Forord

Denne masteroppgaven er levert til Institutt for Datateknikk og Informasjonsvitenskap, IDI, ved Norges Teknisk-Naturvitenskapelige Universitet, NTNU, våren 2005. Oppgaven tar utgangspunkt i Hien Nam Les doktorgradsavhandling, Mobile Transaction System for Supporting Mobile Work, som forøvrig er en del av MOWAHS prosjektet.

Faglig ansvarlig og hovedveileder for oppgaven har vært professor Mads Nygård. Medveileder har vært doktoringeniør stipendiat Hien Nam Le.

Vi ønsker å takke våre veiledere for en interessant oppgave og god veiledning gjennom prosjektet.

Gunnar Gauslaa Bergem

Rune Høivik

Innhold

Tabeller

Figurer

Del I

Introduksjon

Innledning til dokumentet

Introduksjonen gir oversikt over problemstillingen til diplomoppgaven og kaster lys over de målene vi ønsker å oppnå. I tillegg forklarer vi hvilken kontekst hovedoppgaven vår befinner seg i.

- Kapittel 1, Oppgavetekst, inneholder oppgaveteksten vi fikk fra veileder samt vår egen tolkning av denne teksten.
- Kapittel 2, MOWAHS, omhandler forskningsprosjektet som denne oppgaven er en del av.
- Kapittel 3, Introduksjon til eksport-import modellen, gir en kort introduksjon til transaksjonsmodellen som denne oppgaven videreutvikler og implementerer en del av.

Kapittel 1

Oppgavetekst

I dette kapittelet vil vi gjengi oppgaveteksten fra veileder, i tillegg til å gi vår egen tolkning av oppgaven.

1.1 Oppgavetekst fra veileder

Den følgende oppgaveteksten ble utarbeidet av den ene av veilederne våre ved Institutt for Datateknikk, Hien Nam Le:

Project Title: Customizing Isolation Properties for Mobile Transactions

The mobile environment confronts many challenging problems due to its unique characteristics, for example, the frequent and unpredictable disconnection of wireless networks. Challenges such as inconsistency of shared data, asynchronous communication or barricading of on-going activities are common issues in the mobile environment. Furthermore, cooperative activities in mobile environments may suffer due to the interruptions of a communication channel. The duration of disconnected or connected periods is not always as planned, i.e., varying in time, due to many factors, for example, different travel routes of mobile users, unavailable wireless areas or physical obstacles such as buildings. This might result in longer waiting time or interruption periods for other on-going activities.

Traditionally, transactions with ACID (Atomicity, Consistency, Isolation, and Durability) properties have been used to enforce the consistency of shared data and to support the correctness of database operations in distributed environment. However, in mobile environment, traditional transactions may not be able to cope with the above challenging issues efficiently. For example, the isolation property ensures that a transaction does not share its partial results to other transactions while the transaction is being executed. In mobile environment, due to disconnection, a transaction may not be able to release locks or to report final results when it commits. Therefore this could cause a problem of delaying progress of other transactions.

To cope with the above challenges, there are many research proposals that focus on supporting sharing partial results during the execution process of transactions. The two main approaches

are:

1. A transaction delegates its states or status to another transaction.
2. A transaction releases its locks and reports its assumed-to-commit results to other transactions.

However, there are still limitations on these proposals. The first approach requires synchronous communication among transactions, while the later lacks of recovery support if the transaction aborts or rollbacks.

The objective of this project is to investigate and develop a system prototype that supports sharing partial results among transactions while they are being executed in mobile environments.

The project work is suited for 2 students. Experience with transaction processing is an advantage.

1.2 Tolkning av oppgavetekst

Her følger vår egen tolkning av oppgaveteksten og forståelse av målene ved diplomoppgaven:

Oppgaven skal ta sikte på å løse problemer som transaksjoner kan få ved utførelse på mobile enheter. Man må regne med at enheten mister tilkoblingen ofte, og dette gjør at man må revurdere de tradisjonelle ACID egenskapene for å unngå at gjennomstrømningen av avsluttede transaksjoner blir for lav. Dette krever at transaksjonene i større grad må dele informasjon seg i mellom (relaxed isolation). Da den overordnede effekten ved bruk av transaksjoner er å oppnå korrekte kjøringer, krever mobile transaksjoner eksplisitte mekanismer for å opprettholde korrekte utføringer.

Ulike mekanismer for å oppnå dette er blant annet

1. Delegering (krever synkron kommunikasjon)
2. Assumed-to-commit (mangel på feilhåndtering i forbindelse ved abort og tilbakerulling av transaksjoner)

Da disse mekanismene har sine åpenbare svakheter, er det blitt foreslått en transaksjonsmodell basert på import-eksport delingsrom for å utveksle informasjon mellom transaksjoner. Dette dynamiske delingsrommet opprettes av den initiativtakende transaksjonen. En viktig designavgjørelse her er å finne ut hvor det er fornuftig å opprette dette. Foreslåtte alternativer er enten på den mobile enheten eller på den mobile støttestasjonen. Andre transaksjoner kan bli medlem av eller forlate dette felles delingsrommet. Når en transaksjon har forbundet seg til det felles delingsrommet kan den starte en eksporttransaksjon for å dele data med andre transaksjoner. Disse transaksjonene kan få tilgang til dataene ved å starte en importtransaksjon.

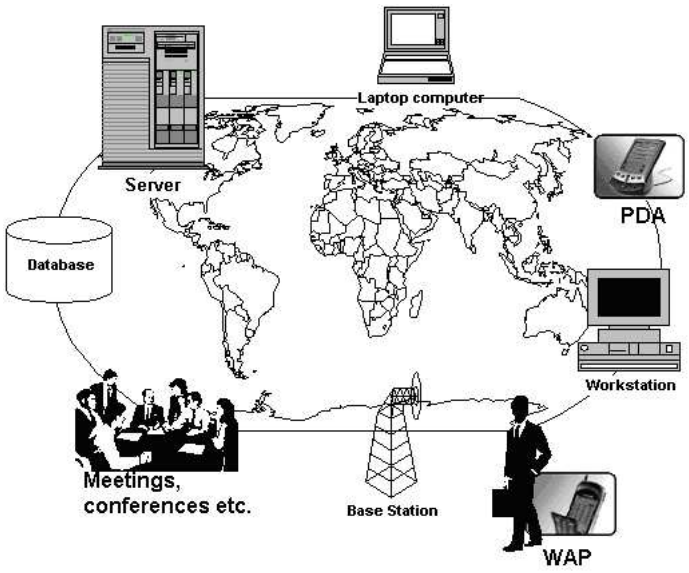
I tillegg må korrekthetskriteriene til transaksjonene gjøres rede for.

Vår oppgave er å videreutvikle denne transaksjonsmodellen, samt lage en prototyp som kan demonstrere prinsippene.

Kapittel 2

MOWAHS

MOWAHS [30], *Mobile Work Across Heterogeneous Systems* er et norsk forskningsprosjekt som ble påbegynt i januar 2001. Prosjektet, som er sponset av norsk forskningsråd, ledes av gruppene for systemutvikling og databaseteknikk ved Institutt for Datateknikk og Informasjonsvitenskap (IDI), Norges Teknisk Naturvitenskapelig Universitet, NTNU. MOWAHS har som hovedmål å studere hvordan man kan oppnå økt støtte for mobil prosessering for alt fra mindre enheter som mobiltelefoner og PDAer til større serverløsninger. Siden våre veiledere i arbeidet med diplomoppgaven er tilknyttet dette prosjektet, vil vårt resultat også bidra til at man kommer et skritt nærmere målsetningen.



Figur 2.1: Den mobile verden, fra [30]

Utgangspunktet for MOWAHS prosjektet er den eksplosive utviklingen innenfor bruk av *internett*. En stadig større andel av mobile brukere er i dag koblet opp mot internett, og har dermed tilgang til stasjonære datamaskiner. Dette skaper grunnlaget for såkalte *virtuelle organisasjoner*, hvor mobile brukere kan samarbeide med hverandre på kryss av tidssoner og landegrenser, se figur 2.1. Denne distribuerte delingen kan være av asynkron karakter hvor brukere laster ned informasjon fra tjenere ved ulike tidspunkt, eller synkron hvor brukere har direkte forbindelse

med hverandre hele tiden. Til tross for at det i dag finnes teknologi som muliggjør distribuerte dataløsninger, er infrastruktur og verktøy som støtter virtuelle organisasjoner svært umoden. Blant annet bør det være utstrakt støtte for homogenitet innenfor kommunikasjonsutstyr (mobiltelefon, PDA), støtteverktøy og arbeidsmodeller. En annen utfordring vil være å synkronisere de mobile enhetene mot stasjonære tjenere og PCer over et ellers ustabilt trådløst nettverk.

På bakgrunn av de overnevnte problemene tilknyttet virtuelle organisasjoner har MOWAHS prosjektet følgende hovedmål:

- Økt forståelse og kontinuerlig vurdering og forbedring av arbeidsprosesser i virtuelle organisasjoner.
- Utvikle et fleksibelt felles arbeidsområde for utveksling og utføring av mobile arbeidsprosesser for ulike elektronisk enheter.
- Spre resultatene til studenter, kollegaer, bedrifter og samfunnet generelt.

MOWAHS er ikke er prosjekt som starter på bar bakke, men har god støtte fra tidligere forskningsprosjekter fra IDI som CAGIS [6] og EPOS [11]. I EPOS (Expert system for Program and ("Og") Software developement) ble det hovedsakelig fokusert på prosessmodellering, håndtering av programvarekonfigurasjon og støtte for kooperativt arbeid. CAGIS (Cooperative Agents in a Global Information Space) har som hovedmål å støtte IT-basert samarbeid mellom mennesker med ulik geografisk lokasjon forankret i et globalt informasjonsmiljø bestående av homogent innhold.

Kapittel 3

Introduksjon til eksport-import modellen

Diplomoppgaven bygger på en mobil distribuert transaksjonsmodell som for tiden utvikles av våre veiledere Hien Nam Le og Mads Nygård ved IDI, NTNU. Modellen kalles for eksport-import transaksjonsmodellen, og den er beskrevet i [32]. I dette kapittelet skal det gis en kort introduksjon til modellen, for en mer omfattende beskrivelse henvises det til kapittel 8 i forstudiumet. Modellen og artiklene som omhandler den er imidlertid en del av Hiens pågående doktorgrad, og det tas forbehold om at de kan endres før publisering.

Eksport-import modellen tar sikte på å støtte deling av verdier mellom mobile transaksjoner under utførelse. Verdiene deles i et såkalt mobilt delingsrom som støtter både synkron og asynkron kommunikasjon. Ved å innføre disse begrepene håper man å kunne øke parallelliteten i forbindelse med kjøring av transaksjoner i systemet, samt å gi fleksible løsninger på vanlige utfordringer i mobile systemer som frakobling og mobilitet.

Siden eksport- og importtransaksjonene innfører et noe mer avslappet krav til isolasjon enn i standard ACID egenskaper som gjelder for tradisjonelle databasesystemer, må det tas spesielle hensyn med tanke på abortregler og konsistens. For å gjennomføre dette defineres det et ekstra transaksjonsnivå som ligger over de klassiske transaksjonene som utføres mot vanlige relasjonsdatabaser. Disse transaksjonene, som kalles for globale transaksjoner, støtter deling av verdier imellom de klassiske transaksjonene på nivået under (heretter kalt subtransaksjoner).

Del II

Forstudie

Innledning til dokumentet

Denne delen beskriver forstudiet vi har utført i forbindelse med diplomoppgaven. Transaksjonsteorien som er nødvendig for å forstå begreper i resten av oppgaven er omhandlet her. Videre beskriver vi det mobile databasemiljøet som programvaren vår kjører i. Et utvalg av mobile transaksjonsmodeller presenteres og forklares før vi introduserer import-eksport modellen som vi baserer oss på i denne oppgaven. Eksport-import modellen sammenlignes så med de andre transaksjonsmodellene vi har gjennomgått. Til slutt beskriver vi aktuelle teknologiplattformer vi har mulighet til å bruke for å utvikle systemet, og deretter vi bestemmer oss for å jobbe videre med en av dem.

- Kapittel 4, Transaksjonsteori, forklarer transaksjonsbegrepet og beskriver samtidighetskontroll, ACID egenskaper og feilhåndtering.
- Kapittel 5, Beskrivelse av mobile databasemiljø, gir et eksempel på den fysiske organiseringen av mobile og stasjonære enheter i et databasemiljø.
- Kapittel 6, Utfordringer i mobile databasemiljø, beskriver de spesielle utfordringene man må ta hensyn til i mobile databasemiljøer.
- Kapittel 7, Mobile transaksjonsmodeller, beskriver fem transaksjonsmodeller og hvilke problemer de tar sikte på å løse.
- Kapittel 8, Eksport-import transaksjonsmodellen, beskriver den mobile transaksjonsmodellen som vi bruker i vår oppgave.
- Kapittel 9, Teknologiplattformer, gjennomgår aktuelle teknologiplattformer og evaluerer dem ut ifra hvor godt vi tror de vil hjelpe oss å løse vår problemstilling.

Kapittel 4

Transaksjonsteori

4.1 Introduksjon

En *database* er en samling dataobjekter som oppfyller et sett av integritetsregler. Disse reglene kan blant annet spesifisere hvilken type informasjon dataobjektene skal inneholde. Et eksempel kan være en persondatabase hvor hver person må være registrert med et personnummer med 11 siffer av typen *heltall*. For å kunne administrere og utføre operasjoner på dataobjektene i databasen er man avhengig av en spesialisert programvare kalt DBMS - *database management system*, ofte kalt et databasesystem. I likhet med et filsystem har et databasesystem også ansvar for at data lagres på en persistent måte, uavhengig av prosessene som bruker dataene. Foruten om å tilby brukere et grensesnitt for å aksessere dataene, skal et databasesystem også gi støtte for å håndtere transaksjoner, noe som vil bli beskrevet grundigere i de neste kapitlene.

4.2 Bakgrunnsteori om transaksjoner

En transaksjon er definert som en serie av atomiske operasjoner utført på objekter i en database. Primært blir en transaksjon initiert av en *begin* kommando og terminert med enten *commit* eller *abort*. Når en transaksjon har startet kan den utføre lese- og skriveoperasjoner på objektene i databasen. Et viktig overordnet krav for en transaksjon er å bevare databasen i en konsistent tilstand. Om transaksjoner feiler underveis eller kjører til de committer, må uansett dette kravet ikke brytes. Det endelige resultatet av å kjøre en transaksjon er som nevnt enten suksess eller fiasko. De terminerende hendelsene for en transaksjon er beskrevet under:

- **Abort:** En feil oppstod under kjøringen av transaksjonen. Det endelige resultatet av kjøringen skal være som om transaksjonen aldri ble kjørt.

- **Commit:** Transaksjonen ble kjørt uten feil, endringene lagres permanent og transaksjonen avsluttes.

4.2.1 ACID egenskapene

Et overordnet mål for et DBMS er å la flere brukertransaksjoner samtidig kunne kjøre og oppdatere en underliggende database. En database skal alltid være konsistent og samtidig være i stand til å overleve både software og hardware feil. En måte å oppnå dette er å la transaksjoner implementere de såkalte ACID egenskapene, som er beskrevet under.

Atomiskhet (eng: atomicity)

En *atomisk* transaksjon skal alltid utføres fullstendig. Med dette menes at enten skal alle eller ingen av operasjonene til transaksjonen reflekteres i databasen. Et eksempel fra en banktransaksjon viser kanskje hvorfor denne egenskapen er så viktig. . Først utføres det en operasjon hvor det tas ut 100 NOK fra konto A. Deretter settes dette beløpet inn på konto B. Hva om bare den første operasjonen ble utført? Poenget med en atomisk transaksjon er at enten skal den utføres korrekt, ellers så skal effekten av den være som om den aldri fant sted. Det er opp til de implementerte feilhåndteringsmetodene (eng: recovery) å sikre atomiskhet for en transaksjon som feiler. Hvis en feil oppstår (abort) må alle tilstander som er berørt rettes opp slik at effekten av operasjonene blir fjernet.

Konsistens (eng: consistency)

En transaksjon skal alltid bevare konsistensen til databasen og dermed bare committe gyldige resultater og sørge for at den befinner seg i en gyldig tilstand. I kontrast til egenskapene *atomiskhet*, *isolasjon* og *varighet* er det opp til transaksjonsprogrammet og det transaksjonelle prosesseringssystemet å sikre at konistensegenskapen blir overholdt. Selv om en database kan befinne seg i ugyldige tilstander under selve kjøringen av en transaksjon, får dette ingen betydning da operasjonene ikke er synlige for andre transaksjoner. Dette skyldes at en transaksjon implementerer egenskapene *atomiskhet* og *isolasjon*.

Isolasjon (eng: isolation)

Isolasjon angir i hvilken grad transaksjoner har mulighet til å se hverandres resultater. Full isolasjon gir en transaksjon inntrykk av at den kjører helt alene i systemet, selv om det naturligvis er flere som opererer samtidig. For å oppnå denne effekten har transaksjoner ikke lov til å lese eller oppdatere data som allerede er modifisert av andre transaksjoner som ikke har committet. Transaksjoner som implementerer isolasjonsegenskapen unngår følgende uheldige situasjoner:

- **Lost Updates:** Resultatet av transaksjoner som gjør oppdateringer på de samme objektene er at de kan skrive over hverandres resultater.
- **Cascading Abort:** Resultatet av at en transaksjon T1 leser objekter som er modifisert av en transaksjon som senere aborterer, er at T1 også må abortere.

Varighet (eng: durability)

Oppdateringer utført av comittete transaksjoner må aldri bli glemt av systemet, og må derfor lagres permanent på stabilt lager. Dette skal sikre at ingenting går tapt om systemet senere feiler.

For å oppnå konsistens i en database hvor flere transaksjoner kjører samtidig, er en avhengig av at DBMS garanterer at ACID egenskapene overholdes. Dette kan oppnås ved å bruke henholdsvis samtidighets- og feilhåndteringsprotokoller. Disse er beskrevet i delkapitlene 4.3 og 4.4.

4.2.2 Transaksjonstyper

Transaksjoner kan struktureres på en rekke forskjellige måter. Strukturen vil blant annet ha betydning for selve oppførselen til transaksjonen. Vi vil her se de to grunnleggende typene av transaksjoner, nemlig *flate* og *nøstede*. Sistnevnte finnes det en rekke varianter av, men her vil vi bare se standardutgaven, samt en variant som heter *åpne nøstede transaksjoner.*

Flate transaksjoner

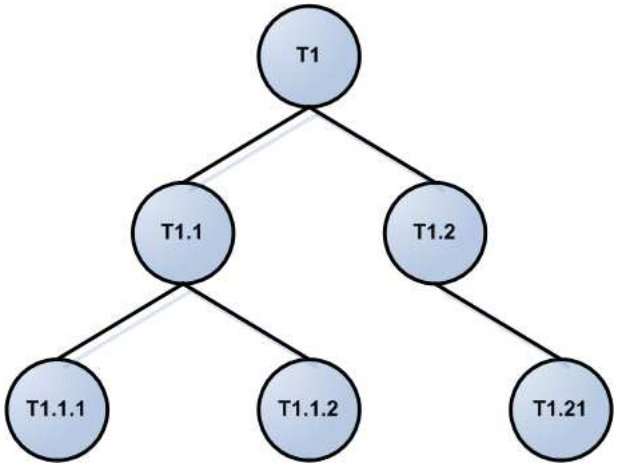
En flat transaksjon har bare ett nivå i sin utførelse i motsetning til en som er nøstet. Den er flat fordi den utfører alle sine operasjoner mellom *beginTransaction* og en *commit* eller en *abort*, og det er dermed ikke mulig å committe eller abortere deler av den som i nøstede transaksjoner.

Nøstede transaksjoner

I en nøstet transaksjonsmodell utvider man vanlige transaksjoner til å kunne være satt sammen av flere transaksjoner, som illustrert i figur 4.1. En nøstet transaksjon har et toppnivå, en modertransaksjon, som igjen kan ha flere nivå av barnetransaksjoner underordnet seg. Transaksjoner på samme nivå kan kjøre samtidig, og så lenge de jobber på ulike objekter kan de kjøres fullstendig i parallell. En stor fordel her er at subtransaksjoner på samme nivå kan committe og abortere uavhengig av hverandre. Hvorvidt transaksjonen som helhet skal committe eller abortere blir avgjort av topptransaksjonen etter at de representative subtransaksjonene er ferdigstilte. Nøstede transaksjoner har også en fordel ved feilhåndtering ved at ucommittete subtransaksjoner kan rulles tilbake uten å påvirke andre subtransaksjoner.

Åpne nøstede transaksjoner

Åpne nøstede transaksjoner [37] er en variant av nøstede transaksjoner, hvor subtransaksjoner har lov til å utføre en endelig commit (eng. final commit), forutsatt at det brukes kompenserende transaksjoner. Fordelen med denne modellen er at man her oppnår mer fleksibilitet ved



Figur 4.1: Nøstede transaksjoner

at transaksjonene kan bryte med *atomiskhet* egenskapen. Dette fører til at ikke alle subtransaksjoner må fullføre korrekt for at en transaksjon skal kunne terminere. Modellen bryter også med *isolasjonskravet*, ved at kjørende transaksjoner tilgjengeliggjør sine delresultater for andre transaksjoner. Det forutsettes at operasjonene som bruker disse delresultatene kan kommutere (eng. commute) [37].

4.3 Samtidighetskontroll

Hensikten med samtidighetskontroll (eng. concurrency control) er å håndtere problemer som oppstår når to eller flere transaksjoner kjører i parallell. Et overordnet mål for all transaksjonshåndtering er at så mange transaksjoner som mulig kan kjøre i parallell, samtidig som at databasen alltid er konsistent. En ukontrollert kjøring av transaksjoner i parallell kan fort skape problemer, som eksempelvis *dirty reads* og *lost updates* [21]. De to sistnevnte hendelsene fører til at innholdet i en database blir ukorrekt, og dermed brudd på det andre ACID kravet, nemlig konsistens, noe som ikke kan aksepteres. På bakgrunn av dette er det derfor viktig å ha mekanismer som muliggjør stor grad av samtidighet, men også sørger for at databasen hele tiden befinner seg i en konsistent tilstand. Målet er at effekten av kjøringen av et sett med transaksjoner er som om transaksjonene hadde blitt utført i en seriell rekkefølge. Dette er kjent som *serialisertbarhet* og vil bli omhandlet i neste avsnitt. Videre vil det også undersøkes hvilke mekanismer (protokoller) som finnes for å oppnå denne typen kjøringer.

4.3.1 Serialiserbarhet - et korrekthetskriterium

Ideen bak serialiserbarhet er å oppnå samme effekt av et sett av transaksjoner som kjører samtidig som om de ble kjørt en for en, i seriell rekkefølge. Om en seriell utførelse antas å være konsistent, så må dette også her gjelde for transaksjoner som kjører i parallell, så sant deres historie er serialiserbar med den sekvensielle. En historie definert for et sett transaksjoner T = $T_{1,2}$, ..., T_n angir rekkefølgen for når de blir kjørt. En historie er serialiserbar om den er ekvivalent med en annen seriell historie, noe som betyr at de begge produserer samme resultat. Å

bestemme om en historie er serialiserbar er ikke alltid like trivielt. Det skilles mellom to typer av serialiserbarhet, nemlig *viewserialiserbarhet* og *konfliktserialiserbarhet*, som er beskrevet i seksjonene under.

Viewserialiserbarhet

En kjøring av et sett med parallelle transaksjoner er definert som viewserialiserbar om det finnes en mulig seriell kjøring slik at hver transaksjon leser de samme verdiene og de endelige verdiene i dataobjektene i databasen er de samme.

Et åpenbar ulempe er at det har vist seg at å å være et NP-komplett problem å forsikre seg om at en kjøring er viewserialiserbar [33]. Dette medfører at det er svært vanskelig å finne en algoritme for viewserialiserbarhet. En annen måte er å undersøke operasjoner som er i konflikt. Dette vil bli studert nærmere i neste avsnitt.

Konfliktserialiserbarhet

En måte å forsikre seg om at en transaksjonshistorie er serialiserbar er å undersøke om operasjoner er i konflikt, eksempelvis lese- og skriveoperasjoner. To operasjoner er definert for å være i konflikt om de tilhører to ulike transaksjoner hvor deres rekkefølge er av betydning. Om man har to operasjoner som begge jobber på samme objekt så vil de være i konflikt om minst en av operasjonene er en *skriveoperasjon* [4]. En måte å undersøke hvorvidt en historie er serialiserbar er å undersøke den tilhørende serialiseringsgrafen [4]. Dette er en rettet graf hvor man undersøker avhengighetsforholdet mellom transaksjoner, hvor nodene representerer transaksjoner og kantene er avhengigheter. Om en histories serialiseringsgraf ikke inneholder noen løkker (ingen transaksjoner er i konflikt), så er historien definert for å være serialiserbar.

Serialiserbarhet er altså en metode for å forsikre seg om en mulig kjøring av parallelle transaksjoner er gjennomførbar med hensyn til korrekhet. Vi skal nå gi en kort oversikt over generelle mekanismer for å oppnå serialiserbarhet; protokoller for samtidighetskontroll.

4.3.2 Protokoller for samtidighetskontroll

Det er vanlig å skille mellom to typer protokoller, nemlig *pessimistiske* og *optimistiske*. De vanligste måtene å implementere disse protokollene på er å bruke *låsing* eller *tidsmerking*, men det finnes også en rekke andre metoder.

Pessimistisk samtidighetskontroll

Denne metoden tar sikte på at konflikter *vil skje*, og synkroniserer transaksjoner ved å begrense lesetilgang til delte objekter. Om man benytter låsing så vil det bli satt en lås (fysisk eller logisk) på objekter slik at bare en transaksjon får tilgang om gangen. Dette gjelder kun for

operasjoner i konflikt, og gjelder i utganspunktet ikke for eksempelvis to leseoperasjoner. Om en operasjoner ikke får tilgang til et objekt (en annen transaksjon har lås på det), så må den vente. En måte for å oppnå serialiserte kjøringer er å bruke såkalt *to-fase låsing* (eng: 2PL). Som navnet antyder så består denne metoden av to faser, hvor transaksjonen i første fase skaffer seg de låsene den trenger, for så slippe dem i andre fase. En ulempe ved bruk av låsing er at vranglås (eng: deadlock) kan oppstå. Det som skjer her at to transaksjoner står og venter på hverandre for alltid (syklisk venting). Dette kan unngås om man bruker *tidsstempel* ordning [4]. En annen stor ulempe er når transaksjoner bruker lang tid. Dette fører til at andre transaksjoner må vente lenge før de kan fortsette å kjøre.

Optimistisk samtidighetskontroll

Denne metoden antar at konflikter sjelden skjer, og lar derfor transaksjoner helt fritt kunne lese og skrive til de objektene de måtte ønske. Når en transaksjon ønsker å committe blir dens operasjoner validert opp mot andre transaksjoner for å finne eventuelle konflikter.

4.4 Feilhåndtering

Feilhåndtering av databaser omhandler teknikker som bringer databasen tilbake til en konsistent tilstand når noe går galt. Hvis en transaksjon må abortere eller man må gjenopprette systemet etter en krasj, trenger man feilhåndtering. Teknikker som logging, undo/redo algoritmer og ARIES [14] er vanlig for å oppnå god feilhåndtering i tradisjonelle databasesystemer. I vår oppgave er slike grunnleggende mekanismer lite relevant fordi de tas hånd om av hver DBMS lokalt. Imidlertid er det et åpent forskningsspørsmål hvordan en skal ta seg av tilbakerulling av transaksjoner med spesielt lang levetid, spesielt de som opererer i distribuerte miljøer. En av metodene som kan brukes til dette kalles for kompenserende transaksjoner.

4.4.1 Kompenserende transaksjoner

Hvis en transaksjon feiler etter den har gjort endringer i databasen, kan det være nødvendig å rulle tilbake transaksjonen. Dette vil si at alle endringer transaksjonen gjorde må fjernes, og tilstanden til databasen må være som om transaksjonen aldri hadde vært utført. Hvis man tillater at en transaksjon T2 har lest en verdi som transaksjon T1 har skrevet før T1 har committet, kan man bli nødt til å abortere T2 hvis T1 av en eller annen grunn må abortere. Dette fenomenet kalles *cascading abort*, og er svært komplekst og ressurskrevende for en database å utføre. Hvis man ønsker et system som lar transaksjoner dele verdier på denne måten må man derfor finne en måte å håndtere slike situasjoner på.

Når oppdateringer (committet eller ikke committet) av en transaksjon T har blitt lest av en annen transaksjon, har T blitt eksternalisert. Kompensering brukes for å omgjøre en eksternalisert transaksjon T, og T kalles da for den transaksjonen det ble kompensert *for*. Transaksjonene som har blitt påvirket av T ved å lese verdiene den har skrevet sies å være avhengige av transaksjonen T. Målet med å bruke kompenserende transaksjoner er å beholde effektene til de transaksjonene

som er avhengige av T, i det tilfellet hvor man omgjør T. Siden T godt kan være en transaksjon som har committet, har man større fleksibilitet enn i tradisjonell feilhåndtering.

Ved å bruke kompenserende transaksjoner garanterer man ikke at alle de direkte og indirekte effektene av T blir fjernet. T blir omgjort på en semantisk måte, ikke ved en fysisk gjenopprettelse av en tidligere tilstand. Tilstanden man får etter at den kompenserende transaksjonen har blitt kjørt kan avvike noe fra tilstanden man hadde fått hvis den transaksjonen man kompenserer for aldri hadde eksistert. Det er opp til applikasjonsprogrammereren å forsikre at ønsket grad av omgjøring blir oppnådd.

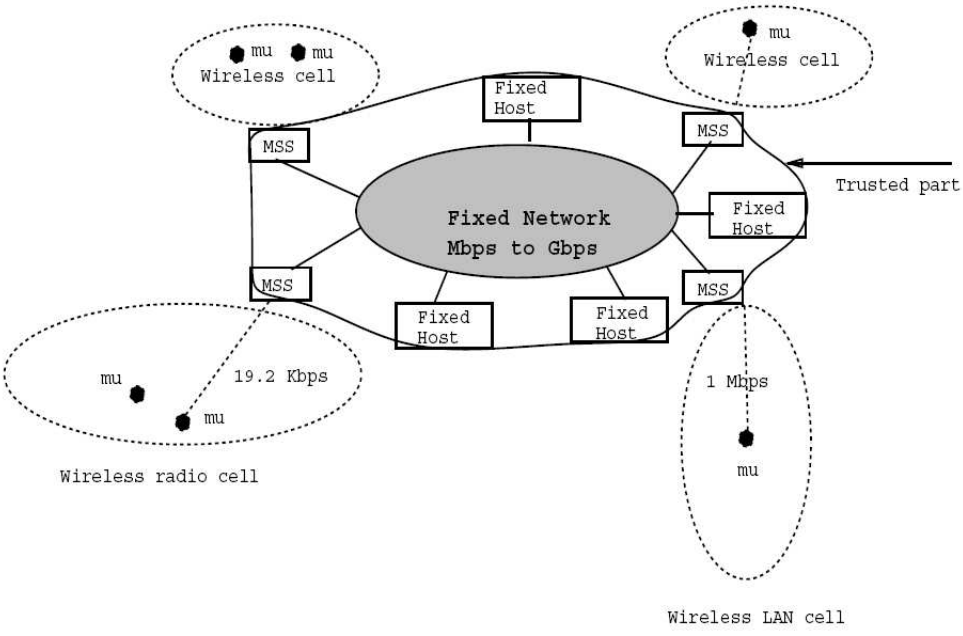
Kapittel 5

Beskrivelse av mobile databasemiljø

Dette kapittelet vil ta for seg karakteristikkene til databasesystemer som involverer mobile enheter. Bærbare datamaskiner, håndholdte enheter og mobiltelefoner kan kobles sammen i trådløse nett og danner dermed mobile databasemiljø som har egenskaper som skiller seg fra tradisjonelle databasemiljøer.

5.1 Generell arkitektur for mobile nettverk

I [25] beskrives det en generell arkitektur for et system som har flere forskjellige typer mobile og stasjonære enheter koblet sammen i et datanettverk. Figur 5.1 viser oppbygningen av denne modellen.



Figur 5.1: Arkitektur for mobile nettverk, fra [25]

Her har en *mobil enhet* (eng: MH - Mobile Host) mulighet til å koble seg opp mot det faste nettverket eller andre mobile enheter via en trådløs forbindelse. En *databasetjener* er en stasjonær maskin i det faste nettverket som ikke har mulighet for å koble seg til en mobil enhet. For å oppnå forbindelse mellom en mobil enhet og en databasetjener trengs det en *mobil støttestasjon* (eng: MSS - Mobile Support Station). En slik mobil støttestasjon fungerer som et bindeledd mellom en mobil enhet og en databasetjener eller mellom en mobil enhet og andre mobile enheter. En mobil støttestasjon trenger et fysisk grensesnitt som for eksempel en basestasjon for å etablere trådløs kommunikasjon med de mobile enhetene.

De mobile datamaskinene er gruppert inn i enheter kalt celler, der hver celle er representert som et geografisk område dekket av en trådløs kommunikasjonsstruktur. En slik kommunikasjonsstruktur kan være et trådløst lokalnettverk (WLAN), et ad hoc nettverk, et mobiltelefonnett eller en kombinasjon av disse. Hver MH kommuniserer med en MSS som dekker cellen den tilhører. Prosessen med å gå over til en ny celle og bli tilordnet en ny MSS kalles for hand-off eller hand-over. En viktig forskjell mellom WLAN nettverk og mobiltelefonnettverk er at enheter koblet sammen i et WLAN kan kommunisere direkte med hverandre, mens enheter tilkoblet mobiltelefonnettverket må gå via basestasjoner for å kunne kommunisere med hverandre.

5.2 Tekniske begrensninger i mobil kommunikasjon

Trådløs kommunikasjon har oftere problemer enn vanlig kommunikasjon gjennom datakabler fordi signalet blir lett kan bli blokkert eller utsettes for støy og ekko. Derfor har gjerne trådløse overføringer lavere båndbredde, høyere feilrate og mer frekvente frakoblinger.

Mobiltelefonnettet i Norge har nettopp fått støtte for UMTS, og her er maksimal overføringshastighet begrenset til 384 kbit/s. Den eldre EDGE teknologien som er en oppgradert versjon av GSM nettet støtter rundt 100-200 kbit/s. Disse hastighetene er imidlertid varierende i forhold til faktorer som: type tjeneste, trafikk i nettet, radioforholdene på aktuelle steder og type telefon eller PC-kort som benyttes [3]. Nåværende trådløse kort for lokalnett har til sammenligning hastigheter rundt 54 mbit/s, noe som åpner for langt flere bruksområder. Trådløse lokalnett har imidlertid de samme problemene som frekvente frakoblinger og støy på signalet.

Mobile enheter har også ofte begrenset batterikapasitet, prosessorhastighet, minnekapasitet og skjermstørrelse. I tillegg er de mer utsatt for tap eller ødeleggelser. Dette gjør at en frakobling i et mobilt databasemiljø må sees på som en normal hendelse, for eksempel kan en mobil enhet velge å koble fra for å spare på batteriforbruket.

5.3 Mobile heterogene distribuerte databasesystemer

Hver enkelt datamaskin med nok ressurser har mulighet for å kjøre et databasesystem (DBMS) som kan tilby tjenester for klienter i nettverket. Når klientene involverer mobile enheter, har vi et mobilt databasesystem. Hvis vi i tillegg involverer flere databasesystemer på flere enheter som er spredt rundt i nettverket, sier vi at vi har et distribuert mobilt databasesystem. I veldig store nettverk med komplekse oppgaver er det overveiende sannsynlig at de forskjellige databasene

ikke er av samme type, men at de er levert av forskjellige produsenter og kjører på forskjellige operativsystemer. Slike databasesystemer kalles heterogene databasesystemer, og dermed kan man definere det mest generelle systemet vi må ta hensyn til som et mobilt heterogent distribuert databasesystem.

Kapittel 6

Utfordringer i mobile databasemiljø

Kapittel 5 beskrev de spesielle egenskapene til det mobile databasemiljøet. I dette kapittelet skal vi bygge videre på dette og se hvilke problemer og utfordringer man står ovenfor i slike systemer og hvilke generelle måter dette kan løses på.

Det er hovedsakelig tre utfordringer som er spesielle for mobile databasesystemer: frakobling, mobilitet og økt samtidighet. Hyppige frakoblinger fører med seg problemer med å committe transaksjoner og med å slippe låser som en transaksjon holder. En høy grad av samtidighet (eng: concurrency) er mer verdifullt i mobile miljøer fordi man har mindre båndbredde og mindre stabil tilkobling slik at venting på at andre transaksjoner skal gjøre seg ferdig med dataverdier er mer kostbart og tidkrevende. Mobilitet av deltakere i en transaksjon fører i tillegg med seg en del problemer i tilknytning til administrasjon av transaksjonen.

6.1 Frakobling

Alle distribuerte systemer må håndtere delvis feil (eng: partial failure) på en god måte slik at ikke hele systemet havner i en ukonsistent tilstand. En vanlig måte å forsikre seg om at alle deltransaksjonene i en stor global transaksjon blir utført er å bruke 2PC protokollen [14], men ulempen med 2PC er at det krever synkron kommunikasjon mellom deltakerne. Hvis en deltaker mister tilkoblingen så lenge at det overstiger timeout perioden, vil hele transaksjonen bli abortert.

En måte å avhjelpe dette problemet på er å bruke asynkrone modeller og protokoller der man ikke er avhengig av svar fra alle deltakerne umiddelbart. Metoden som gjerne brukes er at man sjekker ut en kopi av databasen som man kan bruke mens man er frakoblet fra nettverket. Dette er beskrevet i [24], der man skiller mellom fire forskjellige utsjekkingsmodi: basic sign-off, check-out, relaxed check-out og optimistic check-out. Her er det forskjellige regler som styrer om man har lov til å oppdatere databasen og hvordan disse endringene integreres når hele databasen samles igjen.

6.2 Økt samtidighet

En transaksjon i et mobilt distribuert databasesystem kan bli oppholdt lenge hvis den trenger dataobjekter som er låst av andre transaksjoner som ikke slipper låsene sine innen rimelig tid på grunn av frakobling eller dårlig forbindelse. Økt samtidighet innebærer at flere transaksjoner kan kjøre i systemet på samme tidspunkt, og måten man oppnår dette på er at man bruker låsing i mindre grad enn det som er vanlig.

Hvis man skal øke samtidigheten er man nødt til å tillate at transaksjoner deler verdier seg imellom. Dette strider mot det tradisjonelle isolasjonskravet som transaksjoner er pålagt ifølge ACID egenskapene, se kapittel 4. Det vanlige isolasjonsnivået satt som standard i de fleste databasesystemer er *read committed*, det vil si at en transaksjon aldri vil lese data som en annen transaksjon har skrevet men ikke committet ennå. Hvis man løsner på dette kravet og sier at slike *dirty reads* er tillatt, så må man ta hensyn til at transaksjoner kan lese data som senere vil bli gjort ugyldig ved at transaksjonen som skrev disse dataene ruller tilbake. Da kan man bli nødt til å utføre en *cascading abort* som ruller tilbake alle transaksjonene som har brukt denne verdien.

For å unngå slike massive tilbakerullinger av transaksjoner er det mulig å innføre nye regler som sier at kun enkelte transaksjoner trenger å bli rullet tilbake hvis en innlest verdi gjøres om. Dette er beskrevet i [34], der man bruker såkalte *abort dependencies* for å definere sett av avhengige transaksjoner.

6.3 Mobilitet

Mobilitet har to betydninger i mobile distribuerte databasesystemer. Man har fysisk mobilitet når en mobil enhet flytter seg fra celle til celle, og man har logisk mobilitet når programkode flytter seg mellom noder i nettverket.

6.3.1 Fysisk mobilitet

Her forutsettes det at strukturen på det trådløse nettet er bygd opp som et mobiltelefonnett med mobile støttestasjoner koblet sammen i et fast nettverk, og mobile enheter som er koblet opp mot en mobil støttestasjon via en trådløs forbindelse.

Når den mobile enheten ønsker å utføre en transaksjon sier den fra om dette til den mobile støttestasjonen (MSS) som videresender forespørslene fra den mobile transaksjonen til en eller flere database stasjonert på det faste nettet.

Et viktig spørsmål er hvor selve transaksjonshåndteringen (logging, recovery, commit prosessering etc.) for den *mobile* transaksjonen bør finne sted? En mulig løsning er å la den mobile enheten håndtere transaksjonene selv. Dette er vanligvis ikke et reelt alternativ med den teknologien som finnes i dag, siden mobile enheter har begrenset prosessorkraft, minne og lagringskapasitet. Et annet hinder er at dette vil kreve integrasjon og konflikthåndtering i de tilfellene man er nødt

til å samordne dataene på en mobil enhet med de som ligger på en større databasetjener.

Å logge status til en transaksjon vil også by på problemer, da dette vil føre til unødvendig mye trafikk på en relativt treg trådløs forbindelse. I slike tilfeller vil det også være vanskelig å håndtere uforutsette frakoblinger.

Et alternativ er å la databasetjeneren håndtere de mobile transaksjonene. Dette vil bli en svært vanskelig oppgave om det finnes et stort antall databaser på det faste nettet. Denne løsningen vil i såfall føre til mye ekstra programvare på tjenerene, noe som ikke er å foretrekke om en ønsker autonome databaser.

Det vil nå tas utgangspunkt i en tredelt løsning bestående av den mobile enheten (klient), databasen (tjener) og et mellomlag som vil ta seg av selve transaksjonshåndteringen av den mobile transaksjonen. Dette mellomlaget vil befinne seg på det mobile støttesenteret (MSS). Med bakgrunn i [13] vil ulike måter man kan håndtere mobile transaksjoner på bli studert. I denne artikkelen er det gjort en grundig ytelselsvurderig av tre måter å håndtere mobile transaksjoner på, nemlig HMSS, AMSS og MOVE. Disse vil bli beskrevet samtidig som de viktigste konklusjonene angående deres anvendbarhet vil bli oppsummert. Den viktigste konklusjonen er kanskje at det ikke finnes noen metode som alltid er best.

HMSS

HMSS står for Home Mobile Support Station, og her vil transaksjonshåndteringen foregå på MSSen som assosieres med cellen som den mobile enheten starter i. Om den mobile enheten flytter seg over til andre celler så må den kommunisere via andre MSSer når den kjører transaksjoner. Følgelig er er dette en metode som er å foretrekke når en mobil enhet befinner seg mye innenfor sin egen celle, og hvor det blir kjørt enkle transaksjoner. For mobile enheter som flytter seg mye på tvers av mange celler så vil denne metoden fungere dårlig. Dette skyldes at forespørsler da må sendes via mange andre MSSer til MSSen som den mobile enheten startet i (home).

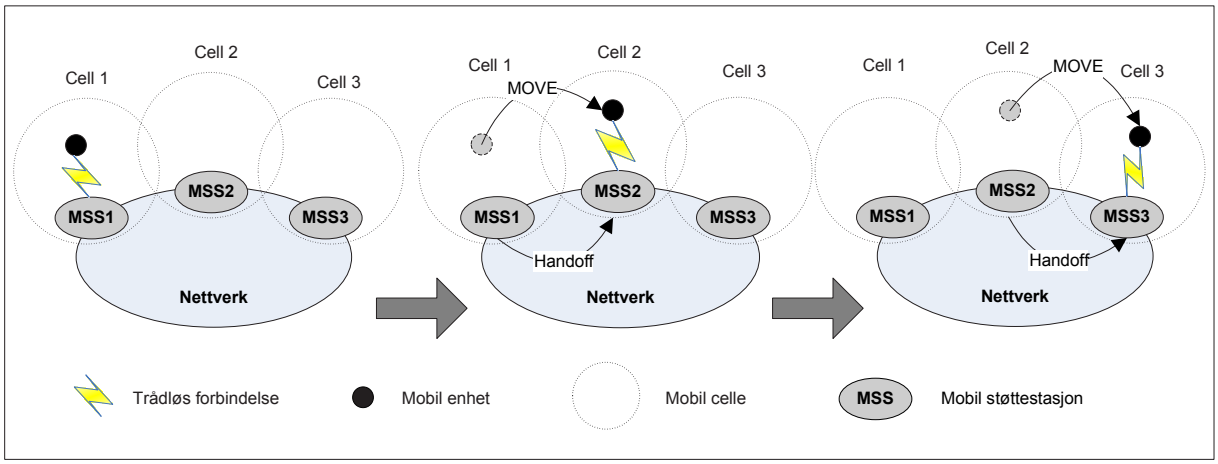
AMSS

AMSS står for Anchor Mobile Support Station, og her foregår transaksjonshåndteringen fra MSSen (ankerstedet) hvor den mobile transaksjonen ble initiert. En transaksjon som flytter seg fra celle til celle samtidig som den hele tiden initierer nye transaksjoner kan da oppleve at transaksjonene blir håndtert fra flere MSSer i det faste nettverket. Dette er en metode som fungerer bra for enkle transaksjoner hvor den mobile enheten beveger seg mellom hjemcelle og naboceller.

MOVE

Når en mobil enhet beveger seg fra celle til celle vil selve transaksjonhåndteringen flyttes over til den gjeldende MSSen, som illustrert i figur 6.1 Dette medfører at all metadata må overføres

fra MSS til MSS etterhvert som den mobile brukeren kommer over i en ny celle. På denne måten håndteres alltid transaksjonene fra den MSSen som assosieres med den cellen den mobile enheten for øyeblikket befinner seg. MOVE metoden fungerer bra for kompliserte transaksjoner.



