NTNU
Det skapende universitet

Skalering av leseoperasjoner med Apache Derby

Snorre Visnes

Master i datateknikk
Oppgaven levert: Juni 2007
Hovedveileder: Svein-Olaf Hvasshovd, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

Det er i en rekke anvendelser behov for replisering av data hovedsakelig motivert ut fra fordeling av leselast på et større antall noder. Dette er anvendelser hvor endringsraten er liten.

Apache Derby har pr i dag ingen støtte for replisering eller lastbalansering.

Det er ønskelig å endre JDBC driveren for å muliggjøre bygging av et cluster med Derby som kan støtte et svært høyt volum leseoperasjoner. Det er videre ønskelig at oppgaven ser på mulighetene for en slik driver til å benytte Derbys innebygde XA-støtte til å transaksjonelt oppdatere data på servernodene i et slikt cluster.

Dersom tiden tillater det, implementer og test en prototype som har den aktuelle funksjonaliteten.

Oppgaven gitt: 16. januar 2007
Hovedveileder: Svein-Olaf Hvasshovd, IDI

Sammendrag

Oppgaven tar for seg hvordan det er mulig å lage et cluster basert på Derby som støtter et høyt volum av lesetransaksjoner. Skriving er ikke i fokus ytelsesmessig, men er mulig gjennom Derbys støtte for XA.

Det faktum at XA er et verktøy for å gjennomføre 2-fase commit, ikke replisering, gjør at skriving kun er mulig for administrator. Hovedsakelig grunnet manglende sekvensering av transaksjoner, samt mangel på automatisk opprydning etter feilede transaksjoner.

Testing viser at skaleringsgraden for et slikt system er på 100%. Det er ingen sammenkobling mellom servernoder, og dermed ingen øvre grense for antall noder.

Det at det ikke er noen sammenkobling mellom servernoder gjør at disse kan spres geografisk. Sammen med en fail-over mekanisme i klienten kan dette systemet oppnå høy tilgjengelighet ved lesing.

Forord

Denne oppgaven er endelig innlevering i ”TDT4900 - Datateknikk og informasjonsvitenskap, masteroppgave” ved Institutt for datateknikk og informasjonsvitenskap ved NTNU.

Målet var å se hvordan det er mulig å lage et cluster basert på Derby som støtter svært høy leselast. Oppgaven er utformet i samarbeid med Sun Microsystems i Trondheim.

Jeg ønsker å takke professor Svein-Olaf Hvasshovd for svært god veiledning, og Sun Microsystems for god hjelp i oppstartsfasen.

Trondheim, 11. Juni 2007

_____

Snorre Visnes

Innholdsfortegnelse

Figurliste

1 Oppgavetekst

Det er i en rekke anvendelser behov for replisering av data hovedsakelig motivert ut fra fordeling av leselast på et større antall noder. Dette er anvendelser hvor endringsraten er liten.

Apache Derby har pr i dag ingen støtte for replisering eller lastbalansering.

Det er ønskelig å endre JDBC driveren for å muliggjøre bygging av et cluster med Derby som kan støtte et svært høyt volum leseoperasjoner. Det er videre ønskelig at oppgaven ser på mulighetene for en slik driver til å benytte Derbys innebygde XA-støtte til å transaksjonelt oppdatere data på servernodene i et slikt cluster.

Dersom tiden tillater det, implementer og test en prototype som har den aktuelle funksjonaliteten.

2 Forkortelser

SPOF	Singel Point of Failure
JDBC	Java Database Connectivity
(R)DBMS	(Relational) Database Management System
TPS	Transaksjoner pr sekund
JVM	Java Virtual Machine, må brukes for å kjøre Java kode.

3 Innledning

Ser man på metoder for å spre informasjon til et stort publikum, peker trenden i stadig større grad mot nettbaserte løsninger. For at dette skal være vellykket er det et krav at tjenestene oppleves som raske for brukerne. Med et høyt antall brukere stilles det store krav til maskinvaren som kjører tjenestene. I mange tilfeller overgår disse kravene det en enkelt maskin kan klare å levere. Løsningen er da å samkjøre prosesseringskraften til flere maskiner i clustere. Utfordringen er at slike systemer ofte er svært komplekse og vanskelige å lage, noe som igjen gir økonomiske konsekvenser og utfordringer knyttet til drift og vedlikehold.

Mye av disse komplikasjonene er knyttet til behovet for å kunne oppdatere, eller legge inn ny, data på tvers av flere noder. Spesielt hvis oppdateringer kommer fra flere kilder, og ankommer forskjellige noder i ulik rekkefølge. Løsninger i bruk kan variere fra å sette inn sentrale punker på serversiden som styrer dataflyt, til delte kommunikasjonskanaler nodene bruker for å oppnå en felles enighet. Problemet med disse løsningene er at hvis clusteret vokser seg stort, kan sentrale punkter eller kommunikasjonskanaler bli flaskehalser og hemme ytelsen. Systemet når en øvre grense med tanke på størrelse, og dermed hvor stor belastning som kan håndteres totalt.

Ser man på de faktiske behovene brukerne av mange store websystemer har, vil man finne at i et stort antall tilfeller skal kun data leses. Det er ofte sjeldent noen ønsker å legge inn noe nytt, eller oppdatere [1]. Som regel vil oppdateringer være en jobb for administrativt personell, eller ansatte, i organisasjonen som driver systemet. For eksempel journalister i en nettavis. Dette gjør det mulig å skrenke inn på mulighetene til å oppdatere data, og dermed klare å minske eller fjerne de negative innvirkningene denne funksjonaliteten har.

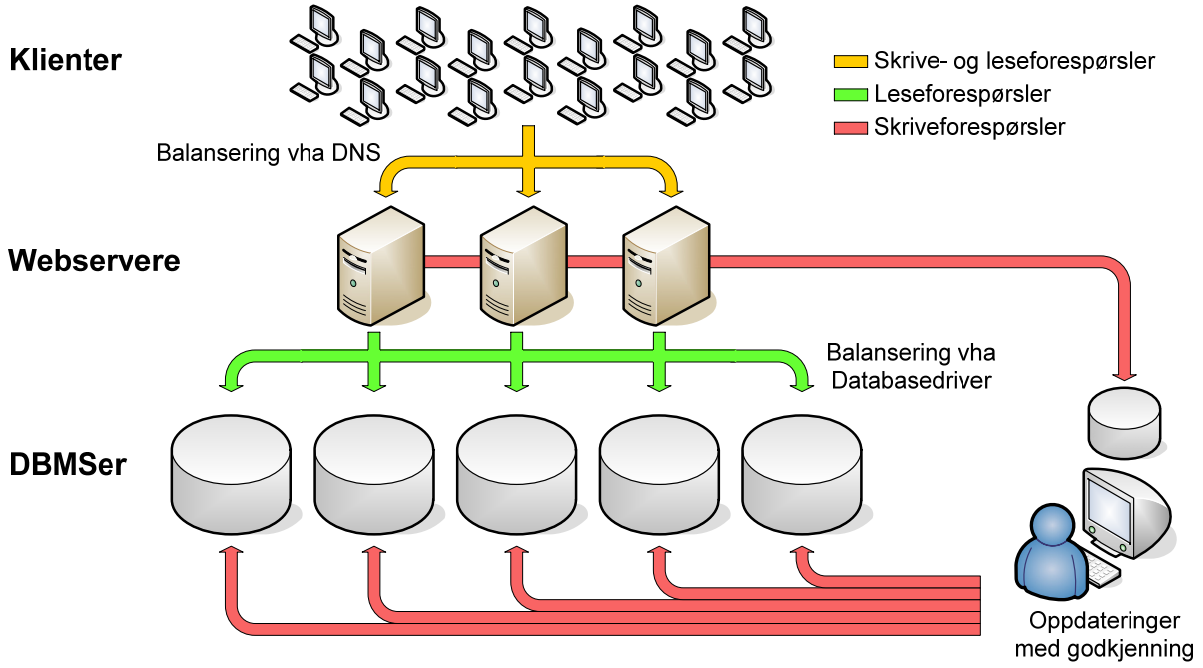
Denne oppgaven vil ta for seg hvordan det er mulig å innføre logikk på klientsiden slik at man kan lage et cluster med Derby som takler svært høy lesebelastning. Det vil fortsatt være mulig å gjøre transaksjonelle oppdateringer, men dette er funksjonalitet som ikke prioriteres med tanke på ytelse. Det at mest mulig logikk i systemet som helhet ligger på klientsiden vil kunne være positivt. Prosesseringskraften som er tilgjengelig vil jo øke etter hvert som flere og flere klienter kobler til, og man vil kunne ha en simplere og mer skaleringsvillig arkitektur på serversiden. Det bringer også med seg noen negative aspekter, men kan det være at de positive veier tyngre?

3.1 Aktuelle bruksområder

Her er noen aktuelle situasjoner hvor det er mulig å benytte et cluster som er bygd med fokus på å håndtere leseforespørsler. I de omtalte eksemplene vil brukermassen kunne være svært høy. Tenk på de omtalte systemene som regnet for nasjonal bruk, gjerne i et større land enn Norge.

3.1.1 Telefonkatalog

Primærfunksjonen vil være enkle oppslag av nummer ut fra navn, eller omvendt. I tillegg skal folk selv kunne endre sin egen informasjon i tilfelle flytting eller lignende. Slike ønsker om endring vil komme svært sjeldent, og i tillegg er det gjerne ønskelig med en viss moderasjon for å hindre misbruk av tjenesten.



Figur 3.1 Foreslått arkitektur for nettbasert telefonkatalog

Et aktuelt oppsett kan være som vist på figuren over. Klienter kontakter en webserverfarm som benytter DNS til å balansere last, ett domene peker til flere IP adresser. Disse serverne vil igjen enten gjøre et enkelt oppslag mot en tilfeldig databaseserver for å finne den informasjonen brukeren er på jakt etter, eller sende en forespørsel om dataendring videre til en moderasjonsstasjon.

Hvis en oppslagsnode har krasjet, og oppdateringen ikke kan gjennomføres, kan man bygge opp en kø på moderasjonsstasjonen. For så å legge inn endringene som en batch på et senere tidspunkt. Krasjer moderasjonsstasjonen ber man simpelthen bare brukeren om å forsøke senere.

3.1.2 Nettavis

Anta følgende krav til funksjonalitet:

- Alle skal kunne lese artikler og kommentarer (primærfunksjon)
- Alle skal kunne kommentere på artikler
- Journalister skal kunne legge inn og endre artikler

Et cluster som har fokus på lesing vil være godt rustet til å løse primærfunksjonen, man kan lett skalere opp for å møte krav til ytelse. Innlegging av kommentarer og nye artikler vil

foregå sjeldent i forhold til lesing av nyheter. Man vil kunne tro at vanlig mengde er en 20-30 nye artikler pr dag, med et mindretall kommentarer for hver. Hvis mengden oppdateringer er så lav vil den være uproblematisk å ta unna.

Hvorvidt det er påkrevd med et eget sidesystem som sekvenserer oppdateringer inn til nodene er en del av denne oppgaven, men hvis dette er faktum kan man benytte en lik arkitektur som for telefonkatalogen. Det er dog ønskelig med en liten endring; brukere som legger inn nye kommentarer vil gjerne se at kommentaren sin er kommet på plass. For å løse dette kan man ganske enkelt blokkere brukeren helt til sidesystemet har utført innleggingen, for så å sende brukeren tilbake til artikkelen. Skulle det være at innleggingskapasiteten er nådd, og endringen må legges i kø på sidesystemet, kan brukeren ganske enkelt gis beskjed om dette.

3.1.3 Karttjeneste

En tjeneste som har blitt svært populær de siste årene er nettbaserte karttjenester. Man kan søke på adresser og få opp kart over det aktuelle området, med mulighet for zoom og panorering.

Dette er en tjeneste som utelukkende lar brukermassen lese. Oppdateringer av kart og adresser gjøres like sjeldent som endringer av den fysiske naturen, og kun av administrativt personell. I dette tilfellet kan det faktisk være positivt at oppdateringer ikke gis prioritet med tanke på ytelse, man ser jo helst at oppdateringene ikke stjeler kapasitet fra den lesing som foregår.

4 State of the Art

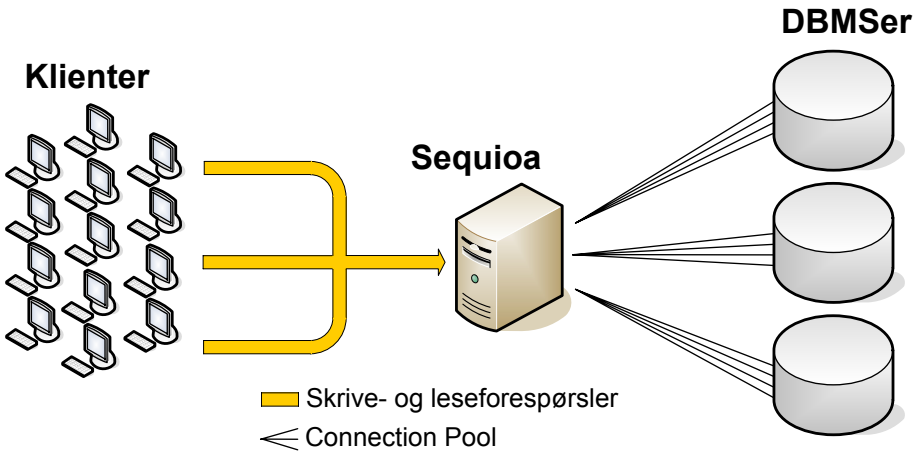
Går her gjennom noen alternative og eksisterende systemer som kan brukes for å sette opp en løsning som kan håndtere et stort antall leseforespørsler. Fokus vil være på lesing, men det vil være et krav at systemet skal kunne klare å ta imot oppdateringer. Ikke nødvendigvis fra alle og enhver, men i hvert fall fra en administrator.

I forrige semesters prosjektoppgave [2] ble et system med navn Sequoia testet. Dette, og erfaringer fra testing, vil bli sett nøyere på da mye av grunnlaget for denne oppgaven kommer derifra.

4.1 Sequoia

Sequoia [3, 4, 5, 6] er en transparent mellomvareløsning som kan brukes til å sette opp en cluster av heterogene databasesystemer. Man kan bygge med noder som kjører for eksempel MySQL [7], Derby [8] eller DB2 [9]. Implementasjonen er utført 100% i Java, og JDBC standarden støttes fullt ut gjennom en egen driver. Applikasjonene trenger dermed ikke endres for å bruke systemet. Sequoia slippes under Apache License v2.0 [10], noe som gjør dette til et prosjekt med åpen kildekode.

For å få dette til å fungere benyttes en shared-nothing arkitektur med en sentral node, kalt en kontroller, som fungerer som et kontaktpunkt utad. Gjennom å analysere alle forespørsler som ankommer, kan kontrolleren enten balansere leseforespørsler eller spre oppdateringer. Data kan repliseres fullstendig, partisjoneres eller en blanding av begge. En klient trenger ikke tenke på å skille lesing fra skriving. Alt som foregår internt i clusteret maskeres totalt fra klientene.

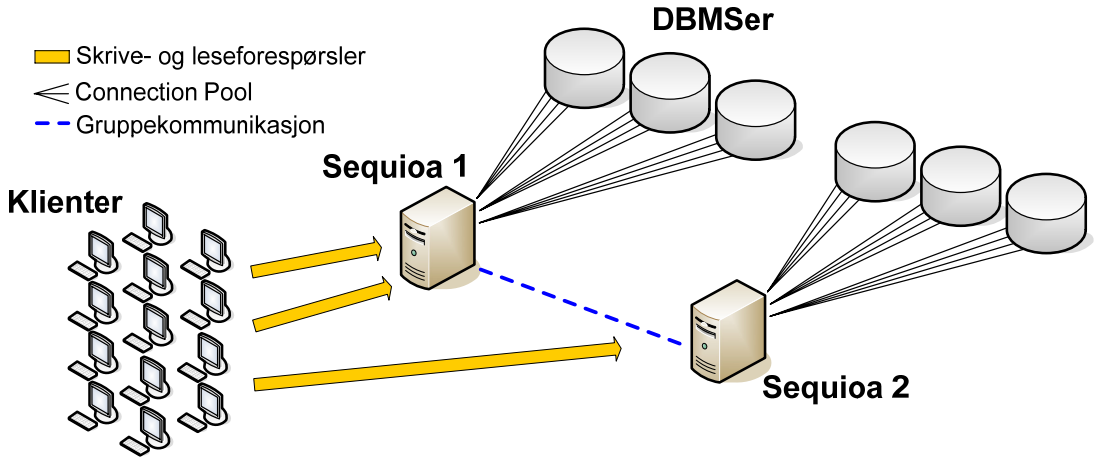


Figur 4.1 Sequoia arkitektur

Det finnes flere problemer med å ha en sentral kontroller. Ett er at kontrolleren blir et SPOF, krasjer den vil hele clusteret være utilgjengelig. Et annet er at kontrolleren også kan bli en

flaskehals. En kontroller vil jo være ansvarlig for å inspisere hver enkelt forespørsel, så det er nærliggende å tenke at det finnes en maksgrense for antall noder man kan sette inn.

For å avhjelpe begge disse problemene er det mulig å sette flere kontrollere i parallell. Klientene vil da ha flere kontaktpunkt til clusteret. Sequoias JDBC driver vil da sørge for å balansere last, forutsatt at driveren har fått opplyst adresser til alle kontrollere. Skriveforespørsler vil kun sendes til en kontroller. Denne må da sørge for å både oppdatere sine egne noder, og sende forespørselen videre til alle andre kontrollere. For å få til dette kommuniserer alle kontrollere sammen gjennom ett av to mulige rammeverk for gruppekommunikasjon, JGroups [11] eller Appia [12]. En bør merke seg at begge disse er tilpasset generell gruppekommunikasjon, og kan ikke sies å være optimale til bruk i databasesammenheng.



Figur 4.2 Parallelle Sequoia kontrollere

Nodene som står bak hver kontroller kan ikke tilhøre mer enn en kontroller om gangen. Hvis en kontroller blir utilgjengelig vil også dens noder bli det. Noder kan flyttes mellom kontrollere, men dette er en manuell oppgave. Hver node må også ha en globalt unik id om man ønsker å sette flere kontrollere i parallell.

4.1.1 Feildeteksjon og ”recovery”

Hvis en node bak en kontroller skulle krasje, vil dette bli fanget opp av kontrolleren og ingen flere forespørsler vil bli sendt til noden. Hendelsen vil loggføres, men ingen vil bli aktivt varslet. Administrator må derfor følge med på dette selv.

I oppsett med flere parallelle kontrollere vil klientenes driver sørge for å oppdage et kontrollerkrasj, og unnlate denne fra videre lastbalansering. De andre kontrollerne vil også oppdage at en mangler fra gruppekommunikasjonen, og merke denne som feilet.

Etter et at en node har krasjet, vil kontrolleren kjøre en vanlig recoveryprosedyre. Det siste checkpointet blir lastet inn, og loggen siden dette ble tatt spilt av ut mot den aktuelle noden. Mekanismen som tar seg av dette er modulært oppbygd for å støtte mange forskjellige DBMS. Den nøyaktige prosedyren kan jo variere fra DBMS til DBMS. Å starte recoveryprosessen må gjøres manuelt av administrator. Det er en relativt komplisert prosess

og bør gjøres med manualen i hånd for å unngå feil. Det samme gjelder for å ta checkpoints. Når loggen begynner å bli stor må administrator selv sørge for å ta checkpoints.

Selve recoveryloggen i Sequoia blir lagret i en separat database over JDBC, ikke rett på disk som normalt i andre systemer. Dette gjør det mulig å skreddersy forholdet mellom ytelse og sikkerhet på loggen. Det er for eksempel godt mulig å benytte et separat Sequoia-oppsett til å lagre loggen på.

Skulle det skje at en kontroller feiler må siste checkpoint og logg overføres fra en annen kontroller. Deretter kjøres det recovery på vanlig måte ut mot den krasjede kontrollerens noder. Alt dette er også en komplisert manuell prosedyre.

4.1.2 Erfaringer

Forrige semesters prosjektoppgave gikk ut på å teste funksjonalitet og ytelse på leseforespørsler i Sequoia. Databasesystemet Apache Derby ble benyttet på alle noder. Det ble brukt 6 like maskiner til å kjøre kontroller og databasenoder, samt en separat og ulik maskin til å kjøre et antall klienter fra. Flere forskjellige kontroller- og nodekonfigurasjoner ble testet på et antall måter.

Drar her frem noen ytterpunkter fra testing av ytelse, og gir en kortfattet utgave av konklusjonen også rettet mot ytelse.

4.1.2.1 Enkeltstående kontroller med flere noder bak

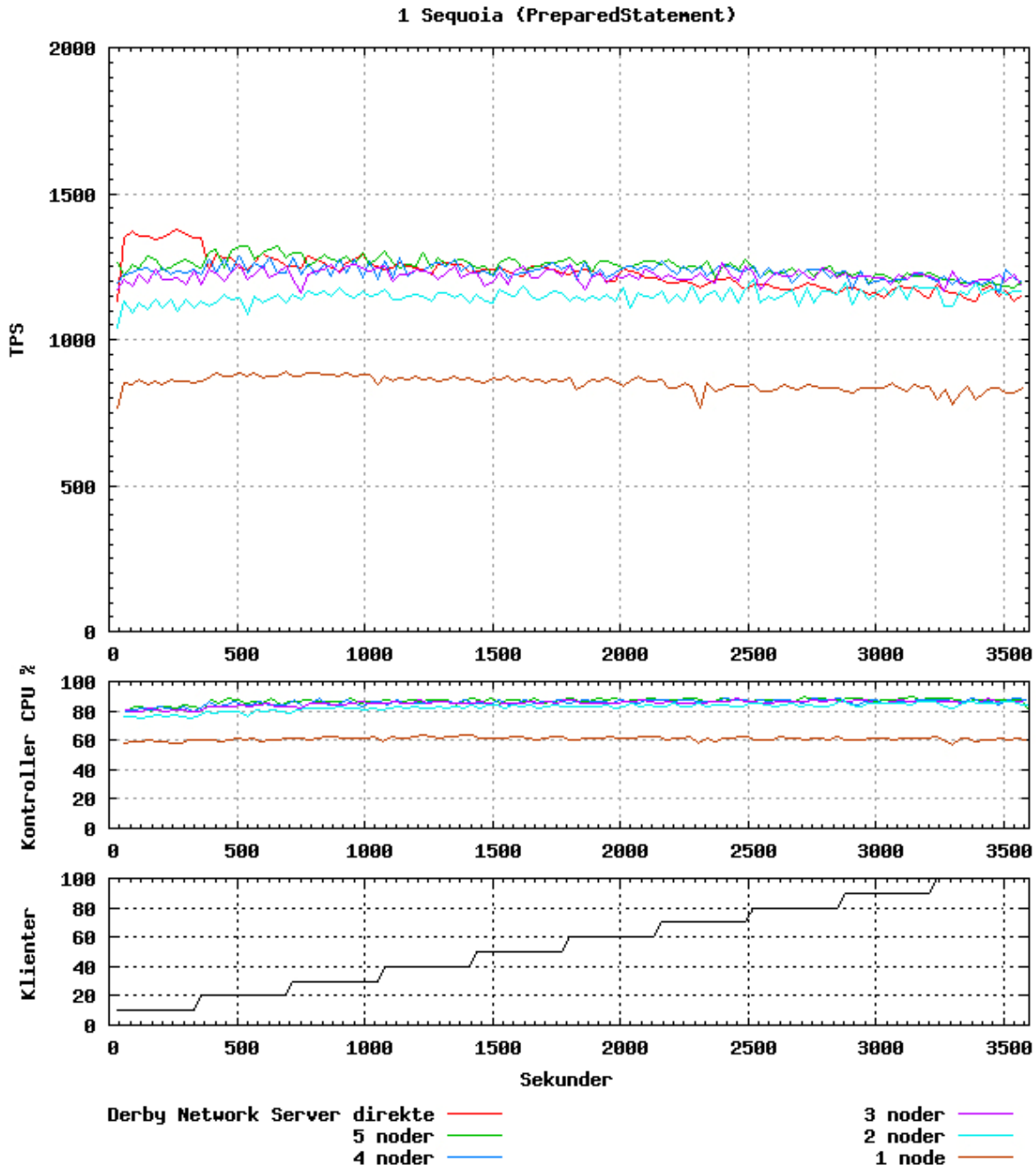
Dette er det mest grunnleggende oppsettet av Sequoia. 1 enkelt kontroller med et antall noder bak å dirigere forespørsler til. Alle noder inneholder komplett kopi av all data.

