Sammendrag

Vi presenterer løsninger på et distribuert ressurskontrolleringsproblem i databasesystemet ClustRa. Ressurskontroll er et konsept som trengs for å sørge for at systemet tilbyr en tjenestekvalitet (eng: Quality of Service) til transaksjoner. Uten ressurskontroll kan systemet oppleve å bli så overbelastet at transaksjonene ikke får den tjenestekvaliteten de ønsker.

Løsningene på det distribuerte ressurskontrollproblemet som vi presenterer i denne rapporten, implementerer vi i en simulator sammen med transaksjonsprosessering og refragmentering fra det opprinnelige ClustRa. Rapporten gir en beskrivelse av hvordan denne simulatoren er oppbygd og implementert, og hvordan simuleringer utføres.

Vi vurderer resultatene fra simuleringene vi utfører, hvor vi simulerer ClustRa med ulike systemoppsett og ser på hvordan ressurskontroll-alternativene virker ved forskjellige systembelastninger. Dette oppnår vi ved å variere tiden mellom ankomster av nye transaksjoner, sannsynligheten for dataaksess på disken og om refragmentering utføres eller ikke.

Simuleringsresultatene viser at ressurskontroll er nødvendig ved systemoppsett med hyppig ankomst av transaksjoner og ved stor sannsynlighet for dataaksess på disk. Ved slike systemoppsett ser vi at systemet stopper å fullføre transaksjoner og utfører kun abort uten ressurskontroll, mens ressurskontroll-alternativene fortsetter å fullføre transaksjoner ved å avvise enkelte av dem. Vi ser også at det å avvise transaksjoner er mindre belastende for systemet enn abortprosessering, slik at vi får flere fullførte transaksjoner og lavere responstid ved å avvise transaksjoner i stedet for å abortere dem. Likevel ser vi at ressurskontroll-alternativene er mindre hensiktsmessig ved enkelte mindre belastende systemoppsett. Her opplever vi at systemene uten ressurskontroll fullfører flere transaksjoner enn systemene med ressurskontroll, på grunn av at systemene med ressurskontroll avviser transaksjoner som ellers ville blitt fullført.

Av løsningsalternativene vi implementerer og simulerer, ser vi at løsningen distribuert nodetabell er den som helhetlig er best. Videre vurderer vi transaksjonsklarering for å være det nest beste alternativet, med sentralisert nodetabell som det dårligste. Hovedgrunnene til at de to siste er mindre gode løsninger, er henholdsvis for mange meldinger og for mange avviste transaksjoner.

Forord

Denne oppgaven er endelig innlevering i faget “TDT4900 Datateknikk, masteroppgave” ved institutt for datateknikk og infomasjonsvitenskap ved Norges teknisk-naturvitenskapelige universitet.

Diplomarbeidet er utført som en følge av en oppgave gitt av professor Svein Erik Bratsberg. Arbeidet er en videreføring av prosjektarbeidet vi gjorde høsten 2004 i faget “TDT4740 Databaseteknikk og distribuerte systemer, fordypningsemne”. Oppgaven har vært å finne mulige løsninger på et distribuert ressurskontrolleringsproblem i databasesystemet ClustRa og å sette opp en simulator hvor vi simulerer de ulike alternativene vi har presentert på ulike systemoppsett av ClustRa.

Vi ønsker å takke professor Svein Erik Bratsberg for god veiledning og hjelp under implementasjonen av simulatoren og konstruktive innspill med henblikk på struktur, omfang og innhold i arbeidet med denne rapporten.

Trondheim, 9. juni 2005

Even Aasland

Magnus Solberg

Innholdsfortegnelse

Figurer

Tabeller

Kode-eksempler

1 Innledning

I dette kapitlet vil vi først se på diplomsoppgavens bakgrunn, før vi kommer med en kort problembeskrivelse av oppgaven som ligger til grunn for dette arbeidet. Videre kommer prosjektets omfang hvor vi presenterer hoveddelene av arbeidet vi har gjort. Deretter følger en oversikt over de mest sentrale definisjoner og begreper som er knyttet til vårt arbeid. Til slutt gir vi en kort oversikt over de kommende kapitler og hva de inneholder.

1.1 Bakgrunn

Denne rapporten er avsluttende innlevering i faget “TDT4900 Datateknikk, masteroppgave”, vårsemesteret i femte årskurs ved sivilingeniørutdanningen for datateknikk ved Norges teknisk-naturvitenskapelige universitet (NTNU). Arbeidet med denne rapporten gir 30 studiepoeng, noe som tilsvarer en gjennomsnittlig arbeidsmengde på 48 timer per uke.

Bakgrunnen for arbeidet med denne rapporten er databasesystemet ClustRa, som er utviklet av sentrale ressurspersoner ved institutt for datateknikk og informasjonsvitenskap (IDI) ved NTNU. Vår diplomoppgave er gitt og veiledet av professor Svein Erik Bratsberg, som har vært en av arkitektene bak ClustRa.

1.2 Problembeskrivelse

Databasesystemet ClustRa består av et sett uavhengige noder som samarbeider for å få utført transaksjoner. En transaksjon utfører et sett med operasjoner på en mengde data. Hver enkelt transaksjon involverer et antall av de uavhengige nodene for å få utført sine oppgaver. Er en node for tungt belastet, vil noden måtte abortere transaksjoner. Skjer dette på en eller flere av nodene en transaksjon utføres på, må transaksjonen avbrytes og alle noder som har utført noe av arbeidet fra denne transaksjonen må rulle dette tilbake for å sikre at alle data holder en konsistent tilstand. Arbeidet med å rulle tilbake utført arbeid er ressurskrevende og gir unødvendig systembelastning og arbeid utføres uten nytteverdi. Det er derfor ønskelig å oppnå distribuert ressurskontroll, slik at man i størst mulig grad unngår å rulle tilbake transaksjoner som allerede har startet sin utførelse. Samtidig er det viktig å holde systembelastningen nede, slik at eventuelle løsninger må unngå å øke systemets belastning for mye. Et annet aspekt ved denne problemstillingen er at overbelastning på systemet gjør at ingen transaksjoner opplever god progresjon i sitt arbeid. Dette er heller ikke ønskelig.

På denne måten har problemet to hovedmål som ønskes løst:

- Lav systembelastning - systemet skal ikke oppleve unødvendig ekstrabelastning som en følge av ressurskontrollen.
- Alle transaksjoner som slippes inn i systemet skal få en viss minstekvalitet i form av behandlingstid.

En mer detaljert beskrivelse av problemstillingen og målene for vårt arbeid følger i kapittel 2.

1.3 Omfang

Vårt arbeid er en videreføring av arbeidet vi utførte i faget “TDT4740 Databaseteknikk og distribuerte systemer, fordypningsemne” høsten 2004. Vi dannet gjennom dette arbeidet et teoretisk fundament for videre arbeid hvor vi kom fram til flere løsningsalternativer på det distribuerte ressurskontrollproblemet etter et studium av andre fagfelt, relevant bakgrunnsteori og studium av databasesystemet ClustRa.

Ut i fra prosjektrapporten som ble skrevet og arbeidet vi utførte har vi i vårens arbeid gått videre og gjennomført et arbeid som i hovedsak har vært tredelt:

- Velge løsningsalternativer på ressurskontrollproblemet og implementere disse i simuleringsmodellen
- Lage en simuleringsmodell for ClustRa som håndterer transaksjonsprosessering og refragmentering
- Gjennomføre simuleringer og vurdere hvilke ressurskontroll-alternativer som er best og hvilke typer av systemoppsett for ClustRa som er mest hensiktsmessig for ressurskontroll

Utover disse tre hovedpunktene har vi jobbet med å forstå simuleringsrammeverket Desmo-J som brukes i simulatoren og å få en skikkelig forståelse av hvordan ClustRa er bygd opp og fungerer.

1.4 Sentrale definisjoner og begreper

I dette underkapitlet vil vi definere og forklare noen sentrale begreper som vi bruker i denne rapporten. Vi velger å samle disse forklaringene her, siden det er viktig å ha dem klart definert før resten av rapporten leses. I tabell 1.1 er de sentrale definisjoner og begreper listet opp og forklart.

Tabell 1.1: Sentrale definisjoner og begreper

Begrep	Forklaring
Abort	Abort (å abortere) er å avbryte en transaksjon og rulle tilbake de deler av transaksjonen som er utført.
Commit	Commit (å fullføre) er å gjennomføre og fullføre en transaksjon.
Hot stanby-node	Hot standby-node er en rolle en node har i forhold til et fragment den lagrer. Noden vil bli bedt om å utføre et antall loggposter som den mottar fra noden som har primærrollen for fragmentet, ved oppdatering av dataene.
Node	Node er hvor data er lagret og hvor transaksjoner utføres i ClustRa. Alle data ligger hos to noder - en primærnode og en hot standby-node. En node kan opptre både som primærnode og som hot standby-node.
Primærnode	Primærnode er en rolle en node har i forhold til et fragment (en mengde av dataene) den lagrer. Aksess mot et fragment skjer via noden som har primærrollen for fragmentet.
Speilnodeforhold	Et speilnodeforhold er et forhold mellom to noder som inneholder de samme dataene. Hver av nodene vil være hot standby-node og primærnode for ulike deler av datamengden lagret på de to nodene.
Transaksjon	En transaksjon er sammensatt av en eller flere operasjoner som skal utføres sammen. Enten utføres alle operasjoner (commit) eller utføres ingen operasjoner (abort). På denne måten blir en transaksjon den minste enheten som kan utføres i en database.

1.5 Oversikt

Vi vil her kort presentere innholdet i de kommende kapitler. Resten av rapporten inneholder følgende kapitler:

- **Kapittel 2 - Diplomoppgaven.** Kapitlet om diplomoppgaven presenterer problemstillingen og hvilke hovedmål vi har satt til vårt arbeid med denne oppgaven.
- **Kapittel 3 - Beskrivelse av ClustRa.** Her beskriver vi de deler av ClustRa som er relevant for vår oppgave og for å implementere simulatoren. Vi fokuserer spesielt på hvordan transaksjoner utføres i ClustRa, da dette er sentralt for implementasjonen av simulatoren.
- **Kapittel 4 - Intern ressurskontroll.** I dette kapitlet ser vi på hvordan intern ressurskontroll kan utføres. Vi ser først på forhold rundt interne tilstander, før vi presenterer de løsningene for å oppnå intern ressurskontroll som vi velger å implementere i simulatoren vår.

- **Kapittel 5 - Distribuert ressurskontroll.** Distribuert ressurskontroll er et kapittel hvor vi presenterer løsningsmålene for distribuert ressurskontroll og de løsningsalternativene som vi har valgt å implementere:
    - Distribuert nodetabell
    - Sentralisert nodetabell
    - Transaksjonsklarering
- **Kapittel 6 - Simulatoren.** Her presenterer vi hvordan vi har bygd opp simulatoren. Vi ser på hvordan simulatoren er bygd opp ut i fra simuleringsrammeverket, av ulike klasser og hvordan hver enkelt av klassene er implementert. Kapitlet har egne underkapitler som beskriver hvordan hoveddelene ressurskontroll, transaksjonsprosessering og refragmentering er implementert.
- **Kapittel 7 - Oppsett av ressurskontroll.** I dette kapitlet presenterer og begrunner vi hvilke verdier vi har valgt for parameterne som styrer utførelsen av ressurskontrollen.
- **Kapittel 8 - Simulering og resultater.** Simulering og resultater er et kapittel hvor vi viser simuleringene vi har utført og presenterer og drøfter resultatene av hver enkelt simulering. Vi gjør også en sammenligning av ressurskontroll-alternativene og drøfter hvordan de ulike alternativene oppfyller løsningsmålene.
- **Kapittel 9 - Konklusjon og videre arbeid.** Her konkluderer vi det arbeidet vi har utført og skisserer hva som kan gjøres av videre arbeid.

2 Diplomoppgaven

I dette kapitlet vil vi presentere problemstillingen som ligger til grunn for diplomoppgaven. Vi vil også se på hvilke mål vi har satt for dette arbeidet.

2.1 Problembeskrivelse

Vi vil i dette underkapitlet presentere selve problemstillingen på en generell måte. En spesifikk beskrivelse av de relevante delene av databasesystemet ClustRa kommer i kapittel 3, mens konkrete løsningsalternativer presenteres i kapittel 5 sammen med mål for god løsning av distribuert ressurskontroll.

Som nevnt i innledningen har vår diplomoppgave sitt utspring i databasesystemet ClustRa. I dette systemet er pålitelighet, responstid og oppetid viktige parametere. For å sikre dette har man sett behovet for en sikker og god distribuert ressurskontroll.

Utgangspunktet er et system med et sett uavhengige noder (eng: shared nothing) som kommuniserer med hverandre. Vi har en kø med transaksjoner som ankommer systemet uavhengig av hverandre. Hvor forespørslene ankommer systemet er tilfeldig, med en antatt uniformfordeling mellom de ulike nodene i systemet. Hver enkel transaksjon vil kreve et ulikt antall noder for å få utført alle deloppgavene til transaksjonen.

Hovedproblemstillingen vår er å stoppe en transaksjon fra å starte utførelse hvis en eller flere av nodene som transaksjonen trenger er opptatt eller tungt belastet. Slippes en transaksjon inn til en eller flere opptatte eller tungt belastede noder, kan transaksjonen måtte abortere og det arbeidet som er utført må rulles tilbake. Følgelig vil dette belaste systemet unødvendig og arbeid er utført uten nytteverdi. Derfor er det ønskelig med ressurskontrollering av de uavhengige nodene for å unngå slike situasjoner. Det er ønskelig at hver node i systemet har en global forståelse av hvordan belastningen er på de andre nodene for å kunne oppnå denne ressurskontrolleringen. Systemet har i utgangspunktet ingen sentrale enheter som kan stå for noen overordnet styring.

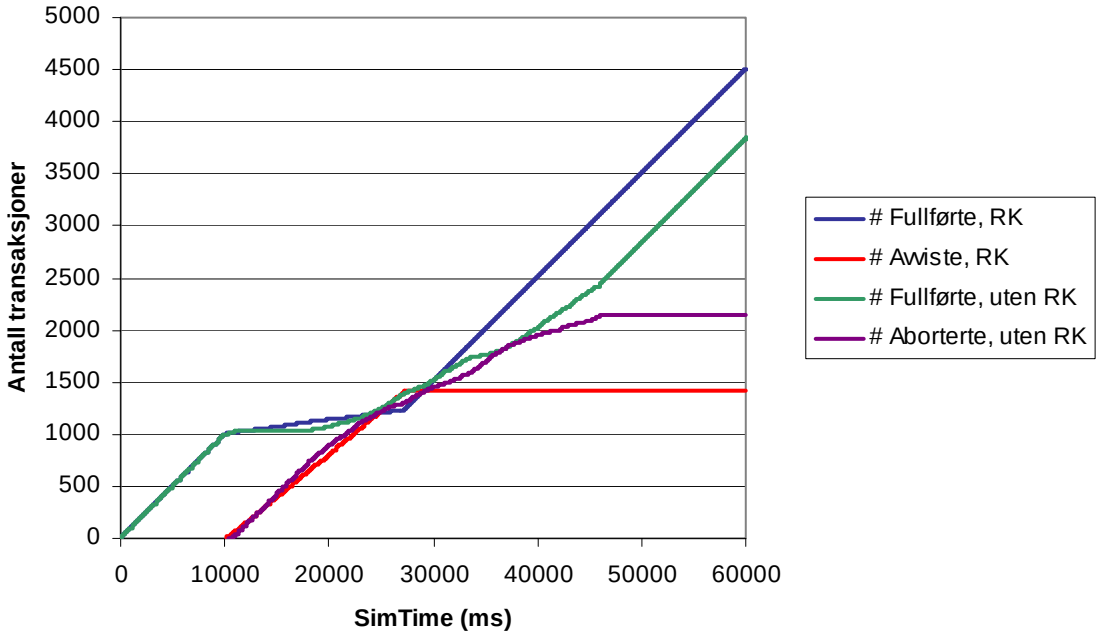
Samtidig er det ønskelig at systemet utnytter de tilgjengelige ressursene på en god måte, slik at eventuelle løsninger for å oppnå ressurskontroll må ha et minimum av kostnad og systembelastning.

I figur 2.1 har vi vist en figur som prøver å illustrere noe av hva vi ønsker å oppnå. Figuren viser en simulering av ClustRa over 60 sekunder i simuleringstid med parametere som gir et relativt tungt belastet system. X-aksen angir antall millisekunder (ms) med simulering, mens y-aksen angir antall transaksjoner. Vi ser her utviklingen på følgende tall:

- Antall fullførte transaksjoner med ressurskontroll
- Antall avviste transaksjoner med ressurskontroll
- Antall fullførte transaksjoner uten ressurskontroll
- Antall aborterte transaksjoner uten ressurskontroll

Vi ser at systemet går fint uten ressurskontroll i starten, ingen transaksjoner aborteres og antall fullførte øker jevnt. Etter hvert får systemet så mye å gjøre at transaksjonene begynner å abortere. Antall aborterte øker mer, mens antall fullførte stiger langsommere. Dette knekkpunktet kommer av at refragmenteringen har startet etter rundt 10 000 ms. Når det samme systemet kjøres med ressurskontroll, ser vi at antall fullførte transaksjoner i starten følger grafen for systemet uten ressurskontroll. Etter at refragmenteringen har startert, ser vi at antall fullførte transaksjoner øker langsommere også med ressurskontroll og flere transaksjoner avvises av systemet. Denne avvisningen slutter etter rundt 28 000 ms av simuleringen og vi får igjen en jevn og sterkere økning i antall fullførte for systemet med ressurskontroll. Ser vi derimot på systemet uten ressurskontroll, slutter ikke dette systemet å abortere transaksjoner før rundt 45 000 ms av simuleringen. Vi ser at dette medfører flere aborterte transaksjoner og færre fullførte transaksjoner enn for systemet med ressurskontroll.

Vi går ikke nærmere inn på detaljene for denne simuleringen her og viser til kapittel 6 og kapittel 8 for forklaring av simulatoren og gjennomgang av resultatene av de ulike simuleringene.



Figur 2.1: Eksempel på nytten av ressurskontroll

2.2 Arbeidsmål

Vi vil i dette underkapitlet presentere hovedmålene vi har satt til diplomarbeidet. Det er her verdt å merke seg at dette gjelder hele arbeidet og at konkrete løsningsmål for ressurskontroll-problemet vil presenteres i kapittel 6.

Vårt diplomarbeid har følgende hovedmål:

- **A1 - Modellering og forståelse av ClustRa**
  Vi skal modellere og forstå de deler av ClustRa som er relevante for å få til en realistisk simulering av systemet for å teste ut distribuert ressurskontroll. Vi må her gjøre nødvendige begrensninger i hvilke deler av ClustRa vi tar med i oppbygningen av simulatoren, da systemet er for stort til at vi kan klare å få til en fullstendig modell.
- **A2 - Definere løsningsmål og løsningsalternativer for distribuert ressurskontroll**
  I prosjektarbeidet vi gjennomførte i høst kom vi fram til flere alternativer for distribuert ressurskontroll. Vi kom også fram til ulike alternativer for å utføre intern ressurskontroll, som er en forutsetning for å få til distribuert ressurskontroll. Vi skal nå ta disse alternativene for både distribuert og intern ressurskontroll videre og se på hvordan vi kan implementere dem sammen med resten av ClustRa, definere løsningsmål for valg av løsninger for distribuert ressurskontroll og velge ut de av alternativene vi ønsker å simulere og teste.
- **A3 - Programmere en simuleringsmodell for ClustRa med distribuert ressurskontroll**
  Vi skal programmere en simulator hvor vi kan simulere fullstendig transaksjonsutførelse i ClustRa, både med og uten de ulike alternativene for ressurskontroll. Det skal være lett å endre systemoppsettet til ClustRa i simulatoren for å kunne simulere ulike systemoppsett av databasesystemet.
- **A4 - Gjennomføre simulering og måle resultatene opp mot løsningsmålene**
  For å finne ut hvilke av ressurskontroll-alternativene som er best og for å finne ut for hvilke systemoppsett ressurskontroll er hensiktsmessig og ikke, skal vi utføre simuleringer som dekker interessante grensetilfeller. Vi skal gjøre en drøfting av de ulike simuleringene og måle resultatene vi finner opp mot løsningsmålene vi setter for distribuert ressurskontroll.

3 Beskrivelse av ClustRa

Vårt problem har sitt utspring i databasesystemet ClustRa. I denne delen vil vi kort beskrive ClustRa-systemet på bakgrunn av [2], [3] og [4]. Vi vil først kort presentere ClustRa generelt, før vi ser på hvordan det fysisk og logisk er oppbygd. Til slutt vil vi se på hvordan transaksjonsprosesseringen utføres i ClustRa.

3.1 Om ClustRa

ClustRa ble opprinnelig utviklet av Telenor, før det ble skilt ut i et eget selskap. I 2002 ble selskapet kjøpt av Sun Microsystems. Nå heter produktet HADB og leveres som en del av Sun Java System Application Server Enterprise Edition. Den opprinnelig tanken med ClustRa var at systemet skulle være passende for telekommunikasjonsanvendelser, hvor rask responstid, høy tilgjengelighet og skalerbar gjennomstrømming (eng: throughput) er essensielle krav.

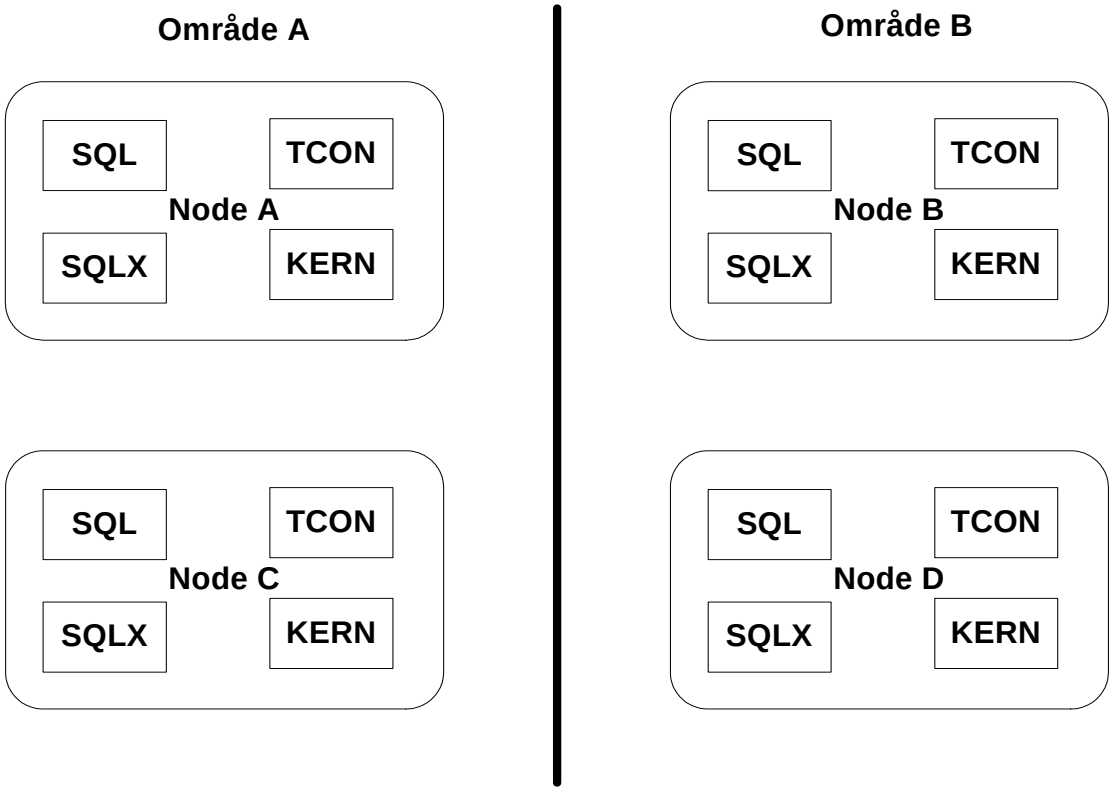
3.2 Oppbygning

Nodene i ClustRa er fordelt på to områder (eng: sites). Områdenes sannsynlighet for å bli utilgjengelig er uavhengig av hverandre. Dette utnyttes ved at hvert område sitter på en kopi av hele databasen, slik at alle data er tilgjengelig dersom et av områdene skulle bli utilgjengelig. En node fra det ene området er en såkalt speilnode til en node i det andre området. Hvert par av speilnoder vil normalt lagre de samme dataene. Den ene av de to nodene vil være primærnode for en del av dataene og hot standby for de øvrige datene. Tilsvarende blir den andre noden primærnode for de dataene den første noden var hot standby for og hot standby for de dataene den første noden var primærnode for.

Hvis vi ser på et system med fire fragmenter av en database nummerert 0 til 3, kan vi tenke oss at dataene er fordelt slik tabell 3.1 viser. Her ser vi hvordan dataene er fordelt på fire noder, A til D. Vi ser at node A og node B er speilnoder for elementene 0 og 1, og det samme er node C og node D for elementene 2 og 3. Den fysiske oppdelingen til dette systemet er illustrert i figur 3.1, hvor vi også ser hvilke elementer de ulike nodene i ClustRa består av.

Tabell 3.1: Datafordeling i ClustRa

Node	Primær-element	Hot standby-element
A	0	1
B	1	0
C	2	3
D	3	2



Figur 3.1: Organisering av ClustRa

Hver node i ClustRa, som vi ser i figur 3.1, har fire logiske prosesser som håndterer transaksjoner. SQL-prosessen får inn en transaksjon fra en klient. Transaksjonskontrolleren (TCON) er ansvarlig for at transaksjonen deles opp i underoppgaver og utføres på riktig sted i systemet. Kjernen (KERN) på en node har kontroll over de data som er lagret. SQLX-prosessen sine oppgaver er relatert til de mer kompliserte transaksjonene, som omfatter bruk av relasjonsalgebra. Vår simulator fokuserer på hvordan TCON og KERN samarbeider på de ulike nodene for å gjennomføre transaksjonsprosesseringen. Vi ser av den grunn ikke nærmere på SQL og SQLX i vårt arbeid. Vi viser til [2] for en nærmere forklaring av disse prosessene.

3.3 Transaksjonsprosessering

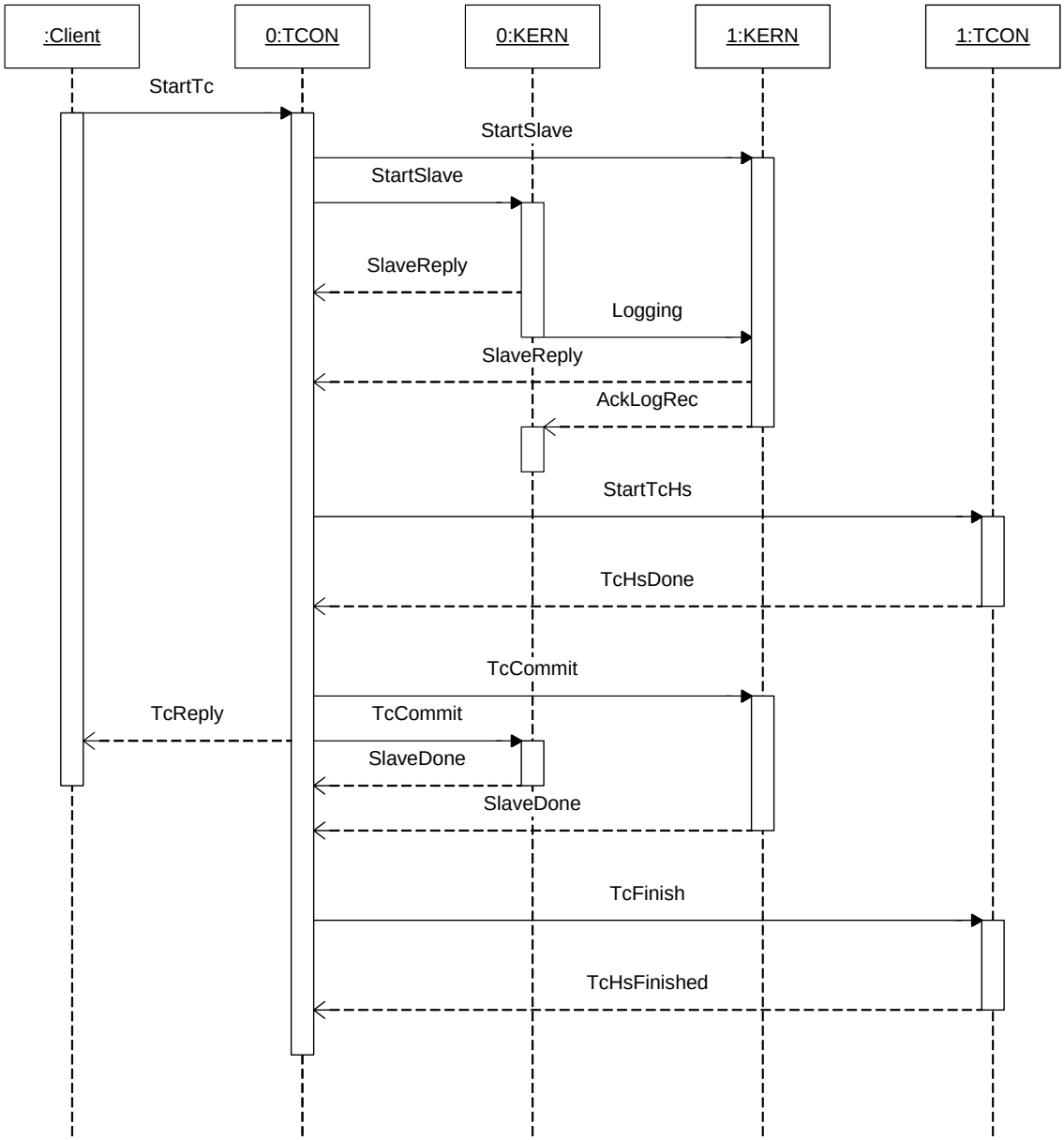
Transaksjonsprosessering i form av samarbeid mellom de ulike TCON- og KERN-prosessene er den sentrale delen av ClustRa for vår del, siden det er transaksjonsprosesseringen vi skal konsentrere oss om under simuleringen. Vi vil i dette underkapitlet først se på hvordan en oppdateringstransaksjon som gjør én oppdatering utføres, før vi ser på hva som skjer ved flere oppdateringer fra samme transaksjon. Til slutt ser vi på hvordan abort-prosessering utføres. Disse beskrivelsene er basert på [5].

3.3.1 Én oppdatering

Forklaringen av transaksjonsprosesseringen av én oppdatering følger sekvensdiagrammet i figur 3.2. I denne figuren er TCON transaksjonskontrolleren som kjører `TrolThread`-tråder. Objektet *0:TCON* er den primære TCON, mens objektet *1:TCON* er hot standby TCON. KERN-prosessene er slavene i systemet som kjører `SlaveThread`-tråder. Her er objektet *0:KERN* den primære KERN, mens *1:KERN* er hot standby KERN.

Meldingen *StartTc* fra klienten vil starte opp hele transaksjonsprosesseringen ved å aktivere den primære TCON. Denne tråden vil nå sende *StartSlave* til hot stanby KERN og primær KERN. Den primære KERN vil svare den primære TCON med meldingen *SlaveReply* når den har utført sin prosessering. Deretter sender den meldingen *Logging* til hot standby KERN. Denne meldingen

inneholder loggen som skal utføres. Hot stanby KERN svarer deretter med *SlaveReply* til primær TCON og *AckLogRec* til primær KERN. Loggen utføres enten før eller etter disse meldingene. Når primær TCON har mottatt begge *SlaveReply*-meldingene sendes *StartTcHs* til hot standby TCON. Når den har gjort nødvendig prosessering for å bli hot stanby TCON svarer den med *TcHsDone*. Nå er den primære TCON klar til å fullføre transaksjonen og sender *TcCommit* til både primær og hot standby KERN. Til klienten sendes *TcReply*, som en indikasjon på at transaksjonen er fullført. Primær KERN og hot standby KERN svarer primær TCON med *SlaveDone*. Når begge disse meldingene er mottatt av primær TCON, vil meldingen *TcFinish* sendes til hot standby TCON. Denne svarer med *TcHsFinished* og transaksjonsprosesseringen er ferdig.



Figur 3.2: Transaksjonsprosessering av én oppdatering i ClustRa

3.3.2 Flere oppdateringer

Ved to eller flere oppdateringer vil fortsatt diagrammet i figur 3.2 være gjeldende som et korrekt utgangspunkt. Forskjellen vil være:

- Systemet vil ha et ekstra sett med primær og hot standby KERN for hver ekstra node transaksjonen involverer. Om vi ser på en transaksjon med fire oppdateringer, kan vi maksimalt få fire primære og fire hot standby KERN-prosesser. Minimalt kan vi fortsatt ha en kun én primær

og én hot standby, om alle oppdateringene skjer på samme node. Antall KERN-prosesser som involveres styres altså av antall oppdateringer og antall involverte noder i transaksjonsprosesseringen.

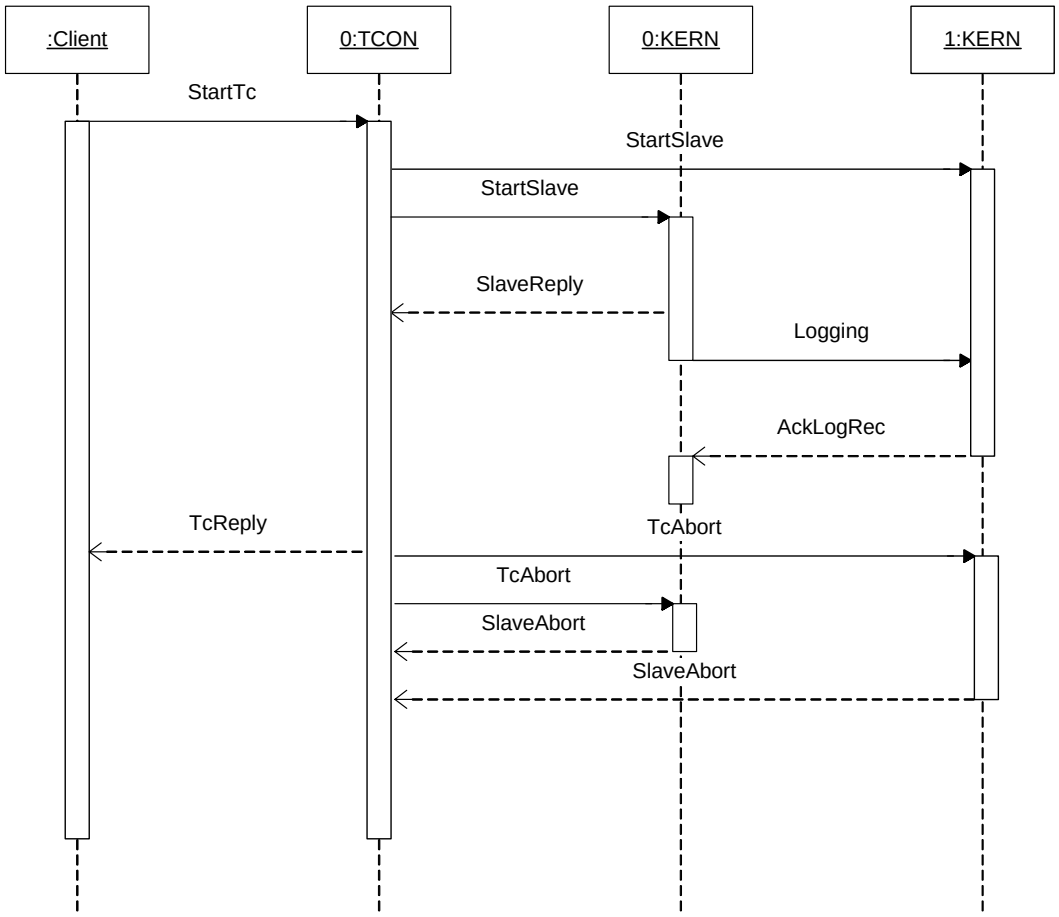
- Primær TCON sender meldingen *StartSlave* til alle primære og hot standby KERN-prosesser. Dette kan gi fra to meldinger til åtte meldinger for en transaksjon med fire oppdateringer.
- Ved flere par av KERN-prosesser vil man også få en tilsvarende økning i antall *Logging*- og *AckLogRec*-meldinger, da dette er meldinger som sendes mellom hvert par av primær- og hot standby-KERN.
- TCON vil få samme økningen som økningen i sendte *StartSlave*-meldinger for antall *TcCommit* den sender ut og antall *SlaveDone* som mottas som svar.

Forholdet mellom primær TCON og hot standby TCON blir uforandret, da det alltid vil være kun én primær og én hot standby av denne prosessen i transaksjonsprosesseringen.

3.3.3 Abort-prosessering

Abort vil si at en transaksjon som har startet må rulles tilbake. Dette kan kun skje før meldingen *StartTcHs* sendes fra primær TCON til hot standby TCON. En abort-situasjon vil oppstå når en eller flere av meldingene *SlaveReply* fra KERN-prosessene ikke kommer til primær TCON eller at de kommer for sent, det vil si etter at maksimal ventetid er oversteget.

I figur 3.3 har vi vist en situasjon med en transaksjon med én oppdatering. Her mottar ikke primær TCON meldingen *SlaveReply* fra hot standby KERN. Følgelig startes abortprosesseringen ved å sende *TcAbort* til alle KERN-prosesser når timeout-tiden er passert og meldingen *TcReply* med indikasjon om abort til klienten. Slavetrådene svarer med *SlaveAbort* når nødvendig abortprosessering er gjennomført. Meldingen *StartTcHs* sendes derfor ikke til hot standby TCON. Av den grunn har vi også utelatt denne tråden fra figur 3.3.

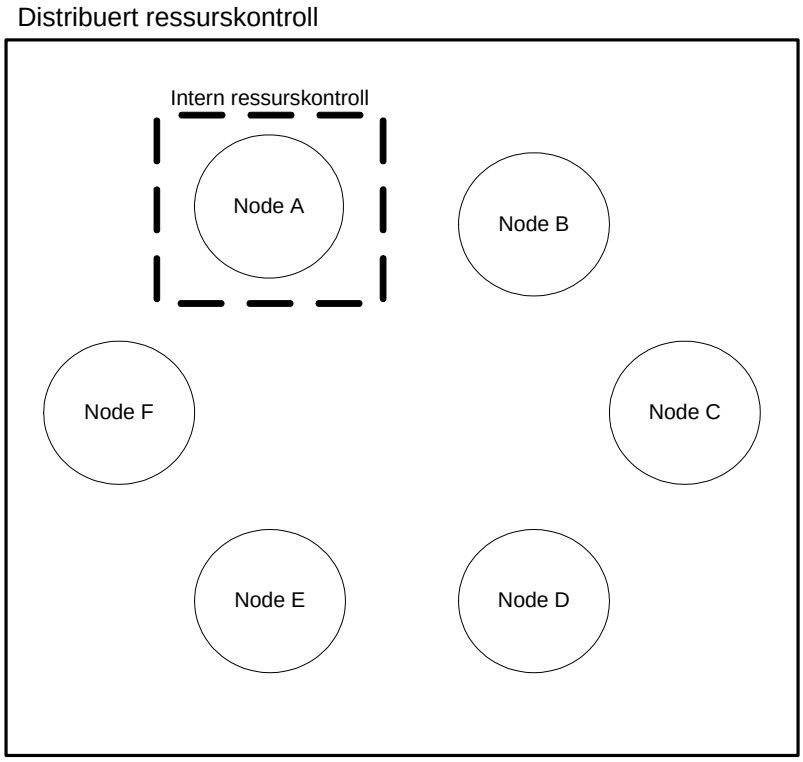


Figur 3.3: Abort-prosessering i ClustRa

4 Intern ressurskontroll

Med intern ressurskontroll mener vi forståelse av egen lastsituasjon for en node. Formålet med en slik ressurskontroll er å unngå overbelastning av noden ved å avvise transaksjoner som kan medføre for stor belastning.

I figur 4.1 har vi skissert forskjellen på intern og distribuert ressurskontroll. Intern ressurskontroll ser på forhold kun på noden, slik som figuren viser for node A med den stiplete linjen. Distribuert ressurskontroll ser derimot på situasjonen til alle noder samtidig, illustrert med den ytterste heltrukne linjen.



Figur 4.1: Forskjell på intern og distribuert ressurskontroll

I dette kapitlet vil vi først presentere hvilke tilstander en node kan få og hva hver enkelt innebærer. Her ser vi også på hvordan disse tilstandene kan oppdateres og vi presenterer et eksempel. Til slutt presenterer vi de løsningsforslagene for å oppnå intern ressurskontroll som vi har valgt å implementere.

4.1 Interne tilstander

I dette underkapitlet vil vi se nærmere på en nodes interne tilstand. Først vil vi definere hvilke tilstander en node kan ha. Deretter ser vi på endringsstrategier for hvordan en node kan endre tilstand. Til slutt vil vi komme med et oppsummerende eksempel.

4.1.1 Tilstandsdefinisjoner

Vi har definert tre ulike tilstander som er relevante for nodene i vårt system. Vi har ikke tatt hensyn til egne tilstander for repair eller take over her, da vi ikke ser på dette i vår simulering. Setting av tilstand på en node vil gjøres ut i fra grenseverdier til en av metodene som presenteres i kapittel 4.2. Hver enkelt tilstand er listet opp sammen med en definisjon i tabell 4.1.

Tabell 4.1: Tilstandsdefinisjoner

Tilstand	Definisjon
Opptatt	En node er opptatt om lasten på noden har oversteget en øvre grense. Dette vil medføre at noden ikke vil motta nye transaksjoner.
Tungt belastet	Tungt belastet vil medføre at noden har mye arbeid, men at den ikke er overbelastet. I denne tilstanden kan noden motta mindre transaksjoner eller transaksjoner med høy prioritet. Vanlige transaksjoner vil bli avvist.
Ledige ressurser	Om en node er i tilstanden ledige ressurser, vil noden ha tilstrekkelige ressurser til å ta i mot alle typer transaksjoner. Følgelig vil ingen transaksjoner avvises her.

4.1.2 Strategier for tilstandsendring

En sentral problemstilling for å oppnå god intern ressurskontroll er hvor ofte hver node skal endre sin tilstand ut i fra den lasten noden opplever. I tabell 4.2 har vi definert de endringsstrategier vi har funnet relevante og aktuelle.

Tabell 4.2: Strategier for tilstandsendring

Endringsstrategi	Forklaring
Hukommelsesløs	Hukommelsesløs tilstandsendring vil medføre at tilstanden endres med én gang de forskjellige metodene (presentert i kapittel 4.2) passerer grenseverdiene for de ulike tilstandene. På denne måten vil nodens tilstand alltid reflektere lasten på noden ved siste lastmåling.
Historiebasert	Historiebasert tilstandsendring er at endringen ikke kun er basert på gjeldene belastning på noden, men ser nåværende situasjon i sammenheng med hvordan nodens last har vært i det siste. Her vil man da endre tilstand kun når noden har endret lastbildet over lengre tid. Hvor lang tid man skal se bakover blir her en sentral parameter å bestemme.

Med hukommelsesløs oppdatering kan vi oppleve at noder jobber i rykk og napp. Vi kan tenke oss en lett belastet node som får for mye å gjøre. Den vil da endre sin tilstand til *opptatt*, og stenge ute nye transaksjoner. Etter en tid vil noden ha utført de transaksjonene den hadde, og tilstanden vil settes tilbake til *ledige ressurser*. Det kan da tenkes at noden enda en gang vil motta en stor arbeidsmengde for så å gå til *opptatt*. Ved å benytte den historiebaserte endringsstrategien vil man kunne begrense denne rykk og napp-oppførselen. Et problem som kan oppstå ved bruk av den historiebaserte oppdateringsstrategien, er at man tar med for mange verdier i beregningene. Dette kan føre til at en travel periode blir mindre framtredende siden gjennomsnittet over måleperioden blir lavt, og dermed kan man risikere å tillate nye transaksjoner å starte i situasjoner der systemet er tungt belastet.

4.1.3 Eksempel

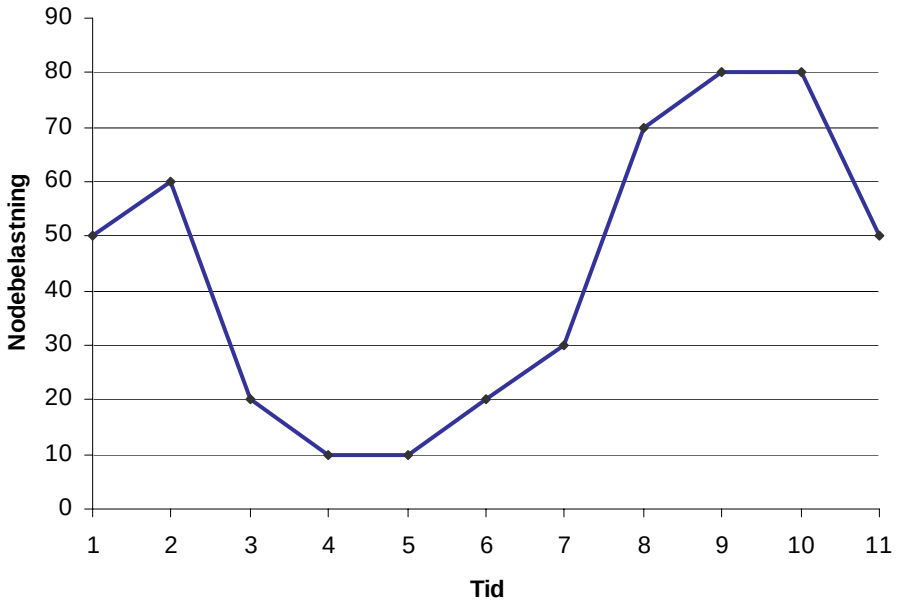
I dette delkapitlet vil vi komme med et eksempel som viser hvordan vi ønsker å bruke tilstander og tilstandsendringer i våre løsninger.

Vi vil her se på en bestemt node i et system som har definert parametere for tilstander slik tabell 4.3 viser. Verdi-kolonnen angir her verdiene for en tilfeldig metode for intern ressurskontroll og representerer prosentvis last på noden. Her har vi satt opp grenser slik at en node er opptatt om belastningsparameteren har verdi over 60, mens den regnes som tungt belastet for verdier mellom 40 og 60. Ved verdier under 40 vil den ha tilstanden *ledige ressurser*.

Tabell 4.3: Eksempel på tilstandsgrenser

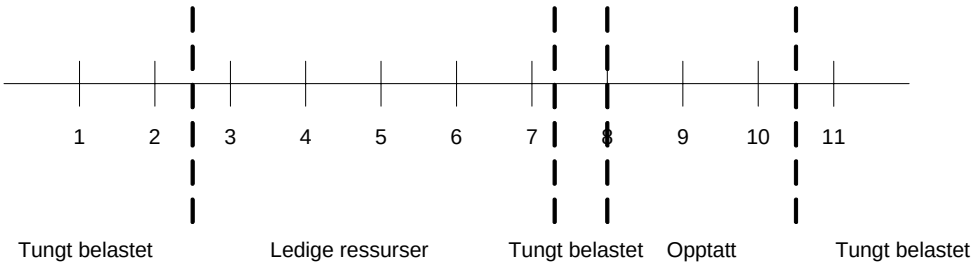
Tilstand	Verdi i %
Opptatt	> 60
Tungt belastet	> 40 og < 60
Ledige ressurser	< 40

I figur 4.2 har vi illustrert hvordan belastningen har endret seg over tid for en tenkt node. Vi ser her tidsaksen langs x-aksen, mens y-aksen har verdier som indikerer nodens belastning i øyeblikket.