Figur 6.1: MOVE - transaksjonshåndteringen flyttes fra celle til celle

6.3.2 Logisk mobilitet

Logisk mobilitet innebærer at programkode og status, gjerne i form av en instans av en klasse i et objektorientert språk, har mulighet til å flytte seg til en annen node i nettverket og eksekvere der. Fordelen med dette er at man får en løs kobling mellom komponentene i det mobile distribuerte systemet, slik at det for eksempel blir lettere å arbeide i frakoblet modus. Man kan se for seg at en mobile kodesnutter forflytter seg fra klienten til forskjellige databasetjenere og utfører transaksjoner før de sammen committer ved hjelp av en koordinator og så gir beskjed til klienten om hva utfallet av transaksjonen var. Hvis klienten skulle miste tilkoblingen i en periode mens dette pågår, har det lite å si for ytelsen til systemet.

Kapittel 7

Mobile transaksjonsmodeller

Mobile transaksjoner er som kjent ganske ulike i forhold til transaksjoner i vanlige sentraliserte og distribuerte databaser. I kapittel 6 ble det presentert en rekke typiske utfordringer i forbindelse med mobil transaksjonshåndtering, som blant annet *frakobling*, *økt samtidighet* og *mobilitet*. En tradisjonell transaksjonsmodell med streng ACID utførelse ville aldri vært i stand til å imøtekomme de kravene som mobile transaksjoner stiller på en effektiv og tilfredsstillende måte. Det er derfor et behov for en transaksjonsmodell som er mer tilpasset og egnet for mobil transaksjonshåndtering.

Til tross for at det i dag finnes en rekke transaksjonsmodeller for mobil transaksjonshåndtering, så er det få eller ingen som gir en god støtte for frakobling, økt samtidighet og mobilitet i en og samme modell. Ofte er modellene spesialiserte innenfor et av områdene. Målet med dette kapittelet er finne modeller som løser bestemte problemer på en god måte, og la dem stå som et godt eksempel for hvordan dette skal gjøres.

I 2004 ble det laget en avhandling som gjennomgår state of the art når det gjelder mobile transaksjonsmodeller [35], og denne rapporten underbygger vår påstand om at det ikke finnes noen transaksjonsmodeller som er gode på alle alle de overnevte problemene. Spesielt er det få modeller som tar for seg økt samtidighet av transaksjoner.

Nedenfor beskriver vi fem modeller som vi har hentet inspirasjon fra i utvikling av eksport-import modellen, og vi mener også at disse modellene representerer deler av state of the art.

7.1 AMDB arkitekturen

I artikkelen 'Sharing Mobile Databases in Dynamically Configurable Environments' [5] blir det presentert en arkitektur som har flere likhetstrekk med arkitekturen vi har planlagt å utvikle for import-eksport modellen seinere i denne oppgaven, og AMDB vil derfor bli kort introdusert her.

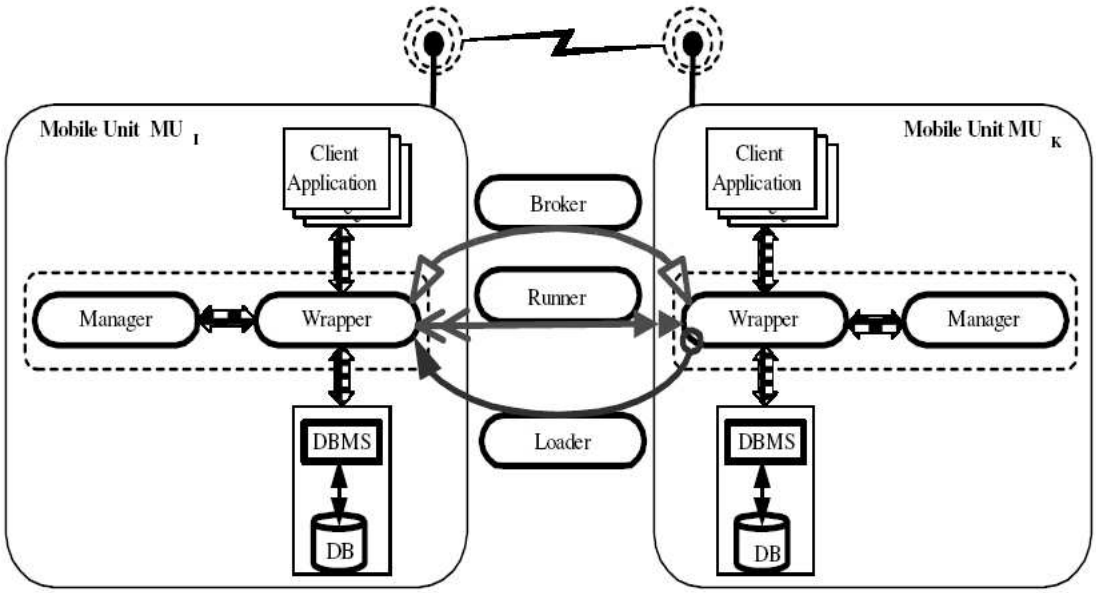
Modellen opererer i et mobilt miljø med flere autonome, distribuerte, heterogene og mobile databaser. Dette miljøet betegnes som et mobilt databasesamfunn (MDBC), hvor hver data-

basebruker kan få tilgang til databasene via trådløs datakommunikasjon. For å få tilgang til disse databasene trengs det en mellomvarearkitektur, og denne har forfatterne kalt for AMDB, *Accessing Mobile Databases*.

7.1.1 Agentklasser

AMDB arkitekturen er basert på programvareagenter, og det finnes to typer av disse: stasjonære agenter og mobile agenter. De stasjonære agentene kan enten være bestyreragenter (eng: manager agents) eller grensesnittagenter (eng: wrapper agents). Bestyreragentene har ansvar for å håndtere lokale ressurser som kan utføre oppgavene som mobile agenter vil ha gjort. Grensesnitt agentene tilbyr et grensesnitt mellom de mobile brukerne og AMDB plattformen, og dermed har de som oppgave å lage et kjørbart miljø for mobile agenter samt støtte en felles datarepresentasjon for de lokale dataene på hver maskin. Siden det er høy sannsynlighet for at databasene er heterogene, brukes XML som standard for datautveksling og distribusjon av databaseskjemaer.

Klassen av mobile agenter har mulighet for å transportere seg fra et MDBC medlem til et annet, og det finnes tre typer av dem: løpeagenter (eng: runner agents), transportagenter (eng: loader agents) og forhandleragenter (eng: broker agents). Løpeagenter har ansvaret for å utføre oppgaver på vegne av de mobile brukerne av systemet. Dette kan være spørringer, oppdateringer eller skjemaevolusjon. Transportagenter tar resultatene fra spørringene tilbake til den enheten de kom fra. Forhandleragenter samler sammen databaseskjemaene til de mobile databasene og distribuerer dem til alle de mobile brukerne.



Figur 7.1: AMDB arkitektur, fra [5]

7.1.2 Datautveksling og spørringer

Figur 7.1 viser arkitekturen til AMDB modellen i et tilfelle med to mobile enheter. Når et nytt medlem skal bli med i det mobile databasesamfunnet, opprettes det en forhandleragent som flytter seg fra database til database og spør etter de lokale databaseskjemaene. Disse skjemaene, som er på XML format, returneres så til det nye medlemmet som kan bruke informasjonen til å utføre spørringer.

Spørrespråket som brukes kalles MXQuery, som er et tillegg til multidatabasespråket XQuery. Ved å bruke MXQuery kan man oversette til det lokale spørrespråket hos hver enkelt database, og man kan dermed hente ut data fra heterogene kilder. Spørringene kan deles inn i to hovedgrupper, nemlig lokale og globale. Lokale spørringer involverer kun operasjoner på objekter i den lokale databasen, mens globale spørringer involverer operasjoner over flere mobile distribuerte databaser.

Når en mobil bruker overfører en global spørring til den lokale grensesnittagenten, oppretter denne agenten en løpeagent som får ansvaret for å utføre den globale spørringen. Spørreagenten genererer en utførelsesplan for spørringen som representeres med et operatortre. Nodene i operatortreet består av algebraiske operasjoner eller datakilder. Roten av treet representerer det endelige resultatet. Treet deles så opp i fragmenter, og hvert fragment flytter seg ved hjelp av en løpeagent til den mobile databasen som har den de aktuelle objektene lagret lokalt. Til slutt samles alle løpeagentene hos den brukeren som initierte spørringen og setter sammen svaret der.

7.1.3 Commit prosessering

For å sikre at transaksjonene overholder ACID egenskapene, brukes en variant av 2PC som kalles mobil 2PC. Her klones det flere løpeagenter som forflytter seg til de medlemmene som er deltakere i transaksjonen. Alle medlemmene gir agenten sin stemme (ja/nei), og løpeagentene drar så tilbake til koordinatoren. Koordinatoren sjekker om det finnes en nei-stemme, og hvis det gjør det sendes det en løpeagent med en melding til alle medlemmene om at transaksjonen skal aborteres. Hvis alle stemmene er ja-stemmer, sendes det melding om at transaksjonen skal committes.

7.1.4 Evaluering av modellen

Evalueringen av modellen tar utgangspunkt i hvordan de mobile utfordringene som er beskrevet i kapittel 6 har blitt løst.

Frakobling

AMDB arkitekturen er basert på agenter, og den er derfor sårbar hvis en agent befinner seg på en mobil enhet som mister tilkoblingen. Agenten vil da sitte fast, og hvis den er nødt til å besøke flere mobile enheter for å fullføre arbeidet sitt vil transaksjonen gå i stå. I det spesialtilfellet

hvor man bruker en agent som koordinator til å gjennomføre mobil 2PC for å committe en transaksjon, vil det lages en backupagent som kan overta koordinatorjobben hvis den originale agenten låser seg. Sårbarheten er imidlertid tilstede ved normal bruk av agenter til innsamling av skjemaer og utførelse av operasjoner på de mobile databasene.

Videre er utførelsen av mobil 2PC en synkron protokoll, og den krever derfor at alle medlemmene svarer ja eller nei til å committe innen en viss tidsperiode. Hvis en av medlemmene mister tilkoblingen sin over lengre tid, blir hele transaksjonen nødt til å abortere.

Imidlertid støtter modellen dynamisk opprettelse av det mobile databasemiljøet, og når nye medlemmer kommer til eller gamle medlemmer kobler seg fra vil oppdaterte databaseskjemaer distribueres.

Økt samtidighet

Modellen støtter ingen form for deling av data mellom transaksjonene under utførelse, den baserer seg kun på tradisjonelle ACID krav. For å øke arbeidsmengden som kan bli utført samtidig, klones det flere agenter som kan utføre sitt fragment av operatortreet på en mobil database. Samtidig utførelse forutsetter at det ikke er avhengigheter mellom fragmentene.

Mobilitet

Fysisk mobilitet er lite omtalt i AMDB, men logisk mobilitet er godt dekket siden alle de mobile agentene implementerer denne egenskapen. Agentene flytter seg fra enhet til enhet og utfører sine oppgaver der, og de tar med seg programkode som gjør dem i stand til å utføre sine operasjoner mot databasen. Agentsystemet kan sies å være en full gjennomføring av prinsippet om logisk kodemobilitet.

7.2 PRO-MOTION transaksjonsmodellen

PRO-MOTION [36] er en transaksjonsmodell som støtter utførelse av databaseoperasjoner på mobile enheter i frakoblet modus. For å få til dette bruker man *compacts* som inneholder aksessmetoder, tilstandsinformasjon og konsistensrestriksjoner slik at lokal utførelse av transaksjoner er mulig.

7.2.1 Problemscenario

Modellen tar utgangspunkt i en vanlig mobil databasearkitektur der man har databasetjenere som befinner seg på en eller flere stasjonære maskiner, og en samling av mobile enheter som utfører operasjoner mot data på disse databasetjenerne. Når de mobile enhetene utfører disse operasjonene settes det i gang en spørsmål-svar sekvens for hver operasjon, og dette krever

mye av båndbredde og batterikapasitet. Hvis den mobile enheten skulle bli frakoblet, stopper også utførelsen opp siden spørsmål-svar sekvensen blir brutt.

7.2.2 COMPACT objektet

PRO-MOTION ønsker å forbedre transaksjonshåndteringen ved å cache data på de mobile enhetene i form av *compacts*. En compact er en slags kapsel som innholder en kopi av et utvalg data fra databasetjeneren sammen med *forpliktelser* (eksempelvis en deadline), *restriksjoner* (eksempelvis et sett av tillatte operasjoner) og *tilstandsinformasjon* (eksempelvis antall aksesser til objektet). Databasetjeneren delegerer kontroll over dataene til den mobile enheten, og den mobile enheten tar på seg ansvaret for dataene og for å overholde de reglene som har blitt definert. Databasetjeneren mottar dermed periodiske oppdateringer for en compact istedenfor å prosessere alle operasjonene som den mobile enheten utfører.

En master-kopi av de fragmenterbare dataene ligger på databasetjeneren, og de mobile enhetene spesifiserer granulariteten av dataene som skal caches når de etterspør et compact objekt. Datapartisjonen fjernes så fra master-kopien og kopieres til den mobile enheten innpakket i en compact. Dataene inni compact objektet er kun tilgjengelige for transaksjoner på den mobile enheten. Når en compact skal oppdateres igjen på databasetjeneren, må effektene av operasjonene som ble utført av transaksjoner på den mobile enheten beholdes.

7.2.3 Infrastruktur

Ofte vil det ikke være mulig å implementere logikken for å støtte compact objekter direkte i databasetjeneren. I et slikt tilfelle kan man legge denne funksjonaliteten hos en *compact manager* som fungerer som en front-end mot databasetjeneren. Compact manageren kan kjøre på en uavhengig maskin, eller den kan kjøre på den samme maskinen som databasetjeneren. PRO-MOTION bruker så en åpen nøstet transaksjonsmodell som basis for samtidighetskontroll og recovery for mobile transaksjoner som utføres mot tjeneren. For databasetjeneren vil compact manageren se ut som en ordinær klient som utfører kun en langvarig og stor transaksjon. Denne langvarige transaksjonen blir roten i den nøstede transaksjonen, og mobile transaksjoner blir dannet som barn til roten. Hver subtransaksjon kan comitte eller abortere på egen hånd så lenge konsistenskravene i compact objektet overholdes. Ansvaret for korrekt utførelse av mobile transaksjoner ligger på den mobile enheten og gjennomføres ved å bruke de metodene som ligger i compact objektet. Rottransaksjonen håndteres av databasetjeneren og committes av compact manageren.

7.2.4 Evaluering av modellen

Evalueringen av modellen tar utgangspunkt i hvordan de mobile utfordringene som er beskrevet i kapittel 6 har blitt løst.

Frakobling

Modellen søker i høy grad å løse problemene som oppstår ved frakobling av mobile enheter. Muligheten for å utføre operasjoner mot lokale kopier av dataene i form av compact objekter gjør at asynkron kommunikasjon mellom databasetjeneren og den mobile enheten er mulig. Synkronisering mellom den mobile enheten og databasen utføres når dette er nødvendig, for eksempel når det kreves av applikasjonen på den mobile enheten eller når enheten er i ferd med å gå tom for batteri. Et annet positivt aspekt ved å bruke denne måten å utføre oppdateringer på er at man sparer båndbredde i det trådløse nettverket.

Økt samtidighet

PRO-MOTION har ingen mekanismer som øker mengden transaksjoner som kan kjøre samtidig i systemet.

Mobilitet

Modellen tar høyde for at en mobil enhet skal kunne flytte seg fra en mobil støttestasjon til en annen og utføre en såkalt *handoff* protokoll. Denne protokollen kan være så enkel som å opprette nye kommunikasjonskanaler, eller så kompleks som å forflytte prosesser og database-transaksjoner som er under utførelse. PRO-MOTION definerer ikke hvordan dette skal gjøres, siden dette ikke er fokus for problemet modellen søker å løse.

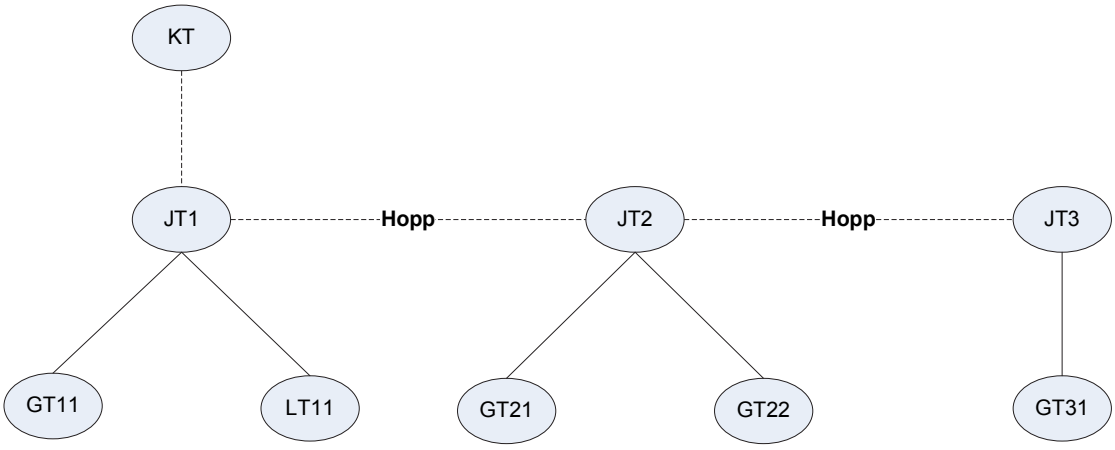
7.3 Kangaroo transaksjonsmodellen

I mobile transaksjonsmiljø er det viktig å ha gode mekansismer som kan håndtere kontrollen av transaksjoner når en mobil enhet flytter seg fra en mobil celle til en ny. Da mobile transaksjoner rent tidsmessig ofte er svært lange, er det fordelaktig å kunne bevare transaksjon selv om brukeren beveger seg mye. Kangaroo er kanskje den transaksjonsmodellen som har blant de beste løsningene når det gjelder støtte for mobilitet. Først av alt vil selve strukturen til de mobile transaksjonene bli presentert. Videre vil ulike moduser Kangaoo kan kjøre under bli beskrevet. Til slutt vil modellen bli evaluert ut fra aspekt som *frakobling*, *samtidighet* og *mobilitet*. Dette kapittelet er basert på den opprinnelige versjonen av Kangaroo modellen [12], så det er ikke blitt tatt hensyn til eventuelle modifikasjoner og forbedringer.

7.3.1 Kangaroo transaksjoner

Kangaroo modellen er basert på bruk av tradisjonelle transaksjoner *Local Transactions (LT)*, hvor selve transaksjonshåndteringen blir styrt av de lokale databasesysteme i det faste nettverket. Videre opererer Kangaroomodellen med *globale transaksjoner* (GT), som består av en sekvens av globale og lokale transaksjoner. Globale transaksjoner er dermed definert rekursivt av andre globale transaksjoner som til slutt ender opp i lokale transaksjoner som blir kjørt av de lokale databasesystemene.

Globale transaksjoner utgjør grunnlaget for de mobile transaksjonene i kangaroo modellen. Mobile transaksjoner blir ofte karakterisert som ’hoppende’ transaksjoner, da de migrerer fra basestasjon til basestasjon i det mobile nettet. De mobile transaksjonene i denne modellen heter derfor Kangaroo transaksjoner (KT), oppkalt etter det hoppende dyret i Australia.



Figur 7.2: Kangaroo transaction

Figur 7.2 illustrer strukturen til en Kangaroo transaksjon. Når en mobil enhet utfører en forespørsel om en transaksjon, opprettes en KT av den mobile støttestasjonen som er tilknyttet den mobile enheten. Subtransaksjonene til en KT kalles *Joey transaksjoner (JT)* og blir initiert og styrt av den mobile støttestasjonen. Når en KT opprettes, initieres det en JT som koordineres av MSSen som den mobile enheten for øyeblikket kommuniserer med. Når en mobil enhet beveger

seg over til en ny mobilcelle, opprettes det kontakt med en ny MSS, noe som medfører at en ny JT må opprettes. På denne måten 'hopper' den mobile transaksjonen (KT) fra MSS til MSS samtidig som at kontrollen av den også overføres til den nye MSSen (hand-off prosess) ved at en JT opprettes. At JTer koordineres av MSSer (del av KT) er forøvrig det som skiller de fra de globale transaksjonene.

I Kangaroomodellen kan JTer committe uavhengig av hverandre. Om man tar utgangspunkt i figur 7.2 så medfører dette at JT1 kan committe uavhengig av JT2 og JT3.

7.3.2 Prosesseringsmoduser

Kangaroomodellen skiller mellom to ulike moduser når det gjelder prosessering av transaksjoner; *kompenserende modus* og *splitt modus*.

Kompenserende modus

Om en JT feiler under denne modusen så må effekten av denne og alle foregående eller etterkommende JTer omgjøres (undo). Dette medfører at alle foregående JT som har committet må kompenseres. Denne modusen krever at bruker (eller systemet) gir tilstrekkelig med informasjon slik at det lar seg gjøre å opprette kompenserende transaksjoner.

Splitt modus

Denne modusen regnes som standardmodusen i Kangaroomodellen. Hvis en JT feiler så vil ikke flere globale eller lokale transaksjoner bli opprettet som del av den mobile transaksjonen. Avgjørelsen om å committe eller abortere de som allerede kjører er overlatt til de lokale databasesystemene. Foregående JTer vil heller ikke bli kompensert når en JT feiler.

Verken kompenserende modus eller splitt modus garanterer serialiserbarhet for kangarootransaksjoner. Serialiserbarhet kan bare garanteres for joeytransaksjoner som kjører i kompenserende modus.

7.3.3 Evaluering av modellen

Frakobling

Da Kangaroo modellen ble konstruert ble det tatt høyde for at den skulle ha god støtte for frakobling. Spesielt ble det lagt vekt på at brukere som kjørte lange transaksjoner skulle kunne koble fra for så ved et senere tidspunkt gjenoppta arbeidet. Ved at joeytransaksjonene kan committe uavhengig av hverandre så gir Kangaroomodellen rom for fleksibel atomiskhet (eng: relaxed atomicity), noe som medfører at kangarootransaksjoner kan ferdigstilles i flere steg (forutsatt kompenserende modus). Dette gjøres på bekostning av at *isolasjonskravet* for transaksjoner kan bli brutt. Årsaken til dette skyldes at låser settes og slippes på et lokalt transaksjonsnivå.

Så sant den mobile brukeren gjør kontrollerte frakoblinger, så kan den mobile transaksjonen stoppes og startes igjen uten problemer. Nå har det seg slik at de fleste frakoblinger i mobile

miljø ikke er forutsette og kan bestemmes av den mobile brukeren. Kangaroo modellen gir liten eller ingen støtte for slike uforutsette frakoblinger. I slike tilfeller vil den gjeldende joeytransaksjonen feile, og alle foregående JTer vil bli kompensert eller abortert. Det er derfor ikke mulig for den mobile brukeren å koble seg til ved et senere tidspunkt for så å gjenoppta arbeidet.

Økt samtidighet

Når det gjelder samtidighet mellom mobile transaksjoner så kan Kangaroo modellen kanskje være litt vanskelig å vurdere. Modellen fokuserer primært på mobilitet, og er ikke spesielt utformet for at mobile transaksjoner skal kunne dele verdier seg imellom. Kangaroo er som kjent konstruert for at de mobile transaksjonene skal kunne benytte seg av lokale transaksjoner fra autonome databasesystemer (gjennom globale transaksjoner). Konsekvensen av dette er at de lokale transaksjonene blir håndtert lokalt, noe som medfører at låser settes og slippes lokalt. Med andre ord, det er mulig for en mobil transaksjon å kunne lese verdier som en annen mobil transaksjon har lest, og dermed kan man si at Kangaroo støtter en form for deling. Nå er det viktig å merke seg at resultatet av dette er at *isolasjons* kravet (ACID) kan bli brutt uten at noen form for mekansime blir brukt for å kontrollere dette og sikre korrekthet. Så lenge denne delingen av verdier ikke er kontrolert så kan man ikke påstå at Kangaroo støtter noe form for samtidghet mellom mobile transaksjoner.

Mobilitet

Kangaroo er som kjent en modell som primært fokuserer på mobilitet for mobile transaksjoner. Modellen har en nøstet struktur kombinert med bruk av splittransaksjoner, og er dermed godt egnet for å kunne håndtere kontroll av transaksjoner som flytter på seg. Høy grad av mobilitet oppnås ved at den mobile transaksjonen (KT) kontrollers av en joey transaksjon (JT) som assosieres til en bestemt mobil støttestasjon (MSS) som brukeren er koblet til. Etterhvert som den mobile brukeren flytter på seg og beveger seg over i nye mobile celler, så vil kontrollen av den mobile transaksjonen flyttes til den nye MSSen ved at den gjeldende JTen splittes til to nye JTer, en gammel og en ny. Kangaroo er kanskje den modellen som aller best reflekterer selve mobilitets aspektet for mobile transaksjoner.

7.4 Reporting transaksjonsmodellen

Reporting [9] er en åpen nøstet transaksjonsmodell som tillater deling av data mellom transaksjoner under utførelse. Modellen opererer i et mobilt databasemiljø der transaksjoner på mobile enheter kan bygges opp av flere nivå med subtransaksjoner. Forfatterne argumenterer med at åpne nøstede transaksjonsmodeller er bedre egnet i mobile databasesystemer enn tradisjonelle ACID transaksjoner som ikke støtter deling av data, siden de begrensede ressursene til mobile enheter gjør at deler av transaksjonen bør utføres på den mobile enheten, mens andre deler av transaksjonen bør utføres på en fast tjener. Åpne nøstede transaksjonsmodeller som for eksempel Sagas [18] er mer fleksible siden de kan splitte opp sine operasjoner, samt at de tillater subtransaksjoner å committe eller abortere unilateralt slik at de delvise resultatene blir synlige. Imidlertid støtter ikke slike modeller deling av midlertidige data mellom transaksjoner under utførelse. Reporting transaksjonsmodellen tar sikte på å løse denne problemstillingen ved å innføre flere transaksjonstyper, og disse beskrives i neste avsnitt.

7.4.1 Transaksjonstyper

Modellen består av fire forskjellige transaksjonstyper: atomiske, ikke-kompenserbare, reporting og co-transaksjoner.

- **Atomiske transaksjoner:** Disse transaksjonene har vanlige abort og commit egenskaper. Kompenserbare transaksjoner og kompenserende transaksjoner er av denne typen.

- **Ikke-kompenserbare transaksjoner:** Slike transaksjoner er ikke assosiert med en kompenserende transaksjon. Ikke-kompenserbare transaksjoner kan committe når som helst, men siden de ikke er kompenserbare må de delegere alle sine operasjoner til foreldretransaksjonen slik at denne har ansvaret for å utføre commit.

- **Reporting transaksjoner:** En reporting transaksjon kan dele noen av sine midlertidige verdier til andre transaksjoner. Denne rapporteringen av verdier kan skje når som helst under utførelsen av reporting transaksjonen.

- **Co-transaksjoner:** Disse transaksjonene er rapporterende transaksjoner der ansvaret for utførelsen av co-transaksjonen overføres til den transaksjonen som mottar rapporten. Co-transaksjonen suspenderes når verdiene delegeres, og den fortsetter igjen når den mottar en rapport tilbake.

Reporting transaksjoner og co-transaksjoner er sterkt bundet sammen med andre transaksjoner og vil trolig utveksle mye data. Derfor bør disse transaksjonstypene migrere fra en enhet til en annen i det mobile databasemiljøet slik at de befinner seg på de enhetene som fører til minst mulig belastning av det trådløse nettverket. Hvordan denne migreringen bør foregå er imidlertid ikke beskrevet i modellen.

7.4.2 Delegering

Måten verdier utveksles på i reporting modellen er at transaksjonene foretar delegering. En slik delegering betyr at verdier eller operasjoner overføres fra en transaksjon til en annen. En delegering mellom to transaksjoner kan ikke gjennomføres om den transaksjonen som operasjonen delegeres til allerede har avsluttet eller abortert. På samme måte kan ikke en transaksjon delegere en operasjon som allerede har blitt avsluttet eller abortert.

7.4.3 Evaluering av modellen

Evalueringen av modellen tar utgangspunkt i hvordan de mobile utfordringene som er beskrevet i kapittel 6 har blitt løst.

Frakobling

Reporting modellen benytter seg av delegering for å dele midlertidige verdier mellom transaksjoner, og her skal vi se på hvordan denne mekanismen oppfører seg ved frakobling. For at en transaksjon skal kunne utføre en delegering må den ha direkte (synkron) kontakt med transaksjonen som den kommuniserer med. En nøstet transaksjon som prøver å delegere verdier eller operasjoner til en annen transaksjon den ikke får kontakt med har mulighet til å overføre ansvaret til en subtransaksjon. Denne subtransaksjonen må ligge å vente til den transaksjonen den forsøker å delegere til blir tilgjengelig, og dette vil komplisere utførelsen av transaksjonene. Samtidig vil det føre til tregere gjennomstrømning av transaksjoner i systemet. Dette kan begrunnes med at den transaksjonen som eventuelt må holdes tilbake kan holde låser som andre transaksjoner ønsker. En alternativ måte å møte problemet med delegering og frakobling er å la mobile enheter rapportere når de blir frakoblet, slik at en delegerende transaksjon kan forberede seg og eventuelt finne noen andre å utføre delegasjonen med. Dette kan fungere bra i tilfeller av kontrollerte frakoblinger. På bakgrunn av hva som er diskutert om delegering så vil det helt klart vært en fordel med asynkron kommunikasjon mellom transaksjoner, noe delegering ikke har støtte for.

Økt samtidighet

Hovedmålet til Reporting modellen er å øke gjennomstrømingen av transaksjoner ved å dele verdier dem i mellom. Bruk av kompenserende transaksjoner gjør at subtransaksjoner kan committe på egenhånd slik at de midlertidige resultatene blir tilgjengelige. De spesielle reporting transaksjonene og co-transaksjonene sørger i tillegg for deling av verdier mellom transaksjoner som er under utførelse. Reporting modellen har derfor god støtte for økt samtidighet mellom transaksjoner.

Mobilitet

Modellen beskriver ingen mekanismer for deling, men den legger til rette for at reporting transaksjoner og co-transaksjoner skal kunne migrere fra en enhet til en annen i det mobile miljøet.

7.5 To-delt replikering

To-delt replikering [22] er en solid transaksjonsmodell for mobile transaksjoner og tar høyde for å ha god støtte for blant annet frakobling og høy tilgjengelighet av data. Årsaken for at modellen er inkludert i dette dokumentet, er at den har likhetstrekk med eksport-import modellen. To-delt replikering er en todelt modell, med noder for mobile enheter og for stasjonære enheter. Kommunikasjonen mellom dem er asynkron, noe som gjør at modellen er spesielt sterk på frakobling. De mobile nodene kan lese data fra noder som eier dataene, koble fra, for så gjøre tentative opersjoner på dem. Når den mobile noden ved et senere tidspunkt kobler opp mot det faste nettet, oppretter den kontakt med de involverte base nodene (som den har lest data fra), for så forene de tentative opersjonene med de ekte dataene. Modellen bruker en modifisert utgave av master replication [22] for å oppnå korrekthet i de transaksjonelle utførelsene. Først av alt vil arkitekturen til modellen bli presentert. Videre vil de transaksjonelle utførelsene bli beskrevet, før modellen til slutt vil bli evaluert.

7.5.1 Arkitektur

Hovedkomponentene som inngår i denne modellen vil her bli presentert.

Modellen skiller mellom to ulike typer noder:

- **Mobile noder:** er noder som er frakoblet store deler av tiden. Disse lagrer kopier (replikater) av databasen, og kan være opphavet til såkalte *tentative* transaksjoner [22]. En mobil node kan være *master* (ansvarlig) for dataelementer.
- **Base noder:** er alltid tilstede (i motsetning til mobile noder). De lagrer kopier fra databasen. De fleste dataelementer er styrt fra denne typen noder.

Replikerte data har to versjoner på de mobile nodene:

- **Master versjon:** inneholder den nyeste verdien mottatt fra objekt *masteren*. Versjoner hos objekt masteren legger grunnlaget for masterversjoner, men det kan også finnes eldre versjoner hos noder som er frakoblet eller hos noder som ikke er blitt oppdatert (lazy replica nodes).
- **Tentativ versjon:** Lokale objekter kan bli oppdatert av tentative transaksjoner. De nyeste verdien forårsaket av en lokale oppdateringer blir behandlet som tentative versjoner.

Modellen skiller også mellom to ulike typer transaksjoner:

- **Base transaksjoner:** utfører bare operasjoner på masterdata,og produserer bare nye masterdata. Minst *en* mobilnode (tilkoblet) er alltid involvert i utførelsene sammen med et sett av basenoder.
- **Tentative transaksjoner:** utfører bare operasjoner på lokale tentative data, og produserer nye tentative versjoner. En tentativ transaksjon fører også til at en base transaksjon må kjøres på et senere tidspunkt på en basenode.

7.5.2 Utførelse av transaksjoner

Tentative transaksjoner kan involvere objekter styrt fra basenoder eller fra den mobile noden hvor transaksjonen ble initiert, og må her følge såkalte definisjonsområderegler (eng. scop rules). Målet er at når den tentative transaksjonen skal bli kjørt som en ekte basetransaksjon, så må den mobile noden ha kontakt med de involverte base nodene. På denne måten kan den mobile noden få rett kopi (masterkopi) fra de involverte nodene og kjøre transaksjonen på riktig måte (innenfor rett definisjonsområde).

Når en base transaksjon generert av en tentativ transaksjon blir kjørt så kan den feile eller produsere ulike resultater. Base transaksjoner har derfor et *akseptanse kritere*, en test om transaksjonens resultat er innenfor gitte rammer. Et eksempel på dette er at saldoen på bankkonto aldri må bli negativ.

Hvis en tentativ transaksjon feiler, så må noden (hvor den stammer fra) og enheten som opprettet transaksjonen få beskjed om dette.

7.5.3 Evaluering av modellen

Frakobling

To-delt replikering er konstruert for å håndtere mobile transaksjoner med hyppige frakoblinger, og har derfor veldig god støtte for nettopp dette. En mobil node kan eksempelvis lese kopier fra andre base noder, for så koble fra men fortsette transkasjonsutførelsen ved å kjøre tentative transaksjoner. Modellen støtter dermed asynkron deling av verdier. Når denne mobile enheten senere kobler seg til fastnettet, må den opprette kontakt med de basenodene som er involvert i transaksjonsutførelsen (kopiert data), og kjøre base transaksjoner for å gjøre effekten av de tentative transaksjonene ekte. Dette forutsetter at transaksjonene ikke bryter med det definerte *akseptanse kriteriet*.

Økt samtidighet

Da mobile noder har mulighet for å lese data for senere gjøre tentative opersjoner på dem, har modellen i utgangspunktet god støtte for at flere noder (mobile og base) kan kjøre transaksjoner

i parallell. Ulempen er uansett når de mobile nodene skal forene sine oppdateringer med noden som har master kopien av dataene. I denne prosessen blir det utvekslet mye informasjon, og i tilfeller hvor mange mobile noder har gjort oppdateringer på samme data, kan utførelsen bli tidskrevende, samtidig som sannsynligheten for *vranglås* blir stor.

Mobilitet

Modellen tar ikke opp dette temaet eksplisitt, så ingen eksakte løsninger er beskrevet for kontroll av mobile transaksjoner. En todelt løsninger med mobile- og base-noder er et uansett et bra utgangspunkt for å konstruere mekanismer for nettopp dette.

7.6 Oppsummering - mobile transaksjonsmodeller

På bakgrunn av vår evaluering vil vi her oppsummere de mobile transaksjonsmodellene vi har tatt for oss. Alle modellene ble vurdert utfra hvor bra de støttet henholdsvis *frakobling*, *økt samtidighet* og *mobilitet*. Følgende graderingsskala er brukt for å oppsummere hvor bra de imøtekommer de valgte evalueringskriteriene:

- **LAV:** modellen har ingen eller liten støtte for dette kriteriet.
- **MEDIUM:** modellen har tilfredsstillende støtte for dette kriteriet.
- **HØY:** modellen har god støtte for dette kriteriet.

Tabell 8.1 oppsummerer modellene vi har vurdert.

Modeller / Kriteria	**Frakobling**	**Økt samtidighet**	**Mobilitet**
Kangaroo	MEDIUM, kontrollert frakobling	LAV, ingen deling av verdier mellom de mobile transaksjonene	HØY, overføring av kontroll ved flytting mellom MSSer
AMDB	MEDIUM, agentbasert	LAV, ingen deling av verdier mellom de mobile transaksjonene	MEDIUM, logisk mobilitet
PRO-MOTION	HØY, støtte for frakoblet modus	LAV, ingen deling av verdier mellom de mobile transaksjonene	LAV, ingen mekanismer beskrevet
Reporting	LAV, synkron kommunikasjon nødvendig	HØY, deling av verdier mellom transaksjoner under utførelse	MEDIUM, tilrettelagt for mobilitet, men ikke beskrevet hvordan
To-lags replikering	HØY, asynkron deling av verdier	MEDIUM, bryter med isolasjons-kravet, benytter tentative transaksjoner	MEDIUM, tilrettelagt for mobilitet, men ikke beskrevet hvordan

Tabell 7.1: Oppsummering av mobile transaksjonsmodeller

Kapittel 8

Eksport-import transaksjonsmodellen

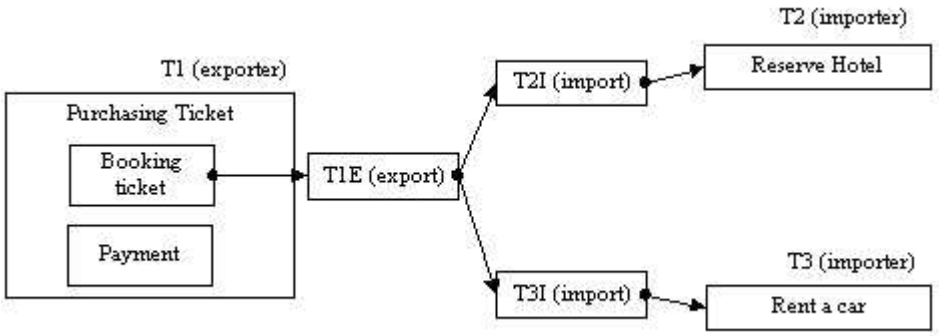
I kapittel 7 ble det presentert en rekke modeller og mekanismer som er spesielt utviklet for å møte kravene til mobile transaksjoner. Imidlertid er det ingen av disse modellene som dekker alle de tre hovedutfordringene mobilitet, frakobling og økt samtidighet på en god måte. I dette kapittelet vil det bli presentert en transaksjonsmodell, *eksport- import modellen*, som tar høyde for å støtte asynkron deling av midlertidige resultater for mobile transaksjoner. Modellen har også støtte for mobilitet og hyppige avbrudd blant transaksjoner. Denne modellen er et arbeid under utvikling i samarbeid med oss og veilederne våre på diplomoppgaven. Det er spesielt Hien Nam Le som har stått for det arbeidet som blir gjennomgått i dette kapittelet.

Oppbygningen av modellen vil her bli presentert, samtidig som det vil bli undersøkt hvordan modellen støtter deling av midlertidige resultater for transaksjoner. Vi starter med et eksempel på et scenario hvor en mobil bruker er avhengig av å utføre flere transaksjoner for å oppnå sitt mål. På bakgrunn av dette eksempelet vil de grunnleggende elementene i modellen bli undersøkt.

8.1 Scenario: Bestilling av flybillett

En mobil bruker har planer om å bestille reise til et sted for å delta på en konferanse. Først og fremst er vedkommende avhengig av å *bestille flybillett* (T_1) til destinasjonsstedet. Foruten om dette så ønsker han også å *leie bil* (T_2) og *reservere hotellrom* (T_3) for oppholdet. Vi har her tre transaksjoner som er avhengig av å bli utført for at turen skal gå i boks, hvor henholdsvis T_2 og T_3 er avhengige av transaksjon av T_1. Med dette går det frem at så sant det ikke er mulig å bestille flybillett så er det ikke aktuelt å bestille eksempelvis leiebil for det planlagte oppholdet, noe som medfører at T_2 og T_3 ikke kan starte å kjøre før T_1 har gjort kjent *ankomsttiden* for flyet.

Kritisk verdi i dette er eksempelet er altså *ankomsttiden*, som på en eller annen måte bør deles med de andre transaksjonene. I henhold til navnet på eksport- import modellen, så har transaksjoner mulighet for å gjøre verdier kjent (dele) med andre transaksjoner ved å initiere en *eksporttransaksjon*, og likedan få tilgang til disse verdiene ved å starte en *importtransaksjon*. Som illustrert i figur 8.2 er det her T_1 (eksportør) som starter en eksport transaksjon T_1E. T_2



Figur 8.1: Eksport-import transaksjoner, fra oppgavetekst fra Hien Nam Le

og T_3 (importører) får tilgang på kritisk verdi ved å initierere importtransaksjonene T_2I og T_3I. Modellen har dermed to deler: en eksport-import transaksjonskomponent, og et eksport-import delingsrom. Sistnevnte angir hvor selve utvekslingen av data mellom transaksjoner som er avhengige av hverandre foregår.

8.2 Transaksjonstyper

Eksport-import modellen består av tre transaksjonstyper: globale transaksjoner, subtransaksjoner og kompenserende transaksjoner. Den overordnede distribuerte transaksjonen som er gjenstand for global commit eller abort er den globle transaksjonen. Hver globale transaksjon kan ha et vilkårlig antall subtransaksjoner under seg. Disse subtransaksjonene utføres mot databaser som befinner seg på enten mobile eller stasjonære enheter. Hvis regelsettet tillater det, kan hver subtransaksjon committe eller abortere på egenhånd før den globale transaksjonen har fullført. En eventuell abort av den globale transaksjonen fører til at alle subtransaksjoner som har utført commit må omgjøres av kompenserende transaksjoner.

8.3 Det mobile miljøet

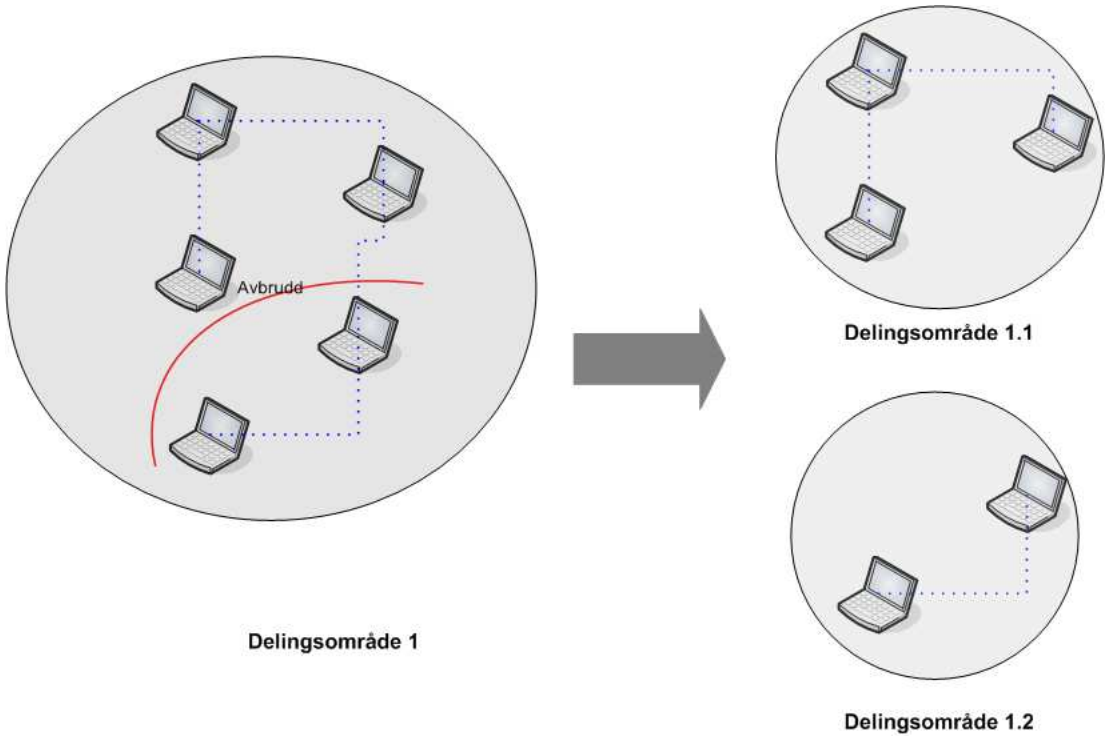
Modellen opererer i et miljø som tilsvarer det vi beskrev i kapittel 5 der man har tre hovedkomponenter: mobile enheter (MHs), mobile støttestasjoner (MSSs) og databasetjenere (DBs). De mobile støttestasjonene har som oppgave å videreformidle spørringer, transaksjonsstatus og resultater mellom mobile enheter og databasetjenere. En *tilslutning* (eng: affiliation) er en dynamisk gruppe av mobile og ikke-mobile enheter som ønsker å danne en midlertidig arbeidsgruppe for å utveksle informasjon og støtte hverandres operasjoner. Det vil si at en enhet i en tilslutning må kunne koble seg opp mot andre enheter i denne tilslutningen. En *mobil tilslutning* er en tilslutning der alle enhetene er mobile enheter. En mobil enhet vil bli fjernet om den mister tilkoblingen til de andre enhetene i tilslutningen den er med i. Dette kan skje hvis den mobile enheten for eksempel går tom for batteri eller forflytter seg utenfor rekkevidden til den mobile tilslutningen.

8.4 Eksport-import delingsrommet

Som illustrert i figur 8.2, angir dette hvor selve informasjonsutvekslingen mellom importerende og eksporterende transaksjoner finner sted. Dette oppbevaringsstedet er på ingen måter et fast definert område, opprettet ved et bestemt tidspunkt, men skapes heller etter behov, og er bare tilgjengelig for de involverte transaksjonene. Når en transaksjon (eksportør) ønsker å dele data, tar den initiativ til å opprette dette oppbevaringsstedet. Transaksjoner som ønsker å delta i delingsprosessen kan så bli medlem av dette stedet. Informasjonsutvekslingen kan først finne sted når eksportøren har opprettet en eksporttransaksjon. Siden et delingsområde bare er tilgjengelig for de involverte transaksjonene, er det fullt mulig for en transaksjon å operere med flere eksport-import delingsrom på en gang. Som allerede vist, er et import-eksport delingsrom en *dynamisk struktur* som opprettes og fjernes etter behov.