Testdata var en enkelt tabell bestående av et Integer felt (primærnøkkel) og et tilfeldig valgt tekstfelt på 96 tegn. Antall rader var 524 288, og gav en total datamengde på 50 MB. Hver innkommende transaksjon var en Select-setning som kun returnerte en enkelt rad.

I denne testrunden startet Sequoia med kun 1 databasenode bak seg. Det ble så kjørt transaksjoner mot kontrolleren fra 10 samtidige klienter. Antallet klienter ble økt med 10 hvert 360. sekund, og testen stoppet etter en time når det var 100 klienter kjørende. Deretter ble det lag til en node til, og prosessen gjentatt til det stod totalt 5 noder bak.

Forventningene var at TPS skulle synke med en viss prosent i forhold til når klientene koblet rett til en ensom Derby, men ved å legge til flere noder skulle ytelsen øke og overstige den enkeltstående Derbyen.

Resultatene kan ses på figur 4.3.

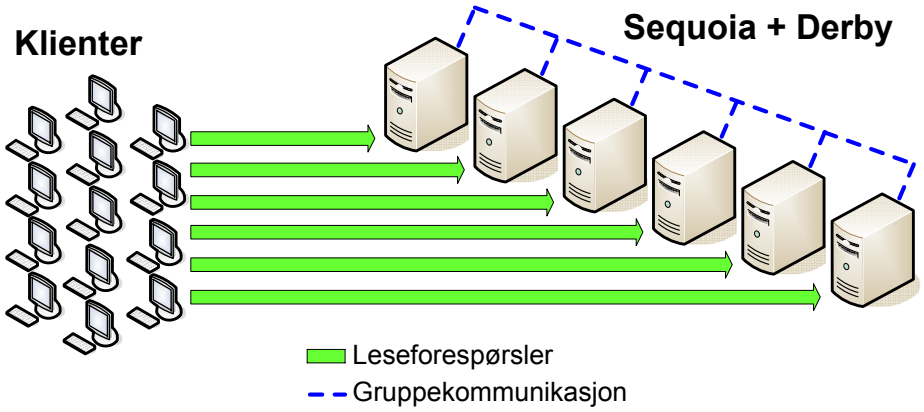


Figur 4.3 Testresultater - enkel Sequoia

Av resultatene er det tydelig at Sequoia, selv med et såpass enkelt oppsett, ikke hadde bra nok ytelse. Som forventet sank TPS litt når Sequoia kjørte med 1 node, men ytelsen steg aldri nok til å overgå det en enkelt Derby klarte uansett antall noder.

4.1.2.2 Parallelle Sequoia med 1 node

I denne testrunden ble Sequoia konfigurert til å ha en databasenode kjørende på samme maskin som den selv. Det ble benyttet samme testdata og forespørsler som i forrige testrunde.

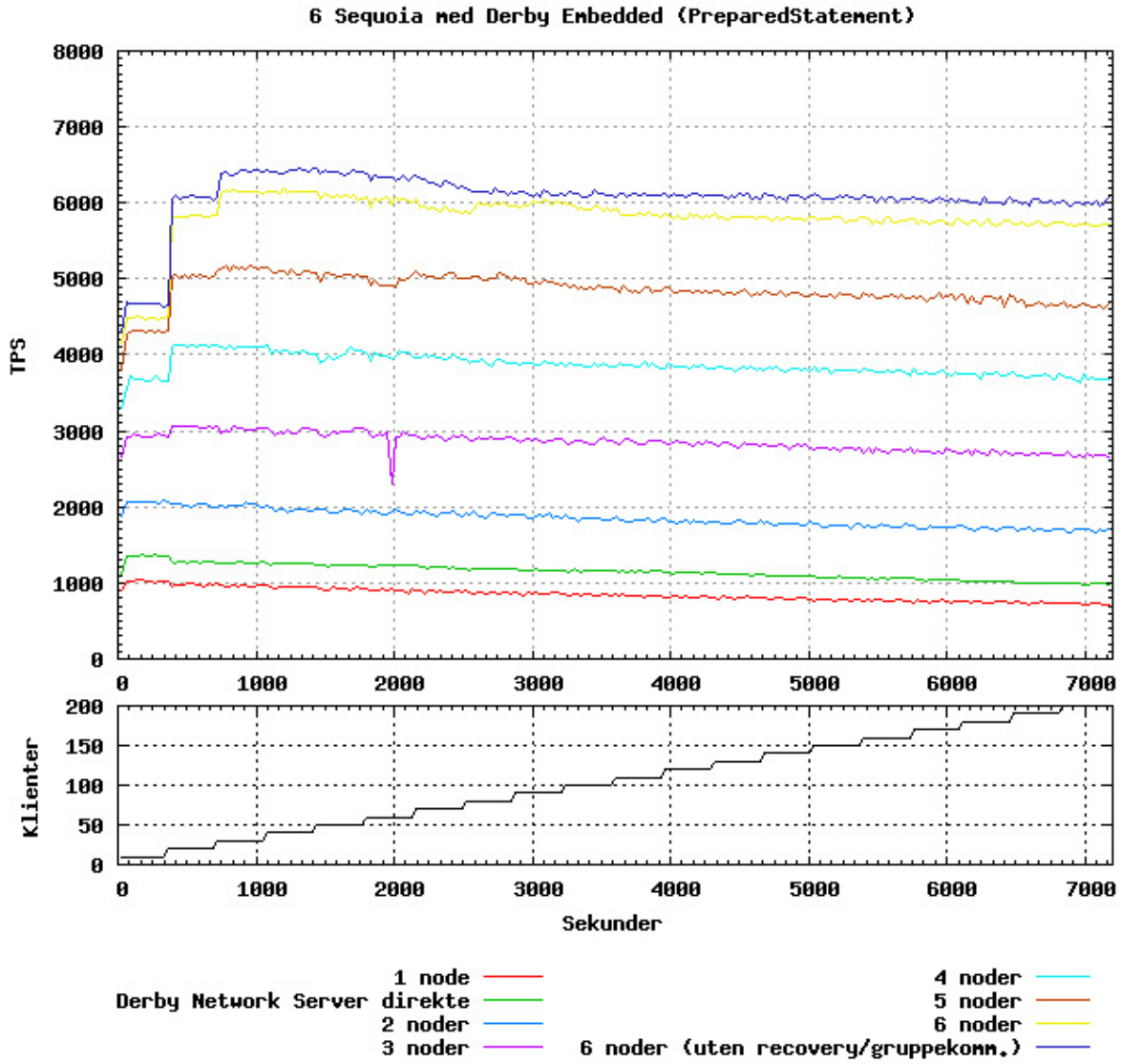


Figur 4.4 Testoppsett - parallelle Sequoia

Gjennom testrunden skulle det startes med 1 kontroller + node. 10 samtidige klienter skulle kjøre spørringer mot den, og antallet klienter økes med 10 for hvert 360. sekund. Testkjøring varte i en time, og det var totalt 100 samtidige klienter kjørende på slutten. Deretter ble det lagt til enda en kontroller + node, og prosedyren gjentatt til det var 6 slike noder kjørende.

Forventningen var at antall TPS skulle skalere 100% for hver nye kontroller som ble lagt til. Altså 6 maskiner tok unna 600% av det 1 maskin greide.

Resultatene kan ses på figur 4.5.



Figur 4.5 Testresultater - parallelle Sequoia

Av grafene er det tydelig at forventningene ble innfridd. Ytelsen sank litt når Sequoia kjørte sammen med Derby på samme maskin i forhold til kun Derby alene, men steg 100% for hver nye node med kontroller + node som ble lagt til.

Til slutt ble det testet mot 6 kontroller + noder hvordan ytelsen ble når recoverylogg på Sequoia og gruppekommunikasjon mellom kontrollerne var skrudd av. Dette hadde som forventet en veldig liten effekt da testtransaksjonene ikke oppdaterte noe, og både logg og gruppekommunikasjon stod ubrukte.

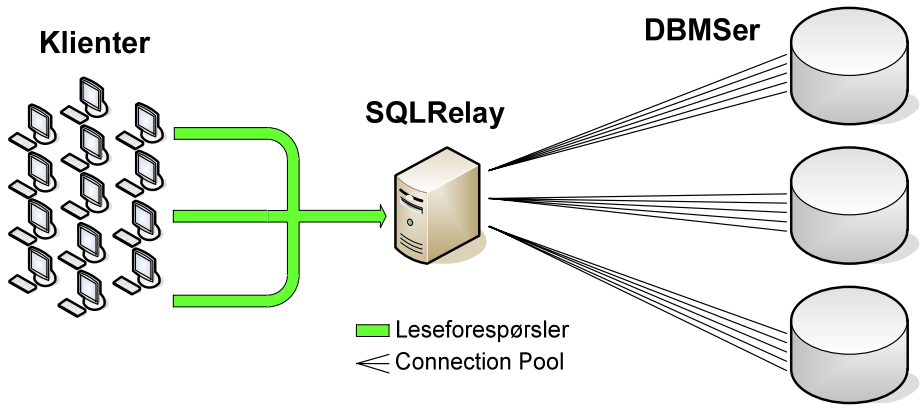
4.1.2.3 Kortfattet konklusjon av ytelse

I de konfigurasjoner der Sequoia fungerer som en sentral fordeler er det ikke mulig å oppnå noen særlig høy ytelse. Som sagt går ikke TPS over det en ensom databasenode greier på egen hånd. Derimot kan man oppnå svært god ytelse hvis man setter flere kontrollere parallelt. En

skaleringsprosent på inntil 100% er da å forvente. Det er viktig å få med seg at dette gjelder kun for leseforespørsler.

4.2 SQL Relay

SQL Relay [13] er en mellomvareløsning som kan benyttes til å sette opp et system med stor kapasitet med hensyn på leseoperasjoner. Tankegangen minner litt om Sequoia, men har en simplere virkemåte og flere begrensninger. SQLRelay slippes under GPL lisens [14] og har dermed åpen kildekode.



Figur 4.6 SQLRelay arkitektur

Måten SQLRelay fungerer på er at den oppretter, og holder, en connection pool til flere DBMS servere bak. Noden med SQLRelay vil så lytte etter forespørsler fra klienter og formidle disse videre til serverne bak som en lastbalanserer. All data må være fullstendig replisert på alle databasenoder.

SQLRelay støtter flere forskjellige DBMSer som noder:

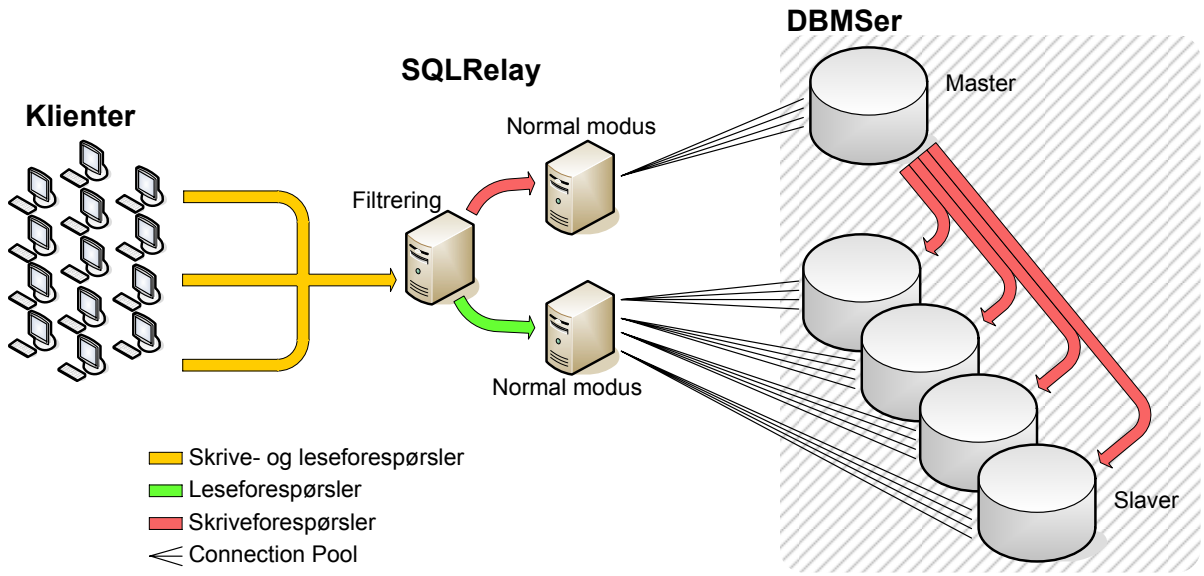
- Oracle
- MySQL
- PostgreSQL
- Sybase
- MS SQL
- DB2
- Interbase
- SQLite.

Ut mot klientene tilbys det APIer for C, C++, Perl, Python, Zope, PHP, Ruby, Java og Tcl. I tillegg finnes det også drivere som kan benyttes som en drop-in erstatning for MySQL og PostgreSQL klienter. SQLRelay baserer seg altså ikke på generelle standarder som JDBC, men direkte API mellom klientene, seg og nodene.

Oppdateringer til data, og replisering av disse, blir ikke sett på som en direkte oppgave for SQLRelay. Det er fortsatt mulig å ha et oppsett der oppdatering av data er mulig, men det

krever at DBMSet som benyttes støtter replisering. Hvis oppdateringer kan ankomme en vilkårlig databasenode som sprer videre trenger man kun en enkelt SQLRelay node i normal modus. Skriveforespørslene vil da sendes gjennom som om de var en leseforespørsler.

I mange tilfeller vil det være nødvendig at oppdateringer går gjennom en utpekt master. Da trenger man flere noder med SQLRelay, og sette en av dem i såkalt filtreringsmodus.



Figur 4.7 SQLRelay og oppdateringer

Det skraverte området er funksjonalitet som det benyttede DBMS må bidra med.

Oppdateringer vil her bli filtrert ut og sendt videre til master databasenoden gjennom en normal SQLRelay. Masteren vil deretter være ansvarlig for å spre oppdateringene videre til slavenodene. Leseforespørsler må behandles på samme måte. De filtreres ut og sendes videre til slavenodene gjennom en annen SQLRelay.

Det kan virke tungvint at man må sette opp tre noder med SQLRelay, men det er fult mulig å kjøre alle tre instansene på en og samme maskin.

Et problem med SQLRelay er at man har et SPOF. Går noden med SQLRelay ned vil hele systemet stoppe opp, men programvaren fremstår som mindre komplisert og dette minker sjansene for et krasj. Dog er man like sårbar ovenfor problemer med maskinvaren.

Et annet mulig problem er ytelse. Skalerer man veldig ut med mange databasenoder kan SQLRelay nodene bli flaskehalser. Skulle dette problemet oppstå kan det være vanskelig å komme med en løsning, spesielt hvis maskinvaren på SQLRelay noden allerede er tipp-topp.

4.3 MySQL

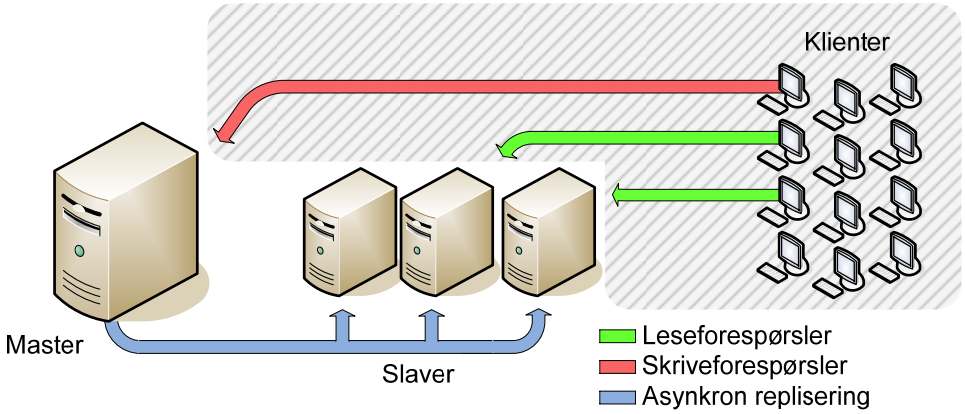
MySQL [7] er et alternativ som kan brukes til å balansere leseoperasjoner på en fin måte. Dette databasesystemet slippes under GPL lisens, og har åpen kildekode.

En løsning vil være todelt. For det første trenger serverne å være satt opp med master-slave replisering. Alle servere kan leses fra, men oppdateringer må gå til en utpekt server. Endringene vil deretter spre seg fra masteren til slavene. Repliseringen foregår asynkront ved at slavene konstant leser loggen på masteren. Det er også mulig at en slave opptrer som master for andre slaver slik at man får en trestruktur. Støtte for denne typen replisering har vært å finne i MySQL siden versjon 3.23.15 [15].

Det er viktig å få med seg at data vil bli replisert og ikke partisjonert. Alle noder vil altså inneholde en komplett kopi av alle data. Man vil bare kunne øke ytelse og ikke lagringskapasitet, men man unngår situasjoner der popularitet for en undergruppe av data forårsaker ujevn lastbalansering.

Del to av løsningen er å benytte en spesiell JDBC driver fra MySQL som heter ReplicationDriver [16]. Denne driveren vil sende leseforespørsler mot en tilfeldig slave, men oppdateringer vil gå til masteren. Hvorvidt en transaksjon inneholder bare leseforespørsler, eller trenger å skrive noe, må klienten spesifisere på forhånd. Denne driveren har vært å finne i MySQLs Connector/J pakke [17] siden versjon 3.1.7.

Siden replisering foregår asynkront er det ikke sikkert en klient som akkurat har oppdatert noe vil kunne se sine oppdateringer ved neste transaksjon. Dette er noe applikasjonen må ta hensyn til. I enkelte tilfeller kan dette være et problem. Løsningen er da at klienten later som om neste transaksjon også skal skrive. Oppkoblingen vil da gjøres mot masteren, og endringene fra forrige transaksjon vil være synlige. Et poeng er at dette ikke bør skje for ofte slik at masteren blir for hardt belastet.



Figur 4.8 MySQL replisering + ReplicationDriver

ReplicationDriver bidrar med funksjonaliteten som er på det skraverte området.

En klient som vil benytte denne driveren kan ikke samtidig laste inn MySQLs normale JDBC driver. Dette grunnet at begge aksepterer det samme url-formatet. Ønsker man å bruke begge driverne må dette gjøres i forskjellige JVMer.

4.3.1 Feildeteksjon og ”recovery”

På serversiden finnes det ingen mekanismer for feildeteksjon annet enn at hver node passer på seg selv. Master følger ikke med hvilke slavenoder som henter data hos den, så hvis en slave feiler vil master fortsette som normalt. Skulle det skje at masternoden feiler vil slavene fortsette kjøring med den data de allerede har mottatt, og jevnlig prøve å koble til master igjen.

Hvis driveren hos en klient prøver å koble til en feilet node vil den avbryte oppkoblingsforsøket og prøve på nytt mot en annen node.

En feilet node vil kjøre et normal recoveryløp, dvs. laste inn siste checkpoint og spille av loggen siden den gang.

4.4 Slony-I

Slony-I [18] er et tillegg til PostgreSQL [19] som gir støtte for replisering fra en master til flere slaver. Det er viktig å merke seg at dette systemet kun gir mulighet for replisering på serversiden, og ikke tilbyr noen løsning på:

- hvordan oppdateringer skal rutes til masteren.
- hvordan leseoperasjoner skal balanseres.

Slony-I har heller ingen støtte for å oppdage feil eller gjennomføre fail-over. Dette settes altså bort til 3. parts programvare. Som PostgreSQL har prosjektet åpen kildekode.

Repliseringen mellom master og slaver blir utløst av triggere på masteren, og endringene spres asynkront. Det er dermed en mulighet for at data kan være globalt inkonsistent i små tidsrom.

Skulle man trenge mange servere for å ta unna last er det mulig å nøste repliseringen. Altså en slave kan oppføre seg som en master for en annen slave osv. For å holde orden på et slikt oppsett er 3 forskjellige roller definert:

- Origin

  Dette er den faktiske masterserveren, noden som alle oppdateringer ankommer først.

- Provider

  Brukes om slaver som oppfører seg som mastere for andre slaver.

- Subscriber

  Er en slave som ikke sender oppdateringer videre til andre. Kan ses på som en løvnode i repliseringstreet.

Det at repliseringen er asynkron gjør at fail-over bør gjøres forsiktig. Hvis en master krasjer brått kan man risikere å ha transaksjoner som er commited på master, men som ikke har rukket å finne sin vei til slavene.

4.5 Times Ten

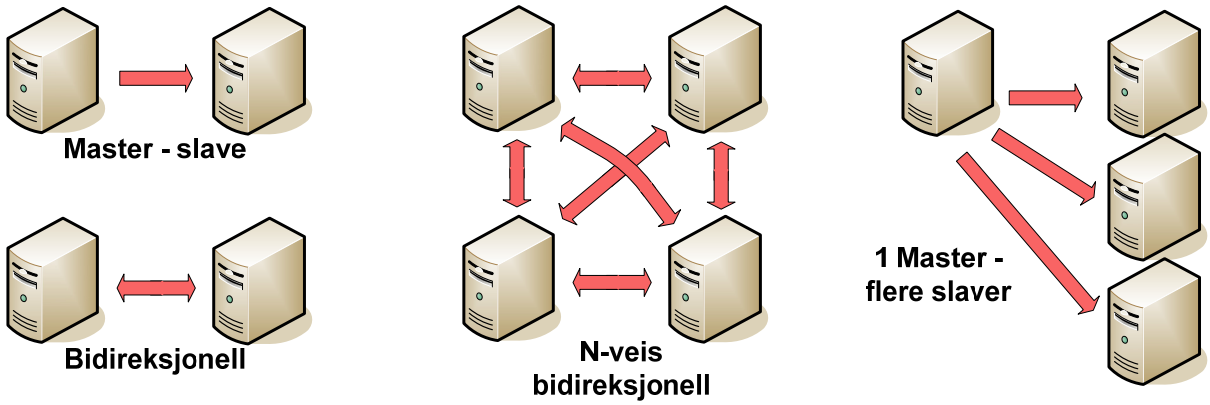
Times Ten [20, 21] er et RDBMS som benytter primærminnet på en maskin til lagring av sine data. Systemer som benytter en harddisk som lagringsmedium har ofte en virkemåte som er tilpasset måten harddisken jobber på. Dette er positivt så lenge man faktisk lagrer til disk, men kan sinke systemet unødvendig hvis det lagres til primærminnet. Times Ten er i utgangspunktet designet for å lagre i minnet, og har ikke disse tilpasningene. Dette gjør den svært kjapp og godt egnet til å bygge løsninger som kan ta unna svært mange forespørsler.

For å forhindre tap av data hvis Times Ten kræsjer kan den konfigureres til å lagre checkpoints og logg på disk som et normalt diskbasert RDBMS gjør.

Det finnes hovedsakelig to forskjellige måter å bruke Times Ten på. Den ene er replisering og den andre er å benytte flere Times Ten som hurtigbuffer for en tradisjonell diskbasert Oracle database. Det finnes flere forskjellige måter å la klienter koble til på, men det eksisterer ingen ferdiglaget driver som balanserer last eller dirigerer skriveforespørsler. Dette må altså tas hånd om av applikasjonen eller 3. parts programvare.

4.5.1 Replisering

Det finnes flere forskjellige typer replisering man kan benytte [22].