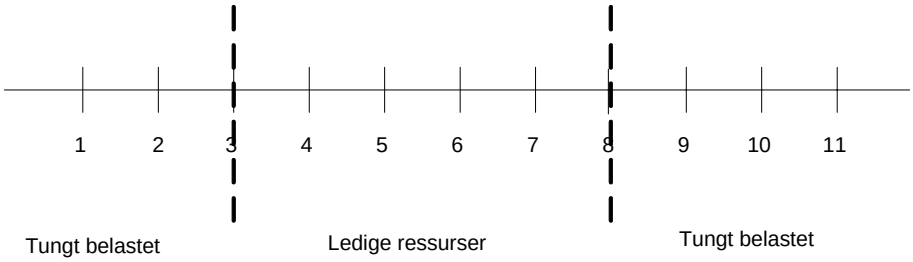
Figur 4.2: Eksempel på nodebelastning

Hvis vi nå velger strategien for hukommelsesløs oppdatering, slik vi presenterte i kapittel 4.1.2, vil vi få tilstander som figur 4.3 viser. Figuren følger samme tidslinje som figur 4.2 og har stiplede linjer på tvers ved tilstandsendringer. Disse tilstandsendringene vil følgelig følge nodens belastningsendringer slavisk, illustrert med grafens endringer i figur 4.2.



Figur 4.3: Tilstander ved hukommelsesløs tistandsendring

Ser vi derimot på samme eksempel, men med historiebasert oppdatering, vil vi få en situasjon slik figur 4.4 viser. Her ser vi hvordan den gjør oppdateringer senere enn ved hukommelsesløs oppdatering og at det blir to tilstandsendringer i stedet for fire. Dessuten går ikke noden nå inn i tilstanden *opptatt* mot slutten av tidsperioden vi ser på, siden denne metoden er basert på historiske data i tillegg til de nåværende. En ulempe med at systemet ikke går over i tilstanden *opptatt*, kan være at belastningstoppen ikke blir registrert. Dette kan føre til at systemet tar imot nye transaksjoner i en periode det har for mye å gjøre, slik at det fører til at systemet blir overbelastet. Dersom systemet går over i tilstanden *opptatt* for tidlig, vil dette kunne føre til at transaksjoner avvises unødvendig. I dette eksemplet har vi beregnet tilstand ut i fra de tre siste tidsenheter ved historiebasert oppdatering.



Figur 4.4: Tilstander ved historiebasert tilstandsendring

4.2 Løsningsforslag

Vi vil i dette underkapitlet presentere de løsninger som vi finner aktuelle for å utføre intern ressurskontroll. Her vil vi kun presentere de løsninger som vi har valgt å implementere i vår simulator. I prosjektrapporten fra i høst presenterte vi flere alternativer som er utelatt her. Grunnen til at de er utelatt er at vi ikke har mulighet til å implementere så mange løsninger på grunn av tiden vi har til rådighet. Vi har valgt løsningene vi presenterer her fordi de sier mye om lasten på en node og samtidig er enklere å implementere sammen med simulatoren for øvrig, enn de andre løsningsalternativene vi presenterte i høstens rapport.

4.2.1 Prosessor

Ved å måle prosessorbruk på en node, vil man kunne avgjøre belastningen på denne noden. Prosessorbelastningen vil endre seg hele tiden, slik at denne størrelsen kan det være interessant å se på over tid. Hvis den gjennomsnittlige prosessorbruken er høy, vil det være nærliggende å anta at noden er høyt belastet.

Vi vil her måle andelen av tiden prosessoren er opptatt og sette tilstandene *opptatt* og *tungt belastet* når denne andelen av tiden overstiger gitte grenser. Verdiene på disse grensene bestemmes i kapittel 7 sammen med øvrige ressurskontroll-parametere.

4.2.2 Diskkø

Diskkøen vil være en sentral parameter for lasten til en node i de tilfeller hvor man får mange diskaksesser og hvor antall diskaksesser utgjør en betydelig del av de totale antall dataaksesser i systemet. Dermed vil man kunne få situasjoner hvor disken blir systemets flaskehals framfor prosessoren.

Vi ser her for oss en løsning der diskkøens lengde vil avgjøre nodens tilstand. Er denne køen for lang vil det være et tegn på at mange transaksjoner er inne i systemet og venter på disken og vil dermed gi dårlig kjøretid for nye transaksjoner. Valg av grense for diskkøen gjøres i kapittel 7.

4.2.3 Kombinasjon av prosessor og diskkø

Denne løsningen vil være en kombinasjon av løsningene presentert i kapittel 4.2.1 og kapittel 4.2.2 som er henholdsvis å måle prosessorbruken og diskkøen.

Vi ser her for oss at man tar hensyn til begge disse målene når man setter tilstanden til en node og setter tilstand ut i fra den målingen som viste at systemet var mest belastet.

5 Distribuert ressurskontroll

Distribuert ressurskontroll omhandler hvordan nodene i ClustRa kan oppnå global lastforståelse og ressurskontroll over de distribuerte ressursene. Hovedformålet med dette er å kunne avvise transaksjoner før de sendes ut til de deltakende noder om en eller flere av nodene kan komme til å abortere transaksjonen som en følge av overbelastning. På denne måten vil man unngå ressurskrevende tilbakerulling og dermed spare systemet for unødvendig belastning.

I dette kapitlet vil vi først presentere de overordnede løsningsmål for distribuert ressurskontroll. Videre vil vi presentere de løsninger vi vil simulere. Disse løsningene er basert på vårt studium i høst hvor vi kom fram til seks løsningsalternativer. Vi har valgt å jobbe videre med tre løsningsalternativer:

- Distribuert nodetabell (Tilsvarer løsnigen nodetabell fra høstens rapport)
- Sentralisert nodetabell (Tilsvarer løsningen sentralisert styring fra høstens rapport)
- Transaksjonsklarering

Løsningene vi presenterte i høstens rapport som baserer seg på gruppering av nodene er utelatt, delvis på grunn av at dette krever et stort antall noder og at dette er vanskelig å simulere og prøve ut i praksis for å få satt riktige parametere. Vi har også valgt å konsentrere oss om færre løsninger og heller gå mer i detalj på dem, enn å prøve å rekke over mange uten samme detaljnivå.

Vi vil i løsningsbeskrivelsene i dette kapitlet fokusere på de mer generelle aspektene. En nærmere beskrivelse av hvordan løsningene vi presenterer er implementert i vår simulator er å finne i kapittel 6.3.2.

5.1 Løsningsmål

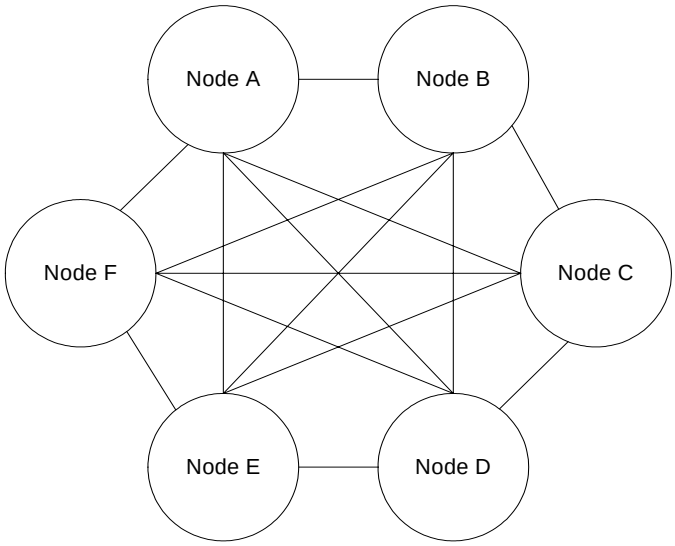
Vi har definert følgende hovedmål med undermål for en løsning av problemstillingen med distribuert ressurskontroll:

- M1 - Lav systembelastning:
  - M1A - Færrest mulig meldinger
  - M1B - Færrest mulig transaksjoner aborteres som en følge av systemoverbelastning
  - M1C - Minst mulig bruk av prosessor for å utføre ressurskontroll
  - M1D - Ressurskontrollen skal tilby maksimal ressursutnyttelse uten at maksimal responstid overstiges
  - M1E - Ressurskontrollutførelsen skal være skalerbar slik at den totale systembelastningen skal øke minimalt ved en systemekspansjon
- M2 - Kvalitetskrav til transaksjoner:
  - M2A - Transaksjoner skal kunne ha en maksimal responstid som holdes
  - M2B - Transaksjoner skal avvises om maksimal responstid ikke kan overholdes

5.2 Distribuert nodetabell

Distribuert nodetabell er en løsning der hver enkelt node har en tabell med ett innslag for alle andre noder i systemet. I denne tabellen vil det ligge informasjon om lastsituasjonen til de enkelte nodene. På denne måten vil en node som mottar en transaksjon ut i fra denne tabellen kunne avgjøre om en transaksjon skal tas i mot og gjennomføres på de deltakende noder, eller avvises.

For å illustrere denne løsningen, vil vi se på et tenkt nodesystem med seks noder. I figur 5.1 har vi illustrert et slikt system, hvor vi ser at alle noder har kommunikasjonslinjer med alle andre noder i systemet.



Figur 5.1: Nodetabell

Om vi ser på nodetabellen til node A i dette systemet, kan tabellen se ut som eksemplet presentert i tabell 5.1. Her ser vi lastbildet node A har over de fem andre nodene i systemet. Følgelig kan node A her godta en transaksjon som skal involvere node F, da noden har ledige ressurser i følge tilstandsinformasjonen. Kommer derimot en transaksjon som krever node D, vil denne bli avvist da node D står som *opptatt*.

Tabell 5.1: Eksempel på nodetabell for node A i figur 5.1

Node	Tilstand
B	Tungt belastet
C	Ledige ressurser
D	Opptatt
E	Opptatt
F	Ledige ressurser

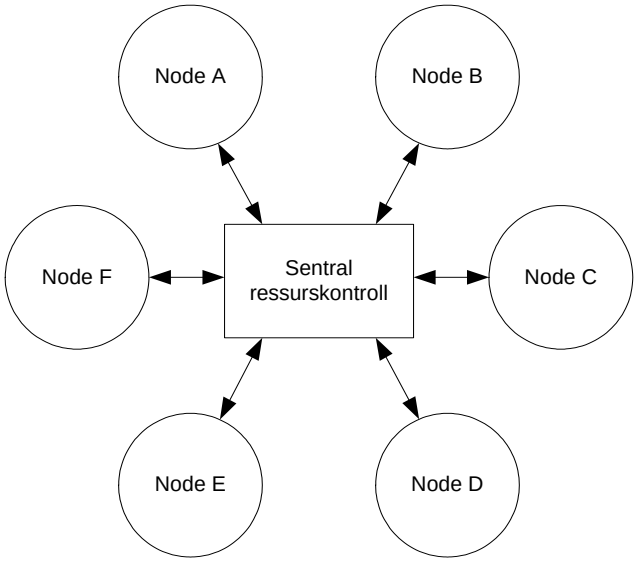
Det sentrale her i forhold til en løsningsvurdering er hvordan og hvor ofte en slik tabell skal oppdateres. Avveiningen som må gjøres er mellom en sikker vurdering av lastsituasjonen mot kostnaden ved å ta gale avgjørelser, og kostnaden ved å oppdatere tabellen ofte mot mer sjelden. En viktig faktor er også hvordan selve oppdateringen utføres med antall meldinger det medfører. Denne oppdateringen kan skje med et av alternativene periodiske oppdateringer eller hendelsesbaserte oppdateringer. Periodisk oppdatering vil innebære at lastbildet sendes med jevne mellomrom, mens hendelsesbaserte oppdateringer vil innehære at spesielle hendelser trigger lastoppdateringen. Vi har valgt å implementere løsningen med periodisk oppdatering kombinert med en sjekk av at tilstanden har endret seg på noden. Har ikke tilstanden endret seg fra forrige gang lastbildet ble sendt, vil ikke noden sende noen oppdatering til de andre nodene. På denne måten vil vi spare systemet for unødvendige meldinger. Vi viser til høstens rapport [1] for å studere alternativene periodisk og hendelsesbasert oppdatering nærmere.

Meldingsutvekslingen for å spre lastinformasjonen kan her skje ved kringkasting, ringbasert distribusjon, unikasting, multikasting, piggybacking eller piggybacking med timeout-funksjonalitet. Disse alternativene ble beskrevet i høstens rapport og vi viser til denne rapporten for en beskrivelse av de ulike mulighetene. Vi har valgt å gå videre med løsningen med unikasting i vår

simulering. Dette vil innebære at hver enkelt node sender egne meldinger til de andre nodene om sin lastsituasjon.

5.3 Sentralisert nodetabell

Sentralisert nodetabell er en løsning der en sentral enhet står for ressurskontrolleringen. En slik løsning er skissert i figur 5.2, hvor vi har den sentrale enheten i midten. Dette er en løsning der denne sentrale enheten er en av de vanlige nodene i systemet. I figur 5.2 vil dette innebære at en av nodene A til F tar rollen som den sentrale enheten. Dermed vil denne noden oppleve økt belastning i forhold til de andre nodene. Vi har ikke gitt denne noden noen ekstra ressurser i vår simulering. Dette vil være en mulig endring i et reelt system for å unngå at denne noden blir en flaskehals i systemet.



Figur 5.2: Sentralisert nodetabell

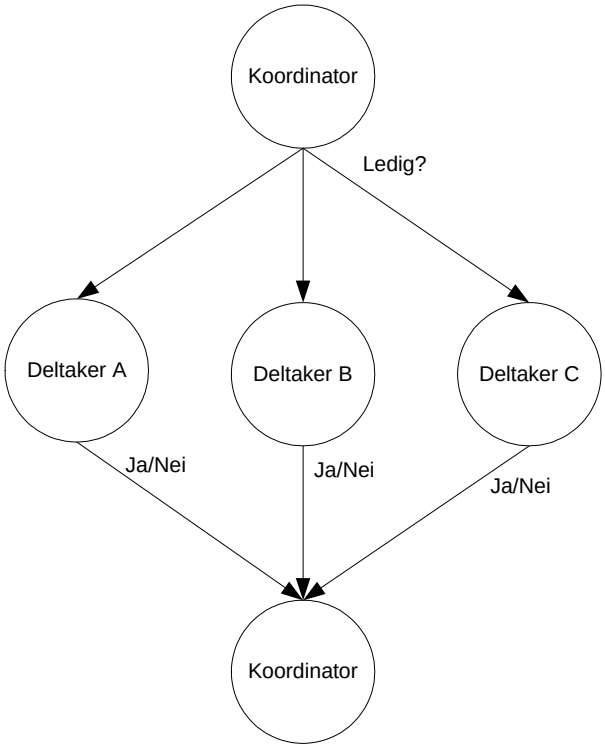
Løsningen sentralisert nodetabell krever at de forskjellige nodene rapporterer sin lastutvikling til den sentrale ressurskontrollenheten. Denne oppdateringen vil skje på samme måte som vi beskrev for distribuert nodetabell i kapittel 5.2.

Også her vil oppdateringshyppigheten mot økt systemoverhead være en sentral avveining man må ta stilling til. Den eneste forskjellen i forhold til systemoverbelastning med denne løsningen i forhold løsningene beskrevet i kapittel 5.2 er at antall meldinger blir færre ved oppdatering, da man alltid kun rapporterer til en bestemt enhet og ikke til alle eller flere noder i systemet. Dermed vil unikasting mot denne enheten være eneste alternative kommunikasjonsmåte. Derimot vil man her få to meldinger for hver transaksjon som skal klareres, noe man ikke får ved den distribuerte løsningen.

Når de ulike nodene nå mottar transaksjoner som skal utføres, vil de sjekke lastsituasjonen på de nodene som transaksjonen involverer ved å spørre den sentrale enheten. Basert på svaret denne enheten gir, vil transaksjonen enten startes eller avvises.

5.4 Transaksjonsklarering

Transaksjonsklarering er en løsning der noden som mottar en transaksjon sjekker tilstanden til de deltakende noder i transaksjonen eksplisitt. Har alle nodene tilstrekkelig ledig kapasitet vil transaksjonen startes, ellers avvises den. I figur 5.3 har vi illustrert denne løsningen. Koordinator er her den noden som mottar transaksjonen. Den sender en melding *Ledig?* til alle noder den ser skal delta i transaksjonen. Hvis alle svarer *Ja*, utføres transaksjonen. Hvis en eller flere svarer *Nei*, vil transaksjonen avvises. Navnet på disse meldingene er ikke de som er brukt i den reelle implementasjonen, men brukt her for å gjøre illustrasjonen enklere.



Figur 5.3: Transaksjonsklarering

Denne løsningen vil med andre ord innebære 2n antall meldinger, eller 2(n-1) meldinger om den mottakende noden selv skal delta i transaksjonen, før transaksjonen starter, der n er antall deltakende noder. Denne løsningen er sikker med hensyn på å unngå å rulle tilbake transaksjoner som en følge av ressurskontroll er utført med for gamle data som grunnlag.

For å unngå at denne klareringsprosessen skal ta for lang tid, har koordinatoren en timeout som gjør at den avviser transaksjoner om den ikke har mottatt respons fra alle involverte noder innen timeout-tiden.

6 Simulatoren

I dette kapitlet vil vi beskrive hvordan simulatoren vår er oppbygd. Simulatoren simulerer transaksjonsutførelse i ClustRa og kan utføre ressurskontroll på nodene i systemet og avvise nye transaksjoner om en eller flere noder i systemet er for tungt belastet.

I simulatoren er variable parametere som kan settes for å påvirke simuleringen, samlet i egne innstillingsfiler. Alle disse filene med parametere er forklart og beskrevet i vedlegg A.

Vi har valgt å benytte Desmo-J versjon 2.0.1 som rammeverk. Desmo-J er et simuleringsrammeverk skrevet i Java og er uviklet ved universitetet i Hamburg. En beskrivelse av Desmo-J som simuleringsrammeverk kan leses i vedlegg B.

Dette kapitlet starter med å se på hvordan vi har utviklet denne simulatoren i forhold til utviklingsmiljø, før vi beskriver simulatorens oppbygning. Videre ser vi på implementeringen av ressurskontroll, før vi ser på utføringen av oppdateringstransaksjoner og til slutt refragmenteringstransaksjoner.

6.1 Utviklingsmiljø

Vår simulator er skrevet i programmeringsspråket Java. Vi har programmert etter versjon 1.5.0 av J2SE. En nærmere forklaring av denne utgaven av J2SE er å finne i [6]. API-en til Java 1.5.0 er å finne i [7].

Vi har fulgt standarden *Code Conventions for the Java Programming Language* i vår programmering og i vår kommentering av spesielle deler av koden. Denne standarden er beskrevet i [8]. På nettstedet til denne referansen kan standarden lastes ned som html-, postskript- og PDF-fil.

Alle metoder, attributter og klasser er dokumentert med Javadoc. Vi følger standarden beskrevet i [9] i vår dokumentasjon.

Vårt utviklingsarbeid er gjort i IDE-verktøyet[1] Eclipse. Vi har brukt versjon 3.1.0 av programvaren. Vi har også brukt CVS for å håndtere versjonkontroll på vår kildekode. Vi har her brukt CVS-verktøyet som er tilgjengelig fra institutt for datateknikk ved NTNU sammen med den integrerte CVS-modulen til Eclipse.

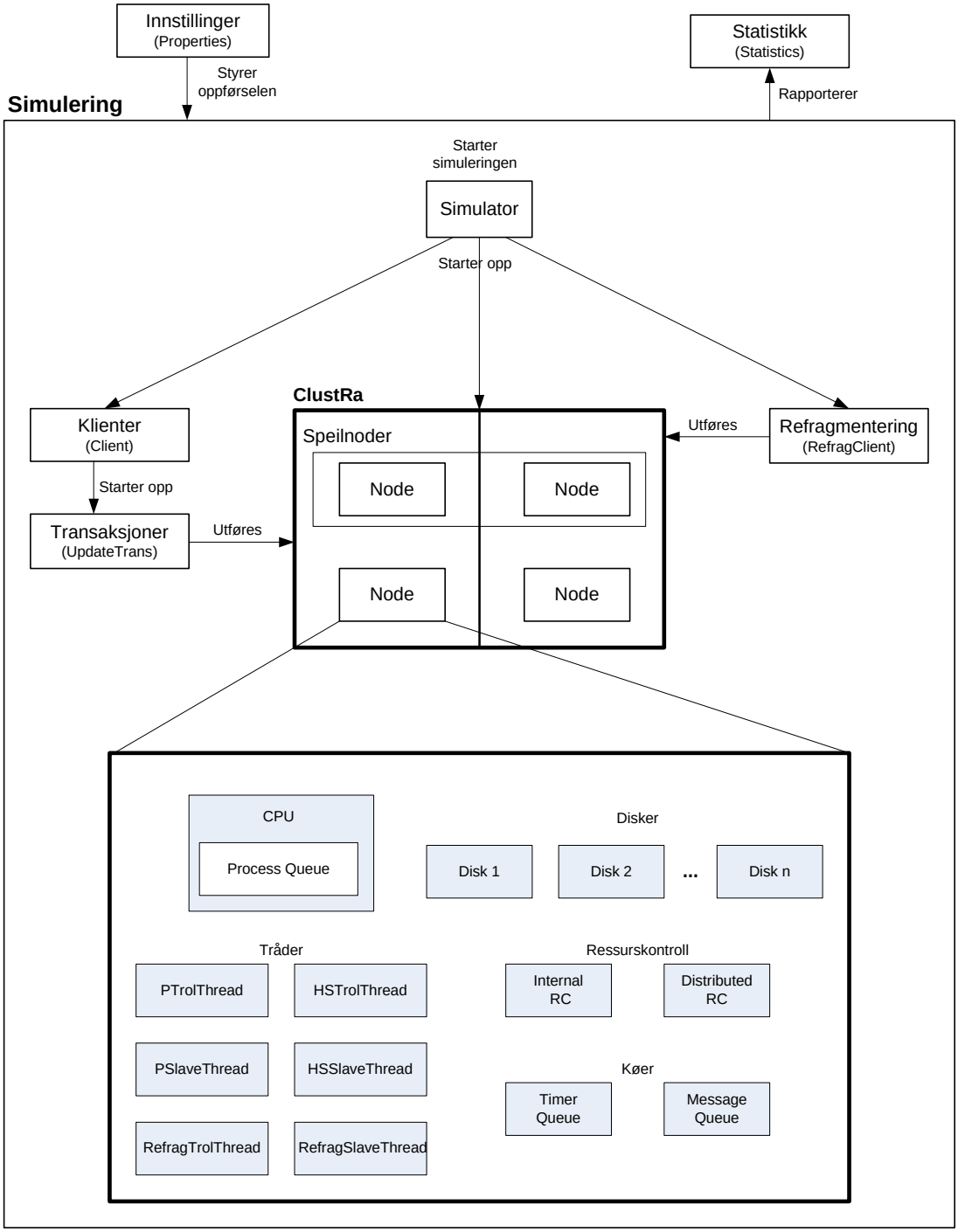
En nærmere beskrivelse av hvordan vår simulator settes opp og kjøres er forklart i vedlegg C.

6.2 Simulatorens oppbygning

Simulatoren består av aktive enheter som utfører handlinger i systemet og støtteklasser som holder på informasjon og referanser, og er ansvarlig for at rammeverket fungerer.

Før vi ser på disse to hoveddelene av simulatoren vil vi se på den overordnede oppbygningen. I figur 6.1 er en oversikt over hovedelementene simulatoren og hvordan de kommuniserer under simulering.

1. *IDE er en fortkortelse for Integrated Development Environment og brukes om verktøy som har innebygde hurtigfunksjoner for programmering med ulike programmeringsspråk*



Figur 6.1: Oppbygningen av en node i simulatoren

I denne figuren ser vi at klassen `Simulator` starter opp hele simuleringen. Denne klassen vil deretter starte opp et nødvendig antall klienter for både transaksjoner (instanser av klassen `Client`) og refragmentering (instanser av klassen `RefragClient`) og alle nodene til systemet som simuleres. Disse nodene er illustrert med rammen *ClustRa*, hvor vi ser et system av fire noder, der to og to noder er hverandres speilnoder. Transaksjonsklientene starter opp transaksjoner som utføres av nodene i systemet. På samme måte utføres refragmenteringen som styres av refragmenteringsklienten. Nederst i figuren ser vi hvordan en node er bygd opp av følgende sentrale deler:

- CPU som er simulatorens prosessor som utfører handlinger og får simuleringen til å gå framover. Den har en viktig kø av prosesser som skal utføre handlinger på CPUen.
- Disker som holder på dataene til noden.
- Tråder som representerer primær og hot standby `TrolThread`, primær og hot standby `SlaveThread` og trådene `RefragTrolThread` og `RefragSlaveThread` som utfører refragmentering på noden. Hver node har ett slikt sett med tråder som er felles for alle transaksjoner den er involvert i.
- Ressurskontroll-objekter som utfører intern og distribuert ressurskontroll.
- Køer for meldinger og timere knyttet til noden.

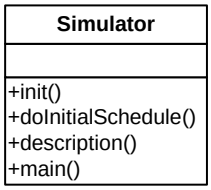
Hver enkelt av disse delene på en node vil forklares mer detaljert under beskrivelsen av de aktive enhetene og støtteklassene.

Øverst på figuren ser vi at det er innstillingene i properties-filene som styrer hvordan en konkret simulering vil bli utført. Av en gjennomført simulering vil det komme statistikkrapporter som håndteres av en egen statistikk-klasse.

Vi vil videre i dette underkapitlet først se på klassene `Simulator` og `Node` som henholdsvis starter opp simuleringen og som holder på informasjonen knyttet til en node. Deretter går vi nærmere inn på de aktive elementene og støtteklassene.

6.2.1 Simulator

Klassen `Simulator`, som arver `desmoj.core.simulator.Model`, setter i gang hele simuleringen. De viktigste metodene i `Simulator` er vist i figur 6.2.



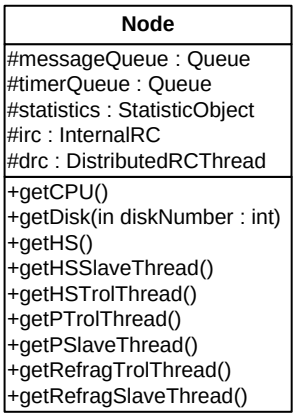
Figur 6.2: Klassediagram Simulator-klassen

I denne klassen blir alle objektene i simulatoren opprettet, eller de blir opprettet av andre klasser som simulatoren har opprettet. `Simulator` har også et statistikkansvar som består av å skrive alle commit-tider til fil i den rekkefølgen transkasjonene i systemet fullfører. Viktige metoder i `Simulator` er:

- `init()` som oppretter alle referanser som `Simulator` trenger.
- `doInitialSchedule()` som aktiverer de objektene som skal være aktive ved starten av simuleringen, slik at den kan starte.
- `description()` som returnerer en tekstlig beskrivelse av simulatoren.
- `main()` som starter hele simuleringen, styrer lengden av simuleringen og vil til slutt stå for skriving av statistikkføring til fil.

6.2.2 Node

Et sentralt punkt i simulatoren er klassen `Node`. Noden har referanse til `CPU` og `Disk` som er plassert på noden, og den har referanse til køer for timere og innkommende meldinger. Videre har noden oversikt over alle trådobjektene som brukes ved transaksjonsprosessering, refragmentering og objektene som brukes til å utføre intern og distribuert ressurskontroll. På hver node finner man også en referanse til den noden som er speilnode. De objektene som inngår i simuleringen vil ha referanse til en node som sin egen node. Via denne referansen vil objektene finne referanser til andre objekter på samme node.



Figur 6.3: Klassediagram Node-klassen

I figur 6.3 ser vi et klassediagram med de mest sentrale attributter og metoder i klassen `Node`. De mest brukte metodene i denne klassen er:

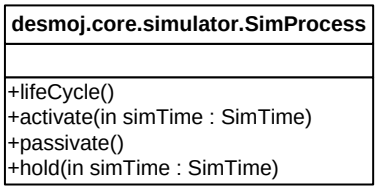
- `getCPU()` som returnerer CPU-objektet på noden, og brukes av trådene som kjører på noden for å få tak i det CPU-objektet som skal aktiveres.
- `getDisk(int diskNumber)` som henter ut disk-objektet med angitt nummer
- `getHS()` som returnerer denne nodens speilnode
- Metodene `getHSSlaveThread()`, `getHSTrolThread()`, `getPTrolThread()`, `getPSlaveThread()`, `getRefragTrolThread()` og `getRefragSlaveThread()` returnerer objektene for henholdsvis `HSSlaveThread`, `HSTrolThread`, `PTrolThread`, `PSlaveThread`, `RefragTrolThread` og `RefragSlaveThread` fra denne noden.

Nodeklassen har i tillegg til noen variabler som har synlighetsnivå protected:

- `messageQueue` er køen over alle innkommende meldinger til node. En ny melding blir lagt til ved å kalle `messageQueue.insert(Entity entity)`.
- `timerQueue` er en kø for de timerene som skal kjøres på noden. Ved å kalle `timerQueue.insertNewTimer(Timer timer)` vil man legge til en ny timer for kjøring.
- `statistics` er referanse til statistikk-klassen som kalles blant annet hver gang en transaksjon fullføres.
- `irc` er referanse til nodens interne ressurskontroll-objekt
- `drc` er referanse til nodens distribuerte ressurskontroll-objekt

6.2.3 Aktive enheter

I dette delkapitlet beskriver vi de aktive enhetene som er CPU, disk og tråder. Felles for disse enhetene er at de arver `desmoj.core.simulator.SimProcess` som er vist i figur 6.4.



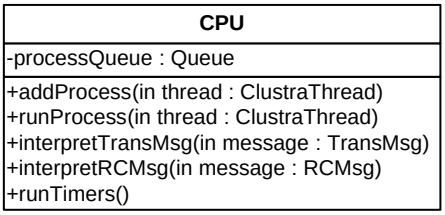
Figur 6.4: Klassediagram desmoj.core.simulator.SimProcess-klassen

Enhetene får da følgende metoder som vi benytter:

- `lifeCycle()` beskriver objektet sin livssyklus. Koden i denne metoden vil bli kjørt første gangen objektet aktiveres. Neste gang objektet aktiveres, vil man fortsette å kjøre der programflyten var da objektet ble deaktivert.
- `activate(SimTime simTime)` aktiverer objektet på tidspunktet nåtid + `simTime`-parameteren. Dersom `simTime = 0`, vil objektet aktiveres på det tidspunktet metoden blir kalt.
- `passivate()` vil deaktivere objektet på det tidspunktet metoden kalles.
- `hold(SimTime simTime)` vil sette objektet på pause for det tidsrommet `simTime` angir. Dette gir samme effekt som å først kalle metoden `activate(SimTime simTime)` for så å kalle metoden `passivate()`.

CPU

CPUen er en aktiv enhet i simulatoren, og de viktigste metodene i CPUen er vist i figur 6.5 og forklart i den etterfølgende punktlisten:



Figur 6.5: Klassediagram CPU-klassen

- `addProcess(ClustraThread thread)` legger en ny tråd inn i listen over tråder som skal bli tildelt kjøretid.
- `runProcess(ClustraThread thread)` aktiverer den tråden som skal kjøres, og deaktiverer CPUen. Når tråden er ferdig, vil CPUen registrere hvor lang tid som ble brukt og oppdatere statistikk-klassen.
- `interpretTransMsg(TransMsg message)` vil tolke en transaksjonsmelding og utføre metodekall på bakgrunn av hvilken meldingstype som tolkes.
- `interpretRCMsg(RCMsg)` tolker en ressurskontrollmelding, og utfører ressurskontrollhandlinger ut fra hvilken melding som blir tolket.
- `runTimers()` plukker ut den første timeren i køen, og aktiverer den aktuelle tråden som skal kjøre. CPUen vil oppdatere statistikk-klassen med tiden det tok å kjøre trådens timerkode.

I tillegg til disse metodene, har CPUen også en referanse til en `desmoj.core.simulator.ProcessQueue`. Denne køen inneholder alle tråder som skal tildeles kjøretid. Siden vi har én tråd per node som har ansvaret for transaksjonsutførelsen, vil køen kunne inneholde flere innslag av samme objekt. På grunn av dette fungerer simulatoren som om hver transaksjon ble kjørt i en egen tråd. I kode-eksempel 6.1 er livssyklusen til CPUen vist, og det kommer fram at vår implementasjon av CPUen er begrenset til å være en scheduler med fire oppgaver.

```
while (true) {
    // Kjør tråder fra listen over aktive tråder til trådene blir ferdige, eller de får diskavbrudd.
    // Les én melding fra meldingskøen.
    // Kjør timere som har kjøretid <= nåtid.
    // Ressurskontroll: måle CPU og/eller diskkø.
}
```

Kode-eksempel 6.1: CPUens livssyklus

For å kunne kjøre tråder har CPUen en lenket liste med de trådene som skal ha kjøretid. Trådene blir kjørt i den rekkefølgen de ligger i køen, ved å ta ut den første tråden og deretter kalle

`runProcess(ClustraThread thread)` med tråden som parameter. Simulatoren er laget slik at en tråd får lov å kjøre til den er ferdig med den instruksjonen den holdt på med, eller til den må aksessere disken. I disse to tilfellene vil CPUen skifte til den etterfølgende tråden i køen. CPUen vil i vårt tilfelle aktivere tråden med det samme, ved å kalle metoden `activate(SimTime simTime)` på tråden med `simTime = 0`. Tiden angir hvor lang tid det skal gå i simuleringen før tråden aktiveres. Etter å ha aktivert tråden, vil CPUen kalle `passivate()` på seg selv, som betyr at den legger seg til å sove. Etter at tråden er ferdig med sin instruksjon eller må vente på disken, vil tråden kalle `activate(SimTime simTime)` på CPU-objektet med `simTime = 0`, og deretter kalle `passivate()` på seg selv. På denne måten skifter CPUen og trådene på å være aktive under simuleringen.

CPUen har tilgang til en meldingskø for innkommende meldinger til noden. CPUen vil hver runde i livssyklusen i kode-eksempel 6.1 lese én melding fra denne køen. Hvis meldingen er av typen `TransMsg`, vil metoden `interpretTransMsg()` bli kalt. Er meldingen derimot av typen `RCMsg`, vil CPUen kalle metoden `interpretRCMsg()`. Meldingen kan enten inneholde en liten instruksjon som CPUen vil utføre mens den tolker meldingen, eller meldingen kan inneholde instruksjoner i forhold til hvordan transaksjonsprosesseringen skal fortsette. Hvis meldingen er av den siste typen, vil CPUen legge til mottakertråden i listen over tråder som skal ha kjøretid. Tråden vil dermed kunne fortsette sin utførelse neste gang CPUen kjører tråder. Tiden CPUen bruker for å prosessere en melding, beregnes ut fra ligningene ligning 6.2 og ligning 6.3 side 28.

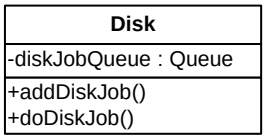
Simulatoren har en mekanisme for timerkjøring. Ved å kalle metoden `runTimers()`, vil CPUen kjøre timere inntil kjøretiden på alle timerene som ligger i køen overstiger nåtiden i simuleringen. Hvis en timer resulterer i at en tråd må tilordnes kjøretid, vil tråden få kjøretid mens timeren utføres. I dette tilfellet vil CPUen aktivere tråden og fortelle den at den kjører som følge av en timer. Når tråden er ferdig med sin timerrutine, vil den igjen gi kontrollen tilbake til CPUen. Dette fungerer på samme måte som for vanlig kjøring av tråder. Timerene i simulatoren brukes blant annet til å sikre at meldinger mellom noder blir sendt på nytt.

Hvilke oppgaver CPUen skal utføre i forbindelse med den interne ressurskontrollen, avhenger av hvilken type intern ressurskontroll som er valgt. De ulike strategiene for å avgjøre nodens tilstand er omtalt under kapittel 4.2.

I de rundene i livssyklusen der CPUen ikke har noen arbeidsoppgaver, vil den telle opp ett tick i simulatortiden. Denne opptellingen gjenspeiler at det å løpe gjennom løkken en runde uten å utføre noe tar litt tid. I tillegg er simulatoren avhengig av denne opptellingen for at simuleringstiden skal gå framover dersom systemet ikke har noen aktive transaksjoner. 50 mikrosekunder er valgt som den tidsenheten CPUen teller opp når den er arbeidsledig. Verdien hentes ut fra innstillingsfil og parameteren har navnet `tickTime`. Denne verdien er større enn den tiden det ville tatt for en moderne CPU å gå gjennom denne løkken. For simuleringen sin del har ikke valget av denne verdien til 50 mikrosekunder noe å si for resultatene, siden ingen hendelser tar kortere tid en dette.

Disk

Simulatoren støtter at hver node har én eller flere disker knyttet til seg. Klassen `Disk` er vist i figur 6.6, der de viktigste metodene er tatt med.



Figur 6.6: Klassediagram Disk-klassen

Disken opererer uten innflytelse fra CPUen, slik at tråder kan kjøres på CPUen parallelt med at disken arbeider. De viktigste metodene i `Disk`-klassen er:

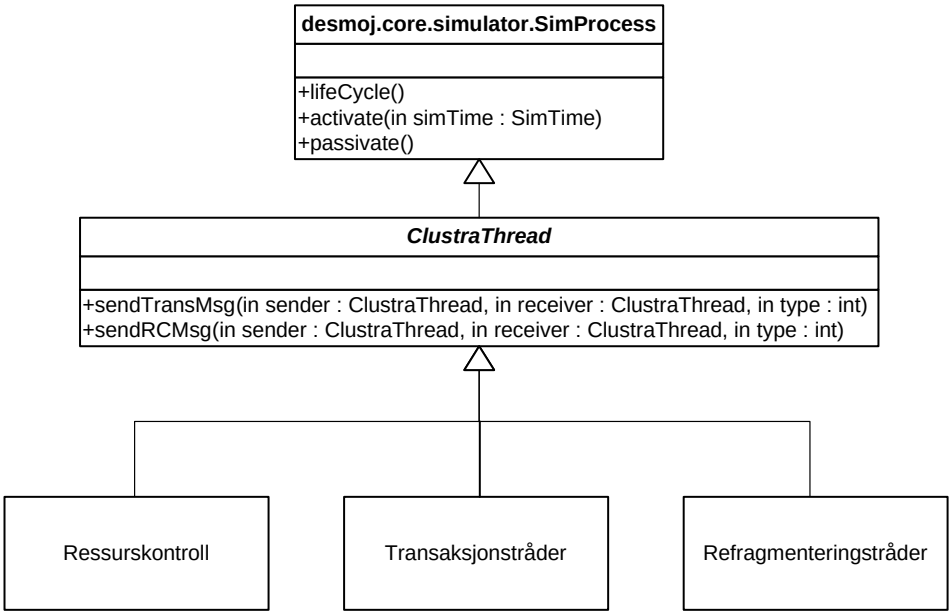
- `addDiskJob()`som oppretter en ny `DiskJob` og legger denne til i køen `diskJobQueue`. `DiskJob`-objektet holder orden på hvilken tråd som sendte jobben og hvilken transaksjon jobben gjelder for. Hvis disken er deaktivert når en diskjobb ankommer, vil denne metoden også aktivere disken omgående.

- `doDiskJob()` som utfører den første diskjobben i køen. Dersom `diskJobQueue` ikke er tom etter at jobben er utført, vil den automatisk fortsette med å utføre neste jobb. Hvis køen er tom vil metoden deaktivere disken.

Vi har valgt å representere disken som en kø av diskjobber, der tiden hver diskjobb tar er forenklet til å være 10 ms for en aksess mot en tilfeldig valgt blokk. Parameteren som bestemmer dette heter `mtWriteRecordDisk`. Hvis disken skal lese etterfølgende diskblokker, beregnes tiden til 1 ms per diskblokk på 16KB. Parameteren som bestemmer denne tiden, heter `mtWriteRecordDiskPr16KB`. Disken vil kalle metoden `hold(SimTime simTime)`, som setter disken som passiv, for så å bli aktivert etter tiden som gis som parameter. Etter at CPUen har mottatt melding fra disken om at en diskjobb er ferdig, er det CPUens oppgave å aktivere tråden som ventet på denne diskjobben.

Tråder

Simulatoren vår inneholder tråder. Tråder er enheter som utfører operasjoner når de blir tildelt kjøretid av CPUen. Hierarkiet for alle trådklassene er vist i figur 6.7.



Figur 6.7: Klassediagram Trådhierarki

Vi ser i figuren at vi har en abstrakt klasse `ClustraThread` som alle trådeklassene arver. I denne klassen finnes følgende to sentrale metoder som brukes under meldingssending:

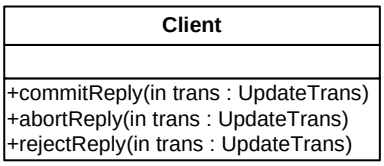
- `sendTransMsg(ClustraThread sender, ClustraThread receiver)` som oppretter en ny transaksjonsmelding og sender den til rett mottaker. Tråden venter en tid før melding sendes, for å simulere at meldingssending tar tid for CPUen.
- `sendRCMsg(ClustraThread sender, ClustraThread receiver)` som oppretter en ny ressurskontrollmelding og sender denne til rett mottaker etter at tråden har ventet den tiden det tar å sende meldingen.

Som vi ser av figur 6.7 er det tre typer tråder i vår simulator som arver ClustraThread:

- Ressurskontroll som er tråder som utfører ressurskontroll på de enkelte nodene. Dette beskrives nærmere i kapittel 6.3.
- Transaksjonstråder som implementerer normal transaksjonsutførelse. Dette er nærmere beskrevet i kapittel 6.4.
- Refragmenteringstråder som implementerer refragmentering. Hvordan dette utføres er nærmere beskrevet i kapittel 6.5.

Klient

Transaksjoner ankommer systemet ved at de blir startet av en klient. Et klassediagram for `Client` er vist i figur 6.8.



Figur 6.8: Klassediagram Client-klassen

Simulatoren har opprinnelig et en-til-en forhold mellom node og klient, slik at en klient sender en transaksjon til en bestemt node. En klient kan ha flere aktive transaksjoner samtidig, og antallet styres av parameteren `clientMPL` fra en av innstillingsfilene. Dersom `clientMPL` settes til -1, betyr dette at den ikke har noen øvre grense for antall aktive transaksjoner. Tiden mellom hver gang en klient sender en ny transaksjon til systemet følger en Poissonprosess[1]. Forventingsverdien til Poissonprosessen er bestemt av parameteren `mtBetweenArrivals`. Denne verdien angir hvor mange millisekunder klienten i gjennomsnitt skal vente før en ny transaksjon startes. Parameteren `clientSendToArbitraryNode` bestemmer om den nye transaksjonen skal sendes til en bestemt node i systemet, eller om den skal sendes til en vilkårlig node. Har parameteren verdi lik 0 vil man sende til en bestemt node, mens verdi lik 1 medfører at det sendes til vilkårlig node. Standardverdi på denne parameteren er 0.

For hver transaksjon som klienten avventer svar på fra systemet, vil den ha et oppslag i en lokal hash-tabell. I denne tabellen er transaksjonen lagret som nøkkel, og starttiden for transaksjonen er assosiert med denne nøkkelen. På denne måten kan klienten beregne transaksjonstiden i det den mottar svar fra systemet. Klienten mottar svar via metodekall fra tråden som styrer transaksjonen. Svaret vil være en av følgende typer: commit, abort eller reject. De viktigste metodene som benyttes i klienten er:

- `commitReply(UpdateTrans trans)` kalles når en transaksjon blir fullført i systemet. Klienten tar da vare på hvor lang tid denne transaksjonen har brukt i systemet og når hver enkelt transaksjon ble fullført i simuleringstiden.
- `abortReply(UpdateTrans trans)` kalles når en transaksjon aborterer. Her vil klienten ta vare på når transaksjonen ble abortert i simuleringstiden.
- `rejectReply(UpdateTrans trans)` kalles når en transaksjon blir avvist av systemet. Ved kall av denne metoden tar klienten vare på når transaksjonen ble avvist i simuleringstiden.

6.2.4 Støtteklasser

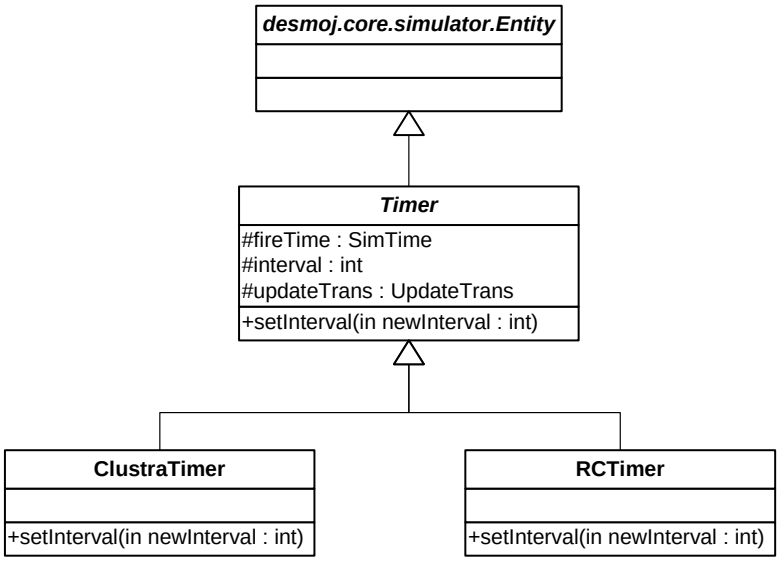
Støtteklasser er klasser som benyttes i tillegg til de aktive enhetene for å utføre transaksjoner og ressurskontroll. De støtteklassene som arver `desmoj.core.simulator.Entity` gjør det med tanke på at de skal kunne legges i Desmo-J sin ferdigdefinerte kø, `desmoj.core.simulator.Queue`. På denne måten får vi statistikk over køer i simuleringsrapportene. Statistikken fra disse køene blir også benyttet under kjøring av ressurskontrollen.

Vi vil videre i dette delkapitlet se på støtteklassene timere, meldinger, transaksjoner, køer, statistikk og klassen `Properties`.

Timere

ClustRa benytter timere til å utføre kjøring av hendelser på bestemte tidspunkt. For den delen av ClustRa som vi simulerer, vil dette innebære sending av ressurskontrollmeldinger og sending av transaksjonsmeldinger på nytt når svar ikke har kommet i tide. Kjøring av timere er relatert til transaksjoner, slik at det er bestemte hendelser knyttet til transaksjonsutførelser som trigger at timere skal kjøres.