Bruk av eksport-import delingsrom gir god mulighet for utstrakt informasjonsutveksling mellom databasetjenere og klienter. Eksempelvis kan en mobil vert importere informasjon fra en gruppe, og eksportere den til en annen.

Som kjent kan mobile tjenere og klienter rammes av hyppige frakoblinger, og dermed bli ute av stand til å nå sitt delingsrom. I grupper av mobile enheter som kommuniserer med hverandre (uavhengig av fastnett) kan medlemmer fra tid til annen miste sin forbindelse. Om disse enhetene har et felles delingsrom, tillater eksport-import modellen å splitte dette i mindre delingsrom når noen eventuelt ufrivillig kobler fra. På denne måten kan de to nye gruppene innad fortsette å dele informasjon.



Figur 8.2: Splitting av delingsområde for mobile grupper

Når mobile verter etter en avkobling igjen kan slutte seg til delingsområdet, må det finnes mekanismer som kan garantere konsistens for data som befinner seg i delingsområdet og på den

mobile verten. Import-eksport modellen tar sikte på at data fra de involverte enhetene flettes sammen ved tilkobling.

Delingsrommet kan også flytte seg blant enhetene i den mobile tilslutningen. Slik kan modellen håndtere problemet med begrenset kapasitet på mobile enheter. For eksempel kan en mobil enhet som snart er i ferd med å gå tom for batteri gi beskjed om dette, og en annen mobil enhet vil deretter ta over delingsrommet.

8.5 Eksport- og importtransaksjoner

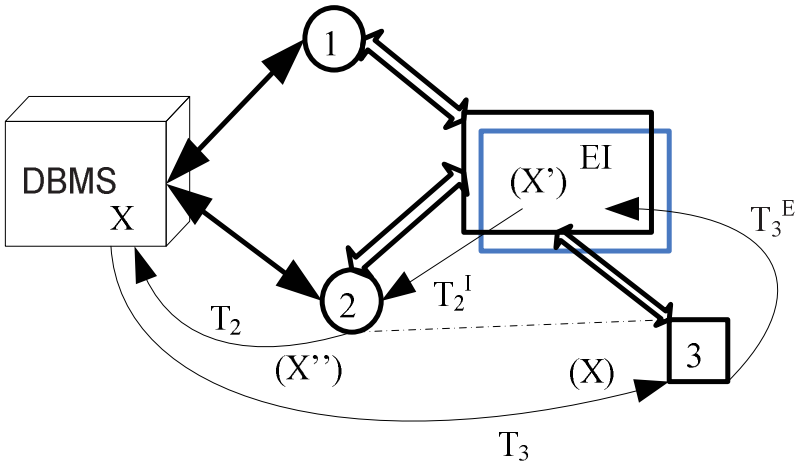
Eksporterende transaksjoner kan dele verdier med andre transaksjoner ved å initiere *eksport-transaksjoner*. Strukturen på disse transaksjonene er avhengig av måten den eksporterende transaksjonen ønsker å dele sine resultater på. Om den deler alle sine verdier i en omgang, så vil eksport transaksjonen ha en flat form. Om den deler resultatene i flere omganger (stegvis), så vil eksporttransaksjonen få en åpen nøstet struktur.

En importerende transaksjon må initiere en *importtransaksjon* for å få tilgang på verdiene som en eksporttransaksjon har delt. Strukturen på denne er avhengig av strukturen på eksporttransaksjonene og måten den importerende transaksjonen ønsker å importere resultatet på. Både eksporterende og importerende transaksjoner har mulighet til å starte en importtransaksjon.

En illustrasjon på denne utvekslingen er vist i figur 8.3. Her hentes først verdien X ut av databasen og transaksjon T_3 skaffer seg skrivelåsen. Verdien X oppdateres til X' av denne transaksjonen og blir midlertidig lagret på den mobile enheten MH_3. Når den mobile enheten MH_3 blir med i den mobile tilslutningen MA, kan den dele den nye verdien X' til eksport-import delingsrommet EI via eksporttransaksjonen T_3E. Den mobile enheten MH_2 som kobler seg til den mobile tilslutningen skaffer seg dataverdien X' via importtransaksjonen T_2I, og den oppdaterer X' til verdien X'' lokalt før den integrerer denne verdien med databasen via transaksjon T_2. Interaksjonen mellom eksporttransaksjon T_3E og importtransaksjon T_2I kan være enten synkron eller asynkron. Hvis begge de mobile enhetene MH_2 og MH_3 kobler seg til den mobile tilslutningen på samme tidspunkt, vil interaksjonen være synkron, hvis ikke vil den være asynkron.

8.6 Konsistens i eksport-import modellen

Det er to viktige tilfeller av informasjonsdeling blant de mobile enhetene i en tilslutning: *del-for-lesing* og *del-for-skriving*. For del-for-lesing er en transaksjons delte verdi lesbar for andre transaksjoner, men den er ikke skrivbar. Den originale transaksjonen har fremdeles skrivelåsen til den delte verdien, og den mobile enheten har fremdeles ansvar for å committe verdien til databasetjeneren. Ved del-for-skriving skriver transaksjonen den delte verdien til delingsrommet og slipper så skrivelåsen til verdien. Andre transaksjoner kan så både lese og skrive verdien som ligger i delingsrommet.



Figur 8.3: Deling av dataverdier med eksport- og importtransaksjoner, fra [32]

8.7 Sammenligning med andre transaksjonsmodeller

I denne delen vil vi sammenligne eksport-import modellen med de fem andre mobile transaksjonsmodellene vi beskrev i kapittel 7. Eksport-import modellen vil bli sammenlignet på grunnlag av dens evne til å løse problemstillingene mobilitet, frakobling og økt samtidighet som beskrevet i kapittel 6.

8.7.1 Frakobling

Blant de sterkeste kandidatene i denne kategorien er PRO-MOTION, som tillater mobile enheter å fortsette arbeidet sitt i tilfelle frakobling ved hjelp av sitt *compact* konsept. De mobile enhetene kan fortsette å manipulere dataene ved hjelp av metoder i compact objektet som overholder kravene til konsistens og korrekthet. En annen interessant kandidat er to-delt replikering som har svært mange likheter med import-eksport modellen.

Eksport-import modellen nærmer seg problemet på en litt annen måte og sørger heller for at den distribuerte transaksjonen ikke blir avbrutt selv om en mobil enhet mister tilkoblingen sin midlertidig. Delte midlertidige verdier, resultater fra spørringer og transaksjonsstatus ligger i delingsrom eller på den mobile støttestasjonen, og en mobil klient eller en mobil databasetjener som mister tilkoblingen kan ta kontakt når den får forbindelsen tilbake og bli oppdatert på hva som har skjedd med transaksjonen. Den asynkrone naturen til eksport-import modellen gjør også at globale transaksjoner kan velge å committe eller abortere utifra et gitt regelsett selv om den ikke får kontakt med enkelte subtransaksjoner som er under utførelse på mobile enheter.

Konseptuelt så er eksport-import modellen svært lik to-delt replikering 7.5 når det gjelder frakobling. Begge modellene støtter asynkron kommunikasjon mellom de mobile enhetene og de faste nodene. I to-delt replikering har de mobile enhetene to versjoner av data, både en masterversjon (nyeste) og en tentativ versjon. Om den mobile enheten kobles fra så kan den fortsette utførelsen ved å kjøre tentative transaksjoner. Disse må så forenes med noden som har ansvar for dataene den har kopiert fra ved tilkobling. I eksport-import modellen vedlikeholdes det ikke to versjoner av dataene på den mobile enheten. Ved eventuelle endringer på dataene i delings-

rommet så må transaksjoner som har importert disse verdiene få beskjed om endringene.

8.7.2 Økt samtidighet

Reporting modellen har støtte for deling av verdier mellom transaksjoner under utførelse, og den oppnår dermed en høy grad av samtidighet. Siden delegering benyttes for å utveksle verdier, forutsetter dette synkron kommunikasjon mellom enhetene som kjører transaksjonene.

Eksport-import modellen gjør bruk av delingsrom som metode for datautveksling. Slik oppnås en asynkron protokoll som ikke forutsetter at alle parter må være tilgjengelig samtidig. Korte avbrudd i tilkoblingen vil dermed bli maskert.

8.7.3 Mobilitet

Kangaroo modellen er skreddersydd for å takle mobilitet ved at de mobile transaksjonene migrerer fra MSS til MSS ettersom den mobile enheten som initierte transaksjonen forflytter seg fra celle til celle.

I eksport-import modellen kan delingsrommet som en mobil transaksjon har opprettet på en mobil enhet flytte seg til andre mobile enheter hvis det trengs. Dette medfører at den mobile tilslutningen kan fortsette som før hvis den enheten delingsrommet tilhørte originalt forflytter seg til et annet sted utenfor rekkevidde. Når den mobile enheten blir medlem av tilslutningen igjen kan den ta igjen eventuelle resultater som har blitt oppnådd i mellomtiden. Siden en tilslutning kan bestå av flere andre tilslutninger kan man tenke seg en situasjon der en mobil enhet forflytter seg fra en celle til en annen celle og likevel har mulighet til å få kontakt med en originale tilslutningen.

8.8 Oppsummering - mobile transaksjonsmodeller

På bakgrunn av vår evaluering vil vi her oppsummere de mobile transaksjonsmodellene vi har tatt for oss, samtidig som disse blir sammenlignet med eksport-import modellen. Sistnevnete modell vil i likhet med de første, som ble beskrevet i kapittel 7 om mobile transaksjonsmodeller, bli vurdert utfra hvor bra de støttet henholdsvis *frakobling*, *økt samtidighet* og *mobilitet*.

Tabell 8.1 sammenligner eksport-import modellen med andre mobile transaksjonsmodeller.

Modeller / Kriteria	**Frakobling**	**Økt samtidighet**	**Mobilitet**
Kangaroo	MEDIUM, kontrollert frakobling	LAV, ingen deling av verdier mellom de mobile transaksjonene	HØY, overføring av kontroll ved flytting mellom MSSer
AMDB	MEDIUM, agentbasert	LAV, ingen deling av verdier mellom de mobile transaksjonene	MEDIUM, logisk mobilitet
PRO-MOTION	HØY, støtte for frakoblet modus	LAV, ingen deling av verdier mellom de mobile transaksjonene	LAV, ingen mekanismer beskrevet
Reporting	LAV, synkron kommunikasjon nødvendig	HØY, deling av verdier mellom transaksjoner under utførelse	MEDIUM, tilrettelagt for mobilitet, men ikke beskrevet hvordan
To-lags replikering	HØY, asynkron deling av verdier	MEDIUM, bryter med isolasjonskravet, benytter tentative transaksjoner	MEDIUM, tilrettelagt for mobilitet, men ikke beskrevet hvordan
Eksport-import	HØY, De delte verdiene er alltid tilgjengelig via delingsrom	HØY Asynkron deling av verdier via eksport-import rom.	MEDIUM Dynamisk delingsrom, kan flyttes mellom MSSer samt splittes og forenes i større og mindre deler.

Tabell 8.1: Eksport-importmodell vs. andre mobile transaksjonsmodeller

Kapittel 9

Teknologiplattformer

Dette kapittelet omhandler ulike teknologier som kan brukes til å implementere eksport-import modellen. En mulig løsning er selvsagt å bygge et system helt fra bunn ved å ta i bruk programmerinsspråk som eksempelvis C++ eller Java. Vi har valgt å ta i bruk mer spesialiserte teknologier i vår oppgave, da vi hovedsakelig vil spare masse tid på dette, og dermed kan konsentrere oss mer om å implementere selve modellen.

Meget sentralt i import-eksportmodellen er kravet om asynkron kommunikasjon mellom de mobile transaksjonene. Fordelen med dette er at det forenklet prosessen med å velge ut teknologier, da svært mange primært støtter synkron kommunikasjon. På bakgrunn av dette kravet fremstår Sun Microsystems implementasjon av *tuppelrom* (JavaSpaces), samt IBMs rammeverk for *agenter* (Aglets), som gode kandidater. For begge teknologiene ble først generelle konsepter beskrevet, før man så nærmere på de enkelte implementasjonene. Videre ble det undersøkt hvordan henholdsvis JavaSpaces og Aglets kunne brukes til å implementere eksport-import modellen. Til slutt ble det konkludert med hvilken teknologi som var best egnet til å løse de utfordringene man stod ovenfor i implementasjonen av eksport-import modellen.

9.1 Tuppelrom

Tuppelrom (eng: tuple spaces) er en egen måte å tenke på når det gjelder distribuerte systemer. I tradisjonelle distribuerte systemer baserer man seg på meldingsutveksling mellom prosesser eller kall av metoder på distribuerte objekter. Det spesielle med tuppelrom er at man lar de distribuerte prosessene bli koordinert gjennom utveksling av objekter i et eller flere *rom*. Denne tankegangen har sin opprinnelse i arbeidet gjort av Dr. David Gelernter ved Yale University der han konstruerte programmeringsspråket Linda [19].

9.1.1 Generelle prinsipper

Et tuppelrom er et oppbevaringssted for objekter som er tilgjengelig over et nettverk. Prosessene bruker rommet som et permanent objektlager og som en måte å utveksle objekter på. Ved å

designe distribuerte datastrukturer og distribuerte protokoller som bruker disse datastrukturene kan man koordinere prosessene på en fleksibel måte. For eksempel kan en ordnet liste med elementer representeres med et sett av objekter, der hvert objekt lagrer verdien og posisjonen til et element i lista. Dermed kan flere prosesser aksessere hvert sitt objekt samtidig uten å stå i kø.

Fordelen med distribuerte protokoller basert på tuppelrom er at prosessene er løst sammenkoblet. Prosessene samhandler indirekte via et rom, i motsetning til det tradisjonelle konseptet der man kommuniserer direkte. Sendere og mottakere av meldinger trenger ikke å vite hverandres identiteter på forhånd, og de trenger heller ikke å være aktive innenfor det samme tidsrommet. Ved å få en løs kobling mellom sender og mottaker kan man lett lage distribuerte protokoller som er fleksible og pålitelige.

Det finnes flere implementasjoner av tuppelrom: JavaSpaces fra Sun Microsystems, TSpaces fra IBM, GigaSpaces, LightTS og mange andre. De fleste baserer seg på programmeringsspråket Java, siden Java er plattformuavhengig og fungerer godt i nettverkssammenhenger. I det neste avsnittet vil JavaSpaces bli presentert.

9.1.2 JavaSpaces - en implementasjon av tuppelrom

JavaSpaces er Suns implementasjon av tuppelrom, og det første man legger merke til er at programmeringsgrensesnittet er påfallende enkelt. Operasjonene *read, write, take og notify* er alt man har tilgjengelig. Figur 9.1 viser flere prosesser som koordineres via JavaSpaces. Grunnen til at disse operasjonene er alt man trenger er at selve tuppelrommet er et såpass kraftig konsept med mange nyttige egenskaper:

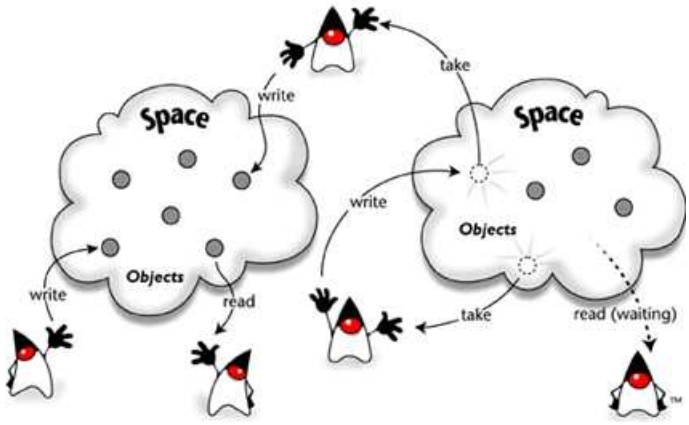
- **Delt minne** - Rommene er delt mellom mange prosesser som kan bruke dem samtidig. Detaljene som er nødvendige for å oppnå dette tas hånd om av hvert enkelt tuppelrom.

- **Permanent lagring** - Hvert tuppelrom er et pålitelig permanent lager for objekter. Siden lageret er permanent er det mulig for prosesser å kommunisere selv om de ikke er aktive samtidig, og meldinger lagret som objekter i rommet kan leveres på et senere tidspunkt.

- **Assossiativt oppslag** - Når man skal finne et objekt lager man en mal for objektet der man fyller ut flere felter man krever at objektet må matche. Identifikatorer som minneadresse, nettverksadresse eller unike navn er ikke nødvendig for å hente ut objekter.

- **Transaksjonssikkerhet** - Tuppelrommene støtter en transaksjonsmodell der man kan definere at flere operasjoner på et eller flere rom tilsammen skal utgjøre en atomisk transaksjon.

- **Utveksling av kjørbar kode** - Objektene man utveksler trenger ikke være kun passive data, det kan godt være kjørbar kode.



Figur 9.1: Prosesser som bruker JavaSpaces for koordinering

9.1.3 JavaSpaces og import-eksport modellen

Delingsområdene til import-eksport transaksjonsmodellen er svært like tuppelrom i prinsippet, og det burde være enkelt å implementere disse delingsområdene ved hjelp av JavaSpaces siden mye allerede vil være lagt til rette. Transaksjonene kan utveksle dataverdier mellom hverandre på en asynkron og løst sammenkoblet måte som passer godt i forhold til modellen. Permanent lagring og transaksjonssikkerhet gjør at det blir lettere å feilsikre systemet mot eventuelle feil i datakommunikasjonen eller i maskinvaren.

Ulempen med JavaSpaces er at enhver instans av et JavaSpace krever en del ressurser for å kjøre, og oppstartstiden er forholdsvis lang. Det kan derfor bli problematisk å implementere den delen som dreier seg om dynamisk opprettelse av delingsrommene, samt splitting og fletting av delingsrommene. På prototypstadiet ser imidlertid teknologien ut til å være meget godt egnet, siden man sparer mye utviklingstid i bytte mot en viss grad av ytelse.

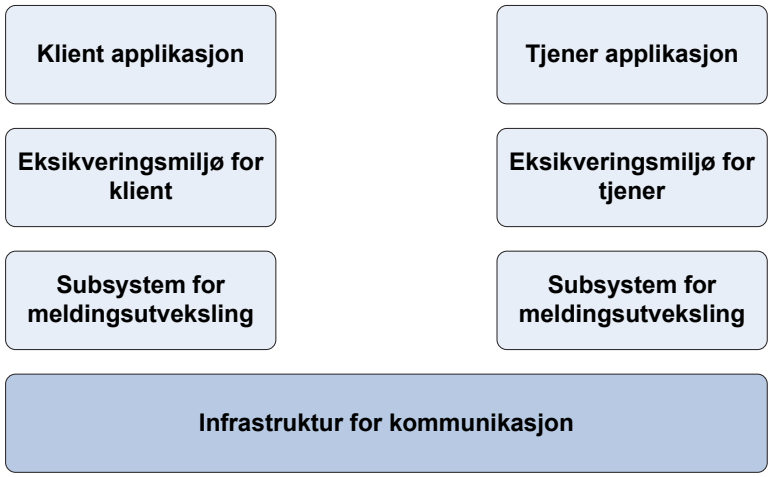
9.2 Agent teknologi

Agent teknologi er en velkjent og mye brukt teknologi innenfor distribuerte programvareløsninger. Den er svært bra egnet i distribuerte systemer hvor det er stort behov for samarbeid mellom entiteter i miljøer som er i stadig forandring. Når det gjelder selve definisjonen av en agent, så er dette et meget omdiskutert tema. Det finnes ingen felles definisjon, men allikevel er det rimelig greit å beskrive de grunnleggende egenskapene til en agent. En agent en selvstendig programvare som har som ansvar å å utføre bestemte operasjoner for andre personer. Foruten om selve oppgaven den er pålagt, så er det også satt bestemte begrensninger for hvordan den får lov til å operere og utføre denne. Franklin og Graesser [20] skiller mellom tre ulike typer agenter:

- **Målorienterte agenter:** agenter som ikke bare opererer i forhold til respons fra miljø.
- **Kommunikative agenter:** agenter som kan kommunisere med hverandre.
- **Mobile agenter:** agenter som kan transportere seg selv fra en host til en annen.

Målet med dette kapittelet er å gi en grundig beskrivelse av hva som kjennetegner en agent, spesielt med vekt på *mobile agenter*. For at en agent skal kunne eksistere i et system så er den avhengig av et rammeverk hvor den kan implementeres og kjøre, kjent som et *agentmiljø*. Det finnes i dag en rekke ulike rammeverk for agenter, hvor henholdsvis IBM‘s Aglets vil bli grundigere studert i denne oppgaven. Først av alt vil selve arkitekturen bli presentert. Videre vil det bli undersøkt hvordan *Aglets* kan brukes for å løse de problemene man står ovenfor i implementasjonen av import-eksport modellen.

9.2.1 Generelle prinsippper



Figur 9.2: Konseptuell modell for mobile agenter

Vår beskrivelse av agenter[7] vil ta utgangspunkt i figur 9.2, som her illustrerer en konseptuell modell over mobile agenter. Det øverste laget viser til klientapplikasjonen, som kjører i et

bestemt applikasjonsmiljø, eksmpelvis OS/2 eller Windows. Disse applikasjonene samhandler med en tjenerapplikasjon på en tjener via bruk av agenter, og kan eksempelvis være programmer for informasjons søk og gjenfinning, transaksjonsutføring eller epostklienter. Under laget for klient- og tjener-applikasjonen finnes eksiskveringsmiljøet for mobile agenter, som selve programvaren er bundet til. Dette laget tar seg av selve utførelsen av agent applikasjonen på bakgrunn av input fra appliksjonene i laget over. Agent eksikveringsmiljøet er avhengig av å benytte seg av en transportprotokoll for å kunne utføre meldingsutveksling over et nettverk.

I en transaksjonsutførelse, vil eksempelvis klient applikasjonen sende en spørring (via APIen) til agent eksikveringsmiljøet, som da initierer en agentprosess som kan håndtere denne typen oppgave. Agenten vil her levere forespørselen til riktig tjener og returnere resultatet av spørringen til klientapplikasjonen. Hvilken tjener agenten her skal besøke kan være adressert eksplisitt, eller agenten kan på egenhånd finne en som kan tilby de ønskede tjenestene. I noen tilfeller kan det være behov for at agenten blir værende på tjeneren, for å uføre bestemte oppgaver for klientapplikasjonen. Dette kan eksempelvis være å returnere bestemte resultater . I eksempelet med spørringen så kan det hende at den er avhengig av resultater fra flere tjenere. Agenten kan da ta turen innom de involverte tjenerene for så levere det endelige resultatet til klientapplikasjonen.

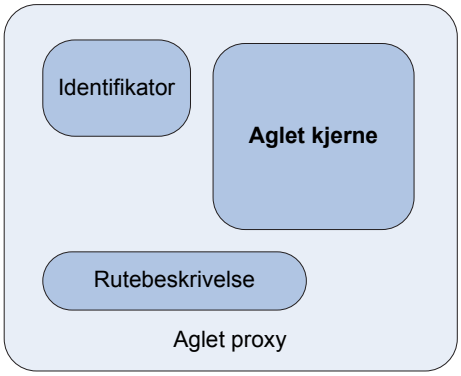
9.2.2 Aglets - et rammeverk for å implementere agenter

Aglets [10] er et rammeverk for å implementere og kjøre agenter, utviklet av IBM. Oppbygningen til en Aglet er presentert i figur 9.3. En *aglet* utgjør selve agenten (programvaren), som er i stand til å migrere fra tjener til tjener i et nettverk. En aglet kan primært deles i to distinkte deler, nemlig en kjerne og en proxy. Kjernen inneholder alle interne variabler og metoder og fungerer som et grensesnitt mellom agenten og dens omgivelser. Proxyen virker som et beskyttende skall om kjernen, og har som oppgave å hindre andre fra direkte aksess til lokale variabler og metoder samt å skjule agletens virkelige lokasjon fra andre agleter med ødeleggende hensikter. Foruten om dette, så har en aglet en unik *identifikator* samt en *rutebeskrivelse* som inneholder en eventuell rute som den må følge (hvilke hoster den må innom).

Aglets fremstår kanskje som en av de sterkeste kandidatene innenfor rammeverk for agenter. Foruten om at Aglets bruker programmeringsspråket Java så ansees de for å ha gode sikkerhets løsninger. Når det gjelder sikkerhet så har dette vært et veldig omstridt tema når det gjelder agentteknologi. Da en agent faktisk er en programvare som kan kjøre på en tjener, så ligger forholdene til rette for at dette kan misbrukes.

Aglets har en rekke egenskaper som har vist seg å være svært nyttige for distribuerte applikasjoner. Disse vil bli presentert under, før man tar stilling til hvordan Aglets kan brukes til å implementere import-eksport modellen.

- **Tjener fleksibilitet:** Et stort problem med bruk av agenter er at de i utgangspunktet er i stand til å endre oppførselen til en tjener uten direkte tillatelse fra 'eier'. I motsetning til de fleste andre mobile agenter så har agleter ikke mulighet å migrere til en en hvilken som helst host, men er avhengig av tjeneren den skal besøke er kompatibel for agleter. På denne måten opererer aglets kun innenfor sikre omgivelser og er dermed bra rustet mot angrep fra agenter med ødeleggende hensikter.



Figur 9.3: Struktur - Aglet

- **Lokal interaksjon:** For å redusere belastningen på en mobil agent så har den mulighet til å gi selve oppgaven til en lokal tjener, og dermed slippe å bruke egne ressurser, for så å bare konsentrer seg om å levere resultatene. Dette er svært nyttig i tilfeller hvor forespørsler er avhengig av andre forespørsler. På denne måten kan agenten reagere med en gang på forespørsler siden den ikke er avhengig av å vente på lange dataoverføringer. Agenter kommuniserer seg i mellom ved utveklse meldinger. Selve utvekslingsprossen kan være *asynkron* eller *synkron*, og dermed tillate agleter å samarbeide og utveksle informasjon på en løs eller tett integrert måte.

- **Redusert belastning på nettverk:**

  Dette er et omdiskutert tema når det gjelder agenter. På den ene siden kan man argumentere med at siden agenten utfører oppgavene sine lokalt på verten, vil dette føre til redusert trafikk over nettverket i forhold til en ren klient-tjener arkitektur. Forutsetningen er at agenten har nok logikk og informasjon til å behandle og svare på de midlertidige resultatene fra verten. På den andre siden kan man si at selve overføringen av agenten fra en vert til en annen er såpass ressurskrevende at en eventuell innsparing i senere meldingsutvekslinger vil bli druknet.

- **Autonomitet:** Agenter kan konstrueres slik at de selv er i stand til å ta egne avgjørelser på bagrunn av omgivelsene de opererer innenfor. De er dermed i stand til å kunne fortsette sin oppgaveutførelse til tross for at verten som opprettet agenten er utilgjengelig, frakoblet fra nettverket eller ute av drift.

  Det er mulig å sette opp mål for agleter ved å definere en rutebeskrivelse. Dette objektet inneholder agletens reiseplan, som inneholder informasjon om hvilke verter den skal besøke, samt hvilke operasjoner som skal utføres her. En aglet er dermed uavhengig av sine omgivelser, og opererer på bakgrunn av reisebeskrivelsen, og er dermed autonom.

9.2.3 Aglets og import-eksport modellen

Aglets har vist seg å ha egenskaper som gjør den til en god kandidat for bruk i implementasjonen av eksport-import modellen. Først og fremst stiller Aglets veldig sterkt når det gjelder mobilitet. Som beskrevet i kapittel 8 om eksport-import modellen så er det viktig for et delingsrom å være dynamisk og mobilt slik at det kan rekonstrueres, flyttes til andre tjenere, splittes i mindre deler

eller flettes sammen med andre delingsrom. Imidlertidig vil det være et problem å opprettholde den dynamiske egenskapen til delingsrom og mobile støttestasjoner siden Aglets ikke har en fullgod discovery protokoll for å finne maskiner som kjører agentprogramvaren. Adressen til disse maskinene må derfor oppgis statisk, og den dynamiske egenskapen vil forsvinne.

Et annet viktig aspekt i eksport-import modellen er asynkron kommunikasjon mellom transaksjoner, slik at mobile enheter kan gjenoppta arbeid etter en eventuell frakobling. Aglets har god støtte for både asynkron og synkron kommunikasjon seg i mellom.

En stor ulemepe med Aglets er at teknologien fortsatt er ganske umoden, og selve videreutviklingen av den har blitt avsluttet av IBM og heller overlatt til det åpne kildekodemiljøet.

9.3 Oppsummering og valg av teknologiplattform

Vi har valgt å bruke JavaSpaces videre i design og implementasjon av prototypen til eksport-import modellen. Nedenfor har vi listet opp de punktene som var viktige for vår avgjørelse:

- **Videreutvikling:** JavaSpaces videreutvikles stadig av Sun og tredjeparts firmaer som GigaSpaces, og det er i ferd med å bli en anerkjent standard innen grid applikasjoner og SOA (Service Oriented Architecture). IBM har sluttet å lage nye versjoner av Aglets rammeverket, og dette er nå overlatt til frivillige.
- **Discovery:** Automatisk oppdagelse av maskiner i det mobile miljøet man kan interaktere med er viktig for å få et dynamisk og feilsikkert system. JavaSpaces bruker den underliggende JINI teknologien og multicast protokoller for å få til dette. Aglets har veldig umoden støtte for dette, og man må basere seg på å vite maskinnavnene til de tjenerne man vil ha tak i på forhånd.
- **Ytelse:** Aglets programvaren som innholder det som er nødvendig for et eksekveringsmiljø for agentene er mye mindre ressurskrevende enn en instans av et JavaSpace, men ulempen er at det må installeres på alle maskiner som ønsker å være i stand til å ta i mot agenter. JavaSpaces implementasjonen fra Sun baserer seg på at klientene laster ned en proxy fra en HTTP server, men alt dette skjer i bakgrunnen uten at programmereren trenger å bekymre seg for det.

Del III

Kravspesifikasjon

Innledning til dokumentet

Denne delen beskriver kravspesifikasjonen som er utledet på grunnlag av oppgaveteksten, våre samtaler med veileder og den foreløpige spesifikasjonen av eksport-import modellen. Først presenteres et scenario der systemet brukes av et reiseselskap. Deretter følger de spesifikke funksjonelle kravene. Til slutt oppsummeres kravene i en tabell med prioritet og kompleksitet.

- Kapittel 10, Motiverende scenario, beskriver et reiseselskap som bruker import-eksport modellen som grunnlag i sitt databasesystem.
- Kapittel 11, Funksjonelle krav, gir en overordnet inndeling av systemet og lister opp krav for hver del.

Kapittel 10

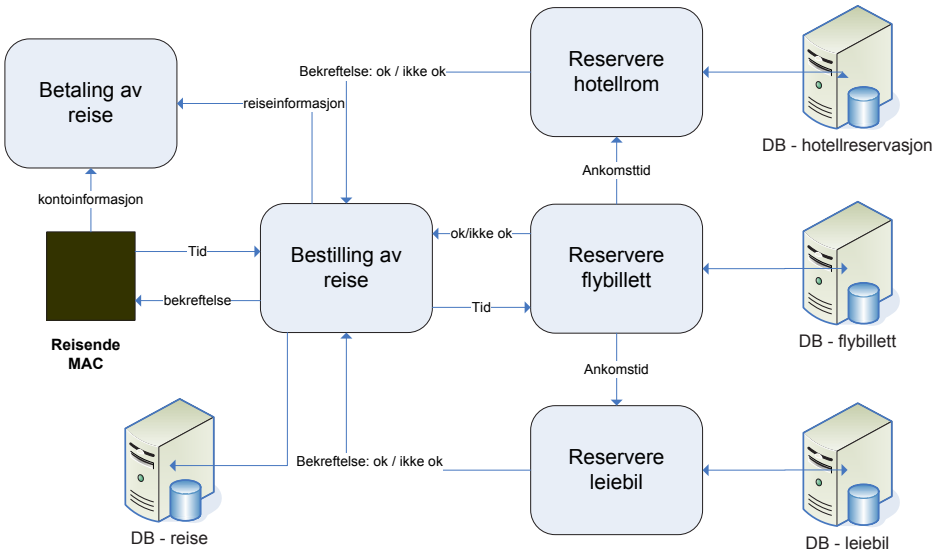
Motiverende scenario

I dette kapittelet skal vi beskrive et case scenario som viser hvordan databasesystemet kan brukes av et stort internasjonalt reiseselskap. Vi vil vise den overordede dataflyten i scenariet, samt beskrive eksempler på hvilke transaksjoner som det er sannsynlig at reiseselskapet vil utføre. Årsaken til at vi velger å beskrive et scenario er at vi ønsker å få et bedre grunnlag for å skrive kravspesifikasjonen, og at vi ønsker å bruke scenariet til å lage tester av systemet vårt når det er implementert.

10.1 Reiseselskapet Ridderreiser

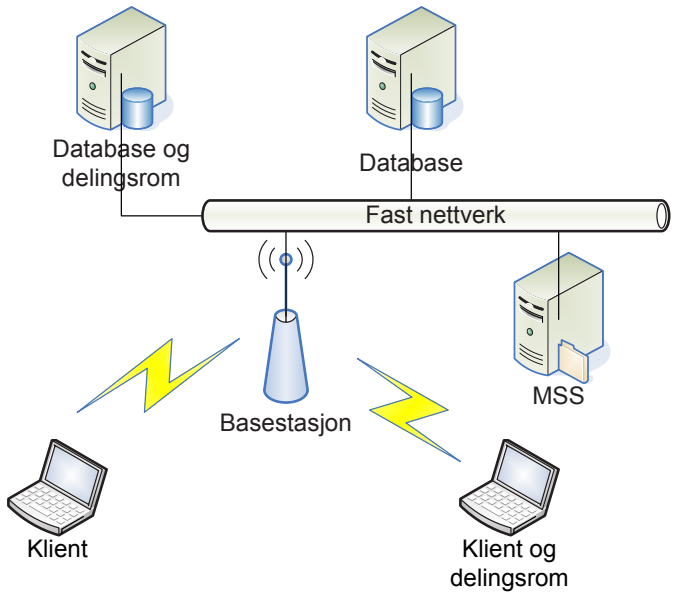
Ridderreiser er et stort multinasjonalt konsern som består av et flyselskap, en hotellkjede og et bilutleiefirma. For å kunne konkurrere i dagens marked ønsker Ridderreiser å integrere sine databaser på en måte som sikrer god feilhåndtering samt høy grad av ytelse og skalerbarhet i den nye mobile dataverdenen. Ved å bruke import-eksport transaksjonsmodellen håper de å oppnå en situasjon der kundene kan bruke en felles webportal for å bestille flyreise, hotellrom og leiebil. De ansatte i Ridderreiser skal også kunne bruke databasesystemet for å hente ut forskjellige typer statistikk, justere priser og mye annet knyttet til markedsføring, logistikk, salg og administrasjon. Webportalen skal fungere godt på både vanlige stasjonære maskiner og forskjellige typer bærbare enheter som for eksempel avanserte mobiltelefoner og bærbare PCer med trådløs forbindelse.

Figur 10.1 gir en overordnet oversikt over de viktigste komponentene for et scenario hvor *reisende Mac* ønsker å bestille seg en reise. Først og fremst er han avhengig av å skaffe seg flybillett for ønsket reisetidspunkt, men han ønsker også å bestille hotell og leiebil for oppholdet. Sistnevnte bestillinger er naturligvis avhengig av når han ankommer reisemålet.



Figur 10.1: Oversikt over prosesser i reisescenariet

I sine reiser bruker Mac ofte en bærbar PC koblet til en mobiltelefon som kommuniserer med databasesystemet via en trådløs dataforbindelse. Databasetjenerne, basestasjonene, de mobile støttestasjonene og delingsrommene har gjerne raske og stabile kommunikasjonslinjer som knytter dem sammen i et fast nettverk. Når Mac skal utføre en bestilling hos Ridderreiser, vil spørringen hans bli overført via den trådløse forbindelsen til basestasjonen og videre til en mobil støttestasjon. I det øyeblikket spørringen når den mobile støttestasjonen vil de databasetjenerne som har registrert sin interesse hos MSSen få vite om dette, og de vil hente ut de delene av spørringen som er relevant for dem. Hvis det eksisterer avhengigheter mellom subtransaksjonene slik at de kan dele verdier seg imellom mens de utføres, gjøres dette i et delingsrom. Delingsrommene kan opprettes både på mobile klienter og på faste maskiner i nettverket. Svarene fra subtransaksjonene vil bli sendt tilbake til MSSen og videre til Mac sin maskin. Figur 10.2 viser denne oppbygningen.



Figur 10.2: Arkitekturen i Ridderreisers databasesystem

10.2 Typiske transaksjoner i Ridderreiser

Selskapets kunder som ønsker å bestille reiser til syden blir gjerne tatt hånd om av en global transaksjon kalt G1. Subtransaksjonene F1, B1 og H1 inngår alle i G1. Subtransaksjon F1 tar seg av bestilling av flyreise, B1 reserverer leiebilen og H1 bestiller hotellrom.

F1 utfører disse operasjonene:

1. Spørring av ledige flyavganger med pris og tidspunkt for en gitt dato.
2. Spørring av ledige flyseter for en gitt flyavgang.
3. Reservasjon av flyavgang og flyseter.
4. Betaling av flyreise.

B1 utfører disse operasjonene:

1. Spørring av hvilke bilmerker som er ledige for en gitt dato.
2. Reservering av leiebil fra ankomsttiden for flyet til slutten av leieperioden.

H1 utfører disse operasjonene:

1. Spørring av ledige hotellrom og pris for en gitt dato.
2. Reservering av hotellrom fra ankomsttiden til flyet til slutten av oppholdet.

Salgssjefene i Ridderreiser vil kjøre flere typer transaksjoner for å kunne få oversikt over antall solgte reiser og gi gode tilbud på restplasser. Den globale transaksjonen G2 består av subtransaksjonene F2 og H2 som tar seg av henholdsvis spørring av ledige flyplasser og hotellrom.

F2 utfører disse operasjonene:

1. Spørring av hvilke tre flyavganger som har flest ledige seter en uke fram i tid.
2. Spørring av hvor mange flyseter som er ledige på alle flyavganger en uke fram i tid.

H2 utfører disse operasjonene:

1. Spørring av hvilke hotellrom som er ledige på de tre destinasjonene som har flest ledige seter.

10.2.1 Tradisjonell utførelse av transaksjonene G1 og G2

Ved tradisjonell utførelse av den globale transaksjonen G1 ville man først gruppert alle lesespørringene sammen, det vil si at man spør om ledige flyavganger, flyseter, bilmerker og hotellrom først. Deretter oppdaterer man databasen med de verdiene brukeren har valgt, og reservasjoner og betalinger blir utført. B1 og H1 trenger ankomsttiden fra flyet for å gjøre sine oppdateringer, og denne er tilgjengelig når F1 har gjort sin reservasjon. Imidlertid må F1 også betale flyreisen før den kan committe, og dermed må B1 og H1 vente til F1 er ferdig før de kan gjøre sine oppdateringer.

Hvis mange transaksjoner av typen G1 kjører i systemet, vil det ofte være låser satt av F1 på tabellene som lagrer informasjon om flyavganger og flyseter. Dette fører til at F2 får en tregere utførelse siden den ikke har lov til å lese data som er endret men ikke committet, og forsinkelsen kan bli spesielt lang hvis G1 blir hindret i sin utførelse av committ på grunn av frakoblinger i datanettverket eller lignende.

Tradisjonell utførelse av 2PC kan også gjøre at G1 må abortere i det tilfellet at koordinatoren ikke får svar fra en av deltakerne innen en fastsatt tid.

10.2.2 Optimalisert utførelse av transaksjonene G1 og G2

For å få raskere utførelse av subransaksjonene F1, B1 og H1 kan man la F1 dele ankomsttiden til B1 og H1 etter den har gjort reservasjonen men før den har utført betaling. Flere operasjoner kan dermed bli utført i parallell. Ulempen er at B1 og H1 må abortere hvis F1 aborterer, og man får dermed en høyere grad av bortkastet arbeid i databasene.

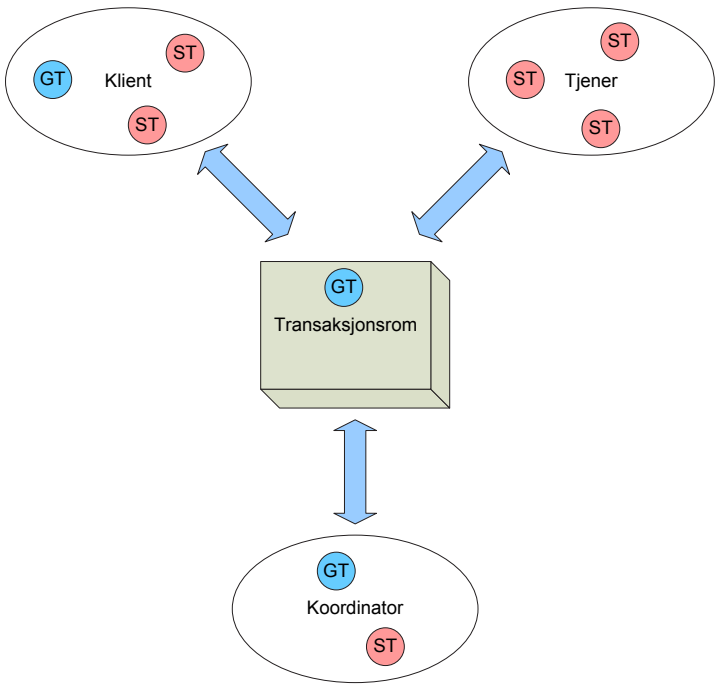
Ved å la subtransaksjonene committe sine verdier før den globale transaksjonen er klar for å committe, oppnår man at låsene slippes tidligere. Dermed vil for eksempel subtransaksjon F2 kjøre raskere siden den ikke lenger vil trenge å vente på at transaksjoner av typen G1 skal committe, men den må bare vente på at eventuelle subtransaksjoner av typen F2 skal kjøre ferdig. Ved abort av den globale transaksjonen G1 må man bruke kompenserende transaksjoner for å fjerne eventuelle endringer gjort av subtransaksjoner som har committet.

Ved å bruke en alternativ protokoll for utførelse av global committ kan man forbedre 2PC slik at man ikke behøver en koordinator og synkron utførelse av committ. Dette kan føre til færre aborter ved midlertidig tap av dataforbindelse.

Kapittel 11

Funksjonelle krav

Dette kapittelet beskriver de funksjonelle kravene til systemet. Kravene er inndelt i flere seksjoner etter konseptuell funksjonalitet, men det er ikke gitt at denne oppdelingen vil være lik den partisjoneringen av systemet vi kommer til å få i designdelen seinere i oppgaven. Til sist vil alle kravene bli oppsummert i en tabell som angir prioriteten og kompleksiteten til hvert krav. Figur 11.1 viser oversikten over de ulike komponentene som behandles og sammenhengen mellom disse. Av figuren går det frem at en koordinator, klient og tjener utveksler globale transaksjoner (GT) og subtransaksjoner (ST) seg imellom via et transaksjonsrom.



Figur 11.1: Oversikt over komponentene i kravspesifikasjonen

11.1 Transaksjoner

Denne seksjonen innheholder kravene til de forskjellige typene av transaksjoner som inngår i eksport-import modellen.

K1 Sammensetning av globale transaksjoner: Systemet skal kunne håndtere globale transaksjoner som er sammensatt av flere subtransaksjoner, der hver subtransaksjon er definert som en serie operasjoner som skal utføres mot kun en bestemt databasetjener.

K2 Unik global transaksjonsid: Alle globale transaksjoner skal ha en unik identifikator. Identifikatoren kan for eksempel være sammensatt av IP adressen til maskinen samt et tidsstempel. En unik id vil være viktig for å kunne hente ut transaksjoner fra transaksjonsrommet.

K3 Statusfelt: Alle transaksjoner skal ha en statusfelt som indikerer hvilken status transaksjonen er i. Dette brukes til commit og abort prosessering.

K4 Destinasjonsfelt: Alle transaksjoner skal ha et destinasjonsfelt som angir hvor de skal for å få utført neste trinn i prosesseringen, og dette kan være enten hos klient, tjener eller koordinator.

11.1.1 Eksporttransaksjoner

Deling av verdier skal skje ved at transaksjoner oppretter eksporttransaksjoner.

K5 Initiering av eksporteringsprosessen: En eksportørtransaksjon må initiere en eksporttransaksjon når den ønsker å sette igang en delingsprosess. For hver verdi den eksporterende transaksjonen skal dele må det opprettes en separat eksporttransaksjon. Eksportverdiene vil være tilgjengelige for tjeneren etter at resultatet fra operasjonen som har bedt om å eksportere en verdi er mottatt fra databasen.

K6 Eksportstruktur: En eksporttransaksjon skal kunne ha en flat struktur. Dette innebærer at en eksportransaksjon kun har et nivå av transaksjoner (ingen subtransaksjoner) og dermed må dele alle sine resultater i en omgang til delingsrommet.

K7 Kobling til importtransaksjoner: En eksporttransaksjon må ha en liste over alle importtransaksjoner som har benyttet verdien som eksporttransaksjonen tilbyr.

11.1.2 Importtransaksjoner

En transaksjon kan lese en delt verdi ved å opprette en importtransaksjon.

K8 Initiering av importeringsprosessen: En importørtransaksjon må initiere en importtransaksjon når den ønsker å importere data fra et delingsrom. For hver verdi den importerende subtransaksjonen er nødt til å importere, må det opprettes en egen importtransaksjon. Denne importtransaksjonen er bindeleddet mellom importørtransaksjonen og eksporttransaksjonen.