Figur 4.9 Times Ten repliseringsstrategier

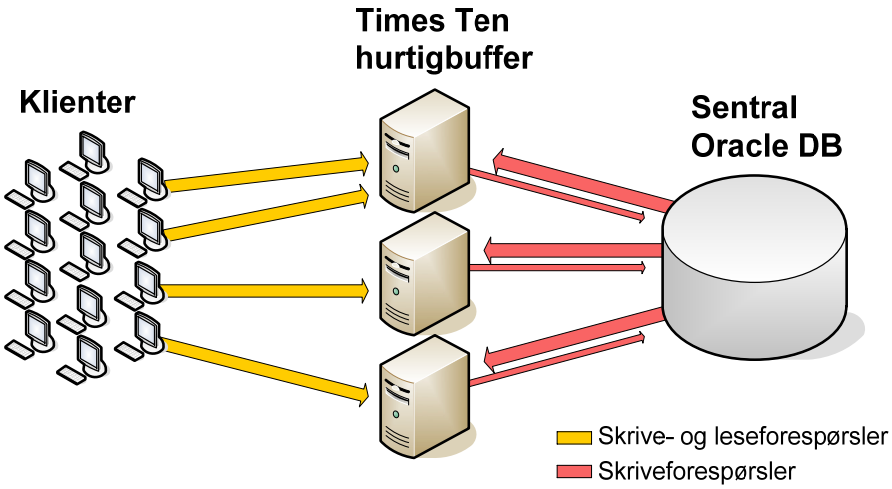
Først finnes den mer tradisjonelle formen der endringer spres enveis fra en utpekt master til en eller flere slaver. Deretter finnes en variant hvor endringer spres begge veier. Repliseringen fungerer ut fra en loggbasert strategi. Det er også mulig å finjustere forholdet mellom ytelse og global konsistens ved å velge mellom ett av tre alternativer for hvor lenge en klient skal blokkeres ved oppdateringer:

- Til oppdateringen er mottatt av master.
- Til oppdateringen er mottatt av alle noder.
- Til oppdateringen er både mottatt og lagret av alle noder.

Selve dataen kan enten repliseres i sin helhet på alle noder, eller partisjoneres slik at man også kan skalere datamengden.

4.5.2 Hurtigbuffer for sentral Oracle database

Det er mulig å benytte en eller gjerne flere Times Ten noder som hurtigbuffer for en sentral diskbasert Oracle database. Mekanismen som muliggjør dette heter Cache Connect [23]. Man står fritt til å velge hvilke tabeller i en database som skal bufres. Spesielt hvis man har store datamengder kan det være aktuelt å kun bufre et utvalg av data som gjerne har høy popularitet.



Figur 4.10 Times Ten Cache Connect

Hvordan data spres fra den sentrale databasen ut til buffernodene kan foregå på følgende måter:

- Manuelt
- Automatisk på fastsatte tidspunkt eller intervaller
- Automatisk fortløpende

Det er mulig at oppdateringer kan mottas av en Times Ten buffernode i stedet for den sentrale databasen. Også her er det flere alternativer for hvordan disse oppdateringene skal finne sin vei til resten av systemet:

- Automatisk fortløpende
  Oppdateringer vil først mottas på buffernoden, og umiddelbart videresendes til den sentrale noden. Denne modusen kan sammenlignes med en såkalt write-through cache.

- Manuelt

- Pass-through
  I stedet for at oppdateringen blir behandlet av buffernoden først, vil den heller videresendes direkte til den sentrale noden. Slik får man ikke en cachenode som er oppdatert før den sentrale noden bestemmer seg for å oppdatere alle. Dette vil foregå helt transparent for klienten.

4.5.3 Feildeteksjon og ”recovery”

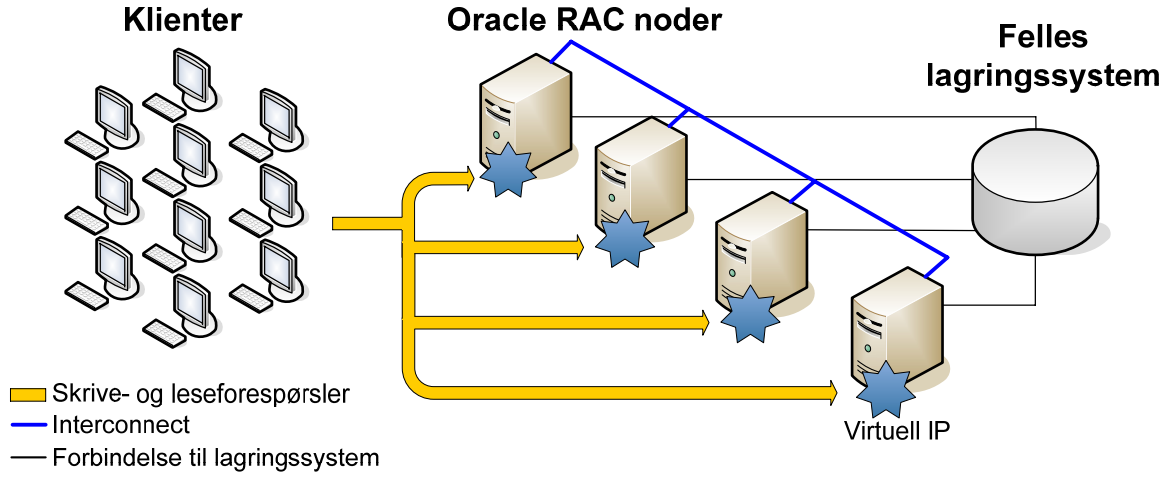
Times Ten har ingen innebygd funksjonalitet for å merke om en node feiler, eller utføre fail-over. Dette har Oracle regnet som å være utenfor systemets område, og setter det heller bort til 3. parts clustermanagere.

Ved ”recovery” vil en vanlig strategi benyttes. Når en applikasjon kobler seg til på nytt etter et krasj vil Times Ten allokere et nytt område i minnet på maskinen, laste inn siste checkpoint fra disk, og spille av loggen siden dette ble tatt. Dette forutsetter at Times Ten er konfigurert til å lagre checkpoints og logg på disk.

4.6 Oracle RAC

Oracle Real Application Cluster [24, 25] er en tilleggspakke til Oracle Database 10g [26] som gjør det mulig å bygge en cluster med inntil 100 noder.

Måten dette gjøres på er at selve Oracle DB instansen (selve programmet) deles fra data. I stedet for at data ligger lagret på samme maskin som instansen kjøres på, legges data på et felles lagringssystem. I tillegg må alle noder være sammenkoblet med et såkalt interconnect med høy båndbredde. Arkitekturen kan kalles shared-everything, og man er avhengig av at lagringssystemet er i stand til å skalere etter hvert som man legger til flere noder. Det finnes ingen SPOF, men dette er igjen avhengig av at lagringssystemet heller ikke har det.



Figur 4.11 Oracle RAC arkitektur

Måten klienter kobler seg til er at alle noder er tilgjengelige for omverdenen og har to IP adresser hver. Den ene er fastsatt til hver node, mens den andre kan ses på som en ”virtuell IP”. Det er den virtuelle klienter benytter for å koble til. Oppkobling og balansering gjøres med en driver som heter Oracle Net [27]. Skulle en node krasje vil den virtuelle IP adressen flyttes til en annen node. Klientene vil da alltid få et svar, og dermed ikke merke noe til eventuelle krasj på serversiden. En robust løsning, forutsatt at mekanismen som skal fange opp feil fungerer kjapt og som den skal.

Nodene man benytter i clusteret trenger ikke være identiske med tanke på maskinvare, men de må alle kjøre samme operativsystem og versjon av Oracle DB.

RAC gir en database kontinuerlig tilgjengelighet ved at databasen alltid er tilgjengelig selv når det utføres vedlikehold. En node kan tas ut av drift, bli behandlet og satt rett inn før neste node gjennomgår samme prosess.

4.6.1 Feildeteksjon og ”recovery”

For å fange opp feil benyttes et verktøy som heter Oracle Clusterware. Dette vil automatisk oppdage feilede noder, og prøve å få dem tilbake i drift. Dette systemet regnes for å være svært kjapt, og grunnet høy automasjonsgrad vil clusteret ofte ha hentet seg inn etter krasj før administrator rekker å finne ut hva som er feil. Det er jo ikke nødvendig å kjøre en langsom recoveryprosess for å sørge for konsistente data på en feilet node.

5 Apache Derby

Apache Derby [8, 28, 29] er et diskbasert RDBMS skrevet i Java. Kildekoden slippes under Apache License v2.0, noe som gjør dette til et prosjekt med åpen kildekode.

I 1997 slapp et firma ved navn Cloudscape sitt databasesystem med samme navn. I 1999 ble dette firmaet kjøpt opp av Informix, og ytterligere 2 år senere kjøpte IBM databaseavdelingen i Informix. IBM beholdt systemet i 3 år før de i 2004 donerte det til Apache Software Foundation. Kildekoden ble da åpnet under Apache License v2.0, og navnet endret til Derby. Sent i 2005 bestemte Sun Microsystems seg for å gi ut Derby som en del av sitt Sun Java Enterprise System, og gikk da inn med ressurser i prosjektet for å utvikle det videre.

Det som gjør Derby spesiell i forhold til de fleste andre RDBMSer er at den har to operasjonsmodi. En der Derby fungerer som en normal tjener i en klient-tjener modell, og en annen der hele Derby kjører lokalt inne i klientens applikasjon (embedded).

Under utvikling har det vært mye fokus på å holde administrasjon og konfigurasjon så enkel som overhodet mulig. I embedded modus vil det normalt ikke være aktuelt å gjennomføre noe administrasjon, bare ta den rett i bruk. Dette gjør Derby svært enkel å leve med og terskelen for å ta systemet i bruk holdes lav.

Det har også vært fokus på å følge anerkjente standarder. For å nevne noen:

- SQL92, -99, -2003
- SQL/XML
- JDBC 2.0 og 3.0
- DRDA (som nettverksprotokoll i klient-tjener sammenheng)

I likhet med de fleste andre RDBMSer støtter Derby samtidige klienter, ulike isolasjonsnivåer, deteksjon av vranglås, recovery etter krasj og mer. Hva Derby mangler er støtte for replisering. Dog har den støtte for XA transaksjoner ved å implementere XAResource interfacet, noe som vil bli brukt i denne oppgaven til å omgå problemet med den manglende replikasjonsstøtten. Mer om hvordan XA virker i kap. 6.

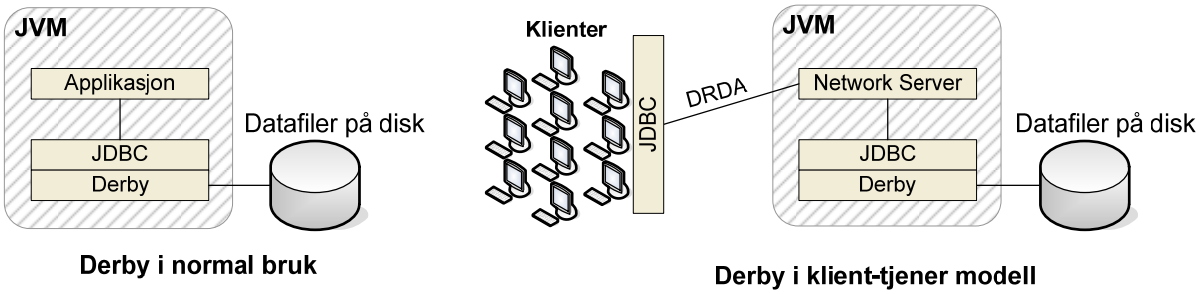
For å kjøre Derby trenger man en Java SE JVM med versjon 1.3 eller høyere. Det også mulig å benytte Java ME.

5.1 Arkitektur

Som nevnt tidligere er det som skiller Derby fra andre RDBMSer at det er et såkalt ”embedded” system. Man har ikke den tradisjonelle virkemåten der serveren lytter etter innkommende forespørsler over nettverket, men heller blir en del av klientapplikasjonen. For å gjøre dette mulig er Derbys runtime kun på ca 2MB og krever ikke mer enn 4MB Java heap størrelse. Måten klienten benytter Derby er som ved vanlige databasetjenere, ved hjelp av en

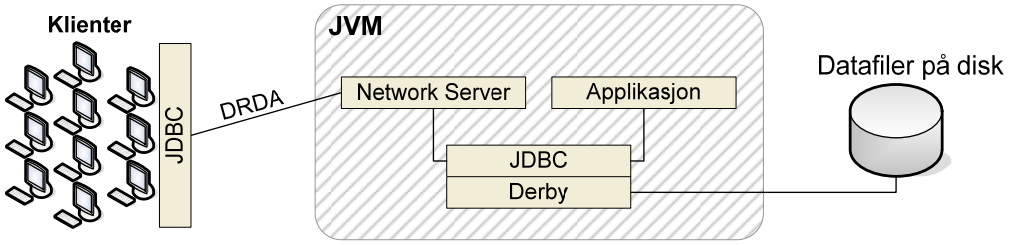
JDBC driver, men i stedet for at driveren tar kontakt med tjeneren over nettverket vil Derbys databasemotor kjøre inne i driveren. Klienten vil altså ikke kunne se noen forskjell fra nettverksbaserte tjenere utenom at responstidene går merkbart ned.

Ønsker man å benytte Derby i en mer normal situasjon der tjeneren må kontaktes over nettverket benytter man et medfølgende rammeverk ved navn Network Server. Dette er et tynt lag med programvare som legger seg rundt databasemotoren og tar seg av kontakt med klienter over nettverket. På klientsiden må man da benytte en annen driver.



Figur 5.1 Derby arkitektur

Det er også mulig å benytte en kombinasjon av disse to virkemåtene. Da vil en lokal klient koble til som vanlig i embedded modus, og i tillegg vil Network Server gjøre databasemotoren tilgjengelig for andre klienter over nettverket.



Figur 5.2 Derby i Embedded Server modus

Skulle det vise seg at ingen av de beskrevede virkemåtene etterkommer de nødvendige krav, er det fult mulig å implementere sitt eget rammeverk.

5.2 Bruksområder

At Derby lever inne i klientapplikasjonen, og at den er skrevet i Java, åpner for bruk i mange utradisjonelle sammenhenger. Mange applikasjoner som står alene og ikke er en del av et større system vil ofte ikke ha den luksus å kunne lagre data i et skikkelig DBMS, men må benytte egenutviklede løsninger for lagring og søk. Disse kan være problematisk å implementere skikkelig, og mange utviklere har nok ønsket å kunne benytte et DBMS. Det er her Derby kommer inn i bildet. Man kan nyte alle fordeler ved et skikkelig RDBMS, uten avhengigheten til en separat tjener. Derby kan rett og slett benyttes der det tidligere har vært utenkelig. De lave systemkravene gjør det for eksempel mulig å benytte Derby i nettlesere ved hjelp av JavaScript, eller i mobiltelefoner. Det er nesten kun fantasien som setter grenser.

I tillegg til dette kan man også selvfølgelig benytte Derby i den tradisjonelle rollen som sentral databasetjener i større systemer med flere brukere. Dog ikke i de helt store sammenhengene da Derby både mangler støtte for replisering og lagring av data på tvers av flere disker.

6 X/Open DTP og XA

XA er en standard som er en del av X/Opens Distributed Transaction Processing (DTP) modell [30]. Denne modellen opererer med tre komponenter, og XA standarden er interfacene, og bruken av disse, mellom to av disse komponentene.

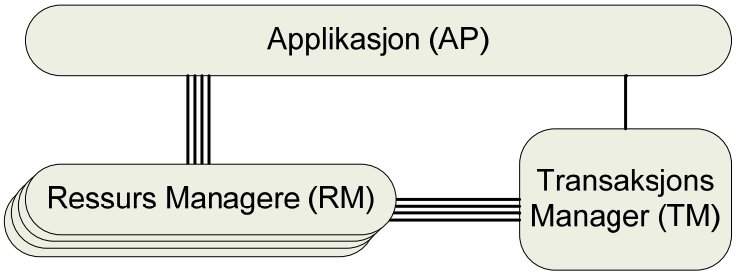
DTP modellen beskriver hvordan man kan gjennomføre distribuerte transaksjoner, spesielt med hensyn på hvordan det er mulig å gjennomføre enten commit eller rollback på alle noder. En forutsetning er at hver node er i stand til å gjennomføre lokale transaksjoner.

De tre komponentene i modellen er:

- Ressursmanagere (RM)
  Representerer en enkel node/kilde som inneholder ressurser en klient kan ha ønske om å benytte seg av. Som oftest kan hver enkelt RM ses på som en server i nettverket, men det er også fult mulig at flere RMer aksesseres via en og samme serverprosess (for eksempel flere databaser som ligger på samme DBMS).

- Applikasjon (AP)
  Dette er klientprogrammet som transaksjonelt ønsker å benytte seg av ressurser på de ulike RMene.

- Transaksjonsmanager (TM)
  Er ansvarlig for å samkjøre flere ulike transaksjoner, styre commit/rollback prosessen for hver og koordinere recovery etter feil.



Figur 6.1 X/Open DTP

En typisk gjennomføring av en transaksjon i DTP modellen kan beskrives på følgende måte:

1. AP ønsker å starte en distribuert transaksjon, og tar kontakt med TM.

2. TM tilordner så en globalt unik ID (Xid) til den nye transaksjonen. TM tar så kontakt med RMene som skal inngå og sender med den genererte Xid. Tråden som kalte TM blir assosiert mot transaksjonen ved å lagre dette i trådens kontekst.

3. AP benytter seg så av ressurser på de forskjellige RMene. Informasjon i trådens kontekst vil la en RM knytte operasjonene opp mot korrekt transaksjon. Det er da altså viktig at dette er den samme tråden som tok kontakt med TM i steg 1.

4. Etter at AP har gjennomført sitt arbeid rundt på de forskjellige RMer vil TM prøve å gjennomføre en 2-fase commit mot RMene.

   Det finnes også optimaliseringer på denne prosessen. Ved første fase i 2-fase commit kan en RM svare at den ikke har skrevet til noen av sine ressurser i løpet av transaksjonen, og vil dermed ikke inngå i fase 2. En annen optimalisering er at hvis TM vet at kun 1 RM har utført oppdatering på data kan fase 1 droppes. Den aktuelle RM vil da bestemme transaksjonens utfall alene.

Skulle det skje en feil er det TMs oppgave å rydde opp. Et eksempel kan være hvis en transaksjon akkurat har rukket å gjøre ferdig steg 3 før nettverket mellom noen av komponentene (AP, RM eller TM) faller ned. Det vil da være mulig at låser, eller kunnskap om transaksjonen, henger igjen på noen av RMene. TM må da sørge for å kontakte de involverte RMene og rydde. Et annet eksempel er hvis TM mister forbindelsen til en eller flere RMer halvveis gjennom fase 2 av 2-fase commit. Noen RMer vil da ha gjennomført en commit, mens andre vil stå og vente. Data vil da globalt sett være inkonsistent. Her også vil det være opp til TM å kontakte de aktuelle RMene og sørge for at commit blir utført der også.

Selve XA interfacene er å finne mellom RMene og TM. Som beskrevet ovenfor er det disse som gjør det mulig for TM å holde følge med hvilke RMer som er medlem av en gitt transaksjon, samt sørge for unison gjennomføring av enten commit eller rollback. Utformingen av disse 2 interfacene er som følger:

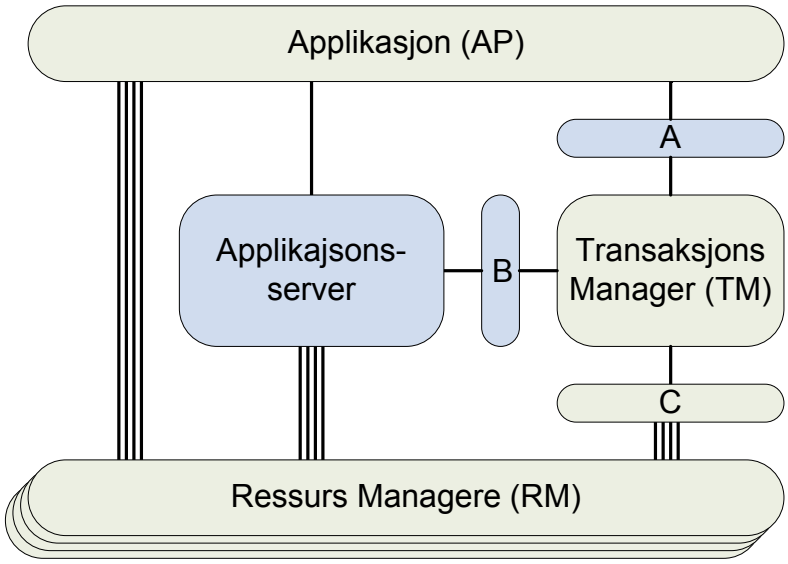
På komponent	Metodenavn	Beskrivelse
TM	ax_reg	Lar en RM bli en del av en transaksjon på eget initiativ
	ax_unreg	Lar en RM selv melde seg ut av en transaksjon
RM	xa_close	Avslutter APs bruk av en RM
	xa_commit	Ber en RM utføre commit av en transaksjon
	xa_complete	Sjekker om en asynkron xa_ operasjon er ferdig
	xa_end	Disassosierer en tråd fra en transaksjon
	xa_forget	Enkelte ganger kan en RM gjennomføre direkte commit i en 2-fase commit. Dette gjør den på eget initiativ, under den forutsetning at den husker dette. Denne metoden gir RM lov til å glemme dette.
	xa_open	Forbereder RM for bruk av AP
	xa_prepare	Ber RM gjøre seg klar til commit og returnere status.
	xa_recover	Henter en liste over transaksjoner en RM har utført xa_prepare på, eller commitet direkte på egen hånd.
	xa_rollback	Ber RM gjennomføre rollback
	xa_start	Assosierer en tråd med en transaksjon

6.1 Java JTA

Dette er en spesifikasjon som bringer konseptene i X/Opens DTP modell inn i Java, med noen modifikasjoner for å gjøre det mer passende for Java og J2EE. Det er viktig å merke seg at JTA [31], som DTP, er en spesifikasjon og ikke ferdig programvare.

I X/Opens DTP var det ikke fastsatt noen interfacer annet enn mellom TM og RMer. Derimot i JTA er det også spesifisert interfacer for bruk mellom AP og TM. En annen forskjell er at JTA skiller mellom to forskjellige typer applikasjoner. En der applikasjonen selv styrer transaksjonens grenser, og en annen der en applikasjonsserver styrer disse.

En grafisk fremstilling av JTA vil kunne se slik ut:



Figur 6.2 Java JTA

De beige boksene på figuren er det som er sammenfallende med X/Opens DTP modell. De blå er hva som er å finne i tillegg i JTA De tre interfacene som JTA spesifikasjonen inneholder er merket med A, B og C.

A. **UserTransaction (javax.transaction.UserTransaction)**
   Gir selve applikasjonen mulighet til å bestemme en transaksjons grenser. Om og når commit gjennomføres, og hvilke RMer som inngår.

B. **TransactionManager (javax.transaction.TransactionManager)**
   Lar en applikasjonsserver ta hånd om transaksjonen på vegne på selve applikasjonen.

C. **XAResource (javax.transaction.xa.XAResource)**
   Er en ren mapping av XA interfacet i X/Opens DTP modell, altså alle metoder som RM tilbyr. Eneste forskjellen i protokoll er at xa_open metoden blir automatisk kalt når man henter en referanse til en RM.

   Dette interfacet er vanlig å se implementert i databasesystemer som støtter JDBC standarden, da dette er nødvendig for å inngå i en transaksjonell J2EE stack.

Benytter man seg av en applikasjonsserver er det vanlig at disse inneholder en ferdig transaksjonsmanager. Det eneste man da trenger å passe på er at de ønskede ressursmanagerne støtter XAResource interfacet, og å implementere selve applikasjonen som kjører på topp.

7 Egen løsning

Presenterer her hvordan det er mulig å gjennomføre selve byggingen av et cluster som tillater god skalering av leseforespørsler. Det skal fortsatt være mulighet til å oppdatere data enkelt og transaksjonelt, men denne funksjonaliteten vil ikke gis fokus ytelsesmessig. Det kan også være aktuelt å fire på kravene til hvor oppdateringer kan komme fra.

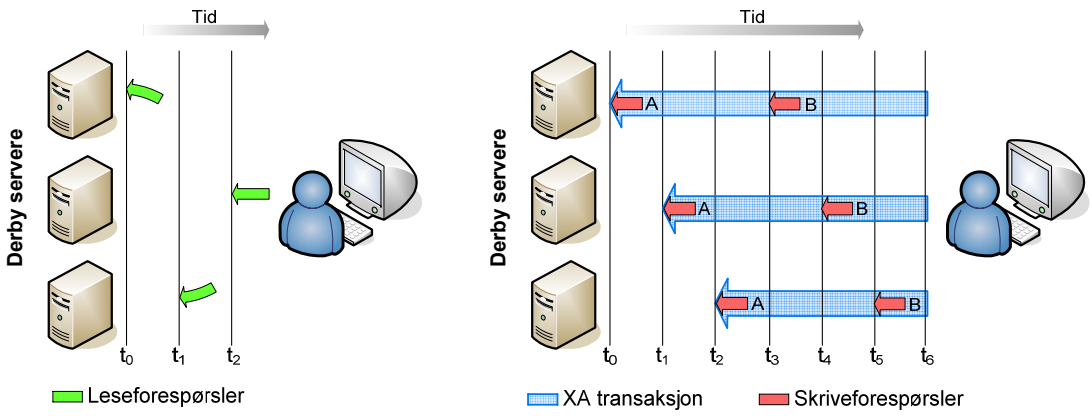
Som oppgavetekst tilsier er det kun aktuelt å innføre logikk i Derbys JDBC driver. Testresultatene i 4.1.2 støtter opp under at dette kan være en god løsning.