1. *En nærmere forklaring av Poissonprosesser er å finne i [10].*



Figur 6.9: Klassediagram Timerhierarki

I figur 6.9 har vi et klassediagram over de klassene som representerer timere i simulatoren vår. Klassen `Timer` er en abstrakt klasse som arver `desmoj.core.simulator.Entity`. Denne klassen har tre sentrale attributter for timerkjøring:

- `fireTime` som er av typen `desmoj.core.simulator.SimTime` sier når dette timerobjektet skal kjøre neste gang. Tildeling av kjøretid styres av CPUen som tidligere forklart under kapittel 6.2.3.
- `interval` er en attributt av typen `int` som sier hvor stort det siste intervallet til timeren er. Dette intervallet er hvor lang tid det går fra forrige kjøring av timeren til neste kjøring av timeren. Ved transaksjonsprosessering vil tiden mellom hver gang timeren kjører dobles. Når meldingen som man venter på er mottatt vil dette intervallet stilles tilbake til sin initielle verdi. Dette er nærmere forklart i kapittel 6.4.
- `updateTrans` er av typen `UpdateTrans` og er en referanse til det transaksjonsobjektet som timerobjektet er relatert til.

Denne generelle timerklassen har også en viktig abstrakt metode: `setInterval(int newInterval)`. Denne metoden oppdaterer attributtene `interval` og `fireTime` ut i fra tallet som angis av `newInterval`. På denne måten styres tidspunktet timerobjektet skal aktiveres neste gang.

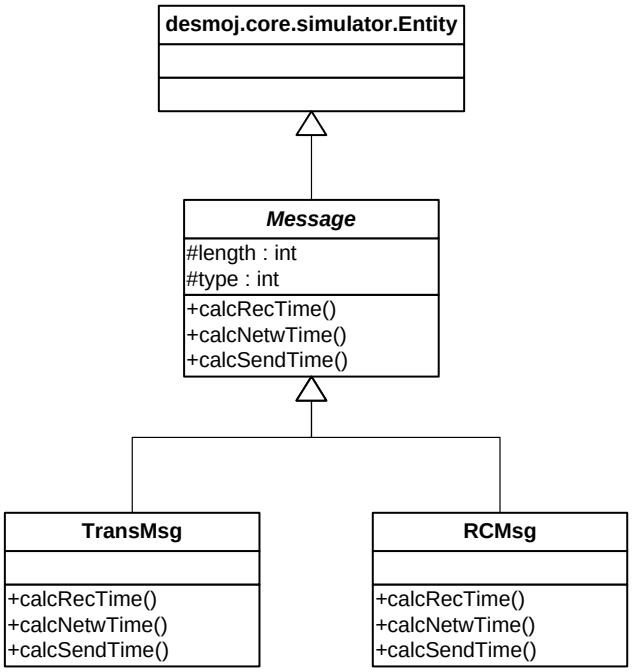
Klassen `Timer` har to subklasser: `ClustraTimer`[1] og `RCTimer`. Disse to klassene representerer timere for henholdsvis transaksjonsprosessering og ressurskontrollhåndtering. Det er viktig å skille mellom disse to typene av timere for å sikre at CPUen klarer å skille mellom de to typene timerkjøring og fordi de har forskjellig implementasjon av metoden `setInterval(int newInterval)`.

Meldinger

En melding er enheten som blir sendt mellom noder i systemet for at disse skal kunne kommunisere. Meldinger kommer som transaksjons- og ressurskontrollmeldinger. Innenfor hver meldingstype har hver melding et typeattributt. Dette er med på å skille de ulike meldingene som forekommer innenfor transaksjonsprosessering og ressurskontroll. CPUen benytter dette attributtet ved tolking av meldinger. I tillegg har en melding også referanser til avsender- og mottakertråd.

1. *Vi har valgt navnet* `ClustraTimer` *siden timere av denne klassen representerer de timere som brukes i ClustRa slik det er i dag.*

I figur 6.10 viser vi et klassediagram over hvordan meldingshierarkiet er oppbygd.



Figur 6.10: Klassediagram Meldingshierarki

Den abstrakte klassen `Message` er superklasse for alle meldinger i simulatoren. Den har to underklasser `TransMsg` og `RCMsg` som brukes til å implementere henholdsvis transaksjonsmeldinger og ressurskontrollmeldinger. I denne klassen ligger alle meldinger i systemet definert som statiske variabler. Disse er ikke vist i diagrammet over. Hvilken type et meldingsobjekt er settes av parameteren `type`.

Viktige metoder i meldingsklassen er de som beregner tiden det tar å sende og motta meldinger:

- `calcSendTime()` beregner hvor lang tid det tar å sende meldingen.
- `calcRecTime()` beregner hvor lang tid det tar å motta meldingen.
- `calcNetwTime()` beregner tiden meldingen bruker gjennom nettverket. Denne metoden returnerer 0 ms i vår simulator, og nettverkstiden er med i beregningene av sendtiden.

Lengden på en melding er av betydning for hvor lang tid det tar å sende og motta meldingen. Vi beregner lengden på en melding slik ligning 6.2 viser.

$$lengde = konstant + antall\ oppdateringer \times faktor \qquad \textit{Ligning 6.2}$$

De meldingene som er uavhengige av antall oppdateringer vil ha en konstant lengde. Faktoren settes følgelig til 0.

Hvor lang tid det tar å prosessere en melding for sending eller mottak, beregnes som vist i ligning 6.3.

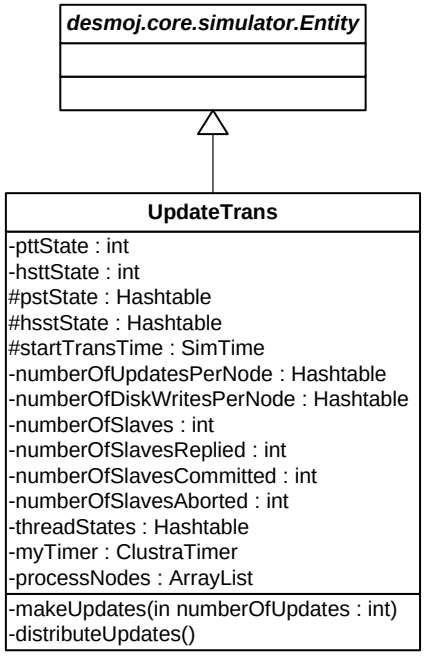
$$tid = konstant + lengde \times faktor \qquad \textit{Ligning 6.3}$$

Konstantene, lengdene og faktorene som er benyttet i ligningene over hentes fra instillingsfiler. Disse er beskrevet i vedlegg A i tabell A.2 og tabell A.3.

Transaksjoner

De transaksjonene som klientene i simuleringen starter er av typen oppdatering. Transaksjonene oppdaterer et antall poster i databasen, og antallet oppdateringer er gitt av parameteren `numberOfUpdates` som finnes i en av innstillingsfilene. Det er en gitt sannsynlighet for om hver av postene ligger ute på disken, og denne sannsynligheten er gitt av parameteren `probPageFault`. Hvis en datablokk må hentes fra disken, vil transaksjonen bli nødt til å vente med å fullføre til datablokken er hentet inn til minnet.

Oppdateringstransaksjoner implementeres i simulatoren av klassen `UpdateTrans` som arver `desmoj.core.simulator.Entity`. Vi vil nå se på de mest sentrale attributtene og metodene i denne klassen for å få en bedre forståelse av hvordan den er bygd opp. Disse attributtene og metodene er vist i figur 6.11.



Figur 6.11: Klassediagram UpdateTrans-klassen

Klassen inneholder følgende sentrale attributter:

- Tilstandsvariablene `pttState` og `hsttState` av typen `int` som holder tilstanden til objektene av klassen `PTrolThread` og `HSTrolThread` for denne transaksjonen.
- Hashtabellene `pstState` og `hsstState` som lagrer nåværende tilstand til hver enkelt av primær SlaveThread og hot standby SlaveThread som denne transaksjonen involverer. Disse hashtabellene har altså en kobling mellom trådobjekter og tilstander.
- `startTransTime` som er en attributt av typen `SimTime` og angir når denne transaksjonen startet sin utførelse. Dette brukes til statistikkføring og kontroll av timeouttiden.
- `numberOfUpdatesPerNode` og `numberOfDiskWritesPerNode` er to hashtabeller som har mapping mellom `Node` og henholdsvis antall oppdateringer totalt og antall oppdateringer mot disken som skal skje på hver enkelt av nodene.
- `numberOfSlaves` er antall slaver (både primære og hot standby) som denne transaksjonen skal utføres på.
- `numberOfSlavesReplied`, `numberOfSlavesCommitted` og `numberOfSlavesAborted` er variabler som teller opp antall tråder som `PTrolThread` har mottatt *SlaveReply*, *SlaveDone* og *SlaveAbort* fra for denne transaksjonen. Disse variablene oppdateres fra `PTrolThread` via set-metoder i denne klassen ved mottak av hver enkelt melding.
- `threadStates` er en `Hashtable` som har mapping mellom slavetråder (`PSlaveThread` og `HSlaveThread`) og tilstandene til objektene, lagret som `Integer`-objekter. Dette er `PTrolThread` sin opplevelse av tilstandene de har. Dette brukes ved mottak av *SlaveReply*, *SlaveAbort* og *SlaveDone* for å håndtere meldinger som er sendt mer enn én gang.
- `myTimer` er `ClustraTimer`-objektet som gjelder denne transaksjonen.
- `processNodes` er en `ArrayList` over alle noder denne transaksjonen involverer.

Klassen `UpdateTrans` har mange metoder. De fleste av disse metodene er get()- og set()-metoder på de omtalte attributtene. Disse metodene omtaler vi ikke i mer detalj her. Vi vil derimot se litt på to private metoder som kalles ved opprettelse av `UpdateTrans`-objektene:

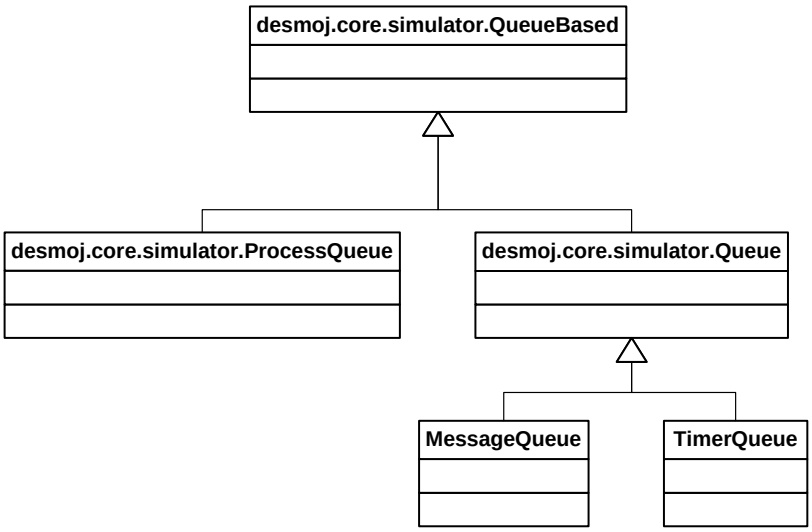
- `makeUpdates(int numberOfUpdates)` er en metode som finner hvilke noder denne transaksjonen skal kjøre på ut i fra antall oppdateringer som skal skje og oppdaterer variabelen `numberOfUpdatesPerNode`.
- `distributeUpdates()` fordeler oppdateringene på hver enkelt node mellom minneoppdateringer og oppdateringer mot disk. Denne fordelingen baserer seg på parameteren `probPageFault`. Dette medfører at variabelen `numberOfDiskWritesPerNode` får verdier.

Køer

Køer bruker vi i simulatoren for å håndtere prosesskøer, meldingssending og timerkjøring. CPUen håndterer køene etter FIFO-prinsippet, som vil innebære at nye hendelser legges bakerst i køen, mens hendelser tas ut først i køen. Det eneste unntaket er timerkøen, som sorteres på tid før objekter tas ut.

I figur 6.12 ser vi hvordan vårt kø-hierarki er implementert. Vi benytter klassene `desmoj.core.simulator.ProcessQueue`, `TimerQueue` og `MessageQueue` til å implementere de køene vi bruker. Disse brukes til henholdsvis å holde på tråder som skal ha kjøretid på CPUen, timere som skal kjøres på noden og ta vare på innkommende meldinger. Timerkøen er implementert slik at nye timere legges inn på rett plass i køen, avhengig av når de skal kjøres. De andre køene legger inn hendelser bakerst i køen.

Klassene `desmoj.core.simulator.ProcessQueue` og `desmoj.core.simulator.Queue` arver `desmoj.core.simulator.QueueBased`, mens `MessageQueue` og `TimerQueue` arver `desmoj.core.simulator.Queue`.



Figur 6.12: Klassediagram Køhierarki

Statistikk

Klassen `Statistics` utfører statistikkføringen i simulatoren. Den tar vare på hvor mye tid CPUen har brukt på kjøring av de forskjellige trådene, kjøring av timere, kjøring av ressurskontroll og tolking av meldinger. Den teller også opp antall fullførte, avbrutte og avviste transaksjoner i systemet. Det ligger to utgaver av alle objektene i statistikk-klassen. De to utgavene fører til at ressurskontrollen vår kan benytte den ene til periodiske målinger for så å nullstille verdiene, mens det andre objektet blir vist i sluttrapporten fra simuleringen med verdier samlet opp gjennom hele perioden.

Andre statistikkobjekter som blir vist i simuleringsrapporten, men som ikke ligger i statistikk-klassen, er alle køene i systemet. Etter at simuleringen er ferdig får vi da vite hvor mange som

totalt har vært i køen, gjennomsnittlig antall i køen og hvor mange det er i køen idet simuleringen avsluttes. Vi får også vite hvor mange enheter som ankom en tom kø.

Properties-klassen

Denne klassen inneholder get-metoder for alle parameterne som hentes fra innstillingsfilene. Alle metodene i klassen er statiske. Vi omtaler ikke disse metodene i mer detalj her og viser til vedlegg A for en nærmere forklaring av parameterne.

6.3 Ressurskontroll

I dette kapitlet vil vi beskrive hvordan vi har implementert ressurskontroll i vår simulator. Vi vil først se på intern ressurskontroll før vi ser på distribuert ressurskontroll.

6.3.1 Intern ressurskontroll

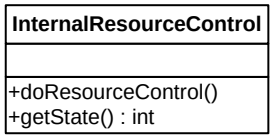
Intern ressurskontroll styrer tilstandsendring på noden. En generell beskrivelse av den interne ressurskontrollen er å finne i kapittel 4 og i rapporten fra i høst [1]. Her presenterer vi tre tilstander en node i systemet kan ha. Vi velger å implementere to av disse tilstandene, med implementasjonsnavnene i parentes:

- Opptatt (`Busy`)
- Ledige ressurser (`Free`)

Dette innebærer at tilstanden *tungt belastet* er utelatt fra simulatoren. Grunnen til dette er at vi ikke har implementert forskjellige transaksjoner hvor de mest prioriterte eller de av en bestemt type ville blitt sluppet inn i systemet ved tilstanden *tungt belastet*, mens de øvrige ville blitt avvist. Derfor vil nodene i vår simulator enten være i tilstanden *opptatt*, hvor den ikke ønsker nye transaksjoner, eller i tilstanden *ledige ressurser*, hvor alle transaksjoner kan tas i mot.

Klassen `InternalResourceControl` er vist i figur 6.13 med de metoder som har synlighetsnivå public. Dette viser klassens to hovedfunksjoner:

- Utføre ressurskontroll internt på noden via metoden `doResourceControl()`
- Gi nodens tilstand til andre noder under distribuert ressurskontroll via metoden `getState()`



Figur 6.13: Klassediagram InternalResourceControl-klassen

Vi vil nå se nærmere på disse to hovedfunksjonene til klassen `InternalResourceControl.`

Utføre ressurskontroll

Alle tallverdiene det vises til i dette avsnittet, er omtalt i kapittel 7. Der begrunnes de tallverdiene som vi presenterer her.

Metoden `doResourceControl()` kalles fra `lifeCycle()`-metoden i `CPU`. Hvor ofte dette kallet skal skje styres av parameteren `interval_rc`, som angir intervallet mellom to metodekall. Standardverdi på denne parameteren er satt til 5 ms.

Hvilken type intern ressurskontroll som skal aktiveres ved kall av metoden `doResourceControl()` styres av parameteren `internalResourceControlType`. Vi har implementert tre typer intern ressurskontroll, som setter tilstand basert på følgende størrelser, med metodenavnene i klassen `InternalResourceControl` i parentes:

- Prosessorlasten (`CPULoad()`)
- Diskkøen (`DiskLoad()`)
- Kombinasjon av prosessorlasten og diskkøen (`DiskCPULoad()`)

Prosessorlasten setter tilstand ut i fra hvor aktiv prosessoren er på hver enkelt node. Aktiviteten til prosessoren styres av klassen `CPU` og vi regner den som aktiv når den gjør annet enn å være

ledig. Grensen for hvilken verdi av prosessorlasten som gir tilstanden *opptatt* er styrt av parameteren `CPUBusyBoundary = 0.8`, hvor verdien angir andelen av tiden CPUen jobber.

Diskkøen måler antall transaksjoner som står i kø for å utføre diskarbeid på hver enkelt node. Overstiger disse verdiene den angitte grensen, vil tilstanden til noden endres til tilstanden *opptatt*. Dette styres av parameteren `diskBusyBoundary`, hvor standardverdien er 10 diskjobber.

Kombinasjon av prosessorlasten og diskkøen er en løsning hvor man vurderer både prosessorlasten og diskkøen. Tilstand settes her ut i fra en kombinasjon av tilstanden som hver enkelt av de to løsningene presentert over gir. Vi velger den tilstanden av de to metodene som gir høyest belastning. Dette medfører at om en metode gir tilstanden *ledige ressurser* og en annen gir *opptatt*, vil tilstanden settes til *opptatt*.

Intern ressurskontroll utføres enten historiebasert eller historieløst. Dette styres av parameterne `numberOfCPUValues` og `numberOfDiskValues` for intern ressurskontroll basert på henholdsvis prosessorlast og diskkøen. Settes verdiene på disse parameterne til 1 impliserer det historieløs oppdatering, siden man kun ser på den siste verdien. Ved verdier større enn 1 får vi historiebasert oppdatering. Dette har vi implementert ved å se på dette antall av de siste målingene og finne snittet av dem. Snittet vil da sammenlignes med parameterne for de ulike tilstandene og danne grunnlaget for endringen av tilstand. Et større tall vil her gi mer historiebasert oppdatering og dermed vil nodens tilstand endres senere. Vi har valgt å bruke følgende standardparametere for vår kjøring:

- `numberOfCPUValues = 3`
- `numberOfDiskValues = 1`

Returnere nodens tilstand

Denne delen av funksjonen til klassen `InternalResourceControl` er kall av metoden `getState()` fra andre klasser. Disse metodekallene brukes når noden skal sjekke sin egen tilstand mot innkomne transaksjoner eller når noden skal spre sin lastinformasjon til andre noder i systemet ved hjelp av distribuert ressurskontroll.

6.3.2 Distribuert ressurskontroll

Distribuert ressurskontroll avgjør om nye transaksjoner skal få starte sin utførelse basert på tilstandene til de involverte nodene. En generell beskrivelse av distribuert ressurskontroll er å finne i kapittel 5 og i rapporten fra i høst [1]. Vi gjentar derfor ikke dette her.

Hvilken type distribuerte ressurskontroll som benyttes styres av parameteren `distributedResourceControlType`. Denne velger en av de tre mulige typene distribuert ressurskontroll eller skrur av ressurskontrollen.

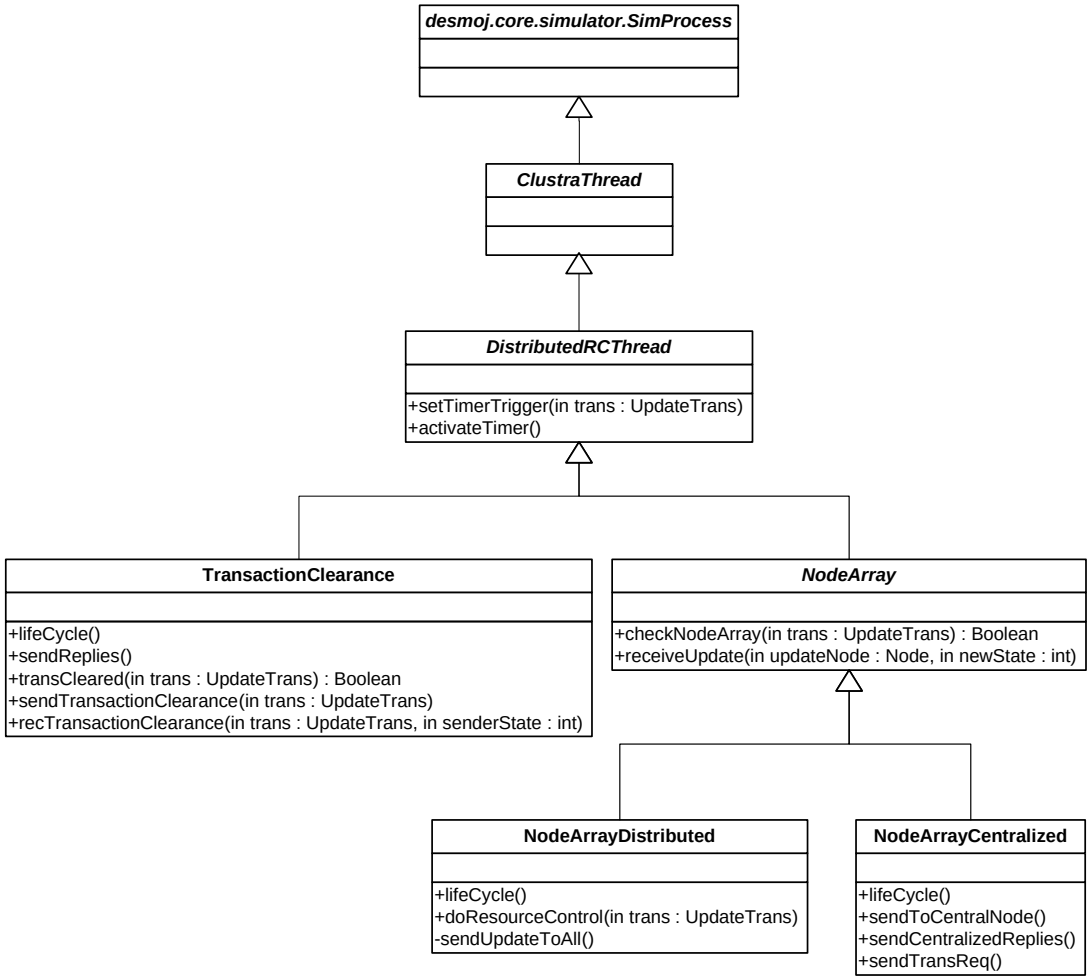
Klassehierarkiet for distribuert ressurskontroll er vist i figur 6.14. Vi ser at alle klassene er subklasser av `ClustraThread`, som igjen arver `desmoj.core.simulator.SimProcess`. Dette gjør at alle klassene arver metoden `lifeCycle()` og dermed kan opptre som tråder som har kjøretid i systemet.

Meldingene som sendes mellom nodene ved distribuert ressurskontroll er av typen `RC_msg`. Ved transaksjonsklarering har vi egne meldinger `Transaction_clearance_req` og `Transaction_clearance_rep` for henholdsvis forespørsel og svar på klareringen.

Distribuert ressurskontroll benytter timere for å styre sendingen av meldinger mellom nodene i systemet. En generell beskrivelse av timere er gitt i kapittel 6.2.4. Timerkjøringobjektene som brukes av ressurskontrollobjektene er av klassen `RCTimer`. Vi bruker disse timerene til to typer oppgaver:

- Sende oppdateringer til alle de andre nodene ved distribuert nodetabell, eller kun til den sentrale noden ved bruk av sentralisert nodetabell som ressurskontroll. Hvor ofte disse meldingene sendes styres av parameteren `interval_rc`. Denne parameteren har standardverdi på 5 ms. Dermed kalles metoden `setInternval(int newInterval)` med denne parameterens verdi som parameter ved hver kjøring av en timer.
- Ved metoden transaksjonsklarering bruker vi timere til å bryte ventingen på svar fra nodene som har fått spørsmål om å delta i en ny transaksjon. Dette styres av parameteren `timeoutTransactionClearance` som styrer intervallet som settes til timeren i metoden

`setInterval(int newInterval)`. Når tiden til denne parameteren har passert uten at alle har svart antas det at de som ikke har svart er for tungt belastet og transaksjonen får ikke starte og blir avvist. Denne parameteren har standardverdi på 20 ms.



Figur 6.14: Klassediagram distribuert ressurskontroll-hierarki

Vi ser nå nærmere på hver enkelt av klassene for distribuert ressurskontroll og omtaler hva hver enkelt har som funksjon.

DistributedRCThread

Dette er en abstrakt klasse som arver `ClustraThread`. Denne klassen har to viktige metoder som er felles for ressurskontrollklassene:

- `setTimer(UpdateTrans trans)` som setter den transaksjonen som aktiverer timerkjøringen på det aktuelle ressurskontrollobjektet. Dermed vil ressurskontrollklassen vite hvilken transaksjon timerhåndtering skal utføres mot.
- `activateTimer()` som aktiverer timerkjøring. Dette vil innebære at timere vil kjøre neste gang den aktuelle ressurskontrollklassen aktiveres. Timerkjøringen slås av etter at timere har blitt kjørt.

TransactionClearance

Denne klassen arver `DistributedRCThread` og brukes når transaksjonsklarering er valgt som distribuert ressurskontroll.

Transaksjonsklarering er basert på at hver enkelt transaksjon klareres eksplisitt ved at hver enkelt av de involverte nodene spørres om de kan motta en ny transaksjon. Basert på disse svarene vil en transaksjon enten avvises eller startes.

Denne klassen inneholder følgende sentrale metoder:

- `lifeCycle()` som styrer hva som skal skje hver gang objekter av denne klassen får kjøretid av CPUen. Dette vil medføre sending, mottak og prosessering av klareringsforespørsler.
- `sendReplies()` er en metode som sender svar til alle transaksjoner som venter på svar fra denne noden. Vi svarer her alle noder med transaksjoner som har spurt denne noden om den kan ta i mot en ny transaksjon. Grunnen til at det kan bli flere transaksjoner å svare samtidig er at noden kan være opptatt når de enkelte forespørsler ankommer og at det dermed kan ha kommet flere transaksjoner som venter på svar. Disse svarene gir meldinger av typen `Transaction_clearance_rep`.
- `transCleared(UpdateTrans trans)` kalles fra CPUen for å sjekke om den angitte transaksjonen er klarert eller ikke.
- `sendTransactionClearance(UpdateTrans trans)` sender forespørsel til alle noder som skal delta i den angitte transaksjonen. Disse forespørslene sendes med meldinger av typen `Transaction_clearance_req`.
- `recTransactionClearance(UpdateTrans trans, int senderState)` er en metode som kalles fra CPUen hver gang den noden som skal klarere transaksjonen mottar meldinger av typen `Transaction_Clearance_rep`. Denne metoden ser på tilstanden som sendes og avgjør ut i fra tilstanden om transaksjonen kan startes eller ikke, når svar er mottatt fra alle nodene, eller når den første noden er for opptatt til å gjennomføre transaksjonen, noe som vil medføre at transaksjonen avvises.

NodeArray

Klassen `NodeArray` er en abstrakt klasse som arver `DistributedRCThread`. Denne klassen inneholder metoder som er felles for de to underklassene `NodeArrayCentralized` og `NodeArrayDistributed`. De mest sentrale metodene som denne klassen har er:

- `checkNodeArray(UpdateTrans trans)` som returnerer en boolsk verdi. Denne verdien angir om transaksjonen som er parameter til metoden kan utføres eller ikke. Dette avgjøres ut i fra nodetabellen som denne noden har. En slik sjekk utføres av alle noder ved distribuert nodetabell, mens det kun er den sentrale noden som gjør dette ved sentralisert nodetabell.
- `receiveUpdate(Node node, int newState)` kalles fra CPUen når den mottar meldinger fra andre noder om deres tilstand i øyeblikket. Denne metoden oppdaterer så nodens nodetabell med den nye tilstanden.

NodeArrayCentralized

`NodeArrayCentralized` er en klasse som benyttes til ressurskontroll når sentralisert nodetabell er valgt som distribuert ressurskontrolltype. Denne klassen arver klassen `NodeArray`. De mest sentrale metodene i denne klassen er:

- `lifeCycle()`som styrer hva som skal skje hver gang objekter av denne klassen får kjøretid av CPUen. Dette medfører å sende tilstand og nye forespørsler for noder som ikke er sentrale, mens den sentrale noden vil behandle forespørsler om transaksjoner.
- `sendToCentralNode()` sender nodens tilstand til den sentrale noden.
- `sendCentralizedReplies()` er en metode som brukes på den sentrale noden til å sende svar til de noder som har nye transaksjoner som skal klareres. Denne metoden sjekker nodetabellen for de involverte nodene for å avgjøre om transaksjonen kan starte eller ikke.
- `sendTransReq()` sender forespørsler om klarering av alle nye transaksjoner til den sentrale noden.

NodeArrayDistributed

Denne klassen arver `NodeArray` og implementerer ressurskontrolltypen distribuert nodetabell. De mest sentrale metodene i denne klassen er:

- `lifeCycle()` som styrer hva som skal skje hver gang objekter av denne klassen får kjøretid av CPUen. Det vil her si å sende oppdateringer om nodens tilstand til de andre nodene i systemet.
- `doResourceControl(UpdateTrans trans)` utfører sjekken av transaksjonen som ønsker å starte ved å kalle metoden `checkNodeArray(trans)`.

- `sendUpdateToAll()` sender nodens tilstand til alle andre noder i systemet. På denne måten får alle nodene oppdatert lastinformasjon om de andre nodene i systemet.

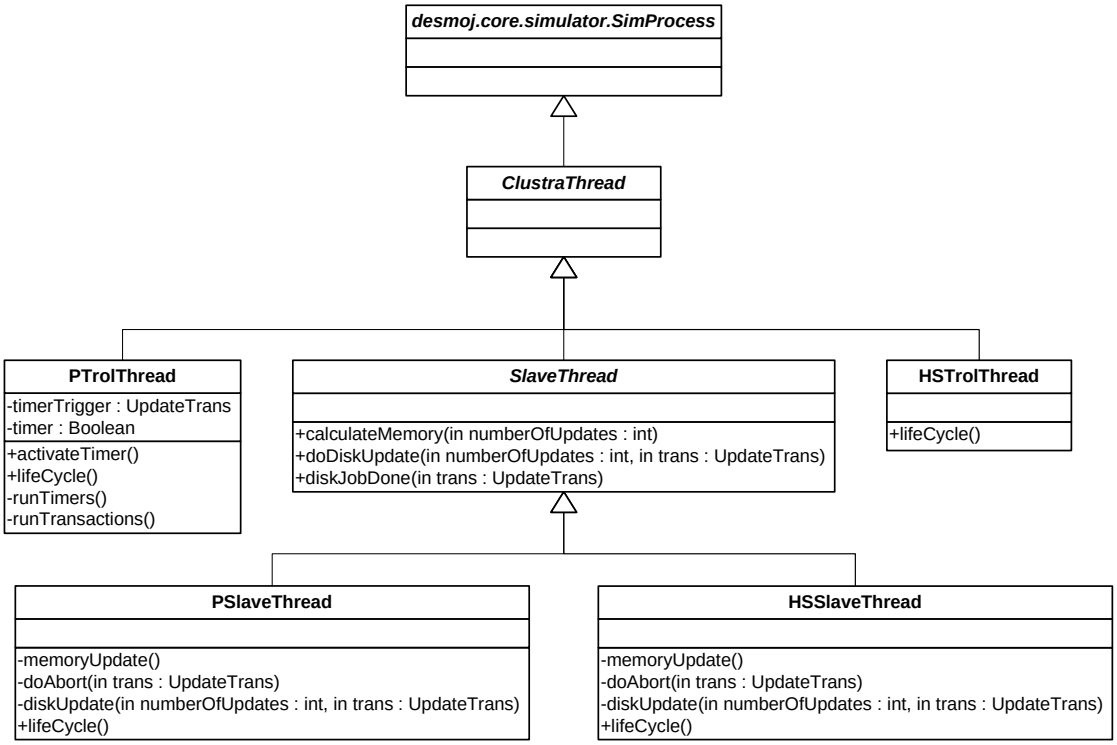
6.4 Prosessering av oppdateringstransaksjoner

Utførelsen av oppdateringstransaksjoner utføres i ClustRa slik vi har forklart i kapittel 3.3. Vi viser til denne forklaringen for generell forståelse av utførelsen og meldingene som sendes. En verifisering av hvordan simulatoren vår utfører transaksjoner er vist i vedlegg D.

Oppdateringstransaksjoner slik vi har implementert det, utføres ved hjelp av fire tråder og en timer. De trådene som benyttes er:

- `PTrolThread` som styrer transaksjonsutførelsen
- `HSTrolThread` som er hot standby for `PTrolThread`
- `PSlaveThread` som oppdaterer tuplene i database
- `HSSlaveThread` som utfører loggposter fra `PSlaveThread` på hot standby-delen av databasen

Disse fire trådene er vist i klassediagrammet i figur 6.15 sammen med klasser de arver og de mest sentrale attributtene og metodene. En generell beskrivelse av dette trådhierarkiet er å finne i kapittel 6.2.3.



Figur 6.15: Klassediagram Transaksjonstråd-hierarki

Vi vil videre i dette underkapitlet først se nærmere på hvordan timerkjøringen ved transaksjonsprosessering er. Underkapitlet vil til slutt se på implementasjon og tilstandsdiagrammer for hver av de fire trådene som styrer transaksjonsutførelsen. Tilstandsdiagrammene forklares ut i fra at det finnes egne trådobjekter for hver transaksjon som utføres. Dette er ikke slik vi har implementert det, da vi har et sett av tråder som er felles for alle transaksjonene på en node. Vi forklarer det likevel slik for å få en enklere forklaring som da stemmer om man ser på én transaksjon isolert. Før forklaringen av `PSlaveThread` og `HSSlaveThread` vil vi også se på den abstrakte klassen `SlaveThread` som de to førstnevnte klassene arver.

6.4.1 Transaksjonstimere

Timeren som brukes ved transaksjonsprosessering er av typen `ClustraTimer`, og skal sikre at meldinger blir sendt på nytt hvis de skulle bli borte og den skal sikre framdrift i transaksjonsprosesseringen. Timere generelt er forklart i kapittel 6.2.4. Vi viser til denne beskrivelsen for forklaring av implementasjonen av denne klassen.

En transaksjonstimer kjøres første gang etter 10 ms. Denne verdien er bestemt av parameteren `initialTimerInterval` fra en av innstillingsfilene. Timeren vil ved kjøring doble dette intervallet, og vil neste gang kjøres etter 20 ms. Slik fortsetter timeren å kjøre ved å doble intervallet mellom hver kjøring til intervallet overstiger parameteren `maxTimerValue`. Denne parameteren har standardverdi på 2000 ms. Når de meldinger som tråden venter på har kommet fram, vil transaksjonsprosessering gå videre og tiden fram til neste gang timeren skal kjøres settes til den intitielle verdien på 10 ms.

I de tilfeller en transaksjon aborteres, vil timeren sikre at abortmeldingene blir sendt på nytt. Tiden for den første kjøringen er bestemt av `startAbortTimer`, og initielt er denne verdien satt til 50 ms. Intervallet dobles for hver gang på samme måte som for vanlig timerkjøring.

6.4.2 PTrolThread

Hver node har ett objekt av klassen `PTrolThread`. Dette gjør `PTrolThread` til en fellesklasse som styrer transaksjonsutførelsen av alle transaksjoner som startes på en node. `PTrolThread` har som oppgave å kontakte de involverte nodene for nye transaksjoner, sende og motta meldinger og å sikre at transaksjonen enten fullfører eller aborterer på korrekt måte.

Vi vil først presentere en beskrivelse av hvordan `PTrolThread` er implementert, før vi ser på hvilke tilstander denne klassen har og hvordan den kommer fra tilstand til tilstand via meldingssending.

Implementasjon

Klassediagrammet vist i figur 6.15 viser de mest sentrale attributter og metoder for å forstå hvordan `PTrolThread` er oppbygd. `PTrolThread` arver `ClustraThread`. Attributtene som vises i figuren er:

- `timerTrigger`, som er en referanse til det `UpdateTrans`-objektet som gjeldende timerkjøring er relatert til.
- `timer` som er en boolsk-variabel som avgjør om timere skal kjøres eller ikke. Denne variabelen settes av metoden `activateTimer()`.

Klassen har i tillegg en rekke attributter som definerer de tilstandene `PTrolThread` kan ha for transaksjonene. Dette er utelatt i klassediagram da det er forklart under tilstander senere i delkapitlet.

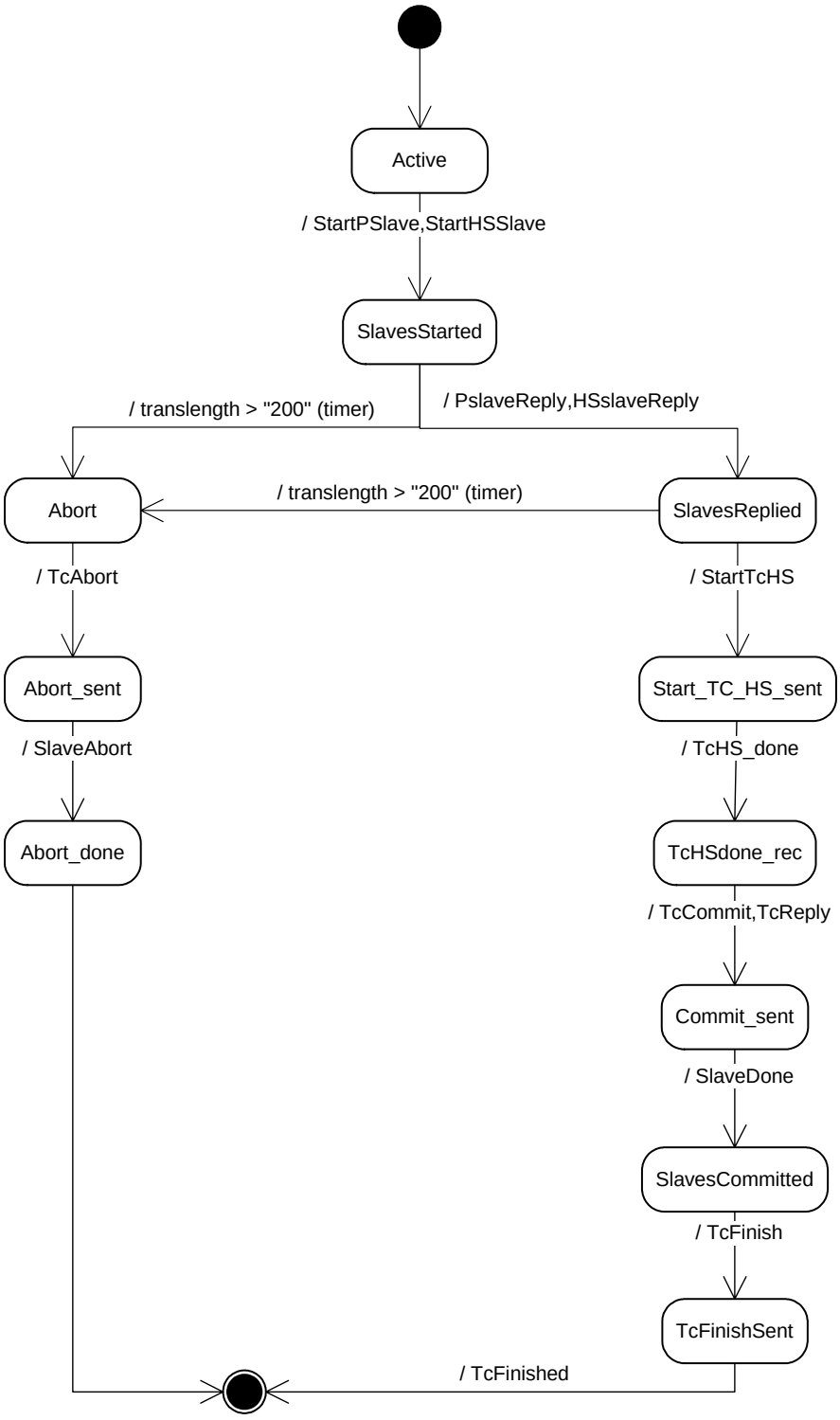
De mest sentrale metodene i `PTrolThread` er:

- `activateTimer()` som setter `timer = true`, det vil si at timerkjøring aktiveres.
- `lifeCycle()` som styrer `PTrolThread` sin oppførsel. Denne metoden vil kjøre timere ved kall av metoden `runTimers()` om `timer` er satt til true, ellers utføres normal transaksjonsprosessering og meldingssending ved metoden `runTransactions()`.
- `runTimers()` kjører timere. Det vil si at den sender meldinger på nytt. Denne sendingen utføres ved å kalle egne resendmetoder for de ulike meldingene. Hvilken melding som skal sendes på nytt styres av tilstanden til `PTrolThread` for den aktuelle transaksjonen. Disse metodene omtaler vi ikke i mer detalj her.
- `runTransactions()` som utfører transaksjonsprosessering. Dette vil innebære sending av meldinger. Hvilke meldinger som skal sendes styres av tilstanden til `PTrolThread` for den aktuelle transaksjonen. Meldingssendingen utføres av egne metoder for hver melding. Disse metodene forklares ikke i mer detalj her.

`PTrolThread` har i tillegg flere metoder for å motta meldinger fra de andre trådene. Disse metodene kalles av CPUen. Vi går ikke mer inn på disse metodene her.

Tilstander

Tilstandsdiagrammet for `PTrolThread` er vist i figur 6.16. Under følger en beskrivelse av dette tilstandsdiagrammet.



Figur 6.16: Tilstandsdiagram for PTrolThread

En nyopprettet `PTrolThread` vil initielt være i tilstanden *Trans_started*. Første gangen den får kjøretid fra CPUen, vil den sende meldingen *startPSlave* til alle primære slaver og meldingen *StartHSSlave* til alle hot standby-slaver. `PTrolThread` endrer så tilstand til *Slaves_started*.

Tråden setter også i gang en timer, som skal kjøres første gang etter 10 ms. Meldingene *StartPSlave* og *StartHSSlave* vil bli sendt på nytt hver gang timeren kjører mens `PTrolThread` er i tilstanden *Slaves_started*. Dersom den totale tiden på transaksjonen overstiger 200 ms i denne første fasen, vil timeren føre til at `PTrolThread` går til tilstanden *Abort* og transaksjonen vil abortere. Abort er beskrevet i det siste avsnittet av dette delkapitlet.

I tilstanden *Slaves_started* venter `PTrolThread` på svar fra alle slavene. Timerfunksjonen sikrer at meldingene *StartPSlave* og *StartHSSlave* blir sendt på nytt til de slavene som ikke har sendt *PSlaveReply* og *HSSlaveReply* til `PTrolThread` innen timeren kjøres. I det øyeblikket `PTrolThread` har mottatt svar fra alle slavene, vil den gå til tilstanden *Slaves_replied* og deretter sende *StartTcHs* som starter `PTrolThread` sin hot standby. Denne meldingen sendes til speilnoden. `PTrolThread` går over i tilstanden *StartTcHs_sent* og timeren vil da sende *StartTcHs* på nytt inntil `HSTrolThread` har svart.

Ved mottak av *TcHsDone* vil `PTrolThread` gå til tilstanden *TcHsDone_rec*, som angir at denne meldingen er mottatt. Deretter vil `PTrolThread` sende *TcReply* til klienten som indikasjon på at transaksjonen gikk i orden, og `PTrolThread` vil sende *TcCommit* til alle slavene som var delaktige i transaksjonsutførelsen. Tilstanden endres deretter til *Commit_sent*. Meldingen *TcCommit* vil bli sendt på nytt ved hjelp av timerfunksjonen dersom svaret *SlaveDone* ikke har kommet til PTrolThread.

Etter at `PTrolThread` har mottatt *SlaveDone* fra alle slavene den venter svar fra, vil tilstanden endres til *Slaves_committed*. Det som gjenstår nå er å underrette `HSTrolThread` om at slavene har committed. Dette gjøres ved å sende meldingen *TcFinish* til `HSTrolThread`. Denne meldingen resendes også ved hjelp av timere, hvis `PTrolThread` ikke har mottatt meldingen *TcHsFinished*.

Den siste meldingen `PTrolThread` vil motta er *TcHsFinished* fra `HSTrolThread`. Hver node har en tabell med alle aktive transaksjoner. Når en transaksjon avsluttes, vil derfor `PTrolThread` kalle metoder på noden som sletter innslaget av transaksjonene i tabellene over aktive transaksjoner, og `PTrolThread` vil også slette tilhørende transaksjonstimer fra timerkøen på noden.

Hvis transaksjonstiden sett fra `PTrolThread` fram til og med tilstanden *Slaves_replied* overstiger en fastsatt grense, vil transaksjonen abortere. Denne grensen er satt til 200 ms. Abortprosseseringen skjer ved at `PTrolThread` sender meldingen *TcAbort* til alle slavene som er delaktige i transaksjonen. Meldingen blir sendt på nytt gjennom timerkjøring inntil alle slavene har svart med *SlaveAbort*. I det `PTrolThread` har mottatt *SlaveAbort* fra alle slavene, vil transaksjonen være avsluttet. Det er ikke behov for å sende *StartTcHS* ved abort, siden transaksjonen ikke skal fullføres og `PTrolThread` dermed ikke trenger noen hot standby til å kunne ta over prosesseringen.

6.4.3 HSTrolThread

`PTrolThread` i simulatoren vår har en hot standby-tråd på speilnoden, som skal kunne overta transaksjonsprosesseringen dersom den primære noden skulle bli utilgjengelig. I vår simulator ser vi bort fra at noder kan gå ned, men for å kunne simulere meldingskostnadene ved å ha en `HSTrolThread` i en transaksjon har vi med denne tråden.

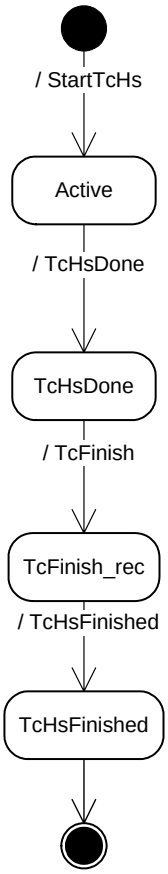
Vi vil først presentere en beskrivelse av hvordan `HSTrolThread` er implementert, før vi ser på hvilke tilstander denne klassen har og hvordan den kommer fra tilstand til tilstand via meldingssending.