K9 Importstruktur: En importtransaksjon skal kunne ha en flat struktur. Dette innebærer at en importtransaksjon kun har et nivå av transaksjoner (ingen subtransaksjoner) og dermed må importere verdien i et steg.

K10 Kobling til eksporttransaksjoner: En importtransaksjon må inneholde identifikatoren til den eksporttransaksjonen den har importert verdien fra.

11.2 Mobil støttestasjon (MSS)

Den mobile støttestasjonen tar seg av koordineringen av transaksjoner samt delingen av verdier transaksjonene imellom. I store trekk fungerer den mobile støttestasjonen som et sted der de partene som er involverte i transaksjonsutførelsen utveksler spørringer, resultater og koordineringsmeldinger. I tillegg er støttestasjonen ansvarlig gi tilgang til delingsrom der verdier kan utveksles.

K11 Bruk av transaksjonsrom: Den mobile støttestasjonen skal være tilholdssted for et transaksjonsrom, hvor globale transaksjoner og subtransaksjoner blir utvekslet. Det er gjennom dette transaksjonsrommet at transaksjonene blir formidlet mellom klient, tjener og koordinator. Disse enhetene kan gjøre oppdateringer på transaksjoner, som for eksempel å oppdatere statusfeltet eller legge til resultater fra spørringer.

11.2.1 Delingsrom

Delingsrommene er de stedene hvor transaksjonene deler verdier seg imellom når de fremdeles er under utførelse.

K12 Dyamisk opprettelse av delingsrom: Nye delingsrom skal kunne opprettes dynamisk av en MSS når det finnes transaksjoner som har behov for dette. Årsakene til at nye delingsrom må opprettes kan være at de eksisterende delingsrommene er overbelastet, eller at man ønsker å opprette delingsrommet på et sted i nettverket som er nærmere klienten og/eller databasetjenerne.

K13 Tilordning av delingsrom: Delingsrommet skal ha en unik identifikator som gjør at det kan brukes av de tjenerne som vil benytte delingsrommet. Det er koordinatoren som gir delingsrommets identifikator til de interesserte tjenerne.

K14 Splitting av delingsrom: Hvis en gruppe noder mister nettverkstilkoblingen skal delingsrommet kunne splittes i mindre fragmenter.

K15 Forening av delingsrom: Hvis en gruppe noder får tilbake nettverkstilkoblingen skal delingsrommene kunne forenes i større fragmenter.

11.2.2 Koordinator

Koordinatoren er ansvarlig for å distribuere spørringer og svar mellom klienter og databasetjenere, og den utfører også logging, feilhåndtering og global commit eller abort.

K16 Bestemmelse av utfall til transaksjoner: Koordinatoren skal lese inn alle de subtransaksjonene som hører til en global transaksjon hver gang det skjer en endring i en av subtransaksjonene. Slik kan den ta en avgjørelse på hva som skal bli utfallet av den globale transaksjonen. Etter at koordinatoren har tatt avgjørelsen skal eventuelle nye statusverdier til den globale transaksjonen og subtransaksjonene oppdateres.

K17 Global commit: Når alle subtransaksjonene samt den globale transaksjonen de tilhører har en statusverdi som indikerer at de er klar til å committe, skal koordinatoren sette statusfeltet hos transaksjonene til commit.

K18 Global abort: Når noen av subtransaksjonene har møtt på en feil under utførelse og fått statusfeltet satt til abortert, skal koordinatoren gjøre en vurdering på grunnlag av regelsettet og sette status til abort hos den globale transaksjonen og alle de andre subtransaksjonene som tilhører denne hvis reglene sier den aborterte subtransaksjonen er kritisk for utfallet.

K19 Delvis abort: Dersom en subtransaksjon blir abortert av brukeren fordi brukeren angrer på at denne subtransaksjonen skal bli utført, vil koordinatoren sørge for at alle eksporttransaksjoner som subtransaksjonen har initiert også vil bli abortert. Dette fører igjen til at alle importtransaksjoner som har brukt verdier fra eksporttransaksjonen blir abortert, og alle subtransaksjoner som har brukt importtransaksjonene vil også bli abortert. Deretter kan eventuelt en ny subtransaksjon startes av brukeren.

K20 Maksimal tid i transaksjonsrommet: Hvis ikke en subtransaksjon blir utført etter en viss brukerdefinert maksimaltid, skal denne subtransaksjonen bli registrert som abortert.

K21 Tilpassing av regelsett: Transaksjonsstasjonen skal kunne motta et regelsett som sier hvilke subtransaksjoner som fører til at den globale transaksjonen blir abortert om de selv må abortere.

11.3 Databasetjener

Databasetjeneren henter ut subtransaksjoner som skal utføres på en database tjeneren har ansvaret for. Den skriver resultatene fra subtransaksjonen tilbake igjen til transaksjonsrommet, og verdier som skal eksporteres til andre transaksjoner blir pakket inn i en eksporttransaksjon og skrevet til rett delingsrom. Importtransaksjoner blir også opprettet her når tjeneren er nødt til å hente inn verdier den ikke har tilgang til for å få fullført transaksjonene.

K22 Unik navngiving av databaser: De forskjellige databasene skal være unikt navngitt på tvers av hele systemet.

11.3.1 Transaksjonshåndterer

Transaksjonshåndtereren fungerer som bindeleddet mellom databasen og den mobile støttestasjonen. Den har ansvaret for å utføre transaksjonene på selve databasen, samt hente ut og skrive spørringer, resultater og meldinger til transaksjonsrommet.

K23 Tilbakemelinger fra transaksjonene: Klienten må få tilbakemelding på utfallet av transaksjonen. Commit eller abort skal angis, og eventuelt hvilke subtransaksjoner som ble abortert.

K24 Skriving av transaksjonsresultat: En subtransaksjon skal kunne skrive resultatet av en spørring til transaksjonsrommet. Resultatet av en spørring lagres i delingsrommet på XML format slik at klienten som ba om spørringen kan hente den ut.

K25 Eksekvering av subtransaksjoner: Transaksjonhåndtereren skal kun hente ut subtransaksjoner som tilhører en database som databasetjeneren har ansvaret for. Operasjonene spesifisert i subtransaksjonen utføres mot databasen, og subtransaksjonens status oppdateres og skrives tilbake til transaksjonsrommet. Subtransaksjonens status skal reflektere at den er klar for commit hvis alt gikk bra, ellers skal status settes til abortert.

K26 Commit etter brudd på nettverksforbindelsen: Hvis en subtransaksjon er klar til å committe men mister tilgang til nettverket, skal den få committe sine resultater når den får tilgang igjen hvis den globale transaksjonen den var en del av har committet.

K27 Commit og abort prosessering: Alle subtransaksjonene som har utført operasjonene sine og har status satt til klar til commit, må vente på oppdateringer som koordinatoren gjør på subtransaksjonen. Hvis tjeneren får beskjed om å committe subtransaksjonen skal alle operasjonene som er utført av subtransaksjonen i databasen committes, og i motsatt fall skal de rulles tilbake.

K28 Tilbakeskriving av subtransaksjon: I det tilfelle en databasenode ikke klarer å gjennomføre en subtransaksjon på grunn av årsaker som systemkræsj og lignende, må subtransaksjonen gjøres tilgjengelig i transaksjonsrommet igjen slik at den kan bli forsøkt utført på et seinere tidspunkt.

11.3.2 DBMS

Databasesystemet implementerer de tradisjonelle ACID egenskapene, og det er her data blir lagret permanent. Hver databasetjener kan ha tilgang til flere slike databaser.

K29 Spesifisering av commit tidspunkt: Databasen må støtte utførelse av operasjoner som ikke blir committet umiddelbart. Det trengs en egen commit operasjon som gjør de siste endringene gjort av transaksjonen i databasen endelige.

11.4 Mobil klient

Den mobile klienten genererer de transaksjonene som utføres i systemet. Disse kan komme fra batch prosessering av lister med transaksjoner, eller via et interaktivt brukergrensesnitt. Resultatene fra transaksjonene blir sendt tilbake til klienten slik at brukeren kan se dem.

K30 Innverdier fra brukeren: Innverdier som initielt står tomme i spørringen og som krever interaksjon fra brukeren, skal kunne formidles til delingsrommet når de er tilgjengelige på et seinere tidspunkt ved hjelp av en eksporttransaksjon.

K31 Visning av resultater fra transaksjoner: Klienten skal kunne lese resultatet av spørringer som klienten har formidlet til transaksjonsrommet.

11.4.1 Transaksjonsgenerator

Transaksjonsgeneratoren leser inn XML filer som er lagret på disk og gir disse til XML parseren som lager transaksjonsobjekter av dem. Transaksjonene blir så matet inn til transaksjonsrommet.

K32 Innlesing av transaksjoner fra fil: Transaksjonsgeneratoren skal kunne lese inn transaksjoner som er spesifisert i dokumenter på XML format. Disse dokumentene gis til XML parseren som konstruerer transaksjonsobjekter av dem.

K33 Skriving av transaksjoner: Transaksjonsobjektene returnert av XML parseren skal skrives til transaksjonsrommet.

11.4.2 XML parser

XML parseren tar inn dokumenter og produserer lister med subtransaksjoner og globale transaksjoner som er klare til å sendes ut til transaksjonsrommet.

K34 Korrekthet av XML dokumenter: XML dokumentene må være skrevet i henhold til den gramatikken som parseren forventer å få som input.

K35 Konstruksjon av transaksjonsobjekter fra XML dokumenter: Parseren skal kunne lese inn dokumenter på XML format og konstruere transaksjonsobjekter av disse.

11.4.3 GUI

Det grafiske brukergrensesnittet tillater brukeren å skrive inn transaksjoner interaktivt.

K36 Spesifisering av transaksjoner interaktivt: Brukeren skal kunne formidle operasjoner til transaksjoner via grensesnittet og få tilbakemeldinger på disse.

11.5 Tabelloversikt over kravene

I tabell 11.1 er det samlet en oversikt over alle kravene med prioritering og kompleksitet gradert som enten lav, medium eller høy.

Tabell 11.1: Oversikt over krav

Nr	Krav	Prioritet	Kompleksitet
	Transaksjoner		
K1	Sammensetning av globale transaksjoner	Høy	Lav
K2	Unik global transaksjonsid	Høy	Lav
K3	Statusfelt	Høy	Lav
K4	Destinasjonsfelt	Høy	Lav
	Eksporttransaksjoner		
K5	Initiering av eksporteringsprosessen	Høy	Medium
K6	Eksportstruktur	Høy	Lav
K7	Kobling til importtransaksjoner	Høy	Lav
	Importtransaksjoner		
K8	Initiering av importeringsprosessen	Høy	Medium
K9	Importstruktur	Høy	Lav
K10	Kobling til eksporttransaksjoner	Høy	Lav
	Mobil støttestasjon (MSS)		
K11	Bruk av transaksjonsrom	Høy	Lav
	Delingsrom		
K12	Dyamisk opprettelse av delingsrom	Lav	Høy
K13	Tilordning av delingsrom	Lav	Medium
K14	Splitting av delingsrom	Lav	Høy
K15	Forening av delingsrom	Lav	Høy
	Koordinator		
K16	Bestemmelse av utfall til transaksjoner	Høy	Medium
K17	Global commit	Høy	Lav
K18	Global abort	Høy	Lav
K19	Delvis abort	Høy	Medium
K20	Maksimal tid i transaksjonsrommet	Medium	Lav
K21	Tilpassing av regelsett	Medium	Medium
	Databasetjener		
K22	Unik navngiving av databaser	Høy	Lav
	Transaksjonshåndterer		
K23	Tilbakemelinger fra transaksjonene	Høy	Lav
K24	Skriving av transaksjonsresultat	Høy	Medium
K25	Eksekvering av subtransaksjoner	Høy	Lav
K26	Commit etter brudd på nettverksforbindelsen	Høy	Medium
K27	Commit og abort prosessering	Høy	Lav
K28	Tilbakeskriving av subtransaksjon	Høy	Medium

Fortsettelse følger på neste side

Tabell 11.1 – fortsettelse fra forrige side

Nr	Krav	Prioritet	Kompleksitet
	DBMS		
K29	Spesifisering av commit tidspunkt	Høy	Medium
	Mobil klient		
K30	Innverdier fra brukeren	Medium	Medium
K31	Visning av resultater fra transaksjoner	Medium	Medium
	Transaksjonsgenerator		
K32	Innlesing av transaksjoner fra fil	Medium	Medium
K33	Skriving av transaksjoner	Høy	Lav
	XML parser		
K34	Korrekthet av XML dokumenter	Høy	Lav
K35	Konstruksjon av transaksjonsobjekter fra XML dokumenter	Høy	Medium
	GUI		
K36	Spesifisering av transaksjoner interaktivt	Medium	Medium
	Tabelloversikt over kravene		

Kapittel 12

Oppsummering

De funksjonelle kravene skal danne grunnlaget for designet av systemet som blir omhandlet i neste del i oppgaven. Imidlertid er det ikke alle kravene som har fått like høy prioritet, dette gjelder spesielt for kravene som omhandler de dynamiske egenskapene til delingsrom som alle har fått *lav* prioritet. Disse kravene vil ikke bli tatt med videre i designet av systemet fordi vi anser disse som for komplekse å få gjennomført gitt den tiden vi har til rådighet. De andre kravene vil inngå videre videre i arbeidet med å skape en prototype som demonstrerer de viktigste konseptene.

Del IV

Design

Innledning

Denne delen inneholder designet av systemet. Først blir systemet partisjonert i mindre entiteter, og disse entitetene blir så brukt videre for å finne avhengigheter mellom de samt for å beskrive de på en så detaljert form at de kan danne grunnlaget for en implementasjon. Entitetene blir også knyttet opp mot både kravspesifikasjonen og implementasjonen.

Arbeidet med å spesifisere relevante aspekter av systemet har tatt utgangspunkt i en anbefalt fremgangsmåte som er beskrevet i *IEEE Recommended Practice for Software Design Descriptions* [23].

- Kapittel 13, Overordnet systemarkitektur, tar for seg oppdelingen av systemet til mindre entiteter og beskriver disse på et overordnet nivå.
- Kapittel 14, Avhengigheter, beskriver avhengighetene mellom entitene i systemet samt innad i hver entitet.
- Kapittel 15, Grensesnitt, omhandler de grensesnittene som systemet må forholde seg til.
- Kapittel 16, Detaljert design, beskriver virkemåten til hver entitet og har også klassediagrammer som viser oppbygningen av systemet.

Kapittel 13

Overordnet systemarkitektur

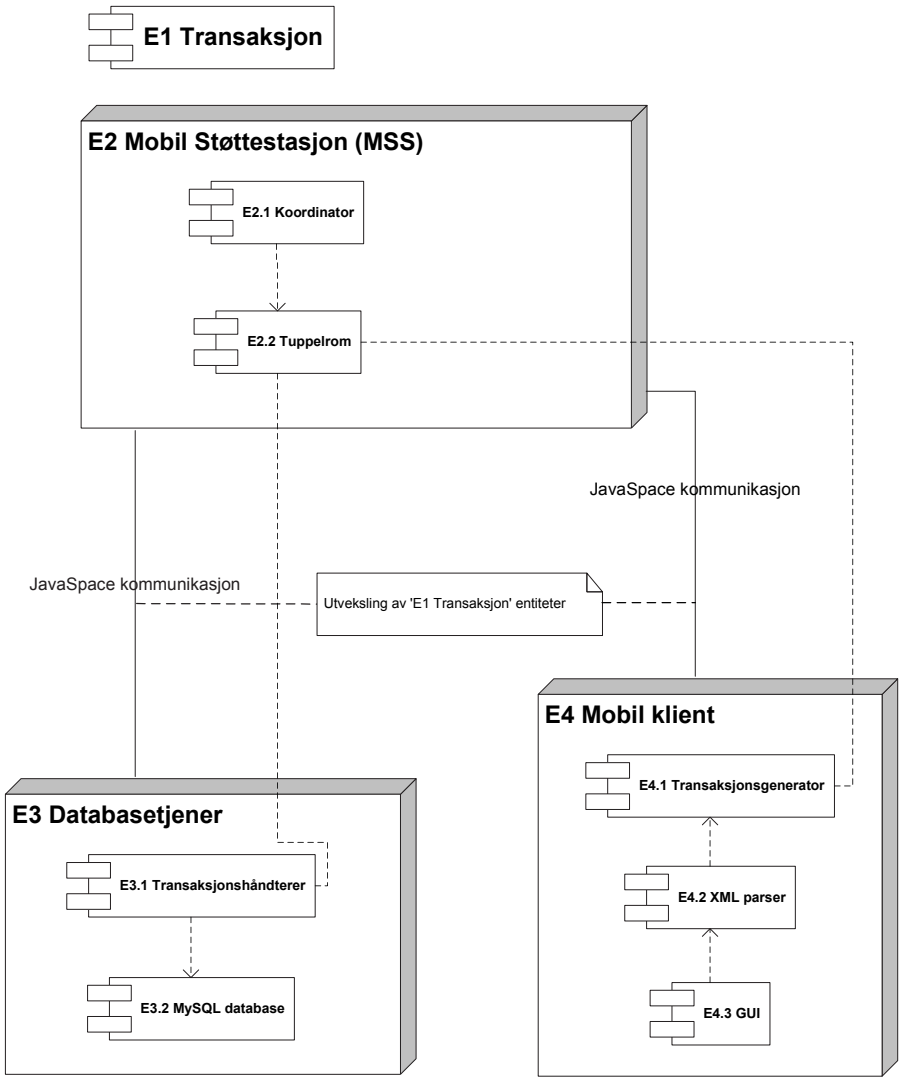
I dette kapittelet vil systemet bli dekomponert i mindre designentiteter. Måten systemet strukturereres på og formålet og funksjonen til hver entitet vil bli grundig beskrevet. Hver entitet vil også bli gitt en unik identifikator som kan brukes for å finne igjen entiteten i det detaljerte designet som følger i et senere kapittel.

Hver entitet består av følgende attributter:

- **Identifikasjon:** En unik identifikator som for eksempel E2.1. Dette gir et hierarkisk navnesystem.
- **Type:** Angir hvilken art entiteten er. Dette kan for eksempel være en modul eller en dataentitet.
- **Formål:** Gir en beskrivelse av hvorfor entiteten eksisterer. De spesifikke kravene som entiteten implementerer skal listes opp her.
- **Funksjon:** Hva entiteten faktisk gjør skal beskrives her.
- **Underordnete entiteter:** De entitetene som denne entiteten består av skal listes opp her.

13.1 Oversikt - toppnivå

Figur 13.1 illustrer toppnivå entitene til systemet og hvordan disse er relatert til hverandre. En del entiteter på nivået under er også tatt med, og disse er tegnet inn i sine respektive overordnete entiteter.

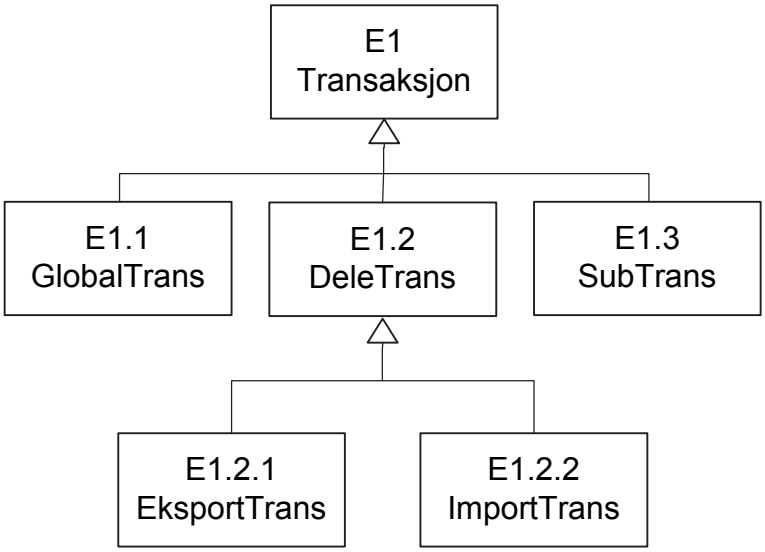


Figur 13.1: Oversikt over de overordnete entitene i systemet

13.2 E1 Transaksjon

E1 Transaksjon er en dataenitiet og utgjør strukturen til transaksjonene.

- **Type:** Dataentitet, *abstrakt*
- **Formål:** Utgjør øverste transaksjonsnivå. Oppfyller kravene K2, K3 og K4.
- **Funksjon:** Lagrer tilstandsinformasjon for de involverte transaksjonene i systemet som *global id* og *status*. Har også et *destinasjonsfelt* for å angi hvor transaksjonen må forflytte seg for å få utført neste trinn i utførelsesprosessen. Destinasjonsfeltet kan være koordinator (E2.1), tjener (E3) eller klient (E4).
- **Underordnete entiteter:** E1.1 GlobalTrans, E1.2 DeleTrans og E1.3 SubTrans.



Figur 13.2: Transaksjonstyper i eksport-import modellen

E1.1 GlobalTrans

E1.1 GlobalTrans består av et sett av E1.3 SubTrans, som utgjør selve oppgaveutførelsene i form av spørringer.

- **Type:** Dataentitet
- **Formål:** Representerer det øverste nivået av instansierbare transaksjonsobjekter, sett fra brukerens ståsted. Hver global transaksjonen består av et sett med subtransaksjoner. Oppfyller kravet K1.
- **Funksjon:** Har som oppgave å lagre identifikatorene til alle de tilhørende subtransaksjonene. En global transaksjon utgjør den enheten som koordinatoren utøver commit- og abortprosessering mot.

E1.2 DeleTrans

E1.2 DeleTrans representerer transaksjoner som deler verdier.

- **Type:** Dataentitet
- **Formål:** Transaksjon som muliggjør asynkron deling av verdier. Eksempler på dette er eksport- og import-transaksjoner.
- **Funksjon:** DeleTrans inneholder navnet og verdien til variabelen som ønskes delt, samt identifikatoren til transaksjonen som eksporterer eller importerer denne verdien.
- **Underordnete entiteter:** E1.2.1 EksportTrans og E1.2.2 ImportTrans.

E1.2.1 EksportTrans

E1.2.1 EksportTrans representerer transaksjoner som kan dele verdier ved å eksportere dem til et felles oppbevaringssted.

- **Type:** Dataentitet
- **Formål:** Muliggjør at transaksjoner kan dele (tilgjengeliggjøre) ønskede verdier til andre transaksjoner. Oppfyller kravene K5, K6 og K7.
- **Funksjon:** Subtransaksjoner som ønsker å dele verdier kan initiere en EksportTrans ved å gi denne navn og tilhørende verdi til den variabelen som skal deles, samt knytte sin egen *id* til eksporttransaksjonen. En ExportTrans må også holde oversikt over hvilke import-transaksjoner som leser (importerer) verdier fra den. Selve delingsprossen foregår ved at verdien deles i en enkelt omgang (flat struktur).

E1.2.2 ImportTrans

E1.2.2 ImportTrans representerer transaksjoner som kan lese verdier som allerede er eksportert til et felles oppbevaringssted.

- **Type:** Dataentitet
- **Formål:** Underordnet DeleTrans, muliggjør at transaksjoner kan lese / importere verdier som allerede er blitt tilgjengeliggjort av en ExportTrans. Oppfyller kravene K8, K9 og K10
- **Funksjon:** ImportTrans inneholder informasjon om navnet på deleverdi med tilhørende verdi. Transaksjonen er knyttet opp mot en subtransaksjon og inneholder derfor subtrans-aksjonens *id*. Hver ImportTrans har referanse til den eksport transaksjonen den er knyttet til, i form av eksporttransaksjonens id. Importtransaksjonene har en flat struktur.

E1.3 SubTrans

E1.3 SubTrans er transaksjoner som inneholder selve oppgaveutførelsen i form av SQL spør-ringer.

- **Type:** Dataentitet
- **Formål:** Utgjør selve oppgaven brukeren ønsker å utføre i form av SQL-spørringer. Vil alltid være tilknyttet en global transaksjon.
- **Funksjon:** Inneholder egen *subId*, som er unik innenfor en global transaksjon. Lagrer også informasjon om hvilke SQL spørringer som skal utføres, samt informasjon om hvil-ken database den skal utføres ved.

13.3 E2 Mobil Støttestasjon (MMS)

E2 Mobil Støttestatsjon tilbyr tjenester som å oppbevare delte verdier, samt å koordiere de mobile transaksjonsutførelsene.

- **Type:** Modul
- **Formål:** MSSen fungerer som et grensesnitt for den mobile klienten mot databasene i det faste nettverket. Den innehar funksjonalitet som gjør det mulig for klienter å kjøre mobile transaksjoner. Oppfyller kravet K11.
- **Funksjon:** MSSens hovedoppgave er å oppbevare informasjon som de mobile transaksjonene utveklser seg imellom. For at transaksjonshåndteringen skal bli korrekt sørger en koordinator for at kjøringene blir utført på en riktig måte.
- **Underordnete entiteter:** E2.1 Koordinator og E2.2 Tuppelrom.

E2.1 Koordinator

E2.1 Koordinator sørger for at de transaksjonelle utførelsene blir korrekte.

- **Type:** Javaapplikasjon
- **Formål:** Koordinatoren har som oppgave å administrere E2.2 Tuppelrom samt å styre transaksjonshåndteringen ved å utveklse transaksjonsobjekter 13.2 med klient og tjener. Oppfyller kravene K16, K17, K18 og K19.
- **Funksjon:** Koordinatoren må hele tiden observere statusendringer for de ulike transaksjonene i tuppelrommet, og ta en avgjørelse for den videre prosesseringen. Koordinatoren kommuniserer med klient og tjener via transaksjonsobjektene i tuppelrommet ved å aksessere og oppdatere disse.

E2.2 Tuppelrom

E2.2 Tuppelrom fungerer som et oppbevaringssted for transaksjonsobjekter og delte verdier.

- **Type:** Javaspace applikasjon
- **Formål:** Tuppelrommet muliggjør asynkron deling av objekter mellom en gruppe av distribuerte prosesser.
- **Funksjon:** Tuppelrommet fungerer som et felles oppbevaringssted hvor henholdsvis de mobile klientene og tjenerene kan utveklse informasjon gjennom å dele objekter. Tuppelrommet oppbevarer både transaksjonsobjekter (transaksjonsrom), som beskrevet i 13.2, samt verdiene som deles mellom transaksjonene.

13.4 E3 Databasetjener

Databasetjeneren består av en transaksjonshåndterer for å hente subtransaksjoner fra transaksjonsrommet og utføre disse mot en MySQL database.

- **Type:** Modul
- **Formål:** Databasetjeneren skal oppfylle kravet K22.
- **Funksjon:** Databasetjeneren henter inn subtransaksjoner fra transaksjonsrommet og utfører disse. Den følger regelsettet definert for transaksjonen i forhold til commit regler og kompenserende transaksjoner. Transaksjonene blir skrevet permanent til databasen når databasen får beskjed om å utføre commit. Eventuelle resultater fra transaksjonen blir skrevet tilbake i transaksjonsrommet. Verdier som deles transaksjoner imellom under utførelse blir lest fra og skrevet til et delingsrom.
- **Underordnete entiteter:** E1.1 Transaksjonshåndterer og E1.2 MySQL database.

E3.1 Transaksjonshåndterer

Transaksjonshåndtereren er ansvarlig for å skrive subtransaksjoner til og fra delingsrommet, samt å gi disse videre til databasen.

- **Type:** Java-programvare
- **Formål:** Transaksjonshåndtereren skal oppfylle kravene K23, K24, K25, K26, K27 og K28.
- **Funksjon:** Transaksjonshåndtereren skal fungere som et bindeledd mellom transaksjonsrommet og databasen. Den henter ut spørringer og gir dem videre til databasen via JDBC grensesnittet. Den mottar også svar fra databasen og skriver de tilbake til transaksjonsrommet.

E3.2 MySQL database

MySQL databasen utfører de spørringene den mottar fra transaksjonshåndereren og gjør disse permanente ved hjelp av commit av transaksjonen, eller forkaster alt ved hjelp av rollback.

- **Type:** DBMS
- **Formål:** Databasen skal oppfylle kravet K29.
- **Funksjon:** Databasen skal lagre transaksjonene på permanent lager. Den implementerer ACID egenskapene og sørger for at hver subtransaksjon utføres korrekt.

13.5 E4 Mobil klient

Den mobile klienten er ansvarlig for å gi spørringer til systemet samt vise resultatene av disse til brukeren.

- **Type:** Modul
- **Formål:** Den mobile klienten skal oppfylle kravene K30 og K31.
- **Funksjon:** Den mobile klienten er brukerens grensesnitt mot systemet. Den tar imot spørringer i XML format fra fil eller direkte fra et GUI. Deretter parses XML dataene og det lages transaksjonsobjekter. Disse transaksjonene videresendes til transaksjonsrommet. Resultatene fra spørringen mottas så fra transaksjonsrommet og presenteres for brukeren.
- **Underordnete entiteter:** E4.1 Transaksjonsgenerator, E4.2 XML parser og E4.3 GUI.

E4.1 Transaksjonsgenerator

Transaksjonsgeneratoren leser inn XML dokumenter fra fil og gir disse til XML parseren.

- **Type:** Modul
- **Formål:** Transaksjonsgeneratoren oppfyller kravene K32 og K33.
- **Funksjon:** Transaksjonsgeneratoren leser XML filer fra disk og gir dem videre til XML parseren. Den støtter batch-prosessering av transaksjoner, det vil si at en stor mengde transaksjoner blir lest inn og eksekvert samtidig.

E4.2 XML parser

XML parseren går gjennom XML dokumentene den mottar og konstruerer transaksjonsobjekter av disse.

- **Type:** Modul
- **Formål:** XML parseren skal oppfylle kravene K34 og K35.
- **Funksjon:** XML parseren mottar XML dokumenter og konstruerer transaksjonsobjekter av dem.

E4.3 GUI

Det grafiske brukergrensesnittet skal gjøre brukeren istand til å spesifisere transaksjoner interaktivt.

- **Type:** Modul
- **Formål:** Det grafiske brukergrensesnittet skal oppfylle kravet K36.
- **Funksjon:** Det grafiske brukergrensesnittet skal ta imot verdier og spørringer fra brukeren og videresende dem til XML parseren. Det skal også kunne aktivere transaksjonsgeneratoren slik at transaksjoner kan leses fra disk. Resultater fra kjørte transaksjoner og statistikker skal presenteres her.

Kapittel 14

Avhengigheter

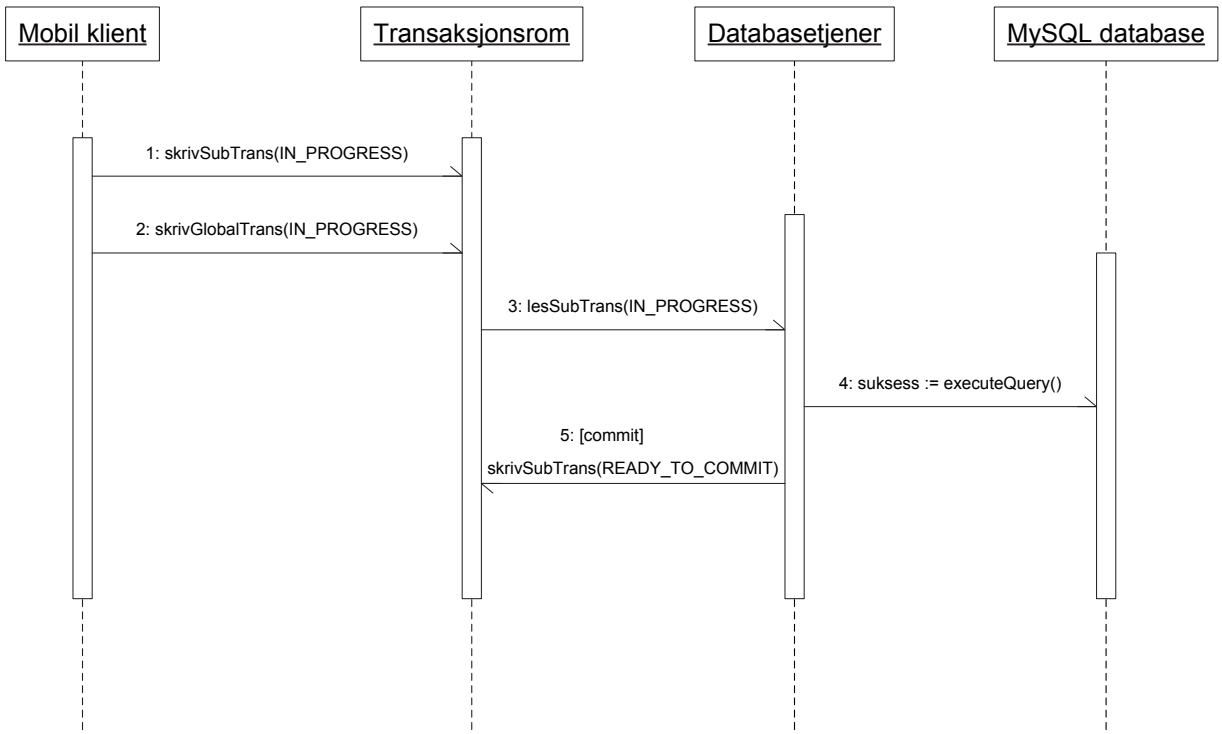
Dette kapittelet inneholder en beskrivelse av avhengighetene mellom entitetene i systemet. Avhengighetene mellom de overordnede entitetene E2 Mobil Støttestasjon, E3 Databasetjener og E4 Mobil Klient består av utvekslinger av meldinger av typen E1 Transaksjon. Disse meldingsutvekslingene vil bli beskrevet i seksjonen om systemavhengigheter. Avhengigheter innad i de overordnede entitetene vil bli beskrevet i de seksjonene som omhandler den spesifikke entiteten.

14.1 Systemavhengigheter

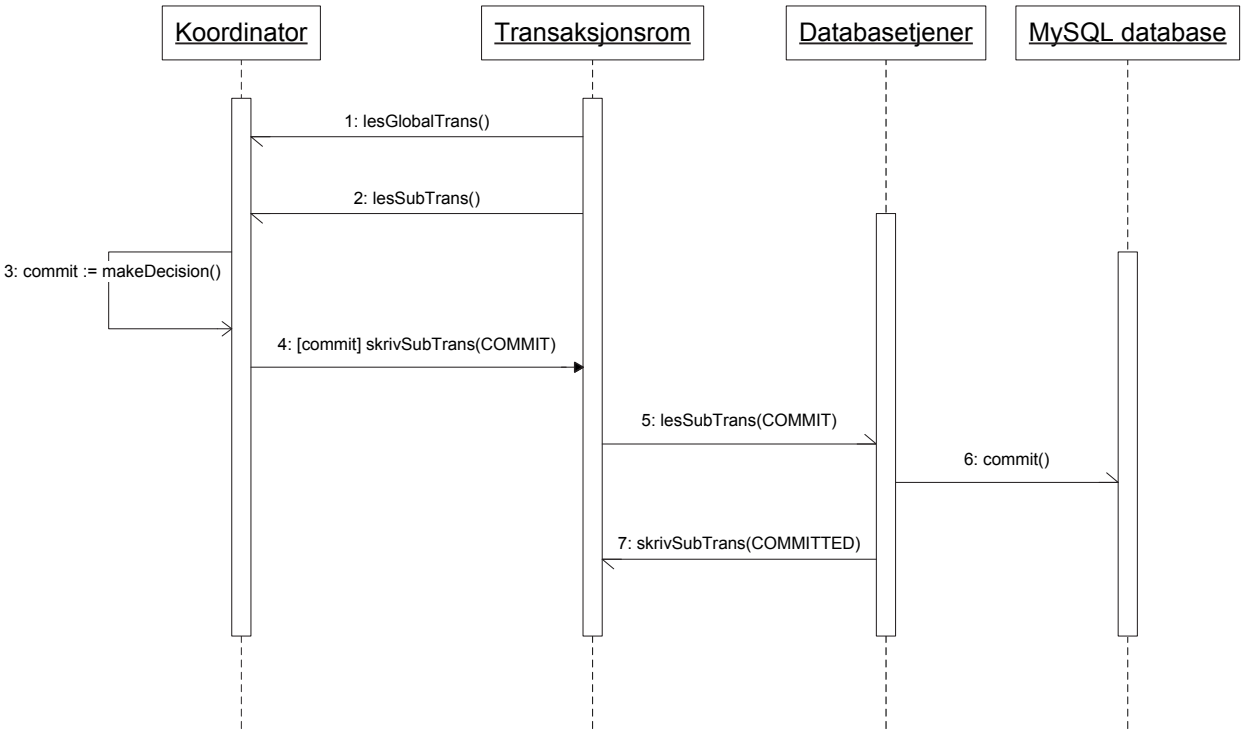
UML sekvensdiagrammer [16] er her brukt for å beskrive de tre forskjellige situasjonene: *klient initierer transaksjoner*, *koordinator committer transaksjoner* og *koordinator aborterer transaksjoner*.

Figur 14.1 viser starten på transaksjonsprosesseringen der en klient skriver ut en global transaksjon og en rekke tilhørende subtransaksjoner til transaksjonsrommet (steg 1 og 2). I steg 3 leser databasetjeneren inn de subtransaksjonene som inneholder spørringer som skal utføres mot en av de databasene som tjeneren har ansvaret for. I steg 4 utføres disse operasjonene mot databasen, og man får en suksess variabel som er sann hvis alt gikk bra, ellers er den usann. I steg 5 blir subtransaksjonen skrevet tilbake til transaksjonsrommet med status satt til `READY_TO_COMMIT` hvis steg 4 var en suksess.

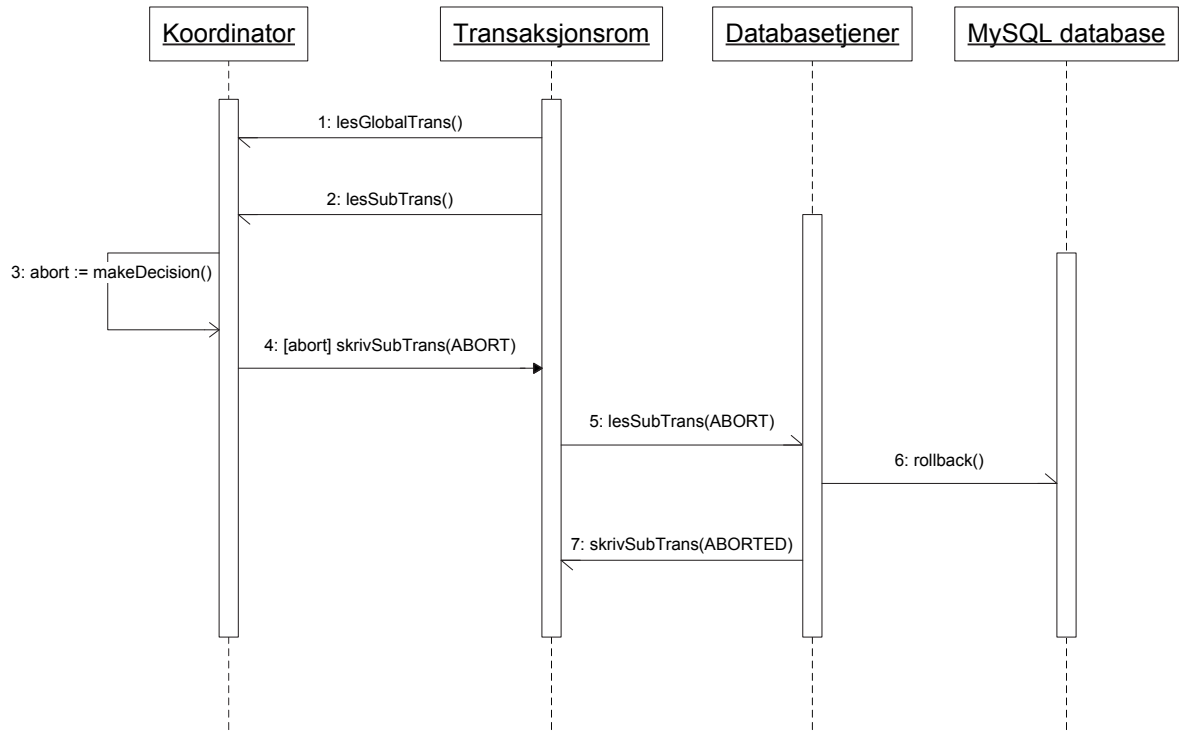
Figur 14.2 beskriver et tilfelle der koordinatoren committer en global transaksjon. I steg 1 leses det inn en global transaksjon, og deretter leses alle de subtransaksjonene som tilhører den globale transaksjonen inn i steg 2. I steg 3 gjør koordinatoren en avgjørelse på om den globale transaksjonen skal committe, abortere eller forbli uendret på grunnlag av statusverdiene til subtransaksjonene. I steg 4 committes alle subtransaksjonene under den betingelsen at den globale transaksjonen committed i steg 3. I steg 5 leser databasetjeneren inn subtransaksjoner med status lik `COMMIT`, og i steg 6 committes disse i databasen. I steg 7 skrives subtransaksjonen tilbake til transaksjonsrommet med resultatene fra spørringen og statusverdi lik `COMMITTED`.



Figur 14.1: Klient initierer transaksjoner



Figur 14.2: Koordinator committer transaksjoner



Figur 14.3: Koordinator aborterer transaksjoner

Figur 14.3 viser et tilsvarende tilfelle som i figur 14.2, men her velger koordinatoren å abortere subtransaksjonene. Steg 1, 2 og 3 er like, men i steg 4 skriver koordinatoren ut subtransaksjoner til transaksjonsrommet med status lik `ABORT`. Databasetjeneren leser disse inn i steg 5 og utfører rollback mot databasen i steg 6. I steg 7 skrives subtransaksjonen tilbake til transaksjonsrommet med status lik `ABORTED`.

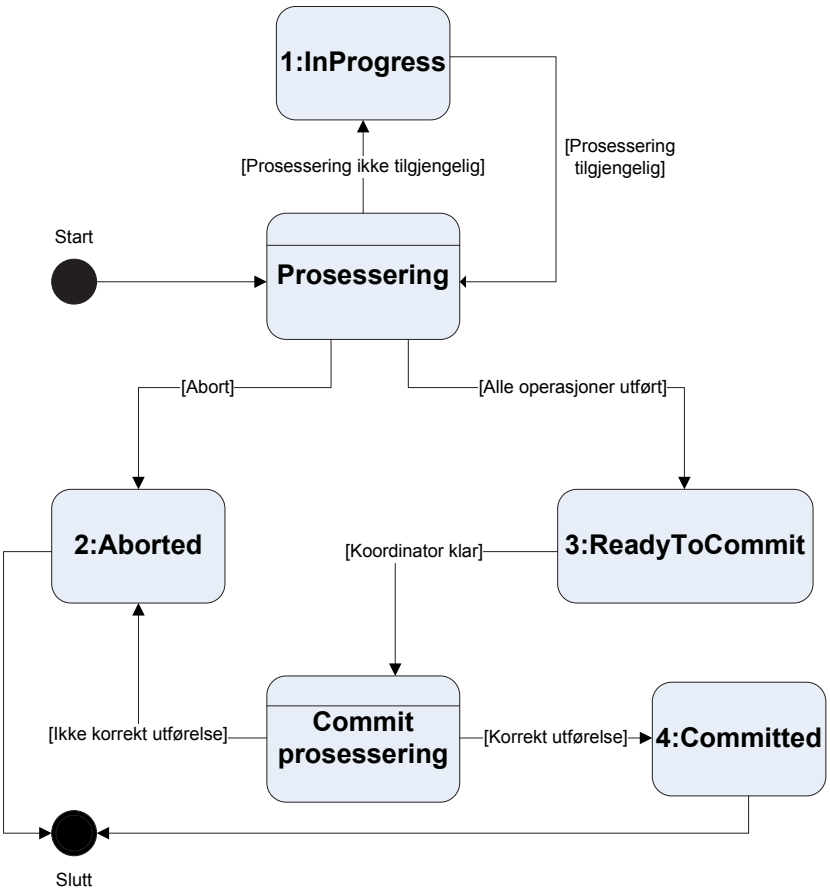
14.2 E1 Transaksjon

Figur 14.4 gir en overordnet oversikt over de ulike tilstandene som globale transaksjoner og subtransaksjoner kan befinne seg i.

Etter initieringen av en transaksjon starter selve operasjonsutførelsen. Normalt vil en transaksjon bestå av en sekvens av *read* og *write* operasjoner utført på dataobjekter i databasen.

Den første fasen en transaksjon befinner seg i vil være selve utførelsesfasen:

- **1: InProgress:** Transaksjonen er i en fase hvor det forsatt er flere operasjoner som må utføres. Først når alle operasjonene er prosessert, eller transaksjonen har mottatt beskjed om at den skal *abortere* kan den bevege seg over i en av de neste tilstandene.
- **2: Aborted:** Denne tilstanden viser at transaksjonen ikke ble uført på en korrekt måte eller at den mottok beskjed om å abortere. Sistnevnte årsak kan skyldes at transaksjonen var i et avhengighetsforhold til en annen transaksjon som måtte abortere.



Figur 14.4: Oversik over tilstander for globale- og sub-transaksjoner

- **3: ReadyToCommit:** Når en transaksjon har utført alle sine operasjoner uten å mottatt melding om å abortere kan den gjøre seg klar til å committe resultatene sine. Den videre utførelsen bestemmes av koordinatoren i systemet, så om transaksjonen har en korrekt utførelse så vil den gå over i fase 4.
- **4: Committed:** Utførelsen til transaksjonen var korrekt og dens resultater er reflektert i databasen.