Med tanke på hvordan data lagres på nodene er det mest aktuelt å holde fullstendig repliserte data på alle noder. Partisjonering ville vært svært komplisert å implementere uten å endre databasemotoren i Derby.

Lesing av data vil kunne foregå rimelig enkelt. Driveren får ved innlasting tildelt et antall servere den skal benytte. Ved oppkobling velges bare en tilfeldig en av dem og transaksjonen gjennomføres som normalt ved bruk av en enkelt server. Skulle det skje at den serveren som oppkoblingen rettes mot har krasjet, kan driveren velge en annen tilfeldig server fra de resterende og prøve på nytt.

For å klare å gjøre transaksjonelle oppdateringer på tvers av nodene tas Derbys støtte for XA i bruk. En transaksjon som gjennomfører en oppdatering vil da starte med å opprette en XA transaksjon på alle noder. Deretter vil de faktiske oppdateringene gjennomføres på alle noder. Til slutt foretas det en 2-fase commit.

Siden dette systemet har leseforespørsler i fokus vil det være aktuelt å gjennomføre oppdateringer på en ikke-intrusiv måte. Altså sørge for å beslaglegge minst mulig ressurser på serverne samtidig. En måte å gjøre dette på er å ikke parallellisere oppdateringer, men i stedet behandle serverne serielt. En transaksjon vil jo gjerne bestå av flere statements, så hvis hver av dem gjennomføres på alle servere før den neste behandles vil ikke mer enn 1 server belastes om gangen.



Figur 7.1 Transaksjoner i egen løsning

En løsning som dette vil ha mulighet til å skalere relativt godt. Dog må hver node, grunnet replisering, ha mulighet til å lagre alle data selv. Dette begrenser datamengden ved at den ikke kan spres over flere noder. Sett på ytelse derimot er det muligheter for å skalere langt. Siden all logikk ligger i klientens driver er det ikke noe behov for at serverne kommuniserer seg imellom. Prosesseringskraften som er tilgjengelig til fordeling øker i takt med antall klienter, og man har dermed ingen risiko for at et knutepunkt på serversiden blir en flaskehals.

Et problem med en slik distribuert løsning er at man mister kontroll over transaksjonsflyten, det finnes ingen sentralisert bevissthet om hva som foregår i systemet. Dette bringer med seg noen utfordringer og begrensninger for en implementasjon når data skal oppdateres.

I det følgende kapitlet vil det trekkes frem fordeler og ulemper ved en slik løsning. Konsekvensene av disse vil så evalueres, og en mer spesifikk og detaljert arkitektur vil bli foreslått.

7.1 Fordeler

7.1.1 Robust ved lesing

På serversiden finnes det ingen punkt som er kritiske for virkemåten til hele systemet. Løsningen vil derfor være svært robust når det gjelds primærfunksjonen, å la klienter lese data. Kræsj på en server vil kunne påvirke ytelsen ved at klienter prøver å koble til en node som ikke svarer, men forutsatt at resterende noder ikke er oversvømt av forespørsler vil klienten til slutt få lest det den ønsker.

7.1.2 Skalerbarhet

Siden det ikke finnes noe sentralt punkt alle forespørsler skal gjennom, vil ytelsen globalt kunne skalere godt. Så lenge nettverksforbindelsen mellom servere og klienter ikke går full, er det aktuelt å forvente nær 100% skaleringsgrad for hver nye servernode. 2 servere yter 200%, 3 yter 300% osv.

7.1.3 Enkel implementasjon

Normalt vil flest vanskeligheter ved implementasjon av et distribuert system være knyttet til oppdatering og skriving av data. Siden det ikke stilles noen store krav til muligheten for å oppdatere data kan denne funksjonaliteten begrenses både med hensyn på ytelse og tilgjengelighet. Dette kan forenkle implementasjonen merkbart ved for eksempel å unngå recovery-situasjoner der flere noder inneholder motstridende data, eller benytte synkron og treg spredning av oppdateringer.

7.2 Ulemper

7.2.1 Økning av vranglåsfare

Hvis to klienter ønsker å kjøre en oppdatering hver vil disse to ha en økt fare for vranglås i forhold til normalt ved en enkelt server. Grunnen til dette er at rekkefølgen serverne oppdateres i velges av klienten, og kan derfor variere.

Eksempel

Man har et oppsett med 5 servere. To transaksjoner, A og B, ønsker å gjøre oppdateringer mot de samme tuplene. A begynner med server 1, fortsetter til 2 osv. B begynner i motsatt ende, først 5, så 4 osv. Når disse transaksjonene møtes vil de gå i vranglås.

En måte å redusere dette problemet på er at alle klienter utfører sine oppdateringer i samme serverrekkefølge. Problemet med to transaksjoner som ønsker å oppdatere de samme tuplene vil da løse seg ved den første serveren. Den transaksjonen som ankommer først vil få lov til å oppdatere, mens den andre pent må vente ved node 1 til den første er ferdig med sitt. Det vil fortsatt være en fare for vranglås på like linje med oppsett med kun 1 server, men man vil slik ha en mulighet for å global låsing.

En mulig måte å implementere at alle klienter gjennomfører oppdateringer i samme rekkefølge er å først sortere etter IP adresse, deretter etter portnummer hvis det skulle være noen servere som kontaktes via samme IP. En forutsetning er da at man ikke har noen brannmurer som også forsøker å balansere forespørsler inn mot serverne.

7.2.2 Lav tilgjengelighet for oppdateringer

Hvis en servernode kræsjer vil det være problematisk å gjennomføre oppdateringer. Klienten er jo den eneste som vet at en oppdatering kun ble gjennomført på server A og B, men ikke C. Og det er jo langt ifra sikkert denne vil koble til igjen for å tilbakeføre kunnskapen til systemet om at node C mangler en oppdatering.

Dette problemet kan også bli verre. Hvis det var forbindelsen mellom klienten i forrige avsnitt og node C som gjorde at oppdateringen mot C ikke ble gjennomført, kan det også hende at en annen klient ikke får oppdatert for eksempel node A på grunn av det samme. Man vil da ikke bare ende opp med en servernode som henger etter resten, men flere noder som potensielt kan inneholde motstridende oppdateringer.

I denne løsningen vil det derfor være uaktuelt å la klienter oppdatere hvis ikke alle servernoder er oppe og kjører.

7.2.3 Hengende låser og uavsluttede transaksjoner

Benyttes det XA transaksjoner til å utføre oppdateringer er det viktig at disse avsluttes på en skikkelig måte, enten ved commit eller rollback. Vanligvis når man benytter XA vil det eksistere en sentral transaksjonshåndterer som sørger for å rydde opp etter feil. I denne løsningen vil en slik ikke eksistere, og dermed oppstår det problemer med transaksjoner som henger igjen på servernodene.

Enten man leser eller skriver til en database vil det normalt settes låser. Hvis en klient brått kobles fra vil serveren ganske enkelt rulle tilbake transaksjonen og låsene fjernes. Ved bruk av XA er dette ikke tilfellet. Blir ikke transaksjonen eksplisitt avsluttet vil den henge igjen på ubestemt tid sammen med låsene. Dette vil skape problemer for videre kjøring av systemet. Avhengig av låsene vil de aktuelle radene være delvis eller helt utilgjengelig for andre klienter.

Pr i dag er dette normal oppførsel i Derby. Det jobbes aktivt med å implementere en timeout mekanisme, men den er ennå ikke på plass [32]. Denne timeouten må settes relativt høyt på grunn av at andre noder kan jo finne på å trenge tid til sin prosessering av transaksjonen. Så dette problemet kan være aktuelt selv etter timeoutmekanismen er på plass.

Et annet problem med å ikke ha en sentral transaksjonshåndterer er hvis en 2-fase commit krasjer under fase 2. Enkelte servernoder kan da ende opp med å anse transaksjonen som utført mens andre ikke. Data vil da være globalt inkonsistente, og kun den klienten som opplevde å få transaksjonen sin avbrutt vet om dette.

7.2.4 Sekvensering

Det er viktig å få med seg at XA ikke er et verktøy for replisering, kun commit/rollback av transaksjoner. Dermed vil man med flere noder som kjører oppdateringer kunne få problemer med at transaksjoner kommer i forskjellig rekkefølge på forskjellige servere. Noe som vil kunne lede til inkonsistens i dataen.

Eksempel

Anta at man har et oppsett med to servere. I dette ønsker to transaksjoner, A og B, å legge til en ny tuppel i en database. Transaksjonene kjøres fra to forskjellige klienter og kan kunne komme til å kjøre i forskjellig rekkefølge på de to serverne. Tabellen er satt opp slik at primærnøkkel blir satt automatisk (forrige nøkkel + 1). Sett at A legger inn sin rad på server 1 først, for så å legge den inn på server 2. Transaksjon B gjør det i motsatte rekkefølge. Kjører A og B samtidig vil man da kunne få forskjellige primærnøkler på tuplene på server 1 og 2.

Dette problemet vil eksistere selv om server-rekkefølgen oppdateringer gjøres i er lik for alle klienter. Selv om transaksjon A starter først vil B som starter senere kunne ”kjøre forbi” A, og man får det samme problemet som beskrevet i eksemplet ovenfor.

7.2.5 Lese eller skrive?

Hvis en klients driver skal ha to forskjellige modi for lesing og skriving medfører dette at en klient må vite på forhånd om en transaksjon skal skrive noe. Dette er ikke alltid trivielt da dette kan avhenge av hva man har lest tidligere i samme transaksjon.

En mulighet er å finne ut om man trenger å skrive i en transaksjon, for så å starte en ny for å gjennomføre skriving. Men da vil man ikke ha en transaksjon som omfatter hele operasjonen og grunnlaget for skrivingen kan potensielt forsvinne ved at noen andre også skriver. Dette kan avhjelpes ved å gjennomføre den samme lesesjekken en gang til innen den nye XA transaksjon, men dette må anses som litt tungvint.

En mulighet er å benytte XA til å også gjennomføre lesing. Problemet her er at hvis klienten eller nettverksforbindelsen krasjer vil alle leselåser denne klienten har forbli låst på ubestemt tid. Noe som vil kunne skape problemer for senere oppdateringer. Som nevnt i 7.2.3 jobbes det med en timeout mekanisme for å avhjelpe hengende XA transaksjoner, men det er ikke selvfølgelig at dette er en god nok løsning.

7.2.6 Utbygging

Ønsker man å sette inn en ny node i et kjørende cluster er man avhengig av at klientene oppdaterer JDBC urlen de bruker for å koble til.

Har man webservere som fungerer som klienter, og disse er under egen kontroll, kan man ganske enkelt gjøre endringer på en og en av dem. I verste fall betyr dette å ta ned serveren en kort tid. Det er fult mulig å lage en applikasjon som støtter å gjøre dette ”live”, men dette krever da at man forutser en evt. utbygging.

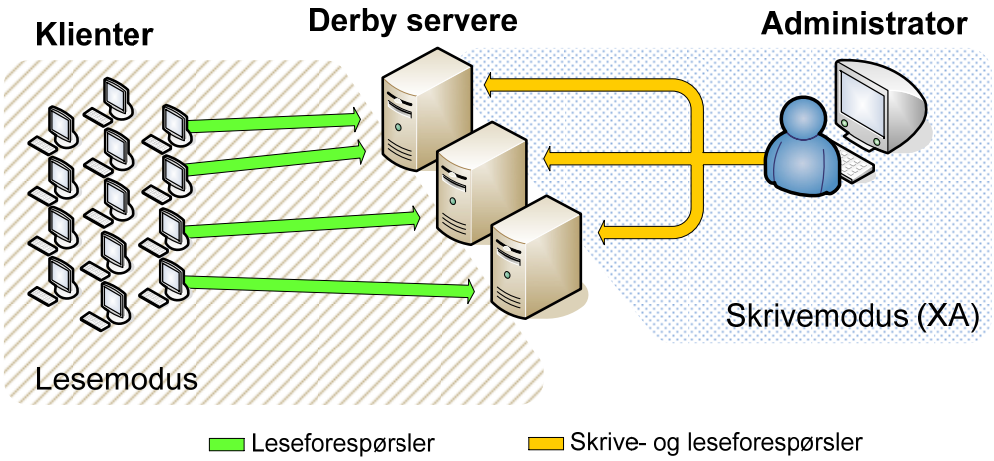
Er klientene faktiske klientmaskiner i store mengder er det verre å bytte url. En mulig løsning kan være å implementere en egen informasjonstjeneste som driveren henter informasjonen om hvilke servere som skal benyttes fra. Dette vil ikke tas med i denne oppgaven, men kan absolutt være en mulig utvei.

7.3 Designkonklusjon

Når det gjelds primæroppgaven for systemet, å lese data, er det kun positive aspekter ved en løsning der logikken ligger helt og holdent på klientsiden. Løsningen kan lages både robust og skalerbar på en svært enkel måte.

For skriving av data finnes det en del problemer, der noen er såpass alvorlige at det må innføres noen begrensninger rundt denne funksjonaliteten. Hovedpunktene er sekvensering og manglende opprydning etter uavsluttede XA transaksjoner. Å løse disse to problemene vil kreve svært mye, og anses derfor som umulige å løse i denne sammenhengen. Dette gjør at skrivefunksjonaliteten må begrenses fra at alle klienter kan skrive ved bruk av XA, til at oppdateringer kun kan skje gjennom et sentralt punkt på serversiden eller via administrator. Dette punktet vil så følge opp feilede XA transaksjoner, og serialisere oppdateringer. Ytelsen vil gå ned og klientene må skille lesing fra skriving, men de kritiske problemene vil forsvinne.

Selv om disse problemene eksisterer er det i denne oppgaven ikke aktuelt å gjennomføre noen endringer på selve Derby databasemotoren. Det hadde vært mulig å lage en separat servertjeneste som tok imot oppdateringer og la disse inn i Derbynodene, men dette vil kreve en god del arbeid og tiden som er tilgjengelig for denne oppgaven vil ikke strekke til. Løsningen i denne omgang blir da å si at oppdateringer må legges inn av administrator, og at systemet som bringer oppdateringer fra klientene til denne administratoren dessverre ikke dekkes av denne oppgaven. Selv om denne begrensningen innføres er det fortsatt aktuelt å lage en driver med to moduser. En for balanserings av leseforespørsler, og en annen som lar administrator gjøre oppdateringer ved hjelp av XA transaksjoner.



Figur 7.2 Egen arkitektur

System for å bringe oppdateringer fra klienter til administrator ikke inntegnet

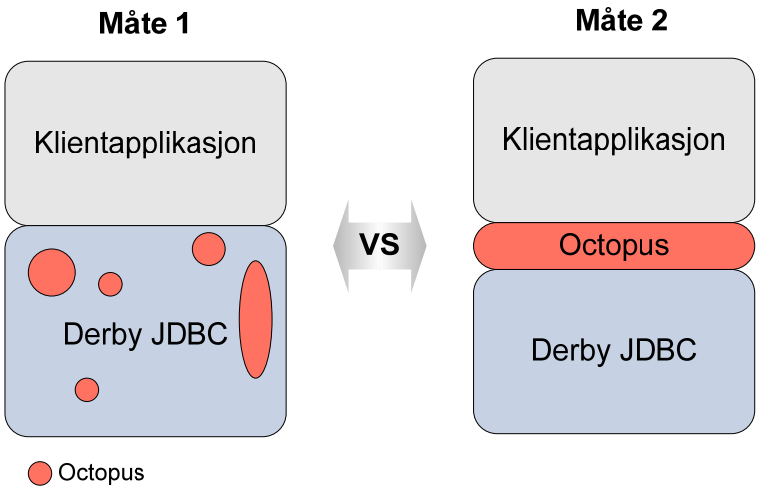
7.4 Implementasjon

For enkel referering så døpes egen løsning Octopus.

Denne implementasjonen er ikke ment å munne ut i programvare som er klar til produksjonssystemer. Sluttproduktet vil kunne regnes som en prototype myntet på forskning. Fokus vil derfor være knyttet til de "store linjene" og ikke små detaljer som for eksempel hva enkelte metoder heter, eller hvordan de kalles.

Som sagt tidligere er det aktuelt å ha to forskjellige modi på driveren, en for å balansere leseforespørsler og en annen for å gjøre oppdateringer ved hjelp av XA.

Det finnes hovedsakelig to forskjellige måter å oppnå dette på. Den første er at Derbys eksisterende JDBC driver utvides til å støtte den nye funksjonaliteten. Den andre er at det bygges et tynt lag som legges mellom klientapplikasjonen og Derbydriveren som implementerer Javas JDBC interfacer mot klientapplikasjonen.



Figur 7.3 Mulige implementasjonsmåter

Måte 2 har flere fordeler fremfor den første:

- Versjonsuavhengighet til Derbys driver
  Kommer Apache med en ny utgave av JDBC driver til Derby må alle endringer som er gjort utføres en gang til, gjerne med enkelte modifikasjoner. Ved å lage et eget lag som implementerer Javas JDBC standard ovenfor klientapplikasjonen, og benytter kun varige APIer på Derbydriveren, er det mulig å implementere en løsning med et relativt langt liv.

- Innfører ikke feil i Derbys driver
  Ved å ikke røre kildekoden i Derbys driver innføres ikke noen potensielle feil og ustabiliteter.

- Ryddig implementasjon
  Implementasjonen kan gjøres på en ryddig måte da man slipper å blande funksjonalitet i Octopus med funksjonalitet i Derbys eksisterende driver. Dette gjør det enkelt for andre å gjøre seg kjent med koden på et evt. senere tidspunkt.

Måten Octopus konseptuelt kommer til å fungere på er svært lik den en transaksjonsmanager i Java JTA gjør. Med det faktum at Derbys JDBC driver kan settes i en ressursmanagerrolle i samme rammeverk tilsier at det vil gå helt fint å implementere Octopus på måte nr 2. Altså trengs det ikke tilgang til noe mer funksjonalitet i Derbys driver enn den som er tilgjengelig via de standard JDBC interfacene den implementerer. Det vil heller ikke være noen problemer med å tilby Javas JDBC interfacer i Octopus opp mot klientapplikasjonen. Octopus blir derfor implementert på måte 2.

Å velge modus Octopus skal fungere i kan gjøres på to forskjellige måter avhengig om man benytter ClientDriver eller DataSource for å opprette forbindelser til servernodene.

Ved bruk av ClientDriver vil man benytte forskjellig formaterte JDBC url'er.

- `jdbc:derby:octopus:read://server1,server2,server3/database`
  Velger en tilfeldig server og returnerer en forbindelse som kun kan benyttes til å lese data.

- `jdbc:derby:octopus:write://server1,server2,server3/database`
  Returnerer en forbindelse som leser fra en tilfeldig valgt server, men skriver til alle sammen. Alle forespørsler vil tilhøre en XA transaksjon.

ClientDriver lastes på følgende måte:

```
Class.forName("org.apache.derby.jdbc.octopus.OctopusClientDriver")
      .newInstance();
```

Skulle man ønske å bruke DataSource måten å hente forbindelser må, velges modus ved bruk av en set-metode. Enkleste måten å forklare dette på er ved to enkle eksempler.

Å hente en forbindelse i lesemodus:

```
OctopusDataSource ds = new OctopusDataSource();
ds.setDatabaseName("database");
ds.addServer("server1");
ds.addServer("server2”);
ds.addServer("server3”);
ds.setProtocol(OctopusDataSource.READ_PROTOCOL);
Connection conn = ds.getConnection();
```

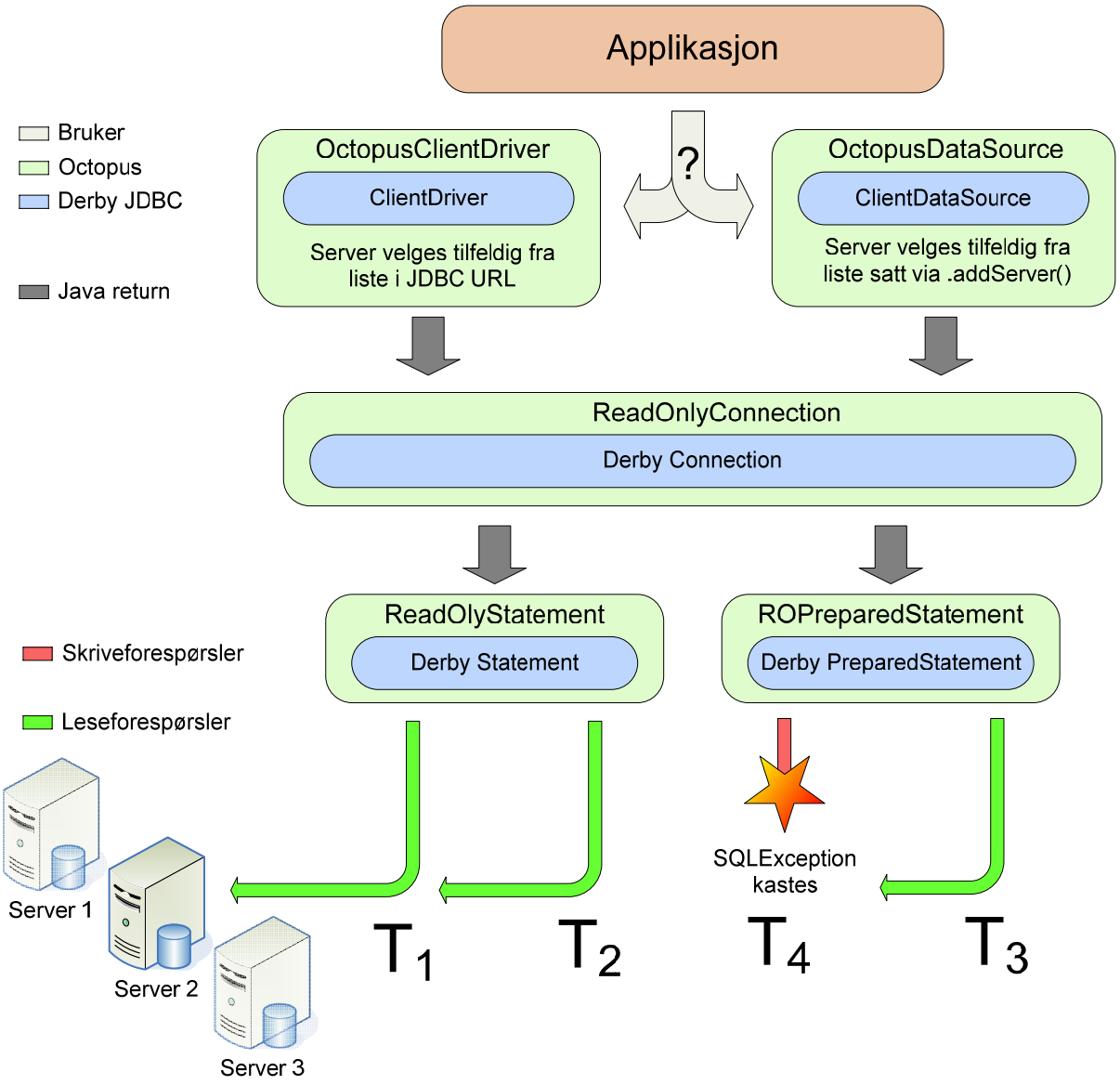
Å hente en forbindelse i skrivemodus:

```
OctopusDataSource ds = new OctopusDataSource();
ds.setDatabaseName("database");
ds.addServer("server1");
ds.addServer("server2”);
ds.addServer("server3”);
ds.setProtocol(OctopusDataSource.WRITE_PROTOCOL);
Connection conn = ds.getConnection();
```

Ved henting av en forbindelse i lesemodus er `setProtocol` metoden valgfri da dette er standard modus.

7.4.1 Lesemodus

Presenterer her detaljer om hvordan lesemodus er implementert og fungerer i Octopus.



