mer i detalj i kapittel 7.

3.3 Global Virtuell Tid - Algoritmer

I [17] gis ingen beskrivelse av en algoritme som estimerer GVT. Det finnes derimot andre som har beskrevet, og implementert ulike varianter av slike. Noen av disse blir kort omtalt nedenfor. La oss først gjøre en presisering. Det er sjelden slik at det er nødvendig eller forsvarlig å regne ut GVT i henhold til definisjon 3.1.1. Dette ville kreve at man hadde muligheten for å beregne GVT for et gitt tidspunkt i reell tid, og dette er selvfølgelig ingen enkel oppgave. Algoritmene som er beskrevet er alle slik at de følger definisjon 3.1.2. I de implementerte algoritmene nøyer man seg altså med å benytte et estimat av GVT. I [8] beskrives noen algoritmer som kan benyttes for å beregne $\mathcal{OGVT}$ og disse beskrives nedenfor..

3.3.1 Definisjon av begreper brukt i GVT-estimatalgoritmer

La oss først definere noen begreper som benyttes for å beskrive GVT-algoritmene som blir omtalt:

Definisjon 3.3.1 *Et* Time Warp *system består av N noder som er nummerert fra* 0 *til* N-1*. Antall objekter i systemet er O der $O \geq N$ og hver node $N_i, i \in \{0 \ldots N-1\}$ har minst en O_{i_j}.*

Denne definisjonen er i praksis bare en formalisering av ting som er sagt før, men gir oss et fast rammeverk for beskrivelse av GVT-algoritmene. Vi trenger så en form for relasjon mellom nodene i systemet der vi kan være sikre på at alle nodene har minst ett felles punkt med hensyn på reell tid.

Definisjon 3.3.2 *For hver node i la [*START$_i$,STOPP$_i$*] være et intervall i reell tid. Disse intervallene må være slik at vi har minst ett punkt som er felles for alle intervallene. Dette punktet (eller punktene) kalles* RTM*, reelltid moment.*

Merk at disse intervallene må settes på bakgrunn av informasjon som må hentes ut av systemet selv. Dette kan for eksempel gjøres ved å kringkaste meldinger til alle objekter[13] om at man ønsker å starte en beregning av GVT. Reell tid ved mottak av en slik melding kan så for eksempel settes lik $START_i$. Som vi skal se er dette metoden som er valgt i den «gamle» GVT-estimatalgoritmen.

Vi definerer så noen begreper som benyttes for å regne ut mellomresultater før det endelige $\mathcal{OGVT}$ estimatet kan gjøres:

Definisjon 3.3.3 *Hver node i kan regne ut* MVT$_i$ *som er minimum av alle tidsstempel i meldinger som enten er på vei fra node i ved reell tid* START$_i$ *eller sendes innenfor intervallet [*START$_i$, STOPP$_i$*].*

Definisjon 3.3.4 *Hos hver node i sier vi at* PVT$_i$ *er den virtuelle klokkeverdien hos det objektet som ligger lengst etter i eksekveringen ved reell tid* STOPP$_i$*.*

Definisjon 3.3.5 LVT$_i$ *for node i er minimum av nodens* MVT$_i$ *og* PVT$_i$*.*

Alle begrepene hittill beveger seg nede på **nodenivå**. Men $\mathcal{OGVT}$ esimatet kan nå enkelt defineres:

Definisjon 3.3.6 *Estimatet av GVT, $\mathcal{OGVT}$, er minimum av alle* LVT$_i$*.*

Vi har nå definert de begreper vi trenger for å se på noen av de ulike algortimene for GVT-estimering som er beskrevet i litteraturen.

[13] Eller gjøre en «broadcast» dersom man er mer fortrolig med den engelske terminologien. Heretter benyttes den norske oversettelsen konsekvent.

3.3.2 Den gamle GVT algoritmen

Dette er algoritmen som det henvises til i den originale artikkelen om *Time Warp* ([17]). Algoritmen benytter seg av *kringkasting* for å definere det tidligere omtalte intervallet og kan beskrives ved hjelp av følgende fem trinn:

1. Node 0 sender en *kringkasting* til alle andre noder i systemet. Meldingen som sendes er en spesialmelding og er av typen *GVT-start*. For node i settes så $START_i$ lik reell tid ved mottak av kringkastingsmeldingen.
2. Deretter sender alle noder en kvitteringsmelding tilbake til node 0. Når alle slike kvitteringsmeldinger har ankommet node 0 er vi ved tid *RTM*.
3. Node 0 sender en ny *kringkasting* ut i systemet. Disse meldingene er av typen *GVT-stopp*. For node i settes så $STOPP_i$ lik reell tid ved mottak av *GVT-stopp* meldingen.
4. Node i beregner så LVT_i etter definisjon 3.3.5, og sender denne tilbake til node 0. Node 0 beregner så minimum av alle LVT_i meldinger når alle har kommet frem. Dette minimum er i følge definisjon 3.3.6 den nye $\mathcal{OGVT}$.
5. Node 0 sender så en ny *kringkasting* av typen *GVT-oppgrader*. Meldingen inneholder det nye estimatet av GVT.

Ved siste punkt i denne algoritmen kan så alle noder N gjøre en opprydding dersom dette er implementert eller ønskelig. Med opprydding menes her for eksempel *fossilfjerning* i henhold til den nye estimerte GVT. Denne algoritmen eksekverer, som nevnt i [17], på tid i $O(d)$, der d angir tiden det tar å gjøre en *kringkasting*. En skisse av programkoden for algoritmen er gjengitt i tillegg A.1.1.

3.3.3 GVT-algoritme med *message routing graph*

Et nytt forslag til algoritme for å regne ut en $\mathcal{OGVT}$ benytter seg av en såkalt «Message Routing Graph». Algoritmen kan beskrives i tre trinn:

1. Node 0 sender ut en *GVT-start* melding til maksimum to andre noder. Hver node mottar så maksimum to *GVT-start* meldinger. Etter at disse er mottatt sender så noden en *GVT-start* melding til maksimum to noder og så videre. Hver noden skal etter at den har mottatt alle sine *GVT-start* meldinger, sette $START_i$ lik reell tid i noden ved meldingsmottaket. Merk at alle meldingene sendes i en rettet graf der det ikke finnes sykler.

2. Den siste noden som mottar sine *GVT-start* meldinger setter så $STOPP_i$ til reell tid ved mottak av siste *GVT-start* melding. Denne reelle tiden vil også tilsvare *RTM* fra definisjon 3.3.2. Node $N-1$ beregner så LVT_{N-1} i henhold til definisjon 3.3.5, og sender dette estimatet til maksimum to andre noder. Merk at vi nå i praksis opererer med en rettet graf som er den inverse av den opprinnelige med hensyn til retningen på kantene. Hver node,unntatt $N-1$, vil motta maksimum to GVT_{lvt} meldinger. Ved mottak av den siste settes $STOPP_i$ lik reell tid for noden. Node i minimerer så alle estimater av LVT_i den har, inkludert sitt eget, og prosessen fortsetter. Når node 0 har mottatt alle LVT_i meldinger har denne et nytt estimat og dermed en ny verdi for $\mathcal{OGVT}$.

3. Den nye verdien for $\mathcal{OGVT}$ kan så distribueres til alle andre noder fra node 0 ved å benytte grafen.

Denne nye algoritmen er i teorien raskere enn den gamle. Denne algoritmen vil eksekvere på en tid av $O(\log N)$, der N er antall noder i systemet. Metoden krever noe administrasjon en gang pr. node for å konstruere grafen som trengs. Dette arbeidet krever eksekveringstid av $O(\log N)$ pr. node i det distribuerte systemet. Hvordan denne grafen bygges opp tas ikke med her, men for spesielt interesserte er dette dokumentert i [8]. En skisse av programkoden for algoritmen er også gjengitt i tillegg A.1.2. Videre er det slik at for hver node må man traversere meldingskøer og så videre, men dette gjøres enkelt i tid av $O(1)$. Algoritmen vil garantert generere mindre enn $4N$ meldinger før $\mathcal{OGVT}$ er ferdig beregnet.

3.3.4 En annen GVT-estimatalgoritme

Visse andre varianter av algoritmer som estimerer GVT er også presentert. Den som nevnes i den omtalte artikkel[8] er en som benytter seg av **token-passing** mellom nodene i det distribuerte systemet for å sende de nødvendige meldingene. Denne eksekverer med «runtime» av $O(1)$ på hver node og av $O(N)$ globalt sett, der N er antall noder. Algoritmen genererer konstant $3N$ meldinger for hver gang $\mathcal{OGVT}$ skal beregnes.

Denne algoritmen omtales ikke videre i denne oppgaven.

3.3.5 Den gamle kontra den nye GVT-estimatalgoritmen

Jeg har valgt å benytte de to første algoritmene jeg har beskrevet som grunnlag for noen kommentarer fordi disse er begge implementert som GVT-estimatalgoritmer i TWOS[14] (se [8]) og de er vurdert opp mot hverandre i noen ytelsestester utført av forfatterne.

[14] Time Warp Operating System

De endelige testresultatene viser ganske tydelig av den nye algoritmen for estimering av GVT er klart raskere enn den gamle. Den nye algoritmen viste seg å være raskere enn den gamle i de tilfeller som ble sjekket. Dette er kanskje ikke så veldig overraskende siden vi allerede i teorien ser at den nye eksekverer i $O(\log N)$ og den gamle i $O(d)$. Administrativ traversering inne i hver enkelt node, som fører frem til LVT_i for noden, vil være den samme for begge algortimene. Dette fullføres i begge tilfeller i tid av $O(1)$.

Dette gjelder for høy belastning såvel som lav. Antall noder hadde heller ikke noen innvirkning på hvem av de to som fremsto som vinner. Den originale algoritmen slik den var henvist i [17] er altså foreldet i den forstand at bedre alternativer finnes. Skifte av denne algoritmen vil selvfølgelig ikke påvirke modellen sin virkemåte på andre måter enn ytelse/eksekveringstid.

Den nye GVT-algoritmen vil gi noe mer implementasjonsarbeid fordi man må konstruere mekanismer som kan benyttes for å bygge opp grafen som benyttes. Det er gitt en forslag til kode i artikkel [8], og denne er gjengitt i tillegg A.1.2. Slik denne er satt opp antar man et fast antall noder, men det vil ikke by på store problemer å implementere dette for systemer der antall noder kan vokse eller synke. Se kapittel 7 for mer diskusjon rundt dette tema.

3.3.6 Tillegg ved implementasjon

Algoritmene gir metoder for hvordan GVT skal estimeres globalt sett mellom de ulike nodene i systemet. Noder er i denne sammenhengen ekvivalent med maskiner. Lite eller ingenting sies om hvordan beregningen av *LVT* foregår ute hos nodene. For enkle systemer med få objekter pr. node vil det være enkelt å gjøre denne beregningen.

Nye verktøy for applikasjonsmodellering via OODS tilbyr en relativt komplisert struktur av objekter. I kapittel 8 beskrives ANSA-modellen med implementasjonen ANSAware. Her innføres et nytt nivå mellom node og objekt. En node kan ha mange **kapsler**, som igjen kan ha mange objekter. Med node, kapsel og objekt menes her henholdsvis maskin, prosess og objekt.

I en slik struktur er beregningen av *LVT* pr. node ikke lenger triviell. I kapittel 6.3.1 presenterer jeg δGVT-algoritmen som er konstruert nettopp for dette formålet.

Kapittel 4

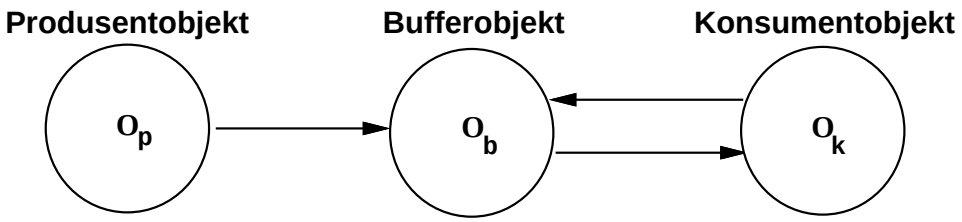
Tidsstempelbasert synkronisering - eksempler

I dette kapittelet ser jeg på to klassiske synkroniseringseksempler. Begge er ofte brukt tidligere for å illustrere teknikker som bruk av semaforer, monitorer og liknende. Her presenteres en variant av produsent/konsument-problemet og en av lese/skrive-problemet. Hvert av eksemplene brukes til å illustrere viktige poeng innenfor TBS og/eller nye aspekter ved bruk av TBS til parallellitetskontroll.

4.1 Produsent/konsument-problemet

En av de problemene som stadig vender tilbake i litteraturen om parallelle systemer og synkroniserings-teknikker er produsent/konsument-problemet. Motivasjonen for dette eksemplet er å vise hvordan TBS kan benyttes for å løse et av de klassiske synkroniseringsprobleme. Fra kapittel 3.1 husker vi at GVT-begrepet benyttes for å bevise global progresjon i et distribuert system. Et slikt bevis gjennomføres for dette konkrete eksemplet.

En eller flere produsenter produserer en eller annen form for data som leveres til en eller flere konsumenter. De produserte data kan for eksempel utveksles mellom de to gjennom et *eksternt* buffer eller lignende. Dette vil gjenspeile den mer tradisjonelle metoden å synkronisere en slik modell på. Her må en konsument vente dersom produsentene ennå ikke har produsert de data som trengs. Dette er tilfelle både dersom man benytter et buffer eller ikke gjør det. I buffervarianten vil dette si at bufferet er tomt. Videre vil produsentene måtte vente dersom ingen av konsumentene er klare til å motta data. I tilfellet der man benytter seg av et buffer vil produsentene måtte vente dersom bufferet blir fullt.



Figur 4.1: Kommunikasjon mellom objektene

De to delen i modellen trenger altså en kontrollmekanisme som sikrer at ulik eksekveringshastighet mellom de to ikke gir oss en inkonsistent modell. For enkelt å bygge opp modellen vår, anta at vi har et lite system med en produsent og en konsument.

Vi trenger en eller annen form for synkronisering/kommunikasjon mellom de to prosessene slik at vi kan garantere at alle data som produsenten har produsert, blir konsumert av konsumenten, og at dette skjer i korrekt rekkefølge. Vi tenker oss en objektorientert modell som består av tre deler:

- Produsenten
- Konsumenten
- Et kommunikasjonsbuffer som holder rede på de produserte data fra produsenten inntil konsumenten ber om dem.

I de tilfeller der man benytter seg av tradisjonelle former for synkronisering vil bufferet typisk bli implementert som en global datastruktur med for eksempel en global teller som forteller om indeksen til siste produserte element og så videre. Produsenten og konsumenten vil typisk bli implementert som prosesser. I en objektorientert modell vil produsenten og konsumenten bli lagd som objekter. Disse skal så leve et selvstendig liv og kommunisere/utveksle data via meldinger. Dette er den enkleste biten av problemet. Hvordan man skal implementere bufferet blir litt vanskeligere, eller i hvertfall mer avhengig av hva slags form for synkronisering man velger. I det tilfellet der man ønsker at modellen skal være så objektorientert som mulig vil bufferet kunne implementeres som et eget objekt. Produsenten ville sende sine produserte data til bufferobjektet, der de lagres inntil konsument objektet ber om dem. Kommunikasjonen mellom de tre objektene kan beskrives med modellen i figur 4.1.

Hvordan skal man så mest mulig effektivt synkronisere en slik distribuert, objektorientert modell? En mulighet er selvfølgelig å benytte seg av velkjente metoder slik som låsing og så videre. Denne metoden vil selvsagt fungere, men kan vel ikke sies å passe inn i et OODS, der uavhengighet mellom objektene samt parallellitet er

en av de mest attraktive egenskaper. Her vil man ved konflikt, bevare alle de effektene man har av å implementere problemet på en tradisjonell, ikke objektorientert måte. En konsekvens er at man mister noe av parallelliteten et OODS tilbyr ved at man låser selv om dette i ettertid viser seg å være unødvendig.

Produsenten og konsumenten må vente i nøyaktig de samme tilfeller som før, det vil si når bufferet er henholdsvis fullt eller tomt. Den eneste forskjellen i eksekvering vil som sagt være at produsenten og konsumenten kan eksekvere parallellt mellom konfliktsituasjonen. Dette vil gi en viss effektivitetsøkning, men *mekanikken* i modellen har ikke forandret seg nevneverdig.

Spørsmålet blir nå om tidsstempelbasert synkronisering vil gi oss en mer strømlinjeformet modell som er minst like effektiv. Umiddelbart kan vi ihvertfall slå fast at denne formen for synkronisering passer bedre inn i den distribuerte, objektorienterte modellen. Implementasjon av bufferet kan i en slik modell bli gjort som et eget objekt.

La oss først gå tidsstempelmekanismen nærmere etter i sømmene. Det skal være slik at hvert av de tre objektene skal ha sin egen lokale logiske/virtuelle klokke. Denne klokken skal være en enkel teller (se [17, 22]) som tikker heltallsverdier slik det er beskrevet i kapittel 2.1.2.2 og 3.1. I denne modellen kan vi godt anta at den tikker mot uendelig og at vi ikke kan få noen form for «overflow».

Kommunikasjonen mellom objektene skjer ved hjelp av meldinger. Disse meldingene sendes ut fra et objekt med adresse til et annet objekt uten at vi har noen garanti for at meldingene kommer frem i den rekkefølgen de ble sendt. Det eneste kravet vi legger på avsending/mottak av meldinger er at en melding som er sendt vil komme frem til riktig mottaker innen endelig tid. Vi sier at meldingsutvekslingen skal være *sikker*.

4.1.1 Meldingene

Alle meldingene som sendes mellom objektene i denne modellen kan presenteres som et tuppel. En melding α er på formen:

$$(\mathcal{T}, \mathcal{R}, \mathcal{S}, \mathcal{M}, \mathcal{D})$$

- $\mathcal{T}$ Inneholder tiden til den virtuelle klokken hos avsenderobjektet da meldingen ble sendt.
- $\mathcal{R}$ Inneholder den virtuelle tiden for når meldingen skal behandles av mottakerobjektet, det vil si meldingens tidsstempel.
- $\mathcal{S}$ Inneholder adressen til avsenderobjektet.

$\mathcal{M}$ Inneholder adressen til mottakerobjektet.

$\mathcal{D}$ Inneholder de produserte data dersom meldingen kommer fra produsentobjektet.

Så kommer vi her til et viktig poeng som illustrerer tidsstempelmekanismen godt. Alle objekter i en slik modell skal ha en innkø. Se [17] og kapittel 3.1 for den orginale *Time Warp*-modellen. I denne køen skal alle innkomne meldinger køes sortert etter verdien på mottakertiden $\mathcal{R}$ som er meldingens *tidsstempel*.

Alle meldinger som blir sendt ut vil da bli køet i innkøen til mottakerobjektet. Tidsstempelmekanismen sikrer altså i seg selv at alle meldinger som er mottatt vil bli køet i korrekt rekkefølge. Dersom vi nå hopper over bufferobjektet i sin helhet får vi et bilde der produsenten sender sine produserte data direkte til produsentobjektet. Bufringen vil så komme gratis gjennom bruk av tidsstempelbasert synkronisering. *Time Warp* forutsetter at en implementasjon skal være så robust at meldinger ikke kan gå tapt. Dette sikrer oss altså at alle meldinger som sendes ut fra $\mathcal{O}_p$ vil komme frem til konsumentobjektet innen endelig tid[1].

Det eneste som nå kan gå galt er dersom en melding kommer frem til mottakerobjektet så forsinket at meldinger som er av nyere dato enn den forsinkede allerede er prosessesert og fjernet fra mottakerobjektets innkø[2]. Men også dette problemet vil bli elegant løst av tidsstempelmekanismen. Her vil mottakerobjektet gjøre en tilbakerulling og hente inn den sene meldingen, før den igjen fortsetter sin eksekvering.

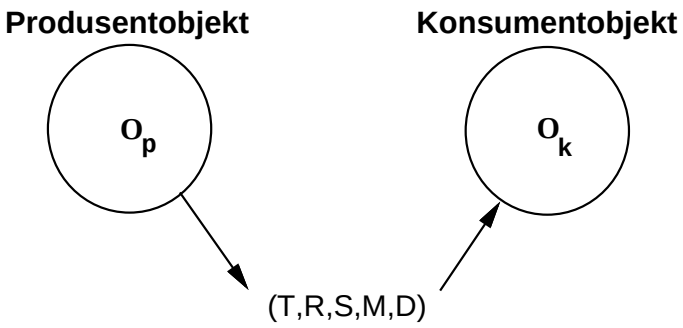
Det kan altså se ut som om denne synkroniseringsmetoden i seg selv vil implementere bufring av data. Dermed vil $\mathcal{O}_b$ helt miste sin funksjon og kan utelates i modellen.

En slik modell vil ha nøyaktig samme type meldinger som beskrevet ovenfor. La oss fortsette å bygge opp en modell, men nå helt uten bufferobjektet. Kommunikasjonen mellom de to objektene illustreres i figur 4.2. Produsentobjektet produserer sine data med ukjent hastighet og sender en melding til konsumenten hver gang data er produsert. Siden produsenten står for produksjon av data og leverer disse til konsumenten der de automatisk blir bufret gjennom tidsstempelmekanismen, vil tilbakerulling ikke forekomme i produsentobjektet. Dette vil bare inntreffe i konsumentobjektet.

Generelt er det slik i TBS at vi baserer oss på en *optimistisk hypotese* om de ulike objektene i systemet og deres hastighet/utvikling i forhold til hverandre. Denne optimistiske hypotesen er generelt en påstand om hvordan systemet vil se ut under hele eksekveringen. Det skal være slik at dersom denne hypotesen holder under

[1] Med dette menes at en melding vil komme frem til mottaker innen endelig reell tid

[2] Merk at de er fjernet fra fremtidsdelen av innkøen. De vil fortsatt ligge i fortidsdelen av innkøen.



Figur 4.2: Kommunikasjon mellom objektene

hele eksekveringen så vil *tilbakerulling* ikke forekomme. Tilbakerulling inntreffer altså hver gang denne optimistiske hypotesen ikke holder. I mitt eksempel vil en slik hypotesen påstå at:

$\mathcal{H}$: Alle meldinger som sendes ut fra produsenten vil bli køet i innkøen til konsumenten før dataene skal benyttes. Det vil si at alle innkomne meldinger vil ha en mottakerverdi som er høyere en konsumentobjektets lokale virtuelle tid.

I en mer tradisjonell *pessimistisk* løsning av synkroniseringsproblematikken vil alle brudd på en slik hypotese typisk kunne føre til låsesituasjoner, eller at deler av systemet stopper opp på en eller annen måte. Merk at tilbakerulling vil forekomme nøyaktig i de tilfeller der tradisjonell modell ville låse og at denne låsingen i ettertid viste seg å være nødvendig.

Legg merke til at dersom denne hypotesen holder til enhver tid under eksekveringen av systemet, vil en *optimistisk* og en *pessimistisk* løsning oppføre seg tilnærmet likt. Hva betyr så det faktum at vi antar denne hypotesen for eksemplet vårt? Først og fremst viser hypotesen at produsenten på mange måter lever sitt eget liv. Objektet produserer sine data og leverer disse til konsumentobjektet uten at det mottar noen svarmeldinger eller har noen form for synkroniseringsrestriksjoner i forhold til eksekveringen.

Konsumenten er derimot mer avhengig av sine omgivelser. Den er avhengig av at data skal være produsert og kommet frem når den trenger dem. Hvis dette ikke er tilfelle så vil konsumenten enten måtte vente, dersom innkøen er tom, eller den vil måtte gjøre en tilbakerulling senere i eksekveringen. Selv dersom hypotesen ikke holder, så vil ikke dette føre til noen forandringer/effekter på produsentobjektet. Det er hos konsumentobjektet slike situasjoner oppstår.

4.1.2 Kort om de to objektene

Produsenten står for produksjonen av en eller annen form for informasjon/data som skal formidles til konsumenten. Hva disse data består av er for denne modellen uinteressant, men la oss for enkelhets skyld anta at den produserer heltallsverdier. Hver gang produsentobjektet produserer data skal den sende disse til konsumentobjektet i en melding. Når en melding sendes skal den lokale virtuelle klokken tikke en verdi.

Når konsumentobjektet henter ut en melding fra køen som stammer fra produsentobjektet henter den ut verdien fra $\mathcal{D}$ parameteren og konsumerer denne. La oss si at dette betyr en oppsummering av alle $\mathcal{D}$-parameterene, det vil si de produserte heltallene fra produsenten. Videre øker den verdien av sin egen logiske klokke til verdien av mottakertiden $\mathcal{R}$ i meldingen, dersom denne er høyere enn klokkeverdien i objektet. Videre vil den lagre sin tilstand før meldingen ble hentet inn i objektets lokale tilstandskø.