14.3 E2 Mobil støttestasjon (MSS)

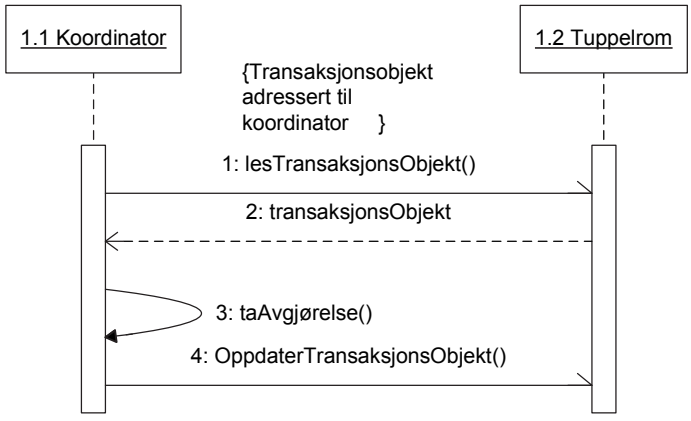
Den mobile støttestasjonen er satt sammen av henholdsvis en koordinator og et tuppelrom. Tuppelrommet legger grunnlaget for alle transaksjonene i systemet ved at de her blir lagret som egne transaksjonsobjekter. Transaksjonene blir styrt av en egen koordinator (E2.1), som sørger for at utførelsene blir korrekte. I dette avsnittet vil det bli fokusert på å få fram avhengighets-forholdet mellom E2.1 Koordinator og E2.2 Tuppelrom, ved å beskrive hvordan koordinatoren styrer transaksjonsutførelsen gjennom å lese og oppdatere transaksjonsobjektene.

Først kommer en oversikt over selve avhengigsforholdet mellom koordinatoren og tuppelrom-met. Deretter beskrives de interne avhengighetene til koordinatoren i forhold til tuppelrommet og E1 Transaksjon.

14.3.1 Interaksjon mellom koordinator og tuppelrom

Figur 14.5 viser en konseptuell oversikt over interaksjonen mellom koordinator og tuppelrom. Transaksjonsobjektene som er lagret i tuppelrommet er utgangspunktet for deres avhengighetsforhold, hvor koordinatoren lytter på objekter som er adressert til seg. En kort beskrivelse av denne interaksjonen vil bli gitt under:

- **1: lesTransaksjonsObjekt():** koordinatoren lytter på transaksjonsobjekter som er lagret i tuppelrommet. Når eksempelvis E3 Databasetjener gjør oppdateringer på et transaksjonsobjekt vil den adressere dette til E2.1 Koordinator. Koordinatoren vil da lese inn dette objektet.
- **2: transaksjonsObjekt:** tuppelrommet returnerer det gjeldende objektet til koordinatoren.
- **3: taAvgjørelse():** koordinatoren vil sjekke status de transaksjonsobjektene som hører sammen og ta en avgjørelse på grunnlag av dette. Denne prosesseringen vil bli beskrevet grundigere i kapittel 16 om detaljert design.
- **4: oppdaterTransaksjonsObjekt():** koordinatoren vil oppdatere transaksjonsobjektet og skrive dette tilbake til tuppelrommet.



Figur 14.5: Interaksjon mellom koordinator og tuppelrom

14.4 E2 Databasetjener

Databasetjeneren har en rekke avhengigheter både internt og eksternt mot andre enheter. Internt er E3.1 Transaksjonshåndterer avhengig av E3.2 MySQL database for å utføre de transaksjonene den mottar. Transaksjonshåndtereren bruker en JDBC *(Java DataBase Connectivity)* driver

for å kommunisere med MySQL databasen. Denne driveren kalles for MySQL Connector/J og er beskrevet i [2].

Transaksjonshåndtereren mottar subtransaksjoner fra transaksjonsrommet og utfører alle de SQL operasjonene som er beskrevet der mot databasen. Databasen returnerer så svaret av spørringen som et sett av tupler hvis SQL spørringen var en SELECT spørring, eller den returnerer antall tupler som ble modifisert hvis spørringen var av typen UPDATE, INSERT eller DELETE.

14.5 E3 Mobil klient

Den mobile klienten er avhengig av E3 Databasetjener og E2 Mobile støttestasjon for å prosessere de transaksjonene som klienten ønsker å utføre. Internt er E4.1 Transaksjonsgenerator avhengig av E4.2 XML parser for å konvertere XML dokumentene over til transaksjonsobjekter.

Kapittel 15

Grensesnitt

Dette kapittelet omhandler de eksterne grensesnittene som benyttes i implementasjonen av prototypen. I den eksterne programvaren brukes det to grensesnitt som er spesielt viktige og som vil bli beskrevet her. Disse retter seg mot JDBC og JavaSpaces.

15.1 JDBC grensesnittet

JDBC (Java DataBase Connectivity) er et Java API for å knytte Java programmer opp mot relasjonsdatabaser. Denne pakken er en del av standard Java installasjoner, og består av klassene i java.sql. Implementasjonen av dette grensesnittet tas hånd om av en JDBC driver, og i dette prosjektet brukes Connector/J som er den driveren som håndterer MySQL databaser. De metodene i grensesnittet som er viktige i designet er beskrevet under.

Connection

Connection grensesnittet tilknyttes en spesifikk database. Alle SQL uttrykk eksekveres innenfor konteksten til en slik connection.

- *commit()* - gjør alle endringer siden den siste commit/rollback permanent og frigir eventuelle låser som holdes.
- *Statement createStatement()* - lager et Statement objekt for å sende SQL uttrykk til databasen.
- *rollback()* - omgjør alle endringer gjort av den nåværende transaksjonen og frigir eventuelle låser som holdes.
- *setAutoCommit(boolean autoCommit)* - angir om autocommit skal brukes eller ikke.

Statement

Statement grensesnittet brukes for å eksekvere statiske SQL uttrykk og returnere de resultatene det produserer.

- *ResultSet executeQuery(String sql)* - eksekverer et gitt SQL uttrykk som returnerer et enkelt ResultSet objekt.
- *int executeUpdate(String sql)* - eksekverer et gitt SQL uttrykk som er av typen INSERT, UPDATE eller DELETE.

ResultSet

Dette grensesnittet representerer et sett rader som representerer resultatet fra en spørring mot databasen.

- *first()* - flytter pekeren til den første raden i ResultSet objektet.
- *next()* - flytter pekeren til den neste raden fra den nåværende posisjonen.
- *getXXX()* - forskjellige get metoder for å hente ut forskjellige datatyper. XXX kan for eksempel være int, float eller string.

15.2 JavaSpace grensesnittet

JavaSpace grensesnittet og implementasjonen er begge en del av Jini teknologien fra Sun Microsystems. Denne er ikke en del av standard Java installasjoner og må lastes ned separat.

Entry

Denne klassen er supertypen til alle objekter som skal kunne overføres til et JavaSpace. Grensesnittet inneholder ingen metoder, men spesifikasjonen krever at alle objekter som implementerer det må ha med en konstruktor som ikke tar noen argumenter for at de skal kunne brukes korrekt av et JavaSpace.

JavaSpace

Dette er grensesnittet som spesifiserer hvilke metoder en JavaSpace implementasjon må støtte.

- *Entry read(Entry templ, Transaction txn, long timeout)* - les ethvert matchende entry objekt fra JavaSpacet, og blokker til minst et slikt objekt er lest.
- *Entry take(Entry tmpl, Transaction txn, long timeout)* - ta ut ethvert matchende entry objekt fra JavaSpacet, og blokker til minst et slikt objekt er tatt ut.
- *Lease write(Entry entry, Transaction txn, long lease)* - skriv et nytt entry objekt til JavaSpacet.

Kapittel 16

Detaljert design

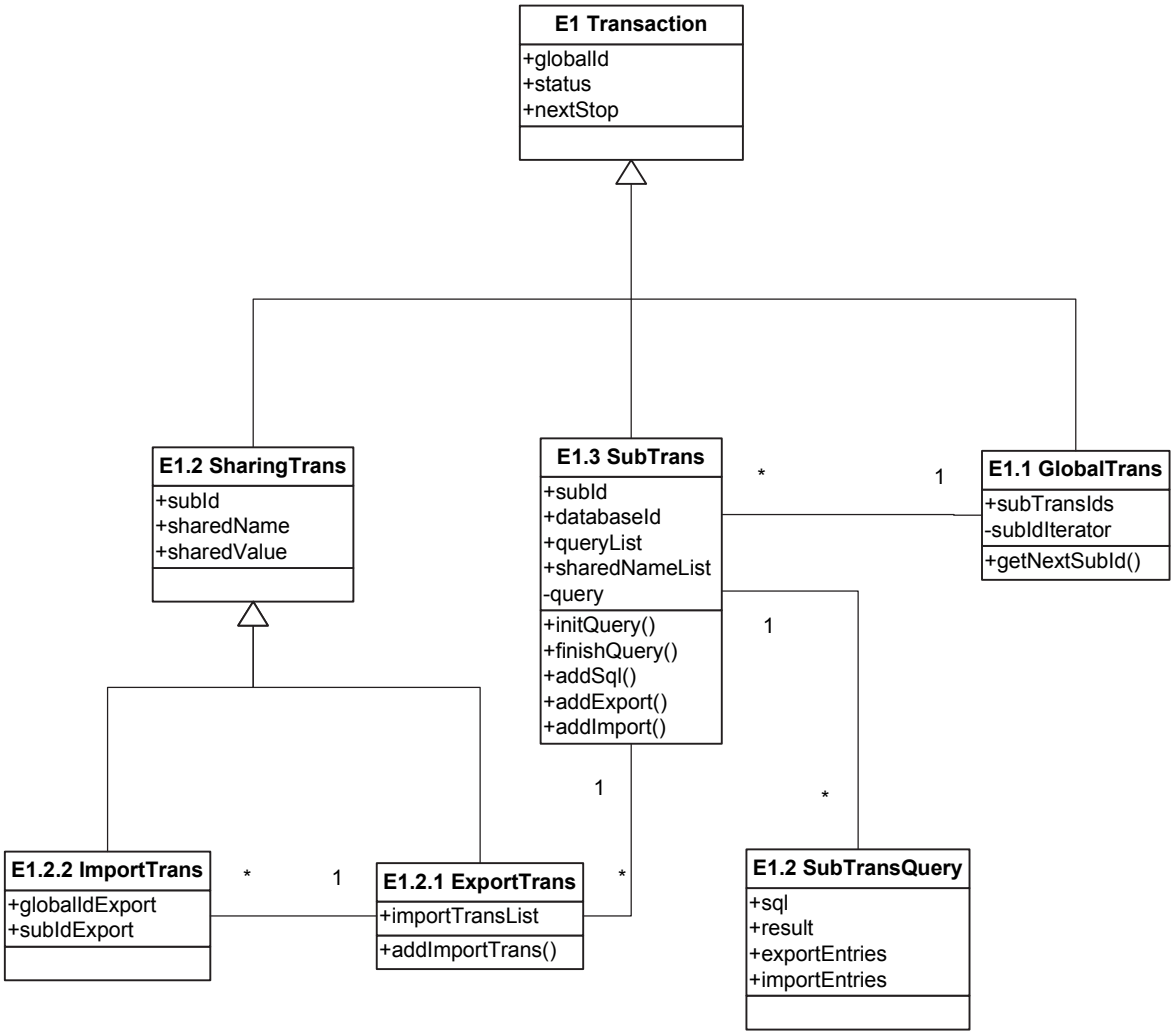
I dette kapittelet blir det presentert klassediagrammer som viser den detaljerte oppbygningen av systemet. Disse vil her bli beskrevet kort og konsist, for en mer omfattende forklaring av variabler og metoder henvises det til tillegg C. Entitetene vi kom fram til i kapittel 13 blir beskrevet i mer detalj her samt i tillegg C slik at det vil være mulig å konstruere en implementasjon med grunnlag i dette designet. Det er ikke gitt at hver entitet svarer til kun en klasse, enkelte entiteter trenger flere klasser for å bli komplette. Den programvaren vi ikke implementerer selv, men som likevel er en del av løsningen, har fått sine grensesnitt beskrevet i kapittel 15.

16.1 Transaksjonsklasser

Figur 16.1 viser de klassene som brukes for å representere de forskjellige typene av transaksjoner i systemet. På toppen av arvehierarkiet sitter *E1 Transaction* klassen, denne innholder variabler som er felles for alle typer transaksjoner. Imidlertid instansieres ikke denne klassen direkte, det er kun subklassene som blir instansiert i systemet. *E1.1 GlobalTrans* representerer en global transaksjon, og den inneholder en liste over alle identifikatorene til alle *E1.3 SubTrans* objektene som hører til under den. *E1.2 SharingTrans* instansieres heller ikke direkte, men den inneholder variabler som brukes av både *E1.2.1 ExportTrans* og *E1.2.2 ImportTrans*. *E1.3 SubTransQuery* brukes av *E1.3 SubTrans* for å representere alle spørringene som en subtransaksjon har.

16.2 Klient, tjener og koordinatorklasser

I Figur 16.2 vises det en oversikt over de klassene som utgjør klienten, koordinatoren og databasetjeneren. Sentralt i diagrammet ligger *E2.2 TransMod*, og denne klassen tilbyr metoder for å lese og skrive transaksjonsobjekter i JavaSpacet. Klienten består av *E4.1 MobileClient* og *E4.2 XMLtoEntryParser*, disse klassene er ansvarlig for å parse XML filer og lage transaksjonsobjekter som sendes til JavaSpacet. *E3.1 MobileHost*, *E3.1 SubTransThread* og *E3.1 QueryThread* utgjør databasetjeneren, og disse klassene har som oppgave å utføre spørringer i parallell mot

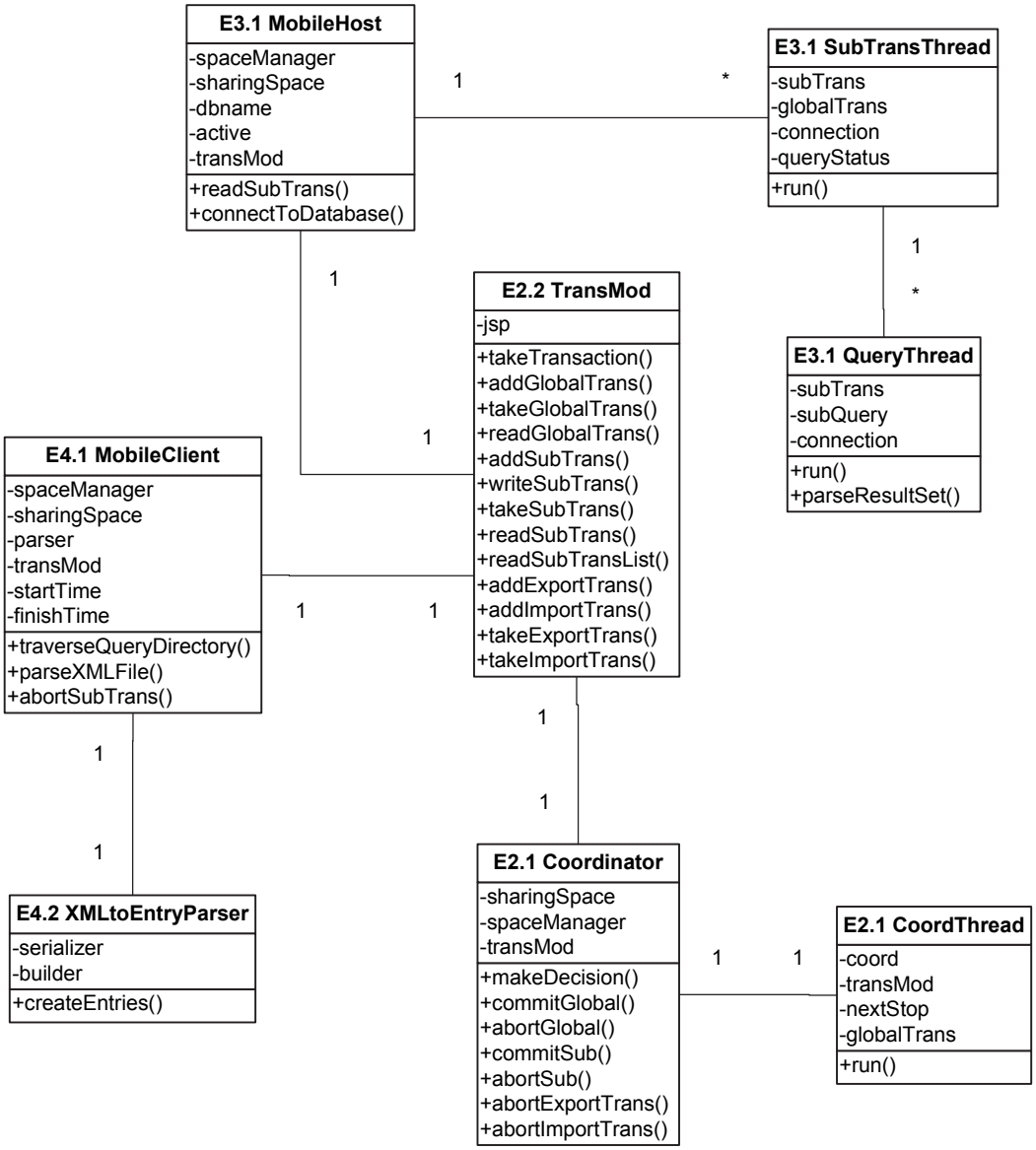


Figur 16.1: Klassediagram for transaksjonsklasser

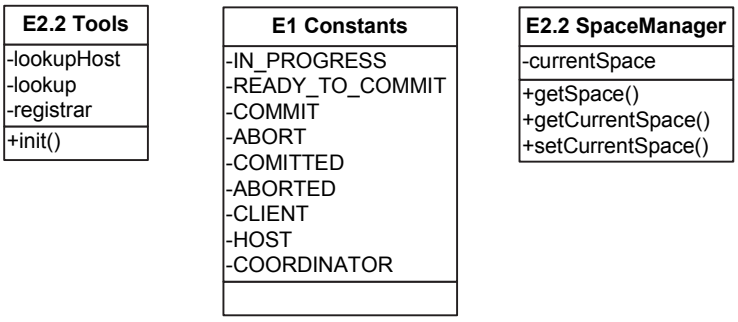
databasen og gi resultatene tilbake. Koordinatoren består av klassene *E2.1 Coordinator* og *E2.1 CoordThread*, og disse klassene henter inn transaksjonsobjekter fra JavaSpacet og endrer status til disse i forbindelse med commit/abort prosesseringen.

16.3 Hjelpeklasser

Figur 16.3 viser hjelpeklassene til systemet. *E2.2 Tools* har blant annet som oppgave å slå opp og finne en navnetjener, som *E2.2 SpaceManager* igjen bruker for å finne fram til JavaSpaces. *E1 Constants* inneholder en rekke konstanter som brukes av transaksjonene som verdier i statusfeltet og neste stopp feltet.



Figur 16.2: Klassediagram for klient, tjener og koordinator



Figur 16.3: Klassediagram for hjelpeklasser

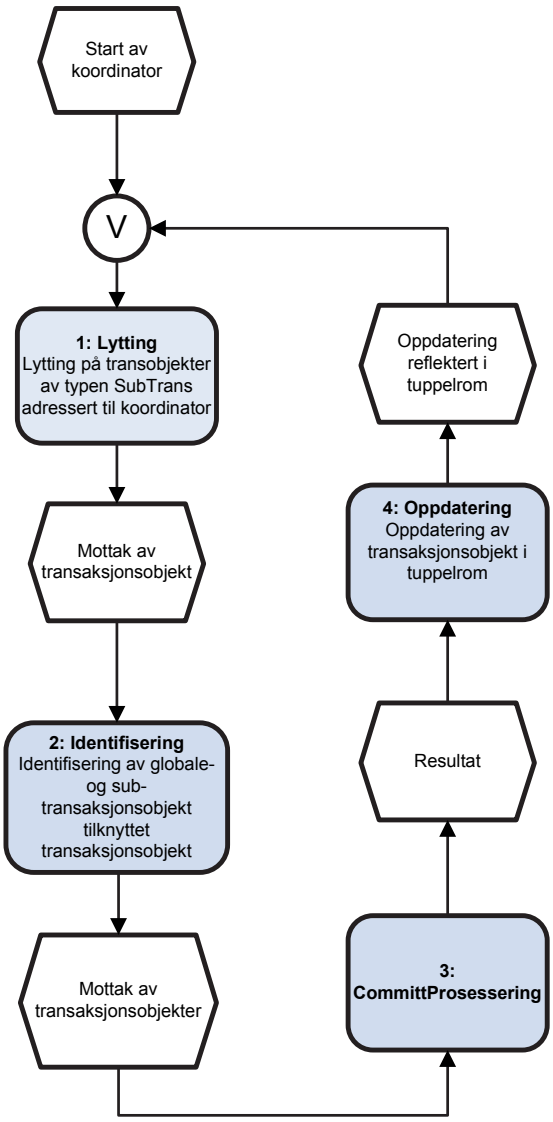
16.4 Prosessering i koordinatoren

Figur 16.4 gir en oversikt over de mest sentrale prosessene hos koordinatoren. Illustrasjonen er basert på Event-driven Process Chains [1], et *petri-nett* lignende modelleringsspråk som fokuserer på målorientert dataflyt.

Med utgangspunkt i nevnte figur vil det interne prosesseringen til E2.1 Koordinator bli beskrevet, samt at dette vil bli satt i forhold til hvordan den interakterer med E2.2 Tuppelrom.

Straks koordinatoren har startet setter den i gang å lytte på transaksjonsobjekter som er adressert til seg selv. De viktigste prosessene vil bli beskrevet under:

- **1 Lytting:** Når koordinatoren er startet lytter den hele tiden på transaksjonsobjekter som er adressert til seg selv. Disse objektene er av typen SubTrans, som beskrevet i seksjon 13.2. Når et objekt av denne typen blir blir oppdaget i tuppelrommet går koordinatoren videre til trinn 2 i sin prosessering.
- **2 Identifisering:** Når koordinatoren mottar et E1.3 SubTrans objekt skyldes dette at E3 Databasetjener har gjort endringer på transaksjonsobjektet (SubTrans), for så å adressere det til E2.1 Koordinator. Det første koordinatoren må gjøre er å identifisere den globale transaksjonen som subtransaksjonen tilhører under. SubTrans objektet har identifikatoren til sitt overordnete GlobalTrans objekt, og koordinatoren leser dermed inn GlobalTrans objektet. Når koordinatoren har GlobalTrans objektet kan den lese inn alle (utenom objektet den allerede har lest) tilhørende SubTrans objekter. Koordinatoren er nå klar for å gå i gang med selve commitprosesseringen.
- **3 CommitProsessering:** På bakgrunn av de innleste transaksjonsobjektene må koordinatoren ta en avgjørelse for den videre utførelsen av transaksjonene. Avgjørelsen tas på grunnlag av avhengighetsforholdet mellom de involverte transaksjonene og av de gjeldende commitreglene. Denne prosessen vil bli dekomponert og beskrevet grundigere i 16.4.1. Når et resultat foreligger, gjenstår det bare å oppdatere de gjeldende transaksjonsobjektene i tuppelrommet.
- **4 Oppdatering:** Når koordinatoren er ferdig med å ta en avgjørelse for den videre utførelsen oppdaterer den transaksjonsobjektene lokalt for så skrive dem til tuppelrommet med riktig adressator.



Figur 16.4: Oversikt over sentrale prosesser hos koordinator

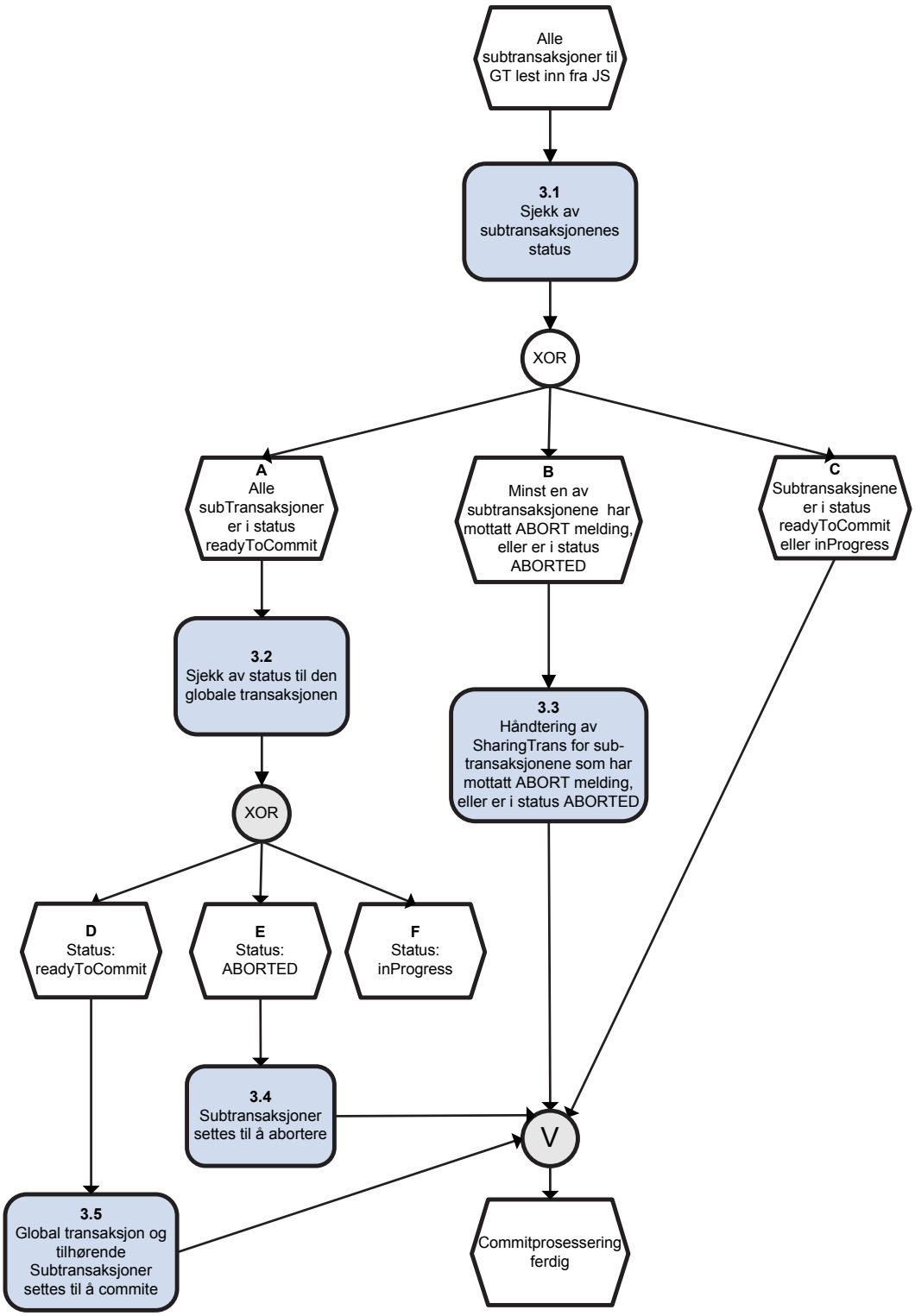
16.4.1 Dekomponering av prosess 3: CommitProsessering

Her kommer en mer detaljert beskrivelse av de interne avhengighetsforholdene i prosess 3: *CommitProsessering*. Figur 16.5 viser hvilke subprosesser prosess 3 består av, og disse vil bli beskrevet under.

- **3.1 Sjekk av subtransaksjonenes status:** Når koordinatoren har lest inn alle de tilhørende subtransaksjonene til en global transaksjon setter den i gang med å sjekke status til disse. Det er definert tre distinkte sammensetninger av statusfelter for av denne sjekken, nemlig:
  - **A:** Alle subtransaksjonene er i status **readyToCommit**.
  - **B:** Minst en av subtransaksjonen er i status **abort** eller i status **aborted**.
  - **C:** Minst en av subtransaksjonene er i status **inProgress**, og ingen subtransaksjoner har status lik **abort** eller **aborted**.

Illustrasjonen viser at kun et av de mulige utfallene skjer om gangen (XOR operator). Utfallet er avgjørende for den videre prosesseringen.

- **3.2 Sjekk av status til den globale transaksjonen:** Koordinatoren sjekker her statusen til den globale transaksjonen, som kan være i følgende tilstander.
  - **D**: Status: **readyToCommit** Klienten er her ferdig med å initiere subtransaksjoner og er klar for å committe.
  - **E**: Status: **inProgress** Klienten er ikke ferdig med å initiere subtransaksjoner og er derfor ikke klar til å kunne utføre en commit.
  - **F**: Status: **aborted** Klienten ønsker å avbryte den globale transaksjonen, eller den har feilet av andre grunner.
- **3.3 Håndtering av E1.2 SharingTrans:** Når en subtransaksjon får beskjed om å abortere sjekkes det om den har en SharingTrans tilknyttet seg. I såfall så må disse behandles etter de reglene som gjelder for dette. Denne prosessen vil videre bli dekomponert og forklart i detalj i 16.4.2.
- **3.4 Subtransaksjoner settes til å abortere:** Subtransaksjonene aborteres og adresseres til databasetjener.
- **3.5 Oppdatering av transaksjonsobjekter:** Koordinatoren oppdaterer transaksjonsobjektet til både den globale transaksjonen og de tilhørende subtransaksjonene til status lik **commit**, samtidig som de blir adressert til henholdsvis E4 Mobil klient og E4 Databasetjener.



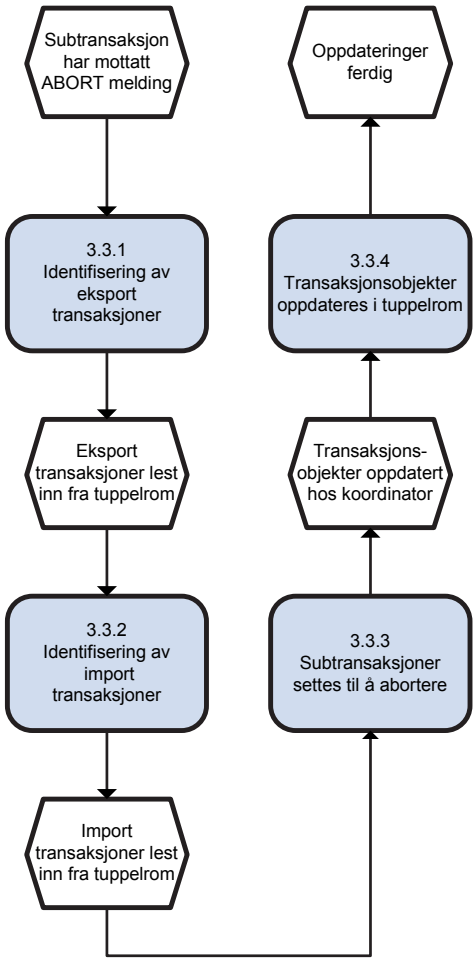
Figur 16.5: Dekomponering av prosess 3

16.4.2 Dekomponering av prosess 3.3: Håndtering av E1.2 SharingTrans

Håndtering av SharingTrans objekter, det vil si transaksjoner som deler verdier, vil bli her bli dekomponert og beskrevet i detalj. Prosessen er illustrert i 16.6, og den er essensiell for hvor-

dan man skal håndtere import- og eksport-transaksjoner når en subtransaksjon må abortere. På bakgrunn av at en subtransaksjon har mottatt en **abort** melding må koordinatoren ta seg av følgende:

- **3.3.1 Identifisering av eksporttransaksjoner:** En subtransaksjon kan ha eksportert en verdi til delingsrommet, så det er derfor viktig å identifisere eventuelle eksporttransaksjoner som er tilknyttet den gjeldende subtransaksjonen.
- **3.3.2 Identifisering av importtransaksjoner:** Transaksjoner som har lest verdier fra delingsrommet, har initiert en importtransaksjon. Med utgangspunkt i en eksporttransaksjon kan man identifisere alle importtransaksjoner som har lest verdien til eksporttransaksjonen. Dette skyldes at selve verdien nå er ugyldig og alle transaksjoner som har lest denne må derfor aborteres, ellers risikerer man inkonsistens i databasen.
- **3.3.3 Abort av subtransaksjoner:** Alle subtransaksjoner som har lest en ugyldig verdi settes til å abortere.
- **3.3.4 Oppdateringer i tuppelrommet:** De berørte transaksjonsobjektene oppdateres av koordinatoren og skrives ut til tuppelrommet.



Figur 16.6: Dekomponering av prosess 3.3

Kapittel 17

Oppsummering

I designdokumentet har systemet blitt partisjonert i mindre entiteter. Hver slik entitet har blitt nummerert og beskrevet i stor detalj. I tillegg har avhengigheter mellom entitetene og eksterne grensesnitt blitt avklart. På grunnlag av dette har implementasjonen blitt gjennomført, og denne implementasjonen er temaet i neste del av rapporten. Imidlertid implementeres ikke det grafiske brukergrensesnittet til klienten, siden dette ikke er essensielt for prototypens formål.

Del V

Implementasjon

Innledning

I implementasjonsdokumentet skal det gis en oversikt over hvilken programvare vi har brukt til å implementere systemet, samt en oversikt over hvordan de forskjellige entitetene har blitt implementert. De spesielle utfordringene vi har støtt på under utviklingen vil bli beskrevet, og en liste over hvilke krav som har blitt implementert vil bli presentert. Til sist vil måten man setter opp systemet på gjennomgås.

- Kapittel 18, Verktøy og programmeringsspråk, gir en kort oppsummering over hvilke verktøy og APIer som har blitt brukt i implementasjonen.
- Kapittel 19, Beskrivelse av implementasjonen, gir en oversikt over hvordan hver hovedentitet har blitt implementert.
- Kapittel 20, Utfordringer i implementasjonen, forklarer hvilke spesielle utfordringer vi har støtt på under utviklingen.
- Kapittel 21, Implementerte krav, lister opp alle kravene fra kravspesifikasjonen og angir om de har blitt implementert eller ikke.
- Kapittel 22, Oppsett av systemet, beskriver hvordan systemet konfigureres og gjøres klart for kjøring.

Kapittel 18

Verktøy og programmeringsspråk

Implementasjonen er blitt produsert med programmeringsspråket Java (v1.4), og de APIene som har blitt brukt er beskrevet nedenfor:

- **Jini v2.0** - De tjenestene som benyttes er Lookup Service (reggie) og JavaSpaces Service (outrigger). Disse tjenestene gjør systemet i stand til å gjøre navneoppslag for å finne JavaSpaces, samt å bruke disse til å utveksle transaksjoner.

  Programvare tilgjengelig på: *http://www.jini.org/downloads/*

- **Xerces2 Java Parser** - Brukes for å parse XML dokumenter.

  Programvare tilgjengelig på: *http://xml.apache.org/xerces2-j/*

- **Connector/J** - JDBC driver som brukes for å utføre operasjoner mot MySQL databasen.

  Programvare tilgjengelig på: *http://dev.mysql.com/doc/mysql/en/java-connector.html*

Kapittel 19

Beskrivelse av implementasjonen

I dette kapittelet blir det presentert en oversikt over implementasjonen av prototypen på transaksjonssystemet. Oversikten er delt inn i de fire delene E1 Transaksjon, E2 Mobil støttestasjon, E3 Databasetjener og E4 Mobil klient. Denne inndelingen er beskrevet nærmere i kapittel 13, Overordnet systemarkitektur. Kildekoden for systemet er lagt ved i tillegg D, og dokumentasjonen til denne kildekoden er lagt ved i tillegg C.

19.1 E1 Transaksjon

Denne entiteten er implementert ved hjelp av klassene Transaction, GlobalTrans, SubTrans, SharingTrans, ExportTrans og ImportTrans. Superklassen er Transaction, og de andre klassene inngår i et arvehierarki under denne. Arvehierarkiet er vist i figur 16.1. Transaction implementerer interfacet net.jini.core.entry.Entry, dette er nødvendig for at transaksjonsklassene skal kunne utveksles via JavaSpaces. Et annet krav er at hver transaksjonsklasse er nødt til å inneholde en tom konstruktor. Dette er påkrevet for å få riktig serialisering og korrekt assosiativt oppslag av Entry objektene.

19.2 E2 Mobil støttestasjon

Den mobile støttestasjonen består av klassene Coordinator, CoordThread og TransMod. I tillegg inngår Jini Lookup Service og Jini JavaSpace Service i denne entiteten. Støttestasjonen kjører først og fremst et JavaSpace der transaksjonsobjekter kan utveksles. Det er også mulig dele opp dette slik at man kjører et eller flere JavaSpaces ekstra som kun tar for seg deling av verdier, men det har vi ikke tatt med i vår implementasjon. Alle utvekslinger foregår i det samme JavaSpacet.

For å få tilgang til JavaSpacet må det registreres som en Jini service, og deretter kan andre deltakere i systemet gjøre oppslag og finne fram til det. Dette gjøres av en Jini Lookup Service, som må kjøres kontinuerlig av den mobile støttestasjonen. For å finne denne navnetjenesten kan man enten bruke en statisk adresse som er kjent for alle deltakerne, eller man kan bruke

multicast teknologi. Siden multicast teknologi ikke er tilgjengelig på store deler av internett slik det er i dag, har vi valgt å bruke en statisk referanse til navnetjenesten.

For å få hentet ut eller skrevet inn transaksjoner til JavaSpacet går man via TransMod klassen, som har implementert alle metodene for dette. Coordinatoren har som oppgave å ta avgjørelser i forbindelse med commit/abort prosessering av transaksjonene, og den henter transaksjoner som er klare for dette fra JavaSpacet.

19.3 E3 Databasetjener

Databasetjeneren er implementert ved hjelp av klassene MobileHost, SubTransThread og QueryThread. I tillegg bruker den en MySQL databasetjener som er installert på mysql.stud.ntnu.no. Her er det opprettet to databaser for å teste systemet, og for å få tilgang til denne basen brukes JDBC APIet. For hver subtransaksjon tjeneren mottar startes det en ny tråd som begynner å utføre operasjoner mot databasen. Hvis det er flere operasjoner får alle disse sin egen tråd. Slik kan tjeneren håndtere mange spørringer på en gang. Databasetabellene bruker MySQLs InnoDB engine, og autocommit er satt til false. Dermed kan det utføres spørringer mot databasen som ikke committes med en gang, men som venter på svar fra koordinatoren som avgjør om alt skal committes eller rulles tilbake.

19.4 E4 Mobil klient

Denne entiteten består av klassene MobileClient og XMLtoEntryParser. For å parse XML dokumenter og konvertere disse til transaksjonsobjekter brukes Xerces2 XML parseren. Klienten leser inn en rekke transaksjoner spesifisert i XML format fra fil og overfører disse til JavaSpacet via TransMod. Deretter venter den på svar fra transaksjonene og viser resultatet av kjøringen på skjermen.

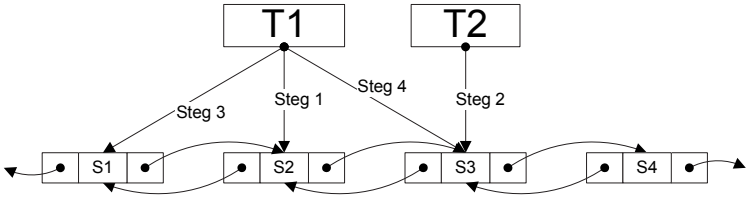
Kapittel 20

Utfordringer i implementasjonen

Implementasjonen bød på en rekke utfordringer, og de største problemene var distribuert deadlock, race conditions og distribuerte events. Disse er beskrevet nedenfor.

20.1 Distribuert deadlock

Det største problemet var distribuert deadlock siden dette kan være vanskelig å forutse og det er svært ødeleggende for systemet. Problemet var relatert til at det ikke er mulig å iterere over alle objekter av en type i JavaSpacet. Siden JavaSpaces bruker assossiativt oppslag der et objekt hentes ut basert på om de matcher en liste av kriterier, vil man få ut det samme objektet neste gang hvis man utfører leseoperasjonen en gang til med de samme kriteriene. Dette skapte problemer når alle subtransaksjonene som hørte til en global transaksjon skulle hentes ut. Riktignok hadde alle subtransaksjonene en unik subId innenfor den globale transaksjonen, men denne var ikke kjent for alle entitetene i systemet. Den første løsningen baserte seg på en distribuert lenket liste der alle subtransaksjonene hadde en forover- og bakoverpeker slik at alle subtransaksjonene kunne hentes ut. Dessverre viste det seg at denne strukturen var svært utsatt for distribuert deadlock, og den måtte kastes etter testing.



Figur 20.1: Distribuert deadlock i en lenket liste

Figur 20.1 viser en situasjon der en deadlock har slått til mellom to deltakere T1 og T2 som henholdsvis prøver å slette subtransaksjon S2 og S3. T1 henter først ut subtransaksjon S2 i steg 1. Deretter henter T2 ut S3 i steg 2. T1 må hente ut de to elementene som er foran og bak S2 for å oppdatere lenkene, og dermed hentes S1 ut i steg 3 og S3 blir forsøkt hentet ut i steg 4. Imidlertid har T2 allerede hentet ut S3, og den venter på å få tilgang til S2 for å oppdatere lenkene til denne subtransaksjonen. Dermed har vi en syklisk ventegraf der T1 venter på T2 og

T2 venter på T1. Denne situasjonen kunne blitt tatt hånd om ved hjelp av en form for distribuert transaksjonskontroll, men dette ville ha blitt for kostbart både med tanke på kompleksitet i implementasjonen og tidsbruk under kjøring.

Deretter fulgte den løsningen som brukes i systemet nå, der hver globale transaksjon vedlikeholder en liste over identifikatorene til hver subtransaksjon som er tilknyttet den. I tillegg har hver transaksjon et felt som angir hvilken funksjonell enhet (koordinator, klient eller tjener) som skal få tilgang til transaksjonen.

20.2 Race conditions

Race conditions oppstår når to eller flere prosesser prøver å aksessere dataelementer samtidig, og timingen av disse aksessene har uventet resultat på utfallet av kjøringen. Dette skjedde i systemet når flere deltakere prøvde å få tilgang på et objekt via de samme kriteriene. En av løsningene gikk ut på at den globale transaksjonen skulle hentes ut av databasetjeneren, og status skulle oppdateres der etterhvert som subtransaksjoner ble klar til å committe eller aborterte. Dette førte til en race condition mellom tjeneren og koordinatoren, som da ikke klarte å utføre commit/abort prosessering på en korrekt måte. For å løse dette ble det klart at alle transaksjonsobjektene måtte følge en spesifisert sti gjennom systemet fra start til slutt og at det måtte angis hvilken funksjonell enhet som var neste trinn i denne prosessen.

20.3 Distribuerte events

Tidlig i implementasjonen ble det brukt et system av distribuerte events som baserte seg på den tradisjonelle eventmodellen i Java med *event kilder*, *event lyttere* og *event objekter*. Men siden eventmodellen i Jini er distribuert, kan kilder og lyttere spres over mange forskjellige maskiner. Man kan da risikere at meldinger blir duplisert, at meldinger kommer i feil rekkefølge eller at meldinger ikke kommer fram i det hele tatt.

I arbeidet med å få deltakerne til å reagere på events i JavaSpacet, kom det fram at disse feilkildene var så absolutt tilstede. Hvis det var mange events som skjedde tilnærmet samtidig, ble eventmodellen lett overbelastet. I tillegg ble event-registreringer liggende latent i JavaSpacet etter at en deltaker ble koblet fra, og dette skapte også problemer.

Den løsningen som blir brukt istedenfor distribuerte events baserer seg på tråder og blokkerende venting. Alle deltakere har en tråd som går i en løkke og utfører en leseoperasjon som blokkerer i et visst antall sekunder, og hvis ikke en transaksjon har blitt mottatt innen den tid, starter leseoperasjonen på nytt. Ved mottak av en transaksjon hos en deltaker startes det opp en ny tråd som får ansvaret for utførelsen av transaksjonen. Dette viste seg å være en mer robust løsning som tålte høy belastning og uventede frakoblinger på en tilfredsstillende måte.

Kapittel 21

Implementerte krav

Tabell 21.1 lister opp kravene fra kravspesifikasjonen i kapittel 11. I tillegg er det tatt med en kolonne som sier hvilken entitet som implementert kravet, og hvis kravet ikke er implementert er dette markert med en strek. Hvis kravet er testet er testnummeret til den respektive testen i

Tabell 21.1: Implementerte krav

Nr	Krav	Entitet	Test
	Transaksjoner		
K1	Sammensetning av globale transaksjoner	E1.1	TOF1-1
K2	Unik global transaksjonsid	E1	TOF1-1
K3	Statusfelt	E1	TOF1
K4	Destinasjonsfelt	E1	TOF1
	Eksporttransaksjoner		
K5	Initiering av eksporteringsprosessen	E1.2.1	TOF1-2
K6	Eksportstruktur	E1.2.1	TOF1-2
K7	Kobling til importtransaksjoner	E1.2.1	TOF1-4
	Importtransaksjoner		
K8	Initiering av importeringsprosessen	E1.2.2	TOF1-3
K9	Importstruktur	E1.2.2	TOF1-3
K10	Kobling til eksporttransaksjoner	E1.2.1-2	TOF1-4
	Mobil støttestasjon (MSS)		
K11	Transaksjonsrom	E2	TOS1
	Delingsrom		
K12	Dyamisk opprettelse av delingsrom	-	-
K13	Tilordning av delingsrom	-	-
K14	Splitting av delingsrom	-	-
K15	Forening av delingsrom	-	-
	Koordinator		
K16	Bestemmelse av utfall til transaksjoner	E2.1	TOF2
K17	Global commit	E2.1	TOF2-1

Fortsetter på neste side

Tabell 21.1 – fortsettelse fra forrige side

Nr	Krav	Entitet	Test
K18	Global abort	E2.1	TOF2-2
K19	Delvis abort	E2.1	TOF2-3
K20	Maksimal tid i transaksjonsrommet	-	-
K21	Tilpassing av regelsett	-	-
Databasetjener			
K22	Unik navngiving av databaser	E3	TOF3
Transaksjonshåndterer			
K23	Tilbakemelinger fra transaksjonene	E3.1	TOF3-1
K24	Skriving av transaksjonsresultat	E3.1	TOF3-1
K25	Eksekvering av subtransaksjoner	E3.1	TOF3-2
K26	Commit etter brudd på nettverksforbindelsen	E3.1	-
K27	Commit og abort prosessering	E3.1	TOF3
K28	Tilbakeskriving av subtransaksjon	E3.1	TOF3
DBMS			
K29	Spesifisering av commit tidspunkt	E3.2	TOF3
Mobil klient			
K30	Innverdier fra brukeren	E4	-
K31	Visning av resultater fra transaksjoner	E4	TOF4-4
Transaksjonsgenerator			
K32	Innlesing av transaksjoner fra fil	E4.1	TOF4-2
K33	Skriving av transaksjoner	E4.1	TOF4-3
XML parser			
K34	Korrekthet av XML dokumenter	E4.2	TOF4
K35	Konstruksjon av trans.obj. fra XML dok.	E4.2	TOF4
GUI			
K36	Spesifisering av transaksjoner interaktivt	E4.3	-
Tabelloversikt over kravene			

Kapittel 22

Oppsett av systemet

Dette kapittelet beskriver hvordan systemet settes opp og konfigureres. Alle deltakerne er nødt til å ha Java v1.4 og Jini v2.0 installert, og i tillegg må klienten ha Xerces2 parseren installert og databasetjeneren må ha Connector/J installert. De forskjellige scriptene som brukes for å starte opp de forskjellige tjenestene er beskrevet under.