Figur 7.4 Octopus beskjedflyt i lesemodus

T1, T2 osv angir tidspunkt i beskjedflyt

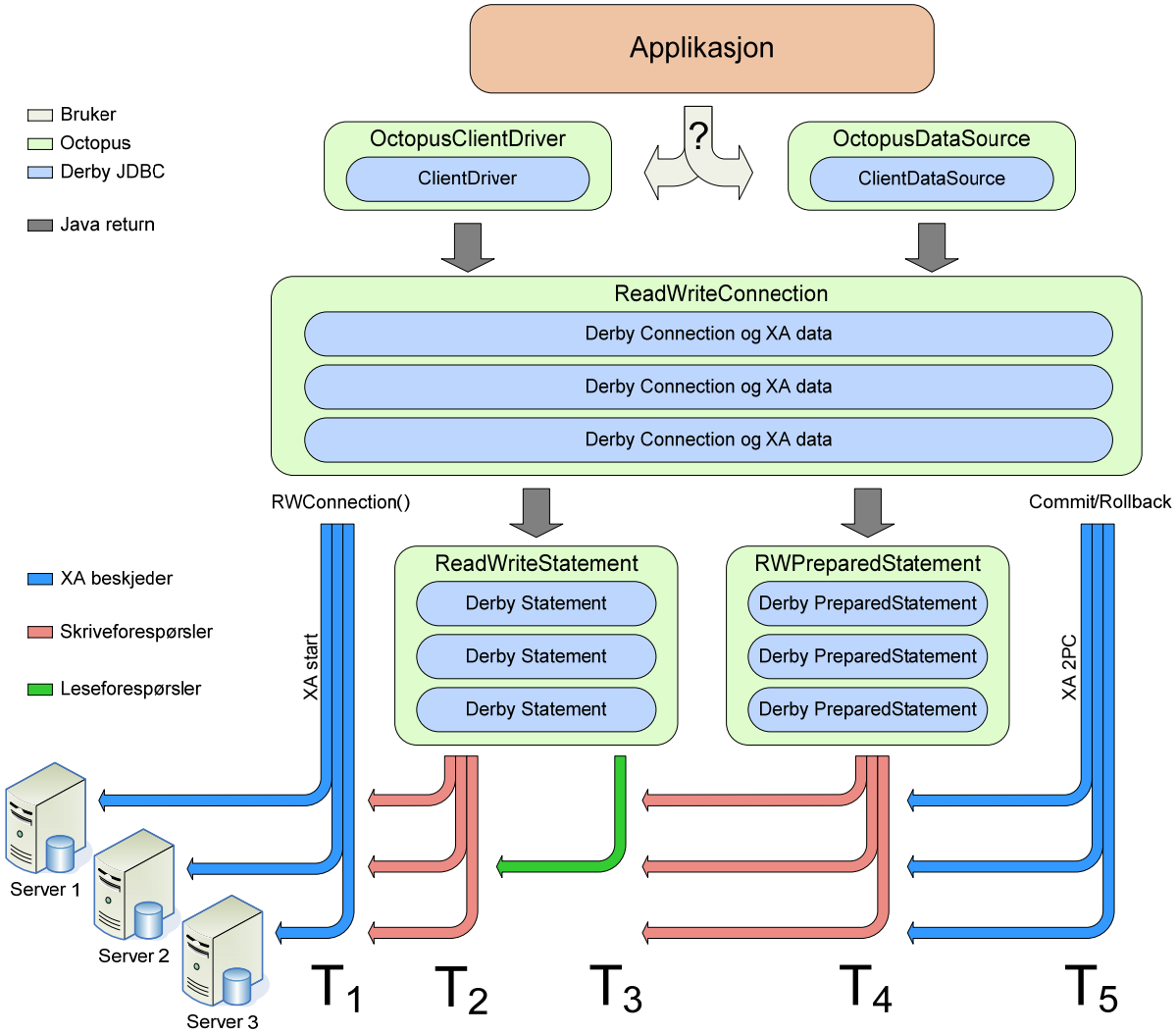
En klient vil bruke enten ClientDriver eller DataSource for å hente et Connectionobjekt. Dette objektet vil tilhøre en klasse fra Octopus, og ligge rundt et Connectionobjekt fra Derbys driver. Statement- og PreparedStatementobjekter kan hentes ut fra dette forbindelsesobjektet. Disse er oppbygd på samme måten, de ligger rundt et tilsvarende objekt fra Derbys driver. På denne måten er det mulig å begrense tilgang til metoder som typisk vil brukes for å oppdatere data, uten at applikasjonen må forholde seg til noe annet enn standard JDBC tankegang. Skulle en klient kalle en metode som er sperret grunnet dens mulighet til å gjøre oppdateringer vil et unntak kastes.

Ved uthenting av forbindelsesobjektet fra ClientDriver eller DataSource velges en tilfeldig node av serverne. Denne noden vil være den som Octopus jobber mot helt til et nytt forbindelsesobjekt hentes ut. Alle statements og preparedstatements som benyttes vil altså være rettet mot den samme serveren.

For å utføre selve valget av servernode benyttes et tilfeldig valg. Skulle den valgte serveren være nede, vil et nytt tilfeldig valg gjøres blant de resterende nodene. Kunnskapen om at en server er nede vil ikke lagres mellom hver gang en ny forbindelse opprettes. Hvorvidt dette er den beste løsningen kan diskuteres. Hvis en klient ofte gjør nye tilkoblinger og flere servernoder er nede vil denne klienten oppleve at ytelsen er lav. Har man en løsning hvor klienten er smartere, og kun prøver de serverne som den vet er oppe, vil man kunne få en stor overbelastning på de nodene som er oppe og kjører. Den beste løsningen vil kanskje være et kompromiss mellom å prioritere responstid på hver enkelt klient med muligheten for overbelastning på flere servernoder, eller dårligere responstid og mindre sjanse for ytterligere serverkrasj.

7.4.2 Skrivemodus

Vil her presentere detaljer om hvordan skrivemodus er implementert og fungerer i Octopus.



Figur 7.5 Octopus beskjedflyt i skrivemodus
T1, T2 osv angir tidspunkt i beskjedflyt

At de ulike objektene som hentes fra Octopus inneholder de tilsvarende objektene fra Derbys driver er svært likt som lesemodus. Forskjellen ligger i at forbindelser og statements vil ikke

bare inneholde ett tilsvarende Derby-objekt, men ett pr server slik at det er mulig å sende oppdateringer utført i et statement til alle servernoder.

En annen forskjell er at ved uthenting av forbindelsesobjekter vil disse starte en XA transaksjon som involverer alle servernodene. Ved commit vil forbindelsesobjektet kjøre en 2-fase commit ut mot alle servernoder, eller ved rollback vil rollback bli kalt på alle noder. Om flere leseforespørsler, eller oppdateringer, blir kalt gjennom et forbindelsesobjekt som akkurat har utført en commit eller rollback vil dette sørge for å starte en ny XA transaksjon på samme måte som først.

Selv om alle oppdateringer vil bli sendt til alle servere, vil lesing kun foregå mot en tilfeldig node. Denne noden vil velges på nytt for hver leseforespørsel.

7.4.3 Begrensninger

Enkelte av begrensningene er allerede omtalt på tidligere tidspunkt, men vil også bli tatt med her for oversiktens skyld.

Lesemodus

- støtter ikke callable statements
- støtter ikke batcher
- på Statement og PreparedStatement er ikke execute() eller executeUpdate() støttet
- bruk av update-metoder i ResultSet må ikke brukes
- støtter ikke pooling av forbindelser

Skrivemodus

- støtter ikke callable statements
- krever at alle servernoder er oppe
- metadata på Connection-objekter er ikke tilgjengelig
- uthenting av warnings fra Connection, Statement eller Preparedstatement støttes ikke
- savepoints støttes ikke
- autocommit støttes ikke, klienten må eksplisitt kalle enten commit eller rollback
- alle set-metoder i PreparedStatement som bruker datastrømmer er ikke støttet
- bruk av update-metoder i ResultSet må ikke brukes
- støtter ikke pooling av forbindelser

8 Testing

8.1 Maskinvare

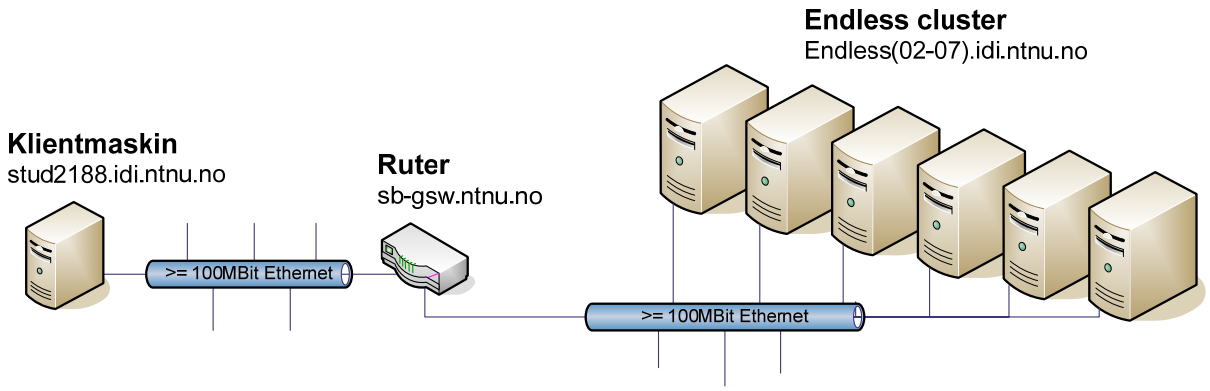
Som servernoder vil 6 maskiner fra IDIs Endless cluster bli benyttet. Alle disse maskinene er like både med tanke på program- og maskinvare. For å generere last vil en maskin på undertegnedes arbeidsplass på datasalen Fiol benyttes. Alle maskinene vil være tilkoblet det samme nettverket på 100Mbit, og det er kun 2 hopp (1 ruter) mellom klientmaskin og clusteret. Dette er de samme maskinene som ble benyttet til testing i forrige semesters prosjektoppgave.

Klientmaskinen har følgende spesifikasjoner:

- 1x Intel 2,4 GHz prosessor med 512 KB cache
- 512 MB primærminne
- Ubuntu Linux (Dapper Drake)
- Java VM 1.6.0-b105

De aktuelle nodene i clusteret:

- Vertsnavn: endless(02 – 07).idi.ntnu.no
- 2x AMD 1,4 GHz prosessor med 256KB cache
- 1024 MB primærminne
- Debian Linux (testing/unstable pr dags dato)
- Java VM 1.5.0_08
- Databasen ligger lagret på en Seagate 18GB SCSI disk med 10 000 RPM. Dette er samme disk som operativsystemet kjører fra.



Figur 8.1 Maskinoversikt

8.2 Programvare

8.2.1 Derby

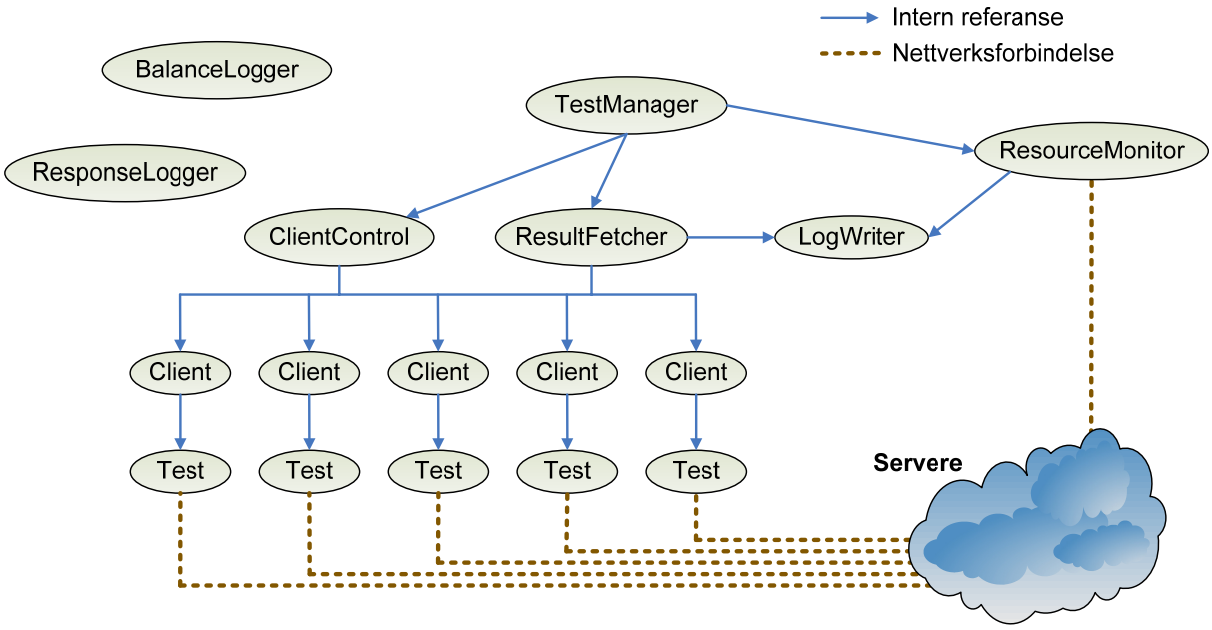
Versjonen av Derby som benyttes er 10.1.3.1 og det er ikke gjort noen endringer fra standard konfigurasjon annet enn at klienter må benytte et satt brukernavn og passord.

8.2.2 Testklient

For å generere last benyttes en egenprodusert testklient [33]. Dette er den samme som ble benyttet i forrige semesters prosjektoppgave, men det er gjort noen utvidelser for å kunne loggføre responstider og balansering av forespørsler. En grov oversikt over funksjonalitet:

- kjøre SQL spørringer fra et gitt antall samtidige tråder
- loggføre antall vellykkede og feilede transaksjoner pr sekund
- overvåke ressurser på alle maskiner som inngår i testprosedyren
- loggføre responstider for alle forespørsler
- loggføre hvilke noder forespørsler rettes mot

Når det gjelds designet er dette svært modulært slik at det er enkelt å gjøre tilpasninger.



Figur 8.2 Testklient arkitektur

TestManager

Er ansvarlig for å styre

- hvor mange klienter som kjører til enhver tid.
- hvordan og når innhenting av resultater skal foregå.
- hvilke maskiner som skal overvåkes ressursmessig

Denne komponenten er mer en rolle enn en faktisk implementert klasse. Meningen er at man skal skrive en ny TestManager for hver gang man ønsker å endre hvordan en testkjøring foregår. Tenk på denne som en ressigør for hvordan testkjøring skal skje.

ClientControl
Tar seg av oppstart og nedstengning av klienttråder som genererer last.

ResultFetcher
Henter jevnlig inn hvor mange vellykkede transaksjoner som er gjennomført.

ResourceMonitor
Kontakter Muninprosesser på angitte maskiner, henter og lagrer den innsamlede statusinformasjonen.

ResponseLogger
En utvidelse som er gjort i forbindelse med denne oppgaven. Denne er ansvarlig for å loggføre tiden det tar å utføre en transaksjon. Loggfilen er tekstbasert og hver linje representerer en fullført transaksjon. Linjen vil bestå av 3 felt:

- en tekststreng for å identifisere typen måling
- antall sekunder fra teststart linjen ble notert
- antall millisekunder det tok å fullføre den aktuelle transaksjonen

Totalt vil en linje typisk være på 12-14 tegn (bruker 14 tegn i videre regning). I tillegg vil det være ett tegn som representerer linjeskift. Siden hver bokstav tar 16 bit i Java vil denne linjen være på totalt:

$$15 \times 16 = 240 \text{ bit}$$

Harddisken i testmaskinen klarer å skrive opptil 50MB/s. Regnes dette om til maks antall transaksjoner som kan loggføres pr sekund blir dette:

$$\frac{(50 \times 1024^2 \times 8)}{240} = 1\ 747\ 626 \text{ TPS}$$

Noe som må kunne sies å være godt over det som er forventet å se av testresultater.

Grunnet enkel implementasjon og svært fleksibel bruk er denne implementert på en statisk måte. Typisk vil Test-komponenten benytte denne loggeren direkte. TestManager vil være ansvarlig for klargjøring av denne komponenten før teststart, hvordan resultater skal lagres.

BalanceLogger
Også en utvidelse som er gjort i forbindelse med denne oppgaven. Er ansvarlig for å loggføre nye forbindelser som klientene gjør. Typisk vil en klienttråd kun gjøre en slik oppkobling når den starter, for så å gjenbruke den eksisterende forbindelsen på senere transaksjoner. Dette er

grunnet ressursene som trengs på klientmaskinen for å kontinuerlig gjøre nye oppkoblinger. Datamengden som skrives til denne loggen vil altså ikke være av betydning.

Av samme årsaker som for ResponseLogger er implementasjonen utført på en statisk måte.

Bruksmønsteret vil også være det samme som for ResponseLogger.

Client
Er ansvarlig for å utføre en gitt transaksjon i løkke, og huske antall vellykkede og feilede kjøringer.

Test
Spydspissen i systemet. Det er i denne komponenten selve testtransaksjonen er definert. Etter kjøring vil det tilhørende Client-objektet varsles om suksess eller feil.

Typisk vil man skrive en ny klasse som brukes til hver testkjøring.

8.2.3 Munin

Munin [34] benyttes av testklienten for å overvåke enkelte nøkkelverdier når det gjelds belastning av maskinene som brukes. Munin har en todelt virkemåte. På alle maskiner som skal overvåkes kjøres en serverprosess som tar imot oppkoblinger og gir ut statusinformasjon. Den andre delen består av en prosess som jevnlig kobler opp mot alle disse serverprosessene og genererer statistikk.

Det er kun serverprosess-delen av Munin som vil bli benyttet da testklienten har overtatt rollen med å sanke inn informasjon. I tillegg vil det kun bli samlet informasjon som ikke krever noe særlig prosessering for skaffe. Forstyrrelser på testresultatene vil dermed være svært små, om ikke helt fraværende. Skulle det være at det likevel finnes noen forstyrrelser vil bruken av Munin holdes konstant i løpet av all testing. Dvs. alle noder vil bli overvåket selv om det kun testes mot et mindretall av noder. Da vil ressursmengden som er tilgjengelig på testmaskinen til å kjøre transaksjoner holdes konstant.

Versjonen som er i bruk er 1.2.4.

8.3 Testdata

I alle tester vil den samme tabellen brukes som datamengde. Den vil bestå av 524 288 rader, der hver rad er bygd opp av et 32bit Integerfelt som primærnøkkel og et tilfeldig tekstfelt på 96 tegn.

Tabellen opprettes med følgende SQL setning:

```
CREATE TABLE testdata (
      id INTEGER NOT NULL,
      data VARCHAR(96) NOT NULL,
      PRIMARY KEY (id)
);
```

Den totale datamengden vil være:

$$\frac{((32/8) + 96) \times 524\,288}{1024^2} = 50 \text{ MB}$$

Denne datamengden er lik den som ble brukt i forrige semesters prosjektoppgave. Testresultater kan dermed lett sammenlignes.

8.4 Leseytelse

8.4.1 Belastning

Siden hovedfokus under testing vil være på balansering og skalering er det viktig at ikke maskinen som genererer last blir en flaskehals. For å være sikker på at dette ikke blir et problem vil hver transaksjon være relativt tung å prosessere for servernodene.

Hver transaksjon vil be en servernode om å regne ut gjennomsnittet av primærnøklene i et tilfeldig valgt intervall. Størrelsen på intervallene vil være fast, og spenne seg over 1000 rader.

```
SELECT AVG(id) FROM testdata WHERE id BETWEEN X AND Y;
```

Hvor X er et tilfeldig tall mellom 1 og 523288 (1000 mindre enn antall rader), og Y er lik X+1000.

Erfaringer fra tidligere prosjektrapport tilsier at en singel server med Derby greier å ta unna ca 130 TPS med denne datamengden og forespørsel. Totalt vil dette bli opp mot 800 TPS med alle 6 nodene, noe som er godt innenfor grensen til hva ResponseLogger komponenten i testklienten klarer å skrive til disk.

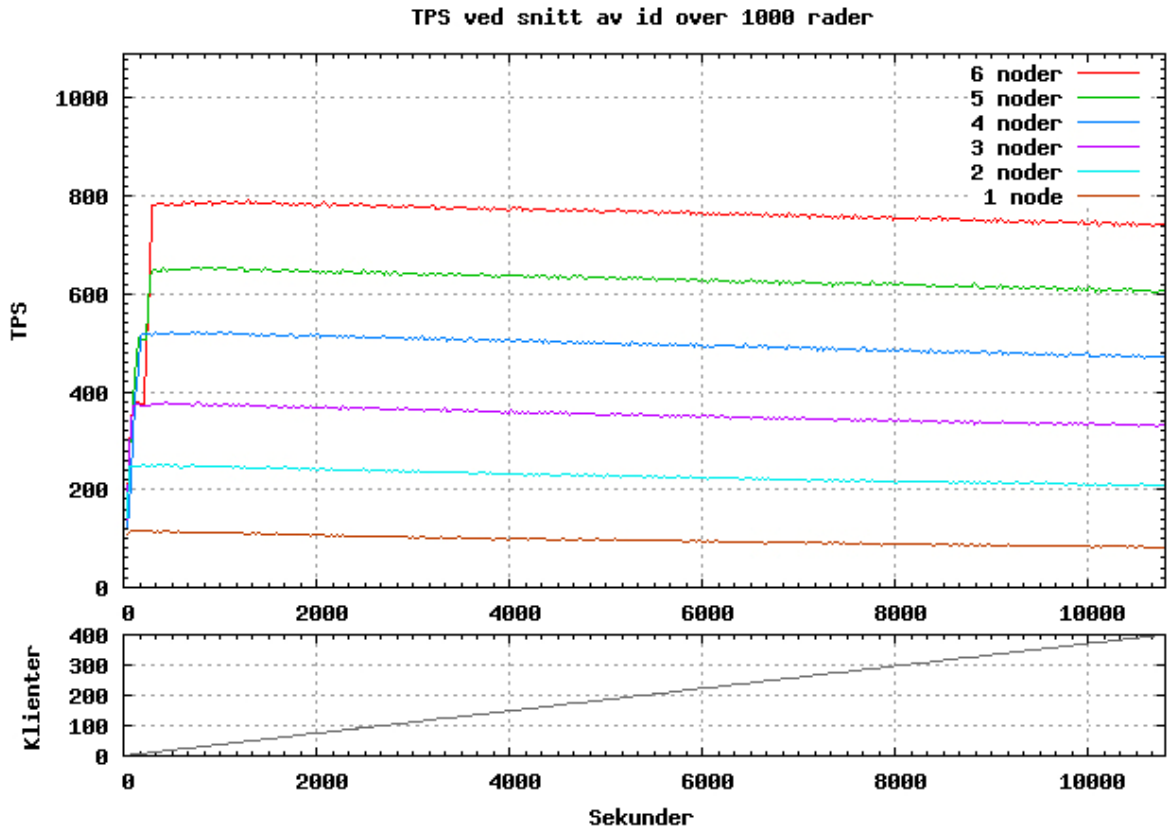
8.4.2 Konfigurasjon

Hver test kjøres i 3 timer. Antallet samtidige klienter starter på 1, og øker med 1 hvert 27. sekund. På slutten av testen vil det da være 400 samtidige klienter.

Deretter legges det til en node, og løpet beskrevet over gjentas helt til det er 6 noder totalt.