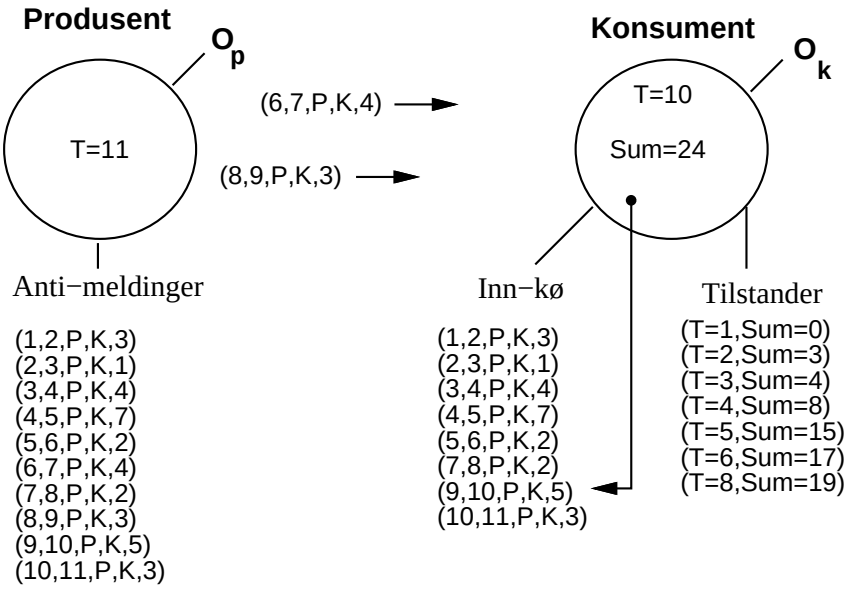
Meldinger fra produsentobjektet kan føre til en situasjon som vil gi tilbakerulling i konsumentobjektet. Dersom konsumentobjektet henter inn en melding fra produsenten som har en tidsstempelverdi $\mathcal{R}$ som er *lavere enn* verdien av konsumentobjektets lokale virtuelle klokke, betyr dette at konsumenten har lest og prosessert en melding for tidlig. Anta i dette eksemplet at konsumenten ikke sender noen meldinger videre. Utsendelse/køing av antimeldinger kan dermed sløyfes.

4.1.3 Produsent/konsument-modellen

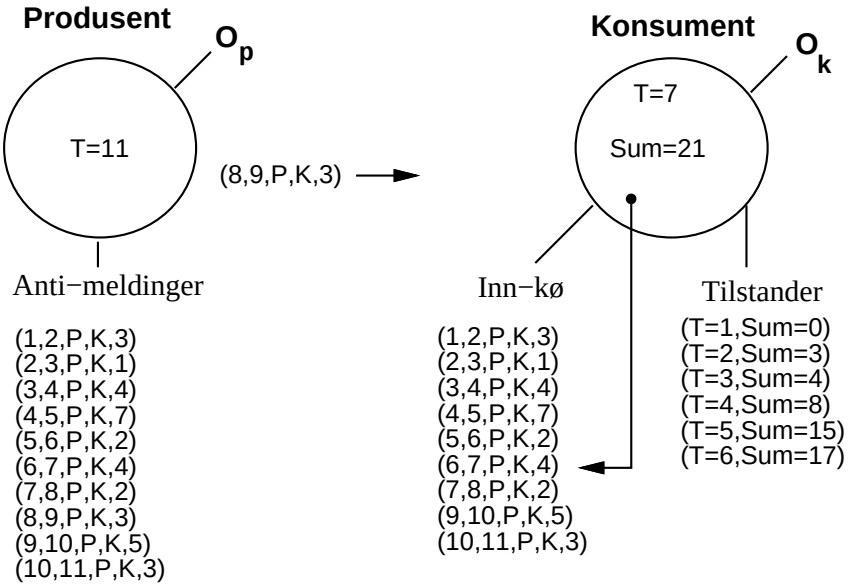
For å illustrere mekanismene, la oss kikke på et mulig utseende av en modell etter en kort stunds eksekvering. La oss anta at den virtuelle klokken hos produsenten har tikket opp til verdien 11. Se figur 4.3.

Figur 4.3 viser oss en situasjon der konsumentobjektet sitter inne med inkonsistente data. Dette skyldes at meldingen som skal mottas av konsumenten ved virtuell tid 7 ikke har kommet frem ennå. Videre er meldingen med virtuell mottakertid 8 allerede prosessert av konsumenten. Denne situasjonen vil føre til en tilbakerulling i konsumenten. Dette inntreffer ved tidspunktet den forsinkede meldingen ankommer konsumenten og denne blir prosessert. Den vil da oppdage at den lokale virtuelle tiden i klokketelleren er høyere enn mottakstiden i meldingen. Den må så rulle tilbake til tiden i meldingen. Korrekt tilstand blir opprettet ved hjelp av tilstandene som ligger lagret i tilstandskøen. Etter en tilbakerulling vil modellen se ut som i figur 4.4.

Konsumentobjektet har nå rullet seg tilbake i tiden til før den skulle ha mottatt meldingen som forårsaket tilbakerullingen. Objektet kan nå fortsette sin eksekvering



Figur 4.3: Kommunikasjon mellom objektene



Figur 4.4: Kommunikasjon mellom objektene

forover igjen. Fra figurene ser vi at det kan bli nødvendig å foreta enda en tilbakerulling fordi meldingene med mottakertid lik 10 og 11 har kommet frem uten at melding med mottakertid 9 har gjort det. Dersom denne kommer frem før melding 10 prosesseres, er alt korrekt. Hvis ikke, må det ved et senere tidspunkt gjøres en tilbakerulling til tiden 9.

4.1.4 Fremdrift i systemet, GVT

La oss nå se på begrepet GVT. Global virtuell tid skal være et mål på hvor langt frem i tid systemet som helhet har kommet. Denne tiden sier oss altså noe om hvilken hastighet systemet beveger seg forover med. Faktisk er det helt essensiellt å vise at denne faktisk må bevege seg forover i tid slik at systemet faktisk kan *terminere*. På grunn av tilbakerullingsmekanismen er det jo slik at lokale virtuelle klokker inne i objektene kan bevege seg *bakover* i tid, før de igjen fortsetter forover. La oss først minne om den originale definisjonen av GVT i [17] som er gjengitt i definisjon 3.1.1.

I modellen i figur 4.4 er da GVT lik 7. Denne stammer fra meldingen som forårsaker tilbakerullingen i konsument objektet. Dette skulle da representere en *sikker* tid i den forstand at tilbakerullinger som går lengre tilbake enn GVT ikke er mulig. Dette fremgår også av figuren.

For lettere å kunne gjennomføre dette beviset, la oss først se på systemet og angi noen enkle lemmaer. Bevisene for disse er trivielle og vi nøyer oss dermed med en kort redegjørelse istedet for formelle bevis. Produsentobjektet produserer data med ukjent hastighet, men det må være en rimelig antagelse at objektet ikke terminerer unormalt. Dette betyr at dersom objektet har produsert et dataelement ved reell tid r_1, så kan vi anta at det enten vil terminere eller produsere et nytt dataelement og inkrementere sin virtuelle klokke ved en reell tid r_2 der $r_1 < r_2$. Når vi så i tillegg vet at i denne modellen vil tilbakerulling aldri forekomme i produsentobjektet, kan vi sette opp følgende lemma:

Lemma 4.1.1 *Den virtuelle klokkeverdien $\mathcal{T}_p$ i $\mathcal{O}_p$ vil alltid øke, inntil produksjonen stopper.*

Videre er det slik at alle meldinger som er sendt, men ennå ikke prosessert, må ha kommet fra produsentobjektet. Dette er de meldinger som enten er på vei fra produsenten, eller har kommet frem til innkøen i konsumentobjektet, men ennå ikke er lest av dette. La oss anta at en av disse meldingene vil forårsake en tilbakerulling i konsumentobjektet. Fra modellen er det relativt enkelt å se at en slik tilbakerulling ikke kan føre til meldinger *utenfor* objektene som har en tidsstempelverdi som er lavere enn den verdien som ligger i meldingen som forårsaket tilbakerullingen. Utfra dette og lemma 4.1.1 kan vi sette opp et nytt lemma som gjelder i modellen:

Lemma 4.1.2 *Den laveste tidsstempelverdien i meldinger som er sendt av $\mathcal{O}_p$, men ennå ikke er behandlet av $\mathcal{O}_k$ vil aldri minke.*

La oss omtale tidsstempelverdien i meldingene som omtales i lemma 4.1.2 som $\mathcal{T}_{sm}$. Siden alle disse meldingene er produsert og sendt av produsentobjektet, vet vi at dette har oppdatert sin egen lokale virtuelle klokke ved avsending av meldingen. Dette innebærer at vi har følgende sammenheng mellom to verdier:

Lemma 4.1.3 $\mathcal{T}_{sm} \leq \mathcal{T}_p$

La oss igjen se på definisjon 3.1.1. Vi ser at lemma 4.1.3 gir oss en relasjon mellom de to elementene i denne definisjonen. Dette innebærer at vi kan definere oss en strengere versjon av GVT fordi elementet $\mathcal{T}_p$ kan utelates på grunn av lemma 4.1.3. Vi får da følgende definisjon av GVT:

Definisjon 4.1.1 *GVT ved reell tid* r *er den minste av (1) den virtuelle tiden i konsumentobjektet og (2) alle virtuelle avsender tidsstempel i alle meldinger som er sendt, men som ennå ikke er prosessert.*

Uttrykt ved hjelp av symboler fra argumentasjonen ovenfor blir dette:

Definisjon 4.1.2 $GVT_{PK} \stackrel{\text{def}}{=} \begin{cases} \mathcal{T}_{sm}, & \textit{dersom } \mathcal{T}_{sm} \leq \mathcal{T}_k \\ \mathcal{T}_k, & \textit{ellers.} \end{cases}$

La oss ta utgangspunkt i denne definisjonen når vi skal bevise at GVT aldri vil minke og at den faktisk vil bevege seg forover inntil systemet *terminerer*. La oss først sette opp et teorem som sier at GVT_{PK} aldri kan minke:

Teorem 4.1.1 *Anta $GVT_{PK} = t_1$ ved reell tid r_1 og at $GVT_{PK} = t_2$ ved reell tid r_2 der $r_1 < r_2$. Da gjelder $t_1 \leq t_2$.*

Bevis 4.1.1 *Bevis for teorem 4.1.1*

Fra definisjon 4.1.2 har vi at det er to ulike representasjonmuligheter for GVT. Beviset deles derfor opp i fire ulike tilfeller. Det essensielle her er at vi ser på hva som skjer ved en endring i GVT og alle de fire ulike mulighetene vi har når det gjelder veksling av representasjon av GVT:

1. *Anta $GVT_{PK} = \mathcal{T}_{sm} = t_1$, ved reell tid r_1*
   og at $GVT_{PK} = \mathcal{T}_{sm} = t_2$, ved reell tid r_2, der $r_1 < r_2$.

2. *Anta* $GVT_{PK} = \mathcal{T}_{sm} = t_1$, *ved reell tid* r_1
   og at $GVT_{PK} = \mathcal{T}_k = t_2$, *ved reell tid* r_2, *der* $r_1 < r_2$.

3. *Anta* $GVT_{PK} = \mathcal{T}_k = t_1$, *ved reell tid* r_1
   og at $GVT_{PK} = \mathcal{T}_k = t_2$, *ved reell tid* r_2, *der* $r_1 < r_2$.

4. *Anta* $GVT_{PK} = \mathcal{T}_k = t_1$, *ved reell tid* r_1
   og at $GVT_{PK} = \mathcal{T}_{sm} = t_2$, *ved reell tid* r_2, *der* $r_1 < r_2$.

For alle disse tilfellene må vi så vise at $t_1 <= t_2$. *Dette må så gjelde for alle reelle tidspunkt* r_i *da* produksjonen *ennå er igang. Det vil si at produsenten faktisk produserer data som den sender ut i meldinger og at ikke systemet er ferdig eller har* terminert.

1. *Anta* $\exists r_2 \mid t_1 > t_2$. *Dette ser vi relativt fort at er umulig.* $\mathcal{T}_{sm}$ *vil alltid være det minste tidsstempelet til* meldinger i omløp. *Lemma 4.1.1 sier at en lavere verdi ikke kan stamme fra en ny melding fra produsenten. Dersom dette skulle være tilfelle måtte da den* nye *meldingen komme fra en tilbakerulling i konsumenten. Dette er umulig fordi den meldingen som kan forårsake en tilbakerulling lengst mulig tilbake i tid i* $\mathcal{O}_k$ *vil nettopp være meldingen som inneholder* $\mathcal{T}_{sm} = t_1$ *ved reell tid* r_1 □

2. *Anta* $\exists r_2 \mid t_1 > t_2$. *GVT er altså her representert ved* $\mathcal{T}_{sm}$. *Dette innebærer at det ikke eksisterer en annen melding som er sendt av produsenten, men ennå ikke prosessert av konsumenten som har en lavere virtuell tid. Det følger da at konsumenten ikke kan rulle tilbake meldinger med lavere virtuell klokkeverdi, for* $\mathcal{T}_{sm}$ *representerer den lavest mulige verdi for denne å rulle tilbake til. Videre sikrer lemma 4.1.2 at slike meldinger heller ikke kan komme fra produsenten. Dermed er antagelsen ovenfor umulig.* □

3. *Anta* $\exists r_2 \mid t_1 > t_2$. *Dette er umulig. Definisjon 4.1.2 gir at ved tiden* r_1 *gjelder* $GVT_{PK} = \mathcal{T}_k \Rightarrow \mathcal{T}_k < \mathcal{T}_{sm}$.

   Dette betyr at det umulig kan komme en melding som ved noe tidspunkt senker $\mathcal{T}_k$ *til en verdi lavere enn* t_1 *på grunn av lemma 4.1.2.* □

4. *Anta* $\exists r_2 \mid t_1 > t_2$. *Definisjon 4.1.2 gir at ved tid* r_1 *gjelder* $GVT_{PK} = \mathcal{T}_k \Rightarrow \mathcal{T}_k < \mathcal{T}_{sm}$. *Det eksisterer altså ingen melding sendt av produsenten, men ikke enda prosessert av konsumenten med virtuell tid lavere enn* $\mathcal{T}_k = t_1$. *Sammen med lemma 4.1.2 gir dette at antagelsen over er umulig.* □

Teorem 4.1.1 er nå bevist og vi vet at GVT_{PK} aldri vil minke. Det gjenstår så å overbevise seg selv om at denne faktisk også vil øke. Men dette er enklere enn det i første øyekast kanskje kan virke som.

Bevis 4.1.2 *Bevis for at GVT er økende.*

Anta $GVT = \mathcal{T}_{sm} = t_1$ ved reell tid r_1. Vi vet fra antagelsene i modellen at en melding som er sendt vil komme frem til mottaker innen endelig tid. Dette betyr at meldingen som inneholder tidsstempelverdien $\mathcal{T}_{sm}$ vil ved en reell tid r_2 der $r_1 < r_2$, komme frem til konsumentobjektet. Om denne nå forårsaker en tilbakerulling eller ikke er uinteressant fordi både $\mathcal{T}_{sm}$ og den virtuelle tiden i konsumentobjektet vil etter dette ha en høyere verdi enn t_1. Og dette er de to mulighetene som finnes i representasjonen av GVT. Denne må derfor i dette tilfellet øke. □

Anta så at $GVT = \mathcal{T}_k = t_1$ ved reell tid r_1. Dette betyr at det ikke eksisterer en melding som er sendt av produsenten, men ikke ennå mottatt av konsumenten som har tidsstempelverdi lavere enn verdien t_1. Den neste gangen konsumenten nå leser en melding må denne ha en høyere tidsstempelverdi enn t_1. I tillegg vet vi at en senere tilbakerulling aldri kan rulle objektet lengre tilbake enn $t_1 + 1$. Det finnes altså en reell tid r_2 der GVT_{r_2} må være større enn GVT_{r_1}. □

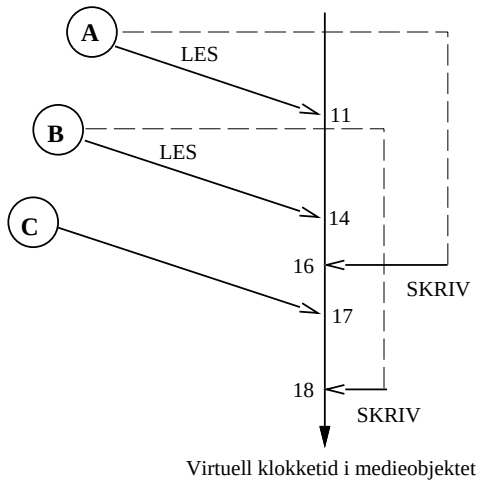
Vi har nå vist at GVT alltid vil øke. Systemet vil altså bevege seg fremover i tid med hensyn på GVT og vi kan da si at systemet kan terminere/avslutte.

Et interessant poeng er at dette beviset også umiddelbart vil holde dersom vi innfører mange produsentobjekter. $\mathcal{T}_p$ er i dette tilfellet uinteressant og er utelatt fra den spesielle GVT-definisjonen i eksemplet.

4.2 Lese/skrive-problemet

Et annet velkjent problem er lese/skrive-problemet. Motivasjonen for dette eksemplet er å vise hvordan TBS kan tenkes benyttet for parallellitetskontroll i en konkret og enkel, men allikevel tenkbar situasjon. Problemet i seg selv er et typisk parallellitetskontrollproblem der flere prosesser eller objekter konkurrerer om tilgang til et medium, for eksempel en database. Eksemplet viser hvordan det er nødvendig å utvide/tilpasse *Time Warp* for effektiv bruk innenfor parallellitetskontroll.

I eksemplet skal vi se på en variant av problemet med et uvisst antall lesere og skrivere. Modellen består av et antall objekter som både kan lese og skrive på et medium. Et objekt leser data fra et medium og utfører, muligens, en skrive operasjon etter at lesingen er ferdig. Om objektet skriver data tilbake til mediet etter at lesingen er ferdig kan ikke forutsies, men skal være avhengig av de data som leses. Vi har altså her et objekt som står for administrasjon av mediet og et antall lese/skrive-objekter som sender meldinger til dette objektet om når de vil lese/skrive. Det er ingen form for *låsing* av mediet slik vi typisk ville ha gjort det i en tradisjonell modell. Vi bygger her modellen vår på en *optimistisk* hypotese:



Figur 4.5: Feilsituasjon lesing/skriving

$\mathcal{H}$: Ingen av objektene blir forstyrret av et annet objekt mellom det tidspunktet det starter sin lesing til det eventuelt avslutter sin skriving. Det vil si at ingen av de andre objektene foretar en lesing med påfølgende skriving innenfor det tidsrom hvor vårt objekt arbeider på mediet.

Som vi ser av figur 4.5 er det selvfølgelig ikke alltid slik at dette vil være situasjonen. Denne hypotesen vil i praksis bli brutt. Hvor ofte dette skjer vil selvfølgelig avhenge av applikasjonen og kan vanskelig sies noe om. Det vi derimot vet er at hver gang denne hypotesen blir brutt så må dette føre til en tilbakerulling i et eller flere av de involverte objektene.

I figur 4.5 vil objektene A og B være i en konfliktsituasjon. Konflikten fører til tilbakerulling innenfor objekt B. Hvordan skal vi så bygge opp modellen vår for å kunne sikre synkronisering av systemet og at systemet garantert skal være *konsistent*, det vil si til enhver tid inneholde korrekte data? Med korrekte data menes at alle dataene skal være slik som hvert av objektene beskriver dem gjennom sin tilstandskø eller objekthistorie. Som vi husker skal hvert objekt ha en tilstandskø som inneholder alle tilstander som objektet har hatt tilbake i tiden. Tilstandskøen skal ihvertfall inneholde alle tilstander hos objektet tilbake til tiden som er representert ved GVT-begrepet.

Alle eventuelle situasjoner der vår optimistiske hypotese ikke gjelder skal altså resultere i tilbakerulling i alle de involverte objektene og de skal tilslutt ende opp med *korrekte data*. Slik konsistens skal finnes for alle tilstander i alle objekter i systemet, som har tidsstempelverdier mindre enn GVT.

4.2.1 Kommunikasjonen mellom objektene

For at vi skal sikre synkronisering og konsistens må det i denne modellen være slik at et objekt sender både en lesemelding og eventuelt en skrivemelding ut i systemet for hver gang den gjør noen av operasjonene på mediet.

Alle meldinger som sendes rundt i systemet skal være av følgende type:

$$(\mathcal{TS},\mathcal{TM},\mathcal{T},\mathcal{M},\mathcal{A},\mathcal{D})$$

De ulike parametrene i et meldingstuppel representerer følgende data:

$\mathcal{TS}$ Avsendertiden på meldingen. Dette er den virtuelle tiden hos avsenderobjektet da meldingen ble sendt.

$\mathcal{TM}$ Tidsverdien da meldingen må prosesseres av mottager, altså meldingens tidsstempel.

$\mathcal{T}$ Meldingstype. Parameteren angir en leseoperasjon (**L**) eller en skriveoperasjon (**S**).

$\mathcal{M}$ Mottakeradresse. Identifikasjon av hvilket mottaker objekt som skal ha meldingen.

$\mathcal{A}$ Avsenderadresse. Identifikasjon av avsenderen.

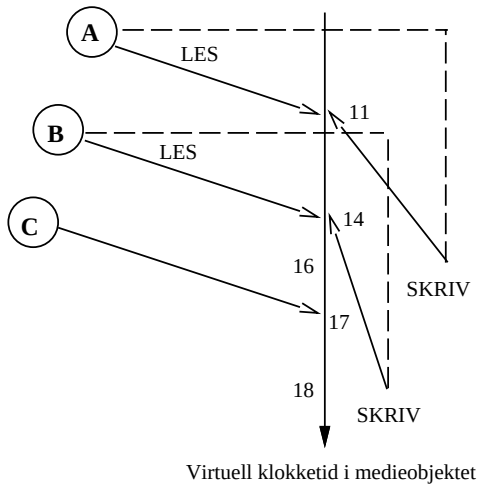
$\mathcal{D}$ Dataparameter.

Som vi skal se konstrueres tidsstempelverdien og avsenderverdiene på en litt annerledes måte i dette eksemplet enn det som er gjort tidligere. Poenget er å foreta en intelligent køing av meldingene med hensyn på meldingstyper samt hvilket objekt meldingen kommer fra.

I tillegg til standard parametere har jeg innført en typeparameter slik som det også er gjort i Dynamo[27]. Denne typeparameteren benyttes for å angi hvilken operasjon som objektet foretar ved det tidspunktet som er angitt i $\mathcal{TS}$-parameteren.

4.2.2 Sikring av konsistens gjennom parallellitetskontroll

Det er her av avgjørende betydning for konsistensen i systemet og de data som til enhver tid ligger i vårt medium at et objekts leseoperasjon som følges av en skriveoperasjon, ikke forstyrres av noen andre objekter. For å kunne sikre en slik parallellitetskontroll kan vi velge en fremgangsmåte som vi kjenner fra littaraturen om transaksjoner. Poenget her er at vi anser at et objekts leseoperasjon etterfulgt



Figur 4.6: Historisk modell av forrige figur ved bruk av nye tidsstempel

av en skriveoperasjon skjer på *samme tidspunkt* sett med virtuell tid for systemet. Dersom et objekt starter en leseoperasjon ved et gitt tidspunkt $\mathcal{TS}_L$ så skal objektet melde fra om dette til objektet som administrerer mediet i systemet.

$$(\mathcal{TS}_L, \mathcal{TM}_L.1, \mathbf{L}, O_{medium}, O_0, 0)$$

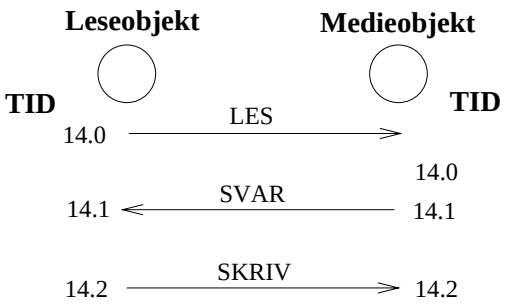
Dersom vi benytter *Time Warp* slik den tidligere er beskrevet og implementert vil vi her få problemer. Anta at et leseobjekt starter en lesing av mediet ved virtuell tid 14. Dersom dette objektet nå mottar et svar fra mediet med de data som er forespurt og bestemmer seg for å skrive tilbake til mediet blir dette gjort ved tidspunkt 15. Anta så at et annet objekt leser data fra mediet ved tid 15, og at meldingen fra dette objektet ankom før skrivemeldingen fra det første objektet. Dette vil resultere i at det andre objektet får inkonsistente data tilbake fra medieobjektet uten at noen tilbakerullingssituasjon oppstår.

Vi må altså innføre regler for hvordan tidsstempel, eller mottakertider i meldinger, skal velges for at det hele skal bli korrekt. Dette kan vi sikre ved å la alle lesemeldinger alltid være på formen $\mathcal{TM}_L.1$ og alle skrivemeldinger alltid være på formen $\mathcal{TM}_S.2$. De to verdien av $\mathcal{TM}$, det vil si $\mathcal{TM}_L$ og $\mathcal{TM}_S$ skal være like dersom de to meldingene kommer fra den samme transaksjonen.