22.1 Konfigurasjonsfiler

For å kjøre navnetjeneren (Jini Lookup Service) må man kjøre følgende kommando:

```
java -Djava.security.policy=policy.all -jar m:\jini2_0_002\lib\start.jar ^
start-transient-reggie.config
```

Her viser `m:\jini2_0_002\lib\start.jar` til stedet man har installert Jini APIet. For å kjøre navnetjeneren trenger man to konfigurasjonsfiler, disse heter *start-transient-reggie.config* og *transient-reggie.config*. Disse ligger under diplom2005/source/config/ mappen på den vedlagte CDen.

For å kjøre JavaSpacet må man kjøre følgende kommando:

```
java -Djava.security.policy=policy.all -jar m:\jini2_0_002\lib\start.jar ^
start-transient-outrigger.config
```

De to konfigurasjonsfilene som trengs, *start-transient-outrigger.config* og *transient-outrigger.config* ligger på samme sted som beskrevet over.

22.2 Kjøring av systemet

For å teste systemet ved å kjøre et sett av transaksjoner som er spesifisert på forhånd kan man starte opp en klient, to tjenere og en koordinator. JavaSpace Service, Lookup Service og web-tjener med Jini APIet kjører for tiden på våre maskiner på arbeidskontoret på NTNU, men disse vil sannsynligvis tas ned i løpet av høsten 2005. Hvis disse ikke er operative må de startes opp igjen som beskrevet i avsnittet over før systemet kan kjøres.

For å starte tjener nummer en utføres følgende kommando:

```
java -Djava.security.policy=policy.all MobileHost Fly
```

For å starte tjener nummer to utføres følgende kommando:

```
java -Djava.security.policy=policy.all MobileHost Bil
```

Dette starter to databasetjenere, en som har ansvar for databasen med navn Fly, og den andre med ansvar for dababasen med navn Bil.

For å starte koordinatoren utføres følgende kommando:

```
java -Djava.security.policy=policy.all Coordinator
```

For å starte en klient utføres følgende kommando:

```
java -Djava.security.policy=policy.all MobileClient shared2
```

Dette vil føre til at klienten leser inn transaksjonene som er spesifisert på fil i katalogen shared2. Informasjon om transaksjonenes gang i systemet vil bli skrevet ut hos de respektive deltakerne.

Kapittel 23

Oppsummering

Prototypen som har blitt uviklet tilfredsstiller de fleste kravene i kravspesifikasjonen, untatt de som dreier seg om de dynamiske egenskapene til delingsrom samt tilpassing av regelsettet til koordinatoren. Disse kravene er dermed kandidater for videre arbeid. Vi støtte på en del problemer under implementasjonen, men vi fant måter å løse de på slik at prototypen fungerte som forventet til slutt. I neste del presenteres testene vi kjørte og resultatene av disse.

Del VI

Testing

Innledning

Sentralt i all programvareutvikling er å forsikre seg om at implementasjonen er i samsvar med hva er beskrevet i kravspesifikasjonen, og at den fungerer i henhold til hva som på forhånd er spesifisert. Dette dokumentet har som hensikt å beskrive tester og gjennomføringer av disse, slik at man kan være sikker på at protypen som er produsert fungerer som den skal, og at den er av høy kvalitet. Utgangspunktet for testingen er de ulike entitene som ble definert i designdokumentet i del IV, som hver for seg dekker et sett av krav. Testingen blir gjennomført ved å bruke en todelt modell, hvor henholdsvis delsystemene testes først, og systemet som helhet testes til slutt. Det er tatt hensyn til at utviklerteamet kun består av to diplomstudenter. Omfattende ansvarfordeling, feilkorrigeringsrutiner og tidsplaner, som ofte er vanlig i større prosjekter ved testgjennomføring er ikke tatt med i denne testgjennomføringen. Dokumentet består av følgende kapitler:

Kapittel 24, Testobjekter, tar for seg hvilke deler av systemet som skal testes og hva som ikke skal testes.

Kapittel 25, Testgjennomføring, beskriver selve testprosessen i form av hvordan man går frem for å utføre testene, hvilke typer tester som benyttes, samt hvilken hardware og software som trengs for å utføre testene.

Kapittel 26, Testresultater, presenterer samtlige tester på tabellform, med informasjon om hva testene går ut på, forventet resultat og hva det endelige resultatet av testen ble.

Kapittel 27, Oppsummering, gir en samlet oversikt og konklusjon for testresultatene.

Kapittel 24

Testobjekter

Dette kapittelet gir en oversikt over hva som skal testes, og hva som ikke skal testes. I designdokumentet i kapittel 13 ble systemet delt inn i tre hovedmoduler, samt en dataentitet (datastruktur). Hver entitet dekker et sett av krav, som spesifisert i kravdokumentet 11.1. I første del av testplanen vil det fokuseres på å utføre tester som bekrefter at funksjonaliteten til systemet er på plass i henhold til de kravene som er spesifisert. Entitene legger derfor grunnlaget for hva som skal testes, og er gjengitt i tabell 24.1. Når testene har bekreftet at funksjonaliteten til systemet er i henhold med hva som er beskrevet i kravdokumentasjonen, vil systemet som helhet bli testet. Først når disse testene er på plass vil det bli utført en ytelsestest og en frakoblingstest av systemet.

24.1 Hva som skal testes

Tabellen gir en enkel oversikt over delsystemene (modulene) som skal testes. Disse delsystemene kan videre dekomponeres i mindre entiteter, som illustret i kapittel 13.1. Hver enkelt delsystemtest tar høyde for å teste de enkelte underentitene. Dette går klart frem i kapittel 26 hvor testresultatene er dokumentert. Hver enkelt test har sin egen unike id. Denne er bestemt av hvilken type test som utføres. Eksempelvis så står **TOF** for *TestObjekt Funksjonalitet*, etterfulgt av nummeret på testen. Tilsvarende for **TOS**, **TOY** og **TOFK** som angir *TestObjekt System*, *TestObjekt Ytelse* og *TestObjekt FraKobling*

Spesifikasjonsnr.	**Beskrivelse**	**Designentitet**	**krav**
TOF 1	Transaksjon	E1	Se designdokument, kapittel 13.2
TOF2	Mobil Støttestasjon	E2	Se designdokument, kapittel 13.3
TOF3	Databasetjener	E3	Se designdokument, kapittel 13.4
TOF4	Mobil klient	E4	Se designdokument, kapittel 13.5
TOS1	System	E1-E4	Se designdokument, kapittel 13.1
TOY1	Ytelse	-	-
TOFK1	Frakobling	-	-

Tabell 24.1: Testobjekter

24.2 Hva som ikke skal testes

Alle delsystemene vil bli testet, men det finnes underentiteter av disse som ikke vil bli testet. Tabell 24.2 gir en oversikt over disse med en forklaring til hvorfor testing er utelatt.

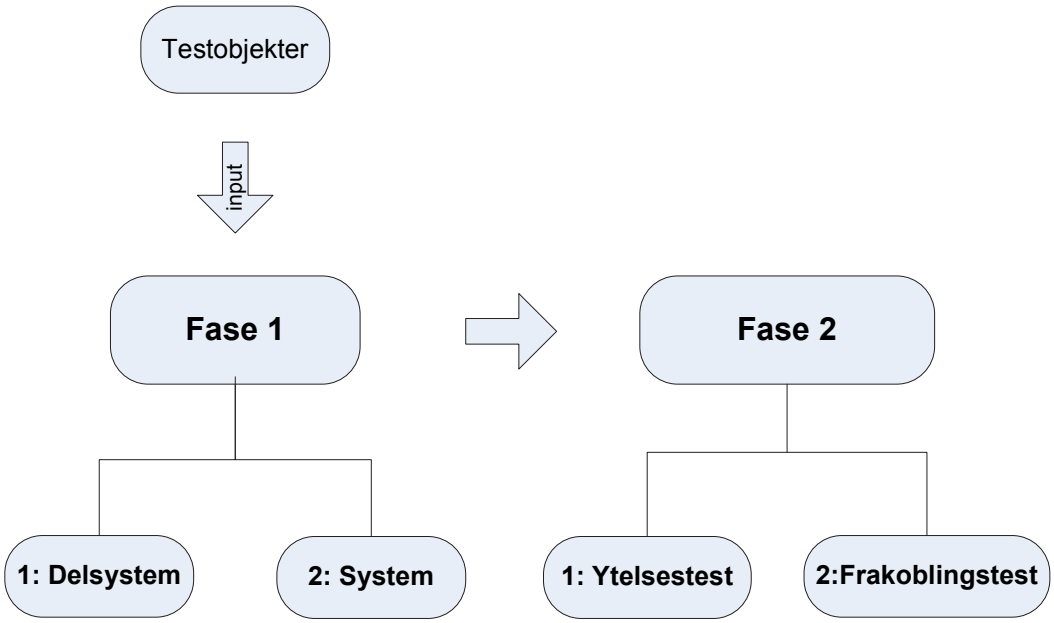
Spesifikasjonsnr.	**Beskrivelse**	**Årsak - ingen testing**
TOF2	E2.2 Tuppelrom	Tuppelrommet som er implementert med JavaSpace teknologi vil ikke bli testet, siden det forutsettes at Jini rammeverket er uten vesentlige feil.
TOF3	E3.2 MySQL database	MySQL databasen vil ikke bli testet, siden teknologien er såpass moden og det forutsettes at den fungerer korrekt.

Tabell 24.2: Entiter som ikke skal testes

Kapittel 25

Testgjennomføring

Dette kapittelet tar for seg hvordan testingen av protoypen vil bli utført. Målet er å organisere testene på en slik måte at selve testutførelsen vil foregå på mest mulig effektiv måte, samtidig som de skal være av så høy kvalitet at de avslører eventuelle feil og svakheter med systemet. Dette kapittelet vil hovedsakelig fokusere på hvordan testene vil bli utført, men også inneholde informasjon om hvilke type tester som skal kjøres. Til slutt vil testmiljøet bli beskrevet i form av hvilken *software* og *hardware* som trengs for å uføre testene.



Figur 25.1: Teststrategi

25.1 Testplan

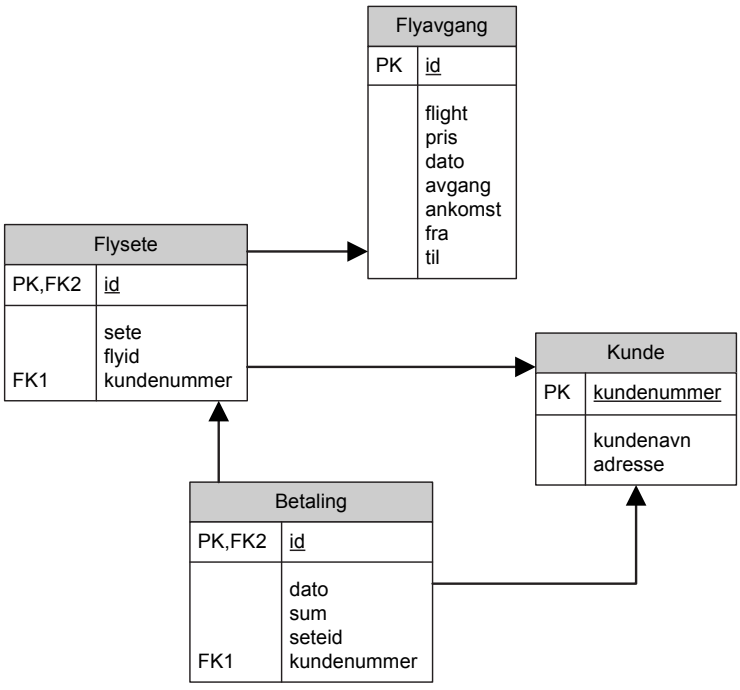
Det er viktig å velge en teststrategi som er tilpasset det systmet man skal teste. Vi har valgt å bruke en todelt modell, hvor vi først tester funksjonaliteten til systemet, som illustret i figur

25.1. Når denne fasen er over, vil resultatene bli evaluert og eventuelle feil vil bli korrigert. Videre beveger man seg over i siste fase. Denne fasen inneholder en ytelsetest og en egen frakoblingstest. For å få ubytte av sistnevnte tester er man naturligvis avhengig at systemet som helhet fungerer, noe testfase 1 som er omhandlet i neste avsnitt skal bekrefte.

25.1.1 Fase 1 - testing av funksjonalitet

Fase 1 er delt inn i henholdsvis en delsystemtest og en systemtest, som illustrert i figur 25.1. Målet med denne fasen er å forsikre seg om at man har et fungerende system. Med utgangspunkt i testobjektene som er definert i kapittel 24, vil det først bli foretatt en test for hver modul av systemet. For å kunne utføre en tilstrekkelig test av en modul er man avhengig av å teste denne med et sett av deltester. Hver enkelt testmodul testes, og hvis eventuelle feil oppstår retestes modulen helt til samtlige av de spesifiserte testene er i orden. Først da kan man starte å teste neste modul. Da disse testene skal bekrefte at hver enkelt modul er korrekt, vil testene bli kjørt på et lokalt system. De neste underkapitlene beskriver de ulike stegene under fase 1.

Figur 25.2 viser et ER-diagram over entitetene som inngår i testdatabasen (MySQL). Transaksjonene vil består av SQL spørringer utført på de aktuelle tabellene. Systemet som her brukes er et enkelt billettbestillingssystem hvor en *kunde* gjør en bestilling av et *flysete* på en bestemt *flyavgang*. Betaling angir selve bestillingen for kunden, hvor flysete for en bestemt flyavgang er registrert.



Figur 25.2: ER-diagram - testdatabase

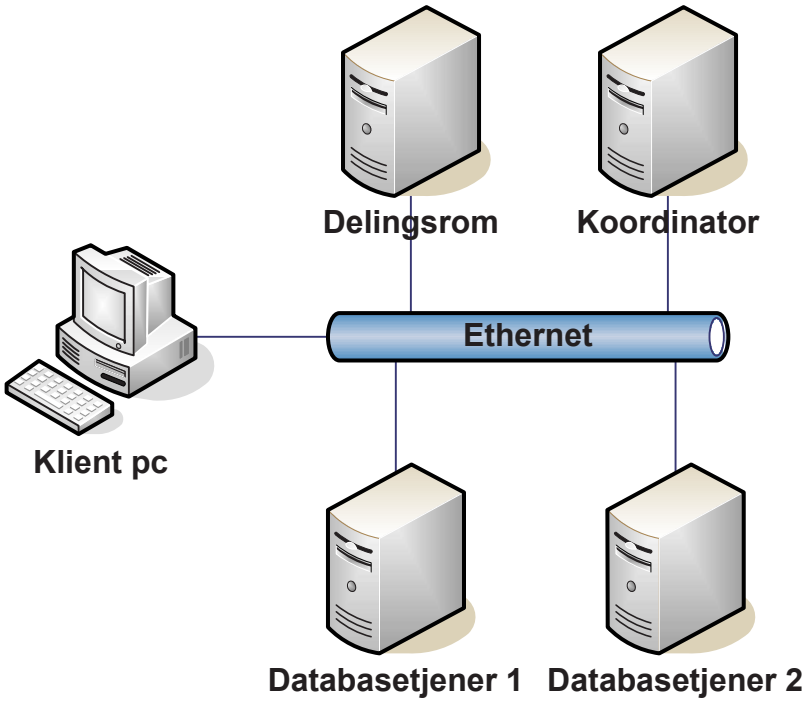
Steg 1: testing av delsystem

Hvert delsystem testes hver for seg. For hver enkelt modul er det spesifisert et sett av tester. Disse testene kan sees på som enhetstester, men vil ikke bli spesifisert som egne tester, men heller samlet utgjøre testen av selve modulen. Det er vanskelig å teste de ulike modulene hver for seg da det er sterk avhengighet mellom de ulike testene. Eksempelvis er E2.1 Koordinatoren avhengig av at E3 Databasetjener har gjort oppdateringer på et transaksjonsobjekt (E1). Transaksjonsentiteten (E1) vil bli testet i en egen test, men vil også inngå i alle de andre testene da utveksling av transaksjonsobjekter er helt essensielt for utførelse av transaksjoner. Følgende modultester er definert for systemet:

- **E1 Transaksjon**
  - Sammensetning av globale transaksjoner
  - Initiering av eksporteringsprosessen
  - Initiering av importeringsprosessen
  - Kobling mellom eksport- og import-transaksjoner
- **E2 Mobil støttestasjon**
  - Global commit
  - Global abort
  - Delvis abort
- **E3 Databasetjener**
  - Tilbakemelding fra transaksjonene
  - Eksekvering av subtransaksjoner
- **E4 Mobil klient**
  - Konstruksjon av transaksjonsobjekter fra XML dokumenter
  - Innlesing av transaksjoner fra fil
  - Skriving av transaksjoner
  - Visning av resultater fra transaksjoner

Steg 2: testing av system

Hensikten med denne testen skal bekrefte at systemet som helhet fungerer. Dette er avhengig av at alle modultestene fungerer og eventuelle feil må være korrigert før denne testen kan utføres. En større test, basert på modultestene, vil bli kjørt, hvor målet er å teste så mye av systemets funksjonalitet i en samlet gjennomkjøring. Spesielt viktig er det å sjekke om funksjonalitet på tvers av modulene er i orden.



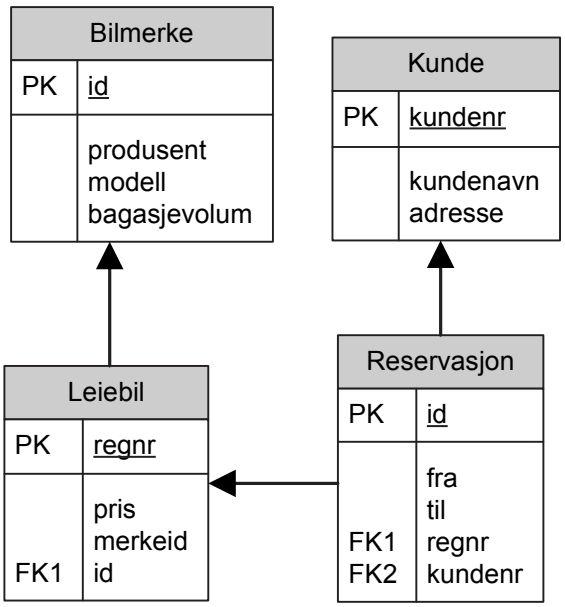
Figur 25.3: Testmiljø - systemtest

Når det gjelder bruk av hardware så vil selve testingen av systemet bli utført i et realistisk miljø, der alle deltakerne er distribuerte. Med dette menes at klienten vil kjøre på egen PC, og koordinator, databasetjenere og delingsrom vil alle være lokalisert på ulike maskiner, som illustrert i figur 25.3. Dessverre er det ikke mobile enheter tilgjengelig på NTNUs datasaler som vi kan bruke i testoppsettet, men i de tilfellene det er viktig å se hva som skjer ved en eventuell frakobling vil dette bli simulert ved å drepe prosesser. Forøvrig har de ulike hardwarekomponentene samme spesifikasjon som beskrevet i avsnitt 25.2.2.

I testen benyttes det to databaser, som begge er lokalisert på hver sin tjener:

- **Databasetjener 1 - flydatabase:** Videreføring av databasen fra avsnitt 25.1.1.
- **Databasetjener 2 - bildatabase:** Database for reservasjon av leiebiler, som illustrer i ER-diagrammet i figur 25.4. En reservasjon består av henholdsvis en *kunde* og en *leiebil*. Leiebilen er knyttet opp mot hvilket *bilmerke* det dreier seg om.

De overnevnte databasene legger grunnlaget fra spørringene som blir kjørt fra klienten. Først gjøres det en spørring mot flydatabasen for å gjøre en reservasjon av et flysete. Nøkkelverdier som *seteid*, *flyid*, *kundenummer*, og *dato* blir eksportert ut til delingsrommet. Disse verdiene blir så importert for å bli brukt i selve reservasjonen av leiebil fra bildatabasen. En mer detaljert beskrivelse av de ulike spørringene er spesifisert i XML-format i tillegg B.



Figur 25.4: ER-diagram, bildatabase

25.1.2 Fase 2 - ytelse og frakobling

Test 2 er delt inn i henholdsvis en ytelsestest og en frakoblingstest.

Ytelsestest

Målet med ytelsestesten er å undersøke hvor effektivt eksport-import modellen håndterer transaksjoner sammenlignet med en mer standard utførelse. Deling av verdier via et delingsrom gjennom bruk av import- og eksport-transaksjoner skal rent teoretisk gi en gevinst i form av økt gjennomstrømning av transaksjoner. Dette skyldes at isolasjonskravet er brutt og at transaksjoner dermed ikke trenger å vente før en annen transaksjon har committet før den kan lese eventuelle verdier som denne transaksjonen har oppdatert. Ytelsestesten er ment som en tilnærming på hvordan forskjellene kan fortone seg i et system med deling av verdier under utførelse sammenlignet med et system uten deling. Man kan ikke gi et riktig bilde av hva den reelle ytelsen vil være men en slik prototyp, men forhåpentligvis kan man gi noen svar om hva som kan forventes. Resultatene kan gi nyttig informasjon for videre diskusjon og drøfting modellens ytelse. Testen fungerer også som en utvidet systemtest, da den kan være med å bekrefte om prototypen oppfører seg som forventet. Sprikende resultater kan eksempelvis være et tegn på at protypen ikke fungerer som den skal.

Utgangspunktet for ytelsestesten er systemtesten som ble beskrevet i kapittel 25.1.1. Testmiljøet forblir også det samme. Ytelsestesten er delt inn i to gjennomføringer:

- Transaksjonell utførelse med bruk av eksport- og import-transaksjoner
- Sekvensiell kjøring av transaksjoner uten deling

Hver enkelt test kjøres med 10 iterasjoner, slik at en gjennomsnittsverdi kan finnes for hver enkelt test. På bakgrunn av dette regnes det ut hvor stor gjennomstrømningen av transaksjoner er for de ulike testene. Resultatet fra testen finnes i tabell 26.6 i kapittelet om testresultatene.

Frakoblingstest

Eksport-import modellen skal som kjent ha god støtte for frakobling. På bakgrunn av dette er det definert en test å sjekke hvordan systemet oppfører seg når ulike enheter kobles ut. Denne testen undersøker ikke konsistens for mobile transaksjoner når mobile klienter kobler fra, men sjekker mer om systemet er tilrettelagt for frakobling. Testen er ikke en komplett test som kan gi noe et endelig svar angående frakobling, men resultatene kan forhåpentligvis gi grunnlag for videre diskusjon i rapporten. Systemtesten er utgangspunktet for også denne testen. Følgende blir testet i frakoblingstesten:

- Frakobling av Koordinator:
- Frakobling av Klient
- Frakobling Javaspace
- Frakobling Databasetjener

Resultatet fra testen finnes i tabell 26.7 i kapittelet om testresultatene.

25.2 Testmiljø

Dette kapittelet inneholder informasjon om hvilken *software* og *hardware* som er nødvendig for å kunne gjennomføre de ulike testene av systemet. Nå kan det skje at de ulike testene krever forskjellig testmiljø. I såfall vil det bli spesifisert på testen om det er avvik fra spesifikasjonen her, som ansees som et standard testmiljø som inneholder de viktigste komponentene.

25.2.1 Software

Tabell 25.1 gir en oversikt programvare som må være installert for å kjøre testene.

Enhet	**Spesifikasjon**	**Beskrivelse**
Operativsystem	MS Windows XP	Operativsystemet har i utgangspunktet ingen betydning, men det finnes del .bat filer som starter opp deltakerne i systemet.
Utviklingsplattform	Java v1.4	For å kunne kjøre programvaren må Java være installert.
Mellomvare	Jini v2.0	Jini må være installert for å benytte seg av navnetjener, JavaSpaces, og så videre.
API	Xerces2 Java Parser	XML parseren må være installert for å lese XML dokumenter fra fil.
API	Connector/J	JDBC driveren for MySQL må være installert for å utføre operasjoner mot databasene.

Tabell 25.1: Software som er nødvendig for å utføre testene

25.2.2 Hardware

Tabell 25.2 gir en oversikt over hardware som er brukt for å utføre testene.

Enhet	Spesifikasjon	Beskrivelse
PC	Pentium 4 - 3,0 GHz, 512 MB RAM	PCene må være kraftige nok til å kunne kjøre Java og Windows XP.
Internet forbindelse	100 Mbit linje	Båndbredden må være såpass stor at utveksling av objekter mellom klient, delingsrom og tjener foregår uten for store forsinkelser.

Tabell 25.2: Hardware brukt for å utføre testene

Kapittel 26

Testresultater

Dette kapittelet inneholder resultatene fra de ulike testene som er blitt kjørt på systemet. Testene er spesifisert i egne tabeller, som inneholder informasjon om type test, hensikt med test, samt beskrivelse av selve testene og resultater. Resultatene vil bli diskutert og oppsummert i kap. 27.

26.1 Fase 1 - Funksjonalitetstesting

Testspesifikasjon for:		Delsystem, TOF3	
Enhet:		E3 Databasetjener	
Hensikt med test:		Testen skal vise om E3.1 Transaksjonshåndterer på E3 Databasetjener håndterer transaksjonene på korrekt måte.	
Beskrivelse av test			
I denne testen skal transaksjonshåndtereren hente ut subtransaksjonene fra transaksjonsrommet (som den har ansvar for) og kjøre disse mot den aktuelle databasen, for så oppdatere status for disse, og skrive dem tilbake til transaksjonsrommet. Transaksjonshåndtereren skal også returnere det endelige resultatet (commit eller abort) av en subtransaksjon til klienten. I testen vil det bli utført en spørring (INSERT) som gjør oppdateringer i E3.2 MySQL databasen. Når transaksjonen er kjørt sjekkes det om spørringen er kjørt i den lokale databasen.			
Deltest	**Hva må gjøres**	**Forventet resultat**	**Resultat**
1	Tilbakemelding fra transaksjonene	Transaksjonene skal gi beskjed til klienten om det endelige resultatet av transaksjonsutførelsen (commit, abort).	OK
2	Eksekvering av subtransaksjoner	Transaksjoner i form av SQL spørringer mot MySQL databasen utføres. Resultatet blir reflektert i databasen.	OK

Tabell 26.1: Testresultat - E3 Databasetjener

Testspesifikasjon for:	Delssytem
	TOF1
Enhet:	E1 Transaksjon
Hensikt med testen:	Målet med testen er å undersøke om systemet oppretter de ulike transaksjonstypene i henhold til hva som er spesifisert.
Kommentar:	Da E1 Transaksjon er en datastruktur er det vanskelig å utføre direkte tester da den i utgangspunktet ikke har noe adskilt funksjonalitet. E1 må derfor testes ved å b ruke modul E2 og E4.

Beskrivelse av test

Denne testen er basert på å undesøke om *input* er lik *output*, det vil si om de transaksjonene som blir opprettet er lik det som på forhånd er spesifisert. Testen er som følgende:

- Opprettelse av en global transaksjon med et sett subtransaksjoner.
- En av subtransaksjonene skal dele en verdi ved å initiere en *eksporttransaksjon*.
- En av subtransaksjonene skal importere den delte verdien ved å initiere en *importtransaksjon*.

Deltest	**Hva må gjøres**	**Forventet resultat**	**Resultat**
1	Sammensetning av globale transaksjoner	Den globale transaksjonen skal ha et sett av subtransaksjoner underordnet seg. Den globale transaksjonen skal ha en unik *id*. Subtransaksjonenes *id* er relativ til den globale transaksjonen.	OK
2	Initiering av eksporteringsprosessen	Verdien som subtransaksjonen deler blir tilgjengelig i delingsrommet via eksporttransaksjonen som opprettes.	OK
3	Initiering av importeringssprosessen	Subtransaksjonen som ønsker å få tilgang til den delte verdien initierer en importtransaksjon.	OK
4	Kobling mellom eksport- og import-transaksjoner	Eksport- og importtransaksjonene kobles sammen. Eksporttransaksjonen legger importtansaksjonen til importlisten.	OK

Tabell 26.2: Testresultat - E1 Transaksjon

Testspesifikasjon for:	Delsystem
	TOF2
Enhet:	E2 Mobil støttestasjon
Dato:	
Hensikt med test:	E2.1 Koordinator har som hovedfunksjonalitet å ta seg av transaksjonshåndteringen av de involverte transaksjonene i delingsrommet. Hensikten med denne testen er å sjekke om commitprosesseringen fungerer.
Kommentar:	Delingsrommet, dvs. javaspacet testes ikke utover det å støtte funksjonalitet til systemet.

Beskrivelse av test

Poenget med denne testen er å undersøke om koordinatoren er i stand til å utføre en riktig commitprosessering. Følgende subtester er definert for denne testen:

- **Global commit:** Det initieres en global transaksjon med et sett av tilhørende subtransaksjoner. De assosierte transaksjonsobjektene lagres i delingsrommet, og oppdateres til status *readyToCommit* og adresseres til koordinatoren.
- **Global abort:** Det initieres en global transaksjon med et sett av tilhørende subtransaksjoner.
- **Delvis abort:** Det initieres en global transaksjon med et sett av tilhørende subtransaksjoner. Status til det globale transaksjonsobjektet settes til *readyToCommit*. En av subtransaksjonene eksporterer så en verdi ut til delingsrommet som en annen subtransaksjon importerer. Den eksporterende transaksjonen aborterer i løpet av utførelsen.

Deltest	Hva må gjøres	Forventet resultat	Resultat
1	Global commit	Koordinatoren setter den globale transaksjonen med de tilhørende subtransaksjoner til å committe. Transaksjonsobjektene oppdateres og skrives ut til delingsrommet.	OK
2	Global abort	Koordinatoren setter den globale transaksjonen og de tilhørende subtransaksjoner til å abortere.	OK
3	Delvis abort	Koordinatoren aborterer eventuelle subtransaksjoner som har importert verdier fra den aborterende subtransaksjonen. De involverte transaksjonsobjektene oppdateres og skrives ut til delingsrommet.	OK

Tabell 26.3: Testresultat - E2 Mobil støttestasjon

Testspesifikasjon for:	Delsystem
	TOF4
Enhet:	E4 Mobil klient
Hensikt med testen:	E4.1 Transaksjonsgeneratoren har som oppgave å lese inn XML-dokumenter (spesifikasjon av transaksjonene) som E4.2 XML-parser lager transaksjonsobjekter av. Disse blir så lagret i transaksjonsrommet. Denne testen har som mål å undersøke om dette skjer på korrekt måte.
Kommentar:	

Beskrivelse av test
Først av alt må et sett transaksjoner defineres i et XML- dokument. På bakgrunn av denne genereres det transaksjonsobjekter som skrives til transaksjonsrommet. Transaksjonene utføres mot de aktuelle databasene. Klienten skal få tilbakemelding om hvordan kjøringene gikk, samt få returnert eventuelle verdier fra spørringen.

Deltest	**Hva må gjøres**	**Forventet resultat**	**Resultat**
1	Konstruksjon av transaksjonsobjekter fra XML dokumenter	Parseren konstruerer transaksjonsobjekter på bakgrunn av XML-dokumentene. Transaksjonsobjektene skal ha de egenskapene som er spesifisert i XML spesifikasjonen.	ikke OK, retest: OK
2	Innlesing av transaksjoner fra fil	Ingen feil skal oppstå under innlesingsprosessen.	ikke OK, retest: OK
3	Skriving av transaksjoner	Transaksjonsobjektene som parseren returnerer skrives til transaksjonsrommet.	OK
4	Visning av resultater fra transaksjoner	Resultatet fra spørringen skrives i delingsrommet på XML form. Klienten skal så hente ut dette.	OK

Tabell 26.4: Testresultat - E4 Mobil klient

Testspesifikasjon for:	System, TOS1
Enhet:	System
Hensikt med testen:	Målet med testen er å undersøke om systemet som helhet fungerer som det skal. I delsystemtestene prøver man å sjekke om de ulike modulene hver for seg fungerte skikkelig. Systemtesten går et steg videre og prøver å avsløre eventuelle svakheter i interaksjonen mellom alle modulene.
Kommentar:	E2.1 Koordinator , E2.2 Delingsrom og E3 Databasetjenere kjører på egne tjenere.

Beskrivelse av test			
En større test, bestående en global transaksjon med et sett av subtransaksjoner i form av SQL spørringer blir kjørt. Transaksjonene skal utføre en reservasjon av en flysete på en flyavgang, samt bestille leiebil. Verdier blir eksportert og importert fra delingsrommet. En mer detaljert beskrivelse av testen finnes i kapittel 25.1.1 og selve testen er spesifisert på XML-format i tillegg B. Testen er for oversiktlighetens skyld illustrert med 4 deltester, men testen i utgangspunktet en stor test, kjørt i en omgang.			
Deltest	**Hva må gjøres**	**Forventet resultat**	**Resultat**
1	Konstruksjon av transaksjoner, visning av resultater, mobil klient	Se tabell 26.4, test TOF4.	OK
2	Opprettelse av transaksjonsobjekter (Global, sub, eksport, import) i transaksjonsrom.	Se tabell 26.2, test TOF1.	OK
3	Transaksjonshåndtering, databasetjener	Se tabell 26.1, test TOF3.	OK
4	Commitprosessering, koordinator	Se tabell 26.3, test TOF2.	OK

Tabell 26.5: Testresultat - System

26.2 Fase 2

26.2.1 Ytelsestest

Testspesifikasjon for:	Ytelse
	TOY1
Enhet:	System
Hensikt med testen:	Målet er å gjøre en sammeligning mellom transaksjonshåndtering basert på eksport-import modellen og en sekvensiell utførelse for å undersøke hvor store forskjellene er med hensyn på ytelse.

Beskrivelse av test	
Ytelsestesten er basert på systemtesten, som er beskrevet i tabell 26.6. Denne testen blir utføre med 10 iterasjoner for transaksjonshåndtering basert på deling og med sekvensiell utførelse. Transaksjonene som utføres er svært tunge, så det er forventet at antall transaksjoner per sekund vil bli lavt. Nøkkeldata for testen: • Antall iterasjoner, n = 10 • Antall transaksjoner pr. kjøring, k = 6	
Sekvensielle transaksjonshåndtering	**Transaksjonshåndtering med deling**
1: *16203 ms* 2: *14397 ms* 3: *15092 ms* 4: *14532 ms* 5: *14349 ms* 6: *14220 ms* 7: *15847 ms* 8: *15087 ms* 9: *11821 ms* 10:*10753 ms*	1: *6602 ms* 2: *6196 ms* 3: *6929 ms* 4: *7379 ms* 5: *6820 ms* 6: *6378 ms* 7: *6873 ms* 8: *6809 ms* 9: *8472 ms* 10:*6055 ms*
Sum: *142301 ms*	Sum: *68513 ms*
Gjennomsnittstid pr. kjøring: 142301 ms / 10 = **14230 ms**	Gjennomsnittstid pr. kjøring: 68513 ms / 10 = **6851 ms**
Transaksjonsgjennomstrømning: 6 tr. / 14,23 s = **0,4 TPS**	Transaksjonsgjennomstrømning: 6 tr. / 6,85 s = **0,8 TPS**

Tabell 26.6: Resultat - ytelsestest

26.2.2 Frakoblingstest

Testspesifikasjon for:	Frakobling
	TOFK1
Enhet:	System
Hensikt med testen:	Formålet med testen er å avdekke i hvilken grad systemet klarer å håndtere frakobling av de enkelte deltakerne.
Kommentar:	

Beskrivelse av test

For å teste egenskapene i forbindelse med frakobling drepes prosessene hos en utvalgt deltaker. For de andre gjenværende deltakerne i systemet vil det da fortone seg som om denne deltakeren har koblet fra. Deltakeren vil deretter startes opp igjen på nytt. Vi skulle gjerne ha testet systemet på mobile enheter i denne testen, men det lot seg dessverre ikke gjøre fordi slikt utstyr ikke var tilgjengelig for oss.

Deltest	Hva må gjøres	Forventet resultat	Resultat
1	Frakobling Koordinator	Commit/abort prosessering stopper opp inntil koordinatoren restartes.	OK
2	Frakobling Klient	Systemet fortsetter å prosessere de transaksjonene klienten har fått formidlet.	OK
3	Frakobling Javaspace	Deltakerne rapporterer at de ikke får kontakt med JavaSpacet og stopper opp.	OK
4	Frakobling Databasetjener	Transaksjonene med åpen forbindelse mot databasen aborteres.	OK

Tabell 26.7: Testresultat - frakobling

Kapittel 27

Oppsummering

Resultatene av testene som er beskrevet i kapittel 26 vil her bli oppsummert.

Målet med de første testene var å undersøke om funksjonaliteten til de ulike delsystemene var i henhold til hva som var beskrevet i design og kravspesifikasjon. Testene TOF1-TOF4 er tester som skulle avkrefte eller bekrefte nettopp dette. Samtlige tester kjørte til slutt med det resultatet som forhånd var definert som korrekt. I test TOF4, som omhandler E4 Mobil klient, oppstod det litt problemer med deltest 1 og 2 som tok for seg innlesing fra fil og konstruksjon av XML dokumenter. Her var det primært filleseren som feilet, og som dermed forårsaket problemer med konstruksjon av XML dokumentene. Feilen ble straks rettet opp, og testene kjørte så tilfredsstillende. At testene gikk så bra, var kanskje ikke helt overraskende. Det ble foretatt kontinuerlig testing under hele implementasjonsfasen, for å forsikre at funksjonaliteten til enhver tid var i henhold til beskrevet.

Videre ble en mer omfattende systemtest definert. Her var det viktig at systemet kjørte i et mest mulig realistisk miljø. Testen ble derfor laget slik at koordinator, javaspace og databasetjenere skulle kjøre på separate maskiner. Foruten om dette, så kan systemtesten sees på som summen av alle deltestene, bare at her skulle man også teste på tvers av dem. Systemtesten kjørte med de resultatene som på forhånd var forventet.

På bakgrunn av systemtesten ble det utført en egen ytelses- og frakoblingstest. Resultatene fra ytelsestesten var på ingen måte overraskende. Deling av verdier (relaxed isolation) fører naturligvis til høyere ytelse. Man fikk bekreftet dette, med resultater som tilsa ytelsesforskjeller på faktor to mellom sekvensiell transaksjonshåndtering (0,4 TPS) og transaksjonshåndtering med deling (0,8 TPS). Da resultatene er i samsvar med forventet er kanskje den viktigste konklusjonen at systemet fungerer som det skal.

Frakoblingstesten svarte også til forventningene, og den underbygget påstandene om at systemet håndterer frakobling av deltakerne på en god måte.

Del VII

Evaluering

Innledning til dokumentet

Denne delen vil ta for seg evalueringen av prototypen og presentere en konklusjon for arbeidet som har blitt utført. I tillegg vil det bli foreslått videre arbeid som kan gjøres med både modellen og prototypen.

- Kapittel 28, Evaluering av prototypen, evaluerer prototypen med hensyn på hvordan den løser utfordringene knyttet til frakobling, økt samtidighet og mobilitet.
- Kapittel 29, Hva vi har lært, gir en oppsummering av lærdommen vi har tatt til oss i løpet av arbeidet.
- Kapittel 30, Videre arbeid, forslår temaer som kan være gjenstand for videre utvikling av modellen og prototypen.
- Kapittel 31, Konklusjon, gir en konklusjon på rapporten.

Kapittel 28

Evaluering av prototypen

I dette kapittelet skal prototypen evalueres med hensyn på de kriteriene som har vært et gjennomgangstema i hele rapporten, det vil si frakobling, økt samtidighet og mobilitet.

28.1 Frakobling

Prototypen viste høy toleranse for frakobling av deltakerne i systemet. Det viste seg under testing at midlertidige utkoblinger av klient, tjener eller koordinator ikke hadde noe å si for stabiliteten til systemet. Grunnen til dette er at all statusinformasjon blir utvekslet via transaksjonsobjekter i JavaSpacet, så hvis en deltaker kobler fra fører dette kun til at et eller flere transaksjonsobjekter blir forsinket i utførelsen. Når deltakeren får tilbake forbindelsen mot JavaSpacet vil den fortsette der den slapp med å utføre arbeidet med transaksjonsobjektene den har blitt tildelt.

Alle deltakerne ble testet på stasjonære maskiner med fast nettverksforbindelse, men både klient og koordinator er klare kandidater for å bli kjørt på mobile enheter. Dette er fordi de tålte svært godt den vilkårlige drepingen av prosessene som de ble utsatt for. Hvis klienten kræsjer etter at den har sendt avgårde sine transaksjoner til JavaSpacet vil alle transaksjonene utføres som om ingenting hadde hendt, og de ferdige resultatene vil ligge i JavaSpacet slik at klienten kan hente de ut når den har startet opp på nytt. Koordinatoren viste også stor toleranse for å bli avbrutt midt under utføring, den vil simpelthen starte på nytt igjen med commit/abort prosessering av de transaksjonene den holdt på med før den ble avbrutt. Databasetjeneren er middels tolerant ovenfor nettverksbrudd siden den har en like løs kobling mot JavaSpacet som de andre deltakerne, men den har i tillegg en eller flere åpne tilkoblinger mot MySQL databaser som gjør den mer sårbar for frakobling.

Imidlertid er det viktig at JavaSpacet kjører i et miljø med stabil nettverksforbindelse siden alle operasjoner går via JavaSpacet. Hvis JavaSpacet er nede eller mister tilkoblingen til nettet vil alle operasjoner stoppe opp helt til JavaSpacet er oppe å kjøre igjen.

28.2 Økt samtidighet

Ved hjelp av eksport- og importtransaksjonene har prototypen vist at den gir en økt grad av parallellitet mellom subtransaksjoner innenfor en global transaksjon. En klar fordel med systemet er at klienten i forkant bestemmer hvilke verdier som skal eksporteres og importeres, og deretter har den ikke noe ansvar for å se til at dette blir utført. Dermed slipper man en spørsmål-svar sekvens mellom klienten og resten av systemet, og man har oppnådd en asynkron kommunikasjonsprotokoll sett fra klienten sitt ståsted. Siden det er sannsynlig at klientene vil være blant de mobile deltakerne i systemet, er det viktig at nettopp disse kan utføre kommunikasjonen asynkront. For databasetjeneren er det viktigere med en stabil tilkobling, siden en forsinkelse her vil føre til at andre subtransaksjoner som er avhengige av en verdi som tjeneren skal produsere også vil bli forsinket.

En begrensning med prototypen er at det ikke er tillatt med deling av verdier mellom subtransaksjoner som hører til forskjellige globale transaksjoner. Dette ble vanskelig å implementere på grunn av at konsistens må opprettholdes i databasesystemet, og hvis man tillatter dirty reads på tvers av globale transaksjoner kan man risikere at en global transaksjon G2 bruker en verdi fra en annen global transaksjon G1, og at G2 committer før G1. Deretter aborterer G1, og man har en inkonsistens i systemet. Dette må unngås ved hjelp av låsing, kompenserende transaksjoner eller andre virkemidler.

Styrken ligger i at brukeren eller applikasjonsprogrammereren kan spesifisere hvilke transaksjoner som skal grupperes innunder en global transaksjon og dermed være istand til å dele verdier seg imellom. Man gir med andre ord et rammeverk for å støtte økt samtidighet, men det er opp til den som skriver transaksjonene å dra nytte av det.

28.3 Mobilitet

Siden prototypen er implementert ved hjelp av Jini Services, er mobilitet støttet i den grad at alle deltakerne kan stoppe utførelsen for senere å starte opp på en annen maskin uten noen form for rekonfigurering. Denne formen for nettverkstransparens er nyttig i systemer med mobile deltakere siden disse ofte blir byttet ut eller forandrer nettverksadresse. I prototypen er det kun navnetjeneren som må ha en statisk adresse, men denne kan også bli funnet ved hjelp av multicast som beskrevet i kapittel 19.

Prototypen støtter ikke opprettelse av nye delingsrom eller flytting av transaksjoner mellom forskjellige delingsrom mens transaksjonene er under utførelse. Dette er et aspekt av systemet som krever mer arbeid med både modellen og prototyp, siden det knyttet tidkrevende utfordringer til feilhåndering av transaksjonene og konsistens i databasesystemet.

Verken klientene eller koordinatoren lagrer tilstandsinformasjon om transaksjonene underveis, så det er fullt mulig å la en koordinator gjøre ferdig sine pågående oppgaver, for deretter å stanse prosessen og heller starte opp en ny koordinator fra en annen maskin i nettverket. Ved å ha så lite tilstandsinformasjon som mulig på de enkelte deltakerne, gjør man det enkelt å bytte ut en deltaker som feiler med en ny likeverdig deltaker.