8.4.3 Resultater – TPS

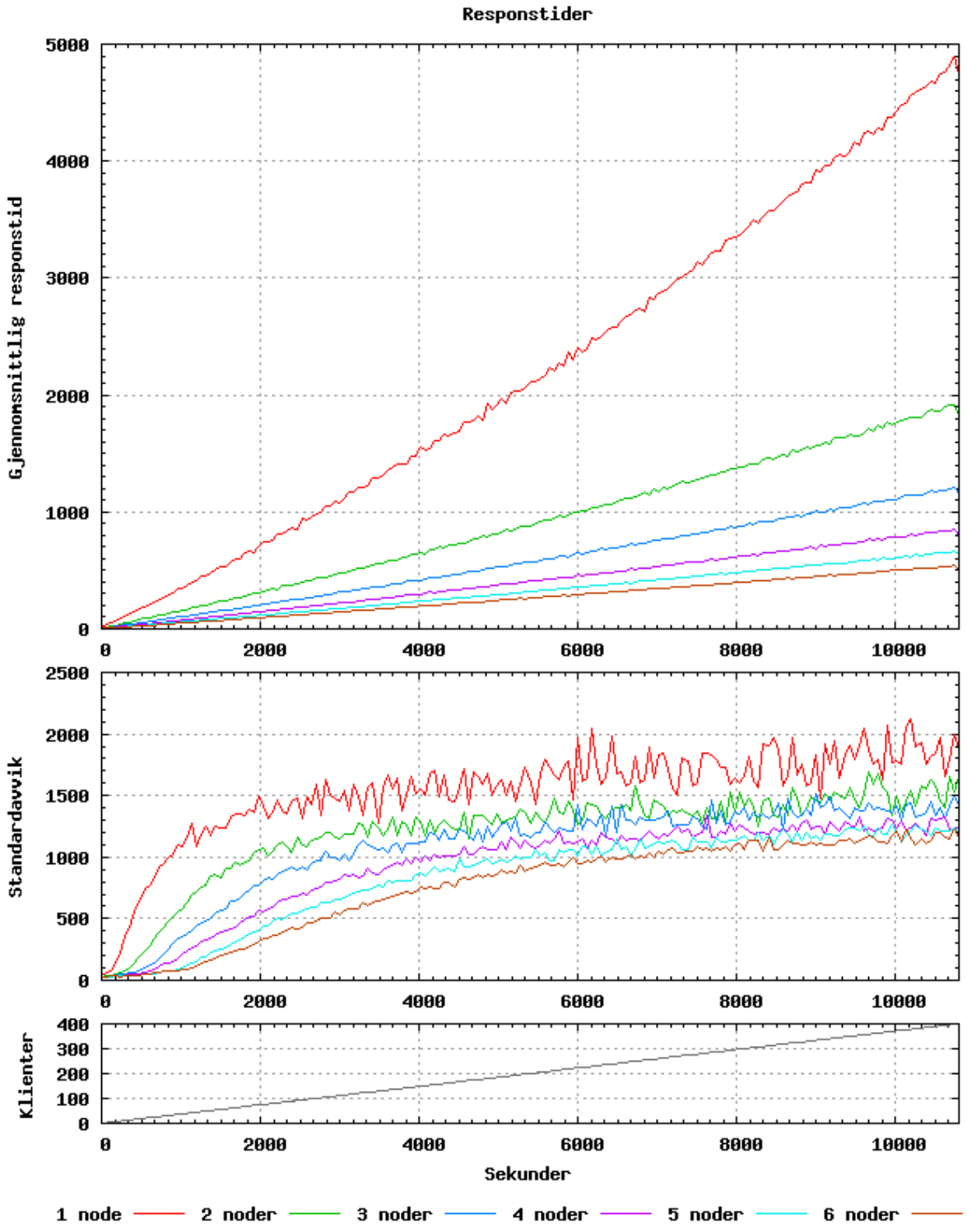
Denne grafen viser hvor mange transaksjoner som gjennomføres pr sekund mot et gitt antall servernoder.



Figur 8.3 Testresultater - lesing

8.4.4 Resultater – responstider

Under testing ble det loggført hvor mange millisekunder hver eneste transaksjon brukte på å bli gjennomført. Både responstider og standardavvik er regnet ut som gjennomsnittet over 60 sekunders perioder.



Figur 8.4 Responstider med standardavvik

For å vise detaljer i fordeling av tider brukes såkalte ”scatterplots”. Her vises responstider hver som en rød prikk. X-komponenten er antall sekunder ut i testing tiden ble notert, og y-komponenten er hvor mange millisekunder transaksjonen tok.

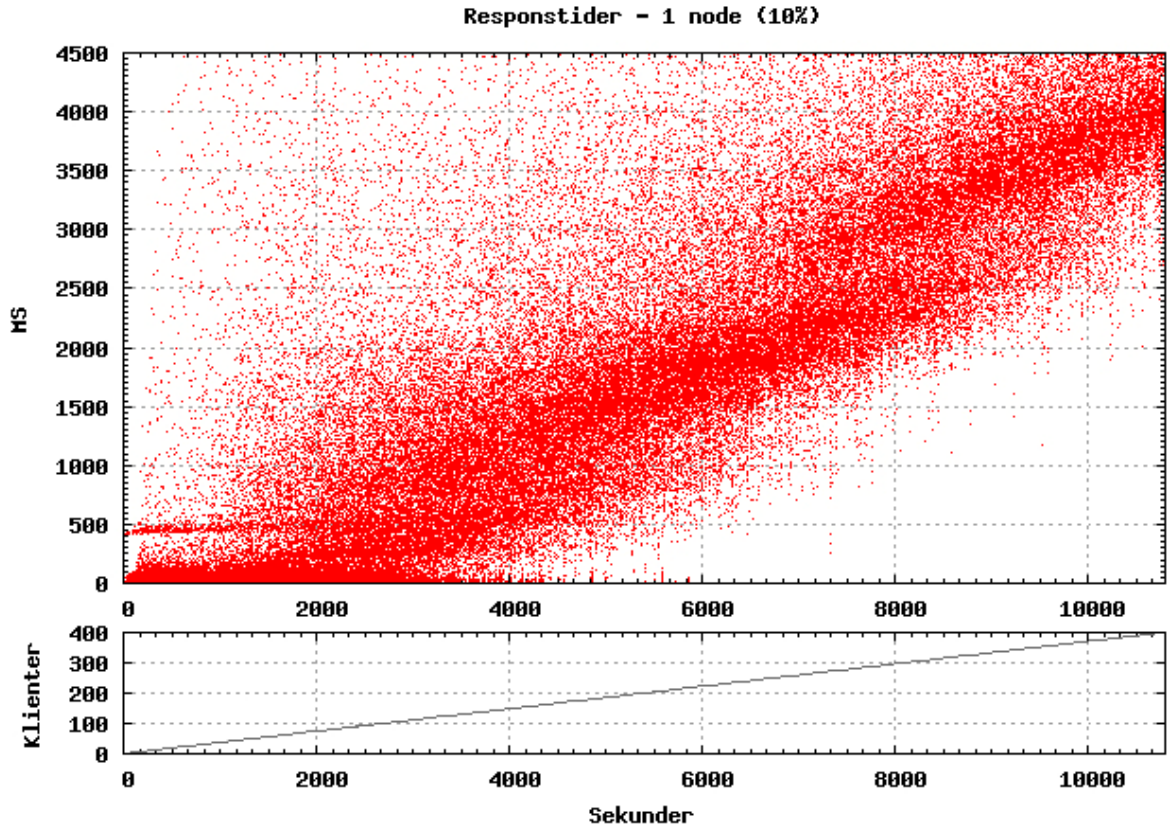
Skulle absolutt alle tidene bli plottet ville resultatene har vært vanskelig å lese. Grunnen til dette er at flere punkter ville ha ligget oppå hverandre og gjort det vanskelig å få et inntrykk

om hvor de helt store ansamlingene var. For å løse dette problemet plottes kun en del av tidene:

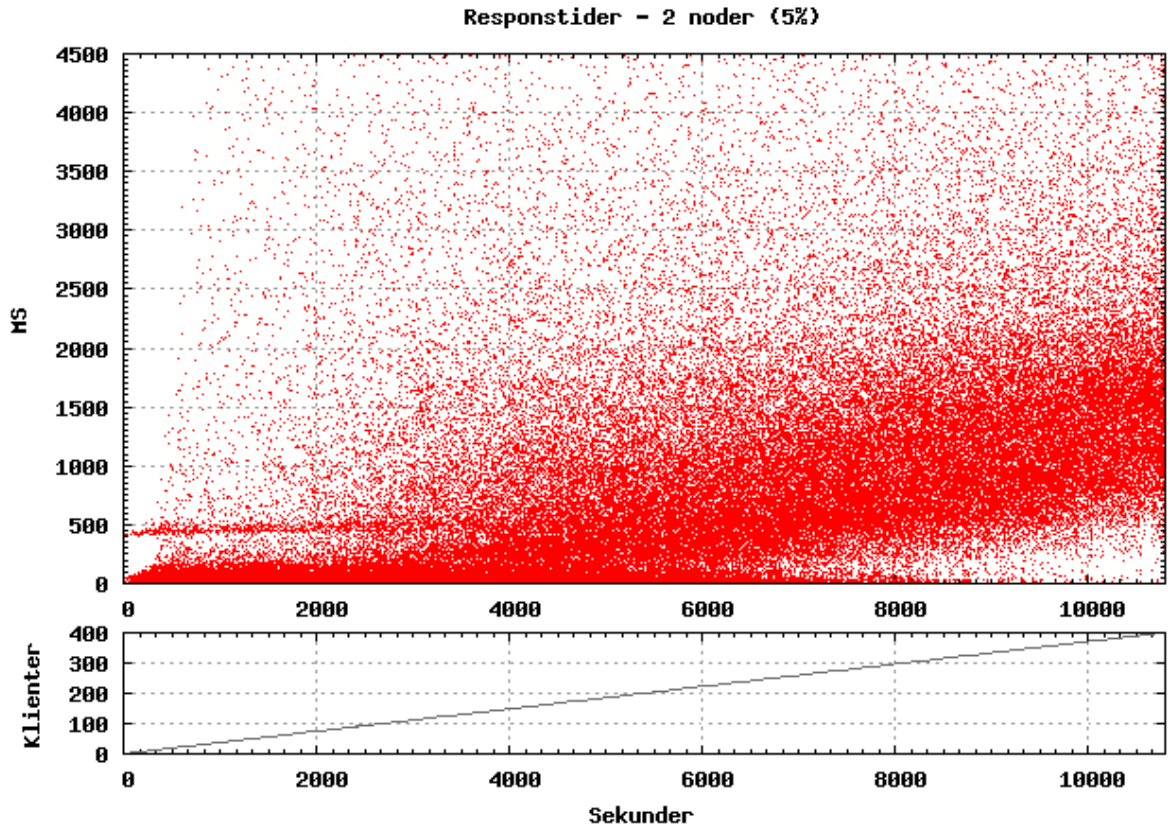
$$\frac{100\%}{(10 \times \text{antall noder})}$$

Måten utvalget gjøres på er at et program vil løpe gjennom datafilene og kopiere hver 10xN linje over i en annen fil. Data i den nye filen vil så bli brukt for å generere grafen. Dette gjør at antallet punkter på grafene holdes likt selv om antallet noder og resultater øker, og problemet med overliggende punkter kommer ikke tilbake. Ved å kun plotte et utvalg av tidene vil man kunne gå glipp av "hendelser", men siden datamengden er så stor regnes ikke dette for å være et betydelig problem.

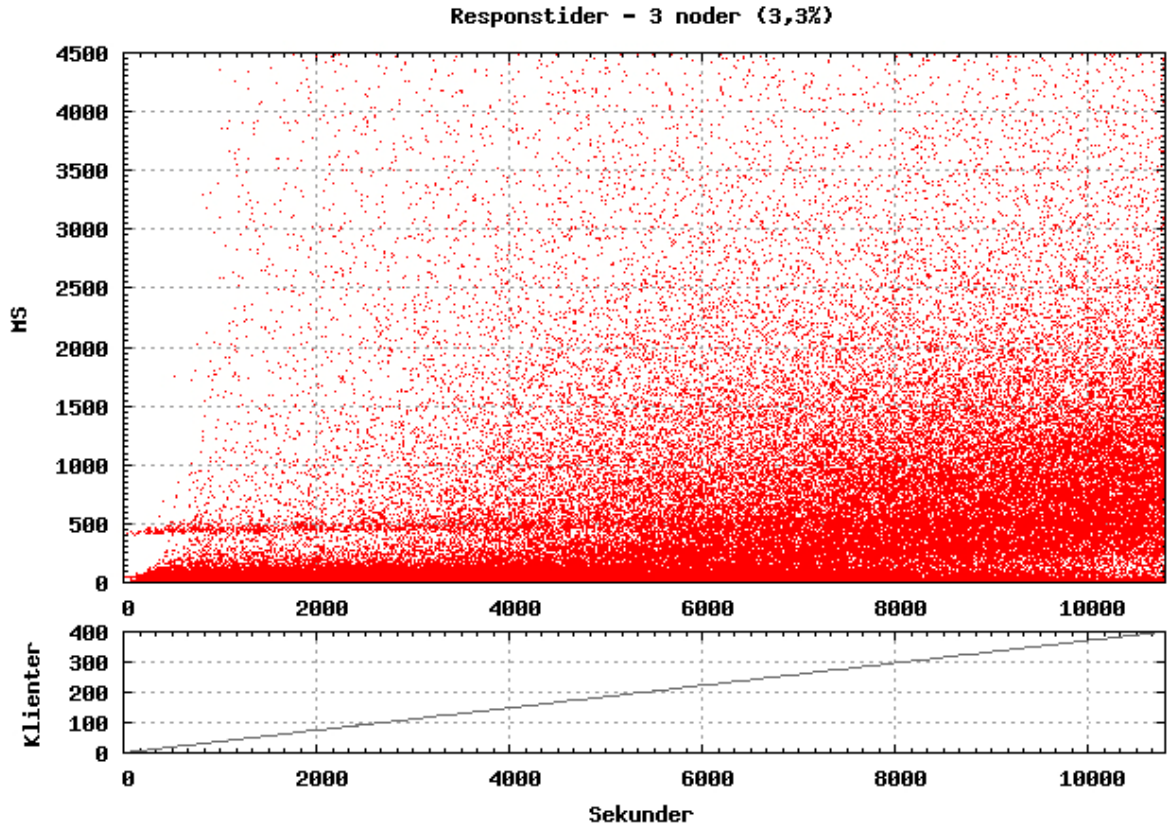
Hvor mange prosent av tidene som plottes for hver gang er notert i grafenes titler.



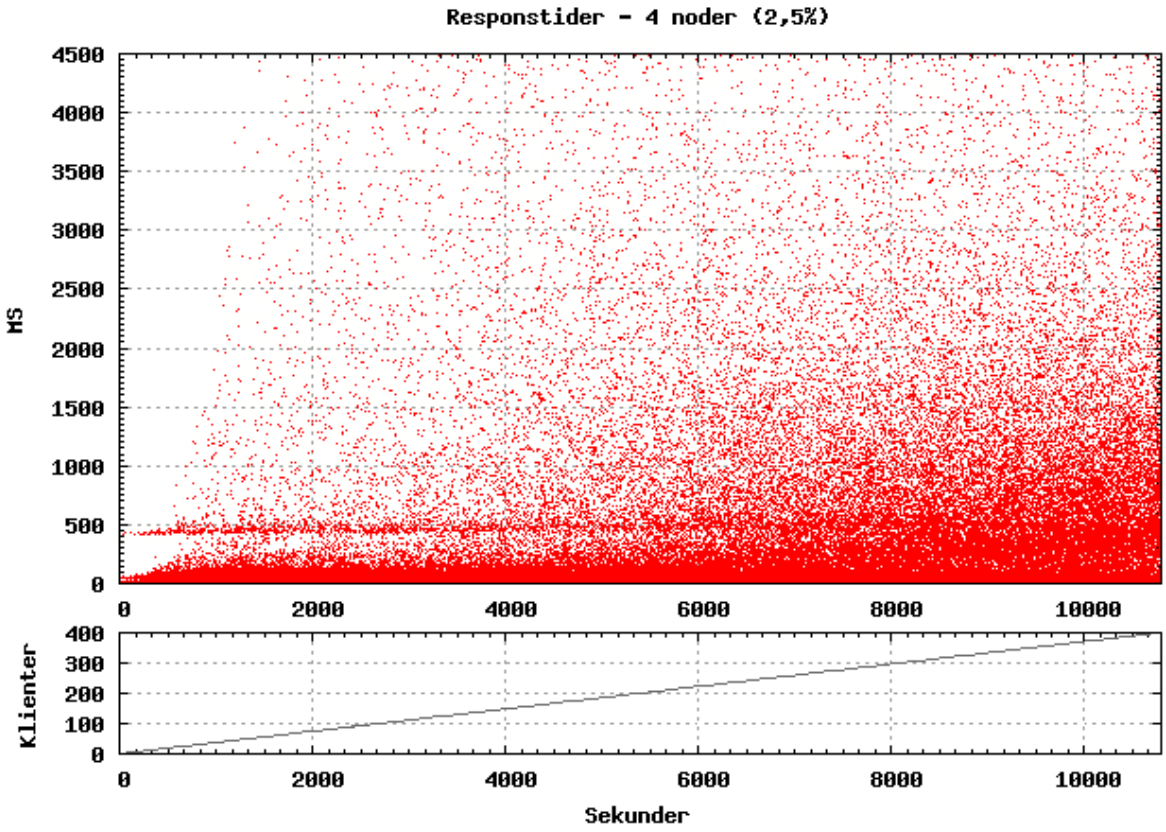
Figur 8.5 Testresultater - responstider lesing - 1 node



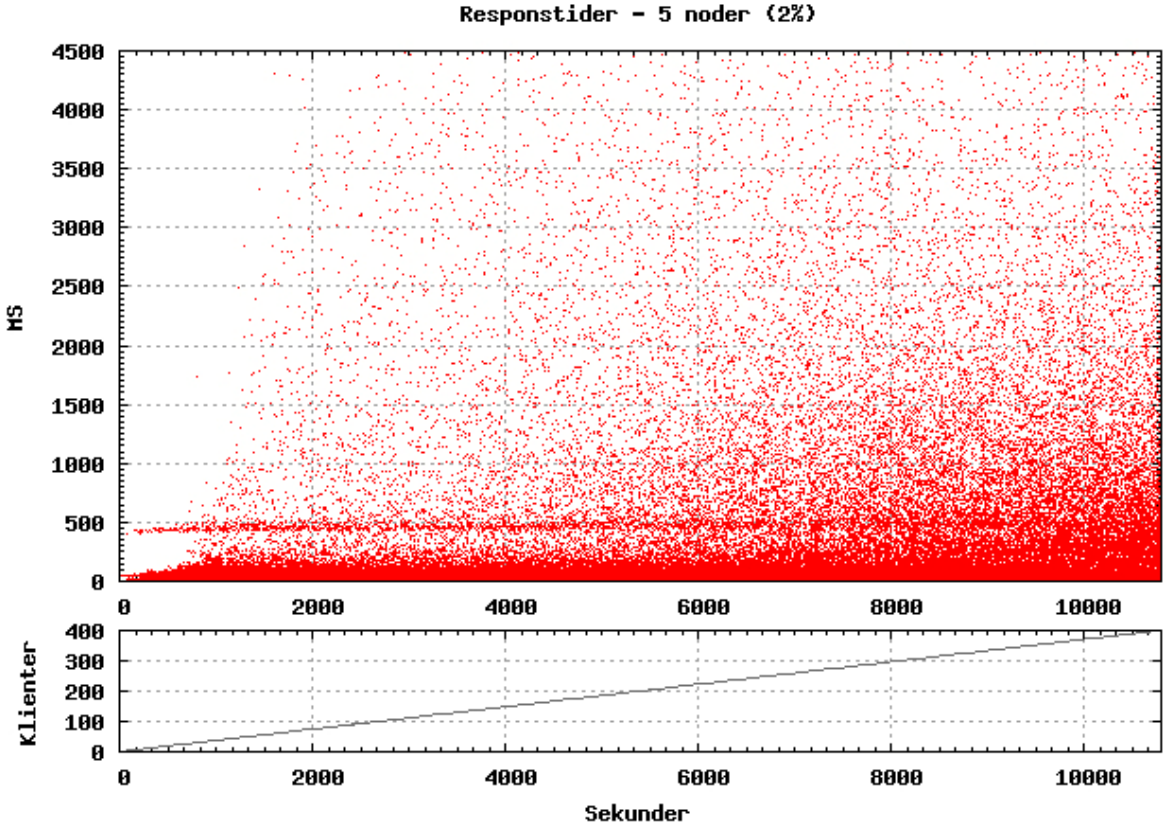
Figur 8.6 Testresultater - responstider lesing - 2 noder



Figur 8.7 Testresultater - responstider lesing - 3 noder



Figur 8.8 Testresultater - responstider lesing - 4 noder

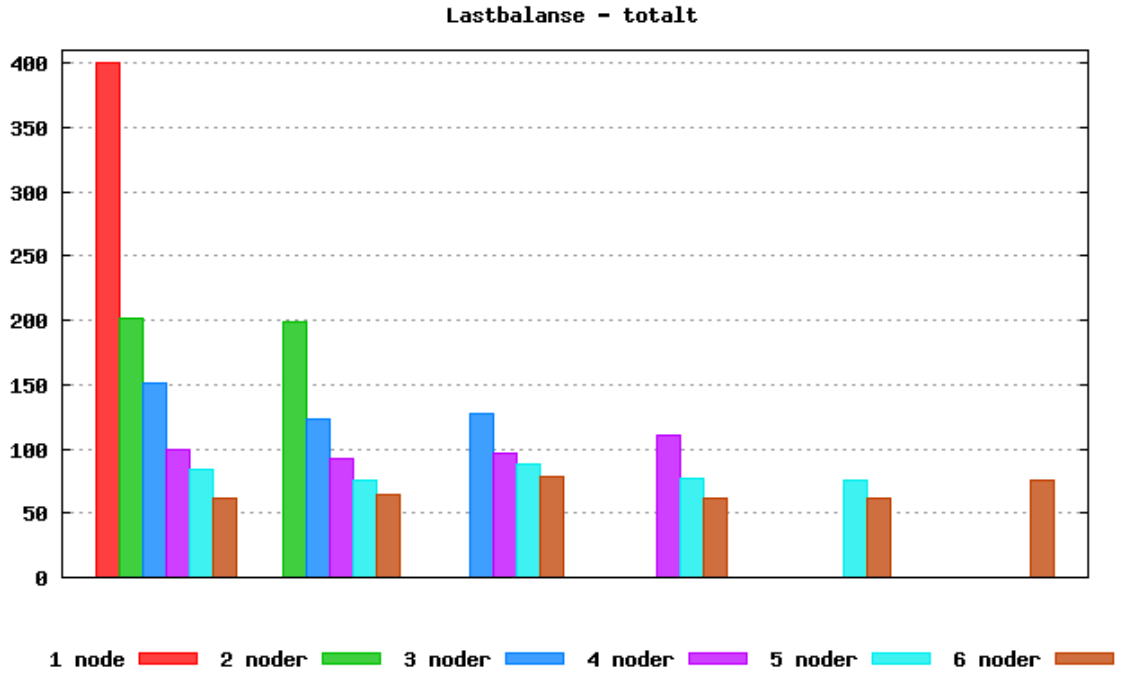


Figur 8.9 Testresultater - responstider lesing - 5 noder

Figur 8.10 Testresultater - responstider lesing - 6 noder

8.4.5 Resultater – lastbalansering

I løpet av en test gjøres det 400 nye oppkoblinger, en pr klienttråd. Den totale fordelingen av disse, for alle testrunder, vises på figur 8.11.



Figur 8.11 Testresultater – lastbalanse lesing

8.5 Innvirkning av oppdateringer

I et system som kun kan oppdateres fra et sentralt sted vil det ofte være aktuelt å kjøre inn oppdateringer som batcher. I hvor stor grad dette kommer til å påvirke evne til å behandle leseforespørsler er derfor ønskelig å teste.

8.5.1 Belastning

Leselasten kommer til å være den samme som for forrige test. Hver transaksjon vil be en servernode om å regne ut gjennomsnittet av primærnøklene i et tilfeldig valgt intervall. Størrelsen på intervallene vil være fast, og spenne seg over 1000 rader.

```
SELECT AVG(id) FROM testdata WHERE id BETWEEN X AND Y;
```

Hvor X er et tilfeldig tall mellom 1 og 523288 (1000 mindre enn antall rader), og Y er lik X+1000.

For å simulere oppdateringer vil det gjøres oppdateringer mot tilfeldige rader i testtabellen.

```
UPDATE testdata SET data = 'Y' WHERE id = X;
```

Hvor X er et tilfeldig tall mellom 1 og 524288, og Y en tilfeldig valg tekststreng.

8.5.2 Konfigurasjon

Først vil klienter som kun leser startes, og få kjøre alene i 7,5 minutter. Deretter startes det 1 klient som kjører oppdateringer mot servernodene i 15 minutter. Etter at denne har kjørt ferdig vil leseklientene få kjøre i 7,5 minutter til. Total en halv time.