Implementasjon

`HSTrolThread` er vist sammen med de andre trådklassene i figur 6.15. `HSTrolThread` arver `ClustraThread`. I denne figuren ser vi kun en metode for `HSTrolThread`: `lifeCycle()`. Metoden `lifeCycle()` vil for `HSTrolThread` styre sending av meldinger til `PTrolThread` etter hvilken tilstand `HSTrolThread` har. Disse tilstandsendringene og meldingssendingene forklares i neste avsnitt. Meldingssendingen utføres av egne metoder for hver melding. Klassen har også egne metoder for mottak av meldingene som sendes til `HSTrolThread`. Disse metodene forklares ikke i mer detalj her.

Tilstander

Tilstandsdiagrammet for `HSTrolThread` er vist i figur 6.17. Under følger en forklaring av disse tilstandsendringene.



Figur 6.17: Tilstandsdiagram for HSTrolThread

`HSTrolThread` startes ved at noden mottar meldingen *StartTcHs* fra `PTrolThread`. I det øyeblikk `HSTrolThread` får kjøretid av CPUen, vil den sende svaret *TcHsDone* til `PTrolThread` som bevis på at den nå er klar over transaksjonen som foregår og at den kan ta over som `TrolThread` om `PTrolThread` skulle gå ned. `HSTrolThread` går deretter til tilstanden *TcHsDone*. `HSTrolThread` har ikke noen mekanisme for å sende meldinger på nytt. Det er `PTrolThread` sin oppgave å sikre at meldinger kommer fram.

`HSTrolThread` vil til slutt motta meldingen *TcFinish* fra `PTrolThread`, og `HSTrolThread` vil da svare med å sende *TcHsFinished* tilbake. I dette øyeblikket har `HSTrolThread` også committed transaksjonen, og den kan nå glemmes. Av tilstandsdiagrammt ser vi at `HSTrolThread` går til tilstanden *TcHsFinished* etter å ha sendt meldingen *TcHsFinished*. Dette gjøres for at `HSTrolThread` ikke skal gjøre noe galt hvis den feilaktig skal få kjøretid etter at transaksjonen har fullført.

6.4.4 SlaveThread

`SlaveThread` er en abstrakt klasse som arver `ClustraThread`. Dette gjør at klassen ikke brukes direkte som aktiv tråd. Klassen har derimot flere sentrale metoder som brukes av underklassene `PSlaveThread` og `HSSlaveThread`. I figur 6.15 er klassen vist med de mest sentrale metodene:

- `calculateMemory(int numberOfUpdates)` som finner hvor lang tid utførelsen av det antallet oppdateringer som er gitt som parameter vil ta mot minnet. Metoden returnerer denne tiden i antall millisekunder. Tiden en oppdatering tar styres av parameterne `mtWriteRecordBuffer` og `mtWriteFourRecordsBuffer` for å skrive henholdsvis én og

fire tupler til minnet. Metoden finner tiden ved å lage en lineær utvikling av tiden ut i fra disse to parameterne og la tiden fordele seg etter en normalfordeling. Parameteren `standardDeviationMemUpdate` setter standardavviket til normalfordelingen.

- `doDiskUpdate(int numberOfUpdates, UpdateTrans trans)` kalles når oppdateringer skal gjøres mot disken. Denne metoden vil finne en av nodens disker for hver oppdatering og legge jobben i diskens kø. På denne måten kan transaksjonen oppleve at jobbene spres på ulike disker om hver node har to eller flere disker. Hvor lang tid jobben tar styres av disken og hvor lang kø den har når jobben ankommer. Ved kall av denne metoden gir tråden kontrollen til CPUen, da vi implementerer at venting på disk gir avbrudd og dermed trådbytte.
- `diskJobDone(UpdateTrans trans)` kalles når en diskjobb er ferdig. Metoden sjekker om alle diskjobbene til denne transaksjonen er ferdig. Er det tilfellet, vil transaksjonens diskarbeid settes som ferdig og transaksjonen kan fortsette sin utførelse.

Denne klassen har i tillegg til disse metodene flere variabler som arves av `PSlaveThread` og `HSSlaveThread`. Dette gjelder blant annet flere felles tilstander. Dette velger vi ikke å gå nærmere inn på her.

6.4.5 PSlaveThread

Tråden `PSlaveThread` er ansvarlig for å utføre de primære oppdateringene mot databasen og å sende loggen den produserer til `HSSlaveThread` som utfører disse loggpostene.

Vi vil først presentere en beskrivelse av hvordan `PSlaveThread` er implementert, før vi ser på hvilke tilstander denne klassen har og hvordan den kommer fra tilstand til tilstand via meldingssending.

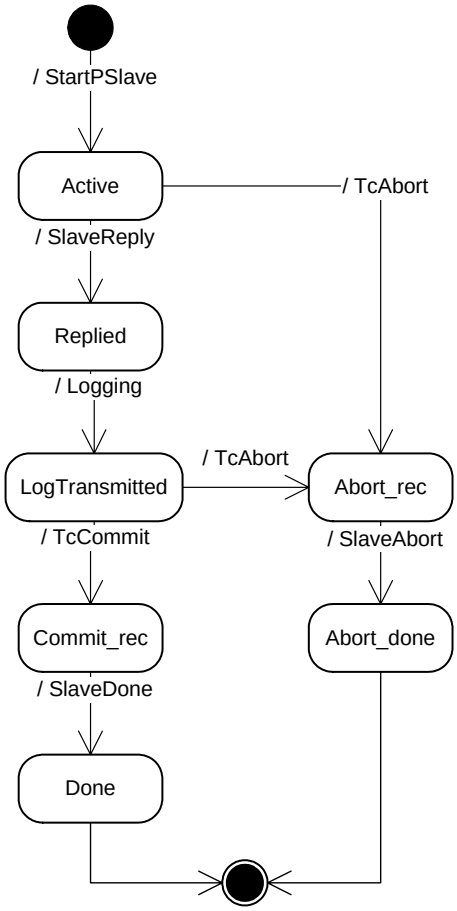
Implementasjon

De mest sentrale metodene fra denne klassen som arver `SlaveThread` er vist i figur 6.15. Disse metodene er:

- `memoryUpdate(int numberOfUpdates)` som finner tiden oppdateringene tar ved å kalle `calculateMemory(int numberOfUpdates)` og sove den angitte tiden ved kall av `hold(SimTime time)`.
- `doAbort(UpdateTrans trans)` utfører prosesseringen ved abort av transaksjonen. Dette vil i vår simulator innebære det samme arbeidet som ved oppdatering mot minnet.
- `diskUpdate(int numberOfUpdates, UpdateTrans trans)` kaller metoden `doDiskUpdate(int numberOfUpdates, UpdateTrans trans)` og legger diskjobbene i kø på disken.
- `lifeCycle()` styrer utførelsen av meldingssending og oppdateringer mot disk og minne for tråden. Dette styres av tilstandsendringer og meldingssendingen utføres ved egne metoder for hver melding. Vi har også implementert egne metoder for meldingene tråden mottar, disse meldingene forklares ikke i mer detalj her. Det samme gjelder de ulike tilstandsattributtene.

Tilstander

Tilstandene denne tråden gjennomgår, er vist i figur 6.18. Etter figuren følger en forklaring av tilstandsendringene.



Figur 6.18: Tilstandsdiagram for PSlaveThread

Noden som sitter på primærreplikatet av dataposten som skal oppdateres, vil motta en melding *StartPSlave* og aktiverer `PSlaveThread`. Tråden vil ved kjøring utføre oppdateringen, og sende *SlaveReply* som kvittering på at oppdateringen er utført. Hvis flere poster ligger på den samme noden, vil `PSlaveThread` oppdatere alle postene før den sender *SlaveReply*. `PSlaveThread` sender i tillegg loggen den produserer til sin `HSSlaveThread` i meldingen *Logging*. Etter at oppdateringene er utført og loggen er sendt, vil `PSlaveThread` gå til tilstanden *LogTransmitted*.

Dersom transaksjonen utføres korrekt, vil nå `PSlaveThread` motta en melding *TcCommit* fra `PTrolThread`. `PSlaveThread` svarer ved å sende *SlaveDone* tilbake, og setter seg selv i tilstanden *Done*. `PSlaveThread` er nå ferdig med utførelse av denne transaksjonen.

Hvis en av deltakerne i transaksjonen ikke kan fullføre innen timeout-tiden, vil `PSlaveThread` motta meldingen *TcAbort*. Dette kan skje mens `PSlaveThread` er i en av tilstandene *Active* eller *LogTransmitted*. `PSlaveThread` vil på en melding av typen *TcAbort* svare med meldingen *SlaveAbort* til `PTrolThread`, og deretter gå til tilstanden *Abort_done*. `PSlaveThread` er nå ferdig med den transaksjonen som aborterer.

6.4.6 HSSlaveThread

Oppgaven til `HSSlaveThread` er å utføre de loggpostene som den mottar fra sin `PSlaveThread` for å sikre to konsistente kopier av databasen. `HSSlaveThread` har mulighet til å svare `PTrolThread` med *SlaveReply* før eller etter at loggpostene er utført. Dette valget spesifiseres av parameteren `sendReplyBeforeLog` som har standardverdi 1, som vil si at svar sendes før loggen utføres. Ved verdi på 0 vil loggen utføres før svaret sendes.

Vi vil først presentere en beskrivelse av hvordan `HSlaveThread` er implementert, før vi ser på hvilke tilstander denne klassen har og hvordan den kommer fra tilstand til tilstand via meldingssending.

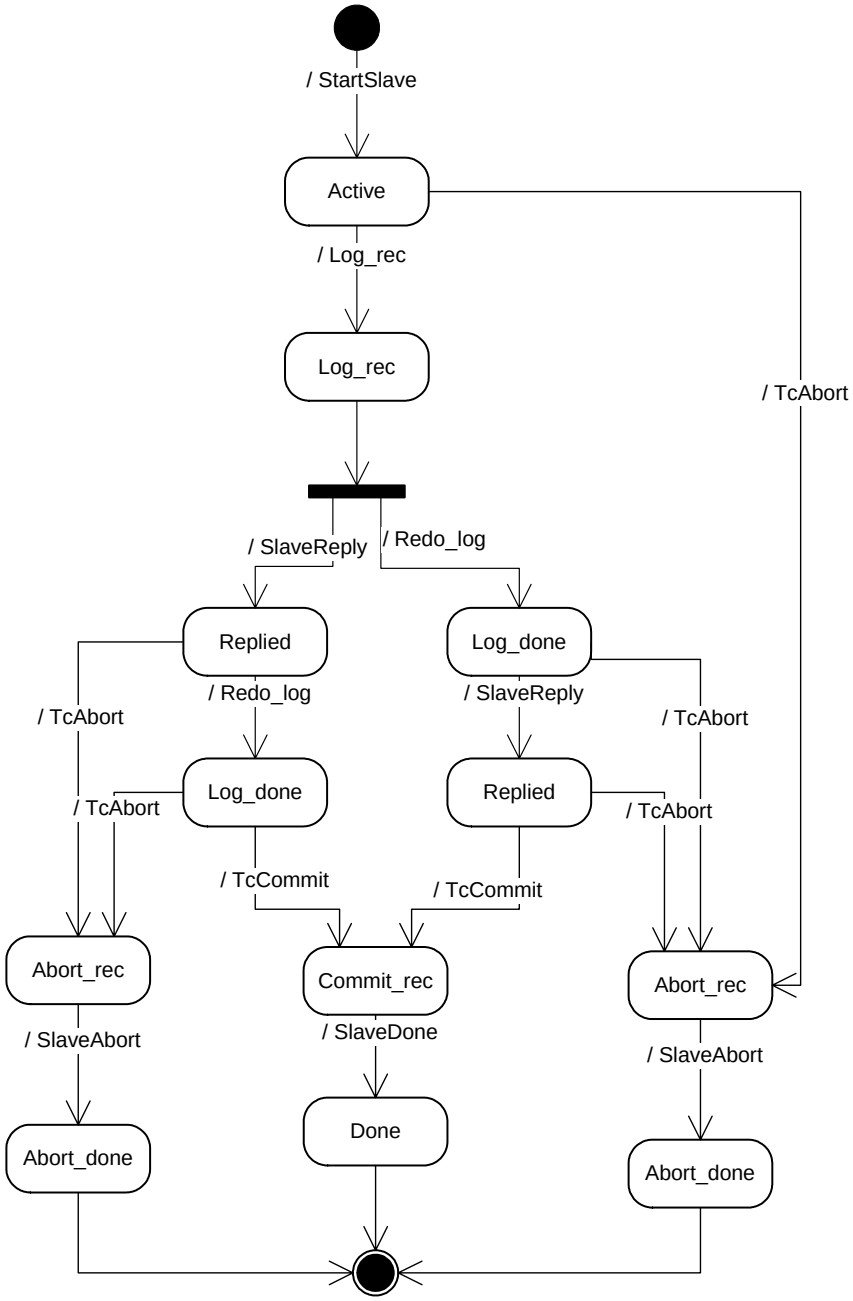
Implementasjon

I figur 6.15 er klassen `HSSlaveThread`, som arver `SlaveThread`, vist sammen med de andre klassene i transaksjonshierarkiet. Denne figuren viser de mest sentrale metodene i klassen:

- `memomryUpdate(int numberOfUpdates)` som finner tiden oppdateringene tar ved å kalle `calculateMemory(int numberOfUpdates)` og sove den tiden som metoden returnerer ved kall av `hold(SimTime time)`. Oppdateringene til `HSSlaveThread` tar ikke like lang tid å utføre som oppdateringene til `PSlaveThread` på grunn av loggproduksjonen og loggsendingen som `PSlaveThread` må utføre. Hvor stor del av tiden til `PSlaveThread` som `HSSlaveThread` skal bruke styres av parameteren `updateScalePrimayHS`. Standardverdi for denne parameteren er 0,7.
- `doAbort(UpdateTrans trans)` utfører prosesseringen ved abort av transaksjonen. Dette vil i vår simulator innebære det samme arbeidet som ved oppdatering mot minnet.
- `diskUpdate(int numberOfUpdates, UpdateTrans trans)` kaller metoden `doDiskUpdate(int numberOfUpdates, UpdateTrans trans)` og legger diskjobbene i kø på disk.
- `lifeCycle()` styrer utførelsen av meldingssending og oppdateringer på disk og i minne for tråden. Dette styres av tilstandsendringer og meldingssendingen utføres ved egne metoder for hver melding. Vi har også implementert egne metoder for meldingene tråden mottar, disse meldingene forklares ikke i mer detalj her. Det samme gjelder de ulike tilstandsattributtene.

Tilstander

Tilstandsdiagrammet for `HSSlaveThread` er vist i figur 6.19. I figuren er tilstandene *Abort_rec*, *Abort_done*, *Log_done* og *Replied* representert to steder. Dette er gjort for å gjøre diagrammet mest mulig oversiktlig.



Figur 6.19: Tilstandsdiagram for HSlaveThread

`HSSlaveThread` startes ved meldingen *StartHSSlave* på samme måte som `PSlaveThread` startes av *StartPSlave*. Forskjellen videre er at `HSSlaveThread` går til tilstanden *Active* og blir der til den mottar meldingen *Logging* fra `PSlaveThread`. Etter at loggmeldingen er mottatt, vil `HSSlaveThread` gå til tilstanden *Log_rec*.

Fra tilstanden *Log_rec* kan `HSSlaveThread` gå til to ulike tilstander, avhengig om loggen skal utføres før eller etter `HSSlaveThread` svarer med *SlaveReply* til transaksjonens `PTrolThread`. Det høyre løpet i figuren fra splitten etter tilstanden *Log_rec* viser det tilfellet der loggen utføres før svaret til `PTrolThread` sendes, mens det venstre løpet viser hendelsesforløpet der *SlaveReply* sendes før loggen utføres. Uavhengig av hvilken metode man velger å benytte, går `HSSlaveThread` til *Commit_rec* etter at den har mottatt *TcCommit*. Fra tilstanden *Commit_rec* og utover er hendelsene de samme som for `PSlaveThread`: `HSSlaveThread` sender *SlaveDone*

som svar på *TcCommit* og går til tilstanden *Done,* som indikerer at transaksjonen er ferdig på hot standby-noden.

`HSSlaveThread` kan motta abort i mange tilstander. Dersom `PSlaveThread` aldri rekker å sende loggen til `HSSlaveThread` før `PTrolThread` avgjør at transaksjonen skal abortere, vil `HSSlaveThread` motta *TcAbort* mens den er i initielltilstanden *Active*. `HSSlaveThread` vil ikke kunne gå fra *Log_rec* til *Abort_rec* siden loggmeldingen da vil ligge foran abortmeldingen i køen, og derfor blir lest først. Loggmeldingene vil føre til at `HSSlaveThread` skifter tilstand før en eventuell *TcAbort* tolkes på noden. Etter at loggen er mottatt vil `HSSlaveThread` kunne motta abortmelding i de to tilstandene *Log_done* og *Replied*, uavhengig av om man svarer `PTrolThread` før eller etter utførelse av loggpostene. Ved mottak av *TcAbort* er prosedyren den samme som for `PSlaveThread`. `HSSlaveThread` sender *SlaveAbort* til `PTrolThread` og går til tilstanden *Abort_done*.

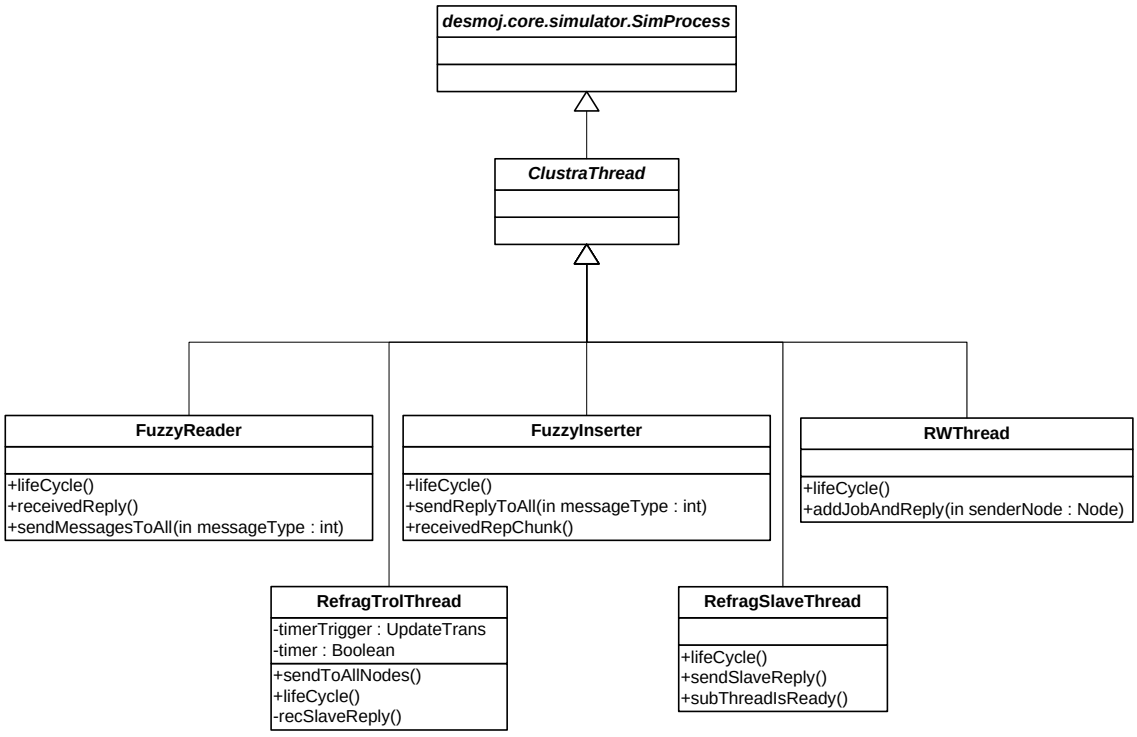
6.5 Refragmenteringstransaksjoner

Refragmenteringen er i hovedsak tatt med i vår simulator for å gi systemet varierende belastning og gi en periode hvor belastningen blir markant høyere enn ellers. Refragmenteringen styres av følgende parametere, med standardverdier angitt:

- `dataVolume = 5000` er antall 132 bytes tupler som en refragmentering involverer
- `refragStartTime = 10000` angir tiden i simuleringen for når refragmenteringen starter

Parameterne som gjelder meldingslengde og tolkingen av meldinger for refragmenteringen er forklart sammen med øvrige parametere i vedlegg A.

I simulatoren er refragmenteringen forenklet i forhold til virkeligheten. Hver node har statisk allokert fem tråder. Disse fem trådene er implementert gjennom klassene `RefragTrolThread`, `RefragSlaveThread`, `FuzzyReader`, `FuzzyInserter` og `RWThread`. Disse klassene arver `ClustThread`. Hierarkiet for disse klassene er vist i figur 6.20 med de mest sentrale metodene i hver av klassene.



Figur 6.20: Klassediagram refragmenteringshierarki

Vi vil nå se nærmere på hver av disse klassene som implementerer refragmentering. Vi velger å ikke omtale metodene og tilstandsendringene i disse klassene spesifikt i beskrivelsene under. Vi

viser til kildekoden og javadoc for nærmere beskrivelse av hvordan disse klassene er implementert.

6.5.1 RefragTrolThread

`RefragTrolThread` er en spesiell `TCON`-tråd som aktiveres av meldingen *RefragTable*. Tråden aktiverer refragmentering ved å sende *RefragSlave* til alle nodene i systemet. Den venter på svar fra alle slavene. Når den har fått det, svarer den til klienten.

6.5.2 RefragSlaveThread

`RefragSlaveThread` er en tråd som kontrollerer refragmenteringsaktiviteten på slaven. Først starter den subtrådene sine, `FuzzyReader`, `FuzzyInserter` og `RWThread`. Deretter venter den og teller opp subtrådene når de blir ferdige. Når den har fått telt opp, vil den svare med *RefragSlaveAck* til kontrolleren.

`RefragSlaveThread` starter også ekstra loggsending. Det vil si alle loggposter sendes nå i tre kopier, en *Logging*-melding og to *SlowLog*-meldinger. Denne ekstrasendingen blir stoppet når refragmenteringen er ferdig. Selve loggsendingen blir gjort av `PSlaveThread`, som for vanlig loggsending.

6.5.3 FuzzyReader

`FuzzyReader` leser dataene sekvensielt og sender dem til alle destinasjonsnodene i parallell. Hver post blir sendt til to noder, en ny primær og en ny hot standby.

Vi antar perfekt fordeling av data. Vi sender fulle meldinger til alle nodene. Egentlig skulle det gått to meldinger til hver node, en med primærdata og en med hot standby-data. Vi har i stedet sendt en dobbelt så stor melding til hver node.

`FuzzyReader` vil typisk lese en del sider uten å få avbrudd, den prefetcher en del blokker. Vi har her antatt at den prefetcher fem blokker. Når den har lest en batch med fem blokker, kan det oppstå diskavbrudd. `FuzzyReader` har et bestemt antall poster den skal lese. Dette tallet er et totalt antall poster delt på antall noder i systemet.

Hver blokk vil inneholde et fast antall poster, som er antatt å være 132 bytes poster 67% fyllt opp i 8KB-blokker. Det er plass til 40 poster i hver blokk og 67 poster i hver UDP-melding.

Når `FuzzyReader` har sendt *RepChunk* til alle nodene, vil den vente på *RepChunkReply* fra alle nodene før den fortsetter å lese og sende data.

Når den har sendt all data og fått alle svar, vil den sende *RepDone* til alle nodene, og vente på *RepDoneReply* fra alle nodene. Til slutt vil den fortelle `RefragSlaveThread` at den er ferdig.

6.5.4 FuzzyInserter

Oppgaven til `FuzzyInserter` er å sette inn data som den mottar fra alle lesetrådene. `FuzzyInserter` vil flette data, slik at innsettingen blir sekvensiell. Dette betyr i praksis at den vil vente på *RepChunk* fra alle nodene før den setter inn data. Når den har mottatt alle *RepChunk*-meldingene, vil den holde CPUen en stund mens den setter inn data. Etter å ha satt inn data, sender den svar, *RepChunkReply*, til alle nodene.

Vi antar at `FuzzyInserter` ikke kan få diskavbrudd fordi den setter inn nye data i buffer. Det er en mulighet at den kan få avbrudd i virkeligheten, da den muligens må vente på at sjekkpunkteren skriver ut skitne blokker fra bufferet. Vi antar at dette ikke skjer i simulatoren, siden vi antar at sjekkpunkteren har kjørt ofte nok.

Når `FuzzyInserter` har fått *RepDone* fra alle nodene, vil den svare med *RepDoneReply*, og så vil den fortelle `RefragSlaveThread` at den er ferdig.

6.5.5 RWThread

`RWThread` er en svært forenklet utgave av `RecoverWindowThread`. Tråden mottar *SlowLog*-meldinger og gjør redo på loggpostene i disse meldingene. `RWThread` kan få diskavbrudd, for den gjør redo på tilfeldige poster rundt omkring i databasen. Vi antar at den gjør redo med det samme, selv om noen loggposter gjelder poster som ennå ikke er ankommet via FuzzyCopy-prosessen.

7 Oppsett av ressurskontroll

I dette kapitlet bestemmer vi parameterne til ressurskontrollen. Dette utfører vi ved å se på andre relevante systemer og å gjennomføre prøve-simuleringer. Vi vil i dette kapitlet først presentere systemoppsettet vi har gjort disse prøve-simuleringene under, før vi til slutt bestemmer verdiene av de ulike ressurskontroll-parameterne.

7.1 Systemoppsett

I dette underkapitlet beskriver vi hvilket system vi velger å sette opp ressurskontroll-parameterne under. Systemet settes opp litt ekstremt slik at vi har et behov for ressurskontroll, men likevel ikke så ekstremt at alle transaksjoner slutter å fullføre under kjøring uten ressurskontroll. Vi har valgt å gjøre disse prøvesimuleringene uten refragmentering.

Systemparameterne som velges er vist i tabell 7.1. De resterende systemparameterne er valgt basert på målinger under kjøring av ClustRa. Verdiene på disse parameterne sammen med en beskrivelse av dem, er å finne i vedlegg A.

Tabell 7.1: Systemparametere under setting av RC-parametre

Parameter	Verdi
mtBetweenArrivals	20 ms
probPageFault	0,2
numberOfNodes	4
numberOfDisks	1
numberOfUpdates	4

Verdien på gjennomsnittlig tid mellom ankomster settes til 20 ms for at systemet skal bli belastet under kjøringene. Vi har også valgt å kjøre med kun én disk og en sannsynlighet for diskavbrudd på 0,2. Disse to valgene fører til at flere transaksjoner får diskavbrudd, og at nodene dermed opplever seg selv som opptatt på grunn av både lang diskkø og CPU-bruk. Systemet vil likevel kunne fullføre transaksjoner gjennom hele kjøringen siden alle oppdateringene statistisk sett ikke må utføres på disken. Antall oppdateringer settes til fire per transaksjon, og systemet er satt opp til å ha fire noder. Dette systemet er også valgt slik da det er mot et slikt ClustRa-oppsett vi har testdata tilgjengelig.

7.2 Ressurskontrolloppsett

I dette underkapitlet bestemmer vi verdiene på parameterne i ressurskontrollen. For de fleste av parameterne har vi ikke noe reelt system å sammenligne med, slik at verdiene blir bestemt på bakgrunn av prøve-simuleringer, erfaringer fra andre relevante systemer og argumentasjon. Dette har sin bakgrunn i at det ikke eksisterer en slik ressurskontroll i ClustRa i dag. I tabell 7.2 vises en oppsummering av parameterne og hvilken verdi de har, etterfulgt av en begrunnelse for disse verdiene.

Tabell 7.2: Ressurskontroll-parametere

Parameter	Beskrivelse	Verdi
CPUBusyBoundary	Andelen av tiden CPUen må være aktiv for at CPUen skal regnes som *Busy*	0,8
numberOfCPUValues	Antall målinger av CPU-lasten som brukes for å sette tilstanden til en node. Ved et antall større enn 1 brukes snittet av målingene.	3

Tabell 7.2: Ressurskontroll-parametere

Parameter	Beskrivelse	Verdi
DiskBusyBoundary	Minimum antall diskjobber i kø på en node for å sette tilstanden *Busy*. Har en node flere disker beregnes snittet av diskkøene.	10
numberOfDiskValues	Antall målinger av disklasten som brukes for å sette tilstanden til en node. Ved et antall større enn 1 brukes snittet av målingene.	1
timeoutTransactionClearance	Timeout-tiden for å motta svar fra de deltakende noder ved transaksjonsklarering	20 ms
interval_rc	Tiden mellom hver gang ressurskontroll skal utføres	5 ms
internal_RC_time	Tiden det tar å utføre intern ressurskontroll	0,479 ms
internalResourceControlType	Bestemmer hvilken type intern ressurskontroll som skal utføres når ressurskontrollen er aktiv. Følgende muligheter: 0 - Disklasten, 1 - CPU-lasten og 2 - Både disk- og CPU-last.	2

CPUBusyBondary

Denne verdien er den grensen som angir CPUen som opptatt. En tallfestet verdi for dette setter IBM til 80% i [11].

numberOfCPUValues

Når CPUens belastning skal beregnes, er antall måleperioder det skal tas hensyn til nødvendig å bestemme. Hvis man ser på én måling, vil gjennomsnittlig arbeid siste periode ligge til grunn for beregningene. Dersom flere perioder tas med i beregningene, vil en høy belastning siste periode totalt sett kunne bli begrenset av et lavt gjennomsnitt i en tidligere periode. Vi får dermed en avveining mellom rask tilstandsendring og mange tilstandsendringer, mot mer sjelden tilstandsendring og færre tilstandsendringer. Gjennom prøve-simuleringer og eksemplet vi har presentert i kapittel 4.1.3, har vi bestemt verdien på denne parameteren til å være 3. Dette mener vi gir en god mellom-løsning som reflekterer endringer i CPU-belastningen raskt, men samtidig ikke får unødvendig mange tilstandsendringer.

diskBusyBoundary

Denne verdien angir hvor lang diskkøen skal tillates å være før disken sees på som opptatt. For å bestemme den lengden på diskkøen som skulle definere en node som opptatt, simuleres systemet presentert i kapittel 7.1 med kølengder på 5, 10 og 15 diskjobber. Resultatene er vist i kode-eksempel 7.1.

```
-----------------------------------------------------------------
LENGDE 5:
distributedResourceControlType: 1
mtBetweenArrivals: 20
probPageFault: 0.2

Refrag completed at: 0.0
Average commit time: 16.74895801497793
Number of rejected: 898
Number of aborted: 0
Number of committed: 3089
Number of transMsgs: 105381
Number of RCMsgs: 4632
-----------------------------------------------------------------
LENGDE 10:
distributedResourceControlType: 1
mtBetweenArrivals: 20
probPageFault: 0.2

Refrag completed at: 0.0
Average commit time: 16.61160876484845
Number of rejected: 847
Number of aborted: 0
Number of committed: 3139
Number of transMsgs: 106587
Number of RCMsgs: 4160
-----------------------------------------------------------------
LENGDE 15:
distributedResourceControlType: 1
mtBetweenArrivals: 20
probPageFault: 0.2

Refrag completed at: 0.0
Average commit time: 17.057905147269643
Number of rejected: 897
Number of aborted: 0
Number of committed: 3091
Number of transMsgs: 105565
Number of RCMsgs: 4528
```

Kode-eksempel 7.1: Prøve-simuleringer for å bestemme lengden på diskkøen

Av figuren kan vi se at gjennomsnittlig responstid er lavest ved maksimalt 10 jobber i diskkøen. Videre ser vi at antall fullførte transaksjoner er høyest ved en lengde på 10 og at antall meldinger for å utføre ressurskontroll er lavest. Dette gjør at vi finner en grenseverdi på 10 jobber som best og at vi velger dette som standardverdi.

numberOfDiskValues

Antall målinger man går tilbake i tid for å beregne lengden på diskkøen, føles riktig ut fra prøvesimuleringer å sette til én. Anta at siste registrerte måling sier at det var 10 jobber i diskkøen og det er 20 jobber i køen nå. Det er da irrelevant at gjennomsnittlig kølengde er 15 diskjobber, siden den faktiske belastningen på disken er 20 diskjobber. Dermed mener vi at den siste verdien bør reflekteres i ressurskontrollen.

timeoutTransactionClearance

Dette er tiden transaksjonsklareringen maksimalt kan vente på svar fra de involverte nodene i en transaksjon. Tidsgrensen er med på å sikre at transaksjonen blir avvist dersom svarmeldinger

uteblir i systemet. I tillegg er grensen med på å sikre at transaksjonen blir gjennomført på en akseptabel tid. Hvis de involverte nodene ikke klarer å svare på en melding innen denne tiden, vil de mest sannsynlig heller ikke klare å utføre transaksjonen innenfor den øvre tidsgrensen for transaksjonsutførelse. Vi har valgt å sette tidsgrensen til 20 ms. Denne verdien er 10% av tidsgrensen for abortering av en transaksjon, og ut fra prøve-simuleringer ser dette ut til å være en fornuftig verdi.

interval_rc

Intervallet betegner hvor ofte intern- og distribuert ressurskontroll skal utføres. Kostnaden ved å utføre distribuert ressurskontroll er alle meldingene som sendes. Foretas distribuert ressurskontroll sjeldnere, vil informasjonen man sitter på kunne være for gammel til å korrekt avgjøre belastningen til noden ved et gitt tidspunkt. Nodene i simulatoren sender ikke ut meldinger om sin egen last dersom det ikke er noen endring i den interne tilstanden fra forrige måling. Denne mekanismen gjør at antall meldinger som følge av ressurskontrollen ikke øker dramatisk selv ved små intervaller for utførelse av ressurskontroll. Vi har valgt å sette ressurskontrollintervallet til 5 ms etter flere prøvesimuleringer.

internal_rc_time

Tiden det tar å utføre en intern ressurskontrollmåling har vi satt lik tiden det tar å gjøre en tuppeloppdatering mot minnet. Tiden en slik oppdatering tar er målt i ClustRa til 0,479 ms ved testing.

internalResourceControlType

Denne parameteren styrer hvilken intern ressurskontroll-type som skal brukes. Vi har her implementert alternativene CPU-last, diskkøens lengde og en kombinasjon av de to. Gjennom prøve-simuleringer har vi sett at om vi kun ser på CPU-lasten eller diskkøens lengde, blir systemoppsettet avgjørende for hvor god ressurskontrollen er. Dette henger sammen med at et systemoppsettet uten diskbelastning følgelig ikke vil gi noen diskø, og at det dermed er irrelevant å måle lengden på diskøen og en nødvendighet å se på CPU-lasten om vi skal få til ressurskontroll. Tilsvarende ser vi at disken blir systemets flaskehals for systemer med større sannsynlighet for diskaksess, og det er derfor av mindre interesse å se på CPU-belastningen. På bakgrunn av dette finner vi det nødvendig og riktig å se på begge målingene samtidig for å sette korrekt tilstand på en node. Denne parameteren får derfor verdien 2.

8 Simulering og resultater

I dette kapitlet vil vi presentere våre simuleringer og se på resultatene av simuleringene. Simuleringene våre utføres ut i fra en rekke parametere som settes i egne innstillingsfiler. Verdiene på de øvrige parameterne er beskrevet i vedlegg A. De verdiene vi har kommet fram til der er basert på testing og reelle måleresultater fra ClustRa, presentert til oss av veileder Svein Erik Bratsberg. I kapittel 7 har vi begrunnet valgene vi har gjort for ressurskontroll-parameterne. De øvrige parameterne som er variable, men likevel faste under vår simulering, er fastsatt ut i fra hvordan ClustRa har vært satt opp under testingen. Følgelig ønsker vi å sette opp systemet på samme måte for å kunne sammenligne best mulig. Vi går ikke nærmere inn på fastsettingen av disse verdiene her.

Vi velger å kjøre simuleringer med og uten refragmentering for å kartlegge ressurskontrollens oppførsel i de ulike systemene. Dette gjør vi for å se på hvordan systemet oppfører seg med og uten den økte belastningen som refragmenteringen medfører. Vi vil i dette kapitlet først presentere hvilke parametere vi ønsker å variere under simuleringene, og begrunne hvorfor disse parameterne er valgt og hvilke verdier de har under simuleringene. Vi ser også på hvordan simuleringene våre gjennomføres og hvilke resultater vi ser på etter hver simulering. Videre presenterer vi simuleringsresultatene våre for simulering med refragmentering, før vi presenterer resultatene fra simulering uten refragmentering. Til slutt oppsummeres simuleringen ved at de ulike ressurskontroll-alternativene blir sammenlignet.

8.1 Valg av simuleringsparametere

I simuleringene vi utfører er det sentralt å finne ut når ressurskontroll er mest hensiktsmessig å bruke og hvilken type ressurskontroll som er best for ulike tilfeller av belastning på systemet. For å oppnå dette er det viktig at vi velger parametere som gir ønskede simuleringssituasjoner.

Vi har endt opp med å variere to parametere for systemoppsettet:

- `mtBetweenArrivals` som styrer den gjennomsnittlige tiden mellom hver ankomst av transaksjoner til systemet
- `probPageFault` som angir sannsynligheten for at en dataaksess må gjøres på disken

Vi har mange parametere som kan endres i vår simulator og har dermed mange frihetsgrader. Det er derfor uoverkommelig å endre alle parameterne under simulering, både med henblikk på tiden dette ville tatt og at resultatene lett ville blitt uoversiktelige å sammenligne. Ved å endre `mtBetweenArrivals` og `probPageFault`, mener vi å oppnå to hovedaspekter som vi i samarbeid med veileder Svein Erik Bratsberg har funnet mest interessante: Å variere systembelastningen og å variere forholdet mellom belastning av prosessor og disk. Ved å se på ulike kombinasjoner av disse to parameterne mener vi å få til dette på en god måte.

Gjennom flere prøve-simuleringer og sammenligninger av resultatene har vi kommet til verdiene som parameterne skal ha under simuleringene. Dette har vi gjort ved å prøve verdier i området 10 til 100 ms for tiden mellom hver ankomst av transaksjoner og verdier mellom 0 og 1 for sannsynligheten for disk. Vi har sammenlignet antall fullførte, aborterte og avviste transaksjoner, antall meldinger og gjennomsnittlig commit-tid for transaksjonene for å komme fram til verdiene.

Vi vil videre i dette underkapitlet først se på hvilke verdier vi har valgt å simulere for `mtBetweenArrivals` før vi ser på `probPageFault`. Vi diskuterer også hvor lenge simuleringene skal kjøre med og uten refragmentering.

8.1.1 Verdier for mtBetweenArrivals

I vedlegg E har vi skrevet om køteori med et illustrerende eksempel. Denne teorien og beskrivelsen finner vi spesielt relevant for å bestemme ankomstraten til systemet, som vi regulerer gjennom parameteren `mtBetweenArrivals`. Det vi her ser i forhold til vår simulering er at antall transaksjoner i systemet og den gjennomsnittlige responstiden stiger veldig raskt ved liten endring i ankomstraten når systemet nærmer seg en bestemt grenseverdi. Dette oppstår når tiden mellom hver ankomst nærmer seg behandlingstiden i systemet. Vi ønsker her å finne simuleringssituasjoner som fanger opp nettopp denne situasjonen og velger parametere hvor vi ser systemets utvikling rundt disse grensetilfellene. I tabell 8.1 er verdiene for `mtBetweenArrivals` vist.

Tabell 8.1: Simuleringsverdier for mtBetweenArrivals

Simuleringssituasjon	Verdier
Med refragmentering	40 - 50 - 60 ms
Uten refragmentering	15 - 20 ms

Vi ser at verdiene er større for simuleringene med refragmentering enn for simuleringene uten refragmentering. Grunnen til dette er at refragmenteringstransaksjonene er så ressurskrevende, at dersom de skal kunne fullføre innenfor simuleringstiden som er satt, er vi nødt til å øke tiden mellom hver ankomst av oppdateringstransaksjoner. Vi ønsker også at hver refragmentering skal ha en viss lengde for å få fram ønsket effekt, slik at vi ikke ønsker å gjøre denne perioden kortere. For simuleringer uten refragmentering er tiden mellom ankomstene av oppdateringstransaksjonene mindre, i den hensikt å belyse ressurskontrollens innvirkning på systemer som befinner seg over og under grensen der ankomstraten er lik behandlingsraten.

8.1.2 Verdier for probPageFault

Parameteren `probPageFault` styrer sannsynligheten for at en dataaksess må utføres på disken. Dette har derfor innvirkning på hvor lang tid hver transaksjon bruker i systemet og hvor sterkt prosessor og disk blir belastet på hver enkelt node. Verdiene for `probPageFault` er vist i tabell 8.2.

Tabell 8.2: Simuleringsverdier for probPageFault

Simuleringssituasjon	Verdier
Med refragmentering	0 - 0,2
Uten refragmentering	0 - 0,2 - 0,4

Verdien på disksannsynlighetene ved simuleringene med refragmentering er satt til 0 og 0,2. Et system med disksannsynlighet lik 0 er en database der alle data ligger i minnet. ClustRa var opprinnelig et slikt system. Når vi skal velge en disksannsynlighet større enn 0, har vi i samarbeid med veileder Svein Erik Bratsberg funnet det naturlig å anta at systemet vårt følger Pareos 80-20-regel [12]. For vårt tilfelle betyr det at 80 % av forespørslene går mot 20 % av dataene. Hvis vi i tillegg antar at systemet vårt har plass til 20 % av dataene i minnet, så medfører dette at vi har et system der disksannsynligheten er 0,2.

For simuleringene uten refragmentering har vi valgt å simulere med disksannsynligheter på 0 og 0,2 av samme grunn som over. I tillegg har vi valgt å simulere et system med disksannsynlighet på 0,4. Et system med disksannsynlighet på 0,4 aksesserer disken unormalt mye, men vi velger å ta det med for å vise hvordan ressurskontrollen virker i et slikt belastet system.

8.1.3 Simuleringstid

Ved å endre verdien til `stopTime` vil man endre lengden på simuleringen. I tabell 8.3 er de ulike verdiene til `stopTime` vist, for systemene med og uten refragmentering.

Tabell 8.3: Simuleringsverdier for stopTime

Simuleringssituasjon	Verdier
Med refragmentering	60 000 ms
Uten refragmentering	20 000 ms

Simuleringens lengde er satt til 60 sekunder for simuleringene med refragmentering og 20 sekunder for simuleringene uten refragmentering. Ved å simulere 60 sekunder der hvor refragmentering er med, vil man gi refragmenteringen mulighet til å fullføre for de kombinasjonene av ankomsttid og disksannsynlighet vi velger å simulere over. At simuleringen er satt til 20 sekunder der refragmentering ikke er med, kommer av at simulatoren skal klare å fullføre kjøringen

selv etter at systemet har gått i stå. Den tiden som er satt er tiden i vår simulator, simuleringstiden, til forskjell fra den reelle tiden. Dette fører til at hvilken maskin simuleringene utføres på, ikke har noen innvirkning på resultatene.

8.2 Gjennomføring av simuleringen

Vi utfører simuleringene ved først å låse verdien på `mtBetweenArrivals`, og for hver verdi av `mtBetweenArrivals` vil vi simulere over alle verdiene til `probPageFault`. For hvert sett av verdier gjennomføres simuleringen uten ressurskontroll og med hver av de tre typene ressurskontroll som vi har implementert. Dette gir oss 24 kombinasjoner for simulering med refragmentering og 24 kombinasjoner for simulering uten refragmentering.

For hver simulering viser vi en tabell med resultater slik vi ser i tabell 8.4. Kolonnenavnene betyr her:

- Ingen rk, ingen ressurskontroll er brukt
- DN, distribuert nodetabell
- TK, transaksjonsklarering
- SN, sentralisert nodetabell

Vi ser på følgende resultater ved hver simulering som er utført:

- Refragmentering ferdig, tidspunktet i simuleringen som refragmentering fullfører
- Gjennomsnittlig transaksjonstid, som er den gjennomsnittlige tiden transaksjonene som fullfører bruker i systemet. Dette er tiden slik klienten opplever det, altså responstiden til systemet.
- Antall avviste, som er antall transaksjoner som avvises som en følge av ressurskontroll
- Antall aborterte, som angir antall transaksjoner som aborterer
- Antall fullførte, som angir antall transaksjoner som fullfører
- Antall transaksjonsmeldinger er antall meldinger som totalt sendes i systemet for å utføre transaksjonsprosesseringen
- Antall ressurskontrollmeldinger er antall meldinger som totalt sendes for å utføre ressurskontroll

Tabell 8.4: Eksempel på tabell med simuleringsresultater

Resultat	Ingen rk	DN	TK	SN
Refragmentering ferdig (SimTime)				
Gjennomsnittlig transaksjonstid (SimTime)				
Antall avviste				
Antall aborterte				
Antall fullførte				
Antall transaksjonsmeldinger				
Antall ressurskontrollmeldinger				

Etter hver tabell presenterer vi en graf som sammenligner utviklingen til antall fullførte transaksjoner for hver av ressurskontroll-alternativene. Vi presenterer også en graf som viser antall fullførte, aborterte og avviste transaksjoner ved bruk av ressurskontroll og antall fullførte og aborterte transaksjoner uten bruk av ressurskontroll for det samme systemoppsettet.

Alle tall, grafer og drøftinger vi presenterer her, er basert på rapportene og filene som genereres ved hver simulering. Rapportene er nærmere forklart i vedlegg F.

8.3 Simuleringsresultater - med refragmentering

I dette underkapitlet vil vi presentere simuleringsresultatene med refragmentering. Denne simuleringen er utført i 60 sekunder i simuleringstiden (`SimTime`). Refragmenteringstransaksjonen startes alltid etter 10 sekunder (`SimTime`).

8.3.1 mtBetweenArrivals: 40 ms

Vi vil her presentere resultatene for simuleringen med refragmentering med 40 millisekunders simuleringstid mellom hver ankomst av nye transaksjoner til systemet.

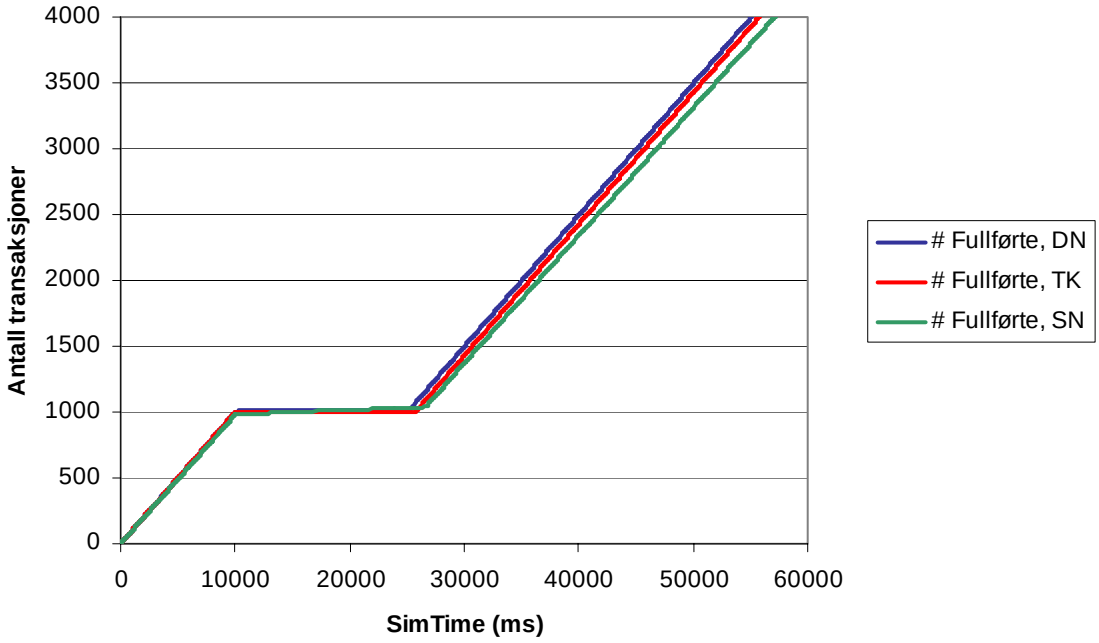
probPageFault: 0,0

I tabell 8.5 ser vi simuleringsresultatene fra simuleringen med en disksannsynlighet på 0,0.