Kapittel 29

Hva vi har lært

Diplomarbeidet fremstår nok som et av de mest lærerike prosjektene vi har vært med på i løpet av studietiden. Å jobbe med eksport-import modellen har gitt oss utfordringer på flere plan, både teoretisk og praktisk. Vi har også dratt mye lærdom rent organisatorisk gjennom å planlegge hva som skal gjøres og iversette dette innenfor de gitte tidsrammer. Vi vil her oppsummere våre viktigste erfaringer innenfor henholdsvis *teoretisk utbytte* og *praktisk erfaring*.

Teoretisk utbytte

Arbeidet med eksport-import modellen krevde god bakgrunnkunnskap om mer avanserte transaksjonsmodeller. Viktig kunnskap her er å vite hvordan disse er bygd opp og strukturert sammenlignet med mer standard modeller med streng ACID utførelse. Først av alt måtte vi identifisere modeller som var mulige kandidater, det vil si modeller som hadde likhetstrekk med eksport-import modellen. Disse måtte videre evalueres og sammenlignes med eksport-import modellen for å finne ut av hva som gjorde denne modellen fordelaktig framfor andre eksisterende løsninger. Når det gjelder teoretisk utbytte kan det hele oppsummeres med at vi har lært mye om hvilke transaksjonsmodeller som finnes for mobile applikasjoner, og om hvordan disse er strukturert og hvordan de fungerer.

Praktisk erfaring

I vårt tilfelle stod vi helt fritt i valg av teknologi. Vi måtte først av alt finne ut hvilke egenskaper med modellen som kom til å bli avgjørende for en implementasjon av en prototype. Et alternativ var naturligvis å starte helt fra scratch, for så implementere hele systemet fra bunn. Vi oppdaget raskt at dette ville bli for krevende, så vi rettet etterhvert fokus på å finne allerede eksisterende løsninger som kunne støtte opp mot vår implementasjon av selve modellen. Bruk av JavaSpacet var et resultat av grundige analyser av modellen, og må kanskje ansees som et av de beste valgene som ble gjort. Det viste seg at denne teknologien passte perfekt for å implementere eksport-import modellen.

Kapittel 30

Videre arbeid

I dette arbeidet har det vært fokus på å vise konsepter, både teoretisk og gjennom implementasjonen av prototypen. Det har derfor ikke vært et mål å utvikle en mest mulig komplett løsning, en oppgave som rent tidsmessig aldri ville latt seg gjøre innenfor de gitte rammene for denne typen oppgaver. Dette kapittelet vil derfor ta for seg hvilke områder av prototypen som kan videreutvikles en gang i framtiden.

Protypen fremstår i dag som et rammeverk for å kunne støtte deling av verdier mellom subtransaksjoner innenfor den samme globale transaksjonen, og tilbyr ikke deling mellom to forskjellige globale transaksjoner. Dette er et viktig område, men også et område som er utfordrende og byr på mange problemer [22]. Deling og konsistens mellom globale transaksjoner kan konstrueres på et nivå over det rammeverket som prototypen tilbyr. Konseptuelt sett gjør man oppdateringer til de delte verdiene i en omgang, eller så kan oppdateringene forplante seg etterhvert. I [22] argumenteres det for at ingen av disse to måtene vil fungere på egenhånd uten modifiseringer i et mobilt miljø. Disse metodene er beskrevet i tillegg A, og den foreslåtte løsningen baserer seg på to-lags replikering som tar i bruk semantiske triks som tidsmerking og kommutative transaksjoner.

Det er også gjort lite når det gjelder det dynamiske aspektet med delingsrommet. Egenskaper som splitting og forening av delingsrommet er ikke implementert, men prototypen er tilrettelagt for å kunne utvides på dette området. Hvis disse aspektene hadde blitt tatt med, ville det forbedret prototypens håndtering av mobilitet blant deltakerne.

Kapittel 31

Konklusjon

Vi startet diplomarbeidet med å skaffe oss en oversikt over de modellene som allerede fantes for å håndtere mobile transaksjoner og deling av verdier. I løpet av kort tid ble det klart at det fantes en myraide av forskjellige publiserte modeller som løste noen av de utfordringene som fantes med mobile distribuerte databasesystemer, men ingen enkelt modell løste alle utfordringene. I tillegg var det svært få modeller som hadde en implementasjon å vise til.

Evalueringen av eksport-import modellen med hensyn på de andre modellene som var publisert, viste at den hadde et stort potensiale for å løse problemer knyttet til mobilitet, økt samtidighet og frakobling. Siden modellen baserer seg på høynivå interaksjoner mellom databaser og deltakere i systemet, ble det rask klart at implementasjonen var nødt til å støtte seg på en form for mellomvare slik at nettverksproblematikk og protokoller ble abstrahert vekk. Valget falt på JavaSpaces teknologien, og den viste seg å fungere bra for formålet. Eksport- og importtransaksjoner, samt spørringer og resultater kunne uten store implementasjonskostnader sendes mellom deltakerne i systemet.

Prototypen beviste at den hadde gode egenskaper i forhold til å tillate frakobling og støtte økt deling mellom transaksjoner under utførelse. Den løse koblingen mellom deltakerne i systemet gjorde at en del mobile egenskaper kom av seg selv, for eksempel kan transaksjonskoordinatoren uten noen form for rekonfigurering starte opp på en ny maskin i det mobile nettverket og fortsette eksekveringen derfra. Se forøvrig kapittel 28 for en grundigere evaluering av prototypen.

Vi kan se for oss at egnede bruksområder for eksport-import modellen vil være i distribuerte mobile databasemiljøer med mye utskiftning og dynamikk i både programvare og maskinvare. Modellen tillater en samlet spesifisering av transaksjonene som skal utføres, selv om disse er distribuerte og må utføres på forskjellige databasetjenere. Assossiative oppslag gir en verdifull form for nettverkstransparens, siden det gjør det mulig å koble inn en ny deltaker uten noen form for konfigurering. Bruk av XML for å spesifisere transaksjonene samt å angi hvilke verdier som skal eksporteres og importeres på forhånd, gjør at arbeidet og mengden tilstandsinformasjon på klientene reduseres i stor grad. Dette er svært positivt for mobile miljøer, som ofte har klienter som kan koble ifra uten forvarsel.

Del VIII

Vedlegg

Bibliografi

[1] W.M.P. van der Aalst. Formalization and Verification of Event-driven Process Chains. Computing Science Reports 98/01, Eindhoven University of Technology, Eindhoven, 1998. 16.4

[2] MySQL AB. Mysql connector/j documentation. *http://dev.mysql.com/doc/connector/j/en/cj-what-is.html*. 14.4

[3] Telenor AS. Mer om umts og edge. *http://telenormobil.no/bedrift/tjenester/3g/merom.do*. 5.2

[4] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. 4.3.1, 4.3.2

[5] Angelo Brayner and José de Aguiar M. Filho. Sharing mobile databases in dynamically configurable environments. In *CAiSE*, pages 724–737, 2003. (document), 7.1, 7.1

[6] CAGIS. Cooperative agents in a global information space. *www.idi.ntnu.no/ cagis*. 2

[7] David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? Technical Report RC 19887 (December 21, 1994 - Declassified March 16, 1995), Yorktown Heights, New York, 1994. 9.2.1

[8] P. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using acta. acm transactions on database systems, vol. 19, no.3, pages 450-491. September 1994.

[9] Panos K. Chrysanthis. Transaction processing in mobile computing environment. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 77–83, Princeton, New Jersey, 1993. 7.4

[10] P. E. Clements, Todd Papaioannou, and John Edwards. Aglets: Enabling the virtual enterprise. In *Managing Enterprises – Stakeholders, Engineering, Logistics and Achievement (ME-SELA'97)*, Loughborough University, UK, 1997. 9.2.2

[11] R. Conradi, O. Larsen, M. Nguyen, A. Wang, and C. Liu. Transaction models for software engineering database. in proceedings of the dagstuhl workshop on software engineering databases. 1997. 2

[12] M. Dunham, A. Helal, and S. Balakrishnan. A mobile transaction model that captures both the data and movement behaviour. June 1997. 7.3

[13] Margaret H. Dunham, Abdelsalam Helal, and Santosh Balakrishnan. A mobile transaction model that captures both the data and movement behavior. *Mobile Networks and Applications*, 2(2):149–162, 1997. 6.3.1

[14] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2000. 4.4, 6.1

[15] G. H. Forman and J. Zahorjan. The challenges of mobile computing. Technical Report TR-93-11-03, 1993.

[16] Martin Fowler and Kendall Scott. *UML Distilled Second Edition*. Addison-Wesley, 2000. 14.1

[17] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, November 1999.

[18] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259. ACM Press, 1987. 7.4

[19] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985. 9.1

[20] S. Granklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. 1996. 9.2

[21] J. Gray and A. Reuter. *Transaction processing: Concepts and Techniques*. Morgan Kaufmann, 1993. 4.3

[22] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. pages 173–182, 1996. 7.5, 7.5.1, 30, A, A, A

[23] Software Design Description Working Group. Ieee recommended practice for software design descriptions. 1998. IV, C

[24] Joanne Holliday, Divyakant Agrawal, and Amr El Abbadi. Disconnection modes for mobile databases. *Wireless Networks*, 8:391–402, 2002. 6.1

[25] Tomasz Imielinski and B. R. Badrinath. Mobile wireless computing: Challenges in data management. *Communications of the ACM*, 37(10):18–28, 1994. (document), 5.1, 5.1

[26] J. Klingemann, T. Tesch, and J. Wäsch. Semantic-based transaction management for cooperative applications. April 1996.

[27] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In *The VLDB Journal*, pages 95–106, 1990.

[28] Sun Microsystems. *Java 2 Platform, Standard Edition, v 1.4.2 API Spesification*. http://java.sun.com/products/jini/2.0/doc/api/, 2003.

[29] Sun Microsystems. *Jini Technology Starter Kit v2.0 API Documentation*. http://java.sun.com/products/jini/2.0/doc/api/, 2004.

[30] MOWAHS. Mobile work across heterogeneous systems. *http://www.mowahs.com*. (document), 2, 2.1

[31] Jan Newmarch. *A Programmer's Guide to Jini Technology*. http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml, October 2004.

[32] Mads Nygård and Hien Nam Le. Mowahs: Mobile transaction system for supporting mobile work. (document), 3, 8.3

[33] C.H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979. 4.3.1

[34] Herindrasana Ramampiaro. *CAGISTrans: Adaptable Transactional Support for Cooperative Work*, October 2001. 6.2

[35] Patricia Serrano-Alvarado, Claudia Roncancio, and Michel Adiba. A survey of mobile transactions. *Distrib. Parallel Databases*, 16(2):193–230, 2004. 7

[36] Gary D. Walborn and Panos K. Chrysanthis. PRO-MOTION : Management of mobile transactions. In *Selected Areas in Cryptography*, pages 101–108, 1997. 7.2

[37] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transaction. 1992. 4.2.2

Tillegg A

Global konsistens og deling

Protypen som er implementert i denne oppgaven har ikke støtte for konsistens og deling av verdier mellom transaksjoner som blir kjørt innunder ulike globale transaksjoner. Med konsistens menes at transaksjonene alltid etterlater databasen i en gyldig tilstand. En eventuell løsning for dette vil være å bygge en mekansime over det rammerverket som er laget for eksport-import modellen. Ulike generelle metoder for å oppnå konsistens vil diskutert i dette kapittelet, som primært er basert på [22].

Eager replication

Ved bruk av denne metoden så vil alle replikater bli oppdatert når en transaksjon selv oppdaterer en instans av et objekt. Her brukes låsing for å sikre serialiserbarhet. Dette er en metode som vil fungere greit i et vanlig distribuert system hvor ikke for mange noder er involvert. Problemer oppstår derimot i mobile systemer. Dette skyldes primært følgende:

- Bruk av låsing fungerer dårlig, da dette krever synkron forbindelse mellom de involverte nodene. For mobile enheter som kan være frakoblet store deler av tiden vil det ikke la seg gjøre.
- Sannsynligheten for vranglås, som forårsaker at mange transaksjoner feiler, øker kraftig med transaksjonsstørrelse og antall noder.

[22] konkluderer med at ivrig replikering ikke er en god kandidat for å forsikre konsistens mellom mobile transaksjoner.

Lazy Group Replication

Ved bruk av denne metoden har transaksjoner lov til å oppdatere hvilke som helst lokale data. Når en transaksjon committer, vil en transaksjon bli sendt til samtlige noder, slik at rottransaksjonens oppdateringer blir oppdatert. Her er det mulig for to transaksjoner gjøre oppdateringer

på samme objekt, noe som kan føre til at begge jager hverandre for å få sine oppdateringer reflektert. Det må brukes egne mekansimer som sørger for at slike tilfeller blir oppdaget, og at begge transaksjonenes oppdateringer heller blir forenet (eng. reconcile). Tidsmerking kan brukes for å løse dette problemet.

Generelt så vil problemet i denne metoden være at transaksjoner må vente på hverandre, før oppdateringer på felles data kan forenes. Dette vil fungere spesielt dårlig i et mobilt system. Eksempelvis så kan en mobil enhet laste ned data tidlig om morgenen, for så koble fra, og forbli frakoblet store deler av dagen. Når så denne kobler seg opp igjen etter eksempelvis 24 timer, først da kan eventuell forening mellom transaksjoner skje. Dette er selvfølgelig avhengig av sannsynligheten for at to transaksjoner gjør oppdateringer på samme data. Sannsynligheten for dette øker krvadratisk med antall transaksjoner, operasjoner og antall noder. Denne metoden skalerer (økning i antall noder) med andre ord dårlig.

Lazy Master Replication

Denne metoden skiller seg fra den forrige ved at hvert objekt har sin egen eier, som holder oversikt over den gjeldende korrekte verdien. Oppdateringer vil først bli utført av eier, for så bli reflektert hos andre replikater. En transaksjon vil dermed først henvende seg til eier av et objekt for så å få låsen til dette. Eiernoden tar seg av oppdateringer av eventuelle replikater når denne transaksjonen committer. Denne metoden fungerer også dårlig for mobile system. Den krever at en transaksjon som ønsker å oppdatere et objekt må ha synkron kontakt med eier av objektet for så utføre en atomisk operasjon.

Oppsummering

I artikkelen [22] som beskriver de ulike metodene, så konkluderer forfatteren med at ingen av de nevnte metodene vil være gode nok for mobile transakjoner. Forfatteren foreslår en løsning basert på to-lags replikering som tar i bruk semantiske triks som tidsmerking og kommutative transaksjoner. Denne modellen er beskrevet i kapittel 7.5 i denne rapporten.

Tillegg B

Systemtest - Spesifikasjon

```
<globalTrans>

<subTrans>
  <databaseId>Fly</databaseId>
  <query>
    <sql>SELECT Flysete.id AS seteid, Flyavgang.id AS flyid, dato, ankomst
      FROM Flysete, Flyavgang
      WHERE Flyavgang.flight = 'BA 622'
      AND Flysete.flyid = Flyavgang.id
      AND Flysete.kundenummer != '9000'
      ORDER BY dato LIMIT 1
    </sql>
    <export>
      <name>seteid</name>
      </export>
    <export>
      <name>flyid</name>
    </export>
    <export>
      <name>kundenummer</name>
      <value>9000</value>
    </export>
    <export>
      <name>dato</name>
    </export>
  </query>
</subTrans>

<subTrans>
  <databaseId>Fly</databaseId>
  <query>
    <sql>UPDATE Flysete SET kundenummer = '9500' WHERE
      id = 'seteid' AND flyid = 'flyid'
    </sql>
    <import>
      <name>seteid</name>
    </import>
    <import>
      <name>flyid</name>
    </import>
  </query>
</subTrans>

<subTrans>
  <databaseId>Fly</databaseId>
  <query>
    <sql>SELECT COUNT(*) FROM Flyavgang, Flysete, Kunde WHERE
      Flyavgang.id = Flysete.flyid AND Flysete.kundenummer = Kunde.kundenummer
    </sql>
  </query>
</subTrans>

<subTrans>
  <databaseId>Bil</databaseId>
  <query>
    <sql>SELECT Leiebil.regnr AS regnr FROM Leiebil, Reservasjon WHERE
      Leiebil.regnr = Reservasjon.regnr
      AND Reservasjon.kundenummer IS NULL
      AND Reservasjon.fra = 'dato'
    </sql>
    <import>
      <name>dato</name>
    </import>
    <export>
      <name>regnr</name>
    </export>
  </query>
</subTrans>

<subTrans>
```

```
  <databaseId>Bil</databaseId>
  <query>
    <sql>INSERT INTO Reservasjon(id, fra, til, regnr, kundenummer) VALUES
      ('200', 'dato', 'dato', 'regnr', 'kundenummer')
    </sql>
    <import>
      <name>dato</name>
    </import>
    <import>
      <name>regnr</name>
    </import>
    <import>
      <name>kundenummer</name>
    </import>
  </query>
</subTrans>

<subTrans>
  <databaseId>Bil</databaseId>
  <query>
    <sql>SELECT COUNT(*) FROM Bilmerke, Kunde, Leiebil, Reservasjon WHERE
      Bilmerke.id = Leiebil.merkeid
      AND Kunde.kundenummer = Reservasjon.kundenummer;
    </sql>
  </query>
</subTrans>

</globalTrans>
```

Tillegg C

Dokumentasjon til kildekode

I dette vedlegget til detaljert design skal vi beskrive den interne virkemåten til alle designentitetene i systemet. Dette kapittelet vil derfor fungere som dokumentasjon til implementasjonen. Enkelte uviktige metoder (slik som get og set metoder) vil ikke bli tatt med for å spare plass. Alle relevante variabler og metoder vi bli beskrevet med et sett attributter. Disse attributtene er basert på retningslinjene for detaljert design beskrevet i [23].

Attributtene brukt for å beskrive variabler er som følger:

- **Type:** Typen til variablen, for eksempel Integer eller String.
- **Synlighet:** Public, private eller protected.
- **Format:** Syntaksen sammensatte verdier må forholde seg til.
- **Verdiområde:** Gyldige verdier for variabelen.
- **Bruksområde:** Hva variabelen skal brukes til og hvor den brukes.

Attributtene brukt for å beskrive metoder er som følger:

- **Argumenter:** Innverdier til metoden.
- **Returtype:** Typen til verdien som returneres.
- **Synlighet:** Public, private eller protected.
- **Initiering:** Når metoden blir kalt og hvem som kaller den.
- **Prosessering:** Hva metoden gjør.

C.1 Beskrivelse av entitetene

E1 Transaction

Transaction er superklassen til alle de andre transaksjonsklassene. Den inneholder ingen metoder, men har tre variable for å bevare tilstanden til en transaksjon. Transaction klassen er abstrakt og instansieres aldri direkte, det er subklassene av Transaction som blir instansiert.

Variabler

globalId

- *Type:* String
- *Synlighet:* Public

- *Format:* Sammensatt av DNS adressen til den maskinen som oppretter transaksjonen pluss Unix tidsstempelet som angir når transaksjonen ble opprettet. Disse to verdiene bli skilt av et kolon. Et eksempel på en gyldig globalId er maskinXXX.idi.ntnu.no:12242345.

- *Verdiområde:* Alle gyldige DNS (Domain Name Service) adresser og Unix tidsstempler kan inngå i verdiområdet til globalId.

- *Bruksområde:* globalId brukes sammen med en eventuell subId for å identifisere hver enkelt transaksjon slik at den kan hentes ut fra et JavaSpace.

status

- *Type:* Integer

- *Synlighet:* Public

- *Format:* Integer som er en forhåndsdefinert konstant i klassen Constants.

- *Verdiområde:* IN_PROGRESS, READY_TO_COMMIT, COMMIT, ABORT, COMMITTED, ABORTED.

- *Bruksområde:* status brukes for å ta vare på hvilken tilstand transaksjonen befinner seg i med hensyn på commit/abort prosessering.

nextStop

- *Type:* Integer

- *Synlighet:* Public

- *Format:* Integer som er en forhåndsdefinert konstant i klassen Constants.

- *Verdiområde:* CLIENT, HOST, COORDINATOR.

- *Bruksområde:* nextStop brukes for å angi hvilken funksjonell enhet som skal få tilgang til transaksjonen. Dette kan sees på som et slags høynivå adressefelt.

E1.1 GlobalTrans

GlobalTrans er en subklasse av Transaction, og denne klassen definerer en enhet med arbeid som skal utføres i systemet. Ansvarsområdet til globalTrans er å ta vare på en liste over alle de subtransaksjonene som er tilknyttet objektet.

Variabler

subTransIds

- *Type:* Vector
- *Synlighet:* Public
- *Format:* Vector som inneholder subId variabler.
- *Verdiområde: null* hvis det ikke finnes noen tilknyttede subtransaksjoner, ellers et Vector objekt som inneholder alle de subId objektene som representerer subtransaksjoner som er tilknyttet den globale transaksjonen.
- *Bruksområde:* Alle subId verdiene brukes for å hente ut alle de subtransaksjonene som er tilknyttet en global transaksjon for å sjekke status til subtransaksjonene. Slik kan koordinatoren ta en avgjørelse på om den globale transaksjonen skal committes eller aborteres som en enhet.

subIdIterator

- *Type:* Integer
- *Synlighet:* Private
- *Format:* Integer
- *Verdiområde:* Denne variablen er et monotont stigende heltall som gir en unik subId innenfor en global transaksjon. Den initielle verdien er 0, og den stiger med 1 for hver subtransaksjon som legges til.
- *Bruksområde:* Brukes for å finne en unik subId.

Metoder

getNextSubId()

- *Argumenter:* Ingen
- *Returtype:* Integer
- *Synlighet:* Public
- *Initiering:* Kalles når en ny subtransaksjon skal knyttes til en global transaksjon.
- *Prosessering:* Legger til 1 til den nåværende subIdIterator verdien, og returnerer denne.

addSubTransId(Integer subId)

- *Argumenter:* subId - identifikatoren til den subtransaksjonen som skal legges til.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles når en ny subtransaksjon skal knyttes til en global transaksjon.
- *Prosessering:* Sjekker om subId fins i subTransIds fra før, hvis den ikke gjør det legges den til.

E1.3 SubTrans

SubTrans er en subklasse av Transaction og den er tilknyttet en global transaksjon. Den inneholder variabler for å ta vare på SQL spørringer og resultatet av disse. Informasjon om eksport- og import-transaksjoner blir også lagret her.

Variabler

subId

- *Type:* Integer
- *Synlighet:* Public
- *Format:* Integer
- *Verdiområde:* Fra 0 opp til antall transaksjoner tilknyttet den globale transaksjonen minus 1.
- *Bruksområde:* subId brukes sammen med globalId for å identifisere en subtransaksjon og hente denne ut fra et JavaSpace.

databaseId

- *Type:* String
- *Synlighet:* Public
- *Format:* String
- *Verdiområde:* En tekstlig identifikator som svarer til en av de distribuerte databasene. Hver databasetjener er tilknyttet en eller flere databaser som alle har en slik unik identifikator.
- *Bruksområde:* Databasetjenere henter kun ut subtransaksjoner som har en databaseId som svarer til en av de databasene tjeneren har ansvaret for.

queryList

- *Type:* Vector
- *Synlighet:* Public
- *Format:* Vector som inneholder SubTransQuery objekter.
- *Verdiområde: null* hvis det ikke finnes noen tilknyttede spørringer, ellers et Vector objekt som inneholder alle de SubTransQuery objektene som representerer spørringene subtransaksjonen ønsker å utføre mot databasen.
- *Bruksområde:* Lagring for SubTransQuery objektene.

sharedNameList

- *Type:* Vector
- *Synlighet:* Public
- *Format:* Vector som inneholder String objekter som svarer til navnene på de verdiene denne subtransaksjonen har delt.
- *Verdiområde: null* hvis det ikke finnes noen tilknyttede delte verdier, ellers et Vector objekt som inneholder navnene på alle de verdiene denne subtransaksjonen har delt.
- *Bruksområde:* Koordinatoren bruker denne listen som en enkel måte å få tilgang til alle eksporttransaksjonene som subtransaksjonen har initiert. Dette brukes når eksporttransaksjonene må aborteres.

query

- *Type:* SubTransQuery
- *Synlighet:* Private
- *Format:* SubTransQuery
- *Verdiområde:* Initialiseres til et SubTransQuery objekt av initQuery(), og manipuleres så av forskjellige andre metoder i SubTrans klassen.
- *Bruksområde:* SubTrans bruker denne variabelen for å holde et midlertidig SubTransQuery objekt. Når objektet er ferdig kalles finishQuery() metoden, og objektet legges så til queryList.

Metoder

initQuery()

- *Argumenter:* Ingen
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles når et nytt SubTransQuery objekt skal bygges.
- *Prosessering:* Initialiserer query til å være et nytt SubTransQuery objekt.

finishQuery()

- *Argumenter:* Ingen
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles når SubTransQuery objektet er ferdig bygget.
- *Prosessering:* Legger query til queryList.

addSql(String sql)

- *Argumenter:* sql - selve SQL spørringen
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles når SQL spørringen skal legges til et SubTransQuery objekt.
- *Prosessering:* Tilordner SubTransQuery objektet SQL spørringen gitt i argumentet.

addExport(String exportName, String exportValue)

- *Argumenter:*
  exportName - navnet til variabelen som skal eksporteres.
  exportValue - verdien til variabelen som skal eksporteres.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles når en av variablene i spørringen skal markeres som en eksportverdi.

- *Prosessering:* Lager en ny ExportEntry fra exportName og exportValue og legger denne til exportEntries i det aktive SubTransQuery objektet.

addImport(String importName, String importValue)

- *Argumenter:*
  importName - navnet til variabelen som skal importeres.
  exportName - verdien til variabelen som skal importeres.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles når en av variablene i spørringen skal markeres som en importverdi.
- *Prosessering:* Lager en ny ImportEntry fra importName og importValue og legger denne til i importEntries i det aktive SubTransQuery objektet.

E1.2 SharingTrans

SharingTrans er en subklasse av Transaction, og den er også superklassen til ExportTrans og ImportTrans. Den inneholder variabler som brukes av både ExportTrans og ImportTrans.

Variabler

subId

- *Type:* Integer
- *Synlighet:* Public
- *Format:* Integer
- *Verdiområde:* Fra 0 opp til antall transaksjoner tilknyttet den globale transaksjonen minus 1.
- *Bruksområde:* subId brukes sammen med globalId for å identifisere hvilken subtransaksjon som delingstransaksjonen hører til.

sharedName

- *Type:* String
- *Synlighet:* Public
- *Format:* Integer

- *Verdiområde:* Alle navn som er tillatt for SQL kolonner.
- *Bruksområde:* Brukes sammen med globalId og subId for å finne igjen en bestemt Export-Trans eller ImportTrans.

sharedValue

- *Type:* String
- *Synlighet:* Public
- *Format:* Integer
- *Verdiområde:* Alt som kan representeres som String.
- *Bruksområde:* Brukes som input eller output verdi alt etter som det er en ImportTrans eller ExportTrans som har blitt instansiert.

E1.2.1 ExportTrans

ExportTrans er den klassen som representerer en eksporttransaksjon, og den er en subklasse av SharingTrans.

Variabler

importTransList

- *Type:* String
- *Synlighet:* Public
- *Format:* Integer
- *Verdiområde:* Alt som kan representeres som String.
- *Bruksområde:* Brukes som input eller output verdi alt etter som det er en ImportTrans eller ExportTrans som har blitt instansiert.

Metoder

addImportTrans(ImportTrans it

- *Argumenter:* it - ImportTrans objektet som skal refereres.
- *Returtype:* Ingen

- *Synlighet:* Public
- *Initiering:* Kalles når en importtransaksjon skal importere data fra en eksporttransaksjon.
- *Prosessering:* Legger til ImportTrans objektet i importTransList.

E1.2.2 ImportTrans

ImportTrans er den klassen som representerer en importtransaksjon, og den er en subklasse av SharingTrans.

Variabler

globalIdExport

- *Type:* String
- *Synlighet:* Public
- *Format:* Samme som globalId i Transaction.
- *Verdiområde:* Samme som globalId i Transaction.
- *Bruksområde:* Brukes sammen med subIdExport for å identifisere eksporttransaksjonen som importtransaksjonen henter data fra.

subIdExport

- *Type:* Integer
- *Synlighet:* Public
- *Format:* Samme som subId i SubTrans.
- *Verdiområde:* Samme som subId i SubTrans.
- *Bruksområde:* Brukes sammen med globalIdExport for å identifisere eksporttransaksjonen som importtransaksjonen henter data fra.

E1.2 SubTransQuery

SubTransQuery klassen brukes av SubTrans for å lagre en enkelt spørring mot databasen. En SubTrans kan inneholde en liste over flere slike SubTransQuery objekter.

Variabler

sql

- *Type:* String
- *Synlighet:* Public
- *Format:* Gyldig SQL syntax.
- *Verdiområde:* Gyldig SQL syntax.
- *Bruksområde:* Denne variabelen inneholder selve SQL spørringen som utføres mot databasen.

result

- *Type:* String
- *Synlighet:* Public
- *Format:* XML dokument som representerer et sett med rader. En rad kan for eksempel se slik ut:
  <rad> <navn>Egil</navn><yrke>Snekker</yrke><inntekt>310000</inntekt> </rad>
- *Verdiområde:* Alt som er gyldige svar fra databasen.
- *Bruksområde:* Sendes tilbake til klienten for å vise brukeren hva resultatet av spørringen var.

exportEntries

- *Type:* Vector
- *Synlighet:* Public
- *Format:* Vector som inneholder ExportEntry objekter.
- *Verdiområde: null* hvis det ikke finnes noen verdier som skal eksporteres. Ellers er det en Vector med ExportEntry objekter der hvert slikt objekt inneholder navnet og eventuelt også verdien til en variabel som skal eksporteres.
- *Bruksområde:* Brukes for å finne ut av hvilke variabler en spørring ønsker å eksportere.

importEntries

- *Type:* Vector
- *Synlighet:* Public

- *Format:* Vector som inneholder ImportEntry objekter.
- *Verdiområde: null* hvis det ikke finnes noen verdier som skal importeres. Ellers er det en Vector med ImportEntry objekter der hvert slikt objekt inneholder navnet og eventuelt også verdien til en variabel som skal importeres.
- *Bruksområde:* Brukes for å finne ut av hvilke variabler en spørring ønsker å importere.

E4.1 MobileClient

MobileClient har ansvaret for å lese inn XML spørringer fra fil og oversette dem til transaksjonsobjekter som formidles videre i det distribuerte systemet. Klassen har også metoder for å abortere transaksjoner under utførelse samt å vise resultatet til kjørte transaksjoner. Slik kan denne klassen brukes for å teste forskjellige aspekter ved systemet.

Variabler

spaceManager

- *Type:* SpaceManager
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til et SpaceManager objekt.
- *Bruksområde:* Brukes for å få tilgang til forskjellige JavaSpaces.

sharingSpace

- *Type:* JavaSpace
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til det JavaSpacet som brukes som transaksjonsrom og eventuelt også delingsrom.
- *Bruksområde:* Brukes som argument til TransMod.

parser

- *Type:* XMLtoEntryParser

- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til et XMLtoEntryParser objekt.
- *Bruksområde:* Brukes for å parse XML filer til transaksjonsobjekter.

transMod

- *Type:* TransMod
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til et TransMod objekt.
- *Bruksområde:* Bruker metodene til TransMod klassen for å utføre forskjellige lese- og skrive-operasjoner mot et JavaSpace.

startTime

- *Type:* long
- *Synlighet:* Private
- *Format:* Tidsstempel i Unix format.
- *Verdiområde:* Alle gyldige Unix tidsstempel.
- *Bruksområde:* Brukes for å beregne hvor lang tid en global transaksjon bruker på å kjøre.

finishTime

- *Type:* long
- *Synlighet:* Private
- *Format:* Tidsstempel i Unix format.
- *Verdiområde:* Alle gyldige Unix tidsstempel.
- *Bruksområde:* Brukes for å beregne hvor lang tid en global transaksjon bruker på å kjøre.

Metoder

traverseQueryDirectory(String queryDir)

- *Argumenter:* queryDir - navnet på mappen som inneholder spørringer i form av XML filer.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles når MobileClient starter opp.
- *Prosessering:* Finner alle XML filene i mappen og kaller parseXMLFile for hver av de.

parseXMLFile(File file)

- *Argumenter:* file - XML dokumentet som skal parses.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles fra traverseQueryDirectory.
- *Prosessering:* Sender XML dokumentet til parseren og får tilbake en Vector som inneholder en GlobalTrans og flere SubTrans objekter. GlobalTrans skrives så ut, deretter alle tilhørende SubTrans objekter. Deretter kalles eventuelt abortSubTrans på en eller flere subtransaksjoner hvis man har satt parametrene slik at man ønsker å teste abortering av transaksjoner. Til sist endres status for GlobalTrans objektet fra IN_PROGRESS til READY_TO_COMMIT. For hver transaksjon som skrives ut startes en også tråd som følger med på resultatet av transaksjonen. Når alle trådene har kjørt ferdig lagres tidsstempelet slik at total tid for kjøringen av den globale transaksjonen kan kalkuleres.

abortSubTrans

- *Argumenter:* Ingen
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles av parseXMLFile når en subtransaksjon skal aborteres.
- *Prosessering:* Henter ut en subtransaksjon med status READY_TO_COMMIT og setter den til ABORT.

E4.2 XMLtoEntryParser

Denne klassen tar inn en XML fil og lager en liste med en global transaksjon og tilhørende subtransaksjoner.

Variabler

serializer

- *Type:* Transformer
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til et Transformer objekt.
- *Bruksområde:* Definerer attributter i forbindelse med parsing av XML dokumentet. på å kjøre.

builder

- *Type:* DocumentBuilder
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til et DocumentBuilder objekt.
- *Bruksområde:* Brukes for å lage et DOM dokument av XML dokumentet.

Metoder

createEntries(File xmlFile)

- *Argumenter:* xmlFile - XML fila som skal parses.
- *Returtype:* Vector
- *Synlighet:* Public
- *Initiering:* Kalles av parseXMLFile i MobileClient klassen.
- *Prosessering:* Parser den globale transaksjonen med alle tilhørende subtransaksjoner, import- og eksporttransaksjoner. globalId beregnes utifra DNS adresse og tidsstempel. Alle transaksjoner settes initielt til status lik IN_PROGRESS. En Vector med alle transaksjonsobjektene returneres til slutt.

E3.1 MobileHost

MobileHost er ansvarlig for å hente ut subtransaksjoner som skal utføres mot en av de databasene den er ansvarlig for.

Variabler

spaceManager

- *Type:* SpaceManager
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til et SpaceManager objekt.
- *Bruksområde:* Brukes for å få tilgang til forskjellige JavaSpaces.

sharingSpace

- *Type:* JavaSpace
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til det JavaSpacet som brukes som transaksjonsrom og eventuelt også delingsrom.
- *Bruksområde:* Brukes som argument til TransMod.

Metoder

readSubTrans()

- *Argumenter:* Ingen
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kjører når MobileHost starter opp.
- *Prosessering:* Henter ut en subtransaksjon som med status lik IN_PROGRESS og nextStop lik HOST. Deretter instansieres et nytt SubTransThread objekt med denne subtransaksjonen som argument.

connectToDatabase()

- *Argumenter:* Ingen
- *Returtype:* Connection
- *Synlighet:* Public
- *Initiering:* Kalles fra konstruktoren til SubTransQuery.
- *Prosessering:* Kobler til en MySQL database og returnerer et Connection objekt som gir tilgang til databasen.

E3.1 SubTransThread

Denne klassen brukes av MobileHost for å skille ut håndteringen av hver subtransaksjon i en egen tråd.

Variabler

subTrans

- *Type:* SubTrans
- *Synlighet:* Private
- *Bruksområde:* Brukes for å hente ut informasjon om subtransaksjonen.

connection

- *Type:* JavaSpace
- *Synlighet:* Private
- *Bruksområde:* Brukes som argument til TransMod.

sharingSpace

- *Type:* JavaSpace
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til det JavaSpacet som brukes som transaksjonsrom og eventuelt også delingsrom.
- *Bruksområde:* Brukes som argument til TransMod.

Metoder

run()

- *Argumenter:* Ingen
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles når tråden startes.
- *Prosessering:* Går igjennom queryList i subTrans objektet og instansierer et nytt QueryThread objekt for hver spørring. QueryThread objektet legges så til en liste over kjørende spørringer. Metoden venter så til alle trådene er ferdig, og deretter sjekkes status for eksekveringen. Hvis en av spørringene ble avsluttet med en exception, rulles transaksjonen tilbake og subTrans objektet skrives tilbake med status lik ABORTED og nextStop lik COORDINATOR. Hvis alle spørringene ble avsluttet korrekt, så skrives subTrans objektet tilbake med status lik READY_TO_COMMIT og nextStop lik COORDINATOR.. Metoden vil så vente til koordinatoren har endret status på subtransaksjonen til enten COMMIT eller ABORT. Hvis status er lik COMMIT vil metoden kjøre commit() på Connection objektet, og så skrive tilbake subTrans objektet med status lik COMMITTED og nextStop lik CLIENT. Hvis status er lik ABORT vil metoden kjøre rollback() på Connection objektet, og så skrive tilbake subTrans objektet med status lik ABORTED og nextStop lik CLIENT.

E3.1 QueryThread

QueryThread brukes av SubTransThread for å utføre en enkelt spørring mot databasen og få resultatet tilbake.

Variabler

subTrans

- *Type:* SubTrans
- *Synlighet:* Private
- *Bruksområde:* Brukes for å få tilgang til informasjon om subtransaksjonen som globalId, subId etc.

subQuery

- *Type:* SubTransQuery

- *Synlighet:* Private
- *Bruksområde:* Brukes for å få tilgang til informasjon om spørringen som SQL setning, exportverdier etc.

connection

- *Type:* Connection
- *Synlighet:* Private
- *Bruksområde:* Brukes for å utføre operasjoner mot databasen.

Metoder

run()

- *Argumenter:* Ingen
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kjører når QueryThread tråden startes.
- *Prosessering:* Metoden går først igjennom importEntries lista til SubTransQuery objektet og henter ut de tilsvarende ExportTrans objektene fra JavaSpacet. For hver verdi som importeres skrives det ut et nytt ImportTrans objekt med identifikatoren til subtransaksjonen som importerer verdien. Deretter settes hver importerte verdi inn i SQL setningen der variabelnavnene er like. Deretter utføres SQL setningen mot databasen, og et eventuelt resultat parses først i parseResultSet før det lagres i SubTransQuery.result variabelen. Hvis spørringen skulle møte på en exception, vil queryStatus variabelen i SubTransThread objektet endres for å reflektere dette.

parseResultSet(ResultSet rs)

- *Argumenter:* rs - ResultSet objektet som skal parses.
- *Returtype:* String
- *Synlighet:* Public
- *Initiering:* Kalles fra run() metoden når et resultat fra databasen foreligger.
- *Prosessering:* Går igjennom alle radene i ResultSet objektet og konverterer denne til XML format. Resultatet når alle radene er prosessert returneres til slutt.

E2.1 Coordinator

Coordinator klassen vurderer transaksjonene som endrer status fortløpende, og avgjør på grunnlag av dette skjebnen til de globale transaksjonene.

Variabler

spaceManager

- *Type:* SpaceManager
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til et SpaceManager objekt.
- *Bruksområde:* Brukes for å få tilgang til forskjellige JavaSpaces.

sharingSpace

- *Type:* JavaSpace
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til det JavaSpacet som brukes som transaksjonsrom og eventuelt også delingsrom.
- *Bruksområde:* Brukes som argument til TransMod.

transMod

- *Type:* TransMod
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til et TransMod objekt.
- *Bruksområde:* Bruker metodene til TransMod klassen for å utføre forskjellige lese- og skrive-operasjoner mot et JavaSpace.

Metoder

makeDecision(GlobalTrans globalTrans)

- *Argumenter:* globalTrans - den globale transaksjonen som skal gjennomgå commit/abort prosessering.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles fra CoordThread når en ny subtransaksjon som er klar for commit/abort prosessering ankommer. Den tilhørende globale transaksjonen hentes ut og brukes som argument til makeDecision.
- *Prosessering:* Går igjennom alle statusfeltene til alle de tilhørende subtransaksjonene og avgjør utfallet utifra det. Hvis minst en subtransaksjon har status lik ABORTED skal hele den globale transaksjonen og alle subtransaksjonene som tilhører den aborteres. Hvis minst en subtransaksjon har status lik ABORT skal denne subtransaksjonen og alle transaksjoner som er avhengig av denne via en eksport-import sammenheng aborteres. Hvis minst en subtransaksjon har status lik IN_PROGRESS så skjer det ingenting. Hvis alle subtransaksjonene har status lik READY_TO_COMMIT settes status til den globale transaksjonen lik COMMITTED og alle de tilhørende subtransaksjonene lik COMMIT.

commitGlobal(GlobalTrans globalTr)

- *Argumenter:* globalTr - den globale transaksjonen som committes.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles av makeDecision.
- *Prosessering:* Setter status lik COMMITTED og nextStop lik CLIENT.

abortGlobal(GlobalTrans globalTr))

- *Argumenter:* globalTr - den globale transaksjonen som skal aborteres.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles av makeDecision.
- *Prosessering:* Setter status lik ABORTED og nextStop lik CLIENT.

commitSub(Vector transactions)

- *Argumenter:* transactions - lista med subtransaksjoner som skal committes.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles av makeDecision.
- *Prosessering:* Setter statusfeltene til alle subtransaksjonene lik COMMIT og nextStop lik HOST.

abortSubTransaction(SubTrans st)

- *Argumenter:* st - subtransaksjonen som skal aborteres.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles av makeDecision.
- *Prosessering:* Setters statusfeltet til subtransaksjonen lik ABORT og nextStop lik HOST. Kaller så abortExportTrans med subtransaksjonen som argument.

abortExportTrans(SubTrans subtrans)

- *Argumenter:* subtrans - subtransaksjonen som skal få alle sine eksporttransaksjoner abortert.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles av abortSubTransaction.
- *Prosessering:* Går gjennom sharedNameList til subtransaksjonen for å finne navnene på alle de variablene som har blitt eksportert. De eksporttransaksjonene som svarer til disse navnene blir tatt ut, og abortImportTrans blir kalt med hver av dem som argument.

abortImportTrans(ExportTrans et)

- *Argumenter:* et - eksporttransaksjonen som skal få alle sine importtransaksjoner abortert.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles av abortExportTransaction.
- *Prosessering:* Henter ut alle importtransaksjoner som svarer til den gitte eksporttransaksjonen, og den subtransaksjonen som har importert verdier fra importtransaksjonen blir deretter abortert via abortSubTransaction.

E2.1 CoordThread

Denne klasse en en hjelpeklasse for Coordinator, og leser inn relevante subtransaksjoner og globale transaksjoner.

Variabler

coord

- *Type:* Coordinator
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* Refererer til Coordinator objektet som dette objektet rapporterer til.
- *Bruksområde:* Brukes for å kalle makeDecision i Coordinator objektet.

Metoder

run()

- *Argumenter:* Ingen
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kjøres når CoordThread objektet blir startet.
- *Prosessering:* Leser inn en subtransaksjon som har nextStop lik COORDINATOR, og henter så inn den globale transaksjonen som er overordnet denne med nextStop lik COORDINATOR og status lik READY_TO_COMMIT. Deretter kalles makeDecision i Coordinator klassen med denne globale transaksjonen som argument.

E2.2 TransMod

Denne klassen inneholder alle metodene for å lese og skrive transaksjoner mot et JavaSpace. Disse metodene er kun av typen *get* og *set* metoder, så de vil derfor ikke bli beskrevet her. Se kildekode for implementasjonsdetaljer.

E1 Constants

Denne klassen inneholder alle de konstantene som brukes på tvers av klassene i systemet.

Variabler

IN_PROGRESS

- *Type:* Integer
- *Synlighet:* Public
- *Format:* Integer.
- *Verdiområde:* Unik i forhold til de andre konstantene.
- *Bruksområde:* Brukes som verdi i statusfelt for å angi at en transaksjon er under utførelse.

READY_TO_COMMIT

- *Type:* Integer
- *Synlighet:* Public
- *Format:* Integer.
- *Verdiområde:* Unik i forhold til de andre konstantene.
- *Bruksområde:* Brukes som verdi i statusfelt for å angi at en transaksjon er klar til å committe.

COMMIT

- *Type:* Integer
- *Synlighet:* Public
- *Format:* Integer.
- *Verdiområde:* Unik i forhold til de andre konstantene.
- *Bruksområde:* Brukes som verdi i statusfelt for å angi at en transaksjon skal committes.

ABORT

- *Type:* Integer
- *Synlighet:* Public
- *Format:* Integer.
- *Verdiområde:* Unik i forhold til de andre konstantene.
- *Bruksområde:* Brukes som verdi i statusfelt for å angi at en transaksjon skal aborteres.

COMMITTED

- *Type:* Integer
- *Synlighet:* Public
- *Format:* Integer.
- *Verdiområde:* Unik i forhold til de andre konstantene.
- *Bruksområde:* Brukes som verdi i statusfelt for å angi at en transaksjon har committet.

ABORTED

- *Type:* Integer
- *Synlighet:* Public
- *Format:* Integer.
- *Verdiområde:* Unik i forhold til de andre konstantene.
- *Bruksområde:* Brukes som verdi i statusfelt for å angi at en transaksjon har abortert.

CLIENT

- *Type:* Integer
- *Synlighet:* Public
- *Format:* Integer.
- *Verdiområde:* Unik i forhold til de andre konstantene.
- *Bruksområde:* Brukes som verdi i neste stopp feltet for å angi at transaksjonen skal til en klient.

HOST

- *Type:* Integer
- *Synlighet:* Public
- *Format:* Integer.
- *Verdiområde:* Unik i forhold til de andre konstantene.
- *Bruksområde:* Brukes som verdi i neste stopp feltet for å angi at transaksjonen skal til en tjener.