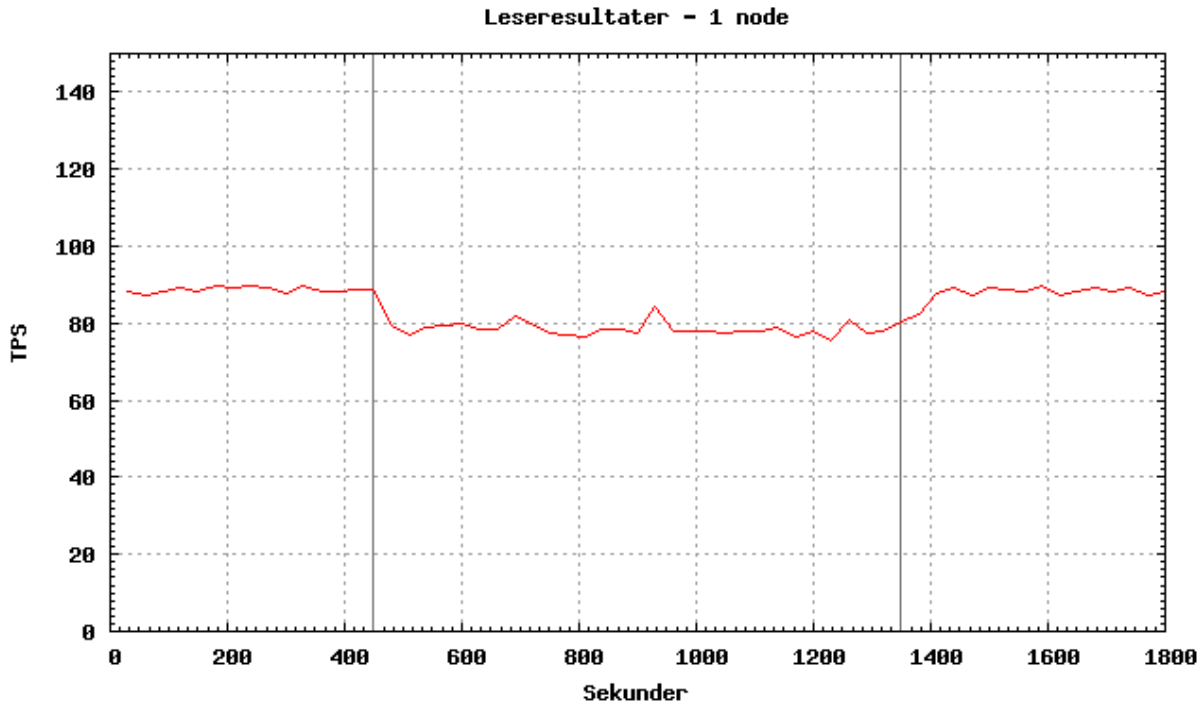
Løpet beskrevet over vil først kjøres mot 1 servernode. Deretter legges det til en ny node og prosessen gjentas, helt til det står totalt 6 servere og jobber sammen.

Antallet klienter som leser vil være 50 pr servernode, for eksempel 300 når det kjøres mot 6 servere.

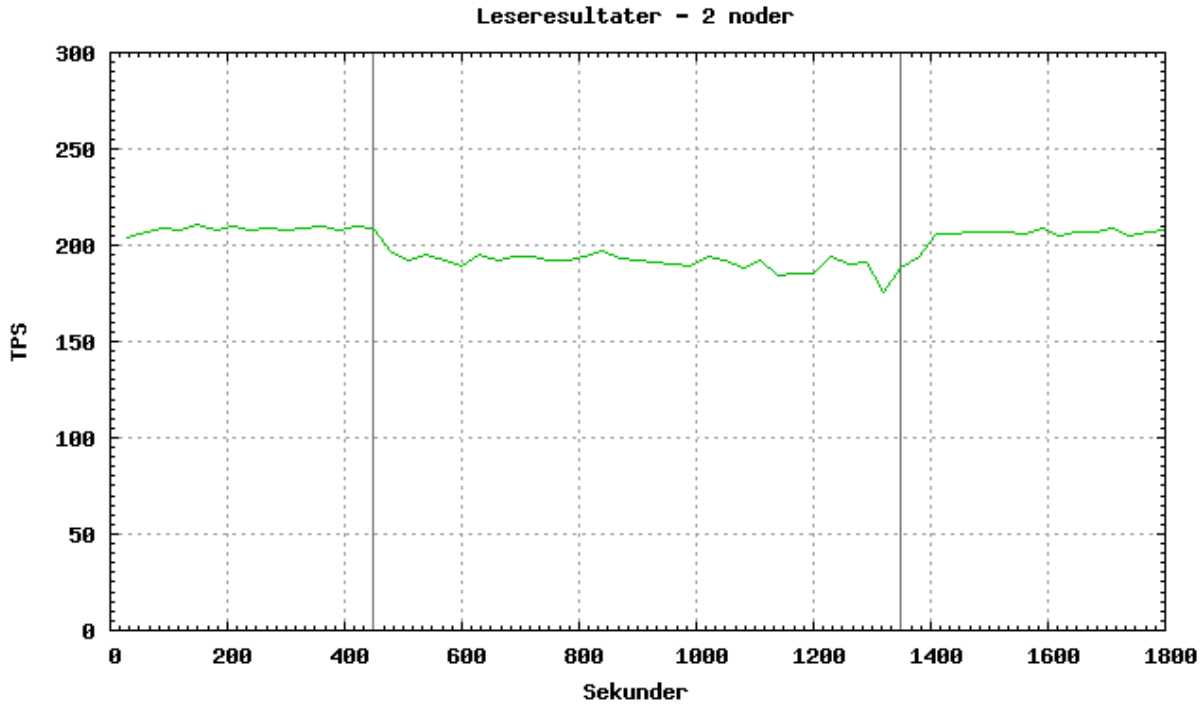
8.5.3 Resultater

Leseresultatene er plottet på separate grafer med forskjellig verdispenn på y-aksen for å fremheve det prosentvise fallet i TPS under oppdateringsperioden.

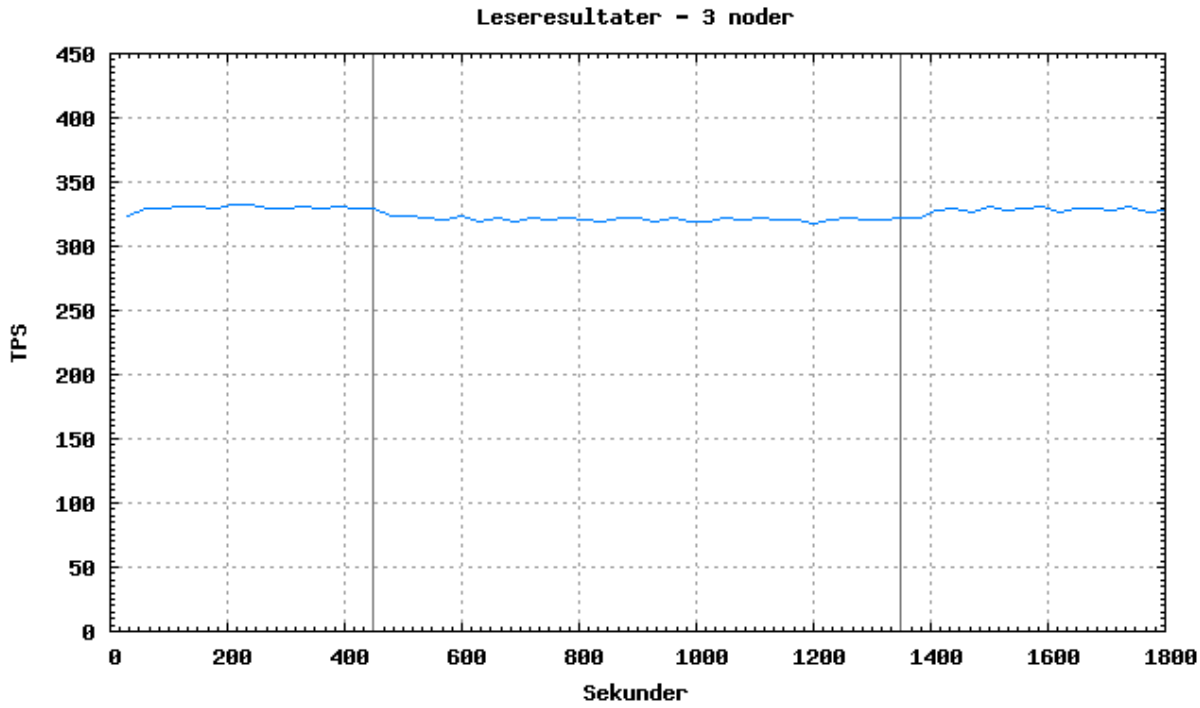
Der grå vertikale linjene tidspunktene for skrivestart og -slutt.



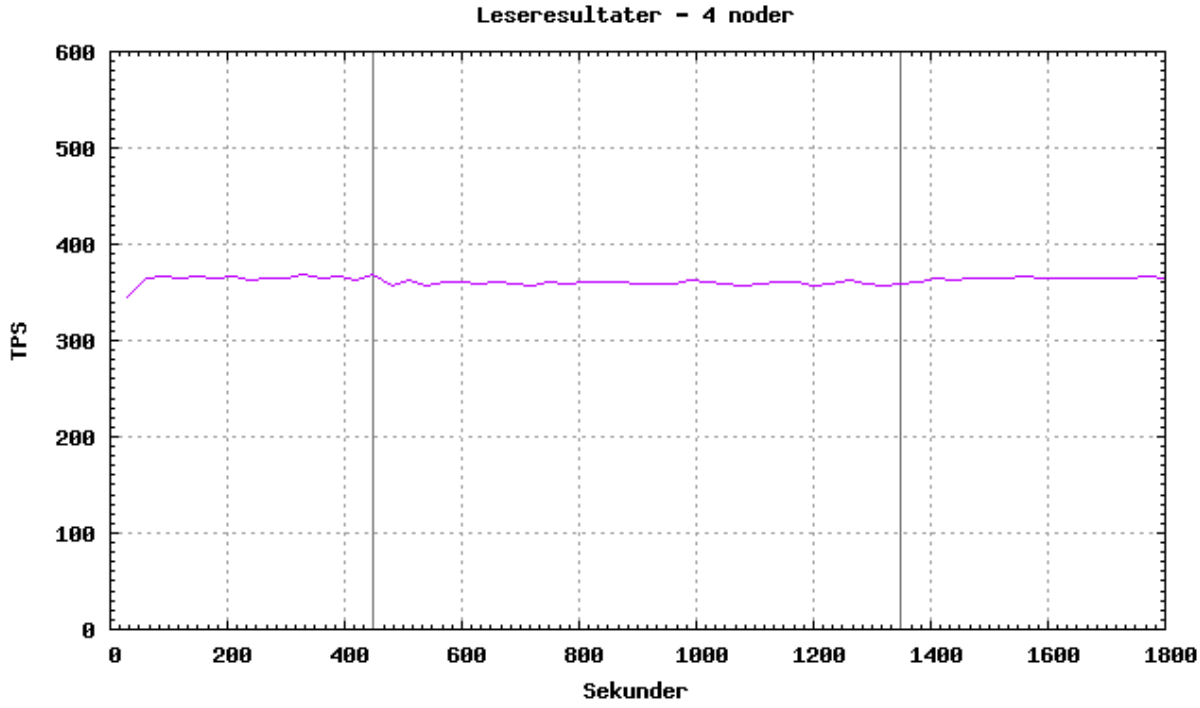
Figur 8.12 Testresultater - innvirkning av oppdateringer - 1 node



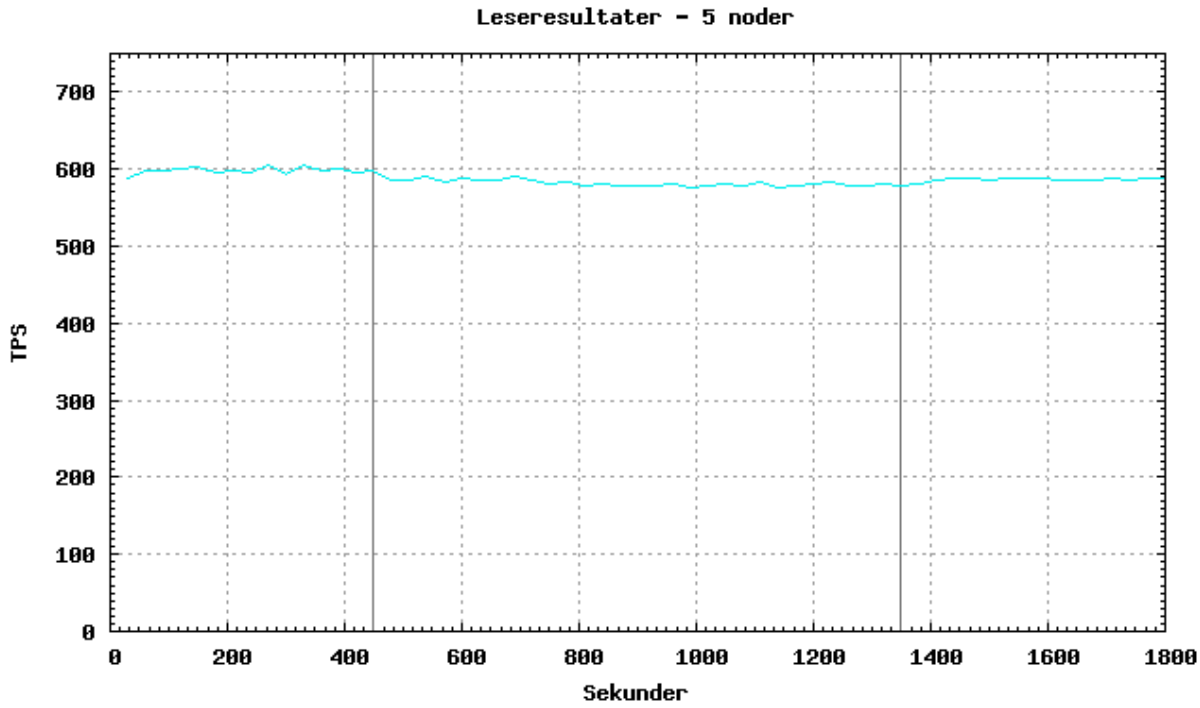
Figur 8.13 Testresultater - innvirkning av oppdateringer - 2 noder



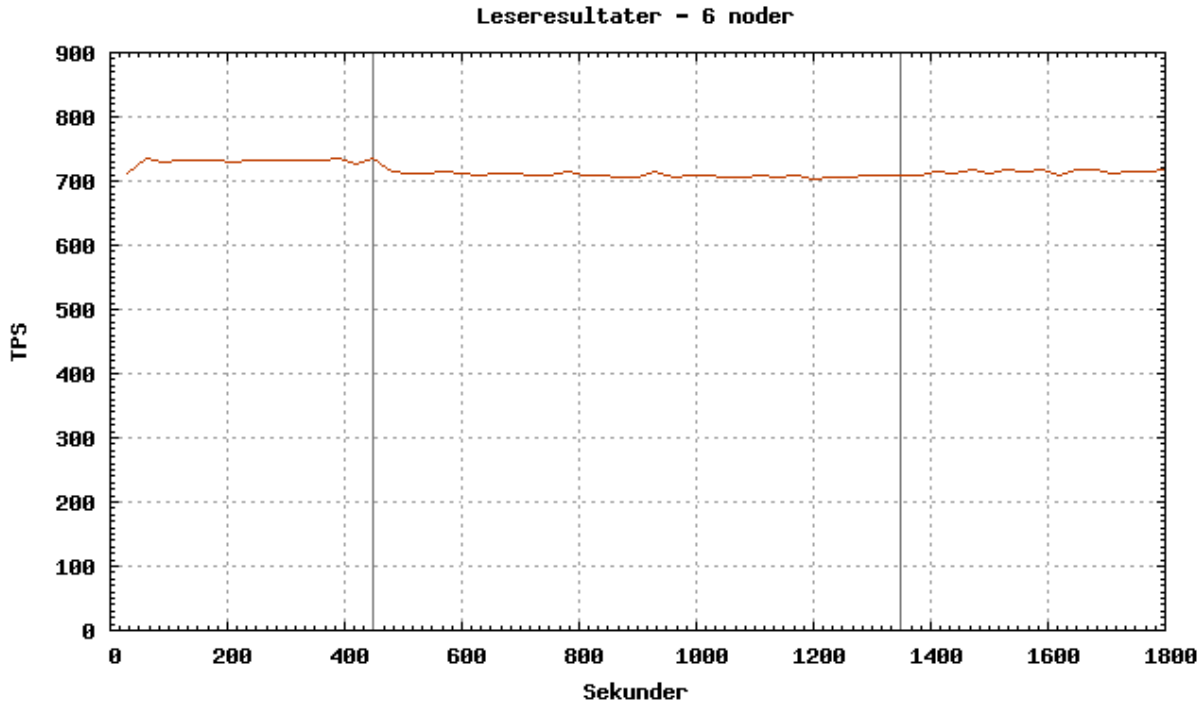
Figur 8.14 Testresultater - innvirkning av oppdateringer - 3 noder



Figur 8.15 Testresultater - innvirkning av oppdateringer - 4 noder



Figur 8.16 Testresultater - innvirkning av oppdateringer - 5 noder



Figur 8.17 Testresultater - innvirkning av oppdateringer - 6 noder

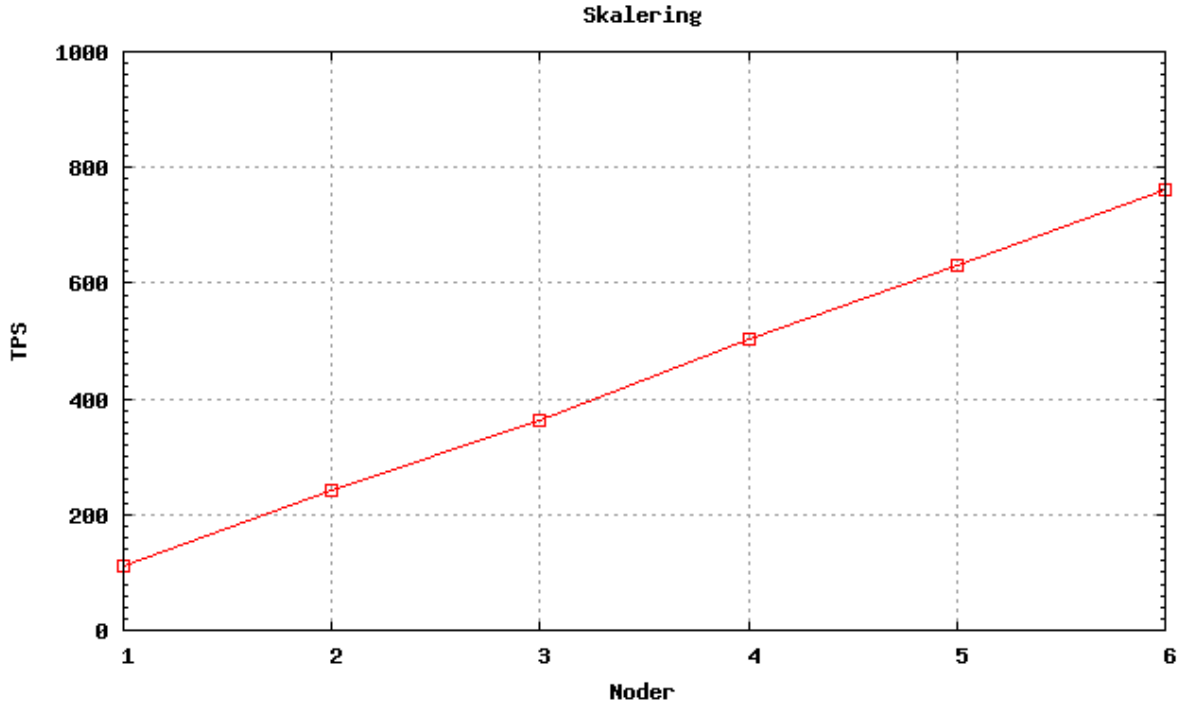
9 Analyse

9.1 Skalerbarhet

Siden det ikke finnes noen aktuelle flaskehalser i systemet antas det at skaleringsgraden er 100%. For eksempel yter 3 noder 300% av det 1 node gjør.

En forutsetning for dette er at nettverksforbindelsen mellom servere og klienter ikke går full, og dermed blir en flaskehals. Skulle dette bli et problem er det ikke vanskelig å løse. Ingen servernoder kommuniserer med hverandre, så det er ingen krav til geografisk nærhet. Noder kan da plasseres på vidt forskjellige steder med forskjellige forbindelser til klientmassen.

Nedenfor er det plottet inn hvordan antallet TPS endres ut ifra hvor mange servernoder man har. Data er hentet fra 8.4.3. Antallet TPS regnes ut som gjennomsnittet av et intervall der hver servernode behandler 30-50 samtidige klienter.



Figur 9.1 Skalering ved lesing

Nøyaktige verdier som ligger til grunn for grafen er:

Noder	TPS	Økning
1	111	111
2	241	130
3	362	121
4	502	140

5	630	128
6	762	132

Ser her at TPS øker med 111-140 for hver nye node, noe som vil tilsvare en skaleringsgrad på ca 100%.

Tenker man på en virkelig situasjon er det mulig å oppnå en skaleringsgrad som er over 100%. På TPS kurven for testing mot 1 node i 8.4.3 ser man at TPS synker etter hvert som antallet klienter økes. Setter man inn en ekstra node vil antallet klienter pr server halveres, og nodene vil hver klare å ta unna mer TPS. Hvorvidt dette kan regnes som et seriøst poeng eller et markedsføringstriks kan dog diskuteres..

9.2 Svartider

Responstidene er mye som man forventer, men det er fortsatt mulig å gjøre bemerkninger.

9.2.1 Stigning

Den første er at stigningen i responstider utover i testen blir mindre etter hvert som det kjøres mot flere og flere noder. Dette er helt som forventet. Det er jo det samme antallet samtidige klienter som kjøres i alle testene, og dette antallet klienter greier å ta unna dobbelt så mange TPS ved 2 noder som ved 1. Selve fallet tilsier også dette da tidene på et gitt punkt på x-aksen er ca halverte ved 2 noder i forhold til 1 node, osv.

En annen er at stigningen i tider i forhold til klienter på x-aksen er svakt eksponensiell. TPS en node klarer å ta unna synker jo etter hvert som mer og mer av prosesseringskraften må brukes til å opprettholde et stigende antall forbindelser. Fenomenet er best synlig på kurven for 1 node i figur 8.4.

9.2.2 Varians

På figur 8.4 kan man se at variasjonen i responstider stabiliserer seg mer og mer etter hvert som hver node håndterer flere samtidige klienter. Kurvene for standardavvik flater mer og mer ut.

Etter hver som flere noder legges til synker også variansen i responstider. Dette på grunn av at antallet klienter synker når man legger til flere noder, for eksempel vil bare første 1/6 av kurven for 1 node kunne direkte sammenlignes med hele kurven for 6 noder.

9.2.3 Horisontal ansamling ved 450ms

På figur 8.5 t.o.m. figur 8.10 kan man finne en ansamling ved ca 450ms som ligger som en horisontal strek på tvers av grafene. Forutsatt at hovedansamlingen av tider ligger under 450ms.

Dette fenomenet var ikke forventet å se før testresultatene ble plottet, og hva som er årsaken er ikke enkelt å si.