Tabell 8.5: Simulering: Refragmentering, Ankomst: 40 ms, P(Disk): 0.0

Resultat	Ingen rk	DN	TK	SN
Refragmentering ferdig (SimTime)	51455,40	25094,83	25708,03	26633,76
Gjennomsnittlig transaksjonstid (SimTime)	115,31	10,71	12,01	15,49
Antall avviste	0	1500	1568	1664
Antall aborterte	2817	0	0	45
Antall fullførte	3175	4492	4424	4283
Antall transaksjonsmeldinger	320492	152592	149037	148086
Antall ressurskontrollmeldinger	0	6528	32724	13820

Av tallene presentert i tabell 8.5 ser vi at distribuert nodetabell gir lavest snitt for transkasjonstiden av resssurskontroll-alternativene og at refragmenteringen fullfører først ved dette alternativet. Vi ser videre at dette alternativet gir færrest avviste transaksjoner, flest fullførte transaksjoner og færrest meldinger. I figur 8.1 ser vi en grafisk framstilling av utviklingen i antall fullførte transaksjoner for de tre ressurskontroll-alternativene. Vi ser at de tre typene starter likt og at alle sammen stopper å fullføre nye transaksjoner mens refragmenteringen pågår fra 10 000 ms til de er ferdige rundt 25 000 ms. Fra dette punktet ser vi at distribuert nodetabell gir mest økning i antall fullførte transaksjoner. Dette henger også sammen med at denne løsningen stopper refragmenteringen tidligere og følgelig gir et mindre belastet system tidligere enn de andre. Ut i fra disse betrakningene finner vi løsningen distribuert nodetabell som det beste alternativet av ressurskontroll-typene her.



Figur 8.1: Simuleringsgraf: Refragmentering, 40 ms, 0.0, # Fullførte ved RK

Om vi sammenligner ressurskontroll-alternativene med systemet uten ressurskontroll, ser vi at alle systemene med ressurskontroll er bedre enn systemet uten for alle parameterne i tabell 8.5. Dette illustreres også av figur 8.2 hvor vi har sammenlignet distribuert nodetabell med systemet uten ressurskontroll i forhold til antall transaksjoner som avvises, aborteres og fullføres.

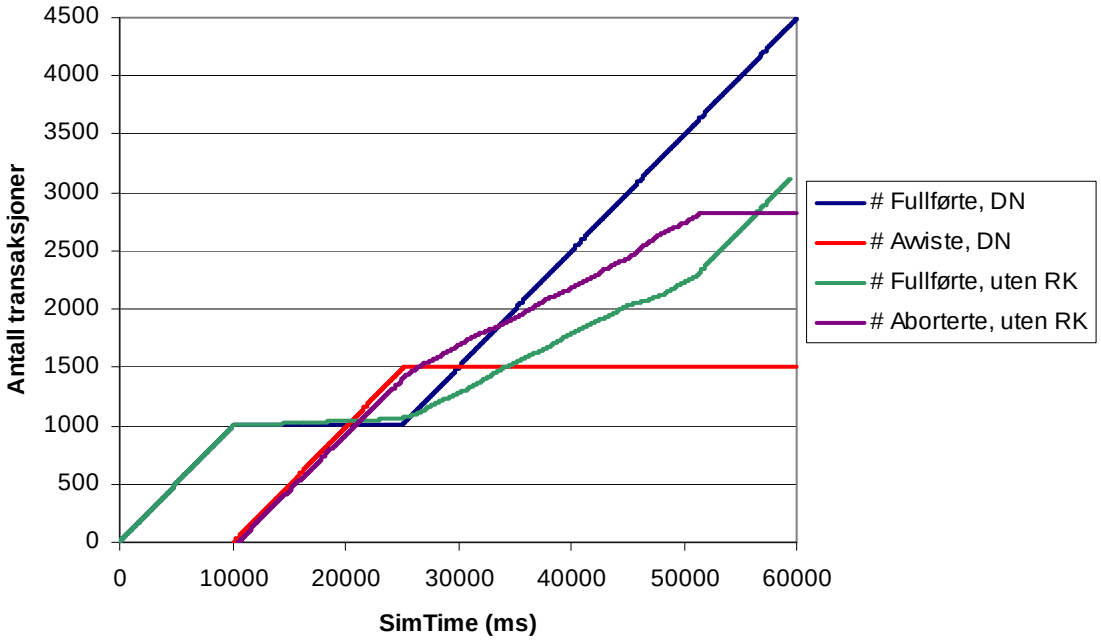
Her ser vi at antall fullførte uten ressurskontroll og med distribuert nodetabell følges i starten, ingen transaksjoner blir abortert eller avvist og følgelig følges de to grafene. Fra refragmenteringen starter ved 10 000 ms, ser vi at systemet med distribuert nodetabell stopper å fullføre nye transaksjoner og alle avvises. Tilsvarende ser vi for systemet uten ressurskontroll at det aborterer de fleste av transaksjonene, men at noen også slipper gjennom. Dette gjør at det etter rundt 25 000 ms er flere fullførte transaksjoner for systemet uten ressurskontroll enn for systemet med distribuert nodetabell. Etter dette punktet ser vi at antall fullførte transaksjoner for distribuert nodetabell starter å øke igjen og ingen flere transaksjoner avvises, som en følge av at refragmenteringen er ferdig. Systemet uten ressurskontroll ser vi at fortsatt aborterer mange transaksjoner helt til refragmenteringen er ferdig etter drøyt 50 000 ms. Nettop denne forskjellen i når refragmenteringen fullføres er hva som gjør ressurskontroll mer gunstig for dette systemoppsettet. Det gjør at antall fullførte transaksjoner blir høyere, og antall avviste transaksjoner blir lavere enn antall aborterte transaksjoner, samtidig som den gjennomsnittlige transaksjonstiden blir mye lavere.

Hovedgrunnen til den store forskjellen i når refragmenteringen fullfører, er at det er mer krevende å abortere transaksjoner enn å avvise dem, da abort krever prosessering i motsetning til avvising som skjer før noe transaksjonsprosessering starter. Dette er også årsaken til det høye antallet transaksjonsmeldinger vi ser i tabell 8.5 for systemet uten ressurskontroll sammenlignet med de ulike ressurskontroll-alternativene. Grunnen til dette er at det å avvise transaksjoner ikke gir noen meldinger i systemet, i motsetning til det å abortere. Før en abort inntreffer vil det også skje flere kjøringer av timere, noe som gir sending av transaksjonsmeldingene på nytt, og dermed ekstra belastning på systemet. Til slutt vil også selve abortprosesseringen kreve flere meldinger etter at timeout-tiden for abort er passert. Abortprosessering er generelt forklart i kapittel 3.3.3, mens timerkjøring er nærmere forklart i kapittel 6.4.1.

Om vi studerer antall meldinger, ser vi at antall meldinger er langt høyere for systemet uten ressurskontroll enn for de ulike ressurskontroll-alternativene. Vi ser at systemet uten ressurskontroll sender 101 % flere meldinger enn distribuert nodetabell. Dette henger sammen med at konstnaden som abortprosesseringen gir med flere meldinger, slik vi forklarte i forrige avsnitt.

Et siste viktig poeng er forskjellen i gjennomsnittlig transaksjonstid. Denne ser vi er mye høyere for systemet uten ressurskontroll (115,31 ms) i forhold til systemene med ressurskontroll (fra 15,49 ms og nedover).

Ut i fra denne drøftingen finner vi ressurskontroll nyttig ut i fra alle målte parametere ved dette systemoppsettet.



Figur 8.2: Simuleringsgraf: Refragmentering, 40 ms, 0.0, Uten RK mot DN

probPageFault: 0,2

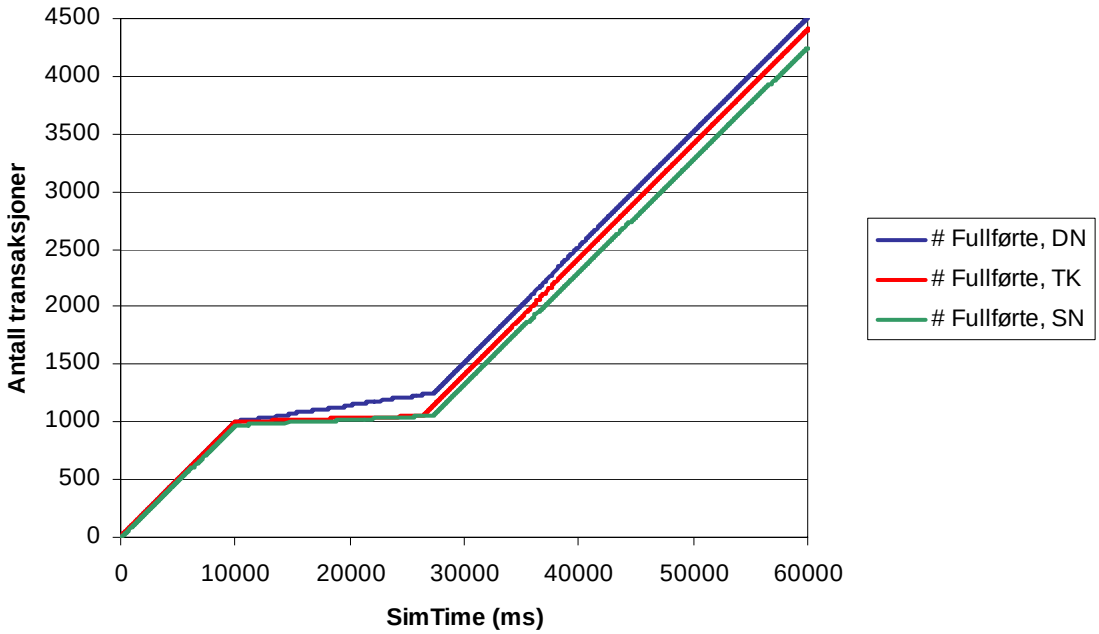
I tabell 8.6 ser vi simuleringsresultatene fra simuleringen med en disksannsynlighet på 0,2.

Tabell 8.6: Simulering: Refragmentering, Ankomst: 40 ms, P(Disk): 0.2

Resultat	Ingen rk	DN	TK	SN
Refragmentering ferdig (SimTime)	51090,61	27338,90	26376,77	27228,29
Gjennomsnittlig transaksjonstid (SimTime)	104,03	29,34	18,04	19,81
Antall avviste	0	1404	1578	1694
Antall aborterte	3237	74	3	47
Antall fullførte	2755	4511	4410	4248
Antall transaksjonsmeldinger	335470	168436	157567	155159
Antall ressurskontrollmeldinger	0	3032	32718	11900

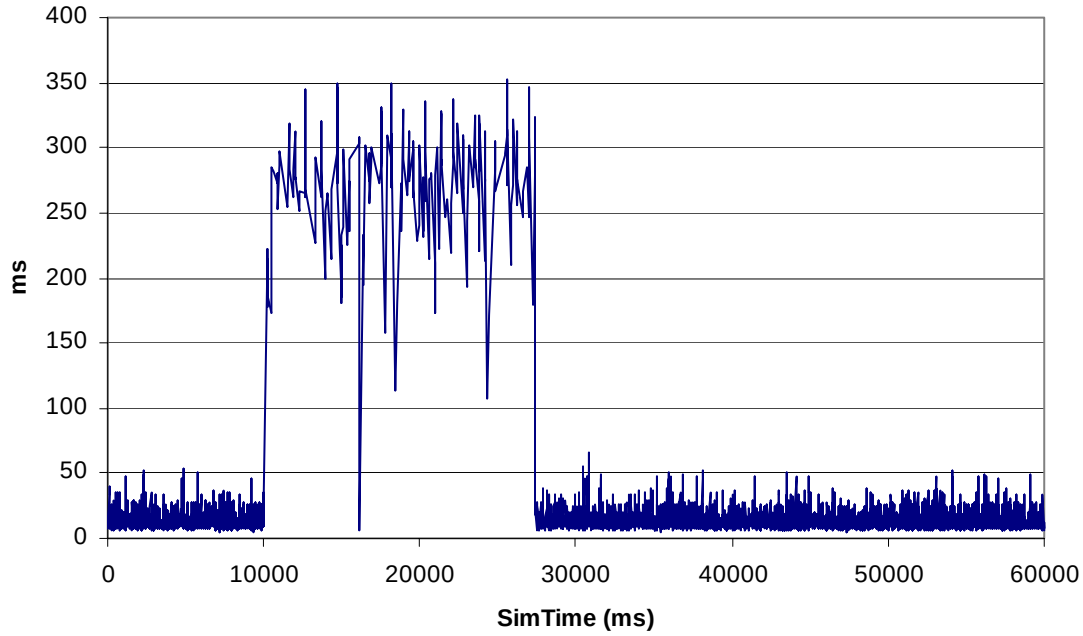
Vi ser her et mindre tydelig simuleringsresultat for ressurskontroll-alternativene. Om vi ser på gjennomsnittlig transaksjonstid, er den lavest for transaksjonsklarering, sentralisert nodetabell har 9,8 % høyere snitt, mens distribuert nodetabell har 63 % høyere snitt. Transaksjonsklarering fullfører også refragmentering først, mens sentralisert nodetabell og distribuert nodetabell er omtrent like, cirka 1 000 ms etter transaksjonsklarering. Ser vi derimot på antall fullførte og antall avviste transaksjoner kommer distribuert nodetabell best ut. Selv om denne løsningen har noe flere aborterte transaksjoner, ser vi at summen av aborterte og avviste transaksjoner er lavest for distribuert nodetabell, og slik sett den beste løsningen. Dette illustreres også av figur 8.3, hvor vi ser utviklingen i antall fullførte transaksjoner. Vi ser her at distribuert nodetabell klarer å fullføre flere transaksjoner under refragmenteringen (fra 10 000 ms til rundt 27 000 ms), mens de to andre nærmest stopper opp fullføringen av transaksjoner. Nettopp denne forskjellen forklarer også forskjellen i gjennomsnittlig transaksjonstid, siden transaksjonstiden under refragmentering er mye høyere enn ellers. Følgelig påvirker dette snittet i negativ retning i sterkere grad for distribuert

nodetabell enn for de andre ressurskontroll-alternativene, da disse ikke fullfører så mange transaksjoner i denne perioden.



Figur 8.3: Simuleringsgraf: Refragmentering, 40 ms, 0.2, # fullførte ved RK

At transaksjonstidene er langt høyere under refragmentering enn ellers vises i figur 8.4. Her ser vi en graf som plotter alle transaksjonstider for distribuert nodetabell. Vi ser her hvordan tidene går mye opp mens refragmenteringen pågår (fra 10 000 ms til rundt 27 000 ms), mens de holder seg stabilt mye lavere både før og etter refragmenteringen. Antall transaksjoner som får fullføre under refragmenteringen får derfor stor innvirkning på snittet på transaksjonstiden. Ut i fra en samlet vurdering finner vi distribuert nodetabell som den beste løsningen her, selv om den gir dårligere snitt på transaksjonstiden og senere fullføring av refragmenteringen. Likevel har den flest fullførte transaksjoner og fullfører transaksjoner også under refragmenteringen, noe vi finner positivt.

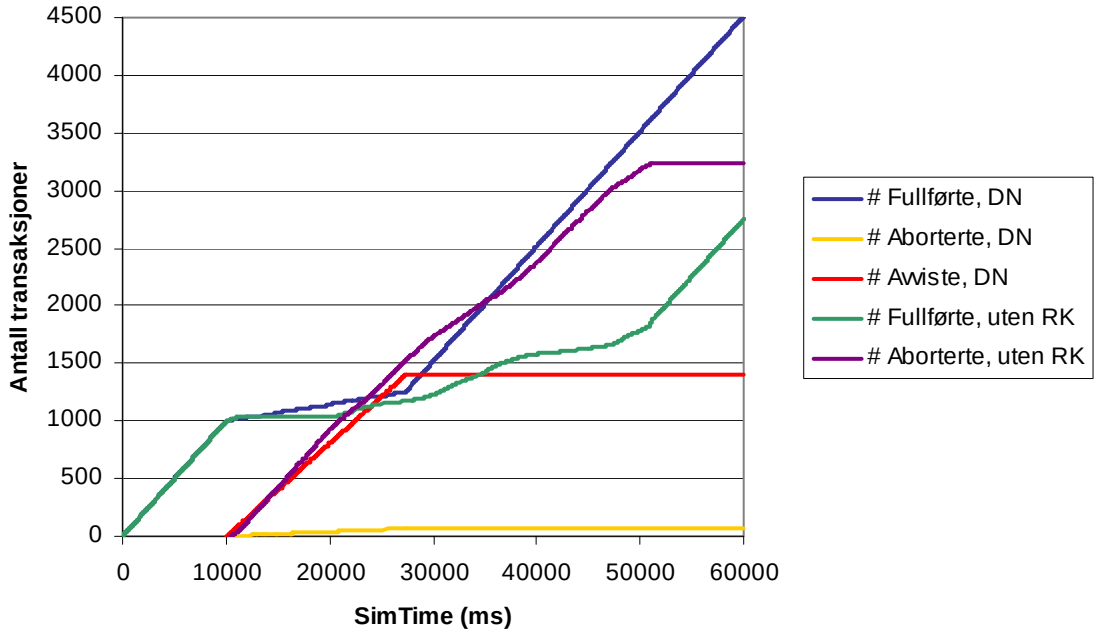


Figur 8.4: Simuleringsgraf: Refragmentering, 40 ms, 0.2, Transaksjonstider DN

I figur 8.5 har vi sammenlignet distribuert nodetabell med systemet uten ressurskontroll, hvor vi ser på antall aborterte, fullførte og avviste transaksjoner. Vi ser en lik utvikling på de to systemene fram til refragmenteringen starter etter 10 000 ms. Nå starter systemet med distribuert nodetabell å avvise og abortere transaksjoner, mens systemet uten ressurskontroll aborterer transaksjoner. Det interessante her er å se at systemet uten ressurskontroll aborterer flere transaksjoner enn systemet med distribuert nodetabell avviser og aborterer. Refragmenteringen fullfører langt tidligere for distribuert nodetabell (etter rundt 27 000 ms mot omtrent 51 000 ms), noe som gjør at ingen flere transaksjoner avvises eller aborteres, og antall fullførte øker igjen jevnt for distribuert nodetabell, mens systemet uten ressurskontroll får få fullførte transaksjoner helt til refragmenteringen er ferdig etter rundt 51 000 ms. Nettopp denne forskjellen i når refragmenteringen er ferdig er avgjørende for at ressurskontroll er hensiktsmessig for dette systemoppsettet.

Et siste viktig poeng er forskjellen i gjennomsnittlig transaksjonstid. Denne ser vi er mye høyere for systemet uten ressurskontroll (104,03 ms) i forhold til systemene med ressurskontroll (fra 29,34 ms og nedover).

Ut i fra denne drøftingen finner vi ressurskontroll nyttig ved dette systemoppsettet.



Figur 8.5: Simuleringsgraf: Refragmentering, 40 ms, 0.2, Uten RK mot DN

8.3.2 mtBetweenArrivals: 50 ms

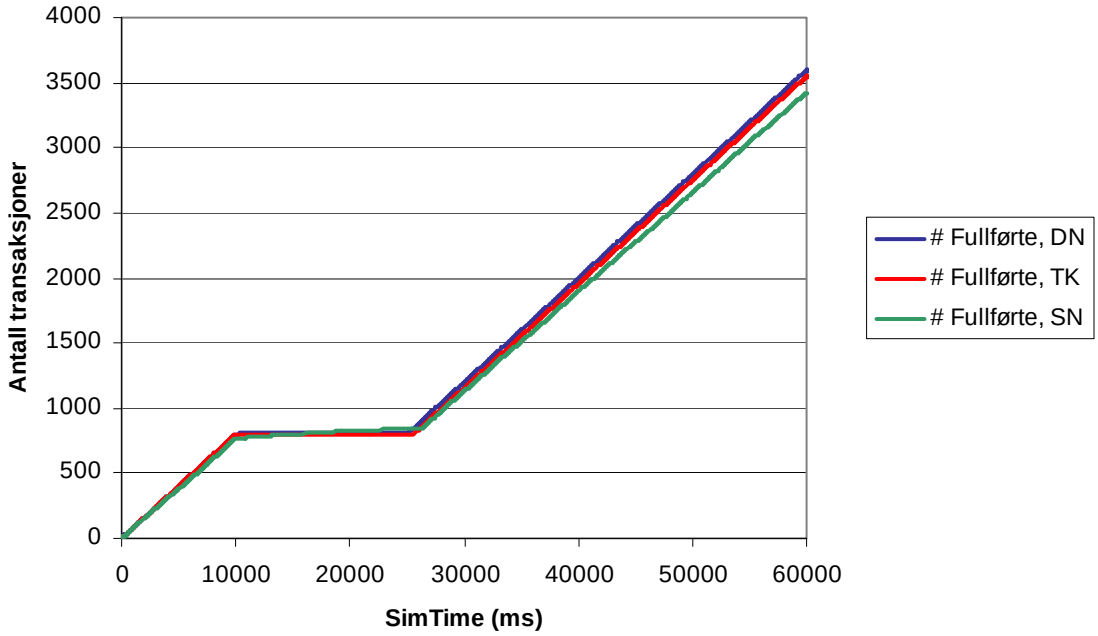
Vi vil her presentere resultatene for simuleringen med refragmentering med 50 millisekunders simuleringstid mellom hver ankomst av nye transaksjoner til systemet.

probPageFault: 0,0

I tabell 8.7 ser vi simuleringsresultatene fra simuleringen med en disksannsynlighet på 0,0.

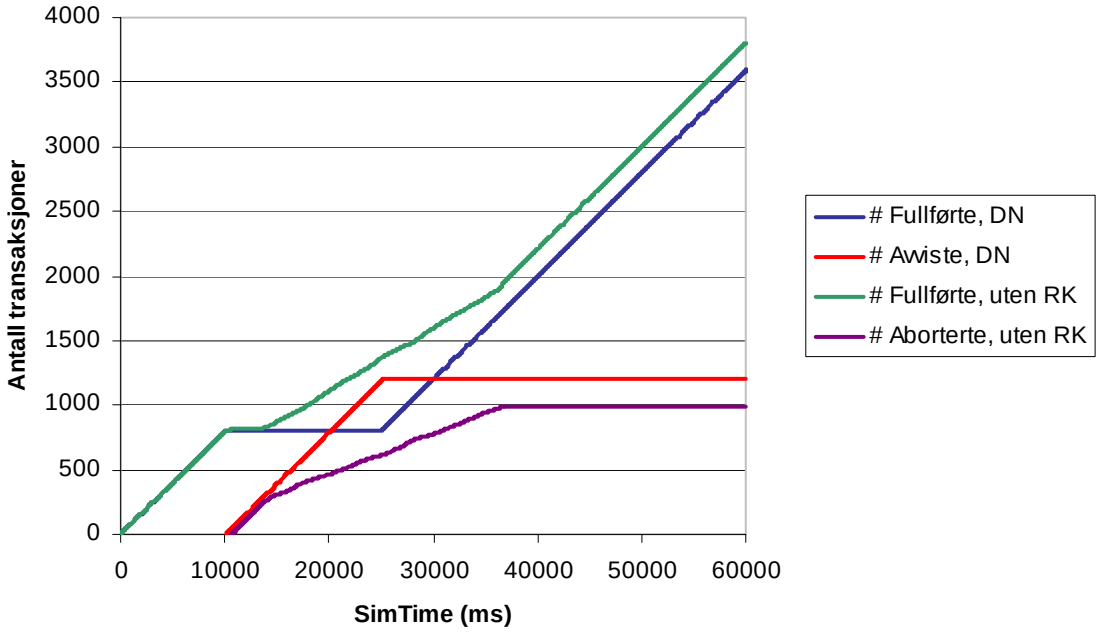
Tabell 8.7: Simulering: Refragmentering, Ankomst: 50 ms, P(Disk): 0.0

Resultat	Ingen rk	DN	TK	SN
Refragmentering ferdig (SimTime)	36664,79	25087,72	25548,83	26340,08
Gjennomsnittlig transaksjonstid (SimTime)	83,63	10,74	12,01	17,15
Antall avviste	0	1200	1244	1362
Antall aborterte	992	0	0	9
Antall fullførte	3804	3596	3550	3425
Antall transaksjonsmeldinger	208862	122584	120853	118555
Antall ressurskontrollmeldinger	0	4848	26228	11082



Figur 8.6: Simuleringsgraf: Refragmentering, 50 ms, 0.0, # fullførte ved RK

Av figur 8.6, som viser antall fullførte transaksjoner for hver av ressurskontrollvariantene, kommer det fram at de ulike ressurskontrollene fullfører omtrent like mange transaksjoner i dette systemet, og at fullføringene følger samme utvikling. Fra tabell 8.7 ser vi at distribuert nodetabell avslutter refragmenteringen tidligst, den har lavest gjennomsnittlig responstid og den fullfører flest transaksjoner. Vi ser også at distribuert nodetabell har færrest ressurskontrollmeldinger i denne simuleringen. Sentralisert nodetabell fullfører noen få transaksjoner gjennom refragmenteringsperioden, men siden den avviser flere transaksjoner enn de andre ressurskontrolltypene før og etter refragmenteringen er ikke denne av spesiell interesse for dette tilfellet. På bakgrunn av disse punktene velger vi å sammenligne distribuert nodetabell mot et system uten ressurskontroll.



Figur 8.7: Simuleringsgraf: Refragmentering, 50 ms, 0.0, Uten RK mot DN

Vi ser av figur 8.7 hvordan antall fullførte og avviste transaksjoner utvikler seg ved bruk av ressurskontroll og hvordan antall fullførte og aborterte transaksjoner uvikler seg i systemet uten ressurskontroll. Under refragmenteringsperioden gjør ikke ressurskontrollen annet enn å avvise transaksjoner. Vi ser at systemet som kjører uten ressurskontroll fullfører transaksjoner under hele simuleringsperioden, med et lite unntak i starten av refragmenteringsperioden. Mens refragmenteringen pågår og en tid etter den er ferdig, fullfører og aborterer dette systemet transaksjoner. Simuleringen ender med at antall fullføringer øker like raskt med og uten ressurskontroll.

Siden ressurskontrollen ikke tillater nye transaksjoner å starte mens refragmenteringen pågår, vil dette systemet være ledig til å fullføre nye transaksjoner etter at refragmenteringen er fullført. I figur 8.7 ser vi dette ved at fra det tidspunktet der systemet med distribuert nodetabell er ferdig med refragmenteringen (etter omlag 25 000 ms) og til det tidspunktet der systemet uten ressurskontroll er ferdig med refragmenteringen (etter omlag 37 000 ms), vil systemet som benytter ressurskontroll fullføre transaksjoner raskere enn systemet uten ressurskontroll. Systemet som ikke har ressurskontroll vil være veldig belastet gjennom refragmenteringsperioden, slik at det tar lengre tid før dette systemet fullfører refragmenteringen og kan bruke alle ressursene på transaksjonsprosessering. Dette betyr at uten ressurskontroll vil antall fullførte transaksjoner igjen øke like raskt som før refragmenteringen startet, senere i simuleringstiden enn for distribuert nodetabell.

Av tabell 8.7 kommer det fram at systemet uten ressurskontroll har omtrent åtte ganger lengre responstid, noe som begrunnes med at systemet fullfører transaksjoner under refragmenteringsperioden. I denne peroiden er den gjennomsnittlige responstiden lengre. Dette er forklart spesielt under kapittel 8.3.1. For systemet uten ressurskontroll blir også refragmenteringsperioden lengre. Dette forklares med at systemet både fullfører og aborterer transaksjoner samtidig som det utfører refragmentering, og at det er en langt større kostnad å abortere transaksjoner enn å avvise dem.

Det er i utgangspunktet ikke positivt å stenge nye transaksjoner helt ute under refragmenteringen, slik ressurskontrollen gjør i dette tilfellet. Systemet uten ressurskontroll fullfører flere transaksjoner i denne simuleringen. I tillegg aborteres færre transaksjoner enn de som avvises av

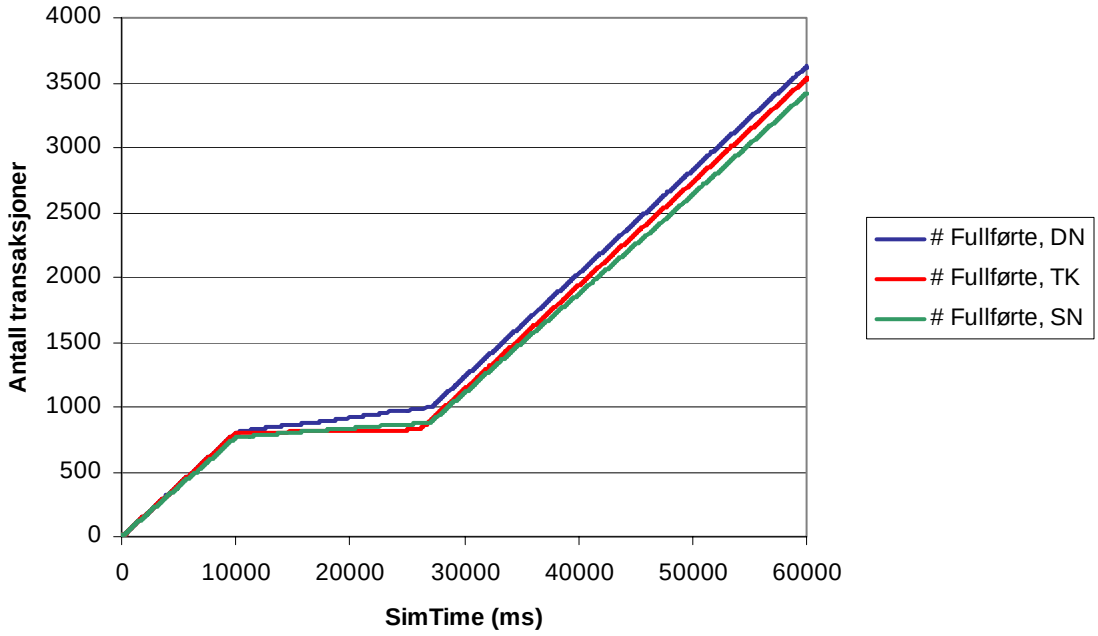
ressurskontrollen. Systemet uten ressurskontroll har 63,9 % flere meldinger er enn systemet med ressurskontroll. Til tross for en slik økning i antall meldinger har ikke systemet mer enn 5,8 % flere fullførte transaksjoner. Grunnen til at økningen i antall meldinger er større enn økningen i antall fullførte transaksjoner, kommer av at systemet uten refragmentering aborterer transaksjoner. At abortering er mer kostbart enn avvisning ved at det genereres flere meldinger, er nærmere beskrevet i kapittel 8.3.1 under systemet med disksannsynlighet på 0,0. Slik sett ser vi at begge systemene har både positive og negative sider.

probPageFault: 0,2

I tabell 8.8 ser vi simuleringsresultatene fra simuleringen med en disksannsynlighet på 0,2.

Tabell 8.8: Simulering: Refragmentering, Ankomst: 50 ms, P(Disk): 0.2

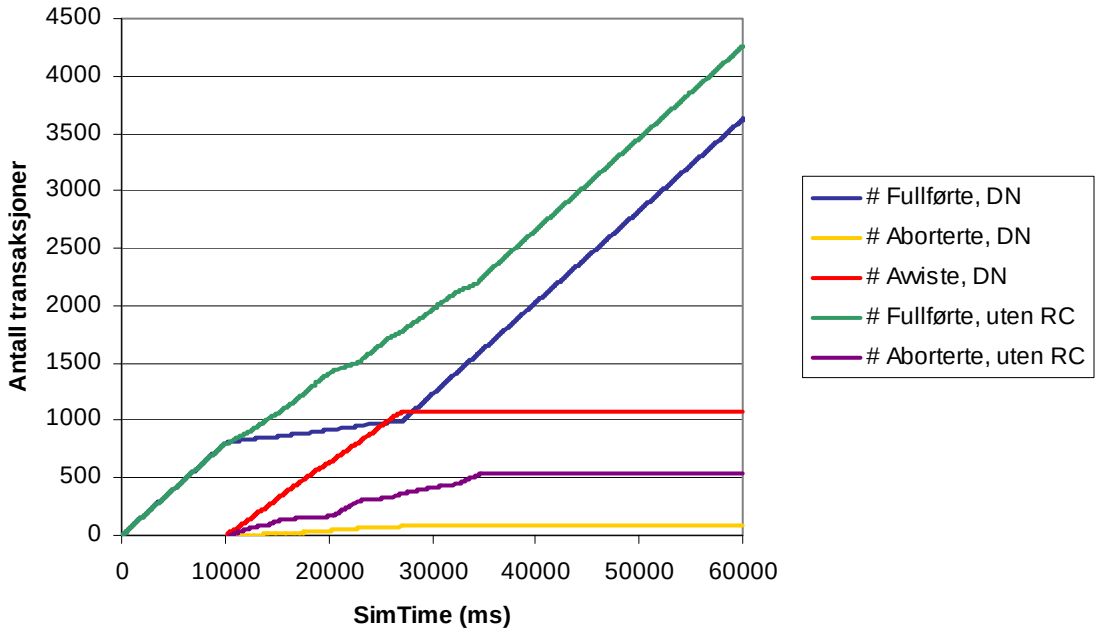
Resultat	Ingen rk	DN	TK	SN
Refragmentering ferdig (SimTime)	34560,02	27128,46	26092,09	27076,92
Gjennomsnittlig transaksjonstid (SimTime)	93,92	29,67	16,86	21,47
Antall avviste	0	1085	1263	1367
Antall aborterte	538	85	1	11
Antall fullførte	4258	3624	3532	3418
Antall transaksjonsmeldinger	208266	139042	126057	124799
Antall ressurskontrollmeldinger	0	2176	26272	9768



Figur 8.8: Simuleringsgraf: Refragmentering, 50 ms, 0.2, # fullførte ved RK

I figur 8.8 ser vi at distribuert nodetabell fullfører flest transaksjoner av ressurskontroll-alternativene i dette systemet. Vi ser at ressurskontrollvarianten også fullfører transaksjoner under refragmenteringsperioden. I tabell 8.8 kommer det fram at den gjennomsnittlige responstiden er høyest for distribuert nodetabell, og dette kommer av at den fullfører flest transaksjoner av ressurskontrolltypene under refragmenteringsperioden (se kapittel 8.3.1). Det som er negativt med distribuert nodetabell i dette tilfellet er at den aborterer 85 transaksjoner. Likevel blir summen av antall avviste og aborterte transaksjoner lavere for distribuert nodetabell, enn tilsvarende sum for

de andre ressurskontrollene. Vi velger å sammenligne distribuert nodetabell mot systemet uten ressurskontroll.



Figur 8.9: Simuleringsgraf: Refragmentering, 50 ms, 0.2, Uten RK mot DN

Fordelingen av antall fullførte, aborterte og avviste transaksjoner som funksjon av tid for systemene med og uten ressurskontroll, kan sees i figur 8.9. Av figuren kommer det fram at systemet uten ressurskontroll fullfører transaksjoner med tilnærmet lik rate gjennom hele simuleringen, mens systemet med ressurskontroll flater ut i perioden som refragmenteringen pågår. Som for alle systemene vi har simulert, vil refragmenteringsperioden bli kortere hvis man benytter ressurskontroll, siden dette systemet nesten utelukkende utfører refragmentering mens refragmenteringstransaksjonen er aktiv. Systemet uten ressurskontroll fullfører og aborterer transaksjoner under refragmenteringsperioden, og dermed blir refragmenteringsperioden lengre.

Etter at systemet med ressurskontroll er ferdig med refragmenteringen (etter omlag 27 000 ms) og fram til tidspunktet der systemet uten ressurskontroll er ferdig med refragmenteringen (etter omlag 34 500 ms) fullfører systemet med ressurskontroll transaksjoner raskere enn systemet uten ressurskontroll. Dette kommer av at systemet med ressurskontroll nå bruker alle ressursene på transaksjonsprosessering. Hvis vi sammenligner dette fenomentet i figur 8.7 og figur 8.9, ser vi at forskjellen er størst i figur 8.7. Grunnen til dette kan være at i det siste tilfellet vil refragmenteringen få diskavbrudd, og dermed vil systemet få mulighet til å utføre andre operasjoner mens refragmenteringen venter på svar fra disken. Dermed vil systemene være mindre belastet gjennom refragmenteringsperioden.

Hvis vi sammenligner responstiden for de to systemene, ser vi at distribuert nodetabell gjennomsnittlig har omtrent 1/3 så lang responstid. Dette bygger på at systemet uten ressurskontroll fullfører flere transaksjoner i refragmenteringsperioden, og at gjennomsnittlig responstid er høyere her (se kapittel 8.3.1).

Vi mener at ressurskontrollen stenger ute for mange nye transaksjoner under refragmenteringsperioden, men ved å innføre ressurskontroll for dette tilfellet vil systemet bli belastet med færre abortprosesseringer. Dette gir seg utslag i antall meldinger som systemet opplever. I systemet uten ressurskontroll har vi 47,5 % flere meldinger enn i systemet med ressurskontroll, men kun 17,5 % flere fullførte transaksjoner. Grunnen til dette er nærmere beskrevet i kapittel 8.3.1 under systemet med disksannsynlighet på 0,0. Systemet uten ressurskontroll fullfører flest transaksjoner og antall aborterte transaksjoner er halvparten av det

antallet som ressurskontrollen aborterer og avviser. På bakgrunn av drøftingen vi har gjort her, kan vi konkludere med at ressurskontroll har positive og negative effekter på dette systemoppsettet. Vi finner likevel at systemet uten ressurskontroll er best, siden dette systemet klarer å fullføre langt flere transaksjoner.

8.3.3 mtBetweenArrivals: 60 ms

Vi vil her presentere resultatene for simuleringen med refragmentering med 60 millisekunders simuleringstid mellom hver ankomst av nye transaksjoner til systemet.

probPageFault: 0,0

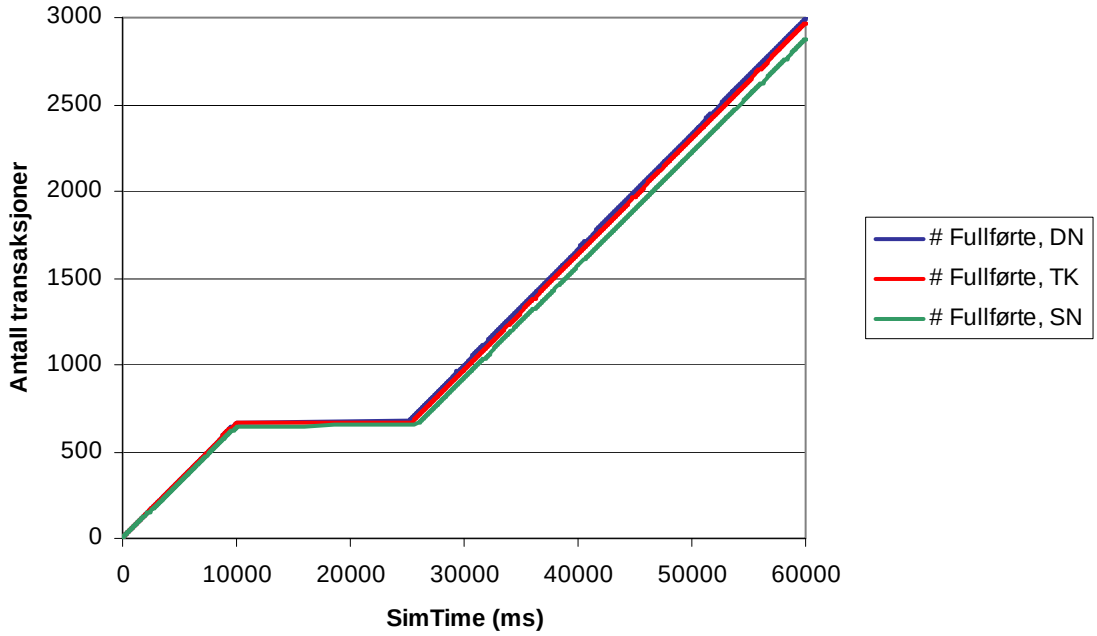
I tabell 8.9 ser vi simuleringsresultatene fra simuleringen med en disksannsynlighet på 0,0.

Tabell 8.9: Simulering: Refragmentering, Ankomst: 60 ms, P(Disk): 0.0

Resultat	Ingen rk	DN	TK	SN
Refragmentering ferdig (SimTime)	32300,18	25063,98	25456,34	25997,02
Gjennomsnittlig transaksjonstid (SimTime)	81,96	10,5	12,04	13,59
Antall avviste	0	1000	1028	1089
Antall aborterte	411	0	0	34
Antall fullførte	3585	2996	2968	2873
Antall transaksjonsmeldinger	164144	103166	101887	100419
Antall ressurskontrollmeldinger	0	3976	21874	9260

For dette systemoppsettet er distribuert nodetabell den beste løsningen av ressurskontroll-alternativene. Dette ser vi ut i fra tabell 8.9 hvor denne løsningen er først ferdig med refragmenteringen, har best snitt på transaksjonstidene, færrest aborterte og avviste transaksjoner, flest fullførte transaksjoner og færrest meldinger. Løsningen har riktignok flere transaksjonsmeldinger enn de to andre ressurskontroll-alternativene, men dette henger sammen med økningen i antall fullførte transaksjoner.

Den samme konklusjonen kan vi trekke ut i fra figur 8.10. Her ser vi at de tre ressurskontroll-typene har omtrent lik utvikling i antall fullførte transaksjoner. Likevel kan vi se at distribuert nodetabell får noe bedre utvikling i antall fullførte transaksjoner etter at refragmenteringsperioden er ferdig.

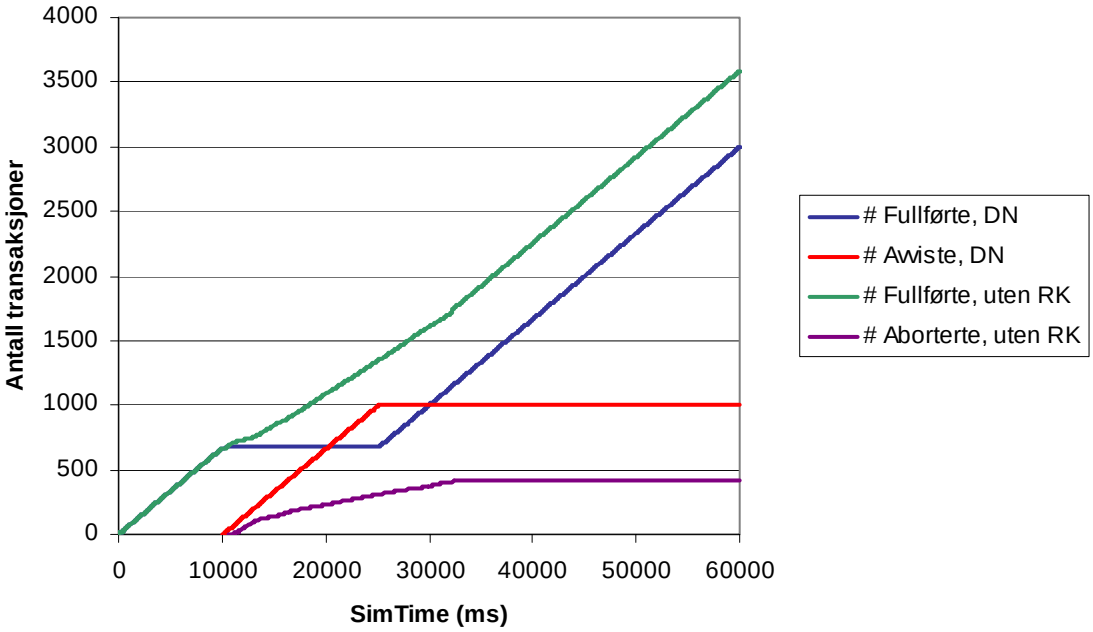


Figur 8.10: Simuleringsgraf: Refragmentering, 60 ms, 0.0, # fullførte ved RK

I figur 8.11 har vi sammenlignet distribuert nodetabell mot systemet uten ressurskontroll. Som for de andre systemene ser vi en lik utvikling fram til refragmenteringen starter. Deretter ser vi at begge systemene får en knekk i antall fullførte transaksjoner og at distribuert nodetabell avviser transaksjoner, mens systemet uten ressurskontroll aborterer transaksjoner. Nå avvises det langt flere transaksjoner ved distribuert nodetabell enn det aborteres uten ressurskontroll. Vi ser at systemet med distribuert nodetabell stopper fullføringen av transaksjoner under refragmenteringen. Det positive med distribuert nodetabell her er at den fullfører refragmenteringen tidligere enn systemet uten ressurskontroll (etter 25 000 ms mot 32 000 ms) og følgelig gjør systemet mindre belastet, raskere.

En annen positiv side med ressurskontroll-løsningene her sammenlignet med systemet uten ressurskontroll er den gjennomsnittlige transaksjonstiden som er mye lavere. Her har vi et snitt på 81,96 ms for systemet uten ressurskontroll, mens snittet er på 10,50 ms med distribuert nodetabell. Den samme trenden ser vi for antall meldinger, hvor det er 52,7 % flere meldinger for systemet uten ressurskontroll. Det skyldes delvis flere fullførte transaksjoner, men her er forskjellen på kun 19,7 %.

Vi ser her et systemoppsett hvor innføringen av ressurskontroll har både positive og negative effekter.