Objektet sender da altså følgende melding til objektet som administrerer mediet:

$$(\mathcal{TS}_S, \mathcal{TM}_S.2, \mathbf{S}, O_{medium}, O_0, \mathcal{D})$$

Vi har nå sikret oss at tilfellet ovenfor som gir inkonsistens ikke lenger vil være et problem. Dersom dette objektet nå leser inkonsistente data ved virtuell tid 15.1, vil dette bli korrigert når medieobjektet senere må rulle tilbake til tid 14.2 på grunn av



Figur 4.7: Et kommunikasjonseksempel fra lese/skrive-modellen

en skrivemelding fra det første objektet. Figur 4.6 viser et bilde av modellen etter at tilbakerullinger og så videre er foretatt. Vi ser at alt nå er i sin skjønneste orden. Så kan man da spørre seg, hva om to objekter forsøker å gjøre en lesing samtidig, som jo vil gå bra, og så forsøke å gjøre en skriving tilbake til mediet samtidig ? Dette er åpenbart et problem. En mulig løsning kunne være å benytte samme teknikk som ovenfor, og så velge postfiks til tidsstemplene slik at man fikk en prioritering mellom objektene. For eksempel ved å la objekt 1 sende meldinger med tidsverdier $\mathcal{TM}_L.11$ og $\mathcal{TM}_S.12$, objekt 2 med tidsverdier $\mathcal{TM}_L.21$ og $\mathcal{TM}_S.22$. Denne prioriteringen ville så bare få effekt dersom konfliktsituasjoner oppstår. Tidsstempelene i disse meldingene har følgende generelle form:

$$\mathcal{TM}_{L/S}.\{\text{unik objektidentifikator}\}\{\text{unik meldingstypeidentifikator}\}$$

Figur 4.7 viser hvordan en transaksjon mellom et leseobjekt og medieobjektet ser ut i modellen min. Merk her at en intern ordning av meldingstyper løste ett synkroniseringsproblem. Derfor indikeres også hvordan konfliktsituasjoner mellom objekter kanskje kan løses på samme måte. Dette diskuteres nærmere i neste kapittel (4.2.3).

4.2.3 Konsekvenser for Time Warp

Som vi ser av eksemplet fikk jeg innført parallellitetskontroll på en noenlunde tilfredsstillende måte, i allefall i teorien. Det førte imidlertid med seg endel endringer av den originale *Time Warp*- modellen. Men dette var ikke spesielt overraskende siden modellen ikke er utviklet i retning av denne typen problemer. Jeg føler at de problemene jeg støtte på i eksemplet kan være av generell art når det gjelder bruk at *Time Warp* i denne type synkronisering. Det mest umiddelbare som kommer frem er at kontroll av parallellitet har en mye sterkere binding til reell tid enn problemer innenfor distribuert simulering. Triksingen som er gjort med tidsstempelverdiene indikerer at TBS er en hel familie teknikker som kan spesifiseres for forskjellige formål. TBS-familien er derfor et naturlig mål for samling/standardisering innenfor et felles rammeverk. Dette er motivasjonen og tema for kapittel 7.

Utvidelsen av tidsstempelbegrepet er en interessant mekanisme som jeg ikke har sett brukt i *Time Warp*-sammenheng tidligere. I [22] innføres to typer ordninger av hendelsene i et distribuert, objektorientert system. Den ene typen er partiell, og denne strammes så inn til å være total. Den *intelligente* køingsmekanismen jeg innfører i dette eksemplet er ikke noe annet enn å gi en total ordning $\prec$ av objektene i systemet, tilsvarende definisjon 2.1.5, samt at man også innfører en total ordning $\ll$ av de ulike meldingstypene. Denne totale ordningen defineres senere i definisjon 4.2.1.

I lese/skrive-eksemplet gjelder det da for eksempel at $\mathcal{TM}_L \ll \mathcal{TM}_S$. Dette gir oss en total ordning av meldingstypene i systemet. I tillegg gjelder den totale ordningen $\prec$ mellom alle objektene som benytter databasen. Den sistnevnte kan være tilfeldig valgt, eller generert av systemet selv. Dette betyr at vi fortsatt kan tillate dynamisk konfigurering i systemet ved at nye objekter kan komme til eller forsvinne. Ordningen $\ll$ er *intelligent* valgt, og kan generelt bestemmes før eksekvering fordi vi vet hvilke meldingstyper vi tillater. Dermed vil heller ikke dette forhindre dynamisk konfigurering.

Grunnen til at dette, såvidt meg bekjent, ikke er omtalt i litteraturen rundt *Time Warp* er antakelig fordi det vil ha mye større betydning innenfor parallellitetskontroll der man har konkurrerende prosesser. Totale ordninger vil som vist i dette eksemplet kunne redusere antall tilbakerullinger.

På denne måten kan vi enten ved kompileringstid, eller ved oppstart av systemet angi en prioritetsrekkefølge av objektene på basis av kunnskap om systemet. I tillegg kan vi angi en prioritering av meldingstypene dersom dette viser seg å være nødvendig.

Dersom det omtalte medium ovenfor for eksempel var en database der objektene har en interaktiv kommunikasjon med en bruker, ville man være tvunget til å innføre strengere krav til binding mot fysisk tid enn hva som indikeres her. Jeg skal i kapittel 6.1 vise et eksempel på en modell der fysisk tid er forsøkt integrert på en naturlig måte. Nettopp dette må være en forutsetning dersom metoden skal kunne brukes til parallellitetskontroll i en *interaktiv*, distribuert modell.

Disse mekanismene er også forsøkt inkludert på en naturlig måte i rammeverket som presenteres i kapittel 7. I de tilfeller der vi ikke trenger en sterkt binding til fysisk tid vil de vanlige mekanismene innenfor TBS være tilstrekkelige til parallellitetskontroll, med mulig effektivitetsforbedring gjennom de totale ordningene som er omtalt.

La oss se litt på de to typene av tidsstempel.

4.2.4 Endimensjonale tidsstempelverdier

Man kan alltid benytte seg av kombinasjoner av heltall for å representere slike virtuelle tider i tidsstempel. Det å bruke enkle heltall som verdier i meldinger gir oss en partiell ordning av hendelsene i systemet. Se definisjon 2.1.2.

I utgangspunktet sorteres meldinger etter verdien av deres tidsstempel når de kommer inn i innkøen til et mottakerobjekt. Hva skjer så dersom det køes opp mange meldinger med lik tidsverdi, og disse kommer fra mange forskjellige objekter i systemet? Det kan være ødeleggende for en modell og det finnes flere eksempler på at dette kan føre til unødvendig tilbakerulling[3].

Dette betyr igjen at det vil gå utover effektiviteten i systemet. La oss for ordens skyld kalle slike originale tidsstempelverdier for *primære* tidsstempelverdier. En modell vil altså ikke ha noen mulighet for å avgjøre hvilke av meldingene med lik primær tidsstempelverdi det vil være lurest å behandle først. En måte å forbedre dette på kan være å innføre en eller annen form for prioritet eller køing av slike meldinger med lik primær tidsstempelverdi slik det er foreslått i kapittel 4.2.3.

4.2.5 Flerdimensjonale tidsstempelverdier

For å ha muligheten til å kunne sortere innkomne meldinger der det primære tidsstempelet er likt, innfører jeg derfor begrepet *sekundære* tidsstempelverdier. Dersom den primære tidsstempelverdien er lik hos to innkomne meldinger sorteres så disse innbyrdes etter den sekundære. Videre blir dette ekvivalent for enda flere dimensjoner dersom dette er ønskelig. La meg anta to dimensjoner i diskusjonen nedenfor for enkelhets skyld.

Ved innføring av flerdimensjonale tidsstempel må man også benytte flerdimensjonale klokker i objektene for å kunne oppdage konfliktsituasjoner. De sekundære verdiene skal derimot ikke *tikke* slik som den primære skal gjøre. Verdien på det sekundære tidsstempelet bør være av en statisk art, det vil si at den alltid er den samme for meldinger som blir sendt fra det samme objektet, eller den samme for samme type meldinger slik det er gjort i kapittel 4.2. Husk at formålet med å utvide tidsstempelverdien var å kunne sortere meldinger som har den samme verdien for sitt primære tidsstempel og dermed definere en total ordning av hendelsene i systemet. For eksempel kan man her tenke seg at man velger den sekundære tidsstempelverdien gjennom at man foretar en intelligent, unik indeksering av meldingstypene som benyttes i det distribuerte systemet. Denne indekseringen må være intelligent i den forstand at den må sikre oss at to meldinger fra samme objekt med

[3] Se lese/skrive-eksemplet i kapittel 4.2. Her forbedret jeg parallellitetskontrollen mellom ulike objekter og meldingstyper ved å trikse med tidsstempelformatene i modellen.

samme tidsstempel[4] blir sortert i den rekkefølgen vi ønsker for å unngå unødig tilbakerulling i systemet.

Slike flerdimensjonale tidsstempler gir oss også muligheten for å køe meldinger med hensyn på en total ordning av objektene. En slik ordning er gitt i [22]. Dette tilsvarer bruk av relasjonen $\prec$ gitt i definisjon 2.1.6. Denne ordningen er altså definert tidligere. For å benytte en total ordning av meldingstypene som et sorteringskriterium slik det gjøres i kapittel 4.2.3, la meg sette opp følgende definisjon:

Definisjon 4.2.1 *Det eksisterer en total ordning $\ll$ av alle meldingstyper i systemet som vil resultere i en intern sortering av ulike meldinger fra samme objekt, men med lik primær mottakertid, i henhold til relasjonsoperatoren $\ll$.*

En slik ordning av meldinger benyttes som intern synkronisering av meldingstyper innenfor en og samme transaksjon. I eksemplet i kapittel 4.2 vil for eksempel en leseoperasjon med tilhørende skriveoperasjon tilbake til databasen tilhøre samme transaksjon, og det primære tidsstemplet vil følgelig ha samme verdi.

Jeg har nå lagt alle forutsetninger på plass for en fornuftig konstruksjon av tidsstempelverdier innenfor parallellitetskontroll. Dermed kan vi eliminere visse konfliktsituasjoner i systemer allerede ved komplieringstid eller eksekveringstid, gjennom litt enkel analyse av det distribuerte systemet.

[4] Samme *primære* tidsstempelverdi. Dette kan inntreffe for eksempel dersom man sender mange meldinger ut i systemet, der alle meldingene tilhører samme transaksjon i passende forstand. Lese/skrive-eksemplet tidligere i oppgaven er en konkret eksempel på dette.

Kapittel 5

Time Warp i distribuert simulering

5.1 Time Warp og distribuert simulering

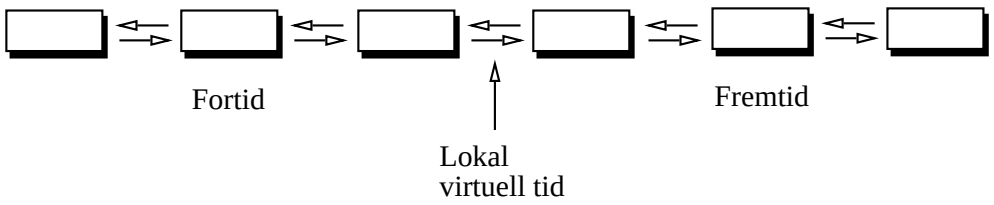
Hovedtyngden av litteraturen om bruk av, og forskning rundt, *Time Warp* finnes innenfor distribuert simulering. Slike systemer består av objekter som i en eller annen forstand samarbeider om å løse en gitt oppgave, eller i dette tilfellet mer spesifikt om å simulere et eller annet gitt system[1]. Slike systemer kan konstrueres lukkede med hensyn på tid. Med dette menes at de kan benytte seg av virtuell tid som en synkroniseringsmekanisme seg i mellom helt uten å ta hensyn til fysisk tid. Dermed kan man i slike systemer benytte seg av den originale *Time Warp*-modellen [17].

Mange av eksemplene som finnes på bruk av metoden innfører riktignok forslag til forbedringer/tilpasninger, men essensielt gjøres det ingen endringer av virkemåten til den originale teoretiske modellen. Forandringene gjøres for å løse praktisk problemer man ofte støter på når en metode skal implementeres på en datamaskin. Dette kan for eksempel dreie seg om reduksjon av minneforbruk og lignende problematikk. Andre forandringer kan igjen gå ut på å legge restriksjoner på hvor langt en prosess kan ligge foran GVT for å forsøke å begrense tilbakerullingshyppigheten og så videre.

5.1.1 Implementasjon av innkø

Den tradisjonelle *Time Warp* beskrevet i [17] gir et bilde av en modell der innkøen i objektene er av lineær karakter. Strukturen på denne køen er kanskje ikke så viktig når det gjelder teorien rundt metode som sådan, men idet man skal ta skrittet ut å implementere den får dette mye å si for effektiviteten av metoden. Blant annet i [29]

[1] På engelsk «cooperating processes».



Figur 5.1: Enkel TWPES - pekerkjede.

beskrives hvordan man på ulike måter kan implementere innkøen i tidsstempelbasert synkronisering.

En melding skal køes i innkøen til et objekt idet den i en passende forstand kommer frem til objektet den er adressert til. Dette impliserer at en implementasjon må ha en operasjon som mottar og legger meldinger inn på *riktig* plass i objektet sin innkø. Videre må objektet ha en peker eller adresse til den posisjonen i innkøen som representerer den neste meldingen som skal hentes inn og prosesseres. Denne pekeren representerer et viktig skille i objektene sine innkøer. Dette er skillet mellom fortid og fremtid sett i forhold til objektets oppfattelse av nåtid, nemlig den virtuelle klokkeverdien til objektet. Operasjonen som legger inn nye meldinger i innkøen opererer altså bare på fortidsdelen av innkøen. I tillegg må vi ha en operasjon som henter ut meldinger fra innkøen. Denne operasjonen kan virke enkel i utgangspunktet fordi den bare henter ut meldingen som pekes på av den tidligere omtalte pekeren eller adressen. Vi skal derimot se at operasjonen har en mye større kompleksitet enn som så fordi den også må implementere tilbakerullingsmekanismen i objeket. Et objekts innkø gir oss altså fire viktige deler som det kan være verdt å diskutere hver for seg:

- Innkø operasjonen (**enqueue** i [29])
- Ut operasjonen (**dequeue** i [29]))
- Innkøens fortidsdel
- Innkøens fremtidsdel

De to ulike delene av innkøen kan implementeres som to uavhengige datastrukturer som kobles sammen av operasjonen som henter ut neste meldingen fra køen. Som vi skal se senere vil denne være ansvarlig for å implementere tilbakerullingsmekanismen. I det minste er det denne operasjonen som er ansvarlig for å kalle en passende rutine som utfører tilbakerullingen. I [29] gis det flere ulike forslag på måter å implementere innkø (TWPES - Time Warp Pending Event Set) funksjonaliteten i tidsstempelbasert synkronisering. Diskusjonen nedenfor baserer seg blant annet på denne artikkelen og de effektivitetsmålinger som er utført av forfatterne.

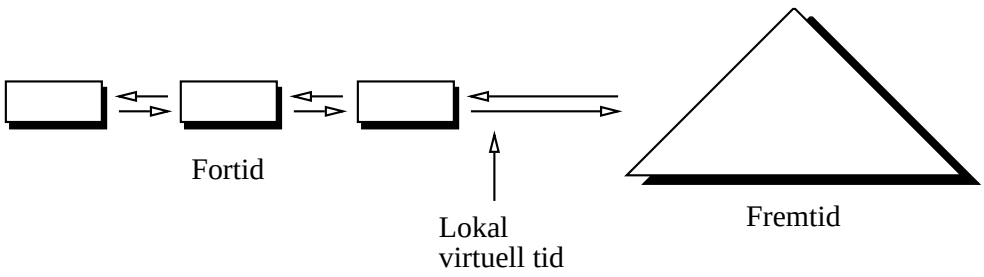
Innkøoperasjonen må implementere fenomenet kansellering av respektive meldinger eller antimeldinger. Dersom en melding som mottas er *positiv*, det vil si ikke en antimelding, skal operasjonen køe meldingen på riktig sted i fremtidsdelen av innkøen. Den må også sjekke den muligheten at den nye meldingen sin antimelding har kommet frem før den positive. Hvis dette er tilfelle, fjernes begge meldingene fra køen. Ingen andre operasjoner er da nødvendig. Tilfellet der den nye meldingen er en antimelding med høyere tidsstempelverdi enn objektets lokale virtuelle klokke sjekkes på samme måte om den tilhørende positive meldingen finnes i innkøen. I såfall kanselleres meldingene og forsvinner. Dersom den positive meldingen ikke finnes, køes antimeldingen etter tidsstempelverdien på vanlig måte. Denne operasjonen opererer helt uavhengig av objektets virtuelle klokke. Tilbakerullingsmekanismen overlates altså helt til den andre operasjonen som er omtalt ovenfor, nemlig den som henter ut *neste* melding fra fremtidsdelen av innkøen. Dersom en melding kommer frem til objektet med en lavere tidsstempelverdi enn objektets egen lokale virtuelle klokkeverdi vil den nye meldingen bli *lagt først* i innkøen. Neste gang operasjonen som henter neste melding blir aktivisert vil denne utføre en tilbakerulling. Dette er uavhengig av om meldingen er positiv eller negativ. Dette medfører at også tilbakerullingsmekanismen må implementere kansellering av en melding og dens respektive antimelding dersom begge disse ligger i fortidsdelen av innkøen etter tilbakerullingen.

Utoperasjonen følger altså den omtalte pekeren, henter ut meldingen og oppdaterer pekerverdien. Dersom den neste meldingen har tidsstempelverdi som er lavere enn objektets lokale virtuelle klokke vil operasjonen måtte utføre en tilbakerulling, eller muligens kalle en annen operasjon som foretar tilbakerullingen. Dette skal utføres uavhengig av om meldingen er en antimelding eller ikke.

En effektiv måte å implementere en tilbakerullingsmekanisme på, kan være å la utoperasjonen hente ut alle meldingene fra fortidskøen med tidsstempelverdi høyere enn den *nye* virtuelle klokkeverdien til objektet. Deretter kan disse bli lagt inn igjen ved hjelp av innkøoperasjonen som allerede finnes til vår disposisjon.

Når det gjelder implementasjon av de to delene av innkøen er det umiddelbart fristende å anta at den mest effektive måten å lage fortidsdelen på må være å benytte seg av en enkel liste eller pekerkjede som for eksempel i figur 5.1.

Det ser ut som om effektiviteten i en implementasjon, ikke overraskende, vil være avhengig av om de ulike objektene i det distribuerte systemet eksekverer på maskiner/prosessorer som har en tilnærmet lik ytelse eller ikke. Et slikt system vil i praksis altså ikke kunne være raskere enn den prosessen som eksekverer med dårligst ytelse i en eller annen forstand. Ytelsen på systemet som helhet vil faktisk være dårligere enn det forskjellen i maskinytelse skulle tilsi. Dette fordi det som nevnt ovenfor vil lede til lengre innkøer i objektene og dermed effektivitetssenking på grunn av lengre søketider ved køing av nye meldinger samt tilbakerullinger over et større tidsrom.



Figur 5.2: TWPES med komplisert datastruktur i fremtidsdelen

Som tidligere nevnt finnes det flere ulike forslag til hvordan man kan tenke seg innkøen implementert. Jeg har ovenfor argumentert for at fortidsdelen av listen kan implementeres som en lineær liste, det vil si for eksempel en pekerkjede. Dette ville gi oss minimal søketid ved tilbakerullinger i objektet fordi man under en tilbakerulling hele tiden er interessert i den første meldingen i køen. Hva så med fremtidsdelen av listen ? Når innkøoperasjonen skal legge inn en ny melding må den finne den rette plassen i listen i henhold til den nye meldingens tidsstempelverdi. Dette betyr at den må gjøre et søk i den allerede eksisterende fremtidsdelen av listen. Det er i dette tilfellet antagelsen ovenfor er av stor betydning. Dersom vi med rette kan anta at meldingene som ankommer et objekt stort sett ankommer i riktig rekkefølge vil kanskje en lineær struktur være den mest effektive. I andre tilfeller vil kanskje et binært søketre eller en *heap* være mest effektivt (se figur 5.2). Valg av struktur på innkøen bør overlates til hver enkelt implementasjon. Ulike strukturer i ulike objekter innenfor det samme OODS må også tillates.

5.1.2 Minneforbruk i Time Warp

Den originale modellen til Jefferson[17] sier at vi må lagre tre ulike typer *minneenheter* i hvert objekt som inngår i den distribuerte modellen. Dette er positive meldinger i innkøen, negative meldinger i utkøen samt en kopi av objektets tidligere tilstander i tilstandskøen. Dette medfører et ikke ubetydelig minneforbruk når man skal implementere tidsstempelbasert synkronisering. Dersom metoden blir implementert rett etter nesa, vil minneforbruket fort kunne nærme seg et uakseptabelt høyt nivå.

Størrelsen på minnet som trengs for å lagre alle positive og negative meldinger i henholdsvis objektenes innkø og utkø vil variere med hensyn til forskjeller i eksekveringshastigheten hos de ulike objektene. La antall objekter i en modell som benytter tidsstempelbasert synkronisering være n. La videre den høyeste lokale virtuelle tiden vi kan finne i et objekt være $\mathcal{T}_{st\o rst}$. Vi kan da gi et noe diffust mål på hvor mange minneenheter som kan genereres i *Time Warp*[2]:

[2] Dette representerer det verst tenkelige tilfellet fordi vi antar at alle objekter har virtuell tid

Definisjon 5.1.1 $\mathcal{M}_\sharp$ =størrelse*(melding)* 2n *($\mathcal{T}_{størst}-$*GVT*), der n er antall objekter og $\mathcal{M}_\sharp$ er antall lagrede meldinger i* Time Warp*-systemet. Merk at med melding menes både vanlige meldinger samt de tilsvarende antimeldingene.*

Vi ser at for hver gang avstanden $\mathcal{T}_{størst}-GVT$ øker med 1, så vil minneforbruket øke med orden $\mathcal{O}(2n)$. En slik bruk av minne kan fort føre til overallokering av minne i maskinen, og det kunne i verste fall føre til at systemet terminerer unormalt. For å sikre seg mot slike situasjoner må man legge restriksjoner på modellen slik at overallokering av minne ikke kan forekomme.

Jefferson beskriver noen mekanismer som kan benyttes for å forhindre overallokering i [18]. Dette går ut på at man setter restriksjoner på hvor stort antall meldinger et objekt har lov til å lagre i sin innkø. Objektet, eller rutinen som legger inn nye meldinger i køen må sjekke at dette antallet ikke overskrides. Dette er tema for kapittel 5.1.2.1

Restriksjonen ovenfor vil i et system være en form for flytkontroll. Essensielt er dette analogt med flytkontrollproblematikk i alle former for distribuerte systemer.

Avstanden $\mathcal{T}_{størst}-GVT$ vil direkte påvirke minneforbruket. I en implementasjon må man altså balansere hyppigheten av GVT-estimering og fossilfjerning slik at minneforbruket holdes på et anstendig nivå. Generelt er det slik at økt frekvens på GVT-estimering vil presse estimert GVT nærmere den teoretiske verdien. Videre vil mekanismen ovenfor med tilbakesending av meldinger legge en øvre grense på avstanden $\mathcal{T}_{størst}-GVT$.

I tillegg til minneforbruket i alle objektenes innkø og utkø, må man også lagre alle tilstander som objektet har befunnet seg i. Med notasjonen ovenfor kunne vi sette opp et mål for minneforbruk når det gjelder tilstandskøene i alle objektene. Vi forutsetter av systemet har en mekanisme som implementerer fjerning av fossiler:

Definisjon 5.1.2 $\mathcal{M}_\mathcal{S}$ =størrelse*(tilstand)* n *($\mathcal{T}_{størst}-$*GVT*), der n er antall objekter og $\mathcal{M}_\mathcal{S}$ er det samlede minneforbruket til tilstandskøene i hele* Time Warp*-systemet.*

Vi ser at dette målet er på samme formen som minneforbruk i innkø og utkø. Hva kan vi så si om forholdet mellom størrelsen på en melding og en tilstand? For et OODS av en viss størrelse der objektene har en stor grad av kompleksitet, er det ikke vanskelig å tenke seg at minneforbruket når det gjelder lagring av en tilstand vil være betydelig større enn ved lagring av en melding.

Forslaget om å begrense antall meldinger som får lov til å være lagret i et objekts innkø vil legge begrensninger på disse måltallene. Det som i praksis vil skje ved en slik restriksjon til *Time Warp* er at man setter en øvre grense for differansen

$\mathcal{T}_{størst}$. Men dette er ikke tilfelle i praksis

$\mathcal{T}_{størst} - GVT$. Dette vil sikre oss at vi ikke kan få overallokering av minne dersom vi velger en øvre grense for denne differansen med omhu. Vi kan si at modellen som helhet vil eksekverer tidsmessig innefor et vindu som ligger mellom GVT og $\mathcal{T}_{størst}$. De tregeste objektene vil ha en lokal virtuell tid rundt GVT (lik eller like i overkant) imens de raskeste vil ha virtuell tid like under eller lik den øvre grensen. Legg merke til at denne øvre grensen ikke er representert ved en fastsatt verdi. En slik øvre grense kan faktisk variere ganske kraftig innenfor det samme systemet sett fra forskjellige tidspunkter. Dersom alle objektene er svært aktive med å sende meldinger vil den øvre grensen nærme seg GVT. I det motsatte tilfellet vil den kunne være lengre fra GVT. Allikevel vil den øvre grensen for minneforbruk kontrolleres fordi det totale antall lovlige ubehandlete meldinger i systemet ikke kan overstige grensen som er satt.

Fra litteraturen har vi nå sett ulike tiltak for å begrense minneforbruk gjennom restriksjoner på lengden av innkøene og som en følge av dette også utkøen og tilstandskøen.