COORDINATOR

- *Type:* Integer
- *Synlighet:* Public
- *Format:* Integer.
- *Verdiområde:* Unik i forhold til de andre konstantene.
- *Bruksområde:* Brukes som verdi i neste stopp feltet for å angi at transaksjonen skal til en koordinator.

E2.2 Tools

Denne klassen inneholder funksjonalitet for å skaffe referanser til navnetjeneren som brukes for å finne fram til de forskjellige JavaSpacene.

Variabler

lookupHost

- *Type:* String
- *Synlighet:* Public
- *Format:* URL
- *Verdiområde:* Alle gyldige URLer som peker til Jini navnetjenere er tillatt.
- *Bruksområde:* Brukes for å angi navnetjener statisk.

lookup

- *Type:* LookupLocator
- *Synlighet:* Private
- *Format:* N/A
- *Verdiområde:* N/A
- *Bruksområde:* Brukes som referanse til en navnetjenerproxy.

registrar

- *Type:* ServiceRegistrar
- *Synlighet:* Public
- *Format:* N/A
- *Verdiområde:* N/A
- *Bruksområde:* Brukes som referanse til en registrarproxy.

Metoder

init()

- *Argumenter:* Ingen
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kjøres når et objekt trenger en referanse til en navnetjener.
- *Prosessering:* Kaller de relevante Jini metodene for å fremskaffe referansen til navnetjeneren som er angitt i lookupHost.

E2.2 SpaceManager

SpaceManager har funksjonalitet for å gi referanser til JavaSpace objekter.

Metoder

JavaSpace getSpace(String spaceName)

- *Argumenter:* spaceName - navnet på det JavaSpacet det skal skaffes referanse til.
- *Returtype:* Ingen
- *Synlighet:* Public
- *Initiering:* Kalles av objekter som trenger referanse til et JavaSpace.
- *Prosessering:* Kaller de relevante Jini metodene for å slå opp referansen til JavaSpacet ved hjelp av ServiceRegistrar objektet fra Tools klassen.

Tillegg D

Kildekode

```
import net.jini.core.entry.Entry;

public class Transaction implements Entry, Constants {
  public String globalId;
  public Integer status;
  public Integer nextStop;

  public Transaction() {
  }
}
```

```
import java.util.Vector;

public class SubTrans extends Transaction {

  public Integer subId;
  public String databaseId;
  public Vector queryList;

  public Vector sharedNameList;

  private SubTransQuery query;

  public SubTrans() {
  }

  public String toString() {
    String result = "**SubTrans**\nglobalId: "+globalId+"\nsubId: "+subId+"\nstatus: "+
      status+"\ndatabaseId: "+databaseId+"\n";
    if (queryList != null) {
      for (int i = 0; i < queryList.size(); i++) {
        SubTransQuery subTransQuery = (SubTransQuery)queryList.elementAt(i);
        result += "query "+i+":\n";
        result += "\t "+subTransQuery.sql+"\n";
        if (subTransQuery.exportEntries != null) {
          for (int j = 0; j < subTransQuery.exportEntries.size(); j++) {
            SubTransQuery.ExportEntry exportEntry =
              (SubTransQuery.ExportEntry)subTransQuery.exportEntries.elementAt(j);
            result += "\t exportName: "+exportEntry.exportName+"  exportValue: "+
              exportEntry.exportValue+"\n";
          }
        }
        if (subTransQuery.importEntries != null) {
          for (int k = 0; k < subTransQuery.importEntries.size(); k++) {
            SubTransQuery.ImportEntry importEntry =
              (SubTransQuery.ImportEntry)subTransQuery.importEntries.elementAt(k);
            result += "\t importName: "+importEntry.importName+"  importValue: "+
              importEntry.importValue+"\n";
          }
        }
      }
    }
    return result;
  }


  public void initQuery() {
    query = new SubTransQuery();
  }

  public void finishQuery() {
    if (queryList == null) {
      queryList = new Vector();
    }
    queryList.addElement(query);
  }

  public void addSql(String sql) {
    query.sql = sql;
  }

  public void addExport(String exportName, String exportValue) {
    query.addExportEntry(exportName, exportValue);
    if(sharedNameList == null) {
      sharedNameList =  new Vector();
    }
    sharedNameList.add(exportName);
  }

  public void addImport(String importName, String importValue) {
    query.addImportEntry(importName, importValue);
```

```
    }
}
```

```
import java.util.Vector;

class SubTransQuery implements java.io.Serializable {
  public String sql;
  public String result;
  public Vector exportEntries;
  public Vector importEntries;

  public void addExportEntry(String name, String value) {
    if (exportEntries == null) {
      exportEntries = new Vector();
    }
    ExportEntry exportEntry = new ExportEntry();
    exportEntry.exportName = name;
    exportEntry.exportValue = value;
    exportEntries.addElement(exportEntry);
  }

  public void addImportEntry(String name, String value) {
    if (importEntries == null) {
      importEntries = new Vector();
    }
    ImportEntry importEntry = new ImportEntry();
    importEntry.importName = name;
    importEntry.importValue = value;
    importEntries.addElement(importEntry);
  }


  class ExportEntry implements java.io.Serializable {
    public String exportName;
    public String exportValue;
  }

  class ImportEntry implements java.io.Serializable {
    public String importName;
    public String importValue;
  }
}
```

```
import java.util.Vector;

public class GlobalTrans extends Transaction {

  public Vector subTransIds;
  private Integer subIdIterator;

  public GlobalTrans() {
  }

  public void addSubTransId(Integer subId) {
    if (subTransIds == null) {
      subTransIds = new Vector();
    }
    boolean found = false;
    for (int i = 0; i < subTransIds.size(); i++) {
      Integer id = (Integer)subTransIds.elementAt(i);
      if (id.equals(subId)) {
        found = true;
      }
    }
    if (!found) {
      subTransIds.addElement(subId);
    }
  }

  public Vector getSubTransIds() {
    return subTransIds;
  }

  public Integer getNextSubId() {
    if (subIdIterator == null) {
      subIdIterator = new Integer(0);
    }
    subIdIterator = new Integer(subIdIterator.intValue()+1);
    return subIdIterator;
  }

  public String toString() {
    return "**GlobalTrans**\nglobalId: "+globalId+"\nstatus: "+status+"\n";
  }

}
```

```
public class SharingTrans extends Transaction {

  public Integer subId;
  public String sharedName;
  public String sharedValue;

  public SharingTrans() {
  }

  public SharingTrans(String globalId, Integer status, Integer subId,
      String sharedName, String sharedValue) {
    this.globalId = globalId;
    this.status = status;
    this.subId = subId;
    this.sharedName = sharedName;
    this.sharedValue = sharedValue;
  }
}
```

```
public class ImportTrans extends SharingTrans {

  public String globalIdExport;
  public Integer subIdExport;

  public ImportTrans() {
  }

  public ImportTrans(String globalId, Integer status, Integer subId,
      String importName, String importValue) {
    super(globalId, status, subId, importName, importValue);
  }

  public String toString() {
    return "\n**ImportTrans**\nglobalId: "+globalId+"\nsubId: "+subId+
      "\nimportName: "+sharedName+"\nimportValue: "+sharedValue;
  }
}
```

```
import java.util.Vector;

public class ExportTrans extends SharingTrans {

  public Vector importTransList;

  public ExportTrans() {
  }

  public ExportTrans(String globalId, Integer status, Integer subId,
      String exportName, String exportValue) {
    super(globalId, status, subId, exportName, exportValue);
  }

  public String toString() {
    return "**ExportTrans**\nglobalId: "+globalId+"\nsubId: "+subId+"\nexportName: "+
      sharedName+"\nexportValue: "+sharedValue;
  }

  public void addImportTrans(ImportTrans it) {

    if(importTransList == null) {
      importTransList = new Vector();
    }
    importTransList.add(it);
    System.out.println("Connected import trans to export trans");
  }
}
```

```
import net.jini.space.JavaSpace;
import java.sql.*;
import java.util.*;

public class MobileHost implements Constants {

  private SpaceManager spaceManager;
  private JavaSpace adminSpace;
  private String dbname;
  private boolean active;
  private TransMod transMod;
  private final Integer OK = new Integer(1);
  private final Integer EXCEPTION = new Integer(2);

  public MobileHost(String db) {
    OmniTools.init();
    spaceManager = new SpaceManager();
    adminSpace = spaceManager.getSpace("autopilot");
    active = true;
    transMod = new TransMod(adminSpace);
    dbname = db;
    //transMod.setDebug(true);

    readSubTrans();
  }

  public void readSubTrans() {
    Thread subTransReader = new Thread() {
      public void run() {
        SubTrans mytrans = null;
        while (active) {
          mytrans = transMod.takeSubTrans(null, HOST, IN_PROGRESS, null, dbname);
          if (mytrans == null) {
            System.out.println("No sub trans read, looping");
            continue;
          }
          SubTransThread subThread = new SubTransThread(mytrans);
          System.out.println("Starting new thread for subtransaction "+mytrans.subId);
          subThread.start();
        }
      }
    };
    subTransReader.start();
  }

  public Connection connectToDatabase() {
    Connection connection = null;
    try {
      Class.forName("com.mysql.jdbc.Driver").newInstance();
      if (dbname.compareTo("Fly") == 0) {
        connection = DriverManager.getConnection("jdbc:mysql://mysql.stud.ntnu.no/"+
          "gunnarga_fly?user=gunnarga_demo&password=smiley");
      }
      else if (dbname.compareTo("Bil") == 0) {
        connection = DriverManager.getConnection("jdbc:mysql://mysql.stud.ntnu.no/"+
          "gunnarga_bil?user=gunnarga_demo&password=smiley");
      }
      connection.setAutoCommit(false);
      System.out.println("Thread "+Thread.currentThread()+" connected to database.");
    } catch (Exception e) {
      System.err.println(e.getMessage());
      // handle the error
    }
    return connection;
  }

  public static void main(String args[]) {
    if (args.length < 0) {
      System.err.println("Please specify the database name this host is responsible for.");
      System.exit(1);
```

```
    }
    else {
      MobileHost mobileHost = new MobileHost(args[0]);
    }
  }

  class SubTransThread extends Thread {
    private SubTrans subTrans;
    //private GlobalTrans globalTrans;
    private Connection connection;
    private Integer queryStatus;

    public SubTransThread(SubTrans subTrans) {
      this.subTrans = subTrans;
      try {
        connection = connectToDatabase();
      } catch (Exception e) {
        System.err.println("Exception caught: "+e.getMessage());
      }
    }

    public void run() {
      queryStatus = OK;
      Vector queryThreads = new Vector();
      boolean threadsRunning = true;

      for (int i = 0; i < subTrans.queryList.size(); i++) {
        SubTransQuery subQuery = (SubTransQuery)subTrans.queryList.elementAt(i);
        QueryThread qthread = new QueryThread(subTrans, subQuery, connection);
        queryThreads.addElement(qthread);
        qthread.start();
      }
      while (threadsRunning) {
        threadsRunning = false;
        for (int i = 0; i < queryThreads.size(); i++) {
          Thread t = (Thread)queryThreads.elementAt(i);
          if (t.isAlive()) {
            threadsRunning = true;
          }
        }
        try {
          Thread.sleep(50);
        } catch (InterruptedException e) {
          System.err.println(e.getMessage());
        }
      }

      if (queryStatus.equals(EXCEPTION)) {
        try {
          System.out.println("Rolling back subtransaction "+subTrans.subId);
          connection.rollback();
          subTrans.nextStop = COORDINATOR;
          subTrans.status = ABORTED;
          transMod.writeSubTrans(subTrans);
          System.out.println("Aborting subtransaction "+subTrans.subId);
          System.out.flush();
        } catch (SQLException e) {
          System.out.println("SQLException caught in rollback: "+e.getMessage());
        }
      }

      else if (queryStatus.equals(OK)) {
        subTrans.nextStop = COORDINATOR;
        subTrans.status = READY_TO_COMMIT;
        SubTrans resultTrans = null;
        transMod.writeSubTrans(subTrans);
        System.out.println("Subtransaction "+subTrans.subId+" is ready to commit.");
        System.out.flush();

        while (resultTrans == null) {
          resultTrans = transMod.takeSubTrans(subTrans.globalId, HOST, null,
            subTrans.subId, subTrans.databaseId);
```

```
        System.out.println("Resulttrans was null, looping");
      }
      if (resultTrans.status.equals(COMMIT)) {
        try {
          connection.commit();
          resultTrans.nextStop = CLIENT;
          resultTrans.status = COMMITTED;
          transMod.writeSubTrans(resultTrans);
          System.out.println("Committing subtransaction "+subTrans.subId);
        } catch (Exception e) {
          System.err.println("Exception caught in subtrans commit processing: "+
            e.getMessage());
        }
      }
      else if (resultTrans.status.equals(ABORT)) {
        try {
          connection.rollback();
          resultTrans.nextStop = CLIENT;
          resultTrans.status = ABORTED;
          transMod.writeSubTrans(resultTrans);
          System.out.println("Aborting subtransaction "+subTrans.subId);
        } catch (Exception e) {
          System.err.println("Exception caught in subtrans commit processing: "+
            e.getMessage());
        }
      }
    }
  }

  class QueryThread extends Thread {
    private SubTrans subTrans;
    private SubTransQuery subQuery;
    private Connection connection;

    public QueryThread(SubTrans subTrans, SubTransQuery subQuery,
        Connection connection) {
      this.subTrans = subTrans;
      this.subQuery = subQuery;
      this.connection = connection;
    }

    public void run() {
      String sql = subQuery.sql;

      if (subQuery.importEntries != null) {
        for (int i = 0; i < subQuery.importEntries.size(); i++) {
          SubTransQuery.ImportEntry importEntry =
            (SubTransQuery.ImportEntry)subQuery.importEntries.elementAt(i);
          ExportTrans exportTrans = null;
          while (exportTrans == null) {
            System.out.println("Looking for export trans "+subTrans.globalId+
              " "+importEntry.importName);
            exportTrans = transMod.takeExportTrans(subTrans.globalId, null, null,
              importEntry.importName);
          }
          String importName = exportTrans.sharedName;
          String importValue = exportTrans.sharedValue;
          ImportTrans importTrans = new ImportTrans(subTrans.globalId, IN_PROGRESS,
            subTrans.subId, importName, importValue);
          importTrans.globalIdExport = exportTrans.globalId;
          importTrans.subIdExport = exportTrans.subId;
          exportTrans.addImportTrans(importTrans);
          transMod.addImportTrans(importTrans);
          transMod.addExportTrans(exportTrans);
          //System.out.println(importTrans);
          if (importValue == null) {
            System.out.println("Import value for variable "+importName+" was null.");
            sql = sql.replaceAll("'"+importName+"'", "NULL");
          }
          else {
            sql = sql.replaceAll("'"+importName+"'", "'"+importValue+"'");
          }
```

```
          }
        }
        System.out.println("Executing query:\n"+sql);
        try {
          Statement stmt = connection.createStatement();
          ResultSet rs = null;
          if (sql.startsWith("SELECT")) {
            rs = stmt.executeQuery(sql);
            System.out.println("Parsing started");
            subQuery.result = parseResultSet(rs);
            System.out.println("Parsing finished");
          }
          else {
            int rows = stmt.executeUpdate(sql);
            subQuery.result = "Affected rows: "+rows;
          }
          //System.out.println("ResultSet: \n"+subQuery.result);
          if (subQuery.exportEntries != null) {
            for (int i = 0; i < subQuery.exportEntries.size(); i++) {
              SubTransQuery.ExportEntry exportEntry =
                (SubTransQuery.ExportEntry)subQuery.exportEntries.elementAt(i);
              String exportName = exportEntry.exportName;
              String exportValue = exportEntry.exportValue;
              if (exportValue == null) {
                if (rs != null) {
                  if (rs.first()) {
                    exportValue = rs.getString(exportName);
                  }
                }
              }
              ExportTrans exportTrans = new ExportTrans(subTrans.globalId, IN_PROGRESS,
                subTrans.subId, exportName, exportValue);
              transMod.addExportTrans(exportTrans);
              //System.out.println("Adding export trans:\n"+exportTrans);
            }
          }
          stmt.close();
        } catch (SQLException e) {
          System.err.println("SQLException caught in QueryThread:\n"+e.getMessage());
          queryStatus = EXCEPTION;
        }
      }

      public String parseResultSet(ResultSet rs) {
        StringBuffer result = new StringBuffer();
        try {
          ResultSetMetaData meta = rs.getMetaData();
          while (rs.next()) {
            result.append("<row>\n");
            for (int i = 1; i <= meta.getColumnCount(); i++) {
              String columnName = meta.getColumnName(i);
              String columnValue = rs.getString(columnName);
              result.append("\t<"+columnName+">"+columnValue+"</"+columnName+">\n");
            }
            result.append("</row>\n");
          }
        } catch (SQLException e) {
          System.err.println("SQLException caught in parseResultSet:\n"+e.getMessage());
        }
        return result.toString();
      }
    }
  }
}
```

```
import net.jini.space.JavaSpace;
import net.jini.core.lease.Lease;
import java.util.*;

public class TransMod implements Constants {
  private JavaSpace jsp;
  private boolean debug;

  public TransMod(JavaSpace jsp) {
    this.jsp = jsp;
  }

  public void setDebug(boolean state) {
    debug = state;
  }

  public Transaction takeTransaction(String globalId, Integer nextStop, Integer status) {
    Transaction t = null;
    try {
      Transaction template = new Transaction();
      template.globalId = globalId;
      template.nextStop = nextStop;
      template.status = status;
      t = (Transaction)jsp.take(template, null, 1000);
    } catch (Exception e) {
      System.err.println("Exception caught in takeTransaction: "+e.getMessage());
    }
    return t;
  }

  public void addGlobalTrans(GlobalTrans gt) {
    try {
      if (debug) System.out.println("writing globalId "+gt.globalId);
      jsp.write(gt, null, Lease.FOREVER);
    } catch (Exception e) {
      System.err.println("Exception caught in addGlobalTrans: "+e.getMessage());
    }
  }

  public GlobalTrans takeGlobalTrans(String globalId, Integer nextStop, Integer status) {
    GlobalTrans gt = null;
    try {
      GlobalTrans template = new GlobalTrans();
      template.globalId = globalId;
      template.nextStop = nextStop;
      template.status = status;
      //if (debug) System.out.println("taking globalId "+gt.globalId);
      gt = (GlobalTrans)jsp.take(template, null, WAIT_TIME);
    } catch (Exception e) {
      System.err.println("Exception caught in takeGlobalTrans: "+e.getMessage());
    }
    return gt;
  }

  public GlobalTrans readGlobalTrans(String globalId, Integer nextStop, Integer status) {
    GlobalTrans gt = null;

    try {
      GlobalTrans template = new GlobalTrans();
      template.globalId = globalId;
      template.nextStop = nextStop;
      template.status = status;
      if (debug) System.out.println("taking globalId "+template.globalId);
      gt = (GlobalTrans)jsp.read(template, null, WAIT_TIME);
    } catch (Exception e) {
      System.err.println("Exception caught in readGlobalTrans: "+e.getMessage());
    }
    return gt;
  }
```

```
public void addSubTrans(SubTrans st) {
  try {
    GlobalTrans gt = takeGlobalTrans(st.globalId, null, null);
    if (debug) System.out.println("adding subId "+st.subId+" to globalTrans "+gt.globalId);
    gt.addSubTransId(st.subId);
    addGlobalTrans(gt);
    if (debug) System.out.println("writing subId "+st.subId);
    jsp.write(st, null, Lease.FOREVER);
  } catch (Exception e) {
    System.err.println("Exception caught in addSubTrans: "+e.getMessage());
  }
}

public void writeSubTrans(SubTrans st) {
  try {
    jsp.write(st, null, Lease.FOREVER);
  } catch (Exception e) {
    System.err.println("Exception caught in writeSubTrans: "+e.getMessage());
  }
}


public SubTrans takeSubTrans(String globalId, Integer nextStop, Integer status,
    Integer subId, String databaseId) {
  SubTrans st = null;

  try {
    SubTrans template = new SubTrans();
    template.globalId = globalId;
    template.nextStop = nextStop;
    template.status = status;
    template.databaseId = databaseId;
    template.subId = subId;
    //if (debug) System.out.println("taking subId "+template.subId);
    st = (SubTrans)jsp.take(template, null, WAIT_TIME);
  } catch (Exception e) {
    System.err.println("Exception caught in takeSubTrans: "+e.getMessage());
  }
  return st;
}

public SubTrans readSubTrans(String globalId, Integer nextStop, Integer status,
    Integer subId, String databaseId) {
  SubTrans st = null;

  try {
    SubTrans template = new SubTrans();
    template.globalId = globalId;
    template.nextStop = nextStop;
    template.status = status;
    template.databaseId = databaseId;
    template.subId = subId;
    //if (debug) System.out.println("reading subId "+template.subId);
    st = (SubTrans)jsp.read(template, null, WAIT_TIME);
  } catch (Exception e) {
    System.err.println("Exception caught in takeSubTrans: "+e.getMessage());
  }
  return st;
}


public Vector readSubTransList(GlobalTrans gt) {
  Vector subTransList = new Vector();

  try {
    Vector ids = gt.getSubTransIds();


    if (debug) System.out.println("globalTrans "+gt.globalId+" have "+ids.size()+
      " subids attached");
    for (int i = 0; i < ids.size(); i++) {
      Integer subId = (Integer)ids.elementAt(i);
```

```
      SubTrans st = readSubTrans(gt.globalId, null, null, subId, null);
      if (st != null) {
        subTransList.addElement(st);
      }
    }
  } catch (Exception e) {
    System.err.println("Exception caught in readSubTransList: "+e.getMessage());
  }
  return subTransList;
}

public void addExportTrans(ExportTrans exportTrans) {
  try {
    jsp.write(exportTrans, null, Lease.FOREVER);
  } catch (Exception e) {
    System.err.println("Exception caught in addExportTrans: "+e.getMessage());
  }
}

public void addImportTrans(ImportTrans importTrans) {
  try {
    jsp.write(importTrans, null, Lease.FOREVER);
  } catch (Exception e) {
    System.err.println("Exception caught in addImportTrans: "+e.getMessage());
  }
}


public ExportTrans takeExportTrans(String globalId, Integer status, Integer subId,
  String exportName) {

  ExportTrans et = null;
  try {
    //if(debug) System.out.println("Taking exportTransaction with globalId: "+ );
    ExportTrans templ = new ExportTrans();
    templ.globalId = globalId;
    templ.status = status;
    templ.subId = subId;
    templ.sharedName = exportName;

    et= (ExportTrans)jsp.take(templ, null, WAIT_TIME);
  } catch(Exception e) {
    System.err.println("Exception caught in takeExportTrans: "+e.getMessage());
  }
  return et;
}

public ImportTrans takeImportTrans(ExportTrans et) {
  ImportTrans it = null;
  try {
    //if(debug) System.out.println("Taking importTransaction with globalId: "+ );
    ImportTrans templ = new ImportTrans();
    templ.globalIdExport = et.globalId;
    templ.subIdExport = et.subId;
    templ.sharedName = et.sharedName;
    it = (ImportTrans)jsp.take(templ, null, WAIT_TIME);
  } catch(Exception e) {
    System.err.println("Exception caught in takeImportTrans: "+e.getMessage());
  }
  return it;
}

public static void main(String[] args) {
  SubTrans s1, s2;
  GlobalTrans g1;
  OmniTools.init();
  SpaceManager spaceManager = new SpaceManager();
  JavaSpace space = spaceManager.getSpace("autopilot");
  System.out.println("** UNIT TransMod EXECUTING SELF-TEST **");
  TransMod transMod = new TransMod(space);
  transMod.setDebug(true);
```

```
    g1 = new GlobalTrans();
    g1.globalId = "test1id";
    g1.status = IN_PROGRESS;
    s1 = new SubTrans();
    s1.globalId = "test1id";
    s1.status = READY_TO_COMMIT;
    s1.subId = new Integer(23);
    s2 = new SubTrans();
    s2.globalId = "test1id";
    s2.status = READY_TO_COMMIT;
    s2.subId = new Integer(19);

    transMod.addGlobalTrans(g1);
    transMod.addSubTrans(s1);
    transMod.addSubTrans(s2);


    Vector v = transMod.readSubTransList(g1);
    System.out.println("list size: "+v.size());


    SubTrans r1 = transMod.takeSubTrans(null, null, READY_TO_COMMIT, null, null);
    System.out.println(r1);
    System.out.println("** UNIT TransMod COMPLETED SELF-TEST **");
  }
}
```

```
import net.jini.space.JavaSpace;
import java.io.*;
import java.util.*;

public class MobileClient implements Constants {

  private SpaceManager spaceManager;
  private JavaSpace adminSpace;
  private XMLtoEntryParser parser;
  private TransMod transMod;
  private final int ABORT_PERCENTAGE = 0;
  private static long startTime;
  private static long finishTime;
  private boolean serialExecution;
  private int iterations;
  private Vector threadList;

  public MobileClient() {
    OmniTools.init();
    spaceManager = new SpaceManager();
    adminSpace = spaceManager.getSpace("autopilot");
    parser = new XMLtoEntryParser();
    transMod = new TransMod(adminSpace);
    threadList = new Vector();
  }

  public void setSerialExecution(boolean value) {
    serialExecution = value;
  }

  public void traverseQueryDirectory(String queryDir) {
    boolean threadsRunning = true;

    startTime = System.currentTimeMillis();
    if (queryDir == null) {
      queryDir = "queries";
    }
    File dir = new File(queryDir);

    FilenameFilter filter = new FilenameFilter() {
      public boolean accept(File dir, String name) {
        return name.endsWith(".xml");
      }
    };

    File[] files = dir.listFiles(filter);
    if (files == null) {
      OmniTools.debug("Query directory is empty or does not exist.");
    }
    else {
      for (int i = 0;i < files.length; i++) {
        OmniTools.debug("Parsing file "+files[i].getName());
        parseXMLFile(files[i]);
        if (serialExecution) {
          waitForThreadsToFinish();
        }
      }
    }

    waitForThreadsToFinish();
    finishTime = System.currentTimeMillis();
  }

  public void parseXMLFile(File file) {
    for (int j = 0; j < 1; j++) {
      Vector entries = parser.createEntries(file);
      String globalId = null;
      for (int i = 0; i < entries.size(); i++) {
        try {
          Transaction trans = (Transaction)entries.elementAt(i);
```

```
          if (trans.getClass() == Class.forName("GlobalTrans")) {
            globalId = trans.globalId;
            transMod.addGlobalTrans((GlobalTrans)trans);
            GlobalTransThread gthread = new GlobalTransThread((GlobalTrans)trans);
            threadList.addElement(gthread);
            gthread.start();

            System.out.println("Started globalTrans "+globalId);
          }
        } catch (Exception e) {
          System.err.println("Exception caught: "+e.getMessage());
        }
      }
      for (int i = 0; i < entries.size(); i++) {
        try {
          Transaction trans = (Transaction)entries.elementAt(i);
          if (trans.getClass() == Class.forName("SubTrans")) {
            transMod.addSubTrans((SubTrans)trans);
            SubTransThread sthread = new SubTransThread((SubTrans)trans);
            threadList.addElement(sthread);
            sthread.start();



            System.out.println("Started subTrans globalId: "+((SubTrans)trans).globalId+
              " subId: "+((SubTrans)trans).subId);
          }
        }
        catch (Exception e) {
          System.err.println("Exception caught: "+e.getMessage());
        }
      }
      int abortChance = (int)(Math.random()*100);
      if (abortChance < ABORT_PERCENTAGE) {
        abortSubTrans();
      }
      GlobalTrans globalTrans = transMod.takeGlobalTrans(globalId, null, null);
      globalTrans.nextStop = COORDINATOR;
      globalTrans.status = READY_TO_COMMIT;
      transMod.addGlobalTrans(globalTrans);
    }

  }

  public void abortSubTrans() {
    SubTrans st = transMod.takeSubTrans(null, null, READY_TO_COMMIT,
      new Integer(1), "Fly");
    st.status = ABORT;
    st.nextStop = COORDINATOR;
    System.out.println("Aborting subtransaction "+st.subId);
    transMod.writeSubTrans(st);
  }

  public void waitForThreadsToFinish() {
    boolean threadsRunning = true;

    while (threadsRunning) {
      threadsRunning = false;
      for (int i = 0; i < threadList.size(); i++) {
        Thread t = (Thread)threadList.elementAt(i);
        if (t.isAlive()) {
          threadsRunning = true;
        }
      }
      try {
        Thread.sleep(50);
      } catch (InterruptedException e) {
        System.err.println(e.getMessage());
      }
    }
  }
```

```
public static void main(String[] args) {
  MobileClient mobileClient = new MobileClient();

  if (args.length > 1) {
    if (args[1].compareTo("serial") == 0) {
      mobileClient.setSerialExecution(true);
    }
  }
  if (args.length > 0) {
    try {
      mobileClient.traverseQueryDirectory(args[0]);
    } catch (Exception e) {
      OmniTools.debug(e.getMessage());
    }
  }
  else {
    mobileClient.traverseQueryDirectory("queries");
  }
  System.out.println("Client done in "+(finishTime-startTime)+" milliseconds.");
  System.out.flush();
}


class GlobalTransThread extends Thread {
  private GlobalTrans globalTrans;

  public GlobalTransThread(GlobalTrans gt) {
    globalTrans = gt;
  }

  public void run() {
    GlobalTrans resultTrans = null;
    while (resultTrans == null) {
      resultTrans = transMod.takeGlobalTrans(globalTrans.globalId, CLIENT, null);
    }
    if (resultTrans.status.equals(COMMITTED)) {
      System.out.println("Global trans "+resultTrans.globalId+" committed.");
      System.out.flush();
    }
    else if (resultTrans.status.equals(ABORTED)) {
      System.out.println("Global trans "+resultTrans.globalId+" aborted.");
      System.out.flush();
    }
  }
}


class SubTransThread extends Thread {
  private SubTrans subTrans;

  public SubTransThread(SubTrans st) {
    subTrans = st;
  }

  public void run() {
    SubTrans resultTrans = null;
    while (resultTrans == null) {
      resultTrans = transMod.takeSubTrans(subTrans.globalId, CLIENT, null,
        subTrans.subId, subTrans.databaseId);
    }
    if (resultTrans.status.equals(COMMITTED)) {
      System.out.println("Subtrans globalId: "+resultTrans.globalId+" subId: "
        +resultTrans.subId+" committed.");
      System.out.flush();
      for (int i = 0; i < resultTrans.queryList.size(); i++) {
        /*
         SubTransQuery subQuery = (SubTransQuery)resultTrans.queryList.elementAt(i);
         if (subQuery.result.compareTo("") != 0) {
         System.out.println("Operation "+i+" completed successfully with result:");
         System.out.println(subQuery.result);
         }
         else {
```

```
            System.out.println("Operation "+i+" completed successfully.");
            }
            */
          }
        }
        else if (resultTrans.status.equals(ABORTED)) {
          System.out.println("Subtrans globalId: "+resultTrans.globalId+" subId: "+
            resultTrans.subId+" aborted.");
          System.out.flush();
          /*
           resultTrans.nextStop = HOST;
           resultTrans.status = IN_PROGRESS;
           transMod.writeSubTrans(resultTrans);
           SubTransThread retryThread = new SubTransThread(resultTrans);
           retryThread.start();
           */
        }
      }
    }
}
```

```
import java.io.*;
import java.util.*;
import java.net.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import org.w3c.dom.*;
import org.w3c.dom.traversal.*;
import org.apache.xpath.XPathAPI;
import org.xml.sax.SAXException;

public class XMLtoEntryParser implements Constants {
  private Transformer serializer;
  private DocumentBuilder builder;

  public XMLtoEntryParser() {
    try {
      // set up a document builder
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      builder = factory.newDocumentBuilder();

      // set up an identity transformer to use as serializer
      serializer = TransformerFactory.newInstance().newTransformer();
      serializer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");
    } catch (Exception e) {
      OmniTools.debug(e.getMessage());
    }
  }

  public Vector createEntries(File xmlFile) {
    Document doc = null;
    NodeIterator nl = null;
    Node n = null;
    String xpath = new String();
    Vector result = new Vector();
    String dns = null;

    try {
      dns = InetAddress.getLocalHost().getHostName();
    } catch ( UnknownHostException e ) {
      System.err.println("Cannot detect localhost: "+e.getMessage());
    }

    try {
      // create a DOM document
      doc = builder.parse(xmlFile);

      // parse GlobalTrans objects
      xpath = "//globalTrans";
      nl = XPathAPI.selectNodeIterator(doc, xpath);

      // first read in all the globalTrans elements
      while ((n = nl.nextNode()) != null) {
        GlobalTrans globalTrans = new GlobalTrans();
        Date currentDate = new Date();
        globalTrans.globalId = dns+":"+currentDate.getTime();
        //globalTrans.nextStop = HOST;
        globalTrans.status = IN_PROGRESS;
        NodeIterator iterator = ((DocumentTraversal)doc).createNodeIterator(
          n, NodeFilter.SHOW_ELEMENT, null, false);

        // read in all the nodes below each globalTrans element
        Node subNode;
        while ((subNode = iterator.nextNode()) != null) {
          if (subNode.getNodeName().compareTo("subTrans") == 0) {
            SubTrans subTrans = new SubTrans();
            subTrans.globalId = globalTrans.globalId;
            subTrans.nextStop = HOST;
            subTrans.status = IN_PROGRESS;
            subTrans.subId = globalTrans.getNextSubId();
            NodeIterator subIterator = ((DocumentTraversal)doc).createNodeIterator(
```

```
            subNode, NodeFilter.SHOW_ELEMENT, null, false);
          Node subContentsNode;
          while ((subContentsNode = subIterator.nextNode()) != null) {
            if (subContentsNode.getNodeName().compareTo("databaseId") == 0) {
              subTrans.databaseId = subContentsNode.getFirstChild().getNodeValue();
            }
            if (subContentsNode.getNodeName().compareTo("query") == 0) {
              subTrans.initQuery();
              NodeIterator queryIterator =
                ((DocumentTraversal)doc).createNodeIterator(
                subContentsNode, NodeFilter.SHOW_ELEMENT, null, false);
              Node queryNode;
              while ((queryNode = queryIterator.nextNode()) != null) {
                if (queryNode.getNodeName().compareTo("sql") == 0) {
                  String sql = queryNode.getFirstChild().getNodeValue();
                  subTrans.addSql(sql);
                }
                if (queryNode.getNodeName().compareTo("export") == 0) {
                  String exportName = null;
                  String exportValue = null;
                  NodeIterator exportIterator =
                    ((DocumentTraversal)doc).createNodeIterator(
                    queryNode, NodeFilter.SHOW_ELEMENT, null, false);
                  Node exportNode;
                  while ((exportNode = exportIterator.nextNode()) != null) {
                    if (exportNode.getNodeName().compareTo("name") == 0) {
                      exportName = exportNode.getFirstChild().getNodeValue();
                    }
                    if (exportNode.getNodeName().compareTo("value") == 0) {
                      exportValue = exportNode.getFirstChild().getNodeValue();
                    }
                  }
                  subTrans.addExport(exportName, exportValue);
                }
                if (queryNode.getNodeName().compareTo("import") == 0) {
                  String importName = null;
                  String importValue = null;
                  NodeIterator importIterator =
                    ((DocumentTraversal)doc).createNodeIterator(
                    queryNode, NodeFilter.SHOW_ELEMENT, null, false);
                  Node importNode;
                  while ((importNode = importIterator.nextNode()) != null) {
                    if (importNode.getNodeName().compareTo("name") == 0) {
                      importName = importNode.getFirstChild().getNodeValue();
                    }
                    if (importNode.getNodeName().compareTo("value") == 0) {
                      importValue = importNode.getFirstChild().getNodeValue();
                    }
                  }
                  subTrans.addImport(importName, importValue);
                }
              }
              subTrans.finishQuery();
            }
          }

          result.addElement(subTrans);

        }
      }

      result.addElement(globalTrans);
    }

  } catch (Exception e) {
    OmniTools.debug(e.getMessage());
  }

  return result;
 }
}
```

```
import java.util.*;
import net.jini.space.JavaSpace;

public class Coordinator implements Constants {

  private JavaSpace sharingSpace;
  private SpaceManager sm;
  private TransMod tm;
  private int globalCommits=0;

  public Coordinator() {

    OmniTools.init();
    sm = new SpaceManager();
    sharingSpace = sm.getSpace("autopilot");

    tm = new TransMod(sharingSpace);

    CoordThread.setListener(this, tm, COORDINATOR, true);
  }


  public void makeDecision(GlobalTrans globalTrans) {
    System.out.println("————————————————————————————————————————————————————————————");
    System.out.println("MakeDecision – globalTransId: "+globalTrans.globalId);
    boolean inProgress = false;
    boolean subTransAbort = false;
    boolean subTransAborted = false;
    Vector subTransList = new Vector();

    subTransList = tm.readSubTransList(globalTrans);

    if (subTransList != null) {

      for (int i=0;i<subTransList.size();i++) {
        Integer status = ((SubTrans)subTransList.get(i)).status;

        if (status.intValue() == ABORTED.intValue()) {
          subTransAborted = true;
        }

        else if (status.intValue() == ABORT.intValue()) {
          subTransAbort = true;
          abortSubTransaction((SubTrans)subTransList.get(i));
        }

        else if (status.intValue() == IN_PROGRESS.intValue()) {
          inProgress = true;
        }
      }


      if (subTransAborted) {
        // abort all subtransactions
        for (int i = 0; i < subTransList.size(); i++) {
          SubTrans subTrans = (SubTrans)subTransList.get(i);
          Integer status = subTrans.status;
          if (status.intValue() != ABORTED.intValue()) {
            SubTrans removedTrans = tm.takeSubTrans(subTrans.globalId, null, null,
              subTrans.subId, null);
            removedTrans.nextStop = HOST;
            removedTrans.status = ABORT;
            tm.writeSubTrans(removedTrans);
          }
          else {
            SubTrans removedTrans = tm.takeSubTrans(subTrans.globalId, null, null,
              subTrans.subId, null);
            removedTrans.nextStop = CLIENT;
            tm.writeSubTrans(removedTrans);
          }
```

```
        }
        abortGlobal(globalTrans);
      }
      if (!subTransAbort && !subTransAborted && !inProgress) {
        // all subtransactions are ready to commit, so we commit them
        commitGlobal(globalTrans);
        commitSub(subTransList);
      }
    }
    else {
      System.out.println("GlobalTransaction: "+globalTrans.globalId+" has got no "+
        "subTrans!");
    }
  }


  public void commitGlobal(GlobalTrans globalTr) {
    globalCommits++;
    System.out.println("Commits globalTransaction: "+globalTr.globalId);
    System.out.println("Number of globalCommits: "+globalCommits);
    //GlobalTrans templ = new GlobalTrans;
    //templ = globalTr;

    globalTr.status = COMMITTED;
    globalTr.nextStop = CLIENT;
    tm.addGlobalTrans(globalTr);

  }

  public void abortGlobal(GlobalTrans globalTr) {

    System.out.println("Aborts globalTransaction: "+globalTr.globalId);
    globalTr.status = ABORTED;
    globalTr.nextStop = CLIENT;
    tm.addGlobalTrans(globalTr);
  }


  public void commitSub(Vector transactions) {

    for(int i = 0; i<transactions.size();i++) {
      SubTrans subTr = (SubTrans)transactions.get(i);
      System.out.println("Commits SubTransaction: "+subTr.globalId+" "+subTr.subId);
      SubTrans subTrans = tm.takeSubTrans(subTr.globalId, null, null, subTr.subId, null);
      subTr.nextStop = HOST;
      subTr.status = COMMIT;
      tm.writeSubTrans(subTr);
    }
  }

  public void abortSubTransaction(SubTrans st) {

    System.out.println("Aborts SubTransaction: "+st.globalId+" "+st.subId);
    SubTrans subTrans = tm.takeSubTrans(st.globalId, null, null,  st.subId, null);
    subTrans.nextStop = HOST;
    subTrans.status = ABORT;
    abortExportTrans(subTrans);
    tm.writeSubTrans(subTrans);
  }


  public void abortExportTrans(SubTrans subtrans) {

    Vector exportTransList = subtrans.sharedNameList;
    System.out.println("Aborts exportTransactions of SubTransaction: "+subtrans.globalId+
      " "+subtrans.subId);

    if (exportTransList == null) {
      System.out.println(subtrans.globalId+" "+subtrans.subId+" has got no "+
        "ExportTransactions!");
    }
```

```
    else {
      for(int i = 0; i<exportTransList.size();i++) {
        System.out.println("Examining export name: "+(String)exportTransList.get(i));
        ExportTrans et = tm.takeExportTrans(subtrans.globalId, null, subtrans.subId,
          (String)exportTransList.get(i));
        abortImportTrans(et);
        //System.out.println(et.toString());
      }
    }
  }

  public void abortImportTrans(ExportTrans et) {

    Vector importTransList =  et.importTransList;
    if(importTransList == null) {
      System.out.println("No import transactions connected to export transaction "+
        "sharing: "+et.sharedName);
    }

    else {

      for(int i=0;i<importTransList.size();i++) {
        //oppdater status, importTrans
        ImportTrans templ = (ImportTrans)importTransList.get(i);
        ImportTrans it = tm.takeImportTrans(et);
        System.out.println("Aborts importTransaction: "+it.toString());
        SubTrans st = tm.readSubTrans(it.globalId, null, null, it.subId, null);
        if (st == null) {
          System.out.println("Did not find subtransaction in abortImportTrans with "+
            "globalId: "+it.globalId+" and subId: "+it.subId);
        }
        else {
          abortSubTransaction(st);
        }
      }
    }
  }


  public static void main(String[] args) {
    Coordinator coord = new Coordinator();
  }
}
```

```
import java.util.*;

public class CoordThread implements Runnable, Constants {
  private Coordinator coord;
  private boolean listen = false;
  private TransMod transM;
  private Integer nextStop;
  private boolean globalTrans;

  public CoordThread(Coordinator coord, TransMod transM, Integer nextStop,
      boolean globalTrans) {
    listen = true;
    this.coord = coord;
    this.transM = transM;
    this.nextStop = nextStop;
    this.globalTrans = globalTrans;
  }

  public void run() {
    if(globalTrans) {
      while (listen) {
        SubTrans st = null;
        while (st == null) {
          System.out.println("Looking for subtransactions...");
          st = transM.readSubTrans(null, nextStop, null, null, null);
        }
        System.out.println("Reading subtrans globalId: "+st.globalId+" subId: "+
          st.subId);

        GlobalTrans gt = transM.readGlobalTrans(st.globalId, COORDINATOR,
          READY_TO_COMMIT);
        System.out.println("Reading globaltrans globalId: "+gt.globalId);
        coord.makeDecision(gt);
      }
    }
  }

  public void stopListener() {
    listen = false;
  }

  public static void setListener(Coordinator coord, TransMod tm, Integer nextStop,
      boolean globalTrans) {
    CoordThread ct = new CoordThread(coord, tm, nextStop, globalTrans);
    new Thread(ct).start();
  }
}
```

```
import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;
import java.rmi.RMISecurityManager;

public class OmniTools {
  public static boolean debug = true;
  private static String lookupHost = "jini://stud338.idi.ntnu.no";
  private static LookupLocator lookup = null;
  private static ServiceRegistrar registrar = null;

  public static void init() {
    try {
      lookup = new LookupLocator(lookupHost);
    } catch (Exception e) {
      e.printStackTrace();
    }
    if (lookup != null && debug == true) {
      System.out.println("Lookup locator found at "+lookup.getHost());
    }
    System.setSecurityManager(new RMISecurityManager());

    try {
      registrar = lookup.getRegistrar();
    } catch (Exception e) {
      e.printStackTrace();
    }
    if (registrar != null && debug == true) {
      System.out.println("Registrar found.");
    }
  }

  public static void debug(String debugString) {
    if (debug == true) {
      System.err.println(debugString);
    }
  }


  public static LookupLocator getLookupLocator() {
    return lookup;
  }


  public static ServiceRegistrar getServiceRegistrar() {
    return registrar;
  }
}
```

```
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.space.JavaSpace;
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.Name;

public class SpaceManager{

  private JavaSpace adminSpace;
  private JavaSpace currentSpace;
  private String spaceString;

  public SpaceManager() {
    //adminSpace = getSpace("AdminSpace");
  }

  public JavaSpace getSpace(String spaceName) {
    Class serviceTypes[] = { JavaSpace.class };
    Entry[] spaceAttributes = new Entry[] { new Name(spaceName) };
    JavaSpace jsp = null;

    ServiceRegistrar registrar = OmniTools.getServiceRegistrar();
    ServiceTemplate template = new ServiceTemplate(null, serviceTypes,
        spaceAttributes);
    try {
      jsp = (JavaSpace) registrar.lookup(template);
    } catch (Exception e) {
      e.printStackTrace();
    }

    if (jsp == null) {
      OmniTools.debug("JavaSpace " + spaceName + " not found.");
    } else {
      OmniTools.debug("JavaSpace " + spaceName + " found.");
    }
    return jsp;
  }

  public JavaSpace getCurrentSpace() {
    return currentSpace;
  }

  public String getCurrentSpaceString() {
    return spaceString;
  }

  public void setCurrentSpace(String spaceString) {
    this.spaceString = spaceString;
    currentSpace = getSpace(spaceString);
  }
}
```

```
public interface Constants {

  public static final Integer IN_PROGRESS = new Integer(1);
  public static final Integer SUBTRANS_SUBMITTED = new Integer(2);
  public static final Integer SUBTRANS_EVENT = new Integer(3);
  public static final Integer READY_TO_COMMIT = new Integer(4);
  public static final Integer COMMIT = new Integer(5);
  public static final Integer ABORT = new Integer(6);
  public static final Integer COMMITTED = new Integer(7);
  public static final Integer ABORTED = new Integer(8);
  public static final Integer CLIENT = new Integer(9);
  public static final Integer HOST = new Integer(10);
  public static final Integer COORDINATOR = new Integer(11);

  public static final int WAIT_TIME = 30000;
}
```