Figur 8.11: Simuleringsgraf: Refragmentering, 60 ms, 0.0, Uten RK mot DN

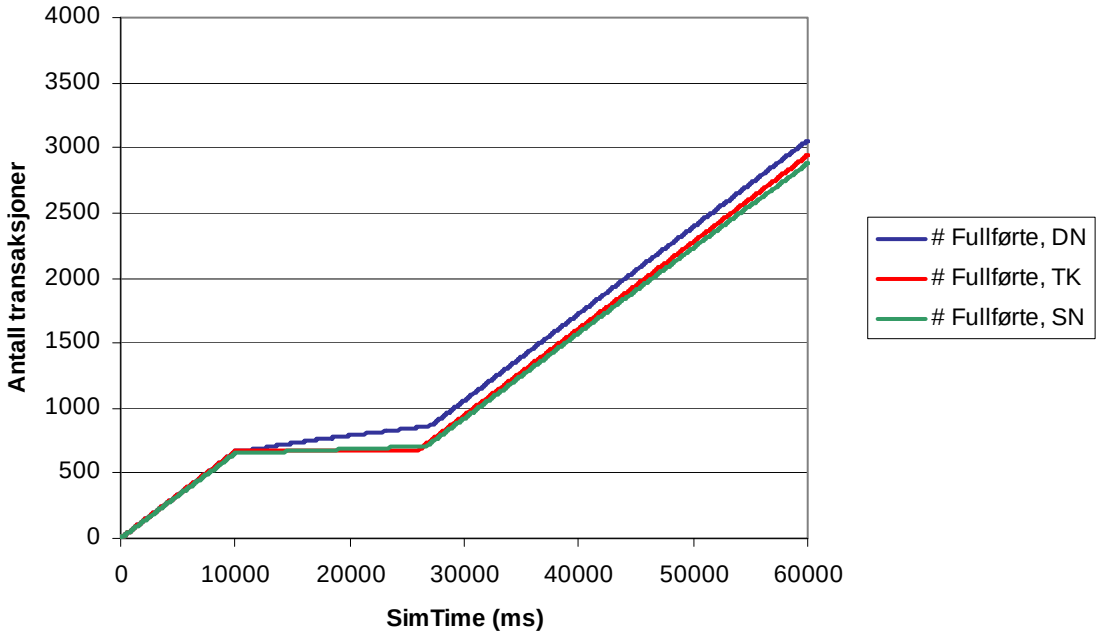
probPageFault: 0,2

I tabell 8.10 ser vi simuleringsresultatene fra simuleringen med en disksannsynlighet på 0,2.

Tabell 8.10: Simulering: Refragmentering, Ankomst: 60ms, P(Disk): 0.2

Resultat	Ingen rk	DN	TK	SN
Refragmentering ferdig (SimTime)	31816,97	27119,29	25959,85	26723,15
Gjennomsnittlig transaksjonstid (SimTime)	88,80	30,68	16,52	19,24
Antall avviste	0	866	1055	1080
Antall aborterte	128	74	0	37
Antall fullførte	3868	3056	2941	2879
Antall transaksjonsmeldinger	169037	117607	106464	106683
Antall ressurskontrollmeldinger	0	2344	21934	7934

Vi ser av tallene at distribuert nodetabell gir flest fullførte transaksjoner og færrest aborterte og avviste transaksjoner og færrest meldinger. Slik sett er dette den beste løsningen. Ser vi derimot på når refragmenteringen fullføres og den gjennomsnittlige transaksjonstiden, kommer transaksjonsklareringen mye bedre ut. Her er også sentralisert nodetabell en bedre løsning. Vi er igjen i en situasjon hvor flere løsninger kan være hensiktsmessige. Vi velger distribuert nodetabell som den beste løsningen av ressurskontroll-alternativene ut i fra grafene presentert i figur 8.12. Vi ser her at distribuert nodetabell har en bedre utvikling enn de to andre under refragmenteringen. Dette gjør at flere transaksjoner fullfører, på bekostning av den gjennomsnittlige transaksjonstiden.

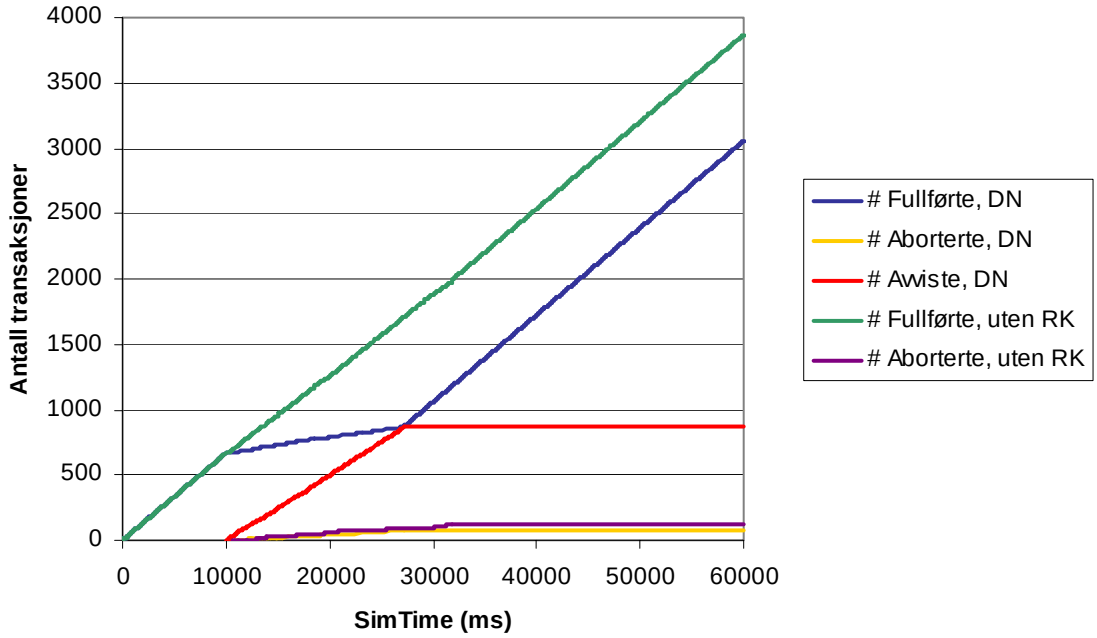


Figur 8.12: Simuleringsgraf: Refragmentering, 60 ms, 0.2, # fullførte ved RK

En sammenligning av distribuert nodetabell med et system uten ressurskontroll er å finne i figur 8.13. Vi ser igjen en lik utvikling til refragmenteringen starter. Deretter øker antall avviste transaksjoner for distribuert nodetabell langt raskere enn antall aborterte transaksjoner øker for systemet uten ressurskontroll. Selv om refragmenteringen fullfører tidligere for distribuert nodetabell, blir det mange flere fullførte transaksjoner for systemet uten ressurskontroll. Vi ser dermed at ressurskontrollen har en negativ effekt på antall fullførte transaksjoner.

Ser vi derimot på tallene for gjennomsnittlig transaksjonstid er denne på 88,80 ms for systemet uten ressurskontroll og på 30,30 ms for distribuert nodetabell. Tilsvarende ser vi at det er 42 % flere meldinger for systemet uten ressurskontroll, mens det er bare 26,6 % flere fullførte transaksjoner. Dette viser at systemet uten ressurskontroll gir flere meldinger per fullført transaksjon og dermed mer belastning på systemet.

Vi kan oppsummere med at dette systemoppsettet har både positive og negative effekter når vi innfører ressurskontroll.



Figur 8.13: Simuleringsgraf: Refragmentering, 60 ms, 0.2, Uten RK mot DN

8.4 Simuleringsresultater - uten refragmentering

I dette underkapitlet vil vi presentere simuleringsresultatene uten refragmentering. Denne simuleringen er utført i 20 sekunder i simuleringstid (`SimTime`).

8.4.1 mtBetweenArrivals: 15 ms

Vi vil her presentere resultatene for simuleringen med refragmentering med 15 millisekunders simuleringstid mellom hver ankomst av nye transaksjoner til systemet.

probPageFault: 0,0

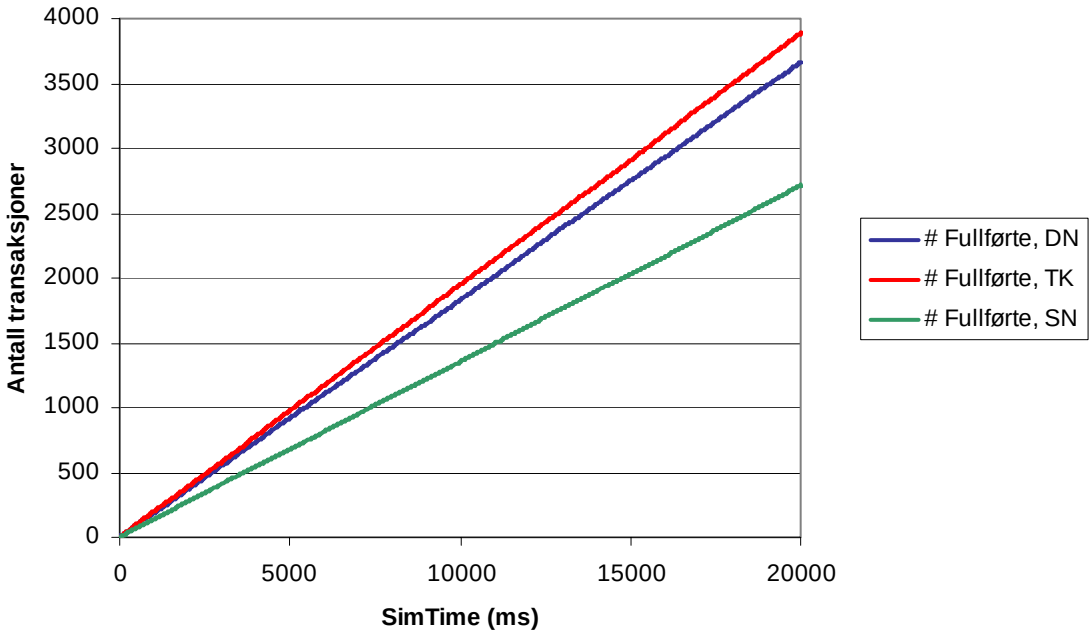
I tabell 8.11 ser vi simuleringsresultatene fra simuleringen med en disksannsynlighet på 0,0.

Tabell 8.11: Simulering: Uten refragmentering, Ankomst: 15 ms, P(Disk): 0.0

Resultat	Ingen rk	DN	TK	SN
Gjennomsnittlig transaksjonstid (SimTime)	9,35	10,12	11,90	11,89
Antall avviste	0	1646	1420	2596
Antall aborterte	0	0	0	0
Antall fullførte	5310	3666	3888	2716
Antall transaksjonsmeldinger	171340	119176	126351	89996
Antall ressurskontrollmeldinger	0	11064	29054	10897

Her har vi en interessant situasjon hvor distribuert nodetabell har lavest gjennomsnitt på transaksjonstidene, mens transaksjonsklarering og sentralisert nodetabell har omtrent lik tid. Antall fullførte og avviste transaksjoner varierer derimot mer mellom de ulike alternativene for ressurskontroll. Her kommer transaksjonsklarering best ut, med flest fullførte og færrest avviste transaksjoner. Denne løsningen gir derimot flere meldinger, slik at vi nå ser en situasjon hvor det er vanskelig å si hvilken løsning som er mest hensiktsmessig.

Ved å studere figur 8.14 ser vi utviklingen i antall fullførte transaksjoner, og vi ser at distribuert nodetabell og transaksjonsklarering er klart bedre enn sentralisert nodetabell.

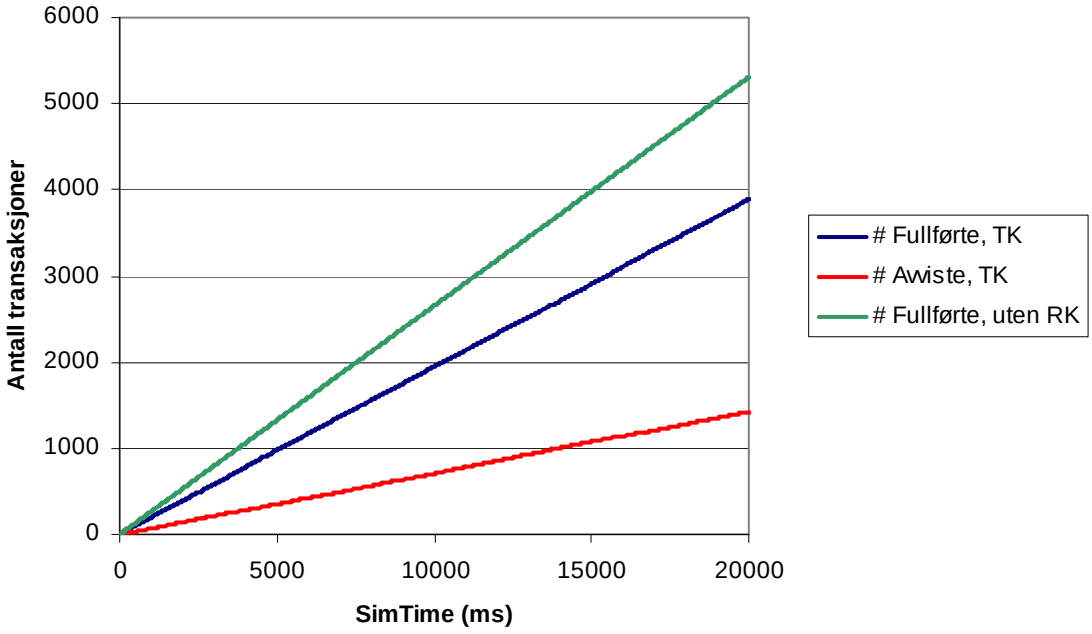


Figur 8.14: Simuleringsgraf: Uten refragmentering, 15 ms, 0.0, # fullførte ved RK

I figur 8.15 har vi sammenlignet systemet uten ressurskontroll med transaksjonsklarering. Vi ser her at systemet uten ressurskontroll gir klart best utvikling i antall fullførte transaksjoner og at det slik sett er å foretrekke.

Ser vi videre på tallene fra tabell 8.11, ser vi at systemet uten ressurskontroll også har lavest gjennomsnitt på transaksjonstiden (9,35 ms mot 11,90 ms ms for transaksjonsklarering). Det samme ser vi ved å studere antall meldinger i forhold til antall fullførte transaksjoner: Systemet uten ressurskontroll har 10,2 % flere meldinger og 36 6 % flere fullførte transaksjoner enn transaksjonsklarering.

Vi kan her konkludere med at det ikke vil lønne seg å innføre ressurskontroll på et slik systemoppsett.



Figur 8.15: Simuleringsgraf: Uten refragmentering, 15 ms, 0.0, Uten RK mot TK

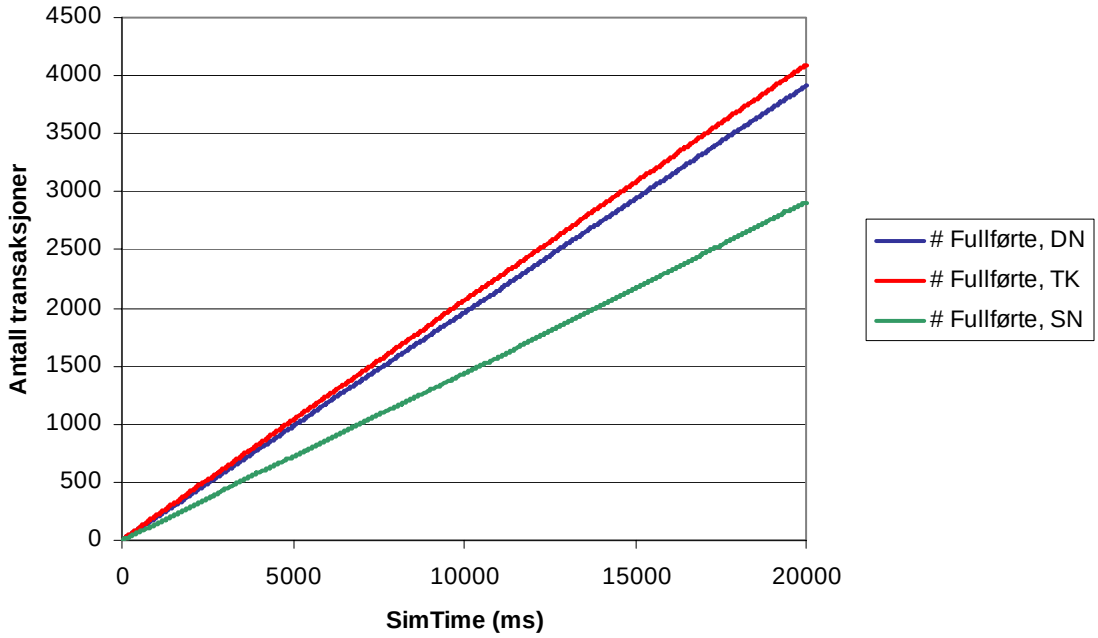
probPageFault: 0,2

I tabell 8.12 ser vi simuleringsresultatene fra simuleringen med en disksannsynlighet på 0,2.

Tabell 8.12: Simulering: Uten refragmentering, Ankomst: 15 ms, P(Disk): 0.2

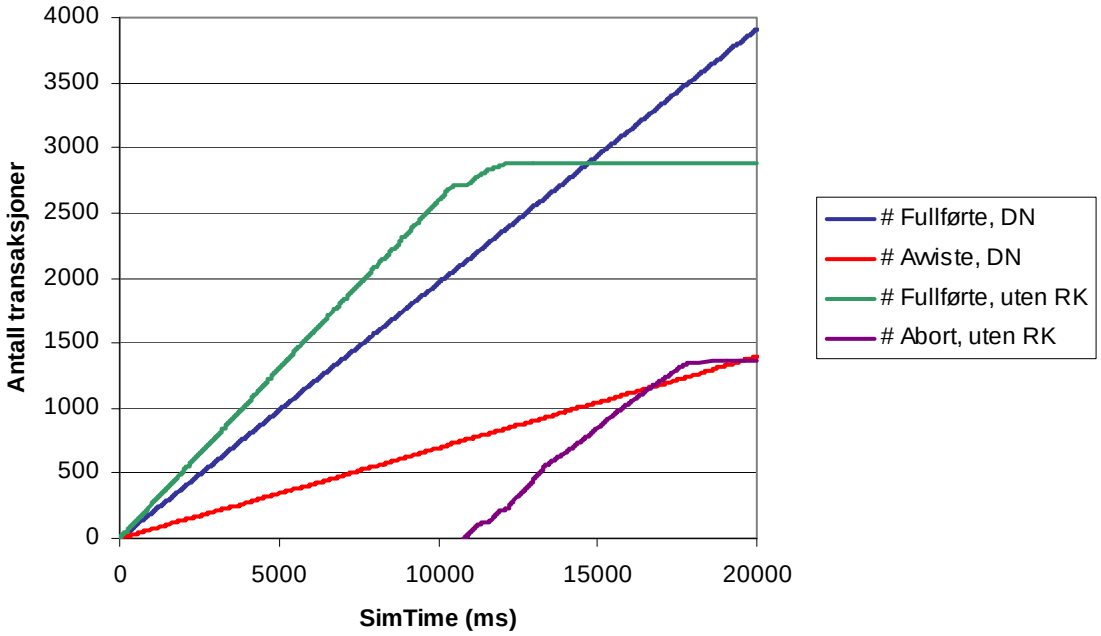
Resultat	Ingen rk	DN	TK	SN
Gjennomsnittlig transaksjonstid (SimTime)	138,21	19,61	24,29	16,60
Antall avviste	0	1396	1214	2399
Antall aborterte	1360	0	0	0
Antall fullførte	2883	3913	4096	2911
Antall transaksjonsmeldinger	155427	132546	139402	100098
Antall ressurskontrollmeldinger	0	10296	29004	10204

Her har vi igjen en situasjon hvor det ikke utpeker seg et klart alternativ som er å foretrekke for ressurskontroll. Sentralisert nodetabell har best gjennomsnittlig transaksjonstid, transaksjonsklarering har flest fullførte transaksjoner og færrest avviste, mens sentralisert nodetabell har færrest meldinger. Distribuert nodetabell ligger nærmest den beste løsningen ved alle parametere, og vi finner derfor den som den totalt beste løsningen slik vi vurderer det. Vi har illustrert utviklingen i antall fullførte transaksjoner grafisk, slik vi viser i figur 8.16. Her ser vi hvordan distribuert nodetabell og transaksjonsklarering følges jevnt, mens sentralisert nodetabell har langt færre fullførte transaksjoner. Dette er også grunnen til at denne løsningen har lavest gjennomsnitlig transaksjonstid, da den avviser flere transaksjoner som ville hevet dette snittet.



Figur 8.16: Simuleringsgraf: Uten refragmentering, 15 ms, 0.2, # fullførte ved RK

I figur 8.17 ser vi en grafisk sammenligning av distribuert nodetabell og systemet uten ressurskontroll. Vi ser her at distribuert nodetabell har en lineær utvikling i antall avviste transaksjoner og i antall fullførte transaksjoner. Systemet uten ressurskontroll har en bedre utvikling enn distribuert nodetabell i starten fram til rundt 10 000 ms av simuleringen. Dette skyldes at distribuert nodetabell avviser transaksjoner. Etter dette punktet ved 10 000 ms når systemet uten ressurskontroll et metningspunkt og systemet starter å abortere transaksjoner. Etter rundt 12 000 ms stopper dette systemet å fullførte transaksjoner og alle transaksjoner aborterer. Vi ser likevel at antall aborterte også avtar. Dette skyldes at systemet har blitt så overbelastet at det ikke klarer å få unna transkasjonene, selv om de aborterer. Grunnen til dette er igjen at abortprosessering krever ressurser i motsetning til å avvise transaksjoner. Denne systemkolapsen kan forklares videre ut i fra køteori slik vi har forklart i vedlegg E, hvor vi nå har fått en høyere ankomstrate enn behandlingsrate, og følgelig et system som får uendelig lange køer. Dette gjør at systemer som dette til slutt kollapser og får ikke gjort unna arbeidet i tide. Dette ser vi også rent konkret på vår simulering, hvor meldingskøen på en node er oppe i 27024 meldinger ved simuleringens slutt for systemet uten ressurskontroll, og dette antallet er fortsatt eksponentielt økende. Ressurskontrollsystemene klarer å holde dette antallet stabilt ved å avvise transaksjoner, og ender på eksempelvis 19 meldinger i køen til en node ved distribuert nodetabell.



Figur 8.17: Simuleringsgraf: Uten refragmentering, 15 ms, 0.2, Uten RK mot DN

Vi ser også at ressurskontroll er hensiktsmessig med henblikk på de andre parameterne for dette systemoppsettet ved å studere tabell 8.12, slik at vi her har en situasjon hvor ressurskontroll er positivt og faktisk helt nødvendig for at systemet skal klare å jobbe videre.

probPageFault: 0,4

Denne simuleringen gir så tungt belastet system uten ressurskontroll at simuleringen ikke klarer å fullføre og vi får dermed ikke generert de rapporter og filer som kreves for å analysere simuleringen. Det vi ser er altså et enda mer ekstremt system enn de vi beskrev ved forrige simulering med `probPageFault = 0,2`. Derfor vil de samme konklusjonene som ble gjort der også gjelde dette systemet. Forskjellen er at metningspunktet kommer enda tidligere og køene blir veldig store enda tidligere, slik at simulatoren stopper helt opp før vi rekker å fullføre simuleringen.

8.4.2 mtBetweenArrivals: 20 ms

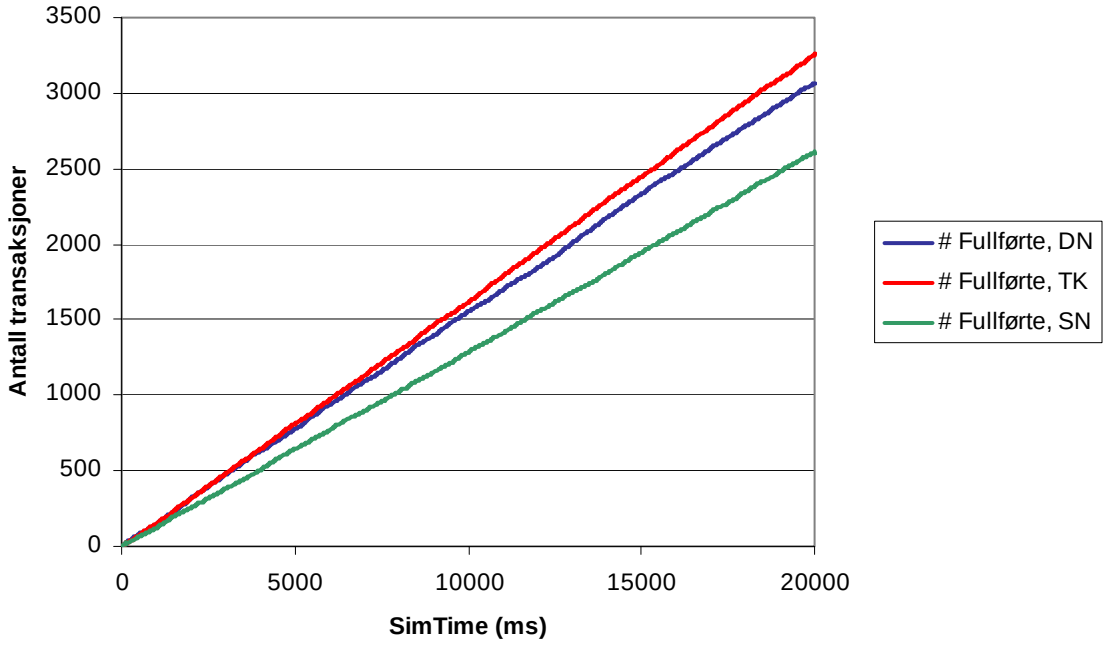
Vi vil her presentere resultatene for simuleringen uten refragmentering med 20 millisekunders simuleringstid mellom hver ankomst av nye transaksjoner til systemet.

probPageFault: 0,0

I tabell 8.13 ser vi simuleringsresultatene fra simuleringen med en disksannsynlighet på 0,0.

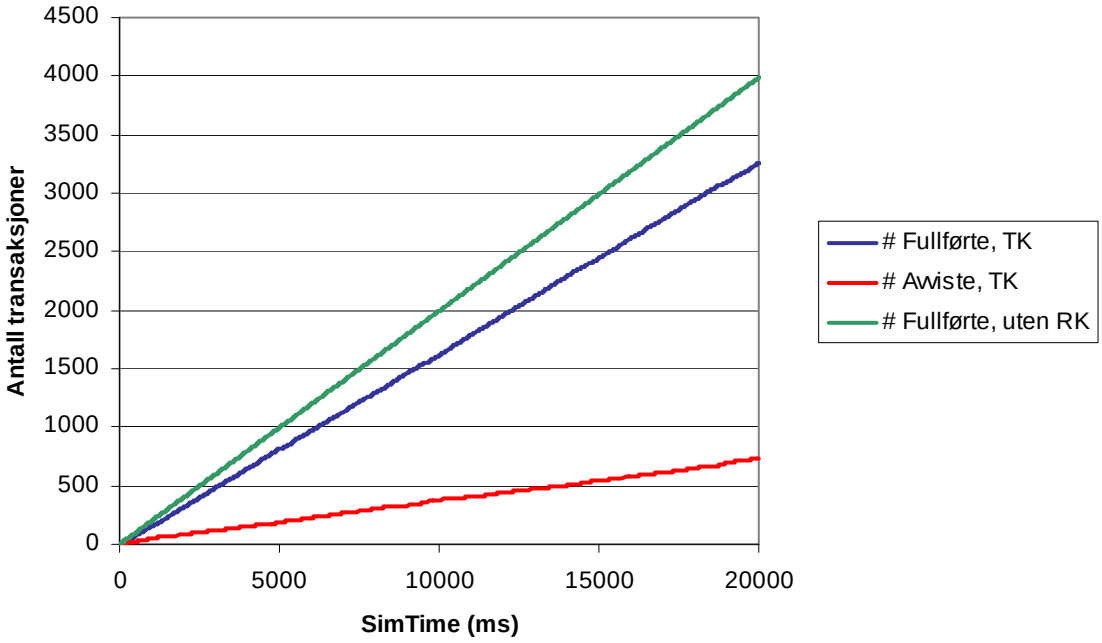
Tabell 8.13: Simulering: Uten refragmentering, Ankomst: 20 ms, P(Disk): 0.0

Resultat	Ingen rk	DN	TK	SN
Gjennomsnittlig transaksjonstid (SimTime)	9,38	9,88	11,64	11,46
Antall avviste	0	920	728	1379
Antall aborterte	0	0	0	0
Antall fullførte	3988	3068	3260	2609
Antall transaksjonsmeldinger	128930	99380	104434	84320
Antall ressurskontrollmeldinger	0	3640	21628	8338



Figur 8.18: Simuleringsgraf: Uten refragmentering, 20 ms, 0.0, # fullførte ved RK

I figur 8.18 ser vi at antall fullførte transaksjoner øker tilnærmet lineært for alle de tre ressurskontrolltypene. Ressurskontrolltypen transaksjonsklarering fullfører flest transaksjoner i denne simuleringen. Sammenligner vi responstidene i tabell 8.13, ser vi at distribuert nodetabell har lavest snitt for responstiden. Forskjellen i responstid mellom distribuert nodetabell og transaksjonsklarering er i dette tilfellet så liten, at vi ser på antall fullførte transaksjoner som den viktigste parameteren her. Dette gjør at vi ser på transaksjonsklarering som det beste alternativet i dette tilfellet. I figur 8.19 er transaksjonsklarering sammenlignet med et system uten ressurskontroll.



Figur 8.19: Simuleringsgraf: Uten refragmentering, 20 ms, 0.0, Uten RK mot TK

Figuren viser antall fullførte og avviste transaksjoner for kjøring med transaksjonsklarering, og den viser antall fullførte transaksjoner uten ressurskontroll. Av figuren kommer det fram at alle de tre størrelsene øker tilnærmet lineært. Antall fullførte transaksjoner uten bruk av ressurskontroll øker raskest. Grafen viser ingen tegn til at denne økningen skal flate ut. I tillegg er et viktig poeng at ingen transaksjoner aborteres uten ressurskontroll, mens transaksjoner blir avvist ved bruk av ressurskontroll. For dette tilfellet hvor responstiden er lavere uten bruk av ressurskontroll, virker ressurskontrollen litt strengt satt opp siden transaksjoner avvises uten at gjennomsnittlig responstid avtar. Siden transaksjonene i liten grad venter på ressurser i dette systemet, vil tiden det tar å utføre ressurskontroll før transaksjoner startes utgjøre den viktigste forskjellen i hvor lang tid transaksjonene bruker i de to systemene. På bakgrunn av dette er gjennomsnittlig responstid lavest uten bruk av ressurskontroll. Ser vi på forholdet mellom antall meldinger, er det 2,3 % flere meldinger for systemet uten ressurskontroll enn for alternativet med transaksjonsklarering. Systemet uten ressurskontroll har 23,8 % flere fullførte transaksjoner, slik at vi ser at antall meldinger per fullført transaksjon langt mindre for systemet uten ressurskontroll.

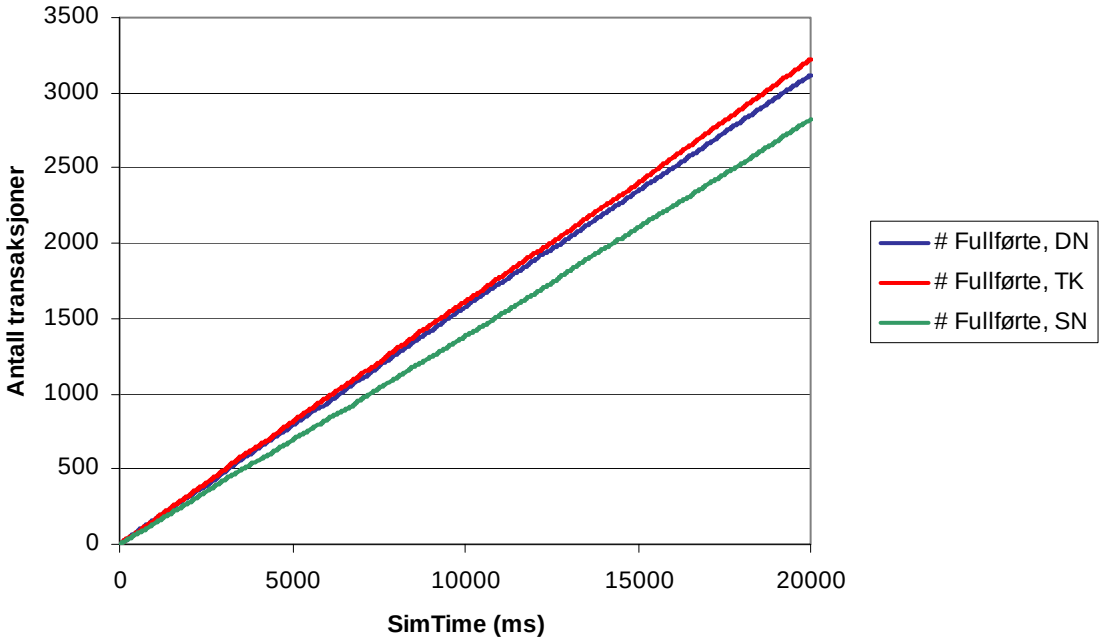
For dette systemet finner vi derfor løsningen uten ressurskontroll som den beste.

probPageFault: 0,2

I tabell 8.14 ser vi simuleringsresultatene fra simuleringen med en disksannsynlighet på 0,2.

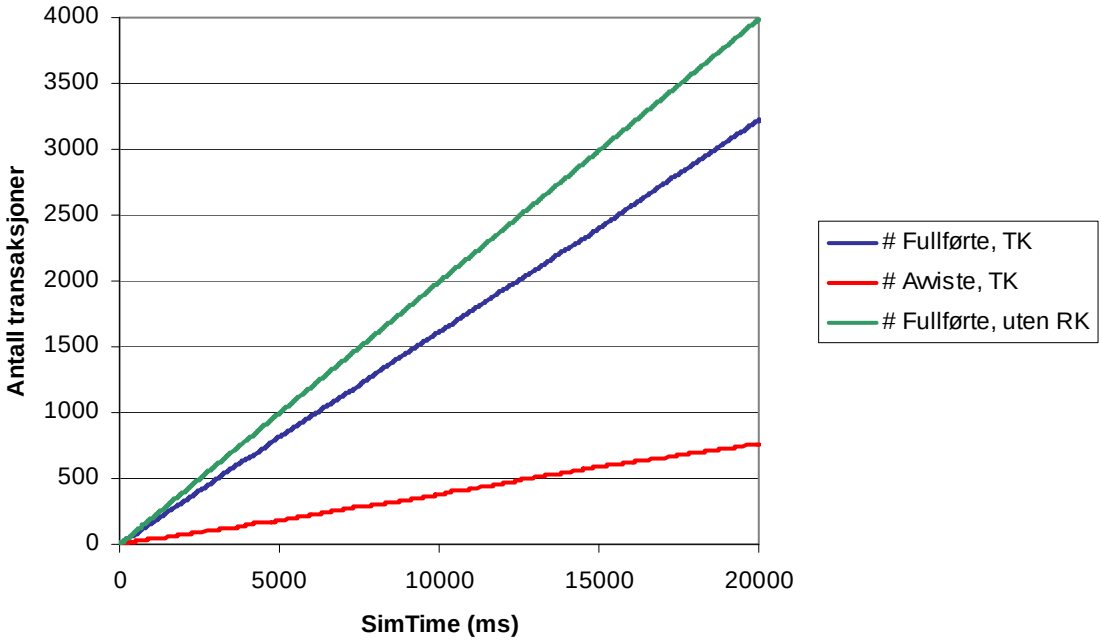
Tabell 8.14: Simulering: Uten refragmentering, Ankomst: 20 ms, P(Disk): 0.2

Resultat	Ingen rk	DN	TK	SN
Gjennomsnittlig transaksjonstid (SimTime)	17,79	16,75	18,31	16,75
Antall avviste	0	868	765	1164
Antall aborterte	0	0	0	0
Antall fullførte	3987	3119	3221	2824
Antall transaksjonsmeldinger	134855	105951	109512	95540
Antall ressurskontrollmeldinger	0	4520	21830	7859



Figur 8.20: Simuleringsgraf: Uten refragmentering, 20 ms, 0.2, # fullførte ved RK

Slik det kommer fram av figur 8.20 fullfører ressurskontrollene transaksjonsklarering og distribuert nodetabell flest transaksjoner, mens varianten sentralisert nodetabell skiller seg ut ved å fullføre litt færre. Antallet transaksjoner som fullføres øker tilnærmet lineært for alle de tre variantene. I tabell 8.14 kommer det fram at den gjennomsnittlige responstiden varierer med 1,6 ms fra distribuert nodetabell til transaksjonsklarering. Transaksjonsklareringen fullfører flere transaksjoner, uten at den gjennomsnittlige responstiden øker betraktelig. Sentralisert nodetabell er ikke interessant å se nærmere på, siden den avviser flest transaksjoner uten at gjennomsnittlig responstid forbedres. Vi velger derfor å sammenligne transaksjonsklarering mot et system uten ressurskontroll.



Figur 8.21: Simuleringsgraf: Uten refragmentering, 20 ms, 0.2, Uten RK mot TK

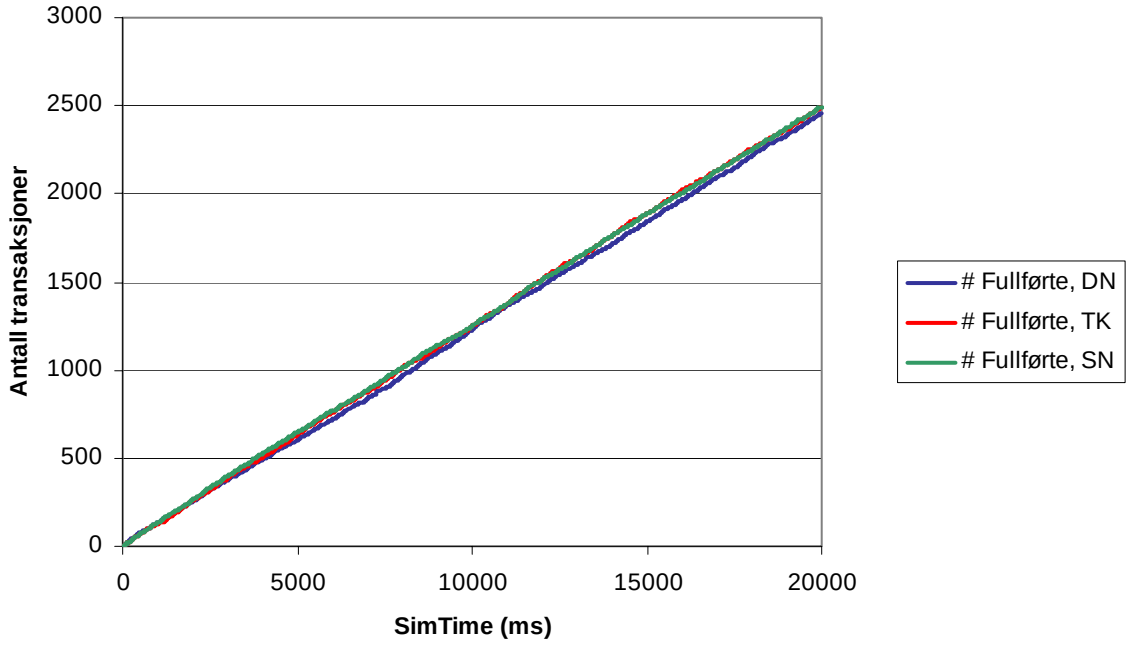
I figur 8.21 ser vi antall avviste og fullførte transaksjoner ved bruk av ressurskontroll, og antall fullførte transaksjoner uten bruk av ressurskontroll. Systemet uten ressurskontroll fullfører flest transaksjoner og antall fullførte transaksjoner øker raskest i dette systemet. I tillegg aborteres ingen transaksjoner i systemet uten ressurskontroll, mens flere transaksjoner avvises i systemet med ressurskontroll. Tatt i betraktning at systemet uten ressurskontroll har den laveste gjennomsnittlige responstiden, ser vi her på ressurskontrollen som i overkant streng, da den ikke gir noen gevinst på denne parameteren. At systemet uten ressurskontroll har lavest gjennomsnittlig responstid, kommer av at systemet ikke er overbelastet i tillegg til at man slipper ressurskontrollfasen i starten av alle transaksjoner. For dette systemet er løsningen uten ressurskontroll best.

probPageFault: 0,4

I tabell 8.15 ser vi simuleringsresultatene fra simuleringen med en disksannsynlighet på 0,4.

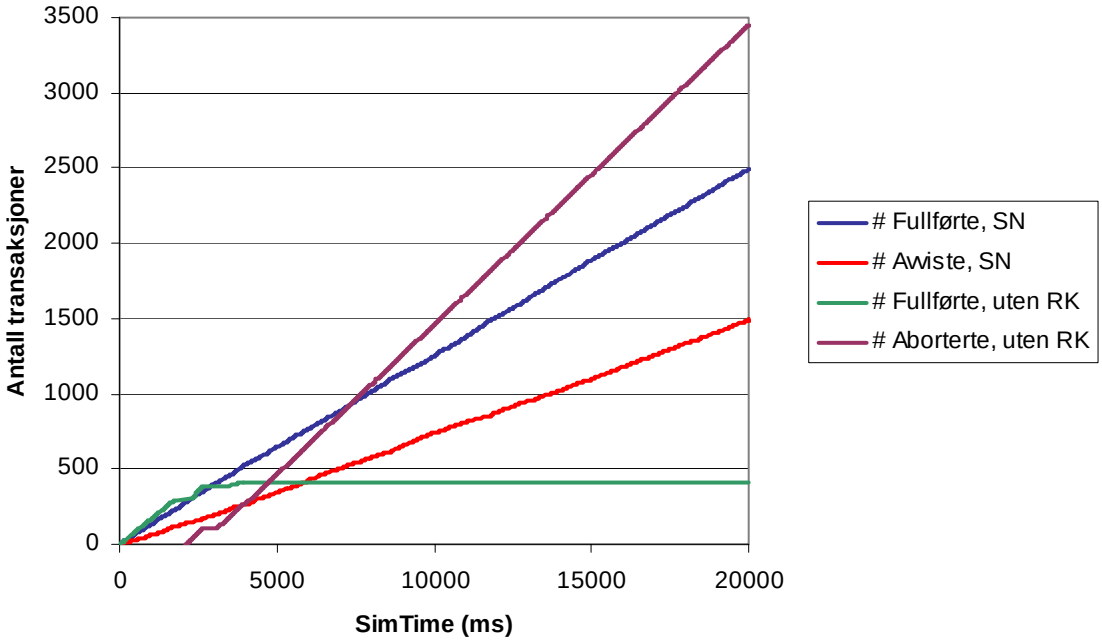
Tabell 8.15: Simulering: Uten refragmentering, Ankomst: 20 ms, P(Disk): 0.4

Resultat	Ingen rk	DN	TK	SN
Gjennomsnittlig transaksjonstid (SimTime)	181,06	61,58	60,91	59,53
Antall avviste	0	1530	1492	1492
Antall aborterte	3453	0	0	0
Antall fullførte	410	2452	2489	2493
Antall transaksjonsmeldinger	173082	87373	88773	88655
Antall ressurskontrollmeldinger	0	5540	21830	7474



Figur 8.22: Simuleringsgraf: Uten refragmentering, 20 ms, 0.4, # fullførte ved RK

For dette systemet viser figur 8.22 at det er liten forskjell mellom hvor mange transaksjoner systemer med de ulike ressurskontrollene fullfører. Det kommer også fram av figuren at utviklingen til alle ressurskontrollene er tilnærmet lineære. Ser vi nærmere på tabell 8.15 ser vi at sentralisert nodetabell fullfører flest transaksjoner og har lavest gjennomsnittlig responstid. Når både responstid og antall fullførte transaksjoner er så likt som i dette tilfellet og spesielt mellom sentralisert nodetebell og transaksjonsklarering, ser vi at sentralisert nodetabell har mange færre ressurskontrollmeldinger enn transaksjonsklareringen. For dette tilfellet er ressurskontrollene så å si like gode, men vi velger å sammenligne sentralisert nodetabell mot systemet uten ressurskontroll.



Figur 8.23: Simuleringsgraf: Uten refragmentering, 20 ms, 0.4, Uten RK mot SN

I figur 8.23 ser vi utviklingen i antall avviste og fullførte transaksjoner ved bruk av ressurskontroll, og antall aborterte og fullførte transaksjoner uten bruk av ressurskontroll. Denne figuren viser at systemet uten ressurskontroll slutter å fullføre transaksjoner, mens ved bruk av ressurskontroll øker antall fullføringer tilnærmet lineært. Fram til punktet der systemet uten ressurskontroll slutter å fullføre transaksjoner, øker antall fullføringer litt raskere uten ressurskontroll. Figuren viser også at antall avviste transaksjoner med ressurskontroll er lavere enn antall aborterte transaksjoner uten ressurskontroll. I tabell 8.15 kommer det fram at systemet uten ressurskontroll har behandlet flere transaksjonsmeldinger enn systemet med ressurskontroll. Dette kommer av at det å abortere transaksjoner er en belastning for systemet, mens avvisning er en billig løsning siden transaksjonene ikke startes i det hele tatt. Dette er omtalt i kapittel 8.3.1 under systemet med disksannsynlighet på 0,0. Vi ser også at ressurskontroll-alternativene er best når vi ser på parameteren gjennomsnittlig transaksjonstid, hvor de har verdier på rundt 1/3 av verdien til systemet uten ressurskontroll. For dette systemet er en løsning med ressurskontroll best og faktisk helt nødvendig for at systemet ikke skal stoppe å fullførte transaksjoner.

8.5 Oppsummering og vurdering

I dette underkapitlet vurderer vi de tre alternativene for ressurskontroll generelt og ut i fra de definerte løsningsmålene, slik vi presenterte dem i kapittel 5.1. Vi starter med distribuert nodetabell, før vi ser på transaksjonsklarering og til slutt sentralisert nodetabell. I vurderingen brukes betegnelsen *maksimal responstid* om tiden `PTrolThread` krever svar innenfor for å ikke starte abortprosesseringen fra alle nodene som er med i transaksjonsutførelsen, i tillegg til den tiden det tar å sende meldingene i den siste fasen av transaksjonsutførelsen.

I kapittel 8.1 bestemmer vi hvilke parametere vi skal variere under simulering. Dette utvalget ble valgt for å dekke flest mulig simuleringssituasjoner, men samtidig ikke gi flere simuleringer enn det vi rekker å gjennomføre og å drøfte resultatene av. Utvalget her gjør det vanskelig å vurdere mål M1E fra kapittel 5.1, og vurderingen her er utelatt. Dette målet kan være aktuelt å studere nærmere i videre arbeid. Vi har likevel gjort en kort drøfting av hvordan vi tror dette målet oppfylles av de ulike ressurskontroll-alternativene i oppsummeringen til slutt i dette underkapitlet.

Hvert av underkapitlene har tabeller som viser hvor mange prosent ressurskontrollen har brukt av CPUens tid, og disse inngår i vurderingen av løsningsmålene. Resultat-tabellene innenfor hvert simuleringskapittel ligger også til grunn for vurderingen, men vi finner det ikke nødvendig å ta med disse tabellene igjen her.

Sist i underkapitlet drøfter vi i hvilke systemer ressurskontroll egner seg best, og hvilke systemer som fungerer best uten ressurskontroll.

8.5.1 Distribuert nodetabell

Varianten distribuert nodetabell viser seg å gjøre mest nytte i systemer med høy belastning. I slike systemer er distribuert nodetabell også det beste alternativet av ressurskontrollene. Den distribuerte nodetabellen viser svakere resultater for systemer med mindre belastning.