Jeg mener den mest kritiske av disse køene med hensyn på minneforbruk er tilstandskøen. Denne skal lagre en tilstand for et objekt for hver virtuell klokkeverdi, noe som lett kan kreve mye minne. Man kan tenke seg følgende metode for å redusere minneforbruket i tilstandskøen:

Alle objekter skal ha et virtuelt tidsintervall der sluttpunktet for et intervall og startpunktet for neste er et sjekkpunkt med tilsvarende tilstandslagring. Et slikt intervall på for eksempel ti virtuelle tidsenheter vil gi oss 90% reduksjon av minneforbruket til tilstandskøen, siden bare hver tiende tilstand lagres. Dette intervallet kan så varieres fra objekt til objekt, og tilpasses det gjennomsnittelige minneforbruket i forhold til totalt tilgjengelig minne. Tilbakerullingsmekanismen må så endres noe. Ved tilbakerulling må man rulle tilbake til det seneste virtuelle sjekkpunktet før det egentlige målet for tilbakerullingen. Hver tilbakerulling blir altså lengre med en slik løsning. Dette medfører også at vi må være litt mer forsiktig med fossilfjerning. Vi kan ikke lengre fjerne alle meldinger med tidsstempel lavere enn GVT, men må også beholde de med tidsstempelverdi mellom GVT og siste sjekkpunkt før GVT. For et OODS der antagelsen om at en tilstand krever mye mer minne enn en melding, vil dette derimot ikke spille så stor rolle. Den største ulempen med en slik løsning vil være at vi generelt får lengre tilbakerullinger.

5.1.2.1 Tilbakekanselleringsprotokollen

Det har vist seg at fossilfjerning alene ikke gir oss stabil minneadministrasjon i *Time Warp* [18]. Fossilfjerning sikrer oss minneadministrasjon av fortiden. Alle tilstander og meldinger som er så gamle med hensyn på virtuell tid at de ikke lengre kan påvirke resultatet av eksekveringen blir fjernet[3].

[3] Det vil si meldinger med tidsstempelverdi lavere enn GVT.

Som en følge av *Time Warp*-metoden og beslektede metoder sin virkemåte viser det seg at man også må foreta minneadministrasjon av fremtiden. I praksis vil dette si at vi foretar flytkontroll i det distribuerte, objektorienterte systemet.

Som Jefferson påpeker i [18] finnes det utallige protokoller for flytkontroll i distribuerte systemer, men ingen av disse kan tilpasses *Time Warp* på en tilfredsstillende måte. Det er flere grunner til dette. I hovedsak stammer alle fra det faktum at *Time Warp* er en optimistisk metode. De tidligere kjente flytkontrollmekanismene er bygget rundt mer pessimistiske metoder. En melding som er lest og behandlet i en pessimistisk modell kan umiddelbart fjernes fordi synkroniseringsproblematikken er tatt hånd om *før* meldingen nådde mottaker. Vi vet nå at dette ikke er tilfelle for *Time Warp*.

Flytkontroll må derfor behandles på andre måter. Som det ble antydet tidligere kan slik flytkontroll implementeres ved å legge restriksjoner på lengden av innkøen, eller mer spesifikt fremtidsdelen av innkøen. Dette er hovedpoenget med tilbakekanselleringsprotokollen. En grense settes for hvor mange meldinger et objekt kan ha liggende i sin innkø samtidig. Dersom innkøen til et objekt er *full* og en ny melding ankommer, skal den meldingen i innkøen med størst tidsstempelverdi, tas ut av køen og returneres til avsender. Et tilleggskrav er at den nye meldingen også må ha lavere tidsstempel enn meldingen som eventuelt skal tas ut av køen. Dersom den nye meldingen har størst tidsstempel, skal denne returneres istedet. Meldingen som returneres til avsender vil da nødvendigvis forårsake en tilbakerulling i dette objektet, og vil tilintetgjøres sammen med den tilsvarende antimeldingen som er køet i utkøen.

En av antagelsene for den opprinnelige tilbakekanselleringsprotokollen er at hele *Time Warp* systemet eksekverer med **shared memory**. Man kan da betrakte alle innkøene som ett felles buffer. Som en konsekvens av dette kan man da få en situasjon der en melding m_1 med avsender $\mathcal{O}_1$ ankommer et objekt $\mathcal{O}_2$. Innkøbufferet er fullt, men dette resulterer i en tilbakekansellering av en melding m_2 som har mottaker $\mathcal{O}_3$ og avsender $\mathcal{O}_4$. Dette er et morsomt fenomen og helt ulikt det som kan hende innenfor tradisjonelle flytkontrollmekanismer. I kapittel 6.1.1 forklarer jeg kort hvorfor vi ikke kan oppnå slike fenomener ved tilpasning av tilbakekanselleringsprotokollen til generelle OODS som benytter TBS til parallellitetskontroll.

Kapittel 6

Time Warp i parallellitetskontroll

6.1 Time Warp og parallellitetskontroll

Som vi har sett passer den originale *Time Warp* godt som synkroniseringsmekanisme i distribuert simulering. Dersom metoden skal anvendes som en parallellitetskontroll metode blir problemene som skal løses av en litt annen natur.

En slik kontroll skal ofte utføres på et system som består av *konkurrerende* objekter. De kan for eksempel konkurrere om felles ressurser slik som en database eller liknende. Mange konflikter kan i slike tilfeller oppstå. For eksempel kan man tenke seg en situasjon der to avdelingskontorer i et reisebyrå forsøker å reservere den samme plassen på den samme flyavgangen. I slike tilfeller må det utføres en form for prioritering eller vurdering av hvem som faktisk var først ute.

Hele poenget er nå at dette er vanskelig å utføre på noen annen måte enn å benytte seg av fysisk tid. Den som var først ute med hensyn til fysisk tid, skal også få plassbestillingen. Videre må man også i slike tilfeller kunne garantere en responstid innenfor en gitt grense. Ved denne grensens utløp skal brukeren ha fått et reelt svar på sin transaksjon, ellers må denne aborteres eller lignende. Dette fører igjen med seg at metoden må knyttes sammen med fysisk tid på en passende måte. Umiddelbart forstår vi at dette må gjøres *løsest* mulig for å kunne beholde så mye som mulig av fleksibiliteten fra TBS-metodene.

Men en av tingene som gjør *Time Warp* så behagelig å arbeide med er at man kan distansere seg vekk fra bruk av fysisk tid. Slik tid er ikke med som parameter i den originale modellen. Utfordringen ligger altså i å innføre fysisk tid som en parameter i *Time Warp*, for så å konstruere en modell vi kan benytte for å løse problemer rundt parallellitetskontroll i distribuerte, objektorienterte systemer.

I tillegg er det, som nevnt ovenfor, et krav at deler av fleksibiliteten beholdes fra den originale modellen. Det man da kunne tenke seg var en modell som benyttet

en krysning mellom virtuell og fysisk tid. Dette er forsøk gjort med Dynamo[13, 27, 25].

For å lykkes med dette må man ha referanser til fysisk tid, samtidig som man må tillate objekter å avvike fra denne tiden, i hvertfall periodevis. Rent intuitivt kunne man tenke seg en modell som benytter *Time Warp* som en grunnleggende metode, men der man hadde synkroniseringspunkter med hensyn på fysisk tid.

Som vi skal se i kapittel 6.2 er dette noe av motivasjonen for Dynamo-modellen.

6.1.1 Fossilfjerning og tilbakekanselleringsprotokollen i parallellitetskontroll

Når det gjelder bruk av fossilfjerning som minneadministrasjon i *Time Warp*, finnes det ingen kjente argumenter for at denne metoden ikke kan benyttes ved tilpasning av metoden til parallellitetskontroll. GVT-begrepet vil fortsatt ha samme mening som før. Begrunnelsene her er så opplagte at dette ikke diskuteres nærmere.

Når det gjelder bruk av tilbakekanselleringsprotokollen som ble presentert i kapittel 5.1.2.1 må man gå antagelsene litt nærmere etter i sømmene. For det første er en av antagelsene at meldingsutvekslingen er sikker. Men dette er jo generelt et krav for at *Time Warp* skal fungere tilfredsstillende, så antagelsen vil pr. definisjon holde for et OODS som benytter *Time Warp*. En annen antagelse er at *Time Warp*-systemet som benytter tilbakekanselleringsprotokollen eksekverer på et system med **shared memory**. Dette ga oss et interessant fenomen med at en melding kunne tilbakekanselleres fra et annet objekt enn det som mottok meldingen og som dermed *oppdaget* at bufferet var fullt. En slik antagelse om **shared memory** kan selvfølgelig ikke holde generelt for et OODS. Dermed blir det et krav ved tilpasning av tilbakekanselleringsprotokollen for dette formålet at restriksjonen på lengden av innkøen må settes lokalt i hvert objekt. Dette innebærer igjen at ved mottak av en melding i et objekt med fullt buffer, må den meldingen som skal tilbakekanselleres komme fra mottakerobjektet sin innkø. Melding som forårsaket tilbakekanselleringen og meldingen som er offer for hendelsen, kan derimot komme fra ulike objekter.

Ingen andre antagelser trenger store endringer. Dermed er tilbakekanselleringsprotokollen aktuell som metode for minneadministrasjon av fremtiden også innenfor bruk av TBS som parallelitetskontrollmekanisme i et OODS.

6.2 Tidsstempel og synkronisering med Dynamo

La oss se på en kort beskrivelse av *Dynamo*-modellen [13, 27, 25].

Dynamo er en modell som benyttes for å beskrive et OODS der man har synkronisering ved hjelp av tidsstempelbegrepet og samtidig benytter seg av en tilnærming til reell tid[1]. Et av hovedmålene med *Dynamo* var nettopp å innføre begrepet tid på en naturlig måte i et distribuert, objektorientert system[2].

Som vi skal se gir *Dynamo* oss en løsning som kan sees på som et kompromiss mellom *Time Warp* og streng synkronisering ved hjelp av reell tid.

Som vi har sett tidligere kan det være problematisk å benytte seg av en datamaskins klokke for så å stole på at denne alltid inneholder korrekt reell tid. I et distribuert system vil det også være slik at en applikasjon kan eksekvere på mange ulike maskiner i et nettverk, som alle har hver sin klokke som representerer fysisk tid. En streng synkronisering mot fysisk tid gir en lite fleksibel løsning fordi det er kostbart å synkronisere klokkene.

I forbindelse med *Dynamo* innfører man begrepet *kvasireell tid*[3]. Dette er et begrep som ligger et sted *mellom* tidsbegrepet slik det er definert gjennom logiske klokker og vanlig fysisk korrekt tid.

6.2.1 Kort beskrivelse av Dynamo

Dynamo-modellen består av objekter. Disse objektene består igjen av følgende deler:

- en entydig objektidentifikator
- et sett med attributter
- et sett med betingelser
- et *tidsstempel*
- en klokke med *kvasireell tid*

I et objektrom kan det eksistere flere objekter med den samme objektidentifikatoren. De ulike utgavene av et objekt, representerer objektet på flere stadier i prosesseringen. Dette betyr at de ulike objektversjonene må ha ulike verdier i sine kvasireelle klokker.

Denne modelleringen medfører at vi vil ha en versjon av alle objekter ved alle tilstander eller klokkeverdier dette objektet har hatt. Videre innebærer dette at et

[1] Begrepene fysisk tid og reell tid benyttes i dette kapittelet om hverandre.
[2] Her snakker vi da om reell (fysisk) tid.
[3] Kvasireell tid forkortes KRT og er en direkte oversettelse av begrepet Quasi-Real Time (QRT) fra de originale artiklene.

objekts fulle historie med hensyn på alle interne variale eller tilstander, også er lagret. Denne egenskapen ved Dynamo vil da i sin helhet erstatte tilstandskøen slik vi kjenner den fra *Time Warp*. Gjenopprettelse av en gammel tilstand innenfor Dynamo-systemet vil da innebære å fjerne alle objektversjoner med tidsstempel verdi $t_O > T$, der T representerer tidsstempelet i meldingen som forårsaket tilbakerullingen.

Hvert av attributtene i et *Dynamo*-objekt består av:

- et navn
- en verdi, som er et par bestående av en *innverdi* som andre objekter bare kan skrive til, og en *utverdi* som bare kan leses av andre objekter.
- En beskrivelse som spesifiserer typene til attributtets *innverdi* og *utverdi* som kan være ulike. Eventuelt kan man ha to sekvenser med eksekverbar kode, *innkode* assosiert med innverdien og *utkode* assosiert med utverdien.

Innverdien eller utverdien til et attributt kan være av en av tre følgende typer:

- en atomisk dataverdi, slik som integer, boolean, tekststeng og så videre.
- en referanse til et objekt av uspesifisert type.
- en liste med atomiske verdier eller referanser.

Alle deler av et attributt, bortsett fra navnet, kan være tomt.

Som vi ser av definisjonen av systemet er dette et meldingsdrevet system i likhet med *Time Warp*. All kommunikasjon mellom objekter foregår via meldinger, som igjen starter en handlingsrekke hos mottakerobjektet med muligens flere meldingsutsendinger som følge. Alle attributter har en utverdi som kan leses ved å eksekvere objektets *utkode* og en innverdi. Ved skriving til innverdien blir objektets *innkode* eksekvert dersom denne er definert. Slike operasjoner på et objekts attributter gjøres via meldinger. Disse er som vi skal se relativt like meldingene i *Time Warp*.

6.2.2 Meldinger og meldingsutveksling

Meldinger i *Dynamo* er på følgende form:

$$(\mathcal{S}, \mathcal{M}, t_s, t_r, type, attr\_navn, attr\_verdi)$$

De fire første parametrene er henholdsvis identifikator for avsender og mottaker, samt deres tilsvarende tidsstempelverdier. En vanlig melding kan være av typen les (*l*), skriv (*s*) eller les-skriv (*ls*). Et objekt $\mathcal{O}_1$ som sender en melding av type *l* til et annet objekt $\mathcal{O}_2$ vil eksekvere utkoden for attributtet *attr_navn* og lese resultatet. Tilsvarende vil en melding av typen *s* tilordne verdien *attr_verdi* til attributtet *attr_navn* og så eksekvere den tilsvarende innkoden. Meldinger av typen *ls* er en kombinasjon av de to ovenfor.

I tillegg finnes det også en fjerde type meldinger. Denne typen benyttes for å gi svar på *l* og *ls* meldinger. I tillegg kan denne meldingstypen benyttes for å gi kvittering på at meldinger er lest eller mottatt der dette måtte være nødvendig.

I Dynamo eksisterer det ikke noen form for klassedeklarasjoner. Ethvert nytt objekt klones fra et allerede eksisterende et. Objektet som det klones fra fungerer som en prototype for det nye objektet. Det nye objektet vil bli tilordnet en unik identifikator, men vil ellers være en nøyaktig kopi av prototypen.

Som i *Time Warp* har hvert objekt en innkø og en utkø for lagring av meldinger sortert med hensyn på deres t_m og t_s verdier. Meldingsstrømmen i Dynamo er også lik den i *Time Warp* ved at den er asynkron. Vi har altså ingen garanti for at meldinger vil ankomme mottaker i den rekkefølgen de ble sendt.

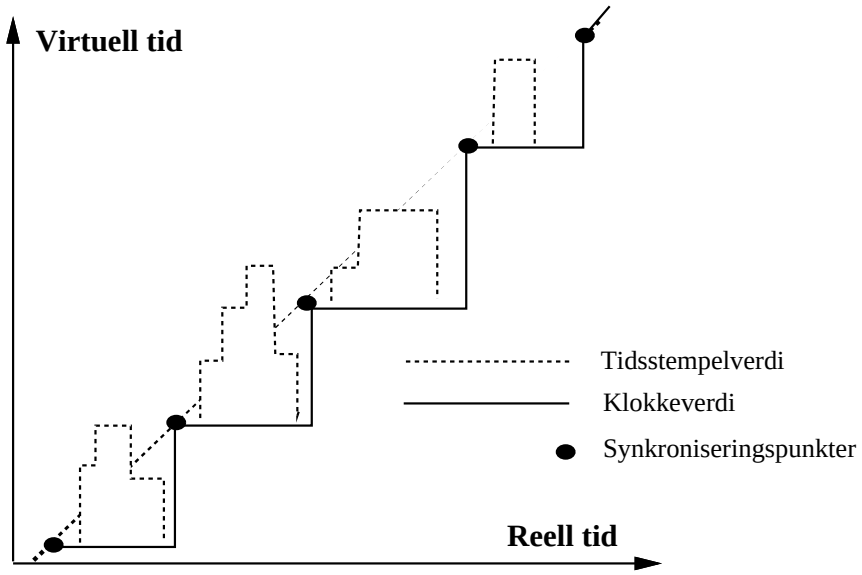
Når det gjelder tilbakerullingsmekanismen i Dynamo er det en viss ulikhet. Et Dynamo-objekt har ingen tilstandskø. I stedet eksisterer det i objektrommet mange ulike versjoner av det samme objektet, med ulik klokkeverdi. Hver av disse objektkopiene representerer en tilstand. Ved tilbakerulling må vi fjerne alle objekter i objektrommet som har klokkeverdi som er større en målet for tilbakerullingen.

6.2.3 Kvasireell tid

Som tidligere nevnt ville man i denne modellen beholde en viss form for binding mellom de virtuelle klokkene i den distribuerte modellen og reell tid. Kort fortalt gjøres dette ved at den virtuelle tiden i modellen synkroniseres mot korrekt reell tid ved visse synkroniseringspunkter. Dette kan for eksempel være dersom et av objektene trenger en slik relasjon til fysisk tid. Mellom disse synkroniseringspunktene tillates objektene å eksekvere med ulik hastighet[4]. Se figur 6.1 for illustrasjon av KRT-begrepet.

I den originale *Time Warp*-modellen benytter man seg av GVT-begrepet. Denne tiden representerer, som vi husker, den laveste tiden som et objekt har i hele det distribuerte systemet.

[4]Noe upresist kan man si at metoden implementerer *Time Warp* mellom alle synkroniseringspunktene. Her kan objekter gjøre tilbakerullinger og så videre slik vi kjenner det fra *Time Warp*. Det vil ikke være mulig for noe objekt og rulle tilbake forbi et synkroniseringspunkt.



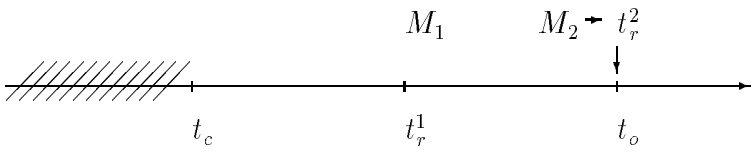
Figur 6.1: Begrepet kvasireell tid

I Dynamo er det slik at den globale virtuelle tiden fra *Time Warp* blir byttet ut med en tilsvarende grense, med to viktige forskjeller:

- GVT byttes ut med en klokkeverdi som viser reell fysisk tid. Denne hentes ut fra en lokal reell klokke i objektet (maskinen).
- Den «nye» GVT-verdien blir desentralisert. I *Time Warp* er dette en sentral verdi som regnes ut av en global mekanisme. I Dynamo er den lokal for hvert enkelt objekt.

I praksis er det slik at den lokale virtuelle klokken vi finner i et *Time Warp*-objekt, byttes ut med to klokkeverdier, en lokal virtuell klokke tilsvarende den i *Time Warp*, samt en klokke med verdi t_{rt} som viser den reelle tiden. Verdien på den sistnevnte klokken vil vise tilnærmet reell tid, men vil tidvis stå stille. Den lokale virtuelle klokken i et objekt vil vise t_m verdien for den meldingen som for tiden er under behandling. Klokken med verdi t_{rt} vil stå stille mens den venter på at en transaksjon skal bli ferdig. Denne ventingen kan for eksempel komme av at en melding har blitt forsinket underveis. Ved avslutning av en transaksjon vil klokken som viser t_{rt} synkroniseres med korrekt reell tid. KRT begrenser altså den forsinkelsen som kan forekomme i slike systemer med en gitt tidsgrense. Dersom denne tidsgrensen overskrides av en transaksjon vil denne bli abortert og KRT synkroniseres med reell tid. I *Time Warp* har vi ingen slik kontroll med forsinkede meldinger.

Ved bruk av KRT vil mottakertidsstempelet t_m angi en øvre grense for når avsenderobjektet *må* ha mottatt en svarmelding med resultatet av transaksjonen. Dersom dette objektet ikke har mottat et svar vil transaksjonen antas å ha feilet og hendelsen, eller transaksjonen, vil bli abortert.



Figur 6.2: Meldingsmottak i et Dynamo objekt

Akkurat som med klokketikking i virtuell tid oppdateres pulsene i kvasireell tid ved avslutning av en atomisk operasjon. Med puls mener jeg her tidsrommet mellom to synkroniseringspunkter med hensyn på reell tid. La oss kalle denne for en transaksjon. Merk at en slik transaksjon som oftest består av flere meldinger. Fra det tidspunktet hvor en slik atomisk operasjon aktiveres, det vil si et objekt mottar en transaksjon, og til det tidspunkt hvor alle operasjoner som tilhører transaksjonen er avsluttet, vil objektet ikke oppdatere sin reelle klokkeverdi. Den reelle tiden står, sett fra objektet sin side, stille. Anta i figur 6.2 at et objekt har mottatt en melding m_2 som stammer fra en transaksjon M_2. Den virtuelle tiden i objektet er derfor satt slik at $t_o = t_r^2$. Som figuren viser har objektet mottatt en ny melding m_1 (fra en transaksjon M_1), med et tidsstempel som er slik at $t_r^1 < t_o$. Dette betyr at vi får en tilbakerulling. Men så lenge den virtuelle tiden til den nye meldingen ikke er mindre enn t_c, så er dette i orden.

Her fungerer KRT, representert ved t_c, nøyaktig slik som GVT gjør i den originale *Time Warp*-modellen. I Dynamo kan det derimot være slik at verdien av t_c kan oppdateres uten at vi er sikre på at ingen nye meldinger kan komme med en lavere virtuell tid. I en slik situasjon ville altså $t_r^1 < t_c$ gjelde. Men KRT skal akkurat som GVT være en grense for hvor langt et objekt har lov til å rulle tilbake i tid innenfor det definerte tidsbegrepet. Dette må medføre at transaksjonen *aborteres* og dette representerer en vesentlig forskjell fra orginal *Time Warp* fordi man her faktisk beregnet GVT ut fra en global tilstandsrapport om hva som faktisk var den sikre tiden i systemet.

Dette er en av de problemene som oppstår når man benytter seg av synkronisering mot reell tid. En slik hendelse som beskrevet over kan komme på grunn av asynkrone, fysiske klokker i datamaskinene, eller som en følge av *timeout*-mekanismen i Dynamo. Denne består kort fortalt av at man kan legge en øvre grense for når en transaksjon skal være ferdig behandlet. Denne *grensen* kommer med meldingen som en parameter, og benyttes for å sikre at avsender får svar tilbake innen et endelig bestemt tidsrom, faktisk endelig med hensyn på fysisk tid. Når en slik grense nås for en transaksjon vil t_c oppdateres uten at man tar hensyn til om alle meldinger er kvittert eller sikre, og transaksjonen aborteres.

6.2.4 Fordeler/ulemper ved Dynamo-modellen

Når vi eventuelt skal implementere en mekanisme for parallellitetskontroll basert på modeller som *Time Warp* og varianter av denne, hva kan vi så lære av Dynamo modellen?

For det første vil jeg si at integreringen av reell tid i modellen har lykkes ganske godt. Man får en kontroll med objektene med hensyn på fysisk tid slik at man kan gi garantier ovenfor brukere eller andre objekter om at en transaksjon skal bli besvart innenfor en gitt tidsgrense. Dersom systemet ikke klarer å behandle transaksjonen kan vi i verste tilfelle få en abortering av transaksjonen, men responstiden blir overholdt. Dette virker også som den mest gjennomtenkte delen av modellen, og var også hovedmålet.

Fjerningen av tilstandskøen gjennom innføring av objektversjoner for hver klokkeverdi virker bra i den teoretiske modellen, men her ser jeg muligens problemer når man skal ta dette med seg over i en implementasjon. Dersom man for eksempel har en implementasjon der et objekt opptrer som en prosess, som ikke ville være unaturlig, ville man ganske raskt kunne fylle for eksempel prosesstabellen på en normal UNIX[5] maskin. Dette vil selvfølgelig kunne begrenses ved å innføre **fossilfjerning** etter hver endt transaksjon. Alle objektversjoner med tid eldre enn den nye tiden til objektfamilien kunne så fjernes. Administrasjon av et slikt antall objekter vil etter min mening kunne bli noe stor. En løsning som i *Time Warp* med tilstandskø av bare de nødvendige variable vil være å foretrekke. Dette fordi man da bare lagrer de verdier som er absolutt nødvendig for å gjenopprette en gammel tilstand, og ikke noe annet.

Gruppering av meldinger som typisk hører sammen til en transaksjon sett fra avsender er en god tilnærming til hva jeg vil vente å finne i noen implementasjoner. Spesielt i databaser der transaksjonsbegrepet er velkjent. Tanken med å koble en slik transaksjon til synkroniseringspunkter mot reell tid ved oppstart og avslutning av transaksjonen er god.

Systemet kan være veldig følsomt med hensyn på effektivitet. Dersom man setter små verdier på *timeout*-grensen vil man øke sannsynligheten for abortering av transaksjoner, men oppnå lav responstid. Hvis man ønsker garantier for lav aborteringsfrekvens må man tilsvarende sette denne tidsgrensen større. Responstiden vil da øke tilsvarende. Denne mekanismen innfører dermed også et behov for *tuning* eller finjustering av systemet. Hvordan en slik tidsgrense velges kan altså være kritisk for effektiviteten av systemet.

I systemer som har en interaktiv dialog med brukere gir denne løsningen garantier til en bruker i likhet med tradisjonelle synkroniseringsmekanismer med hensyn

[5] UNIX is a trademark of AT&T Bell Laboratories.

på responstid og lignende. I tillegg vil fleksibiliteten til *Time Warp* implementeres internt i systemet.