Den første tanken er at måten testing foregår har skyld, eller rettere sagt hvordan prepared statements gjenbrukes. Når en klienttråd startes opp vil den i løpet av første transaksjon kompilere et prepared statement, koble til serveren og be den utføre dette. Ved transaksjon nr 2 vil den samme forbindelsen og det samme preparedstatementet bli gjenbrukt. Dette mønsteret vil kunne produsere et lignende fenomen da kompilering og oppkobling tar ekstra tid. Problemet er at antallet ganger en slik ny oppkobling gjøres er det samme som antallet samtidige klienter. Så i løpet av disse testene vil man da kunne se maks 400 nye oppkoblinger. Tar man med at dette tallet må deles på samme måte som det totale antallet tider som plottes, vil dette bli altfor lavt til å kunne produsere en synlig horisontal linje. Ved testing med 6 noder vil for eksempel antallet plottede tider som genereres av førstegangs oppkoblinger være:

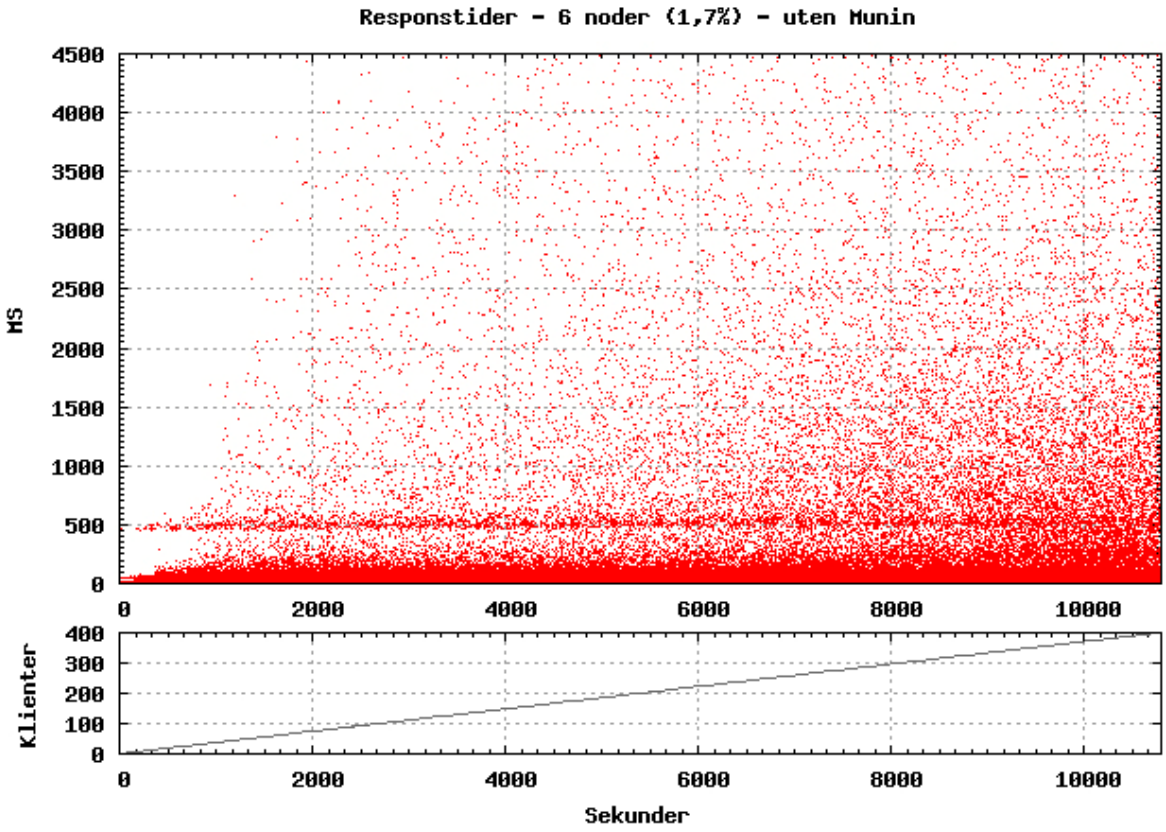
$$\frac{400}{60} \approx 7$$

En annen mulig årsak kan være den periodiske innhentingen av statusinformasjon fra Muninprosessene på serverne. Denne skjer hvert 30 sekund. Dette vil i løpet av 3 timer utgjøre

$$\frac{10800}{30} = 360$$

perioder med forstyrrelser, og disse kan komme frem som en kontinuerlig linje på en graf bestående av ca 550 piksler i bredden. Det finnes sikkert en mengde grafiske løsninger for å sjekke hvorvidt dette er faktum, men den beste måten å finne ut om Munin forårsaker forstyrrelser er å kjøre en identisk test, og plot, uten bruk av Munin. Antallet noder vil holdes på 6, da dette gir best synlighet av den horisontale linjen.

Resultatet er som følger:



Figur 9.2 Testresultater - lesing uten ressursovervåkning - 6 noder

En rask kikk på grafen, og man kan konkludere med at eventuelle forstyrrelser fra Munin ikke er årsak, da linjen er der fortsatt.

Noe annet som kanskje kan være grunnen er at forbindelser som klienter allerede har oppe faller ned, og denne forbindelsen da må opprettes på nytt. Dog skulle man tro at dette vil generere noen unntak, men under testing har ingen slike blitt kastet. I tillegg skulle man tro at tettheten av den horisontale linjen økte etter hvert som flere og flere samtidige klienter kjørte, men linjen ser ut til å holde en noen lunde lik tetthet gjennom hele testen. Det ser dermed ut som om det også spøker for denne teorien.

Hva som er den faktiske årsaken er som sagt veldig vanskelig å sette en konkret og sikker finger på, så er nødt til å sette bort denne oppgaven til en eventuell senere og mer grundig undersøkelse.

9.3 Lastbalanse

Det at valget for hvilken server en ny forbindelse skal bruke er tilfeldig kan i noen tilfeller ha negative aspekter. På figur 8.11 som viser fordelingen av tilkoblinger er det tydelig at det ikke er en helt god fordeling av forbindelser, spesielt synlig når det kjøres mot 3 noder. Dette kan også være med på å forklare de små forskjellene i skaleringsgrad på figur 9.1.

I testene som ble utført var antallet nye forbindelser begrenset til antallet samtidige klienter, slik at maskinen klientene kjørte fra ikke ble en flaskehals. Et såpass lavt tall vil kunne være med på å forårsake at noen noder blir tungt belastet, mens andre kanskje har ”tid til overs”.

I en mer normal situasjon har man gjerne et langt høyere volum med nye forbindelser, og en tilfeldig algoritme vil gi en jevnere fordeling. Med tanke på bruksområdet til dette systemet er det også svært sannsynlig at det blir faktum.

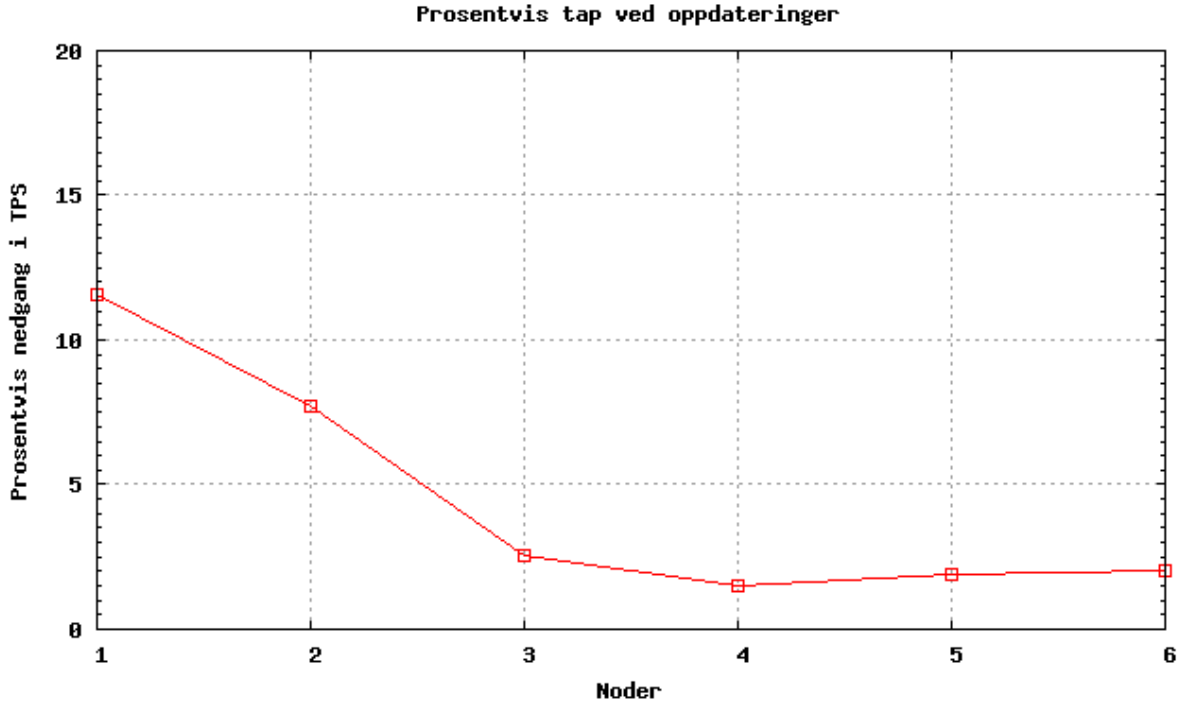
En del andre løsninger benytter gjerne en algoritme som vektes ut ifra hvor tungt belastet hver node er. Men i dette systemet finnes ikke et sentralt punkt som kan ha denne kunnskapen. En mulighet er at hver klient kan ha en oppkobling til alle servere som henter ut status om belastning. En annen er at klientene kan ha en form for Peer-to-Peer kommunikasjon som utveksler informasjon om hvilke servere de benytter mest. Men i begge tilfellene er det sannsynlig at de tekniske vanskelighetene ved en slik løsning blir så store at de ikke er praktisk brukbare.

9.4 Innvirkning av batch-baserte oppdateringer

Når en administrator skal legge inn oppdateringer gjøres dette serielt. Et statement sendes til server A. Når denne har gitt ok tilbake, sendes det samme statementet til server B osv. Når alle servere har behandlet statementet, gis samme behandling til neste statement. Grunnen til dette er å beslaglegge minst mulig ressurser når oppdateringer skal kjøres.

På følgende graf er det brukt data fra 8.5.3. Det som plottes inn er differansen i TPS når det kun ble lest og når det både ble lest og skrevet. Verdiene som er brukt er:

- Gjennomsnittet av intervallene 50-400 sekunder og 1400-1750 sekunder representerer kun lesing
- Gjennomsnittet av intervallet 500-1300 sekunder representerer både lesing og skriving



Figur 9.3 Prosentvis tap ved oppdateringer

Når det kjøres mot 1 og 2 noder kan man tydelig se at oppdateringene har en innvirkning på lesingen som foregår, men ved 3 og 4 er denne innvirkningen forsvinnende liten.

Selv om oppdateringsfunksjonaliteten ikke har stått i høysetet under utvikling, er det tydelig av testene at måten oppdateringer gjøres på kan ha hatt godt av noen justeringer.

For eksempel kunne det vært aktuelt å ta små pauser mellom hvert statement når det er få servere. I det andre tilfellet med mange servernoder kunne det vært aktuelt å parallellisere litt fremfor å kjøre alt serielt mellom nodene, ha et antall tråder som har ansvar for å kjøre oppdateringene ut til et lite antall servere hver. Dog er det vanskelig å kunne bedømme hvor intensivt disse tiltakene skal benyttes for å oppnå den ønskede effekten. En mulighet hadde vært å stole på erfaring. I et system som benytter batch-baserte oppdateringer er jo sannsynligheten til stede for at batchene er svært like, med tanke på hvor mye prosesseringskraft som skal til. Man kunne da hatt en innkjøringsperiode med justering av parametere før man kom fram til en løsning som fungerte. En annen mulighet kunne vært å ha en mekanisme som automatisk tilpasset parallelliseringsparametere underveis i batchen, men dette må regnes som avansert.

10 Konklusjon

Gjennom testing har det vist seg at å legge all logikk i klienten fungerer svært bra ved rene leseoperasjoner. Skaleringsgraden er på 100%, og vil fortsette å være det etter hvert som flere og flere servernoder legges til. Det finnes jo ingen mulige flaskehalser på serversiden, med unntak av nettverket. Men skulle det bli et problem kan man separere servernodene på forskjellige forbindelser.

Skriving derimot har en del problemer knyttet til seg. Det at XA er et redskap for 2-fase commit, og ikke replisering, gjør at oppdateringer kun kan komme sekvensert fra ett sted. Alle servernoder må også være tilstede under skriving. Er en eller flere nede må man avbryte. Ytelsen på skriveoperasjoner er også dårlig da servere oppdateres serielt.

Tenker man på de aktuelle bruksområdene for systemet, store og allment tilgjengelige systemer som brukes for å spre informasjon, er det av mindre viktighet med skrivefunksjonalitet. Det kan faktisk være positivt at oppdateringer har lav ytelse. Negative innvirkninger på leseytelse holdes borte. Det at oppdateringer kun kan komme fra ett sted er akseptabelt, men fortsatt en ulempe.

Implementasjonsmessig er løsningen grei å gjennomføre. Lesedelen er enkel å få til. Det kan legges en del arbeid i skriveløsningen for å håndtere feil mest mulig automatisk, men hvor aktuelt dette er kan variere fra situasjon til situasjon.

Ønsker man å sette inn en ny servernode må informasjonen om dette spres til alle klienter. Disse må så oppdatere serverlisten sin i henhold, enten i JDBC url eller i DataSource objektet sitt. Tenker man på bruksområdene er det sannsynlig at denne situasjonen kommer til å oppstå flere ganger. Prosessen med å oppdatere klienter er tungvint, og dette er helt klart et negativt aspekt ved systemet.

På serversiden er det ingen sammenkoblinger mellom servernodene, så det er som sagt godt mulig å spre disse geografisk. Gjør man dette kan systemet også få svært høy tilgjengelighet, dog kun relatert til leseoperasjoner.

11 Videre arbeid

I denne oppgaven har det kun vært mulig for en sentral administrator å oppdatere data. Hovedsakelig grunnet manglende støtte for sekvensering og automatisk opprydning av uferdige transaksjoner i XA. Videre kan det være ønskelig å se på om det finnes et alternativ til XA som tillater at alle klienter kan kjøre oppdateringer. Skulle dette være tilfellet kan det også være aktuelt å prøve å strømlinjeforme skiftet mellom skrive- og lesetransaksjoner, slik at klienter slipper å deklarere på forhånd om de skal skrive eller kun lese.

I systemet omtalt i denne oppgaven er det også nødvendig å få alle klienter til å manuelt oppdatere seg når man utvider clusteret med en ny servernode. Dette er tungvint, og det er aktuelt å se på mekanismer for å automatisere denne prosessen.

12 Referanser

[1] Martin Arlitt, Tai Jin, HP laboratories Palo Alto
Workload Characterization of the 1998 World Cup Web Site
http://www.hpl.hp.com/techreports/1999/HPL-1999-35R1.pdf

[2] Snorre Visnes, NTNU, IDI
Skalering med Sequoia og Derby
http://entotre.org/arkiv/fordypningsprosjekt-tdt4740/SLUTTRAPPORT.pdf

[3] Stephane Giron
Sequoia Tutorial - a quick start, versjon 1.0, 2006
http://sequoia.continuent.org/doc/latest/sequoia-tutorial.pdf

[4] Emmanuel Cecchet, Julie Marguerite, Mathieu Peltier, Nicolas Modrzyk, Dylan Hansen, Nuno Carvalho
Sequoia User's Guide, versjon 2.9, 2006
http://sequoia.continuent.org/doc/2.10/userGuide/sequoia-user-guide.pdf

[5] Continuent.org
Sequoia Administrator's Guide, issue 2.0
http://sequoia.continuent.org/doc/2.10/sequoia-admin-guide.pdf

[6] Continuent.org
Sequoia 2.8 Installation Guide, issue 1.0
http://sequoia.continuent.org/doc/2.10/sequoia-install-guide.pdf

[7] MySQL hjemmeside
http://mysql.org/downloads/mysql/5.0.html
http://dev.mysql.com/doc/refman/5.0/en/index.html

[8] Apache Derby hjemmeside
http://db.apache.org/derby/
http://db.apache.org/derby/manuals/index.html

[9] IBM DB2 hjemmeside
http://www-306.ibm.com/software/data/db2/9/

[10] Apache License v2.0
http://www.apache.org/licenses/LICENSE-2.0.txt

[11] JGroups hjemmeside
http://www.jgroups.org/javagroupsnew/docs/index.html

[12] Appia hjemmeside
http://appia.di.fc.ul.pt/wiki/index.php?title=Main_Page

[13] SQL Relay hjemmeside
http://sqlrelay.sourceforge.net/
http://sqlrelay.sourceforge.net/documentation.html

[14] GPL license
http://www.gnu.org/licenses/gpl.txt

[15] MySQL dokumentasjon av replisering
http://dev.mysql.com/doc/refman/5.0/en/replication.html

[16] MySQL dokumentasjon av ReplicationDriver
http://dev.mysql.com/doc/refman/5.0/en/connector-j-reference-replication-connection.html

[17] MySQL dokumentasjon av Connector/J
http://dev.mysql.com/doc/refman/5.0/en/connector-j.html

[18] Slony-I hjemmeside
http://main.slony.info/
http://main.slony.info/documentation/

[19] PostgreSQL hjemmeside
http://www.postgresql.org/
http://www.postgresql.org/docs/8.2/static/index.html

[20] Times Ten hjemmeside
http://www.oracle.com/database/timesten.html
http://www.oracle.com/technology/documentation/timesten_doc.html

[21] Oracle Corporation
Oracle TimesTen In-Memory Database Introduction - Release 7.0, Release 7.0
http://download.oracle.com/otn_hosted_doc/timesten/702/TimesTen-Documentation/intro.pdf

[22] Oracle Corporation
TimesTen to TimesTen Replication Guide - Release 7.0, B31684-02
http://download.oracle.com/otn_hosted_doc/timesten/702/TimesTen-Documentation/replication.pdf

[23] Oracle Corporation
TimesTen Cache Connect to Oracle Guide - Release 7.0, B31685-02
http://download.oracle.com/otn_hosted_doc/timesten/702/TimesTen-Documentation/cacheconnect.pdf

[24] Oracle Corporation
Oracle Real Application Clusters 10g, An Oracle Technical White Paper, mai 2005
http://www.oracle.com/technology/products/database/clustering/pdf/twp_rac10gr2.pdf

[25] Oracle Corporation
Oracle Real Application Clusters, Oracle Data Sheet, mai 2005
http://www.oracle.com/technology/products/database/clustering/pdf/ds_rac.pdf

[26] Oracle Database 10g hjemmeside
http://www.oracle.com/database/index.html

[27] Oracle Net
http://download-uk.oracle.com/docs/cd/B19306_01/rac.102/b28759/configwlm.htm#CHDGCCDI

[28] Bernt Johnsen, presentasjon på JavaZone 2005
Apache Derby
http://www4.java.no/javazone/2005/presentasjoner/BerntJohnsen/Bernt_Johnsen-DerbyJavaZone.pdf

[29] Dan Debrunner, presentasjon på 2006 JavaOne Conference
Introduction to Apache Derby
http://developers.sun.com/learning/javaoneonline/2006/coreenterprise/TS-3154.html

[30] X/Open Company Ltd.
Distributed Transaction Processing: The XA Specification, ISBN: 1 872630 24 3
http://www.opengroup.org/bookstore/catalog/c193.htm

[31] Susan Cheung, Vlada Matena, Sun Microsystems Inc., 1. November 2002
Java Transaction API (JTA)
http://javashoplm.sun.com/ECom/docs/Welcome.jsp?StoreId=22&PartDetailId=jta-1.1-spec-oth-JSpec&SiteId=JSC&TransactionId=noreg

[32] Apache Software Foundation
Apache Derby Issue: DERBY-2432
https://issues.apache.org/jira/browse/DERBY-2432

[33] Testklient pakke
http://entotre.org/arkiv/diplom/testklient/testklient.tar.gz

[34] Munin hjemmeside
http://munin.projects.linpro.no/

13 Vedlegg

13.1 Testtransaksjoner

13.1.1 Lesetransaksjon

Dette er transaksjonen som ble benyttet for å simulere lesing i 8.4 og 8.5. Koden er som brukt i Testklient programmet som Test-komponent. Metoden `performTransaction()` er den som utfører selve transaksjonen.

BALClientDriverPerformanceReadAvg1000.java

```
1  package loadclient.tests;
2  import java.math.BigDecimal;
3  import java.sql.*;
4  import loadclient.*;
5  public class BALClientDriverPerformanceReadAvg1000
6      implements loadclient.Test
7  {
8      private int intervalSize = 1000;
9      private int numberOfRows = 524288;
10     private Connection conn = null;
11     private PreparedStatement pstmt = null;
12     private double multiplier = 0L;
13     private double divisor = 0L;
14     private boolean close = false;
15
16     static {
17         TestConfig conf = new TestConfig();
18         try {
19             Class.forName(conf.getJdbcDriver()).newInstance();
20         } catch (Exception e) {
21             e.printStackTrace();
22         }
23     }
24
25     public synchronized void close() {
26         this.close = true;
27     }
28
29     public synchronized boolean performTransaction(TestConfig conf) {
```

```
30          boolean RETURN = false;
31          try {
32              if (conn == null) {
33                  conn = DriverManager.getConnection(
34                          conf.getJdbcURL(),
35                          conf.getJdbcUser(),
36                          conf.getJdbcPassword());
37                  pstmt = conn.prepareStatement(
38                          "SELECT " +
39                              "AVG(id) " +
40                          "FROM " +
41                              "testdata " +
42                          "WHERE " +
43                              "id " +
44                          "BETWEEN ? AND ?");
45
46                  // Note the connection in BalanceMonitor
47                  String url = conn.getMetaData().getURL();
48                  BalanceLogger.writeHit(url);
49              }
50              BigDecimal tmp =
51                  new BigDecimal(numberOfRows - intervalSize);
52              multiplier = 1;
53              while (tmp.doubleValue() > 1) {
54                  tmp = tmp.divide(new BigDecimal(10));
55                  multiplier = multiplier * 10;
56              }
57              divisor = tmp.doubleValue();
58              int start = (int)
59                  ((((int)(Math.random() *multiplier)) +1) *divisor);
60              int stop = start + intervalSize;
61              pstmt.setInt(1, start);
62              pstmt.setInt(2, stop);
63              long tsStmt = System.currentTimeMillis();
64              ResultSet rs = pstmt.executeQuery();
65              if (rs.next())
66                  RETURN = true;
67              rs.close();
68              ResponseLogger.writeStatementResult(
69                  System.currentTimeMillis() - tsStmt);
70              if (close) {
71                  try {
```

```
72                      pstmt.close();
73                  } catch (Exception e) {
74                      e.printStackTrace();
75                  }
76                  try {
77                      conn.close();
78                  } catch (Exception e) {
79                      e.printStackTrace();
80                  }
81              }
82          } catch (Exception e) {
83              e.printStackTrace();
84              RETURN = false;
85          }
86          return RETURN;
87      }
88  }
```

13.1.2 Skrivetransaksjon

Dette er transaksjonen som ble benyttet for å simulere skriving i 8.5. Koden er som brukt i Testklient programmet som Test-komponent. Metoden `performTransaction()` er den som utfører selve transaksjonen.

BALClientDriverPerformanceReadAvg1000.java

```
1   package loadclient.tests;
2   import java.math.BigDecimal;
3   import java.sql.*;
4   import loadclient.TestConfig;
5   public class BALClientDriverWriteInterference
6         implements loadclient.Test
7   {
8       private int numberOfRows = 524288;
9
10      static {
11          TestConfig conf = new TestConfig();
12          try {
13              Class.forName(conf.getJdbcDriver()).newInstance();
14          } catch (Exception e) {
15              e.printStackTrace();
16          }
17      }
```

```
18
19      public void close() {
20          ;
21      }
22
23      public boolean performTransaction(TestConfig conf) {
24          boolean RESULT = false;
25          String jdbcUrl = conf.getJdbcURL();
26          Connection conn = null;
27          try {
28              Class.forName(conf.getJdbcDriver()).newInstance();
29              conn = DriverManager.getConnection(
30                          jdbcUrl,
31                          conf.getJdbcUser(),
32                          conf.getJdbcPassword());
33              Statement stmt = conn.createStatement();
34              String sql =
35                  "UPDATE testdata SET data = "
36                  + "'XA-" + getRandomInt(9999999) + "'"
37                  + " WHERE id = "
38                  + (getRandomInt(numberOfRows) + 1);
39              int numberUpdated = stmt.executeUpdate(sql);
40              if (numberUpdated > 0)
41                  RESULT = true;
42              stmt.close();
43              conn.commit();
44          } catch (Exception e ) {
45              e.printStackTrace();
46          } finally {
47              try {
48                  conn.close();
49              } catch (Exception e) {
50                  e.printStackTrace();
51              }
52          }
53          return RESULT;
54      }
55
56      private int getRandomInt(int max) {
57          double multiplier = 0L;
58          double divisor = 0L;
59          BigDecimal tmp = new BigDecimal(max);
```

```
60          multiplier = 1;
61          while (tmp.doubleValue() > 1) {
62              tmp = tmp.divide(new BigDecimal(10));
63              multiplier = multiplier * 10;
64          }
65          divisor = tmp.doubleValue();
66          int result = (int)
67              ((((int)(Math.random() *multiplier)) +1) *divisor);
68          return result;
69      }
70  }
```