I tabell 8.16 og tabell 8.17 ser vi hvor mange prosent av tiden CPUen kjører distribuert nodetabell i systemer med og uten refragmentering. Vi ser at andelen av tiden CPUen jobber som benyttes til å kjøre ressurskonroll i refragmenteringssystemer øker når tiden mellom transaksjonsankomster og sannsynligheten for diskavbrudd øker. Dette er naturlig siden CPUen da utfører mindre arbeid utenom ressurskontrollen. Siden ressurskontrolljobben er tilnærmet konstant i størrelse, vil andelen CPU-tid øke. Den samme trenden ser vi i systemer uten refragmentering. Ved høyere sannsynlighet for diskavbrudd, vil også CPUen i større grad vente på at disken skal fullføre oppdateringene, og får dermed mindre å gjøre ved siden av å utføre ressurskontroll. Sammenlignet med de andre ressurskontrollene ligger distribuert nodetabell litt lavere i andelen av CPU-tiden den krever, enn sentralisert nodetabell, mens den ligger høyere enn transaksjonsklarering. Spesielt gjelder dette i systemer uten refragmentering.

Tabell 8.16: Distribuert nodetabell med refragmentering

System	RK-tid / CPU-tid totalt
40 ms ankomstid	
probPageFault 0.0	13,8 %
probPageFault 0.2	13,1 %
50 ms ankomstid	
probPageFault 0.0	14,9 %
probPageFault 0.2	14,1 %
60 ms ankomstid	
probPageFault 0.0	15,7 %
probPageFault 0.2	15 %

Tabell 8.17: Distribuert nodetabell uten refragmentering

System	RK-tid / CPU-tid totalt
15 ms ankomstid	
probPageFault 0.0	16,8 %
probPageFault 0.2	15,9 %
probPageFault 0.4	-
20 ms ankomstid	
probPageFault 0.0	17,9 %
probPageFault 0.2	18 %
probPageFault 0.4	22,9 %

I tabell 8.18 vurderes hvordan distribuert nodetabell oppfyller løsningsmålene fra kapittel 5.1.

Tabell 8.18: Vurdering av distribuert nodetabell

Mål	Vurdering
M1A	Antall meldinger per fullført transaksjon er lavest for denne ressurskontrolltypen. Distribuert nodetabell oppfyller dette målet tilfredsstillende ut i fra simuleringen og sammenlignet med de andre ressurskontrollene og systemet uten ressurskontroll.
M1B	Distribuert nodetabell oppfyller dette målet bra, sammenliknet med et system uten ressurskontroll. Sammenliknet med transaksjonsklarering er antallet aborterte transaksjoner for høyt.
M1C	Distribuert nodetabell beslaglegger en større andel av CPU-tiden enn transaksjonsklarering gjør. Spesielt i systemer uten refragmentering. Det er ønskelig at denne andelen hadde vært lavere. I systemer med refragmentering ligger andelen i gjennomsnitt på ca 14,5 %, mens den ligger på ca 18 % i systemer uten refragmentering.
M1D	Ressurskontrollen aborterer flere transaksjoner enn det som er ønskelig. I prinsippet skal ressurskontrollen avvise transaksjoner slik at aborteringer unngås. På dette punktet er ikke målet oppfylt. Distribuert nodetabell avviser også flere transaksjoner enn det som sees på som nødvendig. Spesielt ser vi dette i tilfeller der systemer uten ressurskontroll fullfører alle transaksjonene med lavere gjennomsnittlig responstid enn distribuert nodetabell, som i de samme tilfellene avviser transaksjoner. Det positive ved dette alternativet er antall fullførte transaksjoner som regel er høyere enn for transaksjonsklarering og sentralisert nodetabell. Dette gir en høy gjennomstrømning (eng: throughput), noe som er positvt i forhold til dette målet. For systemer med høy belastning oppfylles målet mer tilfredsstillende, både med henblikk på antall fullførte og antall aborterte transaksjoner.
M1E	Ikke vurdert.
M2A	Siden distribuert nodetabell i noen tilfeller aborterer mange transaksjoner, vil ikke dette målet alltid være oppfylt. Målet er kun oppfylt for de tilfellene der distribuert nodetabell ikke aborterer noen transaksjoner.
M2B	Distribuert nodetabell avviser flere transaksjoner gjennom simuleringen enn de transaksjonene som ikke ville blitt fullført innenfor maksimal responstid. Dette sees spesielt godt i simuleringer der systemet uten ressurskontroll fullfører alle transaksjoner og gjør dette med en lavere gjennomsnittlig responstid. Distribuert nodetabelll oppfyller dermed ikke dette målet tilfredsstillende.

8.5.2 Transaksjonsklarering

Denne ressurskontrollvarianten egner seg best i systemer med høy belastning. I systemer der belastningen er lavere, vil ikke transaksjonsklareringen være til like mye nytte. Denne ressurskontrollen er litt bedre enn de andre ressurskontrollene i systemer uten refragmentering og med hyppig ankomst av transaksjoner.

I tabell 8.19 og tabell 8.20 ser vi hvor mange prosent av tiden CPUen kjører transaksjonsklarering i systemer med og uten refragmentering. Transaksjonsklarering ligger lavest av ressurskontrollene når det gjelder andelen av CPU-tiden den bruker. Den trenden at andelen øker i takt med ankomstrate og disksannsynlighet, gjelder også her på lik linje med det som er omtalt i kapittel 8.5.1. Transaksjonsklarering utpeker seg positivt ved å bruke lavest andel av CPU-tiden i sytemer uten refragmentering.

Tabell 8.19: Transaksjonsklarering med refragmentering

System	RK-tid / CPU-tid totalt
40 ms ankomstid	
probPageFault 0.0	8,7 %
probPageFault 0.2	8,7 %
50 ms ankomstid	
probPageFault 0.0	10,3 %
probPageFault 0.2	10,4 %
60 ms ankomstid	
probPageFault 0.0	11,6 %
probPageFault 0.2	11,7 %

Tabell 8.20: Transaksjonsklarering uten refragmentering

System	RK-tid / CPU-tid totalt
15 ms ankomstid	
probPageFault 0.0	4,2 %
probPageFault 0.2	4,1 %
probPageFault 0.4	-
20 ms ankomstid	
probPageFault 0.0	7,3 %
probPageFault 0.2	7,5 %
probPageFault 0.4	9 %

I tabell 8.21 vurderes hvordan transaksjonsklarering oppfyller løsningsmålene fra kapittel 5.1.

Tabell 8.21: Vurdering av transaksjonsklarering

Mål	Vurdering
M1A	I antall meldinger per fullført transaksjon, ligger transaksjonsklareringen høyest av de ulike ressurskontrollvariantene. Dette medfører en ekstra belastning per transaksjon. Transaksjonsklarering oppfyller dermed ikke dette målet tilstrekkelig godt.
M1B	Transaksjonsklareringen aborterer 4 transaksjoner gjennom simuleringen, og oppfyller dette målet på en god måte.
M1C	Ressurskontrolltypen har best oppfyllelse av alle ressurskontrollene. Transaksjonsklarering ligger ca 3 % lavere enn de andre ressurskontrollene når det gjelder andel av CPU-tiden som brukes til ressurskontroll. I gjennomsnitt ligger den på omlag 10 % i systemer med refragmentering, og omlag 6,5 % i systemer uten refragmentering.
M1D	Denne ressurskontrollen aborterer færrest transaksjoner av ressurskontrollvariantene. På systemer uten refragmentering avviser transaksjonsklarering færrest transaksjoner av ressurskontrolltypene, mens i systemer med refragmentering avvises flere transaksjoner enn det som anses å være nødvendig. Transaksjonsklarering oppfyller målet best i forhold til at antall aborteringer er lavt. Antall avvisninger er noe høyt, da spesielt i systemer med refragmentering. Ser vi på antall fullførte transaksjoner, er dette stort sett på et bra nivå for dette alternativet, litt lavere enn distribuert nodetabell, men klart bedre enn sentralisert nodetabell.
M1E	Ikke vurdert.

Tabell 8.21: Vurdering av transaksjonsklarering

Mål	Vurdering
M2A	Ressurskontrollen har abortert 4 transaksjoner gjennom simuleringene, og i disse tilfellene har ikke maksimal responstid kunne blitt holdt. Med tanke på hvor liten andel av de fullførte transaksjonene dette er, oppnår transaksjonsklarering gode resultater for dette målet.
M2B	I systemer med refragmentering kommer det fram at transaksjonsklareringen avviser transaksjoner, selv om maksimal responstid hadde blitt overholdt. Det kunne være ønskelig at ressurskontrollvarianten slapp igjennom flere transaksjoner i disse tilfellene.

8.5.3 Sentralisert nodetabell

Ressurskontrollen sentralisert nodetabell viser best resultater i systemer med høy belastning. I systemer med lav belastning viser ressurskontrollvarianten de svakeste resultatene av alle ressurskontrolltypene, med unntak av systemet uten refragmentering og en sannsynlighet for diskavbrudd på 0,4. I dette systemet er sentralisert nodetabell like god som de to andre ressurskontrollene.

I tabell 8.22 og tabell 8.23 ser vi hvor mange prosent av tiden CPUen kjører sentralisert nodetabell i systemer med og uten refragmentering. Sentralisert nodetabell er den ressurskontrollen som bruker den største andelen av CPU-tiden i våre simuleringer. Vi ser også for denne ressurskontrollen en økning i andelen CPU-tid som benyttes når gjennomsnittlig ankomsttid og disksannsynlighet øker, slik det er beskrevet i kapittel 8.5.1.

Tabell 8.22: Sentralisert nodetabell med refragmentering

System	RK-tid / CPU-tid totalt
40 ms ankomstid	
probPageFault 0.0	16,3 %
probPageFault 0.2	19,7 %
50 ms ankomstid	
probPageFault 0.0	17,1 %
probPageFault 0.2	17 %
60 ms ankomstid	
probPageFault 0.0	20 %
probPageFault 0.2	17,5 %

Tabell 8.23: Sentralisert nodetabell uten refragmentering

System	RK-tid / CPU-tid totalt
15 ms ankomstid	
probPageFault 0.0	25,4 %
probPageFault 0.2	23,9 %
probPageFault 0.4	-
20 ms ankomstid	
probPageFault 0.0	24,4 %
probPageFault 0.2	23,2 %
probPageFault 0.4	26 %

I tabell 8.24 vurderes hvordan sentralisert nodetabell oppfyller løsningsmålene fra kapittel 5.1.

Tabell 8.24: Vurdering av sentralisert nodetabell

Mål	Vurdering
M1A	Sentralisert nodetabell ligger nesten like lavt som distribuert nodetabell når det gjelder antall meldinger per fullført transaksjon. Den har dermed et lite forbedringspotensiale. For dette målet er sentralisert nodetabell å foretrekke framfor transaksjonsklarering.
M1B	På lik linje med distribuert nodetabell, aborterer sentralisert nodetabell flere transaksjoner enn transaksjonsklarering. Antall aborteringer er lavere enn for distribuert nodetabell, men fremdeles for høyt for at dette målet skal være tilfredsstilt.
M1C	Sentralisert nodetabell beslaglegger den største andelen av CPU-tiden i våre simuleringer, av alle ressurskontrollvariantene. I systemer med refragmentering ligger andelen i gjennomsnitt på omtrent 18 %, mens i systemer uten refragmentering ligger den på omtrent 24,5 %. Sammenlignet med de andre ressurskontrollene, anser vi dette for å være for høyt.
M1D	Ressurskontrollen avviser transaksjoner for å holde systembelastningen nede. Likevel aborteres transaksjoner, noe som viser at maksimal responstid ikke ble overholdt. Vi ser også tilfeller der transaksjoner avvises ved bruk av ressurskontroll, mens systemet uten ressurskontroll ikke har noen problemer med å fullføre transaksjonene innenfor grensen for maksimal responstid. Videre ser vi at sentralisert nodetabell gir færreste fullførte transaksjoner i de fleste simuleringstilfellene. Sentralisert nodetabell oppfyller dermed ikke dette målet tilfredsstillende.
M1E	Ikke vurdert.
M2A	Siden sentralisert nodetabell aborterer transaksjoner, vil ikke dette målet alltid være oppfylt. Det er likevel oppfylt i de tilfellene der abortering ikke forekommer, siden alle transaksjonene som da fullfører gjør dette innenfor maksimal responstid. Målet er ikke oppfylt i alle simuleringene.
M2B	Sentralisert nodetabell avviser transaksjoner i tilfeller der maksimal responstid kan overholdes, og den lar være å avvise i enkelte tilfeller der maksimal responstid ikke kan overholdes. Sentralisert nodetabell oppfyller derfor ikke dette målet tilfredsstillende.

8.5.4 Oppsummering ressurskontroll

Vi ser behov for ressurskontroll i systemer både med og uten refragmentering. Dette behovet er spesielt framtredende i systemer der man har hyppig ankomst av nye transaksjoner. I de tilfellene der ankomsten av transaksjoner er så hyppig at systemet ikke klarer å fullføre like mange transaksjoner som det mottar, er ressurskontrollen helt nødvendig for at systemet ikke skal bryte sammen og abortere alle transaksjonene.

I systemer der transaksjoner ikke ankommer like hyppig, er ikke behovet for ressurskontroll like framtredende. Ressurskontrollens virkning her er blant annet å begrense systembelastningen ved å redusere antall meldinger i systemet, men dette gjøres ofte på bekostning av at færre transaksjoner fullføres. Når vi studerer den gjennomsnittlige transaksjonstiden, ser vi at den som regel er vesentlig lavere for systemene med ressurskontroll, på bekostning av at flere transaksjoner som normalt ville gitt høy responstid eller abortprosessering, avvises. Ressurskontroll-alternativene viser seg også som positive ved at de avslutter refragmenteringsperioden tidligere enn systemet uten ressurskontroll. Dette gjør at systemene raskere blir mindre belastet, men på bekostning av at flere transaksjoner avvises. I de tilfeller der antall avviste transaksjoner overstiger antall aborterte transaksjoner for systemene uten ressurskontroll, finner vi dette spesielt uheldig, da det er et tegn på at for mange transaksjoner avvises.

Ser vi på hvordan sannsynligheten for diskaksess påvirker hvordan ressurskontroll-alternativene oppfører seg, ser vi at ressurskontroll er mer gunstig for systemer med høyere sannsynlighet for diskaksess. Denne trenden ser vi klart på alle systemene vi har simulert. Når vi simulerer systemer som bruker disken, ser vi også at ressurskontroll-alternativene fullfører transaksjoner under refragmenteringen. Dette er positivt, siden vi dermed ikke får den samme stoppen i gjennomstrømningen i systemet. For systemene uten disk, altså med en sannsynlighet for diskaksess på 0, ser vi at det er kun hyppigere ankomster av transaksjoner som gir så tungt belastede systemer at ressurskontrollen er nødvendig.

Gjennom simuleringene gir distribuert nodetabell etter vår mening de beste resultatene av ressurskontrollvariantene, siden den har best resultater totalt sett. Vi ser likevel at denne løsningen ikke er optimal, da den blant annet aborterer flere transaksjoner enn det vi finner nødvendig. Transaksjonsklarering ser vi på som den nest beste løsningen, og ved flere tilfeller bedre enn distribuert nodetabell. Den største ulempen med transaksjonsklarering er at løsningen gir mange meldinger for å utføre ressurskontroll, og slik sett en unødvendig høy ekstrabelastning på systemet. Sentralisert nodetabell er den løsningen vi ser på som dårligst av dem vi har implementert. Dette alternativet viser stort sett dårligste resultater ved alle systemoppsett som vi har simulert.

Mål M1E, som omhandler skalering, har vi som nevnt ikke simuleringsresultater for å gjøre en god vurdering av. Vi vil likevel gjøre noen betraktninger om hvordan vi tror de ulike ressurskontroll-alternativene oppfyller dette målet. Denne diskusjonen gjør vi ut i fra en økning i antall noder i systemet. Distribuert nodetabell er en løsning hvor hver node sender sin tilstand til de andre nodene i systemet ved tilstandsendringer. Dermed ser vi at denne løsningen vil ha en økning i antall meldinger ved et større system. Denne økningen vil være lineær og gi hver node i system økt belastning, om vi antar at meldingene sendes like hyppig som tidligere. Ser vi på transaksjonsklarering, vil vi ikke få en like sterk økning i antall meldinger om vi antar det samme antall transaksjoner i systemet. Grunnen til dette er at denne løsningen sender eksplisitte meldinger for hver transaksjon som skal utføres til de nodene som er involverte. Likevel vil det ved å innføre flere noder bli mindre sannsynlighet for at en node får to eller flere av oppdateringene for en transaksjon, og at det dermed blir flere noder involvert i en transaksjon. Dermed vil vi også her få en økning i antall meldinger. Vi vil anta at denne økningen er mindre enn økningen for distribuert nodetabell, i alle fall om antall oppdateringer per transaksjon er på et relativt lavt nivå i forhold til antall noder. Sentralisert nodetabell vil medføre en lineær økning i antall meldinger som sendes til den sentrale noden, mens de øvrige nodene ikke vil oppleve noen forskjell. Dermed er det kun den sentrale noden som vil oppleve en belastningsøkning ved denne løsningen. Denne belastningsøkningen kan imidlertidig bli så stor at denne noden vil oppleves som en flaskehals i systemet. En mulig løsning på dette problemet kan være å gi denne noden ekstra ressurser i forhold til de øvrige nodene i systemet.

9 Konklusjon og videre arbeid

Vi har gjennom dette diplomarbeidet laget en simulator som simulerer transaksjonsutførelse i databasesystemet ClustRa, ut i fra den modelleringen vi har gjort av ClustRa. I denne simulatoren har vi gjennomført flere simuleringer på ulike systemoppsett som tester de ulike alternativene for ressurskontroll opp mot systemet uten ressurskontroll. Vi har også drøftet hvilke løsninger som er mest hensiktsmessig for de ulike systemene.

Vi vil i dette kapitlet først se på konklusjoner vi kan trekke ut av arbeidet vi har gjort med distribuert ressurskontroll. Videre ser vi på hvordan vi har oppfylt målene vi har satt til diplomarbeidet. Avslutningsvis presenterer vi det vi mener kan gjøres av videre arbeid, og mangler ved det arbeidet vi har gjennomført.

9.1 Konklusjon

Ut i fra resultatene og drøftingene vi presenterer i kapittel 8 og betrakninger vi har gjort på prøve-simuleringene vi har utført, kan vi trekke følgende hovedkonklusjoner:

- Distribuert ressurskontroll sikrer at systemene ikke bryter sammen som en følge av for mange transaksjoner og transaksjonsprosessering. Grunnen til dette er at transaksjoner avvises ved tungt belastede systemer med ressurskontroll, og at dette er lite belastende i forhold til abortprosesseringen som skjer i systemer uten ressurskontroll. Vi har dokumentert i kapittel 8 at systemer uten ressurskontroll kan bryte sammen ved at ingen transaksjoner fullfører, alle aborterer og tiden for å utføre abort øker eksponentielt.
- Distribuert ressurskontroll er hensiktsmessig på tungt belastede systemer, som vi har testet spesielt med refragmentering og kort tid mellom hver ankomst av nye transaksjoner. Under slike systemer klarer ressurskontrollen å holde den gjennomsnittlige transaksjonstiden på et jevnt nivå ved å avvise transaksjoner. Systemer uten ressurskontroll vil i slike tilfeller ha en sterk økning i den gjennomsnittlige transaksjonstiden og bruke lengre tid på å komme seg etter en periode med tung belastning. Dette henger igjen sammen med at abortprosessering krever mer prosessering enn å avvise transaksjoner. Vi ser også at ressurskontrollen gjør at flere transaksjoner får fullføre ved slike tungt belastede systemer.
- Sannsynligheten for diskaksess har betydning for hvor nødvendig ressurskontroll er. Vi ser en klar trend som viser at ressurskontroll er mer nødvendig for høyere sannsynlighet for dataaksess på disk. For systemer med hyppig diskaksess ser vi at det fort blir helt nødvendig med ressurskontroll for at systemet ikke skal stoppe opp på grunn av uendelig lange diskkøer.
- Distribuert ressurskontroll gir færre fullførte transaksjoner ved middels belastede systemer hvor systemer med ressurskontroll avviser flerer transaksjoner enn systemene uten ressurskontroll aborterer. Dette finner vi uheldig. Likevel ser vi at den gjennomsnittlige transaksjonstiden ofte er lavere for systemene med ressurskontroll, noe som er positivt. Dette henger sammen med at det avvises transaksjoner som ville gitt høy responstid. Vi finner altså her situasjoner hvor det er positive og negative sider ved å bruke ressurskontroll. En mulig forbedring er her å endre oppsettet av ressurskontrollen slik at systemene håndterer denne typen situasjoner bedre.
- Ved lite belastede systemer finner vi liten forskjell på oppførselen til systemer med og uten ressurskontroll. Vi ser likevel at innføringen av ressurskontroll er uheldig i noen grensesystemer hvor ressurskontroll-alternativene avviser transaksjoner mens systemet uten ressurskontroll klarer å fullføre alle transaksjonene. Systemer med lavere ankomstrate har vi ikke presentert simuleringer for i kapittel 8, men prøve-simuleringer vi har gjort viser liten forskjell mellom systemer med og uten ressurskontroll.

I kapittel 8.5 har vi gjort en oppsummering av de ulike ressurskontrolltypene og vurdert dem opp mot løsningsmålene. Vi ser av denne drøftingen at det ikke er noe entydig svar på hvilket alternativ som er best, dette varierer ut i fra hvilket systemoppsett vi har simulert under. Vi ser likevel noen trender, hvor de viktigste og mest tydelige er:

- Distribuert nodetabell er best på tungt belastede systemer i forhold til alle målte parametere
- Transaksjonsklarering gir flest meldinger
- Distribuert nodetabell aborterer flest transaksjoner
- Distribuert nodetabell og transaksjonsklarering har som regel flere fullførte transaksjoner enn sentralisert nodetabell, med distribuert nodetabell som den beste

- Distribuert nodetabell og transaksjonsklarering har i snitt omtrent det samme antall avviste transaksjoner, mens sentralisert nodetabell har mange flere
- Den gjennomsnittlige transaksjonstiden er avhengig av forholdet mellom antall avviste og fullførte transaksjoner

9.2 Måloppnåelse

I kapittel 2.2 definerte vi arbeidsmål for vårt diplomarbeid. Disse målene ble definert i samarbeid med veileder Svein Erik Bratsberg. Vi vil nå se på hvordan vi har oppfylt hvert av disse målene. Hvordan våre løsninger for ressurskontroll oppfyller målene for løsning er presentert i kapittel 8.5.

A1 - Modellering og forståelse av ClustRa

I samarbeid med veileder Svein Erik Bratsberg har vi gjort begrensninger på hvilke deler av ClustRa vi har valgt å modellere og å implementere i simulatoren. Vi har her prioritert å få en god generell forståelse av ClustRa og hvordan transaksjonsprosesseringen utføres. Vi har gjort disse valgene for å konsentrere oss om de deler av ClustRa som er relevant i forhold til å se på ressurskontroll og fordi vi ikke har hatt tid til å studere hele systemet. Den delen av ClustRa vi har valgt å simulere har vi fått god forståelse av, og vi har gjennomført en modellering slik vi har dokumentert i denne rapporten.

A2 - Definere løsningsmål og løsningsalternativer for distribuert ressurskontroll

I høstens rapport [1] gjorde vi et bredt studium av andre fagfelt og kom fram til flere mulige løsningsalternativer for det distribuerte ressurskontrollproblemet i ClustRa. Vi studerte også hvordan intern ressurskontroll kan utføres. I arbeidet vi har gjort nå i vår, har vi tatt disse løsningsalternativene videre og plukket ut tre alternativer for både distribuert ressurskontroll og intern ressurskontroll som vi har valgt å simulere. Dette valget har vi gjort på bakgrunn av en prioritering av de løsningsforslagene vi fant mest interessante og de som viste seg best å simulere og teste i forhold til tilgjengelige testdata fra ClustRa. Vi måtte gjøre en slik utvelging for å begrense omfanget av oppgaven.

Vi har også definert løsningsmål for mulige løsninger, slik vi har presentert i kapittel 5.1.

A3 - Programmere en simuleringsmodell for ClustRa med distribuert ressurskontroll

Vi har programmert en simulator som gjennomfører transaksjonsprosessering, refragmentering og ressurskontroll i henhold til beskrivelsen gitt av veileder Svein Erik Bratsberg. Simulatoren er skrevet i Java og benytter simuleringsrammeverket Desmo-J. Implementasjonen inneholder mellom seks og sju tusen kodelinjer og har egne innstillingsfiler hvor alle parametere som påvirker kjøringen settes. Simulatoren er dokumentert gjennom Javadoc og beskrevet i kapittel 6 i denne rapporten.

A4 - Gjennomføre simulering og måle resultatene opp mot løsningsmålene

Vi har gjennomført 48 simuleringer som vi dokumenterer og drøfter i kapittel 8. Dette har vi gjort ved å se på 12 ulike systemoppsett og kjøre fire simuleringer på hvert av oppsettene; én med system uten ressurskontroll og én for hver av de tre ressurskontroll-alternativene.

Utover disse simuleringene har vi gjennomført flere prøve-simuleringer for å sette fornuftige parametere og for å finne de grensetilfeller vi har valgt å simulere. Vi har også gjort en drøfting av de ulike alternativene for ressurskontroll opp mot systemet uten ressurskontroll for hvert av simuleringstilfellene, noe vi har dokumentert i kapittel 8. Dette kapitlet avsluttes med en drøfting av hvordan de ulike alternativene for ressurskontroll oppfyller målene vi har satt til god løsning.

9.3 Videre arbeid og mangler ved vårt arbeid

Vi vil i dette underkapitlet se på hva som kan gjøres av videre arbeid, og mangler ved arbeidet vi har utført.

Som nevnt i kapittel 8.5 har vi ikke studert hvordan våre løsninger er i forhold til skalering. Dette er noe vi har nedprioritert fordi vi har valgt å prioritere andre aspekter som dekker flere av løsningsmålene våre. Det som vi ser for oss av videre arbeid her er å gjennomføre ulike simuleringer med varierende størrelse på systemet og se hvordan de ulike ressurskontrolltypene

utvikler seg. Dette henger sammen med vårt løsningsmål “M1E - Ressurskontrollutførelsen skal være skalerbar slik at den totale systembelastningen skal øke minimalt ved en systemekspansjon”.

Videre har vi vist simuleringssituasjoner hvor distribuert ressurskontroll har vist seg uheldig ved at det har avvist transaksjoner hvor det ikke aborteres transaksjoner uten ressurskontroll. Dette mener vi kan være mulig å gjøre noe med ved å se på alternative implementasjoner av ressurskontroll. Vi kan ikke se å klare å løse dette problemet slik vi har implementert løsningene og satt opp ClustRa.

Vi ser videre at vår modell av ClustRa er en del enklere enn det virkelige databasesystemet. Følgelig kan dette gi utslag i våre måleresultater. Av den grunn kan det være hensiktsmessig med en større simulering som tar med flere aspekter av databasesystemet for å se hvordan ressurskontrollen oppfører seg på denne typen systemer.

Til slutt vil vi poengtere at vi ikke har hatt ressurser til å utføre noen fullstendig simulering, det er mange simuleringsituasjoner som kunne vært dekket bedre. Derfor kan det være nye konklusjoner som kan komme fram ved å gjøre flere simuleringer, gjerne med andre systemoppsett enn det vi har brukt.

Referanser

[1] M. Solberg og E. Aasland, *Distribuert ressurskontroll*, november 2004, NTNU

[2] S. Bratsberg med flere, *Parallel Solutions in ClustRa*, 1997

[3] S. Bratsberg, *Distributed Resource Control in Clustra*, 27. september, 2004

[4] S. Bratsberg og R. Humborstad, *Online Scaling in a Highly Available Database*, 2001

[5] S. Bratsberg, *Beskrivelse av transaksjonsutførelse i Clustra*, 14. mars 2005

[6] Nettstedet Core Java J2SE, http://java.sun.com/j2se/1.5.0/index.jsp, besøkt fram til 12. mai 2005

[7] *JavaTM 2 Platform Standard Edition 5.0 API Specification*, http://java.sun.com/j2se/1.5.0/docs/api/, besøkt fram til 29. mai 2005

[8] *Code Conventions for the Java Programming Language*, http://java.sun.com/docs/codeconv/index.html, besøkt fram til 15. mai 2005

[9] *Javadoc Tool Home Page*, http://java.sun.com/j2se/javadoc/index.jsp, besøkt fram til 20. mai 2005

[10] Sheldon M. Ross, *Introduction to probability models*, eight edition, Academic press, 2003

[11] *IBM information center*, http://publib.boulder.ibm.com/infocenter/tiv3help/index.jsp?topic=/com.ibm.itmfd.doc/mssql511rg38.htm, besøkt fram til 6. mai 2005

[12] *Pareto principle*, http://en.wikipedia.org/wiki/Pareto_principle, besøkt fram til 31. mai 2005

[13] *Desmo-J API*, http://asi-www.informatik.uni-hamburg.de/themen/sim/forschung/Simulation/Desmo-J/api/index.html, besøkt fram til 31. mai 2005

[14] *Desmo-J*, http://asi-www.informatik.uni-hamburg.de/themen/sim/forschung/Simulation/Desmo-J/index.html, besøkt fram til 2. juni 2005

[15] *Desmo-J in a Nutshell*, http://asi-www.informatik.uni-hamburg.de/desmoj/tutorial/nutshell/1.html, besøkt fram til 31. mars 2005

[16] *Download Java 2 Platform Standard Edition 5.0*, http://java.sun.com/j2se/1.5.0/download.jsp, besøkt fram til 23. mai 2005

[17] *DESMO-J Download*, http://asi-www.informatik.uni-hamburg.de/desmoj/download.html, besøkt fram til 23. mai 2005

[18] A. Myskja med flere, *Telektronikk*, Telenor Research & Development, 1995

Vedlegg A: Forklaring av innstillingsfilene

Vårt simuleringsrammeverk har flere parametere som styrer hvordan en konkret simulering skal kjøres. Disse parameterne har vi samlet i egne filer av typen “filnavn.properties”. Vi har valgt en slik løsning for å sikre at koden vår er lett å endre og at det skal være mulig å kjøre simuleringer uten å kjenne til selve implementeringen. Parameterne vi beskriver i dette vedlegget er fastsatt ut i fra tilgjengelige testdata fra ClustRa, prøve-simuleringer vi har gjort og erfaringer med andre relevante systemer. Hvordan paramterne for ressurskontroll-oppsettet er fastsatt, er spesielt beskrevet i kapittel 7.

I beskrivelsen som følger av de ulike parameterne har vi tre kolonner i tabellene som inneholder parameterne:

- Parameter, som er navnet på parameteren
- Forklaring, som forklarer hva parameteren representerer
- Verdi, som angir standardvedien vi har brukt ved våre simuleringer. Der hvor det står “variabel” er det en parameter vi endrer på under simuleringene.

Alle variable parametere er samlet i følgende filer:

- `Meantimes.properties`, hvor vi har samlet forventningstidene til ulike operasjoner i simulatoren. Innstillingene som settes her er forklart i tabell A.1.
- `MessageLengths.properties`, her finnes parametere som setter lengden på meldingene i systemet. Innholdet i denne filen er forklart i tabell A.2.
- `MessageProcessing.properties`, hvor vi har samlet tidene for prosessering av meldinger hos sender og mottaker av meldingene, ut i fra meldingstype. Disse innstillingene er forklart i tabell A.3.
- `Probabilities.properties`, her har vi alle sannsynlighetsparameterne som brukes i simuleringen. Disse innstillingene forklares i tabell A.4.
- `ResourceControl.properties`, som samler alle innstillinger spesifikt knyttet opp mot ressurskontroll. Parameterne fra denne filen forklares i tabell A.5.
- `SystemProperties.properties` inneholder alle generelle parametere til simuleringen. Innstillingene i denne filen er beskrevet i tabell A.6.

Tabell A.1: Parametere fra Meantimes.properties

Parameter	Forklaring	Verdi
standardDeviationMemUpdate	Standardavviket for oppdatering av en databasepost i minnet	0,1
standardDeviationDiskUpdate	Standardavviket for kjøring av en oppdatering på disk	0,1
mtWriteRecordBuffer	Gjennomsnittlig tid for å skrive et tuppel til buffer	0,479 ms
mtWriteFourRecordsBuffer	Gjennomsnittlig tid for å skrive fire tupler til buffer	0,842 ms
mtWriteRecordDisk	Gjennomsnittlig tid for å skrive et tuppel til disk	10 ms
mtWriteRecordDiskPr16KB	Gjennomsnittlig tid for å skrive til disk sekvensielt, målt pr. 16 KB-blokk	1 ms
mtBetweenArrivals	Gjennomsnittlig tid mellom ankomst av transaksjoner fra klientene	variabel

Tabell A.2: Parametere fra MessageLengths.properties

Parameter	Forklaring	Verdi
AckLogRec	Meldingslengden for “AckLogRec”	1 byte
HSSlaveDone	Meldingslengden for “HSSlaveDone”	9 bytes
HSSlaveReply	Meldingslengden for “HSlaveReply”	10 bytes

Tabell A.2: Parametere fra MessageLengths.properties

Parameter	Forklaring	Verdi
Logging	Konstant del av meldingslengden til “Logging”. “Logging” er meldingen som sender logg fra primær til hot standby slave.	10 bytes
LoggingFactor	Faktoren til “Logging”	58 bytes/update
PSlaveDone	Meldingslengden for “PSlaveDone”	9 bytes
PSlaveReply	Konstant del av meldingslengden til “PSlaveReply”	18 bytes
PSlaveReplyFactor	Faktoren til “PSlaveReply”	6 bytes/update
TcCommit	Meldingslengden til “TcCommit”	9 bytes
StartHSSlave	Meldingslengden til “StartHSSlave”	22 bytes
StartPSlave	Konstant del av meldingslengden til “StartPSlave”	43 bytes
StartPSlaveFactor	Faktoren til “StartPSlave”	54 bytes/update
StartTc	Konstant del av meldingslengden til “StartTc”	22 bytes
StartTcFactor	Faktoren til “StartTc”	8 bytes/update
StartTcHs	Meldingslengden til “StartTcHs”	21 bytes
TcHsDone	Meldingslengden til “TcHsDone”	9 bytes
TcFinish	Meldingslengden til “TcFinish”	9 bytes
TcHsFinished	Meldingslengden til “TcHsFinished”	9 bytes
TcReply	Kontant del av meldingslengden til “TcReply”	37 bytes
TcReplyFactor	Faktoren til “TcReply”	4 bytes/update
TcAbort	Meldingslengden til “TcAbort”	30 bytes
SlaveAbort	Meldingslengden til “SlaveAbort”	9 bytes
RC_msg_1	Meldingslengden til ressurskontrollmeldinger mellom nodene i systemet ved “distribuert nodetabell” som ressurskontrolltype.	30 bytes
RC_msg_2	Meldingslengden til ressurskontrollmeldinger mellom nodene i systemet ved “transaksjonsklarering” som ressurskontrolltype.	30 bytes
RC_msg_3	Meldingslengden til ressurskontrollmeldinger mellom nodene i systemet ved “sentralisert nodetabell” som ressurskontrolltype.	30 bytes
RefragTable	Lengden på meldingen “RefragTable”	20 bytes
RefragTableAck	Lengden på meldingen “RefragTableAck”	20 bytes
RefragSlave	Lengden på meldingen “RefragSlave”	100 bytes
RefragSlaveAck	Lengden på meldingen “RefragSlaveAck”	10 bytes
RepChunk	Lengden på meldingen “RepChunk”	16 000 bytes
RepChunkReply	Lengden på meldingen “RepChunkReply”	10 bytes
RepDone	Lengden på meldingen “RepDone”	10 bytes
RepDoneReply	Lengden på meldingen “RepDoneReply”	10 bytes
SlowLog	Lengden på meldingen “SlowLog”	10 bytes
SlowLogReply	Lengden på meldingen “SlowLogReply”	10 bytes

Tabell A.3: Parametere fra MessageProcessing.properties

Parameter	Forklaring	Verdi
factorSendNodeNode	Faktor for prosessering av meldingssending mellom to ulike noder	0,001 ms/byte
constantSendNodeNode	Konstant prosesseringskostnad for meldingssending mellom to ulike noder	0,082 ms
factorSendInternal	Faktor for prosessering av meldingssending internt på en node	0 ms/byte
constantSendInternal	Konstant prosesseringskostnad for meldingssending internt på en node	0,065 ms
factorRecNodeNode	Faktor for prosessering av meldingsmottak av meldinger sendt mellom to ulike noder	0,001 ms/byte
constantRecNodeNode	Konstant prosesseringskostnad for meldingsmottak mellom to ulike noder	0,082 ms
factorRecInternal	Faktor for mottak av meldinger sendt internt på en node	0 ms/byte
constantRecInternal	Konstant prosesseringskostnad for meldingsmottak av interne meldinger på en node	0,065 ms
factorSendRCMsg	Faktoren for prosessering av meldingssending av ressurskontrollmeldinger	0,001 ms/byte
constantSendRCMsg	Konstant prosesseringskostnad for meldingssending av ressurskontrollmeldinger	0,082 ms
factorRecRCMsg	Faktoren for prosessering ved meldingsmottak av ressurskontrollmeldinger	0,001 ms/byte
constantRecRCMsg	Konstant prosesseringskostnad for meldingsmottak av ressurskontrollmeldinger	0,082 ms

Tabell A.4: Parametere fra Probabilities.properties

Parameter	Forklaring	Verdi
probPageFault	Sannsynligheten for at en side/blokk man ønsker å aksessere ikke ligger i minnet slik at man må lese den inn fra disk.	Variabel

Tabell A.5: Parametere fra ResourceControl.properties

Parameter	Forklaring	Verdi
internalResourceControlType	Bestemmer hvilken type intern ressurskontroll som skal utføres når ressurskontrollen er aktiv. Følgende muligheter: 0 - Disklasten, 1 - CPU-lasten og 2 - Både disk- og CPU-last.	2
distributedResourceControlType	Bestemmer hvilken type distribuert ressurskontroll som skal brukes. Vi har følgende valgmuligheter: 0 - Ingen ressurskontroll, 1 - Distribuert nodetabell, 2 - Transaksjonsklarering og 3 - Sentralisert nodetabell	Variabel
CPUBusyBoundary	Andelen av tiden CPUen må være aktiv for at CPUen skal regnes som “Busy”	0,8

Tabell A.5: Parametere fra ResourceControl.properties

Parameter	Forklaring	Verdi
numberOfCPUValues	Antall målinger av CPU-lasten som brukes for å sette tilstanden til en node. Ved et antall større enn 1 brukes snittet av de siste målingene.	3
DiskBusyBoundary	Minimum antall diskjobber i kø på en node for å sette tilstanden “Busy”. Har en node flere disker beregnes snittet av køene til diskene.	10
numberOfDiskValues	Antall målinger av disklasten som brukes for å sette tilstanden til en node. Ved et antall større enn 1 brukes snittet av de siste målingene.	1
timeoutTransactionClearance	Timeout-tiden for å motta svar fra de deltakende noder ved transaksjonsklarering.	20 ms
interval_rc	Tiden mellom hver gang ressurskontroll skal utføres	5 ms
internal_RC_time	Tiden det tar å utføre intern ressurskontroll.	0,479 ms

Tabell A.6: Parametere fra SystemProperties.properties

Parameter	Forklaring	Verdi
numberOfNodes	Antall noder i systemet. Dette antallet må alltid være et partall for å oppnå speilnodeforholdet i ClustRa.	4
numberOfDisk	Antall disker på hver node	2
numberOfSimultaneousTransactions	Det maksimale antallet samtidige transaksjoner startet av en klient. Velges tallet “-1” som verdi her vil systemet kjøre uten noe grense og klientene vil fortløpende starte nye transaksjoner basert på parameteren “mtBetweenArrivals”.	-1
updateScalePrimaryHS	Forholdet mellom tiden en oppdatering vil ta på den primære noden og hot standby-noden. Tiden på hot standby = updateScalePrimaryHS * tiden på primærnoden.	0,7
transTimeout	Tiden for timeout. Timeout må skje før alle slaver har svart og meldingen “StartTcHs” er sendt fra `PTrolThread`. Etter dette skal transaksjoner gjennomføres. Timeout vil gi en abort av transaksjonen.	200 ms
initialTimerInterval	Den initielle tiden for timer-kjøring. Det vil si første gangen en timer skal kjøre for sending av meldinger på nytt etter at meldingen er sendt om ønskede svar ikke er mottatt.	10 ms
maxTimerValue	Den maksimale verdien timere kan få. Overstiges denne verdien antar vi at vi ikke får svar fra den eller de man venter på svar fra.	2000 ms

Tabell A.6: Parametere fra SystemProperties.properties

Parameter	Forklaring	Verdi
startAbortTimer	Den initielle tiden for kjøring av abort-timere, som er en spesiell type timere som brukes ved abort-prosessering.	50 ms
tickTime	Tiden CPUen teller opp når den ikke gjør noe arbeid.	0,05 ms
numberOfUpdates	Antall oppdateringer som hver transaksjon inneholder	4
sendReplyBeforeLog	Hvorvidt "SlaveReply" skal sendes før eller etter at loggen utføres. 1 angir at dette skal skje, 0 angir at det ikke skjer.	1
startTracePeriod	Starttiden for å skrive detaljer om hva som skjer under simuleringen	0
stopTracePeriod	Stoptiden for å skrive detaljer om hva som skjer under simuleringen	0
startDebugPeriod	Startiden for å aktivere skriving av debugfilen	0
stopDebugPeriod	Stopptiden for skriving av debugfilen	0
stopTime	Hvor lenge simuleringen skal kjøre	Variabel
clientSendToArbitraryNode	Hvorvidt klienten sender transaksjoner til en best `PTrolThread` eller om den sender til en tilfeldig. 0 vil si at klienten sender til en bestemt node, 1 angir at den sender til tilfeldig.	0
showMsgTrace	Hvorvidt meldingstracen skal skrives ut eller ikke. 1 om tracen skal vises, 0 ellers.	0
numberOfTransesToTrace	Antall transaksjonen det skal skrives meldingstrace for når dette er valgt	0

Vedlegg B: Simuleringsrammeverket Desmo-J

Desmo-J er et simuleringsrammeverk for programmeringsspråket Java utviklet ved universitetet i Hamburg. Vi har benytter versjon 2.0.1 i vår simulator. I dette vedlegget vil vi forklare de viktigste delene av rammeverket som benyttes i simulatoren vår. Vedlegget er delt inn i fire deler, som forklarer de mest sentrale delene av simulatoren for vårt arbeid:

- De grunnleggende klassene som ligger i bunnen av simuleringsrammeverket, og som står for kjøringen av simuleringsmotoren
- De elementene vi benytter under selve simuleringen
- Desmo-J sine statistikkobjekter og rapporteringsfunksjonerfunksjoner
- Simuleringstiden `SimTime`

Beskrivelsen av Desmo-J bygger på [13], [14] og [15].

B.1 Rammeverkklasser

De to hovedklassene til simuleringsrammeverket Desmo-J er `desmoj.core.simulator.Experiment` og `desmoj.core.simulator.Model`. De to klassene må settes opp for at simuleringen skal kunne starte. Vi ser videre i dette underkapitlet nærmere på hver av de to klassene.

B.1.1 Experiment

En simulering kjører under bestemte tidsparametere. I hvor lang tid man skal simulere og hvilke perioder man skal kjøre sporing og feilsøking, bestemmes i klassen `desmoj.core.simulator.Experiment`. Til et eksperiment er det knyttet en modell av typen `desmoj.core.simulator.Model`. Denne todelingen gjør at man enkelt kan bytte ut enten eksperimentet eller modellen.

Etter at simuleringen er ferdig, vil eksperimentet generere rapporter med simuleringsresultater, rapport for sporing og rapport for feilsøking. Eventuelle feil som har oppstått under simuleringen vil komme i rapporten over advarsler. En slik advarsel kan være at et objekt ligger klar til aktivering på to tider i hendelseskøen samtidig. Rapportene er beskrevet i kapittel B.3.3.

Ved oppstart av en simulering, brukes følgende metoder fra `desmoj.core.simulator.Experiment`:

- `tracePeriod(SimTime start, SimTime stop)` setter sporingsperioden til å gå fra tiden `start` til tiden `stop` i simuleringstiden. Simuleringstid er omtalt i kapittel B.4.
- `debugPeriod(SimTime start, SimTime stop)` setter feilsøkingsperioden til å å gå fra tiden `start` til tiden `stop` i simuleringstiden.
- `stop(SimTime stop)` definerer tidspunktet `stop` som den simuleringstiden hele simuleringen skal stoppe. At dette tidspunktet er nådd, er sluttbetingelsen for simuleringen.
- `start()` starter simuleringen.
- `report()` genererer de fire rapportene som er omtalt i starten av dette delkapitlet.
- `finish()` vil avslutte alle de simuleringsprosesser som var aktive da simuleringen ble stoppet. Dette er en opprydning etter fullført simulering.

B.1.2 Model

Dette er hovedklassen i simuleringsmodellen. For å kunne opprette en simuleringsmodell som er basert på Desmo-J, er man avhengig av at én klasse arver `desmoj.core.simulator.Model`. Klassen har tre metoder som er av spesiell betydning for simuleringen:

- `init()` oppretter alle instanser klassen trenger under simuleringen. Desmo-J anbefaler at alle referanser opprettes her, da dette er med på å sikre at de fungerer som de skal under simuleringen.
- `doInitialSchedule()` vil sette opp objektene slik at de er klare til simuleringen starter. Dette vil for eksempel være å aktivere alle objektene som skal være aktive ved simuleringens start.

- `getDescription()` returnerer beskrivelsen av simuleringen. Denne kan inneholde en tekstlig beskrivelse av hva som simuleres, og den kan inneholde verdier på variabler som er brukt i simuleringen. Beskrivelsen vises i rapporten som html, slik at man ved å skrive beskrivelsen i html kan påvirke framstillingen i rapporten.

B.2 Simuleringsenheter

I en simulering er man avhengig av at enheter er aktive slik at ting skjer. Dette kan gjøres på to måter i Desmo-J, enten ved at simuleringen er hendelsesbasert eller prosessorientert. Vi velger å forklare den siste varianten, siden denne benyttes i vår simulator. De objektklassene vi benytter fra Desmo-J er forklart i dette underkapitlet. Dette er klassene `Entity`, `SimProcess` og `Queue`.

B.2.1 Entity

Klasser som arver `desmoj.core.simulator.Entity` kan legges i køer av typen `desmoj.core.simulator.Queue`. Dette er hovedgrunnen til at enkelte av våre klasser arver `desmoj.core.simulator.Entity`. Klassen `desmoj.core.simulator.Queue` er beskrevet i kapittel B.2.3.