I store systemer med interaksjon mot for eksempel en distribuert database, vil antallet meldinger som blir generert pr. transaksjon lett kunne bli relativt stor, og i slike tilfeller vet vi nå at *Time Warp* tilbyr en behagelig distribuert synkroniseringsmekanisme.

6.3 LVT-beregning i TBS

Time Warp [17, 18] fungerer godt som synkroniseringsmekanisme innenfor distribuert simulering. Dette er vist og skrevet om i mange artikler innenfor feltet se blant annet [29, 31, 26, 11, 14].

Når vi skal se på metoden som et verktøy for å utføre parallellitetskontroll i et OODS er det imidlertid to ting ved metoden som må forandres og utdypes:

- Metoden har ingen referanser til reell tid. I mange tilfeller er dette essensielt i denne type synkronisering.
- Metodene som er presentert for estimering av GVT er mangelfulle. De slår bare fast at det er enkelt å beregne *LVT* lokalt for hver node[6]. Algoritmene opererer så på nodenivå. Som tidligere fastslått vil det i systemer som ANSAware ikke lengre være trivielt å beregne en slik *LVT* fordi man kan ha relativt komplekse strukturer lokalt på hver node[7].

Det første problemet er, som jeg har beskrevet, forsøkt løst i Dynamo (kapittel 6.2). Som en konsekvens av dette løser man også noen av problemene rundt GVT. KRT[8] er en kloning mellom streng fysisk tid og virtuell tid. Jeg trenger nå å diskutere en metode for *LVT*-estimering. Jeg har tidligere påpekt at samtidigheten som ligger i å beregne GVT utfra de gamle forutsetninger[9] gir en ellers så distribuert modell et noe negativt preg. Det er derfor en av mine forutsetninger når jeg beskriver denne algoritmen at *LVT*-beregning gjøres asynkront og ikke som en del av selve GVT-estimatalgoritmen.

Som nevnt i punkt to ovenfor er GVT-algoritmene mangelfulle. Fra kapittel 3.3.2 og 3.3.3 ser vi at algoritmene som presenteres arbeider på høyt nivå mellom nodene eller maskinene i et OODS. Imidlertid er det slik i nye implementasjonsverktøy

[6] LVT er en forkortelse for «lokal virtuell tid». Se kapittel 3.3.1 for diskusjon av begrepet. LVT er definert i definisjon 3.3.5.

[7] ANSAware systemet beskrives i kapittel 8.

[8] KRT er en forkortelse for Kvasireell tid. Se kapittel 6.2

[9] Det vil si at man også finner *LVT* lokalt i nodene som del av GVT-algoritmen

at strukturen på hver node også kan bli relativt kompleks. I ANSAware kan man ha mange kapsler på hver node, og hver kapsel kan inneholde mange objekter. I et slikt system er ikke lengre den lokale beregningen av *LVT* triviell. Vi trenger derfor en lokal algoritme for hver node og for hver kapsel som kan tilby *LVT*-beregning til den globale GVT-algoritmen når dette er aktuelt. Motivasjonen for δGVT-algoritmen som presenteres nedenfor er å kunne tilby en slik *LVT*-beregning innenfor hver kapsel i systemet. Poenget er nå at denne beregningen foregår kontinuerlig innenfor kapselen. En sentral og global GVT-estimatalgoritme vil da ikke involvere objektene, bare en mekanisme i hver kapsel som hele tiden er oppdatert med hensyn til *LVT*.

6.3.1 δGVT-algoritmen

La meg først definere de ulike elementene i dette systemet:

Definisjon 6.3.1 *Et distribuert, objektorientert system som benytter distribuert GVT har følgende struktur:*

- *Et antall noder* $N_i, i \in 1 \ldots n, n >= 1$. *Hver node* N_i *har et antall kapsler* $K_j, j \in 1 \ldots k, k >= 1$. *Hver kapsel har så et antall objekter* $O_k, k \in 1 \ldots o, o >= 1$.
- *Hver kapsel har et kommunikasjonssenter* $\mathcal{KO}_\mathcal{K}$. *Dette sentret* ser *alle meldinger som passerer til eller fra alle objekter* O_k *lokale til kapselen.*

Objektene skal se ut som de gjør i den originale modellen [17]. Vi skal nå la de ulike meldingene som sendes rundt i systemet være budbringere også om *LVT*. Merk at dette vil komme automatisk fordi $\mathcal{OK}_K$ *ser* alle meldinger som sendes fra et objekt innenfor kapselen. Hvert enkelt objekt har en egen *oppfattelse* av hva som er det korrekte GVT-estimatet. Det vil ikke her være slik at alle objekter har lik oppfattelse av GVT.

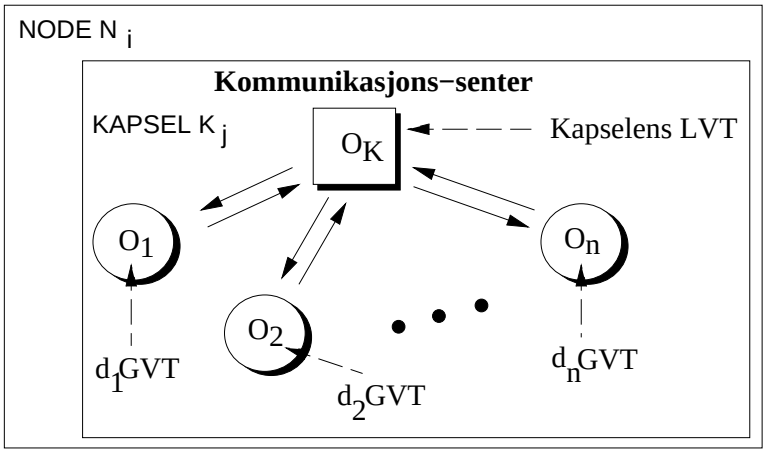
La et gitt objekts oppfattelse av GVT være definert ved:

Definisjon 6.3.2 *Et gitt objekt* $\mathcal{O}_k$ *sin oppfattelse av GVT kalles distribuert GVT, forkortet* $\delta\mathcal{GVT}$.

Ettersom GVT, eller estimert GVT, fortsatt skal representere den største *sikre* virtuelle tiden i systemet, følger det da umiddelbart fra definisjonen:

Lemma 6.3.1 *Det er alltid slik at* $\delta\mathcal{GVT} \leq \mathcal{GVT}$

Følgende definisjon kan settes opp for meldingene i et slikt system:



Figur 6.3: Kommunikasjon mellom objektene gjennom et kommunikasjonssenter

Definisjon 6.3.3 *Alle meldinger i systemet skal være på formen:*

$\mathcal{M} = (t_s, t_r, \mathcal{O}_s, \mathcal{O}_r, \mathcal{D}, \delta\mathcal{GVT}_s)$

De ulike parametrene har følgende betydning:

- t_s står for den virtuelle avsendertiden. Dette er den virtuelle tiden som gjelder i avsenderobjektet da meldingen ble sendt.
- t_r betyr den virtuelle mottakertiden. Akkurat som i *Time Warp* må det være slik at $t_s < t_r$.
- $\mathcal{O}_s$ er en unik identifikator for avsenderobjektet.
- $\mathcal{O}_r$ er en unik identifikator for mottakerobjektet.
- $\mathcal{D}$ er en parameter av uspesifisert type. Denne benyttes for dataoverføring mellom objektene.
- $\delta\mathcal{GVT}_s$ representerer avsenderobjektets *oppfattelse* av GVT, den distribuerte GVT.

Kommunikasjonsmodellen i sin helhet kan så visualiseres som i figur 6.3. Alle objekter har to kommunikasjonskanaler til $\mathcal{KO}_\mathcal{K}$. En for sending av meldinger og en for mottak. Hvorvidt dette skal implementeres som to uavhengige grensesnitt blir en smakssak, men vil være den løsningen som gir best parallellitet.

For at dette systemet skal starte riktig må alle objekter *registrere* seg hos kommunikasjonssenteret $\mathcal{KO}_\mathcal{K}$ før de starter eksekveringen. Dette kan gjøres ved at alle objekter $\mathcal{O}_k$, sender en registreringsmelding ved oppstart. Kommunikasjonssenteret vet nå til enhver tid hva som er den tilnærmet korrekte LVT ved at den leser

av alle avsendertidspunkt for alle innkomne meldinger og vedlikeholder en tabell med de ulike objektenes oppfattelse av GVT. Vi må så la dette objektet avgjøre når et annet objekt må foreta fossilfjerning og oppdatere sin egen $\delta\mathcal{GVT}$. Dette gjøres dersom et objekts $\delta\mathcal{GVT}$ ligger et predefinert antall N virtuelle tidsenheter etter den estimerte GVT. En melding kan så sendes fra kommunikasjonssenteret til objektet med beskjed om å oppdatere seg. En annen mulighet er at $\mathcal{KO}_\mathcal{K}$ konsekvent distribuerer $\mathcal{OGVT}$ ut til objektene innenfor sin kapsel via $\delta\mathcal{GVT}_s$-parameteren i meldingene.

Dette resulterer i en algoritme som er meget distribuert i den forstand at vi kan få, eller vil høyst sannsynlig få, oppdatering av $\delta\mathcal{GVT}$ med fossilfjerning til forskjellige tider ute i de ulike objektene. Et av mine ankepunkter mot antagelsene i de to GVT-algoritmene som er presentert er at de krever samtidig deltagelse fra alle objektene for å fungere. Når et objekt mottar en melding om at beregning av GVT er igang, må denne meldingen behandles «umiddelbart» for at algoritmene skal fungere.

Resultatet av dette er at en distribuert og asynkron *LVT*-algoritme slik den er beskrevet ovenfor automatisk kan utføres gjennom $\mathcal{KO}_\mathcal{K}$ siden alle meldinger, og dermed tilstrekkelig informasjon for å beregne *LVT*, passerer gjennom dette senteret. Merk at det senteret i systemet med minst oppfattelse av *LVT* ikke er tilsvarende den korrekte GVT, men den vil alltid være mindre og «ikke langt bak» den korrekte. Den estimerte GVT i $\mathcal{KO}_\mathcal{K}$ følger definisjon 3.1.2. Jeg kaller derfor den estimerte GVT for $\mathcal{OGVT}$. Grunnen til at dette er et estimat kan illustreres i følgende eksempel:

Eksempel 6.3.1 *Anta at objekt $\mathcal{O}_k$ ved reell tid r_k har virtuell klokketid t_k og at $GVT = t_k$. Anta så at $\mathcal{O}_k$ sender en melding gjennom kommunikasjonssenteret $\mathcal{KO}_\mathcal{K}$ ved reell tid r_k, og leser nye meldinger fra sin innkø. Dette objektet vil så oppdatere sin virtuelle klokke til en høyere verdi enn t_k og mottakerobjektet vil ikke under noen omstendighet rulle lengre tilbake i tid enn $t_k + 1$ siden dette er minimum for mottakerverdien i meldingen*[10]*. Kommunikasjonssenteret $\mathcal{KO}_\mathcal{K}$ vil nå sette* LVT *lik avsendertiden i meldingen, men denne blir ved reell tid $r_k + \Delta r$ noe for lav. Altså er oppfattelsen i $\mathcal{KO}_\mathcal{K}$ av* LVT *et estimat av GVT og noe lavere.*

Fra definisjonen ovenfor kan vi direkte sette opp en observasjon som sier at:

Observasjon 6.3.1 *Det er alltid slik at $\delta\mathcal{GVT} \leq \mathcal{LVT} \leq \mathcal{OGVT} \leq \mathcal{GVT}$*

I en slik modell vil objekter også kunne komme til etterhvert uten at dette vil skape problemer for systemet. Slik er det også i den originale *Time Warp* [17]. Fra [26] vet vi at dette ikke nødvendigvis gjelder for andre *Time Warp*-varianter, men ved at et

[10] I [17] sies det at mottakertiden i en melding må være minst en tidsenhet større enn avsendertidspunktet.

objekt registrerer seg hos $\mathcal{KO}_K$ ved oppstart vil dette starte sin eksekvering med riktig $\delta\mathcal{GVT}$ og dermed også «riktig» klokketid, og kan ikke gi oss problemer med hensyn på virtuell tid i det nye objektet i forhold til resten av systemet.

Anta at et nytt objekt kommer til under eksekveringen. Objektet sender da først en registreringsmelding til kommunikasjonssenteret $\mathcal{KO}_K$. Dette må så sende en svarmelding som inneholder $\mathcal{OGVT}$. Det nye objektet setter så $\delta\mathcal{GVT} = \mathcal{OGVT}$ og eksekveringen kan så tas opp med virtuell klokke tid lik $\delta\mathcal{GVT}$.

6.3.2 Kanselleringsstrategier og $\delta\mathcal{GVT}$-algoritmen

Det eneste punktet som nå trenger litt diskusjon er kanselleringsmekanismen ved tilbakerulling i et slikt system. Ettersom ulike objekter kan ha ulike oppfattelse av GVT vil de også være på forskjellige stadier med hensyn på **fossilfjerning**. Dette innebærer at teorem 3.1.1 fra side 81null-teorem ikke holder for dette systemet. Dette sier at den totale meldingssummen av meldinger i systemet til enhver tid er lik null[11]. Egenskapen følger direkte av at man har en GVT algoritme av global karakter. Objektene vil gjøre GVT oppdatering og *fossilfjerning* samtidig med hensyn på en sammenhengende reell tidsperiode. I min modell med distribuert GVT vil det derimot med stor sannsynlighet være slik at $\mathcal{M}_t = \sum_i \oplus m_i + \sum_j \ominus m_j \neq 0$ for ethvert tilfeldig valgt virtuelt tidspunkt t i eksekveringen.

Jeg må derfor vise at dette ikke vil skape problemer for konsistensen i systemet. Hvis jeg går mekanismen litt nærmere etter i sømmene ser jeg at det som kan skape problemer er det faktum at en antimelding kan bli sendt ut som følge av en tilbakerulling uten at noen tilsvarende positiv melding finnes i fortidsdelen av innkøen til mottakerobjektet.

Men jeg kan lett vise at dette ikke vil kunne gi problemer. Faktisk vil slike antimeldinger aldri kunne bli sendt ut i systemet ved tilbakerullinger. La meg gi en definisjon før jeg setter opp et teorem som nettopp sier dette, og så gi beviset.

Definisjon 6.3.4 *En antimelding $\ominus m_i$ i et objekt O_j sin innkø, sies å være **inaktivt** dersom virtuell mottakertid t_m i meldingen er mindre enn $\mathcal{OGVT}$ hos $\mathcal{KO}_K$. Alle andre antimeldinger sies å være **aktive**.*

Et objekts inaktive antimeldinger vil alltid være samlet i den eldste delen av objektets innkø. La oss så sette opp teoremet som sikrer oss mot utsendelse av slike antimeldinger:

Teorem 6.3.1 *En **inaktiv** antimelding $\ominus m_i$ i innkøen til et objekt O_j vil aldri sendes ut ved en tilbakerulling i objektet.*

[11] Det vil her si ved ethvert virtuelt tidspunkt.

Som teoremet sier er det slike *inaktive* antimeldinger som kan skape problemer. Disse vil bare være farlige dersom de blir sendt ut i systemet ved en eventuell tilbakerulling. Det er derfor nok å vise at slike meldinger ikke vil bli sendt ut, men vil ligge i utkøen til objektet inntil de blir fjernet ved *fossilfjerning*.

Bevis 6.3.1 *Anta O_j har utkø $\ominus m_i, \ominus m_{i+1}, \ldots, \ominus m_{i+k}$ og at $\mathcal{OGVT}$ i $\mathcal{KO}_K$ er større enn mottakertiden t_m i $\ominus m_i$. Vi har da en situasjon der alle meldinger $\ominus m_{i+l}, l = 0 \ldots m, m \leq k$ med mottaker tid $m_{i+m} < \mathcal{OGVT}$ er inaktive. En start av tilbakerulling vil måtte komme som et resultat av en melding som mottas med $t_m <$ virtuell klokketid i objekt O_j. Men vi vet at $t_m \geq \mathcal{OGVT}$ siden dette er en av premissene for oppdatering av $\mathcal{OGVT}$. Det finnes ingen objekter i systemet med virtuell klokketid mindre enn $\mathcal{OGVT}$. Dermed må alle meldinger som sendes ut fra et hvilket som helst annet objekt i systemet ha egenskapen $t_m \geq \mathcal{OGVT} + 1$, og ingen inaktive antimeldinger vil følgelig bli sendt, rett og slett fordi ingen objekter kan sende meldinger som vil gi tilbakerullinger så langt tilbake i virtuell tid.* □

Dette viser nettopp at inaktive antimeldinger ligger uvirksomme i utkøen inntil objektet fjerner de ved **fossilfjerning**. Inaktive antimeldinger vil ikke kunne gi problemer for et objekt selv om dette har fjernet den tilsvarende meldingen fra sin innkø.

Resultatet gir oss da muligheten til fritt å kunne velge hvilken kanselleringsstrategi vi ønsker uten at problemer vil oppstå i modellen. Både aggressiv og lat kansellering vil fungere som forventet og antydningen om mulighet for valg av kanselleringsstrategi slik det blir antydet i kapittel 3.2, blir like aktuell selv om man benytter $\delta\mathcal{GVT}$-algoritmen til asynkron LVT-beregning.

Kapittel 7

Standardisering og implementasjon

TBS-metodene er tilpasset generelle, objektorienterte modeller der vi har et symmetrisk klient/tjener forhold mellom objektene. Jeg vil her benytte objektorienterte programmeringsteknikker for å definere et standardisert rammeverk med *utskiftbare* deler for forskjellige varianter av TBS.

Dersom vi ser slike TBS-objekter i sammenheng med dagens implementasjoner av distribuerte systemer så ligger den største forskjellen i at alle objektene i en slik modell tenkes å være selvstendige objekter. Disse skal fungere som klienter mot og tjenere for den samme tjenesten samtidig. Dagens objekter er derimot som regel fast implementert i en rolle som enten klienter eller tjenere.

I en fullstendig objektorientert modell slik jeg tenker den her, vil alle objektene fungere vekselvis eller også parallelt som tjenere eller klienter overfor hverandre eller overfor en interaktiv bruker av systemet. Det er altså ønskelig med en mye mer fleksibel modell en den som pr. idag er mest benyttet når det gjelder implementasjon av distribuerte systemer.

7.1 Fellestrekk for alle tidsstempelbaserte, objektorienterte modeller

Alle datamodeller som benytter seg av tidsstempelbasert synkronisering i en eller annen form, har mange fellestrekk. For å kunne implementere tidsstempelbasert synkronisering i en eller annen kontekst, må man følge visse spilleregler som ligger i modellens natur.

I denne oppgaven ser jeg spesielt på bruk av slik synkronisering til parallellitetskontroll i distribuerte, objektorienterte systemer, hvilket i tillegg vil gi enda flere retningslinjer man må følge ved en eventuell implementasjon. For det første er det

jo slik at alle objektene i følge *Time Warp*-modellen (se [17]) skal inneholde visse standardattributter som er helt nødvendige for effektivt å kunne implementere synkroniseringsmetoden. Hvert objekt skal ha en egen innkø, utkø og så videre.

Når man skal implementere en slik synkroniseringsmekanisme i praksis vil det være behagelig om man har tilgang til en ressurs der alle slike basale tidsstempelmekanismer allerede er implementert i objektene. Slike objekter kan så standardiseres i grensesnittet, og applikasjoner som blir bygget rundt en slik ressurs kunne benytte seg av de tjenestene som slike standardiserte objekter tilbyr.

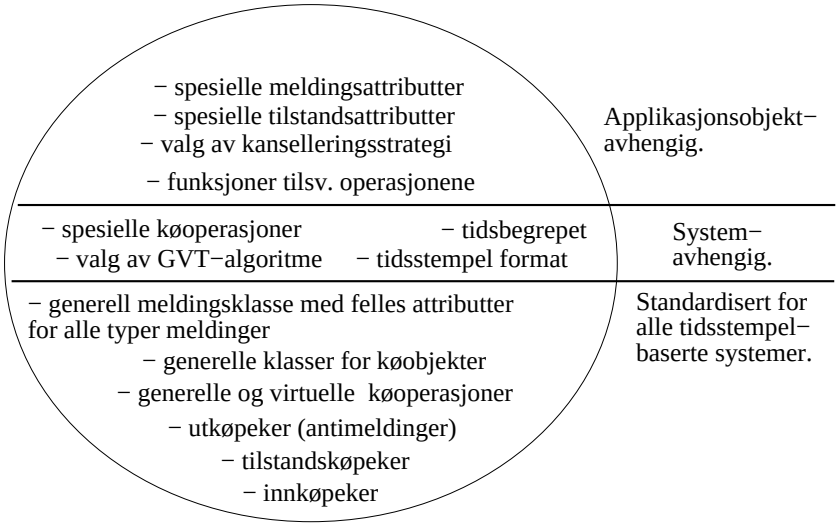
Hvilket verktøy som skal benyttes for å implementere en slik standardisert ressurs er ikke så viktig. Det viktige er at man tilbyr en konsistent måte å bygge opp nye objektorienterte applikasjoner på, og at man tilbyr et konsistent grensesnitt inn mot de basale funksjoner som må ligge i slike objekter[1]. Hvilke funksjoner kan man så legge inn i slike standardiserte objekter uten at det legges altfor store begrensninger på fleksibiliteten for en eventuell bruker som skal programmere en applikasjon? I utgangspunktet høres det ut som om mekanismene for å administrere de ulike lokale tidene i objektene kan overlates til objektene selv. Dette vil typisk innebære oppdatering av den lokale virtuelle klokken hver gang en ny melding blir behandlet og så videre. Som vist i to eksempler i kapittel 4 kan dette være vanskelig å standardisere på en entydig måte.

Effektiviteten i metoden kan forbedres ved *intelligent* valg av tidsstempelformat. Man kan ha løsninger der man bare benytter seg av enkle heltallstellere som slike tidsstempelverdier, eller man kan ha mer komplekse, sammensatte verdier. Valg av slike verdier er avgjørende fordi man kan benytte dem til å automatisere en intelligent køing av innkomne meldinger til et objekt, og dermed redusere sannsynligheten for at tilbakerulling i systemet skal inntreffe. Et eksempel på en slik intelligent, automatisk køing av innkomne meldinger er vist i kapittel 4.2 som tar for seg en mulig modell for implementasjon av tilgang til en felles database som normalt ville vært løst via for eksempel låsing.

Av dette kan vi så lese at man skal vokte seg vel for hvor mye man skal si angående valg av tidsstempelverdier i en eventuell standardisert, objektorientert modell. I hvertfall må det ikke legges restriksjoner på formatet til tidsstempelverdiene, og dermed heller ikke på hvordan de ulike klokkene i objektene skal *tikke* forover i tid.

En mulig løsning på problemet er at man lar implementasjonen angi et sett med regler for hvordan en slik tidsstempelverdi skal se ut, hvilke kriterier meldinger skal sorteres etter på basis av disse verdiene, og hvordan de ulike lokale virtuelle klokkene skal *tikke* fremover i tid. Selve mekanismene for å administrere køene og å motta eller sende meldinger, kan så standardiseres innenfor objektmodellene.

[1] Les her mekanismer for å implementere egenskapene abstrasjon, innkapsling og polymorfi i henhold til definisjonen av objektorientering i definisjon 2.1.8.



Figur 7.1: Standardisert objekt.

Dette blir et kompromiss mellom to ytterpunkter, men dersom man for eksempel standardiserer bruk av vanlig heltalls, flerdimensjonale og kvasireelle klokker som i Dynamo, skulle de mest interessante muligheter etter min mening være dekket. I tillegg vil utskiftbarheten gjøre at en applikasjonsprogrammerer kan redefinere standardobjektet til å passe sitt formål gjennom polymorfi. En annen ting som også selvfølgelig må overlates til hver enkelt implementasjon er hvilke dataattributter som hvert enkelt objekt skal ha, og hvilke atomiske operasjoner som skal gjøres på disse. Dette kan for eksempel implementeres ved at de standardiserte objektene gis en liste med operasjoner og en tilsvarende liste med funksjoner i hver enkelt implementasjon. De ulike operasjonene sendes så ut i meldinger og objektene tar seg av å kalle den tilsvarende funksjonen med de korrekte dataattributtene. Disse kan enten gis verdier gjennom meldingen eller hentes ut fra de interne attributtene i objektene. Innholdet i de ulike funksjonene blir således helt implementasjonsavhengig og er helt overlatt til brukeren å spesifisere. En implementasjon må også spesifisere hvordan man skal beregne GVT i systemet. Dette kan for eksempel gjøres gjennom en enkelt funksjon som implementasjonen angir, og som kalles med jevne mellomrom. I praksis vil det være helt nødvendig med en slik algoritme fordi man ikke kan spare objekthistorien fra hele eksekveringen av hensyn til minneforbruk.

De to angitte GVT-algoritmene bør tilbys som alternativer. Innenfor systemer som ANSAware[2] kan man også eventuelt vurdere $\delta\mathcal{GVT}$-algoritmen for LVT-beregning.

Hva så med administrasjon av objekthistorien til hvert enkelt objekt, og mekanismene for hva som skal gjøres dersom tilbakerulling innenfor et eller flere objekter forekommer? Selve objekthistoriene kan man relativt enkelt implementere innen-

[2] Se kapittel 8 for en kort gjennomgang av systemet.

for rammene av det standardiserte objektet. Dette er jo i realiteten bare en kopi av hver tilstand objektet har befunnet seg i fra start til nåtid, eller i praksis fra GVT og frem til nåtid for det aktuelle objektet, se [17]. Det vil si at man bare trenger å lage en kopi av alle verdier som de interne dataattributtene har hver gang objektets lokale virtuelle klokke *tikker en verdi frem*. Siden dette vil inntreffe nøyaktig hver gang objektet henter og behandler en ny melding fra sin innkø og attributtene og deres verdier og typer er spesifisert i implementasjonen, bør dette kunne standardiseres.

Et slikt valg vil allikevel føre til at man lagrer flere interne variable enn nødvendig for å kunne gjenopprette gamle tilstander. Alternativet er å la brukeren spesifisere hvilke variable som er kritiske og må lagres. Hva som er mest fornuftig diskuteres senere.

Figur 7.1 viser en generell modell for hvordan et slikt objekt kan se ut. Det er angitt hvilken funksjonalitet som kunne tenkes å være applikasjonsobjektavhengige, systemavhengige og hvilke som kan tenkes implementert som en del av objektet. Figur 7.2 viser de abstrakte datatypene i et TBS-system som klassehierarkier.

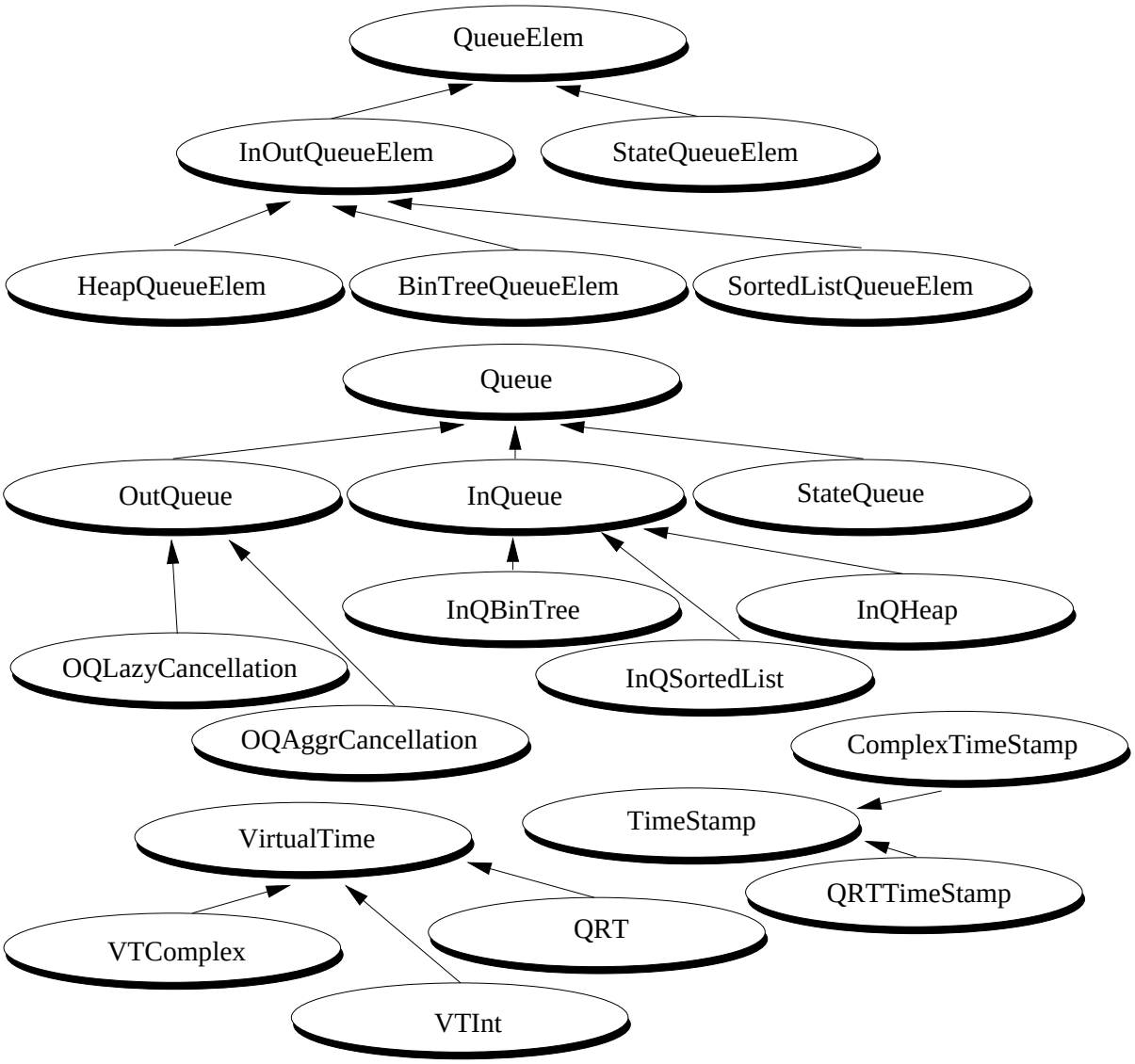
7.2 Implementasjon av de standardiserte elementene

De ulike elementene som inngår i en slik standardisert utgave av metoden må velges slik at de best passer sammen med de ulike typer av TBS-modeller som kan tenkes benyttet. Nedenfor diskuteres ulike aspekter ved en slik standardisering for de viktigste elementene. Aspekter rundt implementasjon av disse køene er diskutert i litteraturen (se [29]) og disse resultatene diskuteres i kapittel 5.1.1, men noen punkter jeg føler litt dårlig belyst diskuteres nedenfor. Artikkelen som er nevnt, diskuterer i hovedsak erfaringer og ideer rundt implementasjon av innkøen. De andre elementene som tas opp diskuteres i den nevnte litteraturen bare som en konsekvens av innkøimplementasjonen.

Rammeverket som presenteres nedenfor er ikke fullstendig. *Kvasikode* basert på Simula er benyttet for å presentere modellen i figur 7.2. Det blir ikke presentert mer kode enn det jeg føler er nødvendig for å kommentere hvilke mekanismer som kan implementeres i standarden, og hvilke som må overlates til applikasjonsobjektene.

7.2.1 Abstrakte datatyper for køelementer, meldinger og virtuell tid

De tre køene som skal implementeres skal benyttes seg av abstrakte datatyper som tar hensyn til de ulike køstrukturer som kan benyttes. Innkøen sin fremtidsdel kan være av mange ulike typer. Utkøen og tilstandskøen bygges opp som like strukturer, det vil si lineære, sorterte lister. Av ulike grunner bør disse, som vi skal se



Figur 7.2: De abstrakte datatypene i standardisert TBS.

senere, være dobbelt linkede lister. Videre er det slik at innkøen og utkøen skal køe meldinger samtidig som tilstandskøen skal køe tilstander. La oss samle alle disse ulike datatypene i en felles standardisert struktur. Merk at køene referer til klassen **TimeStamp**. Denne defineres litt senere i kapittelet:

```
class QueueElem
begin
end

QueueElem class InOutQueueElem
begin
  ref (Message) message;
end

QueueElem class StateQueueElem
begin
  ref (TimeStamp) at_time;
  ref (StateQueueElem) next,prev;
end

InQutQueueElem class HeapQueueElem
begin
  ref (HeapQueueElem) left, right;
end

InQutQueueElem class SortedListQueueElem
begin
  ref (SortedListQueueElem) next, prev;
end

InQutQueueElem class BinTreeQueueElem
begin
  ref (BinTreeQueueElem) left, right;
end
```

Merk her at superklassen for denne strukturen av datatyper er tom. Ingen interne attributter i køelementene knytter tilstandskø og innkø/utkø sammen. Poenget er å standardisere købegrepet på en naturlig måte. En annen fordel er også at man kan samle alle køoperasjoner slik at man gjør et høynivåkall på en kø. Inne i funksjonene/prosedyrene kan man så skille mellom typene ved å benytte mekanismer som for eksempel **is**-notasjonen fra Simula på pekeren til superklassetypen **QueueElem**.

Merk også at **StateQueueElem** ikke inneholder annet enn pekere for å implementere en dobbelt linket liste samt tiden for tilstanden. Det blir dermed applikasjonen

sin oppgave å benytte **StateQueueElem** som supertype for en egendefinert tilstandsklasse. Her foretar vi et valg som sier at vi bare skal lagre de attributter som er nødvendige for å gjenopprette en gammel tilstand. I det motsatte tilfellet der vi lagrer hele innmaten til et objekt må vi benytte mekanismer som er kommentert i kapittel 6.2 gjennom Dynamo-modellen. Her vil man ha kopier av objekter liggende i objektrommet.

Den abstrakte datatypen **Message** er utelatt fra figuren fordi den innenfor stadardisert TBS ikke vil operere med noen subtyper. Merk at slike subtyper må defineres for hver applikasjon fordi vi for eksempel ikke vet hvilke dataparametere meldingene i en spesiell applikasjon skal inneholde. Det som i praksis kan tas inn i en superklasse er de fire predefinerte attributtene avsendertid, mottakertid, avsenderidentifikator og mottakeridentifikator. En Simula-aktig klasse med definisjon av disse attributtene vil for eksempel se slik ut:

```
class Message
begin
  text SendIdentificator, ReceiveIdentificator;
  ref (TimeStamp) SendTime, ReceiveTime;

  integer TotalOrderId;
end
```

Merk at i tillegg til de fire attributtene nevnt ovenfor har jeg tatt med en enkel heltallsvariabel. Denne kan benyttes dersom man i en applikasjon ønsker å definere en total ordning av de ulike meldingstypene i systemet. Denne ordningen kan så sammen med flerdimensjonale tidsstempler benyttets for intelligent køing i henhold til definisjon 4.2.1..

Tilsvarende situasjonen med **StateQueueElem**-klassen, vil en standardisert meldingsklasse måtte benyttes som superklasse for applikasjonens egendefinerte meldinger. Alle attributter som skal benyttes til dataoverføring mellom objekter defineres her.

Standardiseringen av tidsstemplene blir som vi skal se, relativt analogt med standardiseringen av virtuell tid og kvasireell tid. La oss først kikke på den abstrakte datatypen for tidsstempelbegrepet:

```
class TimeStamp
begin
  integer pre_stamp;
end

TimeStamp class ComplexTimeStamp
```

```
begin
  integer array post_stamp(1:MAX_TIMESTAMP_DIMENSIONS);
end

TimeStamp class QRTTimeStamp
begin
  integer limit;
end
```

Standardisering av virtuell tid gjøres på samme måte som med de ulike elementene beskrevet ovenfor. Meningen med dette er å tilby en enhetlig datatype for utenforstående. Klassen **VirtualTime** kan tenkes på som abstraksjon av *hele* den virtuelle tidsaksen for eksekveringen. Datatypen har så en peker inn på denne aksen som forteller hvor vi befinner oss, samt virtuelle operasjoner for å bevege pekeren langs den virtuelle tidsaksen. Disse operasjonene kan så definerers som virtuelle prosedyrer i superklassen. Applikasjoner kan dermed programmeres mot **VirtualTime** klasser uten at vi vet nøyaktig hvilken type som er valgt:

```
class VirtualTime
  virtual:
    procedure TickTack;
    procedure SetClock (m); ref (Message) m;;
    TimeStamp procedure has_value;
    procedure initialize;
    boolean procedure is_larger (m); ref (Message) m;;
begin
  protected integer Clock;
end

VirtualTime class VTInt
begin
  TimeStamp procedure has_value
  begin ref (TimeStamp) t;
        t:-new TimeStamp;
        t.pre_stamp := Clock;
        has_value:=t;
  end
  procudure TickTack;
  begin Clock:=Clock + 1;
  end
  procedure SetClock (m); ref (Message) m;
  begin Clock:=m.ReceiveTime.pre_stamp;
  end
  procedure initialize;
```

```
  begin
    Clock:=0;
  end
  boolean procedure is_larger (m); ref (Message) m;
  begin if m.ReceiveTime.pre_stamp >= Clock then is_larger:=true
        else is_larger:=false;
  end
end

VirtualTime class VTComplex
begin
  protected integer array post_stamps(1:MAX_TIMESTAMP_DIMENSIONS);

  TimeStampComplex procedure has_value
  begin ref (TimeStampComplex) t;
        t:-new TimeStampComplex;
        t.pre_stamp := Clock;
        <t.post_stamps := post_stamps>
        has_value :- t;
  end
  procedure SetClock (m); ref (Message) m;
  begin Clock:=m.ReceiveTime.pre_stamp;
        for i:=1 step 1 until MAX_TIMESTAMP_DIMENSIONS
        begin post_stamps(i):=m.ReceiveTime.post_stamps(i);
        end
  end
  procedure TickTack
  begin if <all post_stamps reached top value>
        then begin Clock:=Clock+1; <initialize all post_stamps> end
        else begin <Tick post-stamps> end
  end
  procedure initialize;
  begin
    Clock:=0;
    for i:=1 step 1 until MAX_TIMESTAMP_DIMENSIONS do post_stamps(i):=0;
  end
  boolean procedure is_larger (m); ref (Message) m;
  begin if m.ReceiveTime.Clock > Clock then is_larger:=true;
        else if m.ReceiveTime.Clock < Clock then is_larger:=false;
        else if <m.post_stamps > post_stamps> then is_larger:=true;
        else is_larger:=false;
  end
end
```

Dette gjøres fordi TBS-metoder generelt ikke skal berøres av om vi har valgt kompleks struktur på våre tidsstempler eller ikke. Selv om kvasireell tid her defineres som en subtype av virtuell tid er det viktig å huske på forskjellene. *pre_stamp* er egentlig her en klokke som viser reell tid, men denne kan jo enkelt representeres som en heltallsvariabel.

Merk at skillet mellom virtuell tid og kvasireell tid blir like markant her som for tidsklassene. Merk også at den abstrakte typen **Message** også kan benyttes som en Dynamo-melding og en Dynamo-transaksjonsmelding. Siden en Dynamo-transaksjon skal ha to tidsparametre som gir henholdsvis en reell starttid for transaksjonen og en reell tidsgrense, har vi de tidsparametrene som er nødvendige.

Poenget er at applikasjoner som benytter kvasireell tid, benytter *Time Warp* sitt virtuelle tidsbegrep til parallellitetskontroll mellom sine synkroniseringspunkter. En standardisert **QRT**-klasse kan lages som følger:

```
VirtualTime class QRT
begin
  VTInt vt;
  proceudure TickTack
  begin
    % Syncronize with real-time
    Clock := gettime();
    vt.initialize;
  end
  % ...
end
```

Her ser vi at klasse **QRT** benytter **VirtualTime** som et eget attributt. Merk at poenget med at kvasireell tid faktisk benytter virtuell tid som en egen synkroniseringsmekanisme *mellom* sine synkroniseringspunkter mot reell tid, reflekteres i prosedyrekallet **vt.initialize** hver gang den kvasireelle klokken tikker en verdi.

7.2.2 Standardisering av køene

Som sagt vil det være behagelig om jeg kunne beholde en samlet oppfattelse av købegrepet. Superklassen for et slikt købegrep deklarerer operasjoner for å legge inn og ta ut elementer fra en kø. Spesifikasjonen av operasjonen gjøres så der det faller naturlig, det vil si lengre nede i hierarkiet av køklasser. Supertypen for en køklasse i standardisert TBS vil da se slik ut:

```
class Queue
  virtual:
    procedure queue;
    QueueElem procedure dequeue;
    procedure fossile_collection (target); ref (TimeStamp) target;;
    procedure rollback (target); ref (TimeStamp) target;;
begin
end
```

De tre køene diskuteres så hver for seg nedenfor.

7.2.2.1 Standardisering av innkøen

Som nevnt ovenfor er det diskutert endel i litteraturen hvordan innkøen skal implementeres effektivt. Jeg har gått gjennom poengene med fortidsdelen og fremtidsdel av innkøen med ulike køstrukturer såpass inngående (se kapittel 5.1.1) tidligere at virkemåter og så videre ikke diskuteres her. En ting som derimot bør sies er at bruk av kompliserte datastrukturer i innkøens fremtidsdel krever en viss meldingshyppighet for å betale seg i økt effektivitet. Det å for eksempel skulle implementere en innkø der fremtidsdelen har struktur som et søketre er knapt forsvarlig dersom vi har et gjennomsnitt på for eksempel 10-20 meldinger i denne delen av listen. Jeg synes det er en rimelig antagelse at meldingshyppigheten er generelt større innenfor distribuert simulering enn innenfor parallellitetskontroll. Derfor må implementasjonskonstnader og økt *administrasjon* ved bruk av kompleks innkøstruktur vurderes nøyere innenfor sistnevnte synkroniseringstype enn innenfor den første.

Dersom vi derimot har et system der meldingshyppigheten og dermed størrelsen av køene er store også innenfor parallellitetskontroll, vil aktualiteten av kompliserte strukturer straks øke. I alle tilfeller må valg av køstruktur overlates til applikasjonen.

Av nye ting som må sies og tas hensyn til er bruk av flerdimensjonale tidsstempler. Disse krever en noe mer sofistikert sorteringsmekanisme for objektenes innkøer. Hvordan en slik sorteringsmekanisme skal lages er i mitt eksempel relativt enkelt. Den sekundære listen av tidsstempel er spesifisert som en tabell med heltall. Sorteringen blir dermed enkel. Det viktige er at den tar hensyn til de ulike dimensjonene i meldingenes tidsstempel og benytter disse for intelligent køing.

I lese/skrive-eksemplet (kapittel 4.2) er de ulike dimensjonene på tidsstemplene med vilje valgt slik at tidsstemplet som helhet kan sees som et reelt tall. Vi kan da sortere i *en* operasjon uten å måtte inspisere hver celle i *post_stamps* tabellen.

Dersom slik konstruksjon benyttes blir ikke køingsmekanismen mer komplisert her enn den er i *Time Warp*. I denne modellen benytter jeg tabeller.

La meg kikke på rammeverket for definisjon av **InQueue** med subklasser:

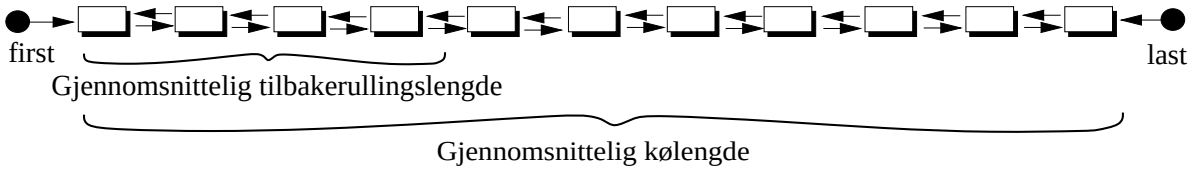
```
Queue class InQueue
begin
  integer cancelback_limit, num_messages_in_queue;
  ref (InOutQueueElem) past, future;

  Message procedure pop_past
  begin
    pop_past:=past.message;
    past:=past.next;% past always SortedList
  end
  Message procedure push_past (m); ref (Message) m;
  begin
    ref (InOutQueueElem) i;
    i:-new InQutQueueElem;
    i.message:-m;
    past.prev:-i;
    i.next:-past;
    past:-i;
end

InQueue class InQBinTree
begin
  procedure queue (m); Message m;
  begin ref (InQBinTreeElem) e;
            <Check for cancelback if used>
            e:-new(BinTreeQueueElem);
            e.message:-m;
            <Split into TimeStamp, ComplexTimeStamp and QRTTimeStamp>
            <queue e in binary tree, check for anhilation>
  end

  InOutQueueElem procedure dequeue;
  begin <get QueueElem with oldest message from future>
        if <use is_larger (message) against global CLOCK> then
        begin rollback (message.ReceiveTime);
              <get new correct message from QueueElem>
              <return it>
        end else
              <return QueueElem>
  end
  procedure rollback (target); TimeStamp target;
```

```
  begin <while first from past > target pop(past)->push(future)>
        <Check for anhilation>
        <GlobalTime> .SetClock (target);
        outqueue .rollback (target);
        statequeue .rollback (target);
  end
  procedure fossile_collection (target); TimeStamp target;
  begin <Delete all QueueElem in past with qm .message .ReceiveTime < target>
  end
end

InQueue class InQHeap
begin
  procedure queue (m); ref (Message) m;
  begin % ...
  end
end

InQueue class InQSortedList
begin
  procedure queue (m); ref (Message) m;
  begin % ...
  end
end
```

Fortidsdelen av innkøen implementeres som en dobbeltlinket, sortert liste for alle varianter av innkøer. Dette vil i alle tilfeller være den beste løsningen og denne delen kan derfor standardiseres innenfor superklassen. Operasjoner på fremtidsdelen er derimot ulike for ulike typer innkøimplementasjoner. Disse deklareres derfor som virtuelle prosedyrer i superklassen med tilhørende spesialimplementasjoner for hver subklasse.

7.2.2.2 Standardisering av utkøen

Her skal det legges inn antimeldinger tilsvarende alle meldinger som sendes ut fra objektet. Dersom objektet foretar en tilbakerulling må alle sendte meldinger kanselleres tilbake til *endepunktet* for tilbakerullingen. Køen og de operasjoner som skal gjøres på den vil bli relativt enkle. Alt som skal gjøres er å hente ut en melding og sjekke at tidsstemplet har høyere verdi enn endepunktet for tilbakerullingen. Dersom dette er tilfelle skal meldingen sendes ut som en vanlig melding. Dette bør gjøres i samme rekkefølge som de opprinnelige meldingene ble sendt. Dersom de siste antimeldingene ble sendt først kunne man risikere at man forårsaket en ny tilbakerulling i et annet objekt, for så å sende en annen antimelding som gir en



Figur 7.3: Eksempel på kø der LIFO-søking er mest effektivt.

ny tilbakerulling i det samme objektet til et tidspunkt tidligere enn den første. Av effektivitetshensyn bør derfor utsendelse av slike antimeldinger gjøres i rekkefølge sortert etter tidsstempel med lavest verdi først. Det kan derfor virke som en fornuftig løsning for alle varianter av *TBS* og bygge søkemekanismen i utkøen som et **LIFO**-søk. Videre starter man så utsending av antimeldinger i **FIFO**-rekkefølge. Merk at **LIFO**-søk er mest effektivt dersom det gjennomsnittelig totale antall antimeldinger i utkøen er dobbelt så stort som det gjennomsnittelige antall antimeldinger som må traverseres ved tilbakerullig. Dette er illustrert i figur 7.3. Klassen må i tillegg implementere en funksjon som administrerer køen ved tilbakerulling. Prosedyren er den som har ansvaret for å sende ut alle aktuelle antimeldinger:

```
Queue class OutQueue
begin
  ref (SortedListQueueElem) first, last;

  QueueElem procedure dequeue;
  begin
    dequeue:−last;
    last:−last.list;
  end

  procedure fossile_collection (target); ref (TimeStamp) target;
  begin
    while <last.prev.message.virtual_time < target> do
    begin
      last:−last.prev;last.next:−none;
    end
  end
end

OutQueue class OQAggrCancellation
begin
  procedure rollback (target); ref (TimeStamp) target;
  begin % Send all antimessages newer than target
  end

  procedure queue (m); ref (OutQueueElem) m;
  begin first.next :− m;
```

```
        first :- m;
  end
end

OutQueue class OQLazyCancellation
begin
  ref (SortedListQueueElem) saved;

  procedure rollback (target); ref (TimeStamp) target;
  begin % Save rollback-queue
  end

  procedure queue (m); ref (OutQueueElem) m;
  begin
    if <there exists a copy from a previous rollback> then
    begin
      if <m and corresponding message differ> then <send old antimessage>;
      else <cancel old antimessage, queue the new one>;
    end else
    begin <queue as usual>
    end
  end
end
```

Ved tilbakerullingssituasjoner vil vi da kunne starte med siste element i utkøen og søke oss frem til tidspunktet for første (mulige) korrupte melding. Alle senere meldinger i køen skal så sendes ut. Dersom vi har et system der alle objektene eksekverer på tilnærmet like plattformer slik at hastigheten er sammenlignbar, og vi har implementert mekanismer for å forhindre såkalte *runaway objects*, vil vi ha et system der avstanden i tid fra GVT[3] til det tidsmessig ledende objektet være relativt liten i forhold til et system der ulike maskiner eller noder eksekverer med ulik ytelse. I slike tilfeller vil søketiden fra første melding i fortidsdelen av innkøen frem til startmeldingen for tilbakerullingen være relativt kort.