B.2.2 SimProcess

Objekter i simuleringen som skal utføre bestemte handlinger ved gitte tidspunkt, vil i vår simulering være av typen `desmoj.core.simulator.SimProcess`. Objekter av denne klassen har en livssyklus som definerer objektets oppgaver i simulatoren. De mest brukte metodene i denne klassen er:

- `lifeCycle()` som beskriver livet til `SimProcess`-objektet. Første gang objektet blir aktivert, vil programflyten starte i denne metoden. Neste gang objektet aktiveres, vil det fortsette fra det punktet der det ble deaktivert.
- `activate(SimTime simTime)` aktiverer objektet på tidspunktet `currentTime` + verdien på `SimTime`-parameteren. Hvis tiden nå er 50 og parameteren har verdi 20, vil objektet aktiveres på tiden 70 i simuleringen. Dersom objektet allerede er aktivt når denne metoden kalles, vil Desmo-J varsle om dette i advarselsrapporten som genereres ved slutten av simuleringen.
- `passivate()` deaktiverer et objekt i det øyeblikken metoden blir kalt
- `hold(SimTime simTime)` deaktiverer objektet og aktiverer objektet igjen ved tidspunktet `currentTime` + verdien på simTime-paramteren. Dette tilsvarer kall av metodene `activate(simTime)` og `passivate()` etter hverandre.

Når det gjelder aktivering og deaktivering av objekter av typen `desmoj.core.simulator.SimProcess`, er det viktig å merke seg det som er illustrert i kode-eksempel B.1.

```
        SimProcess 1
lifeCycle() {
 while (true) {
  simP2.activate(new SimTime(0.0));
  passivate();

  println(“SP1 vekket opp”);
  passivate();
 }
}
```

```
        SimProcess 2
lifeCycle() {

 println(“SP2 vekket opp”);
 println(“Vekker SP1”);
 simP1.activate(new SimTime(0.0));
 passivate();

}
```

Kode-eksempel B.1: Programflyt ved bruk av activate/passivate

I kode-eksemplet over har vi har to klasser `SimProcess 1` og `SimProcess 2` av `desmoj.core.simulator.SimProcess`, og man har opprettet objektene `simP1` og `simP2`. Anta at `simP1` er aktiv ved starten. Det betyr at koden i `simP1` sin `lifeCycle()` blir kjørt. Vi ser at det første som skjer er at `simP2` blir aktivert. Deretter legger `simP1` seg til å sove ved å kalle `passivate()`.

I det øyeblikk `simP2` blir aktivert, vil dette objektet skrive ut at det blir vekket opp. Det neste som skjer er at den skriver ut at `simP1` skal bli vekket opp. Etter at simP1 er vekket opp, legger `simP2` seg til å sove. Objektet `simP1` er igjen blitt aktivert. Til slutt legger `simP1` seg til å sove. Utskriftsrekkefølgen for dette eksemplet er oppsummert i kode-eksempel B.2.

```
"SP2 vekket opp"
"Vekker SP1"
"SP1 vekket opp"
```

Kode-eksempel B.2: Utskrift fra kode-eksempel B.1

Av eksemplet kan vi se at når `simP1` blir aktivert etter det første kallet til `passivate()`, fortsetter kjøringen rett etter dette kallet. Dette er en viktig detalj ved bruk av `lifeCycle()`. Vi ser altså at kontrollen kan slippes midt inne i en metode og at programutførelsen fortsetter fra dette punktet når metoden får tilbake kontrollen. En annen viktig ting å merke seg er at `simP1` har en `while(true)` i sin `lifeCycle()`. Uten denne løkken ville `lifeCycle()` kun blitt kjørt én gang. Dersom `simP2` nå blir aktivert, vil dette objektet ikke utføre noen handling.

B.2.3 Queue

De køene vi benytter i simuleringen er av typen `desmoj.core.simulator.Queue` som holder på objekter av typen `desmoj.core.simulator.Entity`, og `desmoj.core.simulator.ProcessQueue` som holder på objekter av typen `desmoj.core.simulator.SimProcess`. Begge køtypene kan vises i simuleringsrapporten, der man får informasjon om blant annet hvor mange elementer som har ligget i køen, gjennomsnittlig ventetid og hvor mange elementer som ligger i køen ved simuleringens slutt. Lengden på køen kan også hentes ut under kjøring, og statistikkføringen kan også nullstilles under kjøring. De viktigste metodene for køobjektene er:

- `first()` returnerer det elementet som ligger først i køen. For `Queue` returneres et `Entity`-objekt, mens et `SimProcess`-objekt returneres for en `ProcessQueue`.
- `last()` returnerer det siste objektet i køen.
- `isEmpty()` returnerer true dersom køen er tom.
- `insert(Entity/SimProcess)` legger til et nytt element bakerst i køen. Dette elementet er enten av typen `Entity` eller `SimProcess` avhengig av kø-typen.
- `remove(Entity/SimProcess)` fjerner et gitt `Entity`- eller `SimProcess`-objekt, dersom det ligger i køen.
- `succ(Entity/SimProcess)` returnerer etterfølgeren til `Entity`- eller `SimProcess`-objektet. Dersom objektet ikke har noen etterfølger, vil metoden returnere `null`.

B.3 Statistikk, fordelinger og rapporter

I dette underkapitlet vil vi se på ulike statistikkobjekter, sannsynlighetsfordelinger og rapporter som er å finne i simuleringsrammeverket og som er relevant for oss.

B.3.1 Statistikkobjekter

Blant statistikkobjektene vil vi trekke fram `Count` og `Tally`. `Count` egner seg til å ta vare på antall forekomster av en bestemt hendelse. Den kan også ta vare på for eksempel antall millisekunder som er benyttet til ressurskontroll i simuleringen. `Tally` er tar i motsetning til `Count` vare på verdier av desimaltall. I tillegg får man generert standardavvik i forhold til de verdier som objektet har observert gjemmom simuleringen. Under simulering blir følgende metoder kalt:

- `update()` vil telle opp observert verdi på objektet med én.
- `update(long/double verdi)` legger verdi-parameteren til i det observerte antallet. `Count` tar inn verdi av typen `long`, mens `Tally` tar inn verdi av typen `double`.
- `reset()` nullstiller statistikkobjektet.

B.3.2 Sannsynlighetsfordelinger

Pakken `desmoj.core.dist` inneholder ulike sannsynlighetsfordelinger som man setter opp ved å angi enkelte parametere. For å eksempelvis sette opp en normalfordeling vil man angi gjennomsnitt og standardavvik. Etter at en fordeling er satt opp, vil man kunne trekke ut en tilfeldig verdi fra denne ved å kalle metoden `sample()`. Andre nyttige fordelinger som ligger i denne pakken, er eksponential- og poissonfordeling.

B.3.3 Rapportgenerering

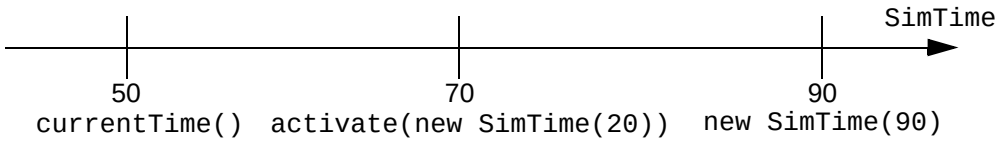
Desmo-J genererer følgende fire rapporter etter at en simulering er gjennomført:

- Resultatrapport med verdier fra statistikkobjekter, fordelinger og køer i simuleringen. Denne rapporten kan brukes for å sammenligne resultatene fra to simuleringer. Rapporten får navnet “eksperimentnavn_report.html”. Et nærmere forklaring av denne rapporten er å finne i kapittel F.1.
- Sporingsrapport viser rekkefølgen objekter blir aktivert. Rapporten får navnet “eksperimentnavn_trace.html”.
- Feilsøkingsrapport inneholder informasjon som kan benyttes under feilsøking. Vi har liten erfaring med denne rapporten, og har i stedet brukt egengenererte utskrifter fra simulatoren under feilsøking. Rapporten får navnet “eksperimentnavn_debug.html”.
- Advarselrapport viser informasjon om for eksempel hvilke prosesser som er forsøkt aktivert på et tidspunkt, mens de allerede ligger i køen for aktivering. Rapporten får navnet “eksperimentnavn_warning.html”.

Ved opprettelse av objekter som deltar i simuleringen, kan man selv bestemme hvilke objekter som skal vises i resultatrapporten og sporingsrapporten. Vår erfaring er at de objektene som skal vises i sporingsrapporten må opprettes i metoden `init()` i modellklassen. Vi har ikke hatt noe problem med at objekter har uteblitt fra resultatrapporten.

B.4 Simuleringens tid, SimTime

Desmo-J opererer med et tidsbegrep `SimTime` som representerer et tidspunkt i simuleringstiden. Eksperimentklassen har en klokke som holder styr på hva simuleringstiden er til enhver tid. Simulatoren har en hendelseskø som er sortert på tiden objektene i køen skal aktiveres. Tiden det sorteres på er simuleringstiden `SimTime`. Et tidspunkt i simuleringstiden kan omformes til en verdi av typen `double`, slik at `new SimTime(10)` vil returnere verdien 10 hvis man ønsker tallverdien. Ønsker man å uttrykke tiden 10 tidsenheter fram i tid, er man nødt til å opprette `SimTime`-objektet `new SimTime(currentTime() + 10)`. Verdien som `currentTime()` returnerer et `SimTime`-objekt som representerer tiden idet metoden kalles. Metoden `activate(SimTime simTime)` i klassen `desmoj.core.simulator.SimProcess` legger automatisk til `currentTime()`, slik at `activate(new SimTime(10))` vil resultere i at objektet aktiveres 10 tidsenheter fram i tid. I figur B.1 er dette illustrert. Vi antar her at tiden akkurat nå er 50.



Figur B.1: SimTime-begrepet

I figuren blir kallene `new SimTime(90)` og `activate(new SimTime(20))` kalt ved tidspunkt 50. Vi kan se at det første metodekallet får verdien 90 i simuleringstiden, mens det andre metodekallet får verdien 70.

Vedlegg C: Kjøring av simulatoren

I dette vedlegget beskriver vi hva som trengs for å kunne kjøre simulatoren, og hvordan simulatoren startes.

C.1 Hva man trenger

For å kjøre simulatoren trenger man følgende:

- Java Runtime Environment 1.5.0. Simulatoren er kompilert for JRE 1.5.0.
- Desmo-J versjon 2.0.1 (gitt ut i februar 2005).

JRE kan lastes ned fra [16] og Desmo-J kan hentes fra [17].

C.2 Hvordan simulatoren startes

Etter at punktene under kapittel C.1 er oppfylt, skal simulatoren kunne startes ved å gjøre følgende:

- Kompilere kildekoden
- Sette parameterne slik man ønsker i alle `.properties`-filene
- Starte klassen `Simulator`

Avhengig av hva slags utviklingsmiljø man benytter, må classpath oppdateres slik at man finner Desmo-J-pakken under kompileringen.

Etter kjøring vil rotkatalogen inneholde filer med commit- og aborttider fra kjøringen i tillegg til rapportene som Desmo-J genererer automatisk.

Vedlegg D: Verifisering av transaksjons-utførelse i simulatoren

ClustRa utfører transaksjoner slik vi har beskrevet i kapittel 6.4 og som beskrevet i [5]. For at vår simulering skal gi pålitelige resultater, er det viktig at transaksjonsutførelsen i simulatoren er i henhold til denne utførelsen i ClustRa.

Vi har valgt å gjøre tre prøve-simuleringer for å verifisere utførelsen her; transaksjoner med henholdsvis én og fire oppdateringer og en transaksjon med fire oppdateringer som aborterer. Systemet vi kjører simuleringen under består av 4 noder, en sannsynlighet for disk på 0 og uten ressurskontroll. Vi har også utelatt prosessering av refragmentering fra denne simuleringen, da dette ikke har innvirkning på den delen av transaksjonsutførelsen som er relevant for vår simulator, i forhold til beskrivelsen fra kapittel 3.1. Vi har valgt dette for å få mest mulig korrekt testing av selve transaksjonsutførelsen i forhold til hvordan det utføres i ClustRa og hvilke testdata vi har tilgjengelig fra testing av selve ClustRa.

D.1 Én oppdatering

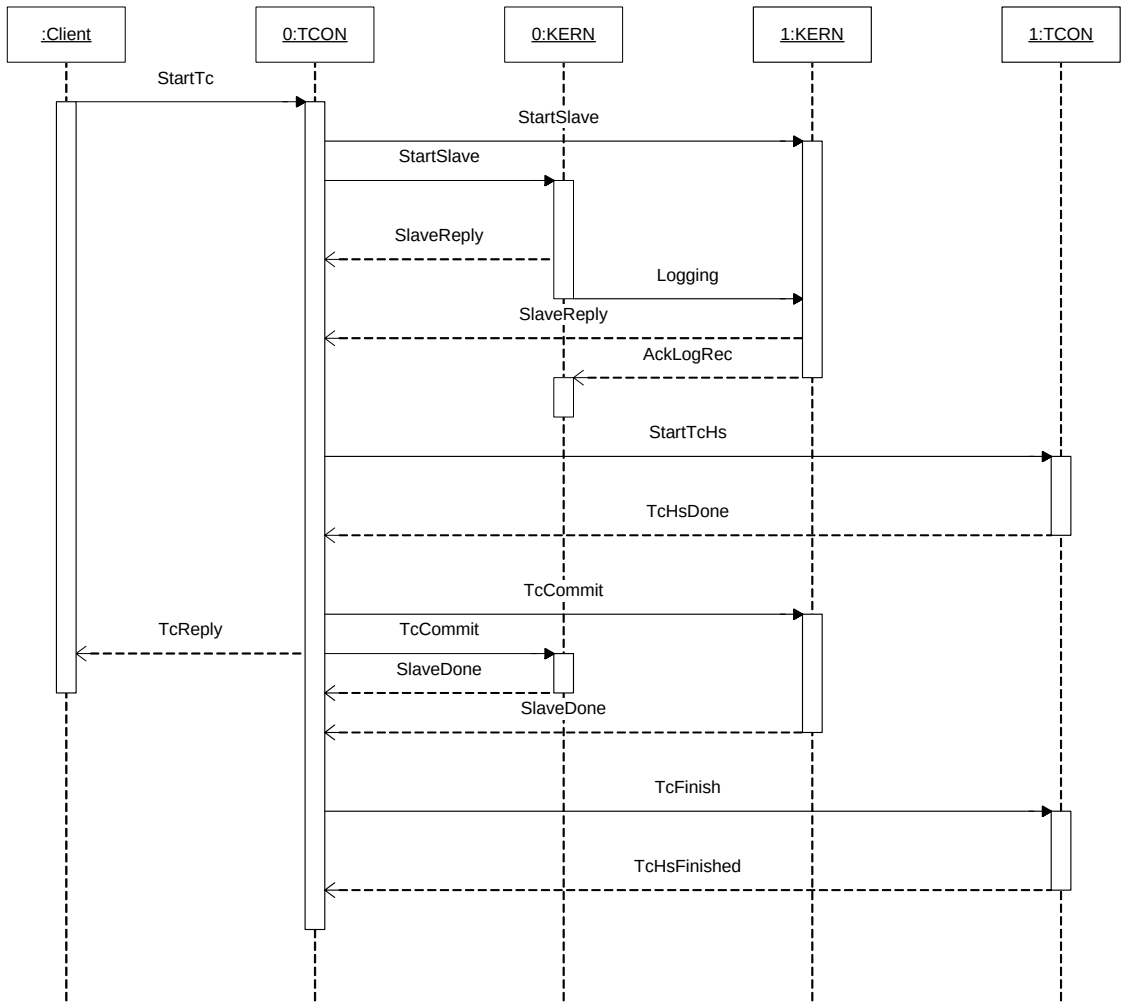
I kode-eksempel D.1 har vi vist utskriften av en kjøring av én transaksjon som utfører én oppdatering. Vi ser her på transaksjonen "`UpdateTrans#1`". I utskriften angir verdien til `time` simulatortiden når den angitte meldingen ble sendt.

Ved å sammenligne denne utskriften med figur D.1 ser vi at meldingene kommer i samme logiske rekkefølge. En nærmere forklaring av figur D.1 er å finne i kapittel 3.3.1, hvor figur 3.2 er den samme figuren som figur D.1.

Det er her verdt å merke seg at utskriften "`UpdateTrans#1, time 3.943: COMMIT: @node0, transTime = 3.943`" tilsvarer commit og at meldingen *TcReply* sendes til klienten. Vi ser at denne fullføringen kommer før meldingen *AckLogRec*, noe som er omvendt i forhold til rekkefølgen i figur D.1. Dette er likevel riktig, da disse to meldingene sendes uavhengig av hverandre og i dette tilfellet fra to forskjellige noder og følgelig blir rekkefølgen i forhold til simuleringstiden tilfeldig.

```
UpdateTrans#1, time 0.000: StartTc sent from Client0#1
UpdateTrans#1, time 0.756: StartHSSlave sent from PTrolThread @0#1
UpdateTrans#1, time 1.506: StartPSlave sent from PTrolThread @0#1
UpdateTrans#1, time 2.402: PSlaveReply sent from PSlaveThread @3#1
UpdateTrans#1, time 2.636: Logging sent from PSlaveThread @3#1
UpdateTrans#1, time 2.944: HSSlaveReply sent from HSSlaveThread @2#1
UpdateTrans#1, time 3.250: StartTcHs sent from PTrolThread @0#1
UpdateTrans#1, time 3.538: TcHsDone sent from HSTrolThread @1#1
UpdateTrans#1, time 3.943: COMMIT: @node0, transTime = 3.943
UpdateTrans#1, time 4.230: AckLogRec sent from HSSlaveThread @2#1
UpdateTrans#1, time 4.320: TcCommit sent from PTrolThread @0#1
UpdateTrans#1, time 4.499: TcCommit sent from PTrolThread @0#1
UpdateTrans#1, time 4.627: SlaveDone sent from PSlaveThread @3#1
UpdateTrans#1, time 4.819: SlaveDone sent from HSSlaveThread @2#1
UpdateTrans#1, time 5.608: TcFinish sent from PTrolThread @0#1
UpdateTrans#1, time 5.912: TcHsFinished sent from HSTrolThread @1#1
```

Kode-eksempel D.1: Transaksjonstskrift ved én oppdatering



Figur D.1: Transaksjonsprosessering av én oppdatering i ClustRa

D.2 Fire oppdateringer

I kode-eksempel D.2 har vi vist en annen kjøring av en transaksjon. Her har vi en transaksjon med fire oppdateringer. Vi følger også her transaksjonen "`UpdateTrans#1`" gjennom en fullstendig utførelse hvor vi ser når de ulike meldingene blir sendt i simuleringstiden.

Vi ser her at det sendes tre meldinger av både *StartPSlave* og *StartHSSlave*. Det medfører at transaksjonen involverer tre forskjellige noder i utførelsen mot primærnodene og at en node dermed gjør to av oppdateringene. Videre ser vi hvordan `PTrolThread` sender *StartTcHs* når den har mottatt tre *PSlaveReply* og tre *HSSlaveReply*. Etter at svaret *TcHsDone* er mottatt regnes transaksjonen som fullført. *TcReply* sendes da til klienten mens *TcCommit* sendes seks ganger, en gang for hver slavetråd. Transaksjonen avsluttes med å sende *TcFinish* til `HSTrolThread` når alle *SlaveDone* er mottatt. `HSTrolThread` svarer da med *TcHsFinished*.

```
UpdateTrans#1, time 0.000: StartTc sent from Client0#1
UpdateTrans#1, time 1.466: StartPSlave sent from PTrolThread @0#1
UpdateTrans#1, time 2.388: StartHSSlave sent from PTrolThread @0#1
UpdateTrans#1, time 2.982: StartHSSlave sent from PTrolThread @0#1
UpdateTrans#1, time 3.161: StartPSlave sent from PTrolThread @0#1
UpdateTrans#1, time 3.650: StartPSlave sent from PTrolThread @0#1
UpdateTrans#1, time 4.214: StartHSSlave sent from PTrolThread @0#1
UpdateTrans#1, time 4.783: Logging sent from PSlaveThread @0#1
UpdateTrans#1, time 6.060: Logging sent from PSlaveThread @3#1
UpdateTrans#1, time 6.774: PSlaveReply sent from PSlaveThread @3#1
UpdateTrans#1, time 6.963: PSlaveReply sent from PSlaveThread @2#1
UpdateTrans#1, time 7.111: PSlaveReply sent from PSlaveThread @0#1
UpdateTrans#1, time 7.203: HSSlaveReply sent from HSSlaveThread @1#1
UpdateTrans#1, time 7.286: AckLogRec sent from HSSlaveThread @1#1
UpdateTrans#1, time 7.378: HSSlaveReply sent from HSSlaveThread @2#1
UpdateTrans#1, time 7.791: Logging sent from PSlaveThread @2#1
UpdateTrans#1, time 8.097: HSSlaveReply sent from HSSlaveThread @3#1
UpdateTrans#1, time 8.847: StartTcHs sent from PTrolThread @0#1
UpdateTrans#1, time 8.853: AckLogRec sent from HSSlaveThread @3#1
UpdateTrans#1, time 9.173: TcHsDone sent from HSTrolThread @1#1
UpdateTrans#1, time 9.392: AckLogRec sent from HSSlaveThread @2#1
UpdateTrans#1, time 9.706: TcCommit sent from PTrolThread @0#1
UpdateTrans#1, time 9.860: TcCommit sent from PTrolThread @0#1
UpdateTrans#1, time 9.928: COMMIT: @node0, transTime = 9.928
UpdateTrans#1, time 9.993: TcCommit sent from PTrolThread @0#1
UpdateTrans#1, time 10.141: TcCommit sent from PTrolThread @0#1
UpdateTrans#1, time 10.249: SlaveDone sent from HSSlaveThread @1#1
UpdateTrans#1, time 10.340: SlaveDone sent from PSlaveThread @2#1
UpdateTrans#1, time 10.405: SlaveDone sent from PSlaveThread @0#1
UpdateTrans#1, time 10.496: SlaveDone sent from HSSlaveThread @2#1
UpdateTrans#1, time 10.775: TcCommit sent from PTrolThread @0#1
UpdateTrans#1, time 11.536: TcCommit sent from PTrolThread @0#1
UpdateTrans#1, time 11.648: SlaveDone sent from HSSlaveThread @3#1
UpdateTrans#1, time 12.021: SlaveDone sent from PSlaveThread @3#1
UpdateTrans#1, time 12.436: TcFinish sent from PTrolThread @0#1
UpdateTrans#1, time 12.867: TcHsFinished sent from HSTrolThread @1#1
```

Kode-eksempel D.2: Transaksjonsutskrift ved fire oppdateringer

D.3 Abort-prosessering

Her har vi simulert en kjøring med en transaksjon som gjør fire oppdateringer hvor abort skjer. Det er verdt å merke seg at prosesseringen vi har gjort her er med spesielle parametere for å framprovosere abort, slik at tiden for abort ikke er realistisk. Meldingstidene er fortsatt korrekt. Utskrift fra denne kjøringen er å finne i kode-eksempel D.3.

Her ser vi at tre ulike noder deltar i transaksjonsprosesseringen, og at det derfor blir tre *StartPSlave* og tre *StartHSSlave*. `PTrolThread` mottar to *PSlaveReply*, men før resten av meldingene mottas har timeout-tiden til transaksjonen gått ut og abort må gjennomføres. Dette medfører at *TcAbort* sendes fra `PTrolThread`. Dette gir utskriften "`UpdateTrans#1, time 5.583: ABORT @node0, abortTime = 5.583`". Vi ser også at de andre *SlaveReply*

kommer, men altså for sent til at transaksjonen kan gjennomføres. Vi ser videre at transaksjonsprosesseringen avsluttes når alle seks *SlaveAbort* er mottatt fra de ulike slavetrådene. Når *AckLogRec* mottas har ingen betydning for abortprosesseringen, slik at vi ser bort i fra disse meldingene.

```
UpdateTrans#1, time 0.000: StartTc sent from Client0#1
UpdateTrans#1, time 1.026: StartPSlave sent from PTrolThread @0#1
UpdateTrans#1, time 1.158: StartHSSlave sent from PTrolThread @0#1
UpdateTrans#1, time 2.247: StartPSlave sent from PTrolThread @0#1
UpdateTrans#1, time 2.471: StartHSSlave sent from PTrolThread @0#1
UpdateTrans#1, time 3.514: StartHSSlave sent from PTrolThread @0#1
UpdateTrans#1, time 3.683: PSlaveReply sent from PSlaveThread @0#1
UpdateTrans#1, time 4.421: StartPSlave sent from PTrolThread @0#1
UpdateTrans#1, time 4.689: Logging sent from PSlaveThread @2#1
UpdateTrans#1, time 4.870: PSlaveReply sent from PSlaveThread @2#1
UpdateTrans#1, time 5.583: ABORT  @node0, abortTime = 5.583
UpdateTrans#1, time 6.915: Logging sent from PSlaveThread @0#1
UpdateTrans#1, time 9.401: TcAbort sent from PTrolThread @0#1
UpdateTrans#1, time 9.749: HSSlaveReply sent from HSSlaveThread @3#1
UpdateTrans#1, time 9.814: TcAbort sent from PTrolThread @0#1
UpdateTrans#1, time 10.216: AckLogRec sent from HSSlaveThread @3#1
UpdateTrans#1, time 10.328: TcAbort sent from PTrolThread @0#1
UpdateTrans#1, time 10.453: PSlaveReply sent from PSlaveThread @1#1
UpdateTrans#1, time 10.518: TcAbort sent from PTrolThread @0#1
UpdateTrans#1, time 11.018: Logging sent from PSlaveThread @1#1
UpdateTrans#1, time 11.542: TcAbort sent from PTrolThread @0#1
UpdateTrans#1, time 12.742: TcAbort sent from PTrolThread @0#1
UpdateTrans#1, time 12.883: HSSlaveReply sent from HSSlaveThread @1#1
UpdateTrans#1, time 12.966: AckLogRec sent from HSSlaveThread @1#1
UpdateTrans#1, time 13.140: SlaveAbort sent from HSSlaveThread @3#1
UpdateTrans#1, time 13.205: SlaveAbort sent from HSSlaveThread @0#1
UpdateTrans#1, time 13.270: SlaveAbort sent from PSlaveThread @0#1
UpdateTrans#1, time 13.361: SlaveAbort sent from PSlaveThread @2#1
UpdateTrans#1, time 13.426: HSSlaveReply sent from HSSlaveThread @0#1
UpdateTrans#1, time 13.517: SlaveAbort sent from HSSlaveThread @1#1
UpdateTrans#1, time 13.608: SlaveAbort sent from PSlaveThread @1#1
UpdateTrans#1, time 14.483: AckLogRec sent from HSSlaveThread @0#1
```

Kode-eksempel D.3: Transaksjonsutskrift ved abort-prosessering

Vedlegg E: Køteori

Køteori omhandler ulike systemer hvor man har en eller flere servere eller tjenester som en eller flere kunder ankommer. Ved ankomst vil kundene enten gå direkte til ønsket tjeneste eller vente i kø til tjenesten er ledig. I slike systemer er det ut i fra køteorien blant annet interessant å finne ut det gjennomsnittlige antall kunder i systemet eller i køen og gjennomsnittlig tid hver kunde er i systemet eller i køen.

Vi velger å beskrive køteori på et generelt grunnlag her siden køen av transaksjoner på hver enkelt node, behandlinsratene, ankomstratene og andre viktige køparametere er sentralt både for å kunne forstå og se løsninger på vår problemstilling. Vi ser kun på de aspekter ved køteori som er relevante i forhold til vårt simuleringsarbeid og viser til [10] for mer utfyllende beskrivelse.

I dette vedlegget vil vi først se nærmere på noen viktige definisjoner, før vi ser på Littles køformel og til slutt noen interessante og relevante køsystemer. Alle formler vi presenterer vil navngis på formen (3.x) for å lettere kunne referere til dem.

Vårt studium av køteori er basert på [10] og [18].

E.1 Definisjoner og begrep

Før vi går nærmere inn på køteorien, vil vi definere og forklare noen sentrale begreper og størrelser. Disse begrepene er listet opp i tabell E.1. Under følger en nærmere forklaring og utledning av noen av begrepene. De størrelsene vi ikke ser nærmere på er kun en direkte definisjon og kan følgelig ikke utledes.

Tabell E.1: Definisjoner av købegrep

Begrep	Forklaring og definisjon
l	Gjennomsnittlig ankomstrate
l_a	Gjennomsnittlig ankomstrate for de som kommer inn i systemet
m	Gjennomsnittlig behandlingsrate
P_n	Sannsynligheten for n kunder i systemet
L	Gjennomsnittlig antall kunder i systemet
L_Q	Gjennomsnittlig antall kunder i kø
W	Gjennomsnittlig tid en kunde bruker i systemet
W_Q	Gjennomsnittlig tid en kunde bruker i kø

λ_a er ankomstraten til de som faktisk ankommer et køsystem. Hvis vi setter N(t) til å være antall kunder som har ankommet ved tiden t, vil λ_a være definert ved

$$\lambda_a = \lim_{t \to \infty} \frac{N(t)}{t} \qquad \text{Ligning 9.1}$$

Ankomstraten vil altså uttrykke hvor mange som ankommer systemet per tidsenhet. Denne raten er ofte antatt Poisson-fordelt om man ikke har en fast ankomstrate.

P_n er sannsynligheten for at systemet har n kunder. Hvis vi setter X(t) til å være antall kunder i systemet ved tid t, og n større eller lik 0, får vi

$$P_n = \lim_{t \to \infty} P\{X(t) = n\} \qquad \text{Ligning 9.2}$$

På denne måten får vi at P_0 er sannsynligheten for at det er ingen i systemet.

L er det gjennomsnittlige antall kunder i systemet. L kan finnes ved

$$L = \sum_{n=0}^{\infty} nP_n \qquad \text{Ligning 9.3}$$

Denne formelen er basert på det generelle utrykket for forventningsverdi, som i det diskrete tilfellet er

$$E[x] = \sum_{x=0}^{\infty} xP(x) \qquad \text{Ligning 9.4}$$

for en stokastisk variabel x.

E.2 Littles køformel

Littles kø-formel er av mange omtalt som selve kjernen i køteori. Littles køformel er definert som

$$L = \lambda_a W \qquad \text{Ligning 9.5}$$

Formelen uttrykker altså en sammenheng mellom antall kunder i systemet, ankomstraten og tida i systemet.

Vi vil bruke denne formelen i neste underkapittel hvor vi presenterer det generelle køsystemet som brukes i vår simulering av ClustRa.

E.3 Singeltjener-system med uendelig køkapasitet

Dette er en køtype der $\lambda_a = \lambda$ siden systemet har uendelig kø og følgelig vil alle prosesser som ønsker å ankomme faktisk ankomme.

Vi vil i dette underkapitlet først presentere en del generelle formler og uttrykk, før vi kommer med regneeksempler for å tydeliggjøre viktige prinsipper ved køteori.

Formler og uttrykk

For køsystemer generelt, gjelder det over tid at raten en prosess ankommer tilstand n med er det samme som raten prosessen forlater tilstand n med. Tilstand n er her at det er n prosesser i kø-systemet. Hvis vi ser på systemet for tilstand 0, får vi

$$\lambda P_0 = \mu P_1 \qquad \text{Ligning 9.6}$$

som sier at avgangsraten til tilstand 0 (den generelle ankomstraten ganget med sannsynligheten for å være i tilstand 0, noe som gjør at man går fra 0 til 1) er det samme som ankomstraten til tilstand 0 (den generelle behandlingsraten ganget med sannsynligheten for å være i tilstand 1, noe som gjør at man går fra 1 til 0).

Hvis vi resonnerer på samme måten for de andre tilstandene fra og med 2, vil ligningen for en tilstand n være

$$(\lambda + \mu)P_n = \lambda P_{n-1} + \mu P_{n+1} \qquad \text{Ligning 9.7}$$

siden venstresiden her gir den totalte raten for å forlate tilstand n (enten ankommer en ny prosess eller forlater en prosess) mens høyresiden angir raten for å ankomme tilstand n (en prosess ankommer fra tilstand n-1 eller en prosess forlater tilstand n+1).

Ved å bruke loven om total sannsynlighet, som sier

$$1 = \sum_{n=0}^{\infty} P_n \qquad \text{Ligning 9.8}$$

vil man etter noe mellomregning ut i fra ligningene over finne

$$P_0 = 1 - \frac{\lambda}{\mu} \qquad \text{Ligning 9.9}$$

$$P_n = \left(\frac{\lambda}{\mu}\right)^n \left(1 - \frac{\lambda}{\mu}\right) \qquad \text{Ligning 9.10}$$

Ut i fra formelen for det gjennomsnittlige antallet i systemet presentert i kapittel E.1, vil vi for et slikt system få

$$L = \sum_{n=0}^{\infty} nP_n = \sum_{n=0}^{\infty} n\left(\left(\frac{\lambda}{\mu}\right)^n\left(1-\frac{\lambda}{\mu}\right)\right) = \frac{\lambda}{\mu-\lambda} \qquad \text{Ligning 9.11}$$

Hvis vi nå velger å se kun på køen, vil vi i denne situasjonen ha at

$$W_Q = W - E[S] \qquad \text{Ligning 9.12}$$

siden W er den totale tiden en kunde bruker i systemet og E[S] er den forventede behandlingstiden til systemets server. Om vi regner videre med dette uttrykket får vi

$$W_Q = W - \frac{1}{\mu} = \frac{\lambda}{\mu(\mu-\lambda)} \qquad \text{Ligning 9.13}$$

siden

$$W = \frac{L}{\lambda} = \frac{1}{\mu-\lambda} \qquad \text{Ligning 9.14}$$

ut i fra Littles køformel og ligning 9.11.

Littles køformel ble presentert i kapittel E.2. Denne formelen vil også gjelde for parameterne som kun ser på kø, med følgende sammenheng

$$L_Q = \lambda W_Q \qquad \text{Ligning 9.15}$$

Ved å nå sette inn uttrykket for W_Q fra ligning 9.13 vil man finne et uttrykk for L_Q.

I uttrykkene presentert over er det viktig å merke seg størrelsen på forholdet λ/μ. Blir dette tallet større eller lik 1, vil med andre ord ankomstraten være større eller lik behandlingsraten. Følgelig vil man etter en tid få et system med uendelig lang kø. Derfor må dette forholdstallet alltid være under 1 for denne typen køsystem. Kommer derimot dette tallet under 1, men nærme 1, vil en endring i ett av tallene ha stor påvirkning på det gjennomsnittlige antall i køsystemet.

Regneeksempler

To mindre regneeksempler vil illustrere hvordan forholdet mellom behandlingsrate og ankomstrate utvikler seg.

Anta først at vi har et køsystem der kunder ankommer hvert 12. minutt, altså en ankomstrate på 1/12. Det samme systemet har en behandlingstid på 4 minutter, noe som gir behandlingsrate på 1/4. Bruker vi formelen for L fra ligning 9.11 vil dette gi gjennomsnittlig 1/2 kunder i systemet. Hvis vi nå øker ankomstraten med 20 % til 1/10, vil L endre seg til 2/3, altså en økning på 33 %.

Anta nå at vi har et annet kø-system der kunder ankommer i gjennomsnitt hvert 6. minutt. Følgelig er ankomstraten 1/6. Videre kan vi anta at systemet behandler ferdig en kunde i løpet av fire minutter i snitt, noe som gir en behandlingsrate på 1/4. Setter vi dette inn i uttrykket for L på samme måten, vil vi få et gjennomsnitt på 2 personer i systemet. La oss nå anta at ankomstraten øker til 1/5, altså en økning på 20 prosent her også. Regner vi ut L på nytt i denne situasjonen, vil det gjennomsnittlige antallet kunder ha økt til 4. En økning på 20% i ankomstraten medfører altså en dobling i det gjennomsnittlige antallet kunder i systemet i dette tilfellet, mens samme økning i forrige eksempel gav en endring på 33%. Tilsvarende vil man se om man studerer W, den gjennomsnitlige tida en kunde bruker i systemet. Dette velger vi ikke å se på her, da man finner størrelsen på W enkelt ved å bruke Littles kø-formel, presentert i kapittel E.2.

Vedlegg F: Genererte rapporter

Simulatoren genererer en del rapporter. I dette vedlegget forklarer vi hva som står i de ulike rapportene og hvordan de kan brukes. Vi har valgt å dele inn rapportene i to kategorier. Den første kategorien inneholder rapporter som simuleringsrammeverket Desmo-J genererer for oss, mens den andre kategorien inneholder tekstfiler som vi selv har generert.

F.1 Rapporter fra Desmo-J

Rapportgenerering i Desmo-J er omtalt i kapittel B.3.3. Der er de fire rapportene resultatrapport, sporingsrapport, feilsøkingsrapport og advarselsrapport omtalt. Vi velger å beskrive resultatrapporten nærmere i dette vedlegget siden denne brukes under normal kjøring.

Et eksempel på en resultatrapport er vist i figur F.1 på side 111, og rapporten er delt inn i følgende fem deler:

- Beskrivelse
- Statistikk over responstid
- Opptellinger
- Køer
- Fordelinger

Beskrivelsen av simuleringen inneholder en oversikt over de viktigste parameterne og hvilken verdi de hadde under kjøringen. Parameterne er delt inn i systemparametere og ressurskontroll-parametere.

I rapportens andre del kan man se statistikk over hvilken responstid klienten har opplevd fra systemet. Responstidstatistikken er vist for hver node i systemet. I denne delen kan man se om en node har brukt lengre tid per transaksjon enn de andre nodene.

Gjennom simuleringen blir blant annet antall transaksjonsmeldinger, ressurskontrollmeldinger, avbrudte transaksjoner og fullførte transaksjoner telt opp. De endelige verdiene blir vist i resultatrapportens tredje del. Opptellingene er sortert per node i systemet. Det er verdt å merke seg at tidene som vises for bruk av CPU, ressurskontroll, tråder og timere alle viser verdier i antall mikrosekunder, mens tiden som disken har brukt vises i antall millisekunder.

I den fjerde delen av rapporten finnes en oversikt over alle køene i simulatoren, og hvordan bruken av disse køene har vært gjennom simuleringen. Her kan man se hvor mange som har vært i køen, antall som er i køen ved simuleringens slutt, det maksimale antallet som har vært i køen, gjennomsnittlig antall i køen, hvor mange som ankom da køen var tom og til slutt gjennomsnittlig ventetid i køen. Dersom køen har en begrensning på hvor lang den kan være, vil dette også vises i denne rapporten. I tillegg får man da også opp hvor mange som har blitt avvist, det vil si hvor mange som ankom mens køen hadde sin maksimale lengde, dersom dette er angitt i systemet man simulerer (denne funksjonen bruker ikke vi).

Til slutt i rapporten kan man se en oversikt over alle fordelingene som brukes under simuleringen. For hver fordeling ser man hvilken type fordelingen er og hvilke parametere denne har.

F.2 Tekstfiler

Gjennom simuleringen genererer vi en del tekstfiler som brukes blant annet for enkelt å kunne tegne grafer og se på viktige tall fra simuleringene. Vi beskriver dem i de kommende delkapitler.

F.2.1 Responstider

Den første typen filer inneholder responstidene for hver node og for systemet totalt sett. Filene *CommitLog<nodenummer>.txt* forteller hvordan responstiden oppleves på en bestemt node, mens filen *GlobalCommitTimes.txt* sier hvordan responstidene har vært i systemet som helhet. Responstidene lagres etter hvert som de oppstår i simulatoren, og filene skrives etter endt simulering. Filene benyttes til å se variasjonen i responstiden gjennom simuleringen.

F.2.2 Tidspunkt i simulering

I den andre typen filer lagres tidspunktet i simuleringen når abort, commit og avvisning forekommer. Tidene finnes i henholdsvis *GlobalAbortSimTimes.txt*, *GlobalCommitSimTimes.txt* og *GlobalRejectSimTimes.txt*. Disse filene benyttes for å se i hvilken del av simuleringen de fleste commit, abort og avvisningene inntreffer. Vi får også et inntrykk av hvordan antallet av hendelsene utvikler seg.

F.2.3 Statistikk

Simulatoren har i tillegg en fil ved navn *OverallSimulationResults.txt* som er vist i kode-eksempel F.1. Denne filen inneholder først en oppsummering av verdien på parameterne ressurskontrolltype, tid mellom transaksjoner og sannsynlighet for diskavbrudd. Etter parameterne vises statistikk fra simuleringen som benyttes til å sammenligne de ulike kjøringene.

```
distributedResourceControlType: 1
mtBetweenArrivals: 30
probPageFault: 0.4

Refrag completed at: 25094.751765870038
Average commit time: 45.4325584636263
Number of rejected: 60
Number of aborted: 0
Number of committed: 734
Number of transMsgs: 26197
Number of RCMsgs: 3752
```

Kode-eksempel F.1: Innhold i OverallSimulationResults.txt

Clustra_Simulation Report

Model Clustra Simulator

Description

A simulation of ClustRa, with the following parameters:

System settings:

mtBetweenArrivals	15ms
probPageFault	0.2
distributedResourceControlType	3
stopTime	20000.0

RC settings:

CPUBusyBoundary	0.8
numberOfCPUValues	3
DiskBusyBoundary	10.0
numberOfDiskValues	1
interval_rc	5.0

Report drawn at 20000.0. Last reset at 0.0.

Tallies

Title	(Re)set	Obs	Mean	Std.Dev	Min	Max
TransTimeTally @node0	0.0	720	18.09532	7.60652	6.65849	46.10848
TransTimeTally @node1	0.0	736	18.10364	7.19387	6.75376	45.20852
TransTimeTally @node2	0.0	741	17.92655	7.07386	6.51286	53.53685
TransTimeTally @node3	0.0	714	12.17949	4.72978	5.89289	42.00887

Counts

Title	(Re)set	Observations	Min	Max
Total CPU time @ node0	0.0	11281147	0	11281147
Total RC time @ node0	0.0	2692706	0	2692706

Disk time all disks, node0	0.0	11849	0	11849
PTrolThreadTimeCountTotal @ node0	0.0	1354282	0	1354282
HSTrolThreadTimeCountTotal @ node0	0.0	280658	0	280658
PSlaveThreadTimeCountTotal @ node0	0.0	2132182	0	2132182
HSSlaveThreadTimeCountTotal @ node0	0.0	1604039	0	1604039
RefragThreadCount @ node0	0.0	0	0	0
MsgInterpretTimeCountTotal @ node0	0.0	2663432	0	2663432
TimerRunningTimeCountTotal @ node0	0.0	455	0	455
InternalRCTimeCountTotal @ node0	0.0	2329377	0	2329377
DistributedRCTimeCountTotal @ node0	0.0	363329	0	363329
CommittedTransCount @ node0	0.0	720	0	720
AbortedTransCount @ node0	0.0	0	0	0
RejectedTransCount @ node0	0.0	608	0	608
TransMsgCount @ node0	0.0	25251	0	25251
RCMsgCount @ node0	0.0	1328	0	1328
Disk Work Count @ node0, disk0	0.0	11849	0	11849
Disk WorkTotal Count @ node0, disk0	0.0	11849	0	11849
Disk Idle Count @ node0, disk0	0.0	7892	0	7892
Total CPU time @ node1	0.0	11303484	0	11303484
Total RC time @ node1	0.0	2694322	0	2694322
Disk time all disks, node1	0.0	11860	0	11860
PTrolThreadTimeCountTotal @ node1	0.0	1382585	0	1382585
HSTrolThreadTimeCountTotal @ node1	0.0	274342	0	274342
PSlaveThreadTimeCountTotal @ node1	0.0	2095805	0	2095805
HSSlaveThreadTimeCountTotal @ node1	0.0	1630460	0	1630460
RefragThreadCount @ node1	0.0	0	0	0
MsgInterpretTimeCountTotal @ node1	0.0	2672328	0	2672328
TimerRunningTimeCountTotal @ node1	0.0	907	0	907
InternalRCTimeCountTotal @ node1	0.0	2330335	0	2330335
DistributedRCTimeCountTotal @ node1	0.0	363987	0	363987
CommittedTransCount @ node1	0.0	736	0	736
AbortedTransCount @ node1	0.0	0	0	0
RejectedTransCount @ node1	0.0	591	0	591
TransMsgCount @ node1	0.0	25427	0	25427

CommittedTransCount @ node3	0.0	714	0	714
AbortedTransCount @ node3	0.0	0	0	0
RejectedTransCount @ node3	0.0	614	0	614
TransMsgCount @ node3	0.0	24524	0	24524
RCMsgCount @ node3	0.0	6220	0	6220
Disk Work Count @ node3, disk0	0.0	11320	0	11320
Disk WorkTotal Count @ node3, disk0	0.0	11320	0	11320
Disk Idle Count @ node3, disk0	0.0	8396	0	8396

Queues

Title	Qorder	(Re)set	Obs	QLimit	Qmax	Qnow	Qavg.	Zeros	avg.Wait	refus.
ProcessQueue CPU@node0	FIFO	0.0	14862	unlimit.	1	0	0.33477	0	0.4505	0
ProcessQueue CPU@node1	FIFO	0.0	14895	unlimit.	1	0	0.3364	0	0.45169	0
ProcessQueue CPU@node2	FIFO	0.0	14378	unlimit.	1	0	0.32498	0	0.45206	0
ProcessQueue CPU@node3	FIFO	0.0	16977	unlimit.	1	0	0.42027	0	0.4951	0
JobQueue@disk0, node0	FIFO	0.0	1186	unlimit.	6	2	0.43388	507	7.29764	0
MessageQueue @ node 0	FIFO	0.0	26579	unlimit.	13	0	0.71834	291	0.54053	0
TimerQueue @ node 0	FIFO	0.0	2864	unlimit.	4	2	1.84153	238	12.85814	0
JobQueue@disk0, node1	FIFO	0.0	1187	unlimit.	7	0	0.43249	476	7.28708	0
MessageQueue @ node 1	FIFO	0.0	26755	unlimit.	13	0	0.74011	265	0.55325	0
TimerQueue @ node 1	FIFO	0.0	2891	unlimit.	4	2	1.86889	258	12.92126	0
JobQueue@disk0, node2	FIFO	0.0	1134	unlimit.	4	1	0.31057	519	5.47625	0
MessageQueue @ node 2	FIFO	0.0	26224	unlimit.	12	0	0.60169	293	0.45889	0
TimerQueue @ node 2	FIFO	0.0	2901	unlimit.	5	3	1.88011	274	12.95237	0

TimerQueue @ node 2	FIFO	0.0	2901	unlimit.	5	3	1.88011	274	12.95237	0
JobQueue@disk0, node3	FIFO	0.0	1133	unlimit.	5	1	0.32466	539	5.72339	0
MessageQueue @ node 3	FIFO	0.0	30744	unlimit.	17	0	1.73934	168	1.1315	0
TimerQueue @ node 3	FIFO	0.0	2368	unlimit.	4	1	1.72916	137	14.5891	0

Distributions

Title	(Re)set	Obs	Type	Parameter 1	Parameter 2	Seed
Trans Arrival Dist	0.0	0	Poisson	15.0		71529105
Trans Arrival Dist	0.0	0	Poisson	15.0		91080577
Trans Arrival Dist	0.0	0	Poisson	15.0		43946618
Trans Arrival Dist	0.0	0	Poisson	15.0		44944377

created using DESMO-J Version 1.6 at Fri May 27 09:16:05 CEST 2005 - DESMO-J is licensed under GNU GPL

Figur F.1: Eksempel på Clustra_Simulation_report.html