7.2.2.3 Standardisering av tilstandskøen

Diskusjonen rundt standardisering av denne køen er relativt lik den for utkøen. Argumentene for å gjøre dette til en enkel kø av for eksempel type **LIFO** eller **FIFO** med hensyn til søking er de samme. Ved en tilbakerullingssituasjon trenger man å gjenopprette tilstanden i et objekt nøyaktig slik den var ved et tidligere tidspunkt, det vil si tiden rett før den korrupte meldingen ankom. Etter at alle antimeldinger

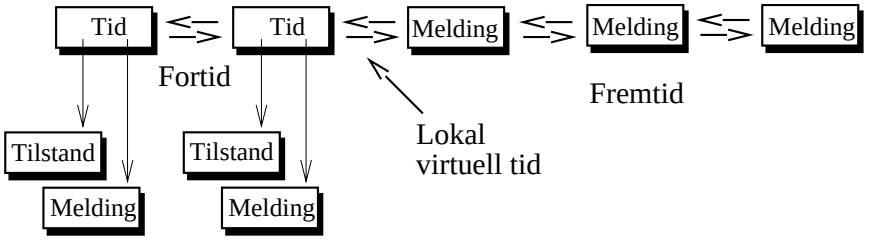
[3] Les her Global Virtual Time *eller* tilsvarende måltall for fremdrift av systemet. I en modell som bygges rundt for eksempel Dynamo snakker vi her om KRT (kvasireell tid).

fra utkøen er sendt skal tilstanden tilsvarende objektets nye tidsoppfattelse gjenopprettes. Hva menes så med et objekts tilstand. Før vi går videre la oss definere begrepet:

Definisjon 7.2.1 *Med et objekt $\mathcal{O}$ sin tilstand ved tiden $\mathcal{T}_o$ menes: En samling av alle interne datavariable som kan forandres ved, eller som følge av, meldingsutveksling mellom $\mathcal{O}$ og et annet objekt i det distribuerte system. Denne samlingen attributter skal unikt definere et objekts tilstand ved tid $\mathcal{T}_o$.*

Denne definisjonen kan ikke gis mer spesifikt før vi kjenner applikasjonsobjektene. Dette betyr igjen at en tilstandsklasse fra det standardiserte rammeverket må benyttes som en superklasse for den faktiske tilstandsklassen. Applikasjonen skal altså spesifisere hvilke attributter som skal lagres i en tilstand. Dette innebærer også at applikasjonsobjektene må angi en operasjon som lagrer de nødvendige attributter og en som gjenoppretter en gammel tilstand. Disse operasjonene defineres passelig som virtuelle prosedyrer i superklassen. Superklassene for elementene i tilstandskøene blir som følger:

```
Queue class StateQueue
  virtual:
    procedure save_state;
    procedure reset_state (s); ref (StateQueueElem) s;;
begin
  ref (StateQueueElem) first, last;

  procedure dequeue;
    while <last.at_time < GVT> do
          last:-last.next;

  procedure queue (m); ref (StateQueueElem) m;
  begin first.next :- m;
        first :- m;
  end

  procedure fossil_collection (target)
  ref (TimeStamp) target;
  begin % Throw away all states older than target
  end

  procedure rollback (target); ref (TimeStamp) target;
  begin
    while <first.at_time <> target> do first:-first.next;
    reset_state (first);
  end
end
```



Figur 7.4: Sammensmeltning av tilstandskø og fortidsdelen av innkøen. Merk at de nye elementene som lages må opprettes for hver gang en ny melding traverseres av objektet.

En slik tilstand skal lagres, sammen med klokkeverdien, hver gang objektene oppdaterer sin klokke normalt[4]. Dette innebærer at et objekts innkø og tilstandskø er nært knyttet sammen. Hver melding som er køet i det vi kan kalle fortidsdelen av innkøen vil ha en tilsvarende tilstand køet. Nedenfor føres en diskusjon om hvordan man kunne tenke seg disse to køene smeltet sammen til en.

7.2.3 Sammensmelting av tilstandskø og innkø

Dette er et forsøk på å la tilstandskøen til et objekt og fortidsdelen av innkøen smelte sammen til en kø. Som nevnt ovenfor finnes det nøyaktig en lagret tilstand i et objekt for hver melding som er køet i fortidsdelen til et objekts innkø. Vi så kapittel 5.1.1 at det er gjort endel arbeid rundt temaet implementasjon av innkøen til et objekt. Alle køer i objektene skal køes sortert på elementenes tidsverdier. Siden dette er felles for alle de tre køene i et objekt er det naturlig å ta dette som utgangspunkt for den nye *historiekøen*. Vi oppretter et nytt element som innholder en tidsverdi (tidsstempel) samt to pekere. Den ene pekeren adresserer meldingen med den samme tidsstempelverdien som tidsverdien i det nye elementet. Videre peker den andre pekeren til et element som inneholder objektets tilstand ved den gitte tiden. Vi får da en modell slik den er vist i figur 7.4.

Merk at dette vil ved en tilbakerulling føre til noe mer administrasjon i systemet. Man må da fjerne tilstandselementet samt det nye elementet og bare sette tilbake meldingselementet i fremtidsdelen av listen. Effektivitetstapet kan derimot ikke bli vesentlig fordi man slipper med en tilordning pr. element i fortidsdelen samt at man slipper å traversere en egen tilstandskø ved tilbakerulling. Ved gjennomsnittelig lange tilbakerullinger kan man også tenke seg en effektivitetsøkning.

Merk at dette ikke innebærer noen endringer av den teoretiske *Time Warp*-modellen. I praksis kan man fortsatt si at man har en tilstandskø og en innkø i objektene.

[4]Det vil si at objektet leser en melding med høyere eller lik tidsstempelverdi enn objektets egen lokale virtuelle klokke.

7.2.4 Valg av GVT-algoritme

Her synes jeg også man må kunne ha muligheten for valg etter systemets karakter. En generell implementasjon bør altså kunne tilby mulighet for både den gamle og den nye GVT-algoritmen som er presentert i henholdsvis kapittel A.1.1 og A.1.2. Som vi har sett eksekverer den nye algoritmen raskere enn den gamle. Det kreves derimot mer administrasjon av systemet for å bygge opp grafen som benyttes og så videre. I systemer med relativt få noder og objekter er det ikke nødvendigsvis gitt at effektivitetsøkningen blir merkbar. Derfor bør både den nye og den gamle algoritmen tilbys.

Hvorvidt δGVT-algoritmen skal tilbys vil avhenge av om implementasjonen benytter seg av et kommunikasjonssenter for meldingssending i systemet. I systemer der dette er tilfelle vil dette ville være et alternativ til direkte deltagelse fra objektenes side ved GVT-estimering. Dette vil gjelde innenfor et system som ANSAware. Dette er tema i kapittel 8.

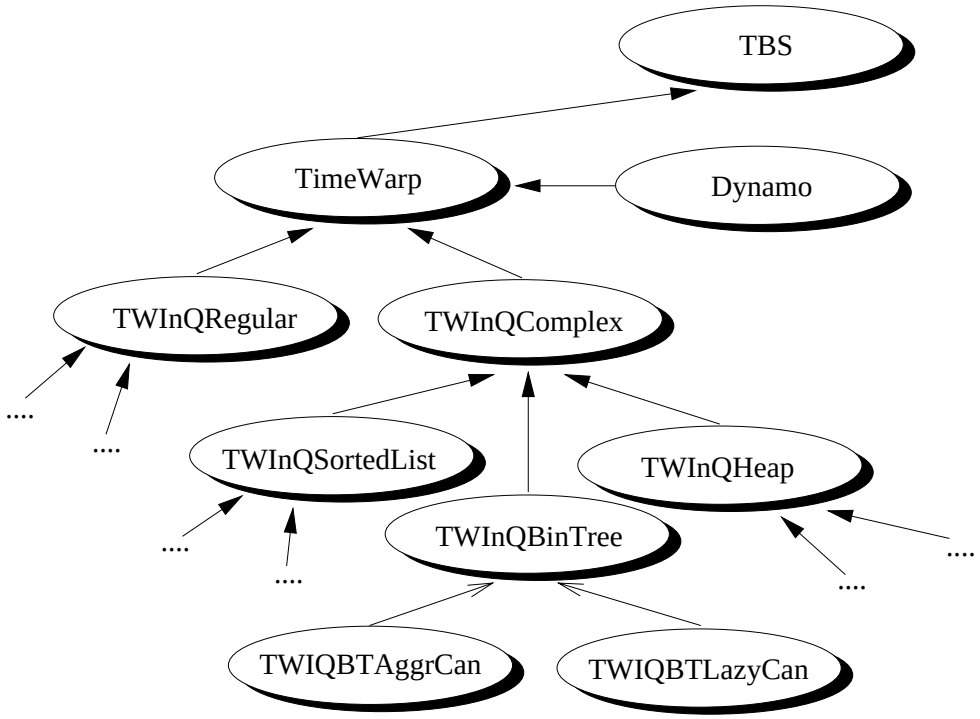
7.3 Definisjon av TBS-klassene

Jeg har nå konstruert en ryddig definisjon av de abstrakte datatypene som trengs for å sette opp hierarkiet for TBS-familiene. Mye arbeid er nå allerede gjort. Det man må gjøre nå er å sette opp de ulike typene TBS-metodene i en passende subklassestruktur. Et forslag til modell er gjort i figur 7.5. Siden jeg skiller en TBS-metode som for eksempel benytter kompleks innkøstruktur fra en som benytter enkel innkøstruktur vil mye av forskjellen på innmaten i ulike subklasser ligge i valg av abstrakt datatype for de ulike attributtene. De ulike køingsmekanismene velges allerede på deklarasjonsnivå fordi de er implementert inne i subklassehierarkiet for de abstrakte datatypene.

Det kan derfor diskuteres om hvorvidt man skal lage egne klasser eller om man skal la implementasjonen velge for eksempel kanselleringsstrategi gjennom subklassing og egne **new**-kall på opprettelse av køene. For å få det hele mest mulig konsistent mener jeg det bør opprettes egne subklasser selv om disse kanskje ikke inneholder så mye funksjonalitet. Denne løsningen indikeres nedenfor.

```
class TBS
  virtual:
    procedure ReceiveMessage (m); ref (Message) m;
    procedure SendMessage (m); ref (Message) m;
    Message procedure ReadMessage;
begin
end
```

```
TBS class TimeWarp
begin
  ref (VirtualTime) time;
  ref (InQueue) inqueue;
  ref (OutQueue) outqueue;
  ref (StateQueue) statequeue;

  procedure SendMessage (m); ref (Message) m;
  begin
    m.SendTime.pre_stamp := time.Clock;
    <Set post_stamps if complex timestamps are used>
    <Send to m.ReceiveIdentificator. use receiver's ReceiveMessage-procedure>
  end
  procedure ReceiveMessage (m); ref (Message) m;
  begin
    inqueue.queue (m);
  end
  Message procedure ReadMessage
  begin
    % The following statement may cause a rollback..
    % The user will not be aware of this..
    ReadMessage:-inqueue.dequeue.message;
  end
end

TimeWarp class TWInQRegular
begin
  inqueue:- new InQSortedList;
end

TimeWarp class TWInQComplex
begin
end

TWInQComplex class TWInQBinTree
begin
  inqueue:- new InQBinTree;
end

TWInQBinTree class TWIQBTAggrCan
begin
  outqueue:-new OQAggrCancellation;
end

TWInQBinTree class TWIQBTLazyCan
begin
  outqueue:-new OQLazyCancellation;
```



Figur 7.5: Klassehierarki for standardisert TBS.

```
end

TWInQComplex class TWInQHeap
begin
  inqueue:– new InQHeap;
end

TWInQComplex class TWInQSortedList
begin
  inqueue:– new InQSortedList;
end
```

Mer kode presenteres ikke fordi de ulike klassene blir meget enkle med så mye funksjonalitet innebygget i de abstrakte datatypene. De ulike TBS-klassene benyttes nå som superklasser for applikasjonsobjektene som skal implementeres. Denne måten å definere objekter på vil i følge definisjonen over gi oss en modell med sterk grad av objektorientering. De tre kravene om *innkapsling*, *abstraksjon* og *polymorfi* skulle være godt ivaretatt av modellen.

7.3.1 Kommantarer til rammeverket

Som man lett ser av de enkle kodeskissene som er angitt vil en fullstendig subklassedeklarasjon av alle mulige kombinasjoner gi et stort og relativt komplekst hierarki selv om innmaten i klassene er enkel. Dette er ingen uoverkommelig oppgave, og arbeidet kan reduseres betraktelig dersom man lar brukeren konfigurere opp sin egen TBS-variant via bruk av **new**-kallet på ulike abstrakte datatyper for køene og så videre. Genrelt ser modellen oversiktlig ut. De forskjellig løsninger som vi har gjennomgått i denne oppgaven er lette å separere slik at de kan standardiseres hver for seg. Med dette menes for eksempel at bruk av aggressiv eller lat kansellering faktisk kan skjules inne i det abstrakte købegrepet som defineres for utkøen. Dette reflekteres også i kompleksitet på henholdsvis klassehierarkiet for de abstrakte datatypene og det faktiske TBS-klassehierarkiet.

En egenskap som ville lette betraktelig på kompleksiteten til en fullstendig standardisert TBS-klasse ville være multippel arv[5]. Som sagt ovenfor er ulike egenskaper ved de ulike delene i en TBS-klasse lette å skjule i abstrakte datatyper. Med arving slik det kan defineres med mutippel arv vil man få en mer elegant løsning. For å illustrere dette, anta en applikasjonsprogrammerer som ønsker et *TimeWarp*-objekt som benytter seg av et binært søketre i fremtidsdelen av innkøen, lat kansellering og kvasireell tid som i Dynamo. Klassen for et slikt objekt kunne da deklareres som følger:

```
TimeWarp class QRT class InQBinTree
  class OQLazyCancellation class MyTimeWarpClass
begin
end
```

Som sagt ville dette gi oss en mer fleksibel modell der man lettere kan bygge opp sine egne varianter og konstruksjoner. Det ville også gi en applikasjonsprogrammerer større frihet fordi man enklere kunne definere inn sine egne egenskaper i tillegg til de predefinerte. Dette er ikke mulig i Simula[6].

Et implementasjonsverktøy som tilbyr denne formen for arving bør nok velges i en eventuell fullstendig implementasjon av standardisert TBS. Her kan man da tilby både et fullstendig klassehierarki slik det antydes her og i tillegg tilby muligheten for implementasjoner der applikasjonsobjektene bygges mer spesifikt av applikasjonsprogrammereren.

De elementene i modellen som krever størst deltagelse fra applikasjonens side er behandlig av tilstander. Siden vi har valgt å minimere de dataattributter som må

[5] På engelsk er ordlyden *multiple inheritance*.
[6] Se for eksempel syntaksskjema for **class**-deklarasjoner i [20] side 252.

lagres for å gjenopprette en gammel tilstand, vil operasjonene som lagrer og gjenoppretter tilstander overlates til applikasjonen. Dette er et negativt resultat av valget fordi det vil kunne bli relativt store operasjoner som må spesifiseres av applikasjonen. Valg av tilstandslagring som det er gjort i Dynamo ville utvilsomt gi en bedre abstraksjon av modellen på dette området, men ville på den annen side gi økt minneforbruk. Dette indikerer at man kanskje burde tilby de to ulike variantene til applikasjonen, det vil si at i tillegg til mekanismene i TBS-modellen tilbys tilstandslagring og tilstandsgjenoppretting som det er gjort i Dynamo.

7.3.2 Implementasjonseksempel

For å vise hvordan objekter i et distribuert system kan implementeres innenfor dette rammeverket, la oss kikke på en implementasjon av produsent/konsument-eksemplet fra kapittel 4.1. Merk at vi forutsetter at de abstrakte datatypene er synlige inne i TBS-klassehierarkiet. Disse kan enten deklareres globalt, eller de kan distribueres inn i hierarkiet. For eksempel ville vi da deklarere klassene «QueueElem» og «Queue» inne i klassen «TBS» og klassene «InQueue» og «OutQueue» i «TimeWarp».

Detaljene her er ikke så viktige. La oss bare anta at klassehierarkiet fra figur 7.2 er synlig når vi deklarerer oss subklasser av en av superklassene fra TBS-hierarkiet i figur 7.5. i kan da tenke oss følgende klassedeklarasjon for objektene **Consumer** og **Producer**:

```
begin
  Message class MyMessage
  begin
    integer produced_value;
  end

  TWIQBTLazyCancellation class Consumer
  begin

    integer Sum;

    ref (MyMessage) m;

    StateQueueElem class MySQElem
    begin
      integer sum;
    end

    StateQueue class MyStateQueue
    begin
      procedure save_state;
```

```
      begin
        ref (MySQElem) sqe;
        sqe.sum := Sum;
        sqe.at_time.pre_stamp := time.has_value; % Assume simple time_stamps
        statequeue.queue (sqe);
      end
      procedure reset_state (s); ref (MySQElem) s;
      begin
        % This procedure is called from the
        % rollback procedure in the StateQueue instans.
        % All we have to do is specify it.
        Sum := s.sum;
      end
    end

    statequeue:-new MyStateQueue;

    while true do
    begin
      % This following line will get the next message
      % All rollback adm. is performed without our
      % knowledge.
      m:-ReadMessage();
      Sum := Sum + m.produced_value;
    end
  end

  ref (Consumer) c;

  % This new-call will now start the loop inside
  % the instans of the Consumer objekt.
  c :- new Consumer;
end


begin
  Message class MyMessage
  begin
    integer produced_value;
  end

  TimeWarp class Producer
  begin
    % This objekt is simple. (See chapter 4.1)
    % We don't need save_state, reset_state or innkø, utkø.

    ref (MyMessage) m;
```

```
    while true do
    begin
      m:– new MyMessage;
      m.produced_value := produce();
      m.SendTime.pre_stamp := time.Clock;
      m.SendIdentificator :– "Producer";
      m.ReceiveIdentificator:– "Consumer";

      SendMessage (m);

      <sleep, or do some work..>
    end
  end

  ref (Producer) p;

  p:– new Producer;
end
```

Som vi ser blir klassene for de to objektene relativt enkle. Dette fordi TBS-familien her kan benyttes som et ferdig synkroniseringsverktøy. Med en slik ressurs ferdig implementert ville TBS bli mer attraktivt som parallellitetskontrollmekanisme i et OODS. Dersom metoden må implementeres for hver applikasjon blir den ikke så attraktiv for en applikasjonsprogrammerer fordi den krever såvidt mye implementasjon.

Kapittel 8

ANSAware og Time Warp

8.1 ANSAware

ANSAware er en objektorientert modell for å implementere distribuert objektorientert programmering ([16, 2, 1, 3, 6, 5, 4]).

8.1.1 Objektmodellen

ANSAware definerer begrepet *objekt* som en enhet med funksjonalitet rundt innkapsling og distribusjon. Mange andre definisjoner av begrepet *objekt* finnes, og de kan være noe ulike.

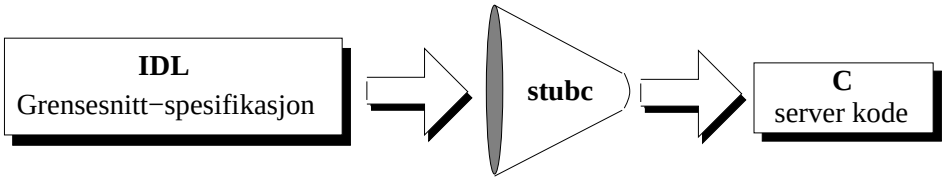
I [16] diskuteres hva som kjennetegner et *objekt* slik begrepet er benyttet i ANSAware. Som det påpekes eksisterer det mange ulike betydninger av begrepet objektavhengig av implementasjon og/eller systemer. Egenskapene som kjennetegner et objekt i ANSAware er følgende:

- *Et objekt representerer en abstraksjon*. Et objekt kan tilby tilgang til interne attributter og operasjoner gjennom sitt grensesnitt. Kravet om at et objekt skal være definerbart gjennom sitt grensesnitt ivaretas i denne egenskapen. Et objekt i denne modellen skal da være implementasjonsuavhengig.
- *Et objekt tilbyr en tjeneste*. Alle objekter skal ha et definert grensesnitt som kan aksesseres av klienter. Dette grensesnittet definerer samtidig hvilke tjenester objektet tilbyr omverdenen.
- *Objekter er innkapslet*. I tillegg til at objektet skal kunne tilby tilgang til interne attributter gjennom operasjoner som nevnt ovenfor, skal de interne attributtene være skjult for omverdenen.

- *Klienter foretar forespørsler til et objekt.* Klientene skal forespørre et objekt om å utføre en tjeneste. Hvordan denne tjeneste utføres intert i objektet er for klienten uinteressant. Forespørslen resulterer i eksekvering av en kodebit i objektet som tilsvarer forespørslen.
- *Alle forespørsler er navngitt.* Klientene identifiserer sine forespørsler via navn.
- *Forespørsler må identifisere objektene.* En klient må kunne identifisere et tjenerobjekt som en forespørsel skal rettes til. I praksis betyr dette at alle objekter må være refererbare.
- *Forespørsler kan ha argumenter og gi returverdier.* Dette er tilsvarende et funksjonskall i et programmeringsspråk.
- *Tjenestene kan beskrives.* Det må eksistere en grensesnittspesifikasjon som klienter kan benytte ved forespørsler. Denne grensesnittspesifikasjonen skal gi en entydig beskrivelse av alle operasjoner i grensesnittet med hensyn til parametere og returverdier.
- *Forespørsler kan være generiske.* Samme forespørsel kan rettes til ulike objekter. Ulik kode og ulike algortimer kan så utføres. Hvilken kode som eksekveres er altså avhengig av hva slags type objekt klienten identifiserer i forespørslen. Dette presiserer at forespørsler må kunne gjøres på en generell måte. Dette letter for eksempel OODS sin bruk av standardmodeller for objekter.
- *Objekter kan organiseres hierarkisk med hensyn til hvilke tjenester de tilbyr. Tilsvarende gjelder også med hensyn til hvilken grad av kodedeling vi finner mellom objekter.* Dette er ekvivalent med måten vi kan definere subklasser på i for eksempel Simula. En prosedyre kan deklareres og spesifiseres i en klasse. Denne klassen kan igjen være supertype for mange forskjellig subklasser. Alle funksjoner/prosedyrer i en slik superklasse kan være gjenstand for kodedeling mellom instanser av subklassene.
- *Objekter kan dele implementasjon.* Dette betyr at vi har instansiering av en klassedefinisjon. En klasse kan altså ha mange instanser. Alle objektene stammer fra den samme implementasjon av dataattributter og operasjoner, men hvert objekt (instans) har egne kopier av attributtene og koden.
- *Objekter kan delvis dele implementasjon.* Dette gir oss polymorfi.

8.1.2 Implementasjonsspråket

ANSAware har en språklig omgivelse for implementasjon av distribuerte objektorienterte systemer. Systemet er deklarativt i den forstand at distribusjonen og tjenestene implementeres i et høynivå språk i programteksten. Hele distribusjonen



Figur 8.1: Generering av server kode fra IDL ved hjelp av **stubc**.

```
GrensesnittTypeNavn1 : INTERFACE =
[IMPLEMENTATION] IS COMPATIBLE WITH GrensesnittTypeNavn2 [FROM Filnavn]
NEEDS GrensesnittTypeNavn3 [FROM Filnavn]
BEGIN
    <Datatyper>
    <Prosedyre−headere>
END.
```

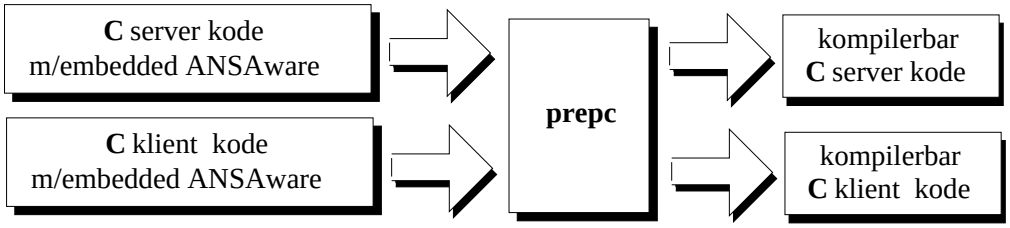
Figur 8.2: Skjellet for en grensesnittspesifikasjon i IDL.

og objektorienteringen bygges altså opp ved kompilering. Tjenestene som et objekt kan benytte eller tilby implementeres innenfor rammene av en klient/tjener-modell, men et objekt kan godt være klient og tjener samtidig. Og dette er et meget viktig poeng ved implementasjon av en *Time Warp*-modell og varianter av denne. En tjener tilbyr en tjeneste til klienter gjennom et predefinert grensesnitt. For å abstrahere seg vekk fra normal programtekst er det implementert et eget grensesnittspråk der hele grensesnittet defineres.

8.1.2.1 IDL - Interface Definition Language

Merk at det er grensesnittet som spesifiseres i IDL og ikke innmaten til de ulike funksjonene som utgjør grensesnittet. Dette spesifiseres senere som C-kode med ANSAware kommandoer innebygget gjennom såkalt **embedded programming**, se 8.1.2.2. For interesserte se kapittel 3 i artikkel [5] for en fullstendig definisjon av IDL. Et generell IDL-spesifikasjon er vist i figur 8.2 Denne modellen passer relativt godt sammen med det standardiserte rammeverket som ble definert i kapittel 7. De ulike datatypene må defineres inne i grensesnittet, men dette er bare en teknisk detalj. **IS COMPATIBLE WITH** gir oss mulighet til subtyping. Dette innebærer at *GrensesnittTypeNavn1* vil arve alle datatyper, samt alle operasjoner fra *GrensesnittTypeNavn2*. Dersom nøkkelordet **IMPLEMENTATION** også benyttes vil også selve implementasjonen av operasjonene i grensesnittet arves. Dette innebærer polymorfi.

Subtypekonstruksjonene fra kapittel 7 er dermed godt ivaretatt gjennom dette implementasjonsverktøyet. **NEEDS** nøkkelordet kan i eksemplet *GrensesnittTypeNavn1* aksessere attributter og operasjoner i *GrensesnittTypeNavn3*. Siden IDL-kode ba-



Figur 8.3: Generering av kompilerbar C-kode fra **embedded ANSAware** i C v.h.a. pre-prosessoren **prepc**

re definerer operasjonssignaturen og ikke gir implementasjonen, vil dette tilsvare bruk av virtuelle prosedyredeklarasjoner i rammeverket i kapittel 7.

IDL har egendefinerte typer. Disse må oversettes eller deklareres som tilsvarende C typer og variable dersom de skal benyttes i C-programteksten. Videre oversettes disse til C-typer av preprosessoren **stubc**, se figur 8.1. Denne tar som input et grensesnitt definert i IDL og genererer de nødvendige filer for implementasjon av *serveren* til den definerte tjenesten. Serverprogrammet implementeres altså med resultatkoden fra **stubc** som utgangspunkt.

8.1.2.2 prepc - C pre-prosessor

Kall til ANSAware gjøres i C-koden via såkalt **embedded programming**. Alle ANSAware setninger i C-koden skal starte med et '!' tegn i første kolonne. Dette er typisk kommandoer der klienten importerer en tjeneste eller en tjener eksporterer en tjeneste.

For å generere kompilerbar C-kode benyttes så preprosessoren **prepc** på koden som inneholder ANSAware tillegget. Dette gjøres så for både tjenerkode og klientkode, se figur 8.3.

8.1.3 Kommunikasjonsstrategier

To ulike kommunkasjonsstrategier kan benyttes i ANSAware for samspillet mellom en tjener og en klient:

- spørsmål/svar
- annonsering

I den førstnevnte strategien vil en klient vente på svar fra tjeneren før den kan starte med noe annet arbeid. Dette blir altså i praksis et funksjonskall der man venter

til returverdien er på plass. Den andre strategien benytter annonsering. Dette innbærer at en klient sender en forespørsel til en tjener om en tjeneste, men annonserer at resultatet av kallet skal mellomlagres av systemet i en predefinert systemvariabel. Når klienten så har tid, hentes resultatet av kallet fra systemvariablen som har mellomlagret resultatet.

For en implementasjon av *Time Warp*-modellen og varianter av denne vil en kommunikasjonsstrategi som bygger på annonsering være å foretrekke. Et objekt som sender en melding til et annet objekt skal ikke være avhengig av noe svar fra objektet. Meldingen skal sendes ut med et tidsstempel og avsenderen skal så umiddelbart kunne fortsette eksekveringen.

Spørsmål/svar-strategien kan tenkes brukt som en kvitteringsmetode for *sikker* meldingsutveksling. Men i dette tilfellet burde man kanskje benytte seg av et meldingsobjekt som kunne mellomlagre meldinger inntil en mottaker eventuelt er klar til å motta den sendte meldingen. Dette er jo ikke sikkert og da ville vi få en situasjon med venting, som ikke er akseptabel når en slik metode skal implementeres.

8.1.4 Standardisert implementasjon i ANSAware

Gjennom det klare skillet som settes mellom grensesnittet og innmaten til et objekt vil vi kunne klare å gi en god generell modell av en TBS-klasse. Slik vi husker denne klassen fra kapittel 7.3 er det stort sett en oppramsing av virtuelle prosedyrer samt deklarasjon av abstrakte datatyper som gjøres i disse klassene.

Slik IDL er bygget opp kan TBS-klassen lett overføres til ANSAware systemet med en abstrakt og dermed implementasjonsuavhengig definisjon av grensesnittet. De abstrakte datatypene kan også deklareres i IDL, men her er vi mer avhengig av koden som **idl**. En ting som byr på litt problemer er måten virtuelle prosedyrer er brukt på i kapittel 7.3. Vi kan arve prosedyrer, eller funksjoner mellom grensesnitt i IDL, men ingenting av innmaten kan spesifiseres. Dette må da gjøres i programmeringsspråkkoden som generes. Den versjonen av ANSAware som er basis for denne diskusjonen genererer C-kode. En nyere versjon som finnes generer derimot C++-kode, og her blir det da litt enklere å følge strukturen fra rammeverket som ble presentert i forrige kapittel, selv om dette ikke kan gjøres direkte i IDL som hadde vært ønskelig i denne sammenhengen.

IDL tilbyr derimot en god måte å spesifisere grensesnitt på. I og med at den definerer egne datatyper som kan benyttes til returverdi og parameterspesifikasjon blir disse helt implementasjonsuavhengige.

Objektmodellen i ANSAware synes også å følge de forutsetningene som ble satt opp i kapittel 2.1.3 om et likeverdig forhold mellom objektene i det distribuerte systemet. Hvert objekt i ANSAware kan fungere som tjener/klient for den samme tjenesten. Dette gir oss rent kommunikasjonsmessige gode muligheter for å

implementere TBS i dette systemet. Vi kan sende meldinger uten å måtte vente på mottaker[1] og det finnes mekanismer som gir oss sikker meldingsutveksling.

[1] Se kapittel 8.1.3 om annonsering.

Kapittel 9

Oppsummering og konklusjon

Med visse tilpasninger synes det som om TBS kan være en effektiv synkroniseringsmekanisme for parallellitetskontroll i distribuerte, objektorienterte systemer. Metoden er elegant i sin optimisme og er tiltalende som metode innenfor slike systemer.

Som en oppsummering av oppgaven og konklusjoner på sentrale emner, la meg gå gjennom de fire sentrale spørsmålene fra kapittel 1.2.1 og gi kommentarer og konklusjoner for hvert av disse.

- Gir tidsstempelbasert parallellitetskontroll av et distribuert, objektorientert system en tilfredsstillende løsning i praksis ?

  Som vist i eksemplene i kapittel 4 kan parallellitetskontrollproblemer løses elegant ved hjelp av tidsstempelbasert synkronisering. Gjennom visse utvidelser av *Time Warp*-modellen ble lese/skrive-problemet løst på en sikker måte uten bruk av tradisjonelle låsemekanismer.

  Metoden krever relativt mye implementasjon i forhold til mer tradisjonelle metoder. For at slik synkronisering lettere skal tas i bruk, bør det derfor eksistere et standardisert rammeverk som tilbyr TBS-mekanismene til en applikasjon. Arbeidet som er gjort i kapittel 7 kan danne basis for et slikt rammeverk.

- Hvordan påvirker en slik løsning effektiviteten i systemet i forhold til mer tradisjonelle løsninger ?

  Dette spørsmålet er vanskelig å besvare uten å implementere metoden og sammenligne den med tradisjonelle løsninger i empiriske tester. Dette gjenspeiles også i oppgaven da dette spørsmålet ikke besvares spesielt inngående. Det man kan si er at metoden krever mye større grad av administrasjon gjennom innføring av køer og implementasjon av mekanismer som tilbakerulling og fossilfjerning. Dette trekker klart i negativ retning når det gjelder effektivitet. Spørsmålet blir om man får en stor nok effektivitetsøkning gjennom

den optimistiske fremgangsmåten til å veie opp for effektivitetstapet man får gjennom økte administrasjonskostnader. Men dette er som sagt et spørsmål som må besvares gjennom empirisk tester.

Det finnes algoritmer som estimerer GVT i slike systemer. De to som presenteres i henholdsvis kapittel 3.3.2 og 3.3.3 kan benyttes for dette formålet. De har begge det fellestrekket at de ikke sier mye om hvordan *LVT* skal beregnes lokalt på hver node eller maskin. I implementasjoner der strukturen på systemet krever direkte deltagelse fra samtlige objekter under GVT-estimering, vil vi få en sentralisert mekanisme i en ellers så distribuert omgivelse. Dette er et negativt aspekt. I systemer som for eksempel ANSAware vil de negative aspektene ved slik GVT-estimering kunne reduseres ved bruk av $\delta\mathcal{GVT}$-algoritmen eller tilsvarende algoritmer.

- Kan metoden effektivt benyttes til parallellitetskontroll, og hvilke tilpasninger eller endringer må i såfall gjøres ?

  Av endringer som må gjøres i metoden kan for eksempel nevnes innføring av flerdimensjonale tidsstempler. Disse er ikke *nødvendige* for at metoden skal fungere som parallellitetskontrollmekanisme, men kan forbedre effektiviteten gjennom reduksjon av antall tilbakerullinger.

  Det viktigste punktet er allikevel muligheten for en sterkere binding til fysisk tid enn det som er tilfelle i *Time Warp*. Dynamo-modellen gir oss denne bindingen og ser i så måte lovende ut. Den innfører behov for «tuning» i systemet gjennom tidsgrensen som innføres i en transaksjon. Videre er det en forutsetning for applikasjoner som skal benytte Dynamo-aktige objekter med KRT til parallellitetskontroll at disse kan operere med en transaksjonsbegrep på en naturlig måte. For applikasjoner som for eksempel opererer mot en database, som modellen også er konstruert for, vil dette ofte være tilfelle. For andre typer applikasjoner er dette langt fra sikkert. Her må man utarbeide andre måter å innføre binding mot reell tid. Under arbeidet med denne oppgaven har det ikke kommet frem mulige løsninger på dette problemet.

- Hvordan er støtten i eksisterende programvare for å benytte seg av slik form for synkronisering ?

  Som det ble påpekt i kapittel 8 er det egentlig bare to egenskaper i et OODS som er nødvendige for å implementere TBS som generell synkroniseringsmekanisme. Det ene er at man har et symmetrisk forhold mellom klienter og tjenere. Det vil i praksis si at et objekt skal kunne fungere både som tjener og klient for den samme tjenesten. Den andre egenskapen som er nødvendig er muligheten for sikker meldingsutvekslig mellom objektene i det distribuerte systemet.

  ANSAware-systemet som er kort presentert i kapittel 8 kan implementere begge disse egenskapene og kan således benyttes som verktøy i en eventuell implementasjon. En ting som ikke er tilstede i ANSAware er muligheten for

å deklarere et subklassehierarki slik det ble gjort i kapittel 7. Dette bør være tilgjengelig for å lage en oversiktlig modell, men er ikke et krav. En nyere versjon av ANSAware er bygget for applikasjonsprogrammering med C++. Med dette klassebegrepet er vi kanskje lit nærmere målet.

Et sentralt tema som ikke er besvart i denne oppgaven og dermed kunne være et tema for videre arbeid er effektivitetsaspektet ved TBS-metoder kontra tradisjonelle løsninger. Dette ville kreve en del implementasjon, men det ville vært interessant å se resultatene av et slikt arbeid. Videre må man utarbeide metoder for å implementere sterkere binding mot fysisk tid *uten* å måtte operere med et transaksjonsbegrep slik det gjøres i Dynamo-modellen 6.2.

En implementasjon av rammeverket fra kapittel 7 kunne også være tema for videre arbeid. Dersom man gjør dette vil også veien frem til empirisk testing være kort.

Tillegg A

GVT-algoritmer

A.1 Kommentarer til dette appendikset

Her gjengis de to GVT-algoritmene som er beskrevet i kapittel 3.3. Den originale algoritmen er konstruert fra en punktvis forklaring funnet blant annet i [8]. Den nye GVT-algoritmen er også rekonstruert fra bekrivelse av forfatteren i [8]. Denne trenger imidlertid ekstra kode for å bygge opp grafen som benyttes i algoritmen. Denne koden finnes i detalj i artiklen, men er også gjengitt nedenfor. All kode er skrevet i *kvasi* C, men burde være forståelig uten videre forklaringer.

A.1.1 Den gamle GVT algoritmen

Denne koden er delt i to deler. En del tar for seg noden som starter GVT algoritmen. Dette er noden som i kapittel 3.3 omtales som node 0. La oss kikke på node 0 koden:
Objects tw_objects[NUM_NODES];

typedef struct {

  twref   Sender, Receiver;
  time   SendTime, ReceiveTime;
  mess_type type;

  time   gvt;

} tw_message;

time lvt,gvt;

```
void gvt_alg ()
{
 int i=1;
 tw_message *rec_mess;
/*
  Send broadcast to all nodes(objects)
*/
 for (;i≤NUM_NODES;i++) {

   tw_message m;
   m.Sender = tw_objects[0];
   m.Receiver = tw_objects[i];
   m.type = GVT_START;
/*
   Set other parametres here...
*/
   send_message (m);
 }
/*
  Receive all acknowledge messages from other nodes
*/
 for (i=1;i≤NUM_NODES;i++) {

   rec_mess = ReceiveMessage (tw_object[i]);

 }
/*
  We've reached RTM
*/
 for (;i≤NUM_NODES;i++) {

   tw_message m;
   m.Sender = tw_objects[0];
   m.Receiver = tw_objects[i];
   m.type = GVT_STOPP;
/*
   Set other parametres here...
*/
   send_message (m);
 }
/*
  Receive all messages from other nodes containing their LVT in
  the m.gvt parameter
  set lvt LARGE
```

```
*/
 lvt = LARGE_VALUE

 for (i=1;i≤NUM_NODES;i++) {

  rec_mess = ReceiveMessage (tw_object[i]);

  if (rec_mess→gvt < lvt)
   lvt = rec_mess→gvt;

 }
/*
  Broadcast the new GVT estimat lvt to all nodes
*/
 for (;i≤NUM_NODES;i++) {

  tw_message m;
  m.Sender = tw_objects[0];
  m.Receiver = tw_objects[i];
  m.type = NEW_GVT_ESTIMAT;
  m.gvt = lvt;
/*
   Set other parametres here...
*/
  send_message (m);
 }
/*
  Update my own gvt value
*/
 gvt = lvt;
}
/*
 End gvt-alg for object 0
*/
```

Denne koden skulle være relativt grei å forstå sammen med den punktvise forklaringen i kapittel 3.3. La oss så kikke på koden som implementerer funksjonene som må gjøres hos et objekt $i, i \in 1 \ldots N$ som *ikke* startet GVT-algoritmen:

```
tw_message *m;

time Start, Stopp, mvt;

while (_FOREVER_) {
```

```
 m = ReceiveMessage ();

 switch (m.mess_type) {

  case GVT_START:
            Start = m.gettime();
            m.type = GVT_START_ACK;
            m.Receiver = tw_objects[0];
/*
            Set rest of message parameters ...
*/
            SendMessage (m);
            break;

  case GVT_STOPP:
            Stopp = m.gettime();
            pvt = time_of_furthest_behind_object_on_node();
            mvt = min ( all_messages_in_transit(Start),
                  messages_send(Start,Stopp));
            lvt = min (pvt, mvt);
            m.gvt = lvt;
            m.Receiver = tw_object[0];
/*
            Set rest of message parameters ...
*/
            SendMessage (m);
            break;
 }
}
```

A.1.2 GVT algoritme med *message routing graph*

Dette er et kodeskjellet for «Message routing graph»algoritmen som er beskrevet i kapittel A.1.2. GVT-estimat algoritmen er delt i tre kodedeler. Først kommer koden for node 0 i grafen. Det er node 0 som har ansvaret for å starte GVT-estimat algoritmen. Dersom andre noder skulle ønske en GVT-oppdatering grunnet minnetilgang eller lignende, må det i tillegg implementeres en mekanisme for dette. For eksempel en enkel melding til node 0 at GVT-estimat trengs. La oss se på koden:

```
/*
 This is code for node number 0 in the message routing graph
*/
Objects from[2],to[2],my_adress;
```

```
typedef struct {

  twref     Sender, Receiver;
  time      SendTime, ReceiveTime;
  mess_type type;

  time      gvt;

} tw_message;

time  lvt,gvt;

void gvt_alg ()
{
  int i=1;
  tw_message *rec_mess;
/*
   Send message to successor nodes(objects)
*/
  for (;i≤2;i++) {

    tw_message m;
    m.Sender = my_adress;
    m.Receiver = to[i];
    m.type = GVT_START;
/*
    Set other parametres here...
*/
    send_message (m);
  }
/*
   Receive two messages from other nodes
*/
  for (i=1;i≤2;i++) {

    rec_mess = ReceiveMessage (from[i]);
    if (rec_mess.gvt < lvt) lvt = rec_mess.gvt;
  }
  gvt = lvt;
/*
   New gvt-estimat is now present in gvt
   Distribute it through graph
*/
```

```
}
/*
End of message-routing-graph algorithm for node 0
*/
```

Koden for nodene $1, \ldots, N-2$:

```
/*
 This is gvt-algorithm code for nodes 1, ... ,N-2
*/
Objects from[2],to[2],my_adress;
int numfrom, numto;

typedef struct {

 twref    Sender, Receiver;
 time     SendTime, ReceiveTime;
 mess_type type;

 time     gvt;

} tw_message;

time  lvt,gvt;

void gvt_alg ()
{
 int i=1;
 tw_message *rec_mess;
```

Koden for node $N-1$ blir ganske enkel. Denne gjør bare meldingsmottak fra sine forgjengere, beregner $\mathcal{LVT}_{N-1}$ og starter distribusjon av denne *baklengs* gjennom grafen:

```
/*
 This is gvt-algorithm code for node N-1
*/
Objects from[2],to[2],my_adress;
int numfrom, numto;

typedef struct {

 twref    Sender, Receiver;
 time     SendTime, ReceiveTime;
 mess_type type;

 time     gvt;
```

```
} tw_message;

time  lvt,gvt;

void gvt_alg ()
{
 int i=1;
 tw_message *rec_mess;
/*
  First, receive all GVT-start messages from "from[i]" nodes
*/
 for (i=0;i<numfrom,i++)
   rec_mess = ReceiveMessage ();
/*
  We've reached the real time {\em STOPP}$_{N-1}$
  which also is the real time RTM
  Compute lvt and start sending backwards in the message routing graph
  Send message to successor nodes(objects)
*/
 for (i=0;i≤numto;i++) {

   tw_message m;
   m.Sender = tw_objects[0];
   m.Receiver = tw_objects[i];
   m.type = LVT_CALC;
   m.gvt  = compute_my_lvt ();
/*
   Set other parametres here...
*/
   send_message (m);
 }
/*
  I'm done.
  Now,- wait for the new gvt-estimat to be distributed ..
*/
```

Det vi nå trenger er kode lokalt på hver node for å bygge opp grafen. Denne er gjengitt, tildels skissemessig i [8]. Den følgende koden er en kopi fra artikkelen:

```
/*
 This is code to build the message routing graph

 This code is just a copy of code presented in [8].
*/
```

```
int numFrom, numTo, from[2], to[2],
   Out0, Out1, numArrive;

/* My node number */
int myNum;
/* Total number of nodes */
FUNCTION gvtcfg()
{
  int Mid0, Mid1, myInv, OutFrom;
  int In0, In1, InTo, pen0, pen1;

  Mid0 = (numNodes - 1)/2;
  Mid1 = (numNodes & 0x01 ) ? Mid0 : Mid0 + 1;
/* Matching node on other tree */
  myInv = numNodes - myNum - 1;
/* For binary tree with root 0 */
   /* Left child number */
   Out0 = 2 * myNum + 1;
   /* Right child number */
   Out1 = Out0 + 1;
   /* Parent Number */
   OutFrom = (myNum - 1)/2;
/* For binary tree with root numNodes - 1 */
   /* Left child number */
   In0 = 2 * myNum - numNodes;
   /* Right child number */
   In1 = In0 - 1;
   /* Parent Number */
   InTo = (numNodes + myNum + 1)/2;

   for (pen0 = 1; pen0 < Mid0 + 1; pen0 = 2 * pen0 + 1) {
/*
      This is the only O(log N) part. Rest is O(1)
*/
     ;
   }

   pen0 = pen0/2 - 1;
   pen1 = numNodes - pen0 - 1;

   if (pen1 ≤ 2 * pen0 + 3) {
     /* the two trees can share the bottom row*/
     Mid0 = pen1 - 1;
     Mid1 = pen0 + 1;
```

```
  }

  if (myNum == 0) {
    numFrom = 0;
  } else
  if (myNum ≤ Mid0) {
    numFrom = 1;
    from[0] = OutFrom;
  } else
  if (Mid1 < In1) {
    numFrom = 2;
    from[0] = In0;
    from[1] = In1;
  } else
  if (Mid1 == In0) {
    numFrom = 1;
    from[0] = In0;
  } else {
    numFrom = 1;
    from[0] = myInv;
  }

  if (MyNum == numNodes − 1) {
    numTo = 0;
  } else
  if (myNum ≥ Mid1) {
    numTo = 1;
    to[0] = InTo;
  } else
  if (Mid0 ≥ Out1) {
    numTo = 2;
    to[0] = Out0;
    to[1] = Out1;
  } else
  if (Mid0 == Out0) {
    numTo = 1;
    to[0] = Out0;
  } else {
    numTo = 1;
    to[0] = myInv;
  }
}
```

Referanser

[1] ANSA. A System Designer's Introduction to the Architecture. Technical Report RC.253.00, ANSA, ANSA - Poseidon House, Castle Park, CAMBRIDGE CB3 0RD, United Kingdom, April 1991.

[2] ANSA. The ANSA Computational Model. Technical Report AR.001.00, ANSA, ANSA - Poseidon House, Castle Park, CAMBRIDGE CB3 0RD, United Kingdom, August 1991.

[3] Architecture Projects Management Limited, Poseidon House, Castle Park, CAMBRIDGE CB3 0RD, United Kingdom. *ANSAware 4.0, System Programmer's Manual*, February 1991. Document RM.099.00.

[4] Architecture Projects Management Limited, Poseidon House, Castle Park, CAMBRIDGE, CB3 0RD, United Kingdom. *An Overview of ANSAware 4.0*, March 1992. Document RM.099.00.

[5] Architecture Projects Management Limited, Poseidon House, Castle Park, CAMBRIDGE, CB3 0RD, United Kingdom. *ANSAware 4.0, Application Programmer's Manual*, March 1992. Document RM.102.00.

[6] Architecture Projects Management Limited, Poseidon House, Castle Park, CAMBRIDGE, CB3 0RD, United Kingdom. *ANSAware 4.0, System Manager's Guide*, February 1992. Document RM.100.00.

[7] D. Ball and S. Hoyt. The Adaptive Time Warp Concurrency Control Mechanism. *Proceedings of SCS90 Multiconference on Distributed Simulation*, 174-177, 1990.

[8] Steven Bellenot. Global Virtual Time Algorithms. *Proceedings of SCS90 Multiconference on Distributed Simulation*, 122-127, 1990.

[9] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2), June 1981.

[10] Gordon S. Blair, Javad Malik, John R. Nicol, and Jonathan Walpole. A synthesis of object-oriented and functional ideas in the design of a distributed software engineering environment. *Software Engineering Journal*, 1990.

[11] Samir R. Das and Richard M. Fujimoto. A performance study of the cancel-back protocol for Time Warp. *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation*, 23(1), May 1993.

[12] Anat Gafni and K.V. Bapa Rao. A Time-based Distributed Optimistic Recovery and Concurrency Control Mechanism. *Advanced Computing Support Center, Inc.*, June 1991.

[13] K.V. Bapa Rao Georg Ræder and Anat Gafni. Computational Aspects of Dynamo. *Technical Report, University of Southern California Advanced Computing Support Center, 3580 Wilshire Blvd., Suite 1910, Los Angeles*, CA 90010, September 1989.

[14] Kaushik Ghosh, Richard M. Fujimoto, and Karsten Schwan. Time Warp Simulation in Time Constrained Systems. *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation*, 23(1), May 1993.

[15] A. Goscinski. *Distributed Operating Systems - The logical Design*, chapter 6. Synchronization. ISBN 0 201 41704 9. Addison-Wesley publishing company, 1991.

[16] Andrew Herbert. Distributing Objects. Technical Report APM/TR.18.00, I-SA PROJECT, ANSA - Poseidon House, Castle Park, CAMBRIDGE CB3 0RD, United Kingdom, January 16 1992.

[17] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3), July 1985.

[18] David R. Jefferson. Virtual Time II : Storage Managment in Distributed Simulation. *9th Annual ACM Symposium on principles of distributed computing*, 1990.

[19] C. Thomas Wilkes John R. Nicol and Frank A. Manola. Object Orientation in Heterogeneous Distributed Computing Systems. *GTE Laboratories Inc.*, June 1993.

[20] Bjørn Kirkerud. *Object-oriented programming with Simula*. Addison-Wesley publishing company, 1989. ISBN 0-201-17574-6.

[21] H.T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2), June 1981.

[22] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Commun. ACM 21,7*, 7. July 1985.

[23] Yi-Bing Lin and Edward D. Lazowska. A Study of Time Warp Rollback Mechanisms. In *ACM Transactions on Modeling and Computer Simulations*, volume 1, pages 51–72. University of Washington, Seattle, January 1991.

[24] Georg Ræder. Standards and Distributed Applications: Implications for Product Model Implementation. Technical Report NR-notat DTEK/02/95, Norsk Regnesentral, December 1994.

[25] Georg Ræder, K.V. Bapa Rao, and Anat Gafni. Temporal Aspects of Dynamo. *Technical Report, University of Southern California Advanced Computing Support Center, 3580 Wilshire Blvd., Suite 1910, Los Angeles*, CA 90010, August 1990.

[26] Hassan Rajaei, Rassul Ayani, and Lars-Erik Thorelli. The local Time Warp approach to parallel simulation. *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation*, 23(1), May 1993.

[27] K.V. Bapa Rao, Anat Gafni, and Georg Ræder. Dynamo: A Time-based Object-orientated Model to Support Distributed Collaborative Development. *IEEE CompEuro 90, Tel Aviv, Israel*, May 1990.

[28] D.P. Reed. Implementing Atomic Actions on Decentralized Data. *ACM Transactions on Computer Systems*, 1(1), February 1983.

[29] Robert Ronngren, Rassul Ayani, Richard M. Fujimoto, and Samir R. Das. Efficient implementation of event sets in Time Warp. *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation*, 23(1), May 1993.

[30] B. Sambadi. Distibuted simulation: Performance and analysis. *Ph.D. dissertation, Dept. of Computer Science, UCLA, Los Angeles*, 1985.

[31] Jeff S. Steinmann. Breathing Time Warp. *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation*, 23(1), May 1993.

[32] Alexander Thomasian and Erhard Rahm. A New Distributed Optimistic Concurrency Control Method an a Comparison of its Performance with Two-Phase Locking. *IEEE - The 10th International Conference on Distributed Computing Systems*, 1990.

[33] Peter Wegner. Dimensions of Object-Based Language Design. *SIGPlan Notices*, 22(12), December 